Knowledge Management of Formal Mathematics and Interactive Theorem Proving

Claudio Sacerdoti Coen

Technical Report UBLCS-2004-05

March 2004

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

Recent Titles from the UBLCS Technical Report Series

2002-10 *Towards Self-Organizing, Self-Repairing and Resilient Distributed Systems*, Montresor, A., Babaoglu, O., Meling, H., September 2002 (Revised November 2002).

2002-11 *Messor: Load-Balancing through a Swarm of Autonomous Agents*, Montresor, A., Meling, H., Babaoglu, O., September 2002.

2002-12 *Johanna: Open Collaborative Technologies for Teleorganizations*, Gaspari, M., Picci, L., Petrucci, A., Faglioni, G., December 2002.

2003-1 Security and Performance Analyses in Distributed Systems (Ph.D Thesis), Aldini, A., February 2003.

2003-2 Models and Types for Wide Area Computing. The calculus of Boxed Ambients (Ph.D. Thesis), Crafa, S., February 2003.

2003-3 MathML Formatting (Ph.D. Thesis), Padovani, L., February 2003.

2003-4 Performance Evaluation of Mobile Agents Paradigm for Wireless Networks (Ph.D. Thesis), Al Mobaideen, W., March 2003.

2003-5 Synchronized Hypermedia Documents: a Model and its Applications (Ph.D. Thesis), Gaggi, O., March 2003.

2003-6 Searching and Retrieving in Content-Based Repositories of Formal Mathematical Knowledge (Ph.D. Thesis), Guidi, F., March 2003.

2003-7 Intersection Types, Lambda Abstraction Algebras and Lambda Theories (Ph.D. Thesis), Lusin, S., March 2003.

2003-8 Towards an Ontology-Guided Search Engine, Gaspari, M., Guidi, D., June 2003.

2003-9 An Object Based Algebra for Specifying A Fault Tolerant Software Architecture, Dragoni, N., Gaspari, M., June 2003.

2003-10 A Scalable Architecture for Responsive Auction Services Over the Internet, Amoroso, A., Fanzieri F., June 2003.

2003-11 WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications, Lucchi, R., Zavattaro, G., September 2003.

2003-12 Integrating Agent Communication Languages in Open Services Architectures, Dragoni, N., Gaspari, M., October 2003.

2003-13 Perfect load balancing on anonymous trees, Margara, L., Pistocchi, A., Vassura, M., October 2003.

2003-14 Towards Secure Epidemics: Detection and Removal of Malicious Peers in Epidemic-Style Protocols, Jelasity, M., Montresor, A., Babaoglu, O., November 2003.

2003-15 Gossip-based Unstructured Overlay Networks: An Experimental Evaluation, Jelasity, M., Guerraoui, R., Kermarrec, A-M., van Steen, M., December 2003.

2003-16 Robust Aggregation Protocols for Large-Scale Overlay Networks, Montresor, A., Jelasity, M., Babaoglu, O., December 2003.

2004-1 A Reliable Protocol for Synchronous Rendezvous (Note), Wischik, L., Wischik, D., February 2004.

2004-2 Design and evaluation of a migration-based architecture for massively populated Internet Games, Gardenghi, L., Pifferi, S., D'Angelo, G., March 2004.

Dottorato di Ricerca in Informatica
Università di Bologna e Padova

Knowledge Management of Formal Mathematics and Interactive Theorem Proving

Claudio Sacerdoti Coen

March 2004

Coordinatore:
Prof. Özalp Babaoğlu

Tutore:
Prof. Andrea Asperti

Acknowledgments

Working in the research group of Prof. Andrea Asperti for the last four years has been a really enjoyable experience. Andrea has been not only a true guide for me in the marvelous and mysterious world of the λ-calculus. He has also created a pleasant atmosphere where everybody can expose his own ideas and receive fruitful comments. The most part of the time spent together working on the implementation has been also really funny.

This dissertation would have not been possible without the support of the HELM team. I am really grateful to all its present and former members for their steady work and their true friendship. In particular, I will never forget the long days — and nights — spent with Luca Padovani struggling with the internals of the OCaml bindings we realized together. The occupants of the adjacent rooms — as well as our poor office colleagues — will never forget our laughs, too.

During my Ph.D. studies I have had the opportunity of co-supervising several master theses. I would also want to thank the work of these brave students that debugged my code and dared use my interfaces. In particular, I would like to thank Stefano Zacchiroli, now a Ph.D. student, for never being tired of executing my orders and spending literally months to develop, upload, debug and update all the Debian packages of the library developed by the HELM team: it seems that the old mantra "release early, release often" has been understood by us as "release always".

I enjoy spending some time abroad to get in touch with other research groups. In particular, I would thank all the sites of the MoWGLI project — and the Lemme group of Sophia-Antipolis in particular — for giving me the possibility of spending short periods of time with them, providing all the facilities I usually miss in Bologna.

Many thanks to Yves Bertot and Herman Geuvers, referees of this dissertation, for their careful reading and the numerous corrections. Any mistake left is surely my fault.

Last but not least, I can not forget my two families. My parents, with whom I lived for the major part of my Ph.D. studies. Although they never really understood what I am studying, they always stimulated me to go on and to face the sacrifices as well as the satisfactions. And my wife. I am been very lucky to find the very patient woman that every computer scientist

deserves. Another one would have already divorced from the ghostly presence that haunted our house during the writing of this dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

I joined the research group of Prof. Andrea Asperti in 1999. At those time, Asperti was starting the investigation of a new field of computer science. The main topic was to make three already existing and almost disjoint fields of knowledge overlap, studying the possible outcomes and mutual benefits. These fields were *interactive theorem proving*, *Web publishing* and *digital libraries* development.

I will introduce the object of this dissertation by describing some of the actual problematics of interactive theorem proving. The pretty complex systems developed in this field are called *proof assistants*. Their aim is to aid the user in developing new results in the field of *formal mathematics*. Due to historical reasons — the development of several of these systems dates back to more than 10 years ago — the software engineering technique adopted in the development of the systems is still the "old" *application centric* design. Basically, it all boils down to thinking that the application is what really matters, and the developed libraries of mathematical results are just part of the tool, that becomes more and more expressive as soon as new results are added to its knowledge base. As a consequence, there was absolutely no effort in making the format of the libraries stable and publicly available, nor in making it backward compatible and based on well-known theoretical descriptions. Hence, it is not surprising that no tool except the system itself was ever developed to modify, index or interact with the library, except by eventually driving the system itself.

Moreover, all the proof assistants suffer from common additional *technological* problems, that are completely unrelated to the logical and theoretical differences that distinguish them and justify the existence of several competing systems. In particular, the two most critical missing features are *indexing and searching* facilities and proper *rendering and browsing* of the libraries. The lack of these facilities is not very surprising: they require skills that are different from those required to develop the core functionalities of the system. Moreover, these are big tasks, that require time and interaction with already developed technologies, such as databases, making the systems no longer self contained.

It should also be noticed that there exists a big gap between the ideal potential user of the system, and the actual average users. In particular, one should wonder in the first place *why* should an user adopt a proof assistant. For a computer scientist, a clear motivation does exist: there is software that is mission critical; this software needs a certification, that is a proof of its correctness w.r.t. its specification; informal proofs are often wrong, and a wrong proof could hide a bug. Thus computer scientists that need to certify a software must resort to formal proofs, either automatically discovered (e.g. by means of model checking) or developed by hand (using a proof assistant).

The same motivation clearly does not exist for a mathematician. We can argue that the aim of a mathematician is not really of being sure of the correctness of a proof. He wants to be sure — or even better convince his peers — that a certain thesis holds. Proofs are just a way to strengthen the belief. The more a statement is obvious, the less effort the mathematician is willing to spend on the proof. Unfortunately, formal proofs require a major effort to the mathematician, especially for the proofs of the "simple", "obvious" results that, once formalized, become non trivial and very time consuming. Moreover, since mathematics is a very complex subject, no interesting results can be addressed unless we already have a huge well-developed and accessible library of "classical" mathematical results. This library is utterly uninteresting to real world mathematicians, that do not want to waste their time in developing it, unless additional benefits arise.

To summarize, the large majority of the users of a proof assistant are computer scientists, whereas mathematicians have been so far looking at these tools with interest, but from a certain distance, waiting for stronger benefits and motivations. This biased user base forces some design choices that just make the systems more likely to be avoided by mathematicians. The main one is the stress put over provability instead of proofs: a computer scientist is just interested in the algorithm and the existence of a proof of correctness. The proof itself is uninteresting. On the contrary, mathematicians are interested in the proofs, and they would like to consult them (but at an appropriate level of detail, hiding all the "obvious" parts). Another example is constituted by the tools to manage the library. We already noticed that mathematical libraries are very stratified and require non trivial tools to manage them. On the contrary, the certification of a program is unlikely to rely on a huge library, unless the algorithm developed is based on non trivial mathematics [60].

My previous claims are supported by evidence: the only proof assistant that has attracted so far a few real world mathematicians and that has a library of non trivial mathematical results is the Mizar system [86]. From the foundational and theoretical point of view, the system is one of the less interesting, being mainly based on set theory and an essentially first order logic.

Nevertheless, since its creation much effort has been spent in improving the way proofs are read and written, adopting a language — and to a much lesser extent a notation — that is as close as possible to the usual mathematical practice. Moreover, there exists a library committee that periodically restructures the library, reorganizing it and making it more homogeneous.

The claim of Prof. Andrea Asperti was that all the described problems of proof assistants can be solved by exploiting technological achievements of the other two fields we have not described yet. The first one is *Web publishing*, that develops technology to communicate and to make accessible any kind of information, including mathematics. For instance, MathML is a markup language to faithfully render mathematical notation using a standard browser. The last field is the one of *digital libraries*, whose main topics are storing, indexing and searching any kind of information, by means of the creation of ontologies and reasoning logics to effectively data mine even huge and distributed libraries.

It should be obvious that Web publishing and digital libraries can effectively provide useful and well-developed tools that proof assistants can incorporate, reuse or interact with. It should also be evident that, since these functionalities are independent from the logical foundation of the tool, redesigning the proof assistant in a more modular way should eventually allow sharing of efforts: at least the searching and rendering components can be outsourced and put in common, allowing to invest more time on the core components of the system that require different knowledge and set of skills.

What is not so evident at first is the benefits of the digital libraries and Web publishing worlds that derive from the integration with mathematical reasoning tools. If we examine the status of the art of published mathematical documents, we notice that these documents, usually in PostScript or PDF formats, look like black boxes or static images: the mathematical information in the document is not machine readable. Thus we cannot search the documents for a mathematical formula. We cannot cut-and-paste a formula in the document. We cannot perform any correctness check over the document.

Let us now imagine a digital library of formal mathematical results, published on the Web by means of advanced rendering technologies and marked up at the statement level. The user could search the library for a particular statement, maybe up to trivial mathematical equivalences and notational differences. Once a set of proofs for the statement is identified, she can ask the system to check the correctness of the proof. She can then data mine the proofs, to understand on which axioms they are based or to derive informations over their complexity. The proofs can also be manually inspected, possibly at different level of details. Finally the user can extract the proof or part of it and reuse it in her own development, effectively building on top of already built results.

Four years have passed since the initial investigations of Prof. Asperti. In the meantime, the first steps to make the idea become a reality have been taken. The UE project IST-2001-33562

MoWGLI (Math on the Web: Get it by Logic and Interfaces), coordinated by Prof. Asperti, is struggling to provide a first prototype of a distributed library of mathematical documents — both formal and not formal — that is accessible on the Web and that is also indexed and effectively searchable. Although formal mathematics is not the only subject of MoWGLI, the focus of the research of Prof. Asperti is biased towards formal mathematics, since this represents at the same time the most ambitious goal and also, paradoxically, the simplest one: a formal library already holds lots of details and redundant information that can effectively be exploited for searching, indexing, parsing and rendering. In the same years, though, other groups have addressed related topics in the field of non-formal mathematics, that from now on we will call *rigorous mathematics*. In 2001 the groups got closer together, feeling a common identity. A new field of computer science was borne, and called *Mathematical Knowledge Management (MKM)*. The name was officially adopted during the First International Workshop on Mathematical Knowledge Management, held at Schloss Hagenberg, Austria, in September 2001. The European network MKM-Net (UE Project IST-2001-37057) was also conceived in the same occasion. In February 2003 the Second International Conference on Mathematical Knowledge Management was held in Bertinoro, confirming the liveliness of the field. At the same time several projects related to MKM have also started in the USA, and cooperations with the European counterparts are envisioned (see, for instance, [103] and the proceedings of the forthcoming Second North American Workshop on Mathematical Knowledge Management[1]). The author of this dissertation has actively participated in both the MoWGLI and MKM-Net projects since their creation.

Of the original goals of Prof. Asperti, the one that does not receive attention either in MoWGLI or MKM-Net is the study of the reorganization of proof assistants according to the *document centric* software engineering approach. It also involves the study of the solutions and new challenges that arise from the implementation of proof assistants on top of MKM technologies and tools. These topics have not been abandoned: starting from 1999 they have been investigated, among others, in the HELM project (Hypertextual Electronic Library of Mathematics, [62]) in Bologna, of which I have been a member almost since its birth. The topic of my Master Dissertation [107] has been the development of several prototypes of tools and libraries for MKM. After completion of my master dissertation, all the prototypes have been refined and turned into stable libraries and Web services by the collaborative work of all the members of the HELM and MoWGLI projects, to whom I am really grateful. Many more libraries and services have also been developed in the two projects. The topic of this dissertation is the development of yet another prototype. It is the prototype of a document centric proof assistant implemented on top of the libraries and services made available in HELM and MoWGLI. I can hardly say that there exists parts of the prototype that were developed by me alone. Prof. Asperti was always on my side, and the other members of

[1] `http://imps.mcmaster.ca/na-mkm-2004/`

the HELM team provided not only the libraries, but also continuous suggestions and sometimes topics of animated discussions. I am really grateful to all of them.

In the next section I will describe in more details the motivations of my work. The rest of the chapter, as well as part one of the dissertation, give introductory background material on MKM and the tools, libraries and services developed in HELM and MoWGLI. Readers interested in detailed descriptions must refer to the papers of the HELM team, to my Master Dissertation or to the Master and Ph.D. dissertations of the other members of the HELM project.

The second part of the dissertation describes the proof assistant prototype in details. Due to the prototypical nature of the implementation, the emphasis is not put on the implementation itself, but on the ideas that underlie it. Algorithms, in particular, are described as syntax-oriented judgements, and proofs of soundness are given. The implementation is not currently in sync with the algorithms described in this thesis, that are sometimes more advanced or simply bug free. They will be brought in sync when the prototype is turned into an actual application.

Following the usual convention for a scientific dissertation, from now on I will use the the plural form for the author of the dissertation. We will be careful in stating explicitly what are the contributions of the author alone, and what part of the work has been developed cooperatively.

1.1 Motivations

The main goal of our work is twofold. On one side we want to investigate the benefits that derive from adopting MKM technologies and tools in the development of a document centric and library centric proof assistant. On the other side, we want to identify new challenges in the development of proof assistants that try to provide MKM services and that are designed to be library centric.

The challenges and solutions did not present themselves all at once. On the contrary, the development of the prototype proceeded in a spiralling way. For instance, we started with the idea of reusing MKM services to manage distributed libraries. The choice implied that the proof assistant cannot have any notion of time, i.e. it cannot "go back" and modify part of the library during the development of the results. As a consequence, we could not implement in our proof assistant sections and discharging as they are provided, for instance, by the Coq proof assistant, that is the system we tried to clone when developing our own. Thus we developed a brand new variation over the logic of Coq, extending the logic (actually a type system). The obtained calculus is expressive enough to give to the user the same facilities as the original calculus. At the same time, though, it does not require any imperative side effect in the implementation of discharging. Thus, adopting a MKM technology ultimately lead up to modifying and extending almost all of the components of the proof assistant.

Let us give another example. There exist problems that are only partially solved by the current proof assistants and that are made critical in the context of MKM. One of them is parsing mathematical expressions. Mathematical notation is utterly ambiguous. To avoid the problem of parsing ambiguous expressions, the existing proof assistants have simply constrained the syntax available to the user, forcing her to adopt a notation that is non ambiguous, but it is also less mathematician friendly. Since a user does not switch frequently from one system to another, this is not a very tight constraint, as much as it is not for a programmer that must learn a new syntax for each new programming language. The problem is unavoidable in our context, though. Indeed, we want to develop "standard" mathematical parsers that can be used from several systems, and we want them to be usable also from novices that just bump into an on-line service such as a search engine. The user of an on-line search engine does not expect to have to learn a complex syntax or to be forced to adopt only one among the several alternative mathematical notations for the same concept. Thus in this dissertation we also developed a prototype of parser for ambiguous mathematical expressions. Although the general problem of parsing ambiguous expressions is extremely complex and has received no ultimate solution so far, we can implement something that works for the domain of mathematical expressions only. Indeed, we base our parser (that is a MKM component) on refinement, that is a logical operation provided by proof assistants.

Once again, the conclusion is that *the synergy between MKM and interactive theorem provers may improve both worlds!* Our thesis is that *we can already fruitfully exploit this synergy*. This Ph.D. dissertation is meant to provide evidence for the thesis.

1.2 Mathematical Knowledge Management

In this section we provide a "gentle" introduction to Mathematical Knowledge Management. In the next section we also identify one major subfield of MKM, Formal Mathematical Knowledge Management, that has always been the subject of the investigation of Prof. Asperti's research group. This dissertation is really concerned only with Formal Mathematical Knowledge Management, even if we will often speak more generally of Mathematical Knowledge Management.

The field of Mathematical Knowledge Management comprises, but is not limited to:

1. *Digitalization and preservation of mathematical knowledge.*
   The amount of mathematics developed in the last century is several orders of magnitude more than that developed in the rest of human history. Notwithstanding this, distinguished experts in the field of mathematical publishing such as Prof. Wegner[2] claim that the ma-

[2]Prof. Wegner is Chairman of the Electronic Publishing Committee of EMS (European Mathematical Society), Editor-in-chief of Zentralblatt MATH, the most comprehensive literature information service in mathematics, with Web access under EMIS and Scientific Coordinator of EMIS, the European Mathematical Information Service, providing a mathematics

jority of the fundamental papers in mathematics were written before the introduction of computers, and they are still available only in printed form[3]. Thus OCR technologies and image interpretation play a fundamental role in MKM [3, 44].

Published documents are not the only source of mathematic knowledge that must be retrieved and preserved. Exportation from the libraries of mathematical reasoning tools is also an important topic. Indeed, what happens to the library of an application centric system when the system itself is abandoned? For instance, it is well known that back in 1979 van Bethem Jutting checked Landau's Grundlagen in the Automath system [67], that has been the first pioneering proof assistant and that is no longer in use. The formalization of Landau's Grundlagen was lost with the dismission of the system, and it is has been no longer available in any exploitable form for several years. Only in 2002 the formalization was retrieved and made live again by Wiedijk, that reimplemented from scratch a version of the Automath system [120].

2. *Encoding of the information in machine readable formats.*
   Digital documents can be accessible at very different levels. The less interesting one is the presentation level, where only the notation is presented, without any meaning attached to it. The second level is that of a machine readable format (also called content level). Formulae and statements are represented at this level as abstract syntax trees. No formal semantics is attached to the syntax tree, but at least it is possible for machine reasoning tools to exchange mathematical information at this level when a *content dictionary* is provided. A content dictionary is a module that associates a semantics (understandable by the tool) to the syntax tree. Of course, there is no guarantee that the semantics attached to the same tree by two different tools is the same. In general, this will be the case only for tools (as CAS) based on a same weak logic foundation (unisorted first order logic).

   The last and more rich level is the level of formal mathematics, where all the statements are made completely unambiguous by encoding them in a particular (and hopefully consistent) mathematical foundation, such as set or type theory. Formal mathematics can be communicated between several tools without any misunderstanding or information loss. Unfortunately, it can be understood only by tools that are based on the same foundation. Translating the knowledge to another foundation is an open problem, whose only solution may be to cast the information back to the content level and try to formalize it again in the new system. Notice that the different problem of encoding one foundation into another is someway easier, and it was already solved for particular pairs of logics, type-systems or set theories. Notice, however, that the translation problem is not reducible in a satisfying way

portal with an electronic library as the main content.

[3]Private communication with Prof. Asperti, 2001.

to the encoding problem. For instance, if we translate set theory to type theory we expect to map the set $\{\emptyset, \{\emptyset\}\}$ viewed as the natural number 1 to the Peano number $(\mathrm{S}\ \mathrm{O})$ and not to a characteristic function that encodes the set. Whereas the encoding preserves provability, we cannot expect it from the translation. For instance, in set theory it is true that $\emptyset \in \{\emptyset, \{\emptyset\}\}$, but it does not make any sense[4] to ask if 0 belongs to 1.

For the case of documents written on paper, OCR technology may be charged of recovering the content associated to the presentation, whenever possible. Mathematical documents that have been type-set using a computer are not much easier to process, though. Indeed, the large majority of these documents are written in LaTeX. LaTeX does not provide any content level macros for mathematics. On the contrary, mathematics is type-set providing only its presentation. Thus, we must still face the problem of reconstructing the intended content. In both cases, the problem of providing a formal semantics to the content level is left open and it cannot be solved without human intervention.

3. *Standardization of the mathematical knowledge and creation of commonly accepted ontologies.*
   In order to improve communication and preservation of mathematical knowledge, MKM proposes standard languages (such as OpenMath [27], MathML Content [79] and OMDoc [97]) for the encoding of the content level. Since mathematical knowledge is not finitary, all these languages must be extensible, to allow the description of new concepts and definitions. Extensibility introduces the problem of consistency, that is typical of ontologies: given two ontologies independently defined, we must find a way to merge the two ontologies and propose automatic translations of models that adopt one ontology to another.

   Published documents are not the only sources of mathematical knowledge. Exportation from the libraries of mathematical reasoning tools is also an important topic. Among the systems that must be considered for exportation we find computer algebra systems (CAS), theorem provers and proof assistants. Exportation (and its dual, importation) may be an effective solution for the time being. In the long term, though, reimplementing the tools to adopt the MKM proposed standards as their native language will be more effective.

4. *Indexing and retrieval.* Marking up mathematical statements allows indexing and searching facilities that are clearly not possible for documents whose format is a page description languages (PostScript or PDF documents).

   Notice, however, that searching for mathematical formulae is much more challenging than full text search for textual documents. Indeed, a search for an occurrence of a formula

[4] We do not expect $\in$ to be translated to $\leq$, since we consider this fact as "accidental". We can easily find other examples where no corresponding predicate exists.

should always be up to the notation adopted, up to renaming of bound variables and up to "trivial" mathematical equivalences (e.g. between a concept as $\leq$ and its dual $\geq$).

5. *Rendering and publishing.* Rendering involves re-mathematizing [11] formal mathematics into its content level, and translating the content into presentation by picking up a notation for each mathematical symbol. Re-mathematization consists in removing all the foundational details and differences that were forcibly introduced during the formalization and that are meaningless at the content level. For instance during the formalization we could be forced to differentiate equality over natural numbers (that is decidable) from Leibniz's equality whose formalization has no computational content. The two equalities, though, are provably equal and they are mathematically the same (at the content level). Re-mathematization also means decoding the proof steps that were encoded as λ-terms if the system is based on the Curry-Howard isomorphism. Notice that the information that is explicit in λ-terms and the information that is usually explicit in a proof made by a mathematician are almost complementary: mathematicians write down the conclusion of each proof step, largely omitting the explanation of the proof step (that can be reconstructed by the human thanks to its knowledge and experience); on the contrary in a λ-term only the justification of the proof steps are recorded, whereas the conclusion of the proof step is left implicit and it can be easily reconstructed by the machine (but not as easily by the human being).

   Publishing does not differ much from traditional publishing, except for the problem of consistency, that is made more critical in the case of mathematics: an inconsistent mathematical development is utterly unuseful.

6. *Exploitation of the knowledge (e.g. for interactive learning).* One project that is focused on this topic is ActiveMath [83]. The experience of the ActiveMath team is that interactively teaching mathematics is more complex than teaching other subjects, and it requires ad hoc techniques, tools and metadata schemata.

7. *Management of intellectual property.*

8. *Enhancement of interoperability between systems dealing with digitalized mathematics.*

Simplifying a bit, we can say that MKM deals with any aspect of mathematical knowledge, except the fundamental task of creating new mathematical knowledge. Nevertheless, it is expected that MKM will bring great benefits in the process of creation of new knowledge.

Notice that MKM deals with rigorous mathematics as well as with formal mathematics. Nevertheless, the complexity of the two scenarios is not comparable. Formal mathematics brings much more information, either implicitly or explicitly. Thus the main difficulty becomes rendering and filtering all the available information to make it understandable and useful to a human being. In other words, dealing with formal mathematics always implies a complex process of re-mathematization of the knowledge base [11]. On the contrary, when we are dealing with rigorous mathematics the problem is exactly the opposite: the information is under-specified and ambiguous. Thus a disambiguation process, followed by reconstruction of the missing information, is required. When disambiguation and information reconstruction are not feasible, MKM boils down to the (not very exciting) task of fixing ontologies, proposing standards and making all the tools agree on them.

One could ask why Mathematical Knowledge Management should be studied as a new subject, instead of being just a specialized branch of Knowledge Management. MKM has several characteristics that demand new methods and that make ineffective those of "general" knowledge management.

- There is no universal agreement even on the most common mathematical concepts. Different textbooks often define the same concept in slightly different and incompatible ways. Mathematicians learn to operate in this inconsistent world of definitions by mentally converting the definitions and adapting the statements to reflect the changes. As a consequence, fixing an ontology is already a major task, and imposing it to the community is an almost hopeless effort, that might be accomplished only over a long period of time.

- Mathematical statements are not logic independent. Fixing a logic means changing the set of mathematical facts that can be validated into the logic. Moreover, the mathematics that can (and actually must) be developed in logics that are "weaker" than classical logic are richer. For instance, in constructive logics double negation is not equivalent to affirmation. Thus, it is not possible to always completely characterize negative concepts in terms of positive ones by means of negation. As a consequence, many definitions must be duplicated into a positive and a negative definition, leading to a much richer theory. An example is given by (constructive) formal topology: whereas in classical topology closed sets can be defined in terms of open sets, in formal topology [112] they must be defined independently (even if duality plays an important role).

  Another difference induced from the choice of a constructive logic is the difference between decidable and undecidable properties. In classical logic the notion of decidability is irrelevant. In constructive logics, instead, it plays an important role, especially when the logic allows program extraction. In this case, the semidecidable half of a judgement bears computational content, whereas its negation does not (and it can simply be defined as the negation of the semidecidable half). As a consequence, the mathematics that can be developed in the logic is richer, since it distinguish between two judgements that are classically equivalent, but that differ in their computational content (as a semidecidable judgement and the negation of its negation). Exploiting the distinction, for instance to minimize the useless code extracted, is a non trivial task that may have important practical consequences [46].

Mathematical Knowledge Management cannot ignore the issue of the logic that underlies a mathematical development. In particular, since the attempt to found a ground logic that everyone should agreed upon failed (consider, for instance, the outcome of the Qed project [58]), MKM should address the problem of the translation of mathematical facts from one foundation to the other. There is no corresponding problem in the general framework of Knowledge Management.

- The gap between content and presentation (in the sense of MathML [79], for instance) is deep. The same notation can be interpreted in several different ways, and a single concept can be given several notations. Although mathematicians are really interested only in the content, for practical reasons they do not abandon presentation as the principal way to communicate and write down mathematics. Forcing a mathematician to put more effort into writing things directly at the content level is a possibility only in restricted and well-motivated communities (e.g. when the request is made from the publisher of an on-line journal, as it is done in MoWGLI [89]). Thus MKM is likely to always be forced to study heuristics to fill in the gap between content and presentation.

  On the contrary, in the general realm of Knowledge Management it is easier to impose tools and languages that close the gap between what is meant and how it is described.

- Mathematical Knowledge is semantically rich, whereas general knowledge is usually quite poor compared with mathematics. The amount of information and semantics available make several tasks of Knowledge Management much easier. In particular, whereas the main problem of Knowledge Management is data mining, i.e. information discovery, the main problem of MKM is information organization, communication and exploitation.

- Indexing and retrieval of mathematical information, formulae in particular, is extremely more complex than textual search, as already noticed. Thus it requires a completely new set of algorithms and tools, already partially studied in the context of search up to isomorphisms [38].

1.3 Formal Mathematical Knowledge Management

This thesis is concerned only with the subfield of Mathematical Knowledge Management that deals with formal mathematics. We call it Formal Mathematical Knowledge Management.

As the problems of Mathematical Knowledge Management differs from those of general Knowledge Management, those of Formal Mathematical Knowledge Management are often peculiar. In this section we briefly present them again. More precise informations can be found in the deliverables of the MoWGLI project [88].

- *Development of document centric formal reasoning tools.*
  This is the main topic of this dissertation.

- *Development of exportation modules for existing formal reasoning tools.*
  It will be addressed in details in Chap. 6.

- *Development of tools to manage distributed libraries of mathematical results.*
  This task is not specific to Formal Mathematical Knowledge Management. We can just develop and reuse general tools for MKM.

- *Development of tools to analyze the macroscopic structure of mathematical developments.*
  For instance, it is interesting to be able to visualize the graph of dependencies between definitions, statements and proofs. Data mining the graph it is possible to understand the relative importance of statements or definitions. It is also possible to understand which parts of a library depend directly or indirectly over a given set of axioms, and it is possible to judge in advance the consequences of a definition change.

- *Development of tools for the re-mathematization of formal knowledge.*
  The problem consists in translating a foundational encoding of a mathematical result, where much information is usually left implicit, into the corresponding content encoding. Reconstructing the implicit information is not at all a trivial task [34]. Moreover, the translation is usually not an injective function, and it may be difficult to recognize all the inputs that produce one content expression. As a consequence, we are often obliged to perform complex logical normalizations of the informations, or to provide only approximated and suboptimal content representations [11].

- *Development of indexing and retrieval tools.*
  Indexing and retrieval are necessary both at the content level (i.e. after remathematization) and at the formal level. Searching at the content level is necessary when we are interested in retrieving a given result, and we do not care about the way the result is formalized. On the contrary, searching at the formal level is necessary when we are developing new formal results and we want to build our development on top of the existing library. In both cases, the search is complicated by the necessity to perform it up to notational conventions, renaming of bound variables and logical and mathematical equivalences. Notice that, at the

content level, the number of equivalences is greater than at the formal level, since different formalizations of the same concept are identified.

- *Development of tools for the rendering of content level mathematical expressions.*
  This task is also not peculiar to Formal Mathematical Knowledge Management.

1.4 Organization of the Dissertation

The dissertation is divided into two parts. The first part is a brief overview of some of the tools for the management of formal mathematics that have been developed in the projects HELM and MoWGLI. The second part describes the theory and very briefly the implementation of the proof assistant prototype that has been developed in HELM. According to our requirements, the prototype of part II is designed and implemented on top of the services described in part I. Thus the division of the dissertation into two parts is quite natural.

The level of detail of the two parts is not homogeneous: the second part is more detailed, while the first one is just a brief overview, although some chapters are definitely more detailed. The lack of coherence is well justified. First of all, the work described in the first part has been developed in a collaborative way, or it has been improved and reimplemented after the initial prototyping experience of the author. Thus, it is not completely suitable for a Ph.D. dissertation. Secondly, it is material that has been already published in the past, either in research papers or as technical reports of the MoWGLI project. We decided to include in this thesis a brief review of the material since several design choices of the second part cannot be understood without knowing the first part. The chapters that are more detailed are those that describe recent or unpublished material, or material that has been entirely developed by the author, or material that is essential in the author's Ph.D. studies.

The first part of the thesis begin with Chap. 2 that describes the distribution model adopted in HELM, that is based on a distributed library of documents. On top of the distributed library we impose a logical and hierarchical view of the library (a tree). In this chapter we also present describe the tool that creates the logical abstraction of a hierarchical library on top of the physically distributed library.

Chap. 3 describes the three levels of encoding of mathematical information that are exploited in the HELM and MoWGLI projects. The XslT stylesheets developed to change more semantically rich encodings into less semantically rich encodings are also described. Finally, we describe UWOBO, that is a Web service that manages XslT stylesheets.

A tool to examine the dependency relation between the documents in the library is very briefly presented in Chap. 4.

Then, in Chap. 5, we discuss the methodology adopted to index and retrieve definitions and lemmas in the distributed library of HELM. The basic idea is to provide search by matching or unification. Since the library is distributed, the implementation cannot simply iterate the matching operation over the whole library. The solution proposed consists in identifying a set of metadata that captures the invariants of the unification procedure, and use them to index the library by means of a Web spider, that is an agent that browses the library to collect informations. Then, when a pattern is submitted, a query over the library of database of metadata is issued. The answer of the query consists in a list of documents that are matching candidates. The documents are downloaded and the matching operation is performed only over them. Proofs of correctness of our approach are also given at the end of the chapter.

The last chapter of the first part of the dissertation is Chap. 6. It is devoted to the description of several guidelines and design principles that we discovered when implementing and updating a couple of exportation modules. An exportation module is a tool that allows to extract the logical information that is present in the library of a proof assistant, saving it in an XML format that reflects the logical system the proof assistant is based on. Our experience shows that writing such a tool is a long and error prone task, that presents several pitfalls and that requires general guidelines to avoid common mistakes.

The second part begins with Chap. 7 that presents a more detailed list of requirements for the proof assistant prototype. It also briefly sketches the architecture adopted.

Chap. 8 starts with a description of the syntax, reduction and typing rules of the Calculus of (Co)Inductive Constructions (CIC), that is the calculus our proof assistant is based on. Then some extensions to the calculus introduced by the developers of the Coq proof assistant are presented. One of these extensions, discharging, is particularly appealing, but it is implemented in Coq in a way that is incompatible with the static nature of the library that is imposed by the MKM philosophy. Thus in Sect. 8.3 we propose an extension of CIC that replaces discharging, but keeps its features from the user point of view. The previous statement is made precise and a proof of it is given. Moreover we prove the consistency of the calculus, by giving a type preserving translation to CIC.

Chap. 9 describes the theory — and partially the implementation — of the kernel of our proof assistant. In particular we describe a reduction machine for our calculus that is parametric over the reduction strategy. We also describe a syntax oriented presentation of the typing rules of the calculus that implicitly defines the algorithm implemented in the type-checker. An important difference w.r.t. the traditional implementation is the way the environment is handled: instead of assuming — and granting as an invariant — the well-formedness of the environment, our operational presentation, given a term to type-check, constructs the well-formed environment that is the smallest subset of a distributed library that is required to type-check the term.

Chap. 10 describes the first layer of the proof assistant that is built around the kernel. This level implements logical operations whose correctness is not critical, in the sense that a bug in the implementation is detected by the kernel. In particular, we present the unification rules adopted by the unification engine, and the refinement rules. We also present a new efficient algorithm for parsing and disambiguating mathematical expressions that is heavily based on refinement.

Chap. 11 deals with the tactics implemented in the proof assistant. Since the implementation and the set of tactics implemented is pretty standard, we prefer to spend some time in introducing a metric over proof objects. Since proof objects are central in our view of MKM libraries, we are deeply interested in their "quality". Our interest is motivated by the poor quality of the proof objects generated by the current implementation of the tactics in the mainstream tools. In other words, more interest in the proof objects is a consequence of putting the focus in the proofs and not in provability[5].

Finally, Chap. 12 presents the graphical user interface of the proof assistant. According to the requirements, the interface allows direct proof object manipulation, and it is based on the rendering technology developed in HELM and briefly described in the first part of the thesis.

1.5 Original Contributions

The starting point of this dissertation is the observation that *the synergy between MKM and interactive theorem provers may improve both worlds.* Our thesis is that *we can already fruitfully exploit this synergy*.

Providing evidence for the thesis is not the only original contribution of this dissertation. The following achievements are all original contributions:

- We identify a generic class of metadata and constraints that can be adopted to efficiently implement unification based searches in a distributed library. The basic idea is to identify a set of unification invariants that can easily be indexed. A Web spider can be run over the distributed library to extract the metadata that capture the invariants, and store them in a database. When an unification pattern is proposed, a query over the metadata set is extracted from the pattern, and submitted to the database. The result is a set of candidates that could be unifiable with the pattern. The candidates are downloaded and unification is attempted.

  This part of the work is the result of a collaborative effort of the HELM team. The original idea was initially proposed by Prof. Asperti. Ferruccio Guidi is the author of the query language that allows an efficient implementation of the idea. The author of this dissertation

[5]It is also a consequence of the fact that we consider the proofs to be the proof objects. This belief is shared only by few members of the community.

contributed with Prof. Asperti to the metadata extraction tool. He also collaborated with Ferruccio Guidi to the translation of the high level constraints to the low level query language. Finally, he is the only author of the formal analysis of the algorithm that is presented in Chap. 5.

- We spent an awful amount of time in developing, implementing, and updating the exportation module of the Coq system. We also collaborated to the development of an exportation module for the NuPrL system, and we had several discussions with authors of other exportation modules. The knowledge derived from this work is described in Chap. 6. The hope is to provide useful suggestions to other authors, in order to avoid common mistakes and wrong approaches.
- The Calculus of (Co)Inductive Constructions with Explicit Named Substitution, introduced in Sect. 8.3, is a new extension of the Calculus of (Co)Inductive Construction, presented for the first time in this dissertation.
- In Sect. 9.3 we introduce an original reduction machine for CIC extended with Explicit Named Substitutions that is parameterized over the parameter passing strategy. This is the result of collaborative work with Prof. Asperti. The extension of the machine to the calculus extended with Explicit Named Substitutions and the proof of correctness of the machine have been made by the author of this dissertation alone.
- The efficient algorithm for disambiguation of mathematical expressions presented in 10.4 has been discovered and implemented by the author alone.
- The definition of metric over tactics and the definition of overkilling tactics is also an original contribution of the author alone.

All the other parts of this dissertation describe either collaborative work, work that is not original or technical work that is an adaptation of already existing results to the calculus and algorithms adopted in the dissertation.

1.6 Related Publications

Most of the research papers and publications of the author are related to the material presented in this dissertation. In particular, a few papers have been included almost verbatim without any major modification. Some other papers represent introductory material to the dissertation. A few other papers are detailed descriptions relative to the first part of the thesis.

- Papers included almost verbatim

  - F. Guidi, C. Sacerdoti Coen. Querying Distributed Digital Libraries of Mathematics. In *Calculemus 2003*, Aracne Editrice S.R.L., Therese Hardin e Renaud Rioboo editors. ISBN 88-7999-545-6, pp. 43–57.
  - C. Sacerdoti Coen. From Proof-Assistans to Distributed Libraries of Mathematics: Tips and Pitfalls. In *Proceedings of Mathematical Knowledge Management 2003*, Lecture Notes in Computer Science, Vol. 2594, pp. 30–44, Springer-Verlag.
  - Tactics in Modern Proof-Assistants: the Bad Habit of Overkilling. In *Supplementary Proceedings of the 14th International Conference TPHOLS 2001*, pp. 352–367. Edinburgh, Scotland, UK, September 2001.

- Introductory material
  - A.Asperti, F.Guidi, L.Padovani, C.Sacerdoti Coen, I.Schena. Mathematical Knowledge Management in HELM. In *Annals of Mathematics and Artificial Intelligence*, 38(1): 27–46; May 2003.
  - C.Sacerdoti Coen, S. Zacchiroli. Brokers and Web-Services for Automatic Deduction: a Case Study In *Calculemus 2003*, Aracne Editrice S.R.L., Therese Hardin e Renaud Rioboo editors, ISBN 88-7999-545-6, pp. 43–57.
  - A.Asperti, L.Padovani, C.Sacerdoti Coen, I.Schena. HELM and the Semantic Math-Web. In *Proceedings of TPHOLS 2001*, Springer-Verlag, Lectures Notes in Computer Science (LNCS) Series, 2512.
  - A.Asperti, L.Padovani, C.Sacerdoti Coen, I.Schena. XML, Stylesheets and the re-mathematization of Formal Content. In *Electronic Proceedings of EXTREME 2001*.
  - A.Asperti, L.Padovani, C.Sacerdoti Coen, I.Schena. Towards a Library of Formal Mathematics. In *Technical Report of TPHOLS2000 Conference*, Portland, Oregon, USA, August 2000.

- Detailed descriptions
  - The more detailed source of informations for the MoWGLI project are the MoWGLI deliverables, that can be downloaded from [88]. So far, the following deliverables have been published:
    * D1.a Preliminary Report on Application Scenarios and Requirement Analysis.
    * D7.a Dissemination and Use Plan.
    * D0.a Self-Assessment parameters and criteria.
    * D1.b Structure and Meta-Structure of Mathematical Documents.
    * D1.c Distributed Digital Libraries: development, archiving, retrieving.

    - D2.a Exportation module.
    - D4.a MathML Rendering/Browsing engine.
    - D3.a Metadata for Mathematical Libraries.
    - D3.b Metadata Model.
    - D0.b First Self-Assessment Report.
    - D2.b Document Type Descriptors.
    - D2.c Stylesheets to intermediate representation (formulae).
    - D2.d Stylesheets to intermediate representation (proofs).
    - D5.a Overall Architectural Design of the Distribution Model.
    - D4.b First MOWGLI Prototype (browsing, rendering and consultation).
    - D4.c Prototype functionalities for assisted annotation.
    - D4.d LaTeX-based authoring tool (first prototype).
    - D2.e Presentational Stylesheets (formulae).
    - D2.f Presentational Stylesheets (proofs).
    - D2.g Tools for automatic extraction of Metadata.
  - Further details over the HELM project can be found in the Ph.D. and Master dissertations of the present and former members of the projects:
    - C. Sacerdoti Coen. *Progettazione e realizzazione con tecnologia XML di basi distribuite di conoscenza matematica formalizzata.* Master dissertation, University of Bologna, 2000.
    - L. Padovani. *MathML Formatting*, Ph.D. dissertation, Bologna, Technical Report UBLCS-2003-3, February 2003.
    - F. Guidi. *Searching and Retrieving in Content-Based Repositories of Formal Mathematical Knowledge.* Ph.D. dissertation, Technical Report UBLCS-2003-6, May 2003.
    - I. Schena. *Towards a Semantic Web for Formal Mathematics*. Ph.D. dissertation, Technical Report UBLCS-2002-6, March 2002.
    - S. Zacchiroli. *Web services per il supporto alla dimostrazione interattiva*. Master dissertation, University of Bologna, 2003.

1.7 Non-Standard Mathematical Conventions

The following non-standard or not universal mathematical conventions will be adopted throughout the dissertation:

- Function composition order: $f \circ g$ is defined as the function $x \mapsto g(f(x))$.

- Overloading of scalar functions: the application $f(l)$ of a function $f : T \rightarrow T'$ to a list $l = [x_1, \ldots, x_n]$ of values of type T is a shorthand for the list $[f(x_1), \ldots, f(x_n)]$ of applications of f to the elements of l. The case where l is a tuple is handled consistently. Since we guarantee that the of f and its argument is always explicitly stated, the notational overriding is never ambiguous.

- Application syntax: for the application of a function f to an argument x we adopt both the usual notation $f(x)$ and the λ-calculus notation $(f\ x)$, according to the topic presented.

Part I

Development of Tools for the Management of Formal Mathematics

Chapter 2

The Management of a Distributed Library

The first and most fundamental design decision in the management of formal mathematics is the choice of a distribution model. To match the requirements of the HELM project, the HELM team opted for a distributed library of mathematical knowledge. The library is constituted of both data (in XML format [43]), metadata (in RDF [106]) and programs that describe XML transformations (in XslT [123]). The preliminary requirements that motivated this choice will not be explained in this dissertation. The interested reader can find them in [10].

The adoption of a distribution model based on a distributed library of documents is absolutely not novel in the realms of digital publishing and software engineering. Notwithstanding this fact, no existing proof assistant adopts it. Indeed, even the simple notion of collaborative development has not been investigated much in the domain of interactive theorem proving. There may be two possible explanations to this situation. The first one is that there is actually no demand for these features. Although this may be partially true — the input of all the mainstream proof assistant is a textual file, that can be managed with version control systems as CVS — the lack of a global view over the knowledge already encoded in the library of one system has tangible negative effects. Indeed, if we consider for instance the libraries of Coq and NuPrL, we face very disorganized structures: general concepts (such as transitivity or the notion of group) are defined and studied, but they are not used in the definition of other concepts; some notions are re-defined several times, often in perfectly equivalent ways; equivalences and relations between results proven by different users are completely missing. The ultimate consequence of the disorganization is useless duplication of efforts and lack of a solid standard ground for the development of more advanced results. Notice that the users are not responsible for the situation. Indeed, they are victims of it: the library is already disorganized and there exists no effective tool to index and retrieve already developed results; moreover, to reuse a single result put in a library L, the user has to load and type-check the whole library L. As a consequence, especially in the case of concepts that are easily redefinable, the user is forced to redefine them from scratch, including them in the current development (that surely is the wrong location!). In this way the situation gets worse and worse. Notice also that CVS is not the appropriate tool for collaborative development of a mathematical theory. Indeed, the only property granted by CVS is that two users cannot modify the same lines at the same time. Unfortunately, it is still possible for two users to apply two changes to two different parts of a file that result in an invalid file (i.e. a no longer valid proof, either rigorous or formal, or a no longer valid script). A possible solution would be an advanced version of CVS, integrated with a proof assistant or a similar tool, that would identify conflicts by checking again the proof for validity. Such a tool does not exist yet.

We mentioned the possibility of the existence of a second motivation for the lack of collaborative development tools and distributed libraries. The motivation could be that the development of a formal mathematical library is too complex to allow distributed management and collaborative development at all. We can search evidence for this hypothesis in the experience of teams involved in the creation of large mathematical libraries.

The only two teams that are taking seriously the task of creating a non trivial distributed library of formal mathematics are the Mizar group and the CCoRN team in Njimegen.

The Mizar project is the oldest. Almost since its beginning, the focus has been on the creation of a coherent library of mathematical knowledge, that is managed centrally. The experience of the Mizar team shows that the library must continuously be revised, almost each time that a new result is accepted into the library. The revisions are made necessary by the introduction of new general definitions and results, that must be propagated over the already existing library. As an example, we expect two-dimensional functions to be defined pretty early in the development of the library, as well as the notion of derivative and integral in the two dimensional cartesian space. Later on, multi-dimensional calculus is developed, and derivatives and integrals are redefined in the more general settings. To avoid having pairs of different and unrelated formulations in the library, the library committee has to revisit the bi-dimensional case, for instance to prove that the definitions given are equivalent to the general ones in the restricted setting.

The need of centralized management is justified by the complexity of updating already developed results. Indeed, to perform the necessary modifications, the user must know quite well the part of the library that she is going to modify. Since the user is probably not the author of that part, she could not be able to apply the changes in a timely manner. Moreover, this is a burden w.r.t. the main aim of the user, that was already reached when the new part of the library was finished. Thus the user is not pushed into performing the task and, if she is not forced to, she will refuse to do it.

The CCoRN project does not provide much insight over this topic for two reasons. First of all the development is not really distributed, since almost all of the authors of the development are physically located in Nijmegen. Secondly, so far the project has covered only algebra and basic calculus, that are both well established mathematical subjects. As a consequence, it was possible for the team to develop a very detailed plan before starting coding, highly reducing the need for revisions. Notwithstanding the previous two observations, the CCoRN library has already been subject to several revisions, and there exists now a paper that describes a set of rules and conventions to extend the library that is aimed at minimizing the need of further revisions.

To summarize, concrete experiences in centralized management of the library exist and can be considered fairly successful. Whether the centralized scenario will scale to larger user communities and larger libraries is an open question. On the contrary, fully distributed development has so far been too chaotic. As an excuse, no tools for collaborative development have been available. One of the main goal of the HELM project is to provide the missing tools, in order to accumulate further experience and allow fair comparisons between the two approaches. It is likely that the ultimate solution will consist in a two levels approach, where there exists "stable" libraries, that are centrally managed, and work in progress that is developed in a collaborative way. Multi-level solutions have also been proposed in the literature [68].

Since the distribution model of the HELM project is based on a distributed library, the more critical design choice of the HELM team has been whether the developed tools work directly over the physical, distributed library, or whether there exists a logical organization of the library that is independent from the physical organization. The solution adopted has been to logically organize the library as a tree whose leaves are the mathematical objects (i.e. theorems and definitions). The objects are identified by means of standard URIs [116], that describe the path of the object in the logical tree. The internal nodes of the tree can be used to provide a first hierarchical organization of the library. Notice, however, that it is well known [61] that a tree shaped hierarchical organization is not sufficient for mathematical knowledge. We expect to be able to impose in the future more refined structures by means of metadata. The high level tool that will manage the additional organization will be independent from the low level tool that manages the logical library as a simple tree.

Since the HELM team opted for a physically distributed library that has a logical hierarchical organization, we need a tool, called the GETTER, to resolve URIs into physical addresses (URLs) and to retrieve the files. The next section briefly describes the GETTER, that is actually implemented as a Web service. The reason to include a brief description of it in this dissertation is the fact that our proof assistant prototype — its kernel, to be precise — behaves as an HTTP client and it is able to retrieve the documents from the distributed library by contacting the GETTER. In a way that will be made more precise in Chap. 9, the proof assistant is designed to blur the distinction between the library and the environment.

2.1 The GETTER.

Before describing the services provided by the GETTER and its implementation, let us review briefly the additional requirements of HELM that influenced its implementation.

An important requirement of HELM is to keep a minimum burden on the user: no client-side software should be required to consult the library and as little as possible should be required to interact with it. In the same way, no client-side software should be required to contribute to the library by publishing a new result[1]. In particular, we want any user with a Web space (either HTTP or FTP) to be able to publish a document/object without having to install any particular software. The main reason is that the Web space of a user is often hosted by a provider that does not allow new software to be run. Moreover, the simple HTTP publishing model has already proved highly effective in creating fully distributed hypertextual libraries of knowledge, which we see as a motivation against a more centralized solution, such as a net of cooperating databases to which the user can submit a contribution (the solution adopted in MathWeb [81]).

Our distribution model has been designed also to exploit the peculiar property of immutability of formal mathematical documents. The reason for having immutable documents is that the correctness of a document A that refers to a document B can only be guaranteed if B does not change. Notwithstanding this, new versions of a mathematical document could be released (for example if a conjecture is actually proved or if an alternative definition is given). To draw a parallel, this is what happens with packages of operating systems distributions that, once released, can be modified only by changing their version numbers.

Since documents are immutable and are already identified by logical names (URIs) rather than physical names (URLs), many copies of them can be stored on different servers, and users are free to retrieve them from the nearest or less overloaded server, achieving load balancing.

Finally, users cannot be forced to retain a copy of their documents forever, even if other documents refer to them. By allowing different copies on different servers, though, we give users who are interested in a contribution the possibility of making a local copy of it and start distributing it. In this way, interesting documents tend to augment the number of their instances, avoiding the danger of disappearing, while uninteresting ones could simply stop wasting space, being deleted.

We leave open the problem of choosing a naming policy that prevents independent users from choosing the same URI for different documents, thus creating name clashes. To face the issue, one solution is a centralized naming authority. A simpler solution could just be considering the URIs as completely meaningless identifiers, leaving to the high level tools the problem of organizing the library by means of metadata. In such case, we could easily adopt a naming scheme that avoid

[1] In this context, contributing means just publishing, i.e. making a development available to others; it does not mean creating a new object or document.

by construction the problem of name clashes, as the one adopted for package names for the Java programming language.

Having illustrated the requirements, we can go on with the description of the implementation. We will skip the actual details since they are not important for the sake of our thesis. The interested reader can consult [124] for precise details on the current implementation.

The *getter* is a Web service whose main task is mapping an URI to the relative URL and then retrieving the identified document, that is returned to the user. According to the similarities with package management systems, the implementation has been largely inspired by the APT package manager[2].

In more details, the main method of the GETTER takes an URI and returns a copy of the document identified by the URI, downloaded from a distribution site that provides it. In order to know which documents a server provides, each server publishes a list of the URIs of its documents associated with the respective URLs. Users of the GETTER, instead, configure the GETTER with an ordered list of servers. Periodically, the getter contacts each server in the list and retrieves the list of documents the server knows, along with their associated URIs. The information retrieved is stored in a local table (implemented as a NDBM database) that maps each URI to the URL of the first server in the list providing a copy of the document. The table is consulted to resolve the URI every time the main method of the GETTER is invoked.

As XML files are very verbose, we give users the possibility of storing them in a compressed form. The GETTER is also responsible for decompressing the documents before giving them back to the client. Moreover, the GETTER is able to retrieve the files from an HTTP, FTP or NFS server, passing them back to the client using a uniform HTTP server interface. Finally, as the GETTER is supposed to reside closer to the user than the distribution server, it implements a caching mechanism that reduces downloading time of already retrieved documents. All these reasons forced us not to use the HTTP redirect method to map URIs to URLs, as done by PURL which is an otherwise similar tool to resolve persistent URLs (URIs, in fact) to URLs[3].

The GETTER also provides other administrative and less interesting methods. Among them there are methods that force the reconstruction of the URI resolution table, methods to notify the GETTER about the publication of a new object and methods to check the existence of a document associated to a given URI.

As a final remark, we highlight again the twofold nature of the GETTER: on one side it is a Web service that can be contacted by HTTP clients, thanks to the HTTP GET binding; on the other side it is an HTTP client that can download files from a remote HTTP server or that can invoke methods of other Web services that have an HTTP GET binding. This compositional nature is typical

[2] `http://www.debian.org`

[3] PURL (`http://www.purl.org`), being general-purpose, cannot rely on document immutability. Hence, at most one copy of a document can be available and no load balancing is provided.

of all the Web services developed in HELM, and it constitutes the architecture of HELM. In particular, in HELM we developed high level Web services that are implemented by means of HTTP calls to lower level Web services. We also implemented Web services that perform recursion by invoking each other methods by means of HTTP calls. Finally, the proof assistant prototype that is the main argument of this dissertation is an HTTP client that contacts several of the HELMWeb services to provide its functionalities.

2.2 Comparisons with Related Work

It is commonly agreed that MKM mathematical libraries must be distributed and that collaborative development must be possible. There are at least two possible interpretations, though, for the notion of distribution: distribution of data and distribution of computation.

The first notion of distribution is the one that is dealt with in this chapter: the library itself is made of several documents (or database entries) that are physically distributed all over the Web. The main problem that must be addressed for distribution of data is that of creating a logical view that hides the physical model. The logical view should grant at least *name persistence* (i.e. the logical name should be independent from the physical location, that can change). If multiple physical locations are associated to a logical name (e.g. for load balancing), the additional problem that must be faced is the one of *coherence* of the information. Moreover, we imposed to ourselves additional constraints on the complexity of the software that is necessary to publish a document. In particular, publishing in HELM only requires a Web space (either HTTP or FTP), and browsing of the library only requires a Web browser. Web services that can be remotely executed are responsible for the communication between the publisher server and the client side, and they create the logical abstraction.

In the context of data distribution, collaborative development must be interpreted in the sense of a control and revision system: the system must grant the coherence of all the copies of a document in a distributed library, and it must allow different users to work on a private copy of a document that will be merged with the official copy only in the future. Conflicts between independent modifications must be spotted and notified to the user, that should also be provided with tools to solve the conflicts properly.

No tool for collaborative development has been developed yet in HELM. One possibility that is under discussion in MoWGLI is to adopt a hierarchical control and revision system, based on the one presented in [68]. To deal with formal mathematics, though, the tool should be enhanced by replacing XML diffing (used to spot the conflicts) with formal tools, based on type-checking, that grant the logical consistency of the whole distributed library[4]. This is yet another example

[4]XML diffing only grants validity of the library. It enhances the textual diffing operation used by CVS by performing

of an area where FMKM tools can have improved functionalities w.r.t. the corresponding tools in the more general framework of MKM.

The second notion of distribution is distributing computation. It deals with concurrent and remotely distributed programs that communicate to retrieve mathematical results and to solve complex problems. For instance, a theorem prover could ask a remote CAS for the normal form of a complex expressions. On the converse, a CAS could ask an external theorem prover to prove the side conditions of a simplification it wants to perform. In HELM, distribution of computation is addressed by means of an architecture based on Web services: each service is implemented as a Web service, that can remotely invoke the methods of another Web service. For instance, to manipulate a document in the distributed library of HELM, all the HELM tools invoke the main method of the GETTER to locate and retrieve the document.

The principal problem involved with distributing computation is the orchestration of the different services provided. In particular, each (Web) service must be able to advertise its functionalities to other services and it must be able to locate the services that are able to solve a problem in the best way. The problem is much more difficult than what is seems at first glance. For instance, even the notion of "best way" is dependent on the actual problem the client needs to solve: best could mean fastest, but also most precise or semidecidable, but in a fixed amount of time. This and related problems are the main topic of the Monet project [87], that proposes an architecture for Mathematical Web services Discovery and Advertisement. In particular, the Monet group is developing ontologies for describing mathematical problems, solutions and algorithms, in order to use them for discovery and advertisement.

In HELM and MoWGLI the problems addressed in the Monet project were intentionally avoided. Indeed, the set of Web services is fixed, and each Web service method has a set of parameters that gives the coordinates of the Web services it needs to contact. Thus there is no dynamic orchestration in HELM: the user chooses once and for all the location of the Web services she intends to use and no discovery phase is required. We expect to be able to easily integrate in our Web services the advertising and discovery components that will be developed by the Monet group, in order to integrate the HELM services in a larger and dynamic network of collaborative mathematical agents.

The Monet project was not the first one to addresses the problem of mathematical service advertisement and discovery, although it is the first project that address the problem in the framework of Web services technology. In particular, the MathWeb Software Bus is an older project developed at Saarbrüken. MathWeb [69] provides a distributed architecture based on brokers and services for the management of service advertisement and discovery; it also provides the

the comparison up to some equivalences induced by the XML based representation (e.g. suppression of spaces between XML elements in a non-mixed content model). The user can also specify additional equivalences that arise from the semantics associated to the document Schema.

communication layer between the services that are interfaced with the system. OMDoc is used as the communication language for the mathematical documents that are communicated on the software bus. The number of systems that have been interfaced with MathWeb is impressive. Among others, we find the Lambda-Clam inductive proof planner, the Omega proof planner, the finite domain enumerator FINDER, the HR system for automated concepts formation, several theorem provers such as Spass and Otter, several CASs such as Maple and finally the MBase knowledge base [69], that is a database for OMDoc documents.

We can easily say that so far the MathWeb system has been the best framework for developing complex distributed systems that exploit the features of several mathematical reasoning tools to solve a complex goal. In particular, the Calculemus project [26] is pursuing the aim of creating a mathematical workbench where any kind of mathematical task could be solved with the help of the machine in a semi-automatic way. The strategy [17] is that of providing an evolving approximation of the workbench by the steady integration of an increasing number of already existing tools, CASs and theorem provers in particular. Even in this case, impressive results of this integration are presented at the annual Calculemus conference.

Since the HELM project is based on the Web services architecture, its integration in Monet should be quite easy. Moreover, the architecture described in Monet is the natural evolution of the MathWeb Software Bus, that predates the introduction of Web services. Moreover, the MKM services provided by HELM are complementary to the reasoning services the Calculemus project is interested in: reasoning services need to locate preliminary lemmas in a distributed library of mathematical knowledge, and they need to render (and maybe publish) their results once they are found. Thus, we expect in the near future to see the ultimate convergence between the field of MKM and the field of mathematical knowledge creation, with mutual benefits on both of them.

Chapter 3

Transformation and Rendering of Formal Mathematical Documents

In the previous chapter we briefly described the distribution model adopted in HELM and the tool that manages the physically distributed library of documents, creating the abstraction of a logically non distributed hierarchical library. Nothing was assumed about the format and content of the library documents[1].

This section attempts a very high level description of the content of the documents of the library, that comprise data (in XML) and descriptions of transformations over data (in XslT). A more precise description of the information stored in the data files will be given in Chap. 6. Metadata documents (in RDF) are also part of the library. The metadata adopted in HELM will be described in Chap. 5.

As we will explain in the next section, mathematical knowledge can be encoded at very different levels. Although the distribution model of HELM allows to manage document at different levels, the HELM team experimented only with documents at the formal level. The choice was motivated by two reasons: the first one is that it is possible to generate the two levels starting from the formal level, although the results are surely sub-optimal. The second motivation is the focus of HELM over Formal Mathematical Knowledge. Notice, however, that in MoWGLI we are currently managing documents at any level of detail using the GETTER.

[1]This is true only in theory. The implementation assumes the documents to be in XML format. The reason is that XML documents refers to DTDs by means of URLs. Moreover, XML instances also use URLs for various tasks (e.g. stylesheets include other stylesheets by means of URLs). Since the distribution model of HELM is based on logical URIs, the XML files in the library must also refer to each other by means of URIs. Before being processed by a standard XML parser or validator, though, the URIs must be transformed to URLs. The solution adopted is to add a boolean parameter to the main method of the GETTER. When the parameter is set, the GETTER replaces all the URIs in the document that is returned to the client with URLs that are invocations to its own main method, passing the URI as a parameter. To apply this transformation, the document must be an XML document, in order to recognize the URIs.

Since we only store in the HELM library the formal level, the HELM and MoWGLI teams developed stylesheets to transform the richer formats into the other ones. These transformations will also be briefly described in this chapter. The aim is not to provide a detailed description (that can be found in the MoWGLI technical reports and in several papers, Ph.D. dissertations and Master dissertations of the HELM team members [113, 88, 62]). Our goal is just to give an idea of the state of the art of the technology available for the implementation of the proof assistant prototype that is the main topic of this dissertation.

3.1 Levels of Encoding of Mathematical Knowledge

As already noticed in the dissertation introduction, mathematical knowledge can be encoded at least at three different levels. The first two levels (from the less semantically rich to the most semantically rich) are *Presentation* and *Content*. The existence of these two levels is pretty obvious: presentation is just the two dimensional visual presentation of a mathematical formula; content is the corresponding abstract syntax tree, where all the symbols have been made unambiguous by fixing their "semantics" (in a rigorous, but not formal sense). The terminology is adopted, for instance, by the MathML standard [79]. Notice that the notational language of mathematics is finite, since it adopts only a relatively small number of symbols and visual notations. This allows the definition of a standard, universally accepted and non-extensible XML format (MathML Presentation, [79]) for writing presentation-level mathematical documents.

On the contrary, a language for content description is much harder to develop. The first reason is that the mathematical language is inherently extensible: new mathematical concepts are created daily, and they must be assigned a content encoding. Another more surprising source of difficulties is the existence of several incompatible definitions of one concept, especially when the definition is subject to arbitrary choices (as the domain of a periodic function, or the limits of a sum) [37]. Finally, one could argue that mathematical knowledge has a very complex structure made of generalizations, instantiations, equivalences and so on. Thus an open catalogue for listing the already standardized mathematical notions cannot be sufficient for the applications.

So far, there have been three proposals for a standard language for mathematical encoding at the content level. The older one, OpenMath [27], defines an XML format for the description of catalogues (called *content dictionaries*). A catalogue is a list of symbol "definitions". A definition is made of an informal description and a list of rigorous mathematical properties. Notice, however, that the rigorous mathematical properties are not expected to completely define the rigorous semantics of the symbol. This task is left to the informal description.

Contributing to a content dictionary and picking up symbols from a content dictionary are two different and non trivial tasks. In both cases, the user must browse the already existing content dictionaries to decide whether a notion already exists. If the notion exists, but the semantics is slightly incompatible, she must decide whether a new symbol must be introduced or not. In the former case, the usefulness of the content dictionary is reduced, since we will end up with too many alternative definitions and no trivial way to move from one definition to the other. In the latter case, the user has to adjust her development (or tool) to understand the new semantics. Often this is not feasible or even possible.

The problem we just described is made more complicated from the existence of generalizations and instantiations. For instance, the plus operator in one of the first content dictionaries developed by OpenMath [8] was meant to be any commutative operator (i.e. the operation of an abelian semigroup). Users of the content dictionary adopted it for the plus operation over numbers, that is just one instance of the plus operation. Since the symbol is now used with this semantics, the original description is no longer accurate and a phrasebook must decide how to interpret it.

Additional problems arise when two content dictionaries independently developed are merged together, forcing the merging of the two ontologies.

We have said that the semantics associated to a content dictionary is rigorous, but not formal. Indeed, content dictionaries are never accessed by OpenMath applications! Any OpenMath application has a module, called *phrasebook* that is responsible of parsing and pretty printing OpenMath documents by translating the document to its own internal representation (that implicitly fixes a formal semantics). In general, the mapping is not accurate: the phrasebook just attempts to best fit the data, raising errors if no reasonable interpretation exists or if the symbol is utterly unkown (which is often the case since OpenMath is an extensible language). The only concrete usage of a content dictionary is as a reference guide for the implementors of phrasebooks, that must decide how each symbol in the content dictionary is interpreted in the system.

MathML Content [79] was initially developed as a non-extensible content language for mathematics with the aim of capturing only K12 mathematics. Since even the notion of K12 mathematics is quite fuzzy, especially when the USA school system is compared to that of european countries, an extension element was added to MathML Content. Thus, nowadays MathML Content is equivalent to OpenMath, with the only difference that there is no notion of content dictionary. This is not a serious limitation since MathML Content allows to use any external description of the semantics of a symbol, comprising OpenMath Content Dictionaries or third party formal descriptions. Moreover, since content dictionaries are never used by applications, there is no real limitation at all. The number of MathML Content aware applications developed is nevertheless smaller than the number of tools able to interpret OpenMath.

The third "standard", OMDoc [97], is at the same time an extension of OpenMath and a proposed replacement for OpenMath content dictionaries. Whereas OpenMath only deals with formulae, OMDoc is meant to encode at the content level whole mathematical proofs, comprising proofs and whole mathematical theories. In particular, the OMDoc language allows to express the notion of generalization and instantiation for whole mathematical theories, that are basically seen as a restricted form of modules, that are not first class citizens [13].

The lack of a formal semantics for the documents described at the content level is at the same time a serious limitation and a source of opportunities. It is a serious limitation since several interesting formal operations, such as proof-checking, missing information reconstruction and so on, are not possible. These are exactly the kinds of operation that Formal Mathematical Knowledge Management — and thus our thesis — is concerned with. At the same time, it is the appropriate level for communication of rigorous mathematical information between tools that are not based on the same formal semantics. Indeed, although the communication can only be approximated due to the rigorous semantics, several systems that actually communicate with each other have been developed in the OpenMath and Monet [87] projects and in the Calculemus [26] network. The communication achieved is really effective, in the following sense: given a formal concept f in a system that encodes a rigorous concept x, if f is transmitted to another system becoming f', than f' corresponds to the natural encoding of x in the second system. Notice that this may be not at all the case when we formally provide an encoding of one logic into another one. In this case, if $[\![\_]\!]$ is the encoding, the translation $[\![f]\!]$ of f is generally different from f'. For instance, if we consider the encoding of set theory into type theory presented in [2, 119], then any "natural number" in set theory enjoys several properties that are not consequences of the Peano axioms. Indeed, the properties are derived from the fact that any natural number is a set, that also enjoys set-theoretical properties. Since the translation $[\![\_]\!]$ respects provability, all these properties must be translated. As a consequence, $[\![f]\!]$ cannot be the natural representation of x in type theory, that is the Peano axiomatization of natural numbers.

Thus, we are forced to introduce at least a third level of encoding of mathematical knowledge, that we call *the formal level* (also called *semantics level* in some papers of the HELM team). Phrasebooks can be seen as a way to translate between the content level and the formal level. Notice, however, that we want to be able to communicate information at the formal level between applications based on the same logic. This implies that, at least in this dissertation, we are actually interested only in those formal level languages that are encodings of a formal and theoretically well-understood calculus. Since we assume the existence of a formal semantics for the calculus and since every calculus is essentially finitary, we are not obliged to adopt an extensible language like OpenMath. On the contrary, we can just develop our own XML dialect that just encodes the syntax of the calculus as briefly as possible. Notice that having a pletora of formal level languages does not hinder in any essential way software reusal. Indeed, introducing a new XML dialect no standard parser (as the one for OpenMath) can be adopted, but also no phrasebook is needed, with a significant overall gain.

To summarize, mathematical knowledge naturally exists at different levels. Each level is justified by the different set of operations that it supports: rendering and interaction for the presentation level; communication between tools based on different logics for the content level; communication between tools based on the same logic and logical operations such as proof checking for the formal level. Moreover, there exists no redundant level, since it is not easy to automatically produce an optimal representation of one level starting from another one. For instance, choosing the layout of the presentation associated to a formula is a complex task, and no automatic algorithm that fully satisfies the human taste exists.

Notwithstanding the latter observation, if we are ready to accept poor approximations for the less semantically rich levels, we can implement automatic tools that translate a semantically richer level into a poorer one. Indeed the HELM and MoWGLI projects provide XslT stylesheets to perform transformations from the formal level (actually only CIC and the logic of NuPrL) to the content level (OMDoc for the proofs, MathML Content for the formulae), and from the content level to the presentational level (MathML Presentation or HtML). The HELM team has also developed a Web service, called UWOBO, that is an XslT stylesheets manager.

In the next section we describe our motivations for the adoption of XslT, and we briefly presents UWOBO.

3.2 XslT and UWOBO

XslT [123] is a W3C standard for the processing of XML documents. Concretely, XslT defines a purely functional language specialized in the processing of DOM trees [42]. A program written in XslT is called a stylesheet. A stylesheet is made of a set of templates. Each template is made of a pattern over a DOM tree and a body that is executed when the pattern is matched. The body is a functional expression that returns a DOM tree fragment. To compute the result, the body can access the input tree and it can perform recursive calls over the input. Moreover, templates have parameters that can be instantiated during the recursive call.

With respect to a standard functional programming language, XslT 1.0 suffers from an heavy limitation: the output of a recursive call, that is a DOM tree fragment (called Result Tree Fragment in the recommendation), can only be included verbatim in the result tree or, in alternative, its string serialization can be computed and reused normally. Roughly, this means that the only result type of non tail recursive functions is the one of strings. Since any data type can be encoded into a string, the limitation does not prevent XslT from being Turing complete. Concretely, though, only simple computations can be performed in XslT.

The limitation is going to be lifted in the next version of XslT, and the library of the language, that provides now only minimal functionalities, will be extended. In a sense, though, the current status of the language is strongly characterized by the above limitation and the lack of a complex and extensible library: the only transformations that can easily be described are those that implement just reorganization of the input information and a minimal amount of computation. These are the only transformations that we expect to have to perform to render a document: if more complex transformations are required, then we can no longer speak of just rendering since we are really creating new information.

This observation is the main justification for adopting XslT technologies in HELM and MoWGLI: since we are willing to implement a standard rendering tool that can be reused as a modular component in the implementation of interactive reasoning tools and MKM tools, we must restrict ourselves to simple rendering transformations that do not involve any logical or domain specific computation.

In spite of its limitations, XslT is already a handy high level language for describing XML transformations, thanks to the complex notion of patterns [122]. Moreover, despite the fact that the small standard library is not extensible without loosing portability, there exists a single function that alone grants extensibility by means of communication with external tools: the `document()` function.

The `document()` function has a parameter that is the URL of an XML document and returns the DOM tree obtained by parsing the document. It is the only function whose output is not subject to the serious limitation we already described. Thus, the DOM tree returned can be processed exactly as the DOM tree that is the main input of the function. The URL can be any HTTP URL, and the document associated to it is retrieved using a full fledged HTTP client. Thus it is possible to pass the URL of a dynamic document, generated, for instance, by a CGI. In particular, it is possible to invoke the methods of a Web service that has a GET binding (i.e. that exposes the interface of a CGI that recognizes HTTP GET invocations).

Thus a stylesheet can invoke the methods of a Web service, as the GETTER described in the previous section. In 2000 Luca Padovani and the author of this dissertation realized that the creation of an XslT interpreter with a Web service interface would have made possible unrestricted XslT recursive calls. The interpreter was implemented by Luca Padovani while visiting the University of Western Ontario, that was also interested in the tool. The Web service obtained was called UWOBO and since its creation it has been a fundamental component of the architecture of HELM. Indeed, any Web service S that needs to produce a mathematical output, such as the SEARCH-ENGINE described in Chap. 5, can simply be composed with UWOBO to provide the functionality: instead of invoking directly S, we can just invoke UWOBO asking the application of a stylesheet to the result of the invocation of S with the appropriate parameters.

To be more precise, UWOBO is not only a simple interpreter. It is a stylesheet manager that provides, among others, the following methods:

- A method whose input is the URL of a stylesheet and a key that downloads and pre-compiles the stylesheet, associating it to the key. Notice that the URL can be a recursive invocation of UWOBO itself. In this way it is possible to use meta-stylesheets, stylesheets whose output is a stylesheet.

- A method to update a stylesheet by downloading and pre-compiling it again.

- A method to drop a stylesheet and its key.

- A method whose input is the URL of a document, a list of keys and a list of triplets key/parameter name/parameter value. UWOBO downloads the document and applies to it all the pre-compiled stylesheets associated to the keys, in sequence. The formal parameters of the stylesheets are instantiated with the actual parameters extracted from the triplets.

Our proof assistant prototype described in the second part of this dissertation adopts UWOBO and the stylesheets developed in MoWGLI for rendering proofs and sequents. Notice, however, that for performance reasons UWOBO is not contacted by means of HTTP calls: the code of UWOBO is directly linked to the code of the proof assistants, and method invocations are performed by means of inexpensive function calls. This is made possible since the proof assistant and the latest implementation of UWOBO, developed by Stefano Zacchiroli [124], are both implemented in the same language (OCaml). Notice, however, that although the implementation of UWOBO is based on the most efficient XslT interpreter available, the rendering of a whole proof can require several seconds. This is not much an issue in the case of on-line browsing, e.g. in the HELM/MoWGLI interface. On the contrary, few seconds can already be annoying in the case of interactive proof development. Thus Prof. Asperti is now working on the direct implementation of stylesheets in OCaml. The OCaml implementation is a couple of order of magnitudes faster. In the near future we may have optimized compilers for XslT (that is now interpreted) and differential stylesheets, that are able to reuse the result of previous applications on the parts of the input that have not been changed. Thus we hope to be able to get rid of the OCaml implementation, in order to avoid the efforts of maintaining two different implementations of the transformations. A different alternative can be generating both the OCaml implementation and the XslT implementation from a common abstract description of the stylesheets.

3.3 Notational Documents

Several stylesheets have been developed in HELM and MoWGLI. They cover both presentational transformations (from OMDoc and MathML Content to MathML Presentation or HtML) and transformations from the formal level to the content level. They are pretty large and complex, since they perform rendering operations that sometimes involve non trivial computations. The most complex operation performed, for instance, is the reconstruction of the proof steps from the proof objects. When the formal level deals with proof objects that are λ-terms, the reconstruction operation must synthetize or integrate missing information (the conclusion of each proof step), it must recognize patterns in the λ-term that encode the proof steps and it must also identify the threads of reasoning of the proof, discriminating the main thread of reasoning from the auxiliary subproofs.

Another much simpler example of complex task performed by the stylesheets is the attempt at automatic line breaking of mathematical formula. There is a stylesheet that is able to efficiently provide an extimation of the size of a formula, that is used to decide whether a line must be broken or not. Moreover, they are structured in order to be easily extensible, since they must reflect the extensibility of the content level: any time a new concept is defined, both classes of stylesheets must be updated to recognize the concept in the formal encoding and to render the concept at the presentation level.

We will not attempt to describe the architecture of the stylesheets in this dissertation, since it plays no role for the sake of our thesis, that just assumes the existence of a generic tool for re-mathematizing formal mathematics into content level mathematics and ageneric tool for rendering content level mathematics in the usual two-dimensional mathematical notation, by translating it into a rendering language such as MathML Presentation. The reader interested in the details can consult the MoWGLI deliverables [88]. In this section we briefly present just one of the problems that were faced, together with its solution, since it will become relevant in the second part of this dissertation.

The problem we want to describe is related to the difficulty of introducing notation for a new mathematical symbol. Indeed, one of the motivations for adopting XslT is the fact that XslT is a standard language whose knowledge is widespread among young computer scientists, since it is usually presented in university courses together with XML. Thus we can assume the user to be able to extend our stylesheets, as soon as we organize them clearly, separating the core functionalities of the stylesheets, that are pretty complex, from the parts that deal with notation and that are the only one that must be extended.

Letting the user manipulate the stylesheets allows complete freedom in the implementation of complex notations and, more interestingly, in the process of re-mathematization. At the same time, the amount of coding required to introduce notation even for a simple, binary infix operator is not negligible:

- From the formal level to content: we need to match the application of a constant to two arguments, generating the corresponding MathML Content. We must also decide how to deal with partial applications of the constant and with the case where the constant is not applied at all, generating the appropriate MathML Content.
- From content to presentation: we must find a pattern for the application of the constant in MathML Content; we must write a template that extimates whether the rendering of a tree that matches the pattern will exceed a given threshold[2]; finally we must write a couple of templates, based on the previous pattern, that generate the MathML Presentation in the two cases of a flat layout and of a line break. Line breaking is actually implemented by means of a table with just one column. Notice also that, according to the precedence of the operator and that of the root of the two trees that are the subterms, parentheses may need to be inserted around the arguments.

Of course, all the work above can easily be implemented by cut-and-paste from the templates relative to another binary operator. Due to the usual cut-and-paste drawbacks, though, a better solution consists in either writing a generic and abstract set of templates for binary operators, or in generating the stylesheets from a specification of the operator properties (i.e. associativity, position of the operator, arity, etc.).

A master student, Pietro Di Lena [41], studied the operators of the libraries of Coq, identifying several classes of operators and developing a DTD for the description of the operators that belong to one class. He also developed meta-stylesheets for the automatic generation of all the templates discussed above. The documents that are valid against the DTD are called *notational documents*.

Among the classes of operators identified, there are first order operators and "regular" higher order operators in higher-order abstract syntax, whose arguments are expected to start with a λ-abstraction. The meta-stylesheets also implement η-expansion in the case of a missing λ-abstraction. Actually, the notational documents are implemented as a stack of languages, from high-level to low-level, the lowest level language being XslT itself. Meta-stylesheets exists between each pair of consecutive languages. The idea is that the user will pick the highest level language that allows the implementation of the notation she is interested in. Experiments on the library of Coq show that the largest part of the operators can be described in one of the high level languages, although there exist numerous cases of operators that need an XslT implementation and several other cases that do not even allow an XslT implementation. For instance, in the library

[2]Due to technical reasons that derive from XslT being a purely functional programming language with limitations over recursive calls, the most efficient implementation of the check consists in a first child – following sibling visit.

we can found the definition of a semigroup and the definition of a ring. We could easily associate a generic notation to the semigroup operator; the solution would not be correct, though. Indeed, the additive operation and the multiplicative operation of the ring need two different notations, and the notations should be propagated to all the lemmas and theorems that describe properties of the semigroup. In other words, the notation for lemmas and theorems that are abstracted over a mathematical notion should be abstracted over the notation for the abstracted notion. An implementation in XslT only seems unfeasible at first sight and has not been investigated yet.

It is important to notice that, although the approach seems effective for notation of mathematical formulae, it does not help for applying notation to proof steps, that is a much less regular task.

Several authors have already developed notational documents. Among them, Kohlhase introduced notational specifications for OMDoc documents ([97], Sect. 3.5.2) and the development team of PCoq introduced notational specifications for Coq operators, together with a nice interface for selecting the notation. The Alfa system also has a similar interface. Some of the parameters that can be specified in the notational documents (e.g. arity and associativity) are common to all these systems. These are usually the parameters that are related to the transformation from the content level to the presentation level. On the contrary, we find several differences in the parameters that determine the transformation from the formal level to the content level, that heavily depend on the way mathematical operators are formalized in the different underlying logics. For instance, polymorphic binary operators, such as the equality, are formalized in logic with dependent types as ternary operators whose first argument is the type of the other two arguments. When the operator is translated into its content level representation, the first argument must be dropped. Thus we need to introduce in the notational language an operator that describe which arguments are to be ignored.

Thus, further work will be required to find a common format for notational documents. In the meantime, in HELM we plan to investigate the possibility of reusing notational documents to parse mathematical expressions (at the presention level) and convert them into richer levels (i.e. content or formal levels). Sect. 10.4 is devoted to the topic of parsing ambiguous mathematical expressions. We present an efficient solution that is implemented using two independent components, a parser and a disambiguator. Our plan is to generate the parser from the notational documents.

3.4 Comparisons with Related Work

OpenMath and MathML Content both introduced the two levels called Presentation and Content. Neither one of the two standards identified the third level, that we call Formal Level. The historical motivations are pretty clear: the majority of the developers of the two standards belonged to the community of Computer Algebra Systems (CAS). Indeed, the main motivation for introducing a content level was to allow communication between CAS. Since almost every CAS is based on a very weak logic, essentially unisorted first order logic, and since CAS only rely on a minimal way on the logic foundation, it is not surprising that no foundational issue was addressed and that no formal level was identified. The same motivation holds when we consider not only CAS, but also automatic theorem provers: since a theorem prover must efficiently discover a proof without human intervention, the logic these systems are based on is chosen to be as simple as possible, trying to minimize its complexity and the number of connectives involved. Thus the translation from one logic to another is still pretty easy, and the actual logic adopted has no major consequences over the semantics of the objects defined in it.

The HELM team pushed the idea of the formal level starting in the year 2000. At first and for a long period, the formal level was rejected by the OpenMath community. Essentially, two criticisms were made. The more phylosophical one states that foundational issues are really just sorts of implementation details that arise from the formalization of mathematics. Thus mathematical notions are invariant w.r.t. the logic adopted, and they can be described independently of the logic foundation. In principle, it should always be possible to write a phrasebook that is able to "formalize" a document in a logic and "re-mathematize" the document. Improvements in proof theory will make the task more feasible by progressively closing the gap between real world mathematics and formalized mathematics.

From our perspective, the criticism just says that the real *mathematical content*, that is the one that really matters to real world mathematicians, is the content level. Although the observation is probably true, at least from the point of view of a mathematician, it does not imply in any reasonable sense that the formal level does not have any reason to exist. Indeed, it completely misses the point of the existence of the formal level: the formal level is justified by the logical operations that we can only perform when the information is encoded in a logical framework. If we could perform the same operations at the content level, comprising proof checking, we would just drop the formal level, and any mathematical foundation as well. Moreover, we could argue that the gap between rigorous mathematics and formal mathematics will never be closed: closing it would really not mean improving proof theory; it would mean making rigorous mathematics loose its peculiar and very appealing flavour, made of flexibility but also sloppiness, in order to make its semantics rigid. For instance, in order to allow proof checking, we will probably be obliged to drop all those beautiful graphical proofs that convince the reader by appealing to his spatial sense.

The second criticism is more pragmatic: since OpenMath and MathML Content are extensible languages, we should just adopt them also for the formal level. Concretely, the criticism consists in the suggestion of developing OpenMath-based exportation modules that map the constants defined in the logic of the proof assistant to a content dictionary. The implementor of the module must decide, for each symbol, whether it is better to reuse a symbol already defined in a content dictionary or whether it is better to introduce a new symbol, for instance because the logic imposes a semantics that is not the standard one.

There are two objections to the second criticism. The first one is that a formal system is, by its own nature, finitary. Thus an exportation module should be a finite object that maps the entities of the formal systems and not the notions that are defined in it. To draw a parallel, it is like writing a pretty-printer for the statements of a programming language, and not a pretty-printer for each program definable in the language. Thus we would be obliged to write an extensible exportation module and to adopt an extensible language for communication with other tools, even those based on the same logic. Concretely, this would mean that it would no longer be possible to write document centric proof checkers, since a document could be parsed only by a tool whose phrasebook understands at least the set of symbols in the document. Alternatively, we are obliged to think that a proof checker is no longer characterized only by the logic it understands, but also by its phrasebook.

The second objection is about the pollution of the content dictionaries space. Indeed, if we ignore the first objection and we try to write a phrasebook[3], we will face for each constant the choice of reusing a symbol or creating a new one. Constants inherit some of their properties from the system they are defined in. For instance, in the Calculus of (Co)Inductive Constructions it is possible to give two different definitions of the relation $\leq$ over natural numbers. The first one has a computational meaning, in the sense that $n \leq m$ for closed terms n and m is convertible to either the proposition `True` or the proposition `False`. The second one has no computational meaning (and there exist models where we can prove that its interpretation is not equal to the interpretation of `True` or the interpretation of `False`). As a consequence, the two definitions enjoy different (meta theoretical) properties, and we cannot simply substitute one with the other without breaking well-formedness. Should we map them to the same symbol in an OpenMath content dictionary or should we create two different symbols? In the former case parsing a document to obtain a well-formed context becomes non deterministic and ambiguous (that is perfectly ok if we consider a transformation from the content level to the formal level, but it is not if we just think of parsing a document). In the latter case we are polluting the content level by introducing details that belong only to the formalization, and we are also hindering communication with other tools (that must either create a new symbol or pick one of the two, even if the meta theoretical property that distinguish them has no sense in their logic).

The HELM and MoWGLI team are still defending their position, keeping the distinction be-

[3]We basically made the attempt together with M. Kohlhase during one of his visits to Bologna.

tween the content and the formal level. Progressively other researchers are recognizing the importance of the difference (see, for instance, [3]).

Adopting stylesheets for rendering content documents into presentational languages is the most natural option, since content documents are stored in XML formats and since the complexity of the task is manageable with XslT, that is the W3C proposed standard for describing rendering transformations. Thus several other groups (e.g. ActiveMath, OpenMath) have experimented with the idea. HELM is the only project that also attempts in XslT more complex transformations, such as rendering formal proofs in natural language. The results obtained by the other groups are comparable to those obtained in HELM.

Although the idea of using the `document()` function to make a stylesheet invoke a Web service is very simple, we were the first to notice it and to develop a stylesheet manager to allow recursive calls. The developers that have faced our difficulties have solved the problem by dropping portability, implementing ad hoc extensions to the language.

Some of the problems that motivated UWOBO will disappear in the next version of XslT. Notwithstanding this, our experience shows that a stylesheet manager is a fundamental component of an architecture for MKM based on Web services, since UWOBO can be dynamically composed with other Web services to provide nice rendering functionalities and XML filtering, much in the spirit of Unix pipes.

Chapter 4

Tracking Dependencies

An important but too often ignored problem in MKM is the possibility to inspect a mathematical development at the logical macro level.

Mathematical documents are usually structured like traditional documents, divided into parts, chapters, sections and so on. This sequential structure is optimized for conveying a message, or for teaching a subject. The rethorical textual parts play a fundamental role in understanding the document, as well as the precise order chosen by the author.

Web publishing technologies allow to exploit all kinds of information in a non sequential way, for instance by means of hypertexts. Digital learning environments, like ActiveMath [83], are trying to stress this idea, in order to assemble on demand a mathematical document. Briefly, the scenario proposed is that of a student or a researcher with his own proper knowledge and skills that is willing to learn a new mathematical field. The student describes to the system what he knows and how well he knows it[1]. Then the system reconstructs the set of preliminary notions that the student should master to understand the new concept. After removing the notions that the student already knows well, the remaining notions are serialized according to their definitional order. Then definitions, lemmas and exercises are proposed to the user by means of the creation of a new mathematical document obtained by adding rethorical text and macro structure to the list of selected definitions, lemmas and exercises. Additional information about the student skills and interests (i.e. if she is a researcher, an undergraduate student, a mathematician, a physicist, etc.) are used to select the amount and difficulty of the exercises. In the same way the proofs of the lemmas and theorems can be shown only to students that are skilled and interested in the field.

Notice that, in this scenario, the traditional document structure (made of chapters, sections, etc.) is seen as just a decoration over a linearization of the definitional order. With definitional order we mean the order induced by the occurrence of a lemma or definition in the statement or

[1] Even better, the system remembers the user model, i.e. a description of the knowledge of the student, and it updates the user model automatically by inspecting the answers to automatically generated tests and exercises.

proof of another lemma, or in another definition. Due to logical consistency, the definitional order can be described by an acyclic graph, that can be serialized in several different ways. We call the graph the *logical macro structure*, since it is induced only by the formal notion of occurrence.

The dependency graph is not only useful in electronic learning. Indeed, it provides several informations about the relevance of a result, its complexity (in terms of axioms and basic definitions it relies on), and the consequences of changing a definition.

For instance, tools for inspecting the dependency graph are often demanded by the users of proof assistants. Indeed, in more than one occasion several people have asked to us during a meeting or by e-mail whether there exists a way to answer the following question for the library of the Coq proof assistant:

- *What part of the library is based on classical logic?*
  Notice that the answer can be surprising (and sometimes it is, since there exist lemmas that are provable in the constructive fragment but that have been proven classically). To develop a completely non classical result, I need to use only lemmas that admit a constructive proof. How can I know whether a theorem in the library has a constructive proof if I cannot inspect the dependency graph[2]?

- *I have to change a definition. What is the part of the library that I will need to change? What if I change the definition, but I am still able to prove some lemmas?*

- *What is the logical complexity of a result?*

In her Ph.D. thesis, Irene Schena describes an RDF [106] metadata schema for mathematical "objects". The most part of the metadata proposed are logic independent and can be applied to rigorous mathematics as well as to formal mathematics. Among them, there are the metadata that describe what are the occurrences of a definition or lemma in another lemma or definition.

To test the metadata proposed, the HELM team wrote a tool to compute and save in RDF format the metadata over the distributed library of XML documents exported from the Coq proof assistant. The RDF documents (called models in RDF terminology) are valid with respect to the schema developed by Dr. Schena.

A few months later, the author of this dissertation developed a Web service to render subsets of the dependency graph visually. The Web service directly interprets the RDF format and uses the

[2]Actually, in the Coq system there exists a syntactical distinction over computational objects and non computational objects. Since in the general case excluded middle has no effectively computable computational content, the axiom is admitted only over non computational propositions. Thus the proof of every statement in the computational fragment is constructive. The question, though, is still actual, for two reasons. First of all a proof of a computational statement can have a subproof that has no computational content. We have no way to know whether the subproof itself is classical or not, if that makes sense to us. Secondly and more generally, the question is left open in the non computational fragment of the system.

GETTER to retrieve the needed RDF models. Thus the tool is not limited to the distributed library of documents extracted from Coq: it works on any set of metadata relative to any mathematical library. The Web service is called DEPENDENCIES GRAPH GENERATOR.

During his Master Thesis, Stefano Zacchiroli reimplemented and polished all the Web services of HELM, including the DEPENDENCIES GRAPH GENERATOR. The new implementation, heavily based on the original one, can be queried on line by means of the Web interface to the HELM library [63]. The next section briefly describes the DEPENDENCIES GRAPH GENERATOR Web service, its interface and the functionalities it provides.

4.1 The DEPENDENCIES GRAPH GENERATOR

The DEPENDENCIES GRAPH GENERATOR is a Web service implemented as an XslT stylesheet that is managed by UWOBO and that operates by invoking other Web services developed in HELM. Thus, the real Web service is UWOBO, as described in Chap. 3, that has a method to apply a stylesheet to a given document and a given set of actual parameters.

In the DEPENDENCIES GRAPH GENERATOR, the input document is always an "empty" XML document (i.e. an XML document formed by just one element). The output is an HtML page that shows an interactive graph that is a subset of the global acyclic graph of dependencies where the minimum (or maximum) is specified by the user and that is the object whose dependencies the user want to analyze. To avoid showing graphs with too many arcs and intersections, only the core of the transitive relation is shown, i.e. only arcs that correspond to direct dependencies are kept. Moreover, to avoid graphs that are too big to be understandable, only the first layers of the graph are shown.

The actual parameters that are taken by the stylesheet are:

1. The URI of an object O.

2. A boolean parameter that tells the stylesheet whether the user is interested in the objects that directly and indirectly depend on O, or in those that O depends on.

3. The maximum number n of objects in the output graph. The graph is generated layer by layer. When the number of nodes in the output graph reaches n, the visit stops and the nodes whose children have not been visited are highlighted to distinguish them from the leaves of the graph.

The output is implemented as an image map that shows the graph whose nodes are the URIs of the objects and whose arcs represent direct dependencies over them. By clicking over a node, a JavaScript menu is activated. The menu allows several operations. Among that:

- jumping directly to the HELM page of the selected object
- repeating the query with the selected object as the new root
- performing the query with the selected object as the new root, but changing the direction of the dependencies

The last two options allow to "pane" the current view of the graph to have a global understanding. In particular, it is often useful to repeat the query over the red nodes of the graph, those whose children were not visited. Visually, this corresponds to moving the focus over the edges of the graph, in order to show the lower layers.

We already said that the stylesheet that implements the DEPENDENCIES GRAPH GENERATOR invokes other Web services. The invocation is implemented as a call to the `document()` function, passing the URL of the Web service that must be invoked. According to the `GET` method of the HTTP protocol, the method to invoke and the actual parameters are encoded in the URL.

The services invoked are:

- The GETTER. The getter is responsible of resolving the URIs of the metadata associated to an object into a physical URL; downloading the file located by the URL; caching the file and returning it to the client. Thus the metadata models can be physically distributed as well as the data they refer to.

  It is also possible to store all the metadata in a relational or RDF database, replacing the implementation of the GETTER with another implementation that retrieves the metadata from the database and produces a valid RDF model. Such an implementation has been developed by Dr. Luca Padovani.

- An unnamed Web service that manages non standard bounded priority queues. At first glance, XslT seems the appropriate language to implement the DEPENDENCIES GRAPH GENERATOR since the tool has to parse XML documents (the RDF models) and it has to produce valid XML files (see next point). Unfortunately, XslT has severe limitations in the output type of the functions that make the implementation of some algorithms unreasonably complex. In particular, to implement the visit by layers, we needed to implement a queue of URIs. We decided to implement the queue in a different programming language[3] and provide a Web service interface to it. The queue implementation is reentrant, since it can handle several queues at a time, one for each concurrent request. Moreover, when the queue is created, the total number of elements to go through the queue can be set. The behaviour of the bounded queue is not standard, since insert operations are no longer allowed once the number of insertions performed reaches the maximum allowed number of insertions. Removal operations, instead, are always allowed.

[3] That was Perl in the original implementation. The current implementation has been written in OCaml.

- Finally, the last Web service is responsible for parsing an XML file that textually describes the graph and producing a pretty printed image map. The current implementation is based on the Graphviz tool[4]. As soon as the support for SVG in Graphviz will be completed, we plan to move from image maps to SVG.

The DEPENDENCIES GRAPH GENERATOR is a nice example that shows that Web services are compositional. Notice also that all the Web services adopted but the one that manages bounded queues were already available since they are reused by several components of the HELM architecture.

4.2 Conclusions and Related Works

The Web service described in this chapter is a simple tool that provides answers to common problems that are of direct interest to the users of the current proof assistants. It is generic, in the sense that its input are metadata models that are valid against a schema that works both for formal mathematics and for rigorous mathematics. We like to consider it a simple prototype to investigate the acceptance by the user base, and to identify a set of features that the users expect.

The prototype could and probably should be extended in at least two ways. The first one deals with a concrete UI problem. The possibility of "paning" over the graph is often sufficient, but globally understanding a complex set of dependencies require time and non neglectable efforts. A much more advanced solution to understand a huge graph is constituted by the hyperbolic viewers, that show a graph all at once, but magnifying just one part of it to understand the local details. We do not intend to spend any time on this improvement. Indeed, as soon as there will be available a Web service to produce SVG graphs, it will become possible to adopt a standard hyperbolic viewer. This is one of the nice features of standard technology: sometimes you just have to sit down and wait for the new release.

The second improvement has deeper motivations. Indeed, the view of a mathematical library as an acyclic graph works for formal mathematics, but it is too simple for rigorous mathematics. Indeed, in mathematics there exist several sets of equivalent axioms or definitions. Mathematicians are used to switch from one set to an equivalent one without even noticing. These equivalences should be representable in the logical macro structure of the development. Notice that the lack of these equivalences in formal theories is annoying, and there exist several efforts [40] to embed some of the equivalences in the calculi that underlay the implementation of several proof assistants. In Sect. 8.3 we will introduce an extension of an existing calculus that easily allows to "simulate" switching from one set of axioms to an equivalent set of axioms. The solution does not work for definitions. Indeed, in the case of definitions the problem is much harder [77]

[4] `http://www.research.att.com/sw/tools/graphviz/`

(and still unsolved), since logically equivalent definitions can have a very different computational behaviour.

A completely different approach, inspired by the module systems of the ML-like languages, is currently being proposed by the Logical team [31]. The idea is to develop a mathematical library as a set of functors, that are modules parameterized over other modules. The input of a functor axiomatically describes the set of axioms and definitions the functor is based on. This approach is promising and it may be effective, but it has two major drawbacks. The first one is that the computational properties of the objects whose existence is assumed in the input module are not specifiable. As a consequence, computation is no longer exploited during proof checking. Allowing the specification of the computational properties means accepting term rewriting systems as functor inputs. The consequence is the integration of term rewriting systems engines and checkers into the kernel of the proof assistants. This is a complex task, but it is already supported by several theoretical investigations [31, 22]. We expect such an integration to be concretely available in main stream tools and its usage to be well understood only in a few years. The second problem is that functors do not fit well with the traditional approach of real world mathematicians. Indeed, a MKM library for a real world mathematician is more naturally seen as an acyclic graph extended with equivalence arcs than as a set of functors.

Chapter 5

Indexing and Retrieval

In the previous chapters we already briefly described the distribution model of HELM and the transformation and rendering technologies of HELM and MoWGLI. We also described the Web service associated to each task: the GETTER to create the abstraction of a logical library on top of a physically distributed library; and UWOBO that is an XslT stylesheets manager invoked by other Web services to change the level of representation of a piece of mathematics, eventually rendering it in a suitable format for the final user. We also presented another Web service, the DEPENDENCIES GRAPH GENERATOR, that can be used to analyze the structure of the distributed library that is induced by the notion of occurrence. We did not yet address the fundamental topic of indexing and retrieving information inside the distributed library, although we already explained that, according to the HELM requirements, we expect to base the indexing techniques on RDF technology [106].

The need for indexing and retrieval tools for both rigorous and formal mathematics is absolutely evident. Indeed, as Jorge Luis Borges pointed out in his beautiful novel "The Library of Babel" [23], a huge library is completely useless unless we have effective methods to retrieve what we are looking for. Unfortunately, the largest part of the mathematical documents available over the Web are Postscripts or PDF documents. Thus, the only reasonable way to look for a given mathematical definition or result is to rely on textual matching or to perform queries over the provided metadata (when available). In particular, the organizations of librarians developed several useful classification schemes to index mathematical textbooks and papers [80, 39, 74]. This approach can be effective enough when we are looking for a general work on a given mathematical topic; but it cannot be used to retrieve a single theorem or definition over the Web, especially when the theorem is not fundamental enough to be given a well-known name.

The development of Web services for automated deduction and computation put high pressure for the development of alternative effective methods to retrieve a single result. Luckily, at the same time more and more mathematical knowledge is put on the Web in a structured form, thanks to the introduction of standard markup languages [27, 79, 97] for the encoding of mathematical formulae and expressions. Once the statements in a library are marked up, say, in OpenMath, we can use standard pattern-matching or unification techniques to retrieve a theorem given a part of its statement. For example, the pattern $\forall x : ?_1 .\ \forall y : ?_1 .\ (?_2\ x\ y) = (?_2\ y\ x)$ will retrieve any theorem stating the commutative property of a binary operation. The question marks are *metavariables*, which are non-linear placeholders for any (well-typed, if this notion can be enforced) sub-expression closed in the context.

To answer a query based on pattern matching, the trivial approach consists in iterating the matching operation over the whole library. This solution does not scale well and it cannot be applied in those cases where the library itself is physically distributed and no code can be run on the servers[1]. This is exactly the scenario we face in HELM.

To solve this problem, the HELM team propose an approach based on four distinct phases:

- *Data-mining*: using a Web spider, we extract a small set of automatically computed metadata from each theorem or definition in the distributed library. These metadata are devised to capture some of the invariants of the chosen matching operation.
- *Pattern compilation:* when the user submits a pattern, we extract a set of constraints over these metadata from the pattern and we generate a low-level query[2] starting from these constraints.
- *Filtering:* we execute the generated query over the database built by the Web spider, obtaining our set of candidates.
- *Matching:* we iterate the matching operation only on the set of candidates.

If the filtering operation is both quick and correct (in the sense that no good candidates are dropped), we are able to achieve both accuracy and performance.

Since the metadata must capture the invariants of matching, we need to identify several kinds of metadata. In Sect. 5.1–5.3 we will progressively consider several examples of matching criteria and we will identify the related metadata. We will also consider the issue of correctness and a trade-off between correctness and performance. In Sect. 5.4 we will present a generic set of metadata and constraints that can subsume all the previous examples. All the work described in these first four sections has been developed in a collaborative way by the author of this dissertation

[1]Indeed, if the library is distributed, we only have two possibilities: either we can perform the matching where the theorem is located or we need to download it. In the second case the download times make the operation unfeasible even for very small libraries. In the first case we need to install software were the library is published, which is something we would rather avoid.

[2]In our prototype, we generated the query in the MathQL Level 1 language [54]. Generating a query in other languages is surely possible.

and Dr. Ferruccio Guidi. The original intuition and ideas are due to Prof. Asperti, that is also the main implementor of the HELM data-mining tool.

In Sect. 5.7 and Sect. 5.8 we present a small formal framework to reason about sets of metadata and constraints. In particular, in Sect. 5.7 we define the framework and we prove the admissibility of the metadata identified in the first use case. Sect. 5.8 is entirely devoted to the admissibility of the generic set of metadata presented in Sect. 5.4. The formal framework has been developed by the author of this dissertation alone.

Finally, Sect. 5.6 and Sect. 5.9 are devoted to comparisons with other work and future developments.

The material presented in this chapter can also be found with only minor modifications in [55].

5.1 Use Case 1: Finding the Applicable Theorems

The first use case arises in the context of backward reasoning proofs: we are interested in knowing which theorems can be applied to prove a given goal by reducing it to an (eventually empty) set of new goals. A goal is a sequent made of several hypotheses and a conclusion. A theorem can be applied if its conclusion, where every bound variable has been replaced with a metavariable, matches the conclusion of the sequent. A new goal is opened for each hypothesis of the theorem: the hypotheses of the new goal are inherited from the initial goal; the conclusion of the new goals is the corresponding hypothesis of the applied theorem.

We restrict ourselves to the case of *first order matching*, i.e. only one expression (the conclusion of the theorem) has metavariables in it; and the expression is rigid[3]. For example:

- The theorem $\forall n.\ \forall m.\ \forall a.\ a \geq 0 \Rightarrow n \leq m \Rightarrow an \leq am$ can be applied to prove $2x \leq 2(x+1)$ since the pattern $?a?n \leq ?a?m$ matches $2x \leq 2(x+1)$.

- The theorem $\forall n.\ \forall m.\ \forall a. n \leq m \Rightarrow an \leq am$ cannot be applied to prove $x \leq 2x$ since the pattern $?a?n \leq ?a?m$ does not match $x \leq 2x$.

- The induction principle over natural numbers $\forall P.\ (P\ 0) \Rightarrow (\forall n.\ (P\ n) \Rightarrow (P\ (n+1))) \Rightarrow \forall n.\ (P n)$ does not match $(\mathrm{is\_prime}\ 5)$ since the pattern $(?P\ ?n)$ is not rigid.

In Sect. 5.7 the interested user can find the absolutely standard formal definition of first order matching that we adopt in this use case.

We identify a set of invariants of the matching procedure that we want to capture as metadata:

[3]The motivations for this second requirement will be explained in the section devoted to the second use case, where we will extend our work to non-rigid patterns.

- The constant in *head position* in the conclusion of the theorem. Since the pattern is rigid, any matching term must have the same head constant as the pattern.
- The list of constants in the conclusion. Since we are doing matching and not unification, any constant in the pattern must also occur in the matching term (since instantiating a metavariable cannot remove an occurring constant).

Thus we identify the following set of metadata: for each theorem in the library we remember the name of the constant in head position in the conclusion and the name of all the constants occurring in the rest of the conclusion. Now, to find the theorems that are candidates for the query "*give me all the theorems that can be applied to this goal*", we simply compute the metadata over the conclusion of the goal and we look for every theorem in the library such that:

- the "constant in head position" metadata is the same as the computed one
- the set of "constant in conclusion" metadata is a subset of the computed one

The following theorem is a trivial consequence of the fact that our metadata reflects the matching invariants:

Theorem 5.1 *The filtering operation is correct, i.e. no pattern that matches the goal conclusion is filtered out.*

The proof of the theorem can be found in Sect. 5.7. Let us see how the filtering operation works in practice: given the goal $2x \leq 2(x+1)$

- $?a?n \leq ?a?m$ is a candidate that will match
- $?n \leq ?n^2$ is not a candidate, since the power operator does not occur in the goal
- $?n?a \leq ?m?a$ is a candidate that will not match (a *false match*)

We could also capture much stronger invariants. For instance, the *rigid skeleton* of the term (formally defined at the end of Sect. 5.7) is preserved by metavariable substitutions. Nevertheless, a stronger invariant may have two drawbacks: the first one is that it may produce complex metadata (e.g. contexts, i.e. terms with holes) that cannot be handled efficiently within the relational database model or the RDF model (which is often reduced to the relational model); the second drawback is that the invariant may be too strong. For instance, remembering the exact position of the occurrences of the metavariables, it could reject the third candidate of the previous example. Even if that candidate turns out to be a false match, it is pretty obvious (at least to a human user) that the statement found is useful to proceed in the proof. Very often, indeed, the user is also interested in theorems that can "almost" be applied, since they can reveal an error in the proof (when a required assumption is missing) or they can suggest a different way to prove the goal.

Of course, the danger of not adopting the stronger invariant is that of finding too many useless false matches. To evaluate our invariants, we tried several queries over the library of the Coq proof assistant that holds about 40.000 theorems. The queries were not artificial: one of the authors used the query engine in order to retrieve some useful lemmas while developing a new medium-sized Coq contribution [109]. Even for queries involving only frequently used notions (e.g. algebraic operations), the accuracy of the filtering phase was always very high: the number of uninteresting false matches was very low and in more than one occasion the SEARCH-ENGINE found false matches that we were actually able to exploit. Thus, the productivity level of the author was definitely raised by the use of the SEARCH-ENGINE.

Let's face now the issue of performance, considering for instance the goal $2x \leq 2(x+1)$ which generates the constraints:

- `≤ in head position in the conclusion`
- `only 2 and + in other positions in the conclusion`

The generated query (in a pseudo-language inspired by MathQL level 1 [54]) is:

```
select every t in the library such that
 t.head = '<=' and t.in_conclusion subset of {'2','+'}
```

Since the second test cannot be performed in a relational DB, we are forced to extract from the DB all the theorems `t` that satisfy the first constraint together with their `in_conclusion` field and then iterate the second test. Thus, the computational cost is linear over the size of the result set of the first test. If the head position of the goal is a frequently used constant, the result set is large. For example, there is an equality in head position in 11% of the theorems of the library of Coq, i.e. in 3660 theorems. To reduce the computational cost of the query, we propose a user-controlled trade-off with the accuracy of the query: the user provides an additional set of constraints consisting in a list of constants that *must* appear in the conclusion of the matched theorems. To avoid confusion, we will call these constraints *must constraints* and we rename the previous set of constraints *only constraints*. In order to retrieve a non-empty subset of results, the set of constants occurring in the *must constraints* must be a subset of the constants occurring in the *only constraints*.

To better understand the *must constraints*, let us consider the goal $2x \leq 2(x+1)$. The generated *only constraints* are $\leq$ in head position and 1,2,$*$ and $+$ in other positions in the conclusion. Let's suppose that the user choice of *must constraints* is $\leq$ in head position and $*$ in other positions. Then

- $?a?n \leq ?a?m$ is a candidate that will match (a true match)

- $?n?a \leq ?m?a$ is a candidate that will not match (an interesting false match)
- $?n \leq ?n + 1$ is not a candidate since it satisfies the *only constraints* but not the *must constraints* ('$*$' does not occur). Anyway, it would have been an uninteresting false match.

Note that the *must constraints* are meaningful to the user: in the previous example, the *must constraints* are stating that the user is interested in a property of multiplication. Thus, it is not difficult to provide an user-friendly interface to select these constraints. It is also easy to provide some heuristics to automatically choose the *must constraints*[4].

A non-empty set of *must constraints* reduces the accuracy of the query, since some theorems that could have been applied are not found any longer. For example, the previous query is unable to find the transitivity principle for the $\leq$ relation, since it states nothing about multiplication[5]. The performance gain achieved can be remarkable. To understand why, let us see the query generated for the last example:

```
let S =
 select every t in the library such that
  t.head = '<=' and '*' occurs in t.in_conclusion
in
 select every t in S such that
  t.in_conclusion subset of {'1','2','+','*'}
```

The first part of the query (the first `select`) can be performed by a single call to the database. Then, as before, we need to iterate the second test over the set returned by the first select. The more *must constraints* we have, the smallest the set `S` will be and the smallest the time required to perform the query will be.

In the next two sections we will try to apply the same approach to another couple of examples, choosing the matching operation we are interested in and deriving the *must* and *only constraints* from it.

5.2 Use Case 2: Finding the Elimination Principles

Let us consider queries for the retrieval of all the elimination principles "à la McBride" [82] over a given datatype. Briefly, these are elimination principles that allow to prove a generic property P over an element n of a datatype T by proving that P holds in several cases, some of

[4] As an example, we provide a predefined heuristic that chooses only the constants occurring in the first n levels of the syntactic tree of the goal thesis.

[5] In our experience, very often loosing these "very general" theorems can be considered a feature.

them under additional inductive hypotheses. For example, there are several elimination principles over lists of natural numbers. Among them, we find the structural induction principle[6] $\forall P.\ (P\ \mathit{empty}) \Rightarrow (\forall l.\ \forall n.\ (P\ l) \Rightarrow (P\ n :: l)) \Rightarrow \forall l.\ (P\ l)$ and the induction principle over ordered lists $\forall P.\ (P\ \mathit{empty}) \Rightarrow (\forall n.\ (P\ [n])) \Rightarrow (\forall l.\ \forall n.\ \forall m.\ n \leq m \Rightarrow (\mathit{ordered}\ m :: l) \Rightarrow (P\ n :: m :: l)) \Rightarrow \forall l.\ (\mathit{ordered}\ l) \Rightarrow (P\ l)$.

Note that, since the pattern $(?P\ ?x)$ is not rigid, every elimination principle can always be used to progress in any proof, independently of the goal. Since there exist literally thousands of elimination principles, it is not a good idea to include them as results of the query of the first use case we considered[7]. Nevertheless, we are interested in giving the user the possibility to retrieve all elimination principles over a given datatype. In particular, given a datatype T, we want to retrieve all the theorems whose statement matches[8] the schema:

$$\forall \overline{x} : \overline{S}.\ \forall P : T \rightarrow \mathrm{Prop}.\ \forall \overline{y} : \overline{S'}.\ \forall x : T.\forall \overline{z} : \overline{S''}.\ (P\ x)$$

We identify the following set of constraints:

- The sort `Prop` (i.e. the type of every proposition) must occur in the head position of an hypothesis which is a product[9] of length 1.
- The datatype must occur in the head position of an hypothesis which is a product of length 0 and also in another hypothesis (not in main position).
- The head of the conclusion must be an occurrence of a bound variable.

Every constraint captures a matching invariant, since context metavariables substitution cannot change the shape of the conclusion nor the shape of an hypothesis (which are all rigid parts). The formal proof for the generalized case that we will present in Sect. 5.4 can be found in Sect. 5.8. As in the previous use case, there exist stricter invariants: the set of constraints identified here has been fine-tuned by hand to be able to retrieve also a few useful false matches. Even in this case, though, the final stable set of invariants was reached quite early and it did not require any major effort.

[6]The ML notation $hd :: tl$ stands for the list whose first element (CAR in Lisp notation) is hd, followed by the rest of the list tl (CDR in Lisp notation).

[7]That is the reason why we constrained ourselves to first order unification in Sect. 5.1.

[8]A formal definition of the kind of match adopted can be found in Sect. 5.8, where it is called *extended first order matching*. Briefly, we still require that all the metavariables are not used as functions by applying them to arguments. Moreover, we add sort of *context metavariables* (that correspond to the "vector parts" of the schema). Intuitively, context metavariables are contexts of formulae that must be instantiated with a context made of repeated quantifications or implications, concluded by a single contextual hole.

[9]The terminology comes from type-theory and is derived by the Curry-Howard isomorphism, where universal quantifications are seen as (dependent) products. A product of length n is a term of the form $\forall x_1.\ \ldots\ \forall x_n.\ t$ where t does not start with an universal quantification. An implication is just a special case of universal quantification: it is a non-dependent quantification where the bound variable does not occur in the sub-expression.

Note that, unlike the previous use case, we derive the matching pattern from the user input and not from the statements of the theorems in the library. In Sect. 5.1 we had just one term to match against several different patterns; here we have just one pattern to match against the theorems in the library. This duality is reflected in the way the constraints are used to generate the query: in the previous case the identified constraints were *only constraints* (only the constants occurring in the goal can occur in the statement of the theorem); in this case the constraints are *must constraints* (all the constants in the pattern must occur in the statement of the theorem)[10].

The query generated from the constraints is the following:

```
select every t in the library such that
 Prop occurs in head position at depth 1 in an hypothesis of t and
 T occurs in head position at depth 0 in an hypothesis of t and
 T occurs not in head position in an hypothesis of t and
 a bound variable occurs in head position in the conclusion of t
```

Unlike the case of Sect. 5.1, the generated query is already very efficient: this time we do not need any trade-off to improve the performance. This is a trivial consequence of the fact that the imposed constraints are all *must constraints*.

5.3 Use Case 3: Finding the Proofs of a Statement

The last use case we present is locating a proof (or a definition) whose statement (i.e. its type) is known. The query seems quite uninteresting, but the inability to perform it may have very bad implications in terms of waste of efforts. Indeed, every time we need to prove a lemma, we would like to issue a query that checks if somebody else already proved it; the Coq system provides no way to perform this operation and, as a result, it is not at all unusual to find simple arithmetical properties proved five, eight or even more times by different persons working in remotely located teams[11].

The kind of matching we are interested in is extremely simple: no metavariables are present in both the statement of the theorem and the statements in the library. Close enough false matches, though, play an important role: for example, when looking for a proof of $\forall n.\ n = n * 1$, any of the following statements would be satisfactory: $\forall n.\ n = 1 * n$; $\forall n.\ n * 1 = n$; $\forall n. 1 * n = n$

[10] Both sentences are instances of the following invariant that is true for any kind of matching: "all the constants in the pattern must occur in the matched term".

[11] E.g. the proof of the fact that 1 is the neutral element of multiplication was given five times. Of course everybody suspects that a proof should already exist. Nevertheless, proving it again is faster than guessing who gave the proof and in what library. As a consequence, the theorem is proved again and the new proof is stored in the library under development by the user, which is likely to be unrelated to arithmetical properties. Thus, without an automatic searching procedure, it is virtually impossible to know that a proof of the theorem was put in that library.

We identify the following set of metadata, which constitute a close approximation of the first two levels of the abstract syntax tree of the pattern:

- For each occurrence of a constant, we record its *position*, as an element of the following set: {`MainConclusion, InConclusion, MainHypothesis, InHypothesis`}. The prefix `Main` means that the constant occurs in head position in the `Conclusion/Hypothesis` of the theorem.

- For each occurrence of a bound variable, we record its *position* in the set {`MainConclusion, MainHypothesis`}. Since bound variables occur almost in every statement, they are not very effective for searching. Thus we are interested only in second order or higher order variables, that are much less frequent. We syntactically recognize a subset of these variables by the fact that they occur in head position: these are the only occurrences we record in the metadata.

- For each occurrence of a sort Prop or Set[12], we record its *position* in the set {`MainConclusion, MainHypothesis`}

- For each occurrence in {`MainConclusion, MainHypothesis`} we record its *depth*, i.e. the number of universal quantifiers in the statement or hypothesis.

On these metadata we automatically impose both *must constraints* and *only constraints*[13] to achieve both performance and accuracy at the same time[14]. As in the previous cases, our experiments over the Coq library show that the chosen set of constraints is a good compromise between the strictness of the query (in terms of false matches) and the need to get results close to the expected ones (interesting false matches).

Expert users of the system can also edit the set of generated constraints, relaxing both the *must constraints* or the *only constraints*. As a result, they are able to answer similar queries such as: *Retrieve every binary relation* or *Retrieve any n-ary relation over natural numbers*.

Instead of describing here these and other use cases with the introduction of new ad hoc constraints, we will present in the next section a generic set of constraints that have been identified so far in the projects HELM, MoWGLI and MKM-Net. The set of constraints is general enough to subsume the constraints identified in the previous use cases. Moreover it is relatively stable, since

[12] Prop is the type of every proposition and Set is the type of any data-type. They are used to capture second order and higher-order quantifications over properties and types. They can also be used to capture axiom schemas in first-order theories.

[13] With *only constraints* we mean that every occurrence of a constant, bound variable or sort in the matched term must occur in the *only constraints*. In Sect. 5.1 this test was restricted to the occurrences in the conclusion.

[14] There is no trade-off here: the *must constraints* cannot filter out any good candidate since there are no metavariables in the pattern.

no new constraint was added in the last few months. The current work is focused on the automatic generation of larger and larger classes of queries based on these constraints. For example, we are now able to write by hand queries to retrieve every theorem in the library of Coq stating the associative or commutative property of any binary operator on any datatype. The number of false matches for these kinds of queries is surprisingly low[15]. It is surely possible to generate these queries automatically starting from the formal definition of the wanted property. The interested reader can find the query that retrieves all the theorems that state the associative property of some binary operator in [55].

5.4 A Generic Class of Metadata and Constraints

There is an obvious common pattern for the metadata identified in the use cases above: they provide an approximate description of the abstract syntax tree of the statements[16] in the library in terms of the constants and bound variables occurring in them, together with the occurrence locations. This observation leads us to describe our metadata model using *objects* having a list of *references*: the objects represent the statements in the library at the metadata level, i.e. they are the entities the metadata are about, while the references are the metadata describing the occurrences of constants and bound variables in the statements. As we said, an object may have three kinds of references. Namely:

- `refObj` describes the occurrence of a primitive or defined constant in terms of an `occurrence`, a `position` and a `depth` (see below).
- `refSort` describes the occurrence of a sort in terms of a `sort`, a `position` and a `depth` (see below). A sort is the type of propositions or data-types in higher-order logics (i.e. the second order quantification *"for every property* P*"* can be thought as the typed universal quantification $\forall \mathtt{P} : \mathtt{Prop}$). Sorts can also be used to capture axiom schemas in first-order theories.
- `refRel` describes the occurrence of a bound variable in terms of a `position` and a `depth` (see below).

The value of a reference is a record made of several fields, whose semantics is:

- `occurrence` specifies the referred defined constant. Its value is an object.
- `sort` specifies the referred sort. Its value belongs to a predefined collection of entities representing the possible sorts at the metadata level.

[15]For example, looking for every associative property in the Coq library gives 82 good matches and just 6 false matches.
[16]With statements here we mean the statements of axioms and theorems, and also the types of definitions.

- `position` specifies the position of the described occurrence in the statement. Its value belongs to a predefined collection of entities representing the possible positions at the metadata level. According to the previous use cases, we identified the following positions:
  - `MainHypothesis`: in head position of a statement premise.
  - `InHypothesis`: in another position of a statement premise.
  - `MainConclusion`: in head position of the statement conclusion.
  - `InConclusion`: in another position of the statement conclusion.
  - `InBody`: not in the statement (for instance in its proof).
- `depth` specifies a depth index associated to the position of an occurrence in the statement. Its value is a natural number. As we saw in the use cases, the `depth` of a reference is defined only when its `position` is set to `MainHypothesis` or to `MainConclusion` and it represents the number of premises of that hypothesis or conclusion.

The above description shows that every reference has some fields for locating the corresponding occurrence in the statement (`position` and `depth`) and may have one field specifying the referred entity (`occurrence` and `sort`)[17].

The queries we generate on these metadata are once again inspired by the use cases we saw in the previous sections. Our general approach concerning query generation is that complex queries should be obtained composing basic queries generated from the imposed atomic constraints on the wanted objects.

We identify the following two classes of basic queries:

A. **The wanted objects must have a reference to a given object R (or to a given primitive constant S or to a bound variable) in a given position P with a given depth index D.**

   Here we search for the objects having the assigned values in corresponding fields of at least one of their references. Using the pseudo-language exploited in the previous sections, a query of this class looks like:

```
select every t in the library such that
 the set {
  select every r in t.refObj such that
   r.occurrence = R and r.position = P and r.depth = D
 } is not empty
```

   This class of queries includes what we called the basic *must constraints*.

[17] `refRel` does not have this field because, up to now, we did not meet any use case in which we need to discriminate between the occurrences of different bound variables.

B. **The wanted objects may have a reference to an object (or to a primitive constant or to a bound variable) only if its position is not included in a given set U of positions, or if it concerns a given object R (or a primitive constant S, or a bound variable) in a given position P with a given depth index D.**

Here we query the objects such that there are no references whose fields do not have the assigned values. A query of this class may look like:

```
select every t in the library such that
 the set {
  select every r in t.refObj such that
   not
    (r.occurrence = R and r.position = P and r.depth = D)
   and r.position belongs to U
 } is empty
```

This formulation exploiting the logical double negation is more convenient, since it can be expressed more easily in SQL. This class of queries includes what we called the basic *only constraints*.

Note that the parameters R, S, P, D, U are always optional.

Coming to the issue of how the above queries should be composed, let us suppose that we are interested in finding the objects satisfying a given set (call it K) of "basic constraints". If the set K contains p constraints in the class A, the wanted objects are meant to satisfy all of them. Since a query of the class A (a *must constraint*) is of the general form

```
M(i) =
 select every t in the library such that
  the set m(i) is not empty
```

the composition of M(1) . . . M(p) is:

```
M = M(1) + ... + M(p) =
 select every t in the library such that
  the set m(1) is not empty and
  ...
  and the set m(p) is not empty
```

Dually if the set K contains some constraints in the class B, the wanted objects are meant to satisfy at least one of them. Since a query of the classes B (an *only constraint*) is of the general form:

```
O(j) =
 select every t in the library such that
  the set o(j) is empty
```

the composition of O(1) . . . O(q), provided that they concern the same kind of reference, say `refObj`, is:

```
O = O(1) + ... + O(q) =
 select every t in the library such that
  the set o(1) intersected ... intersected o(q) is empty
```

and the same holds for the queries O′(1) . . . O′(q′) concerning `refSort` and for the queries O″(1) . . . O″(q″) concerning `refRel`. Now the final query is obtained composing M, O, O′ and O″ conjunctively:

```
M + O + O' + O'' =
 select every t in the library such that
  the set m(1) is not empty
    and ... and the set m(p) is not empty  and
  the set o(1) intersect ... intersect o(q) is empty  and
  the set o'(1) intersect ... intersect o'(q') is empty  and
  the set o''(1) intersect ... intersect o''(q'') is empty
```

5.5 The HELM Search-Engine

The HELM team provides an implementation of the searching technology described in this chapter. The implementation comprises:

- A data mining tool, developed mainly by Prof. Asperti, that extracts metadata from the distributed library of Coq theorems that is the main test case of HELM. The metadata generated are in RDF format. They are collected in a relational database.
- An implementation of a high level query language, called MathQL Level 1, developed by Dr. Ferruccio Guidi. An old version of the language is described in Guidi's Ph.D. dissertation [54]. MathQL Level 1 allows to perform schema-aware queries over RDF metadata. In Sect. 5.7 we will justify the development of the language identifying a set of basic operations that the language must support and that are not supported, for instance, by SQL.
- A library that compiles the high level constraints described in this chapter into MathQL Level 1.

- A Web service, called SEARCH-ENGINE, that provides several methods. Each method correspond to one of the high level queries described in the previous sections. The Web service computes from the query input the high level set of generic constraints; then it consults the library to obtain a MathQL Level 1 query, that is remotely submitted to the language interpreter; the interpreter answers the query by issuing a sequence of lower level queries to the SQL engine of the relational database where the metadata were stored. The output of the Web service is a simple XML document that just holds the list of URIs of the objects in the library that satisfy the query. Thus, the intended usage of the SEARCH-ENGINE is to be composed with UWOBO: the output of the SEARCH-ENGINE is fed to UWOBO, that applies to it a stylesheet that decides the appropriate rendering of the query result. For instance, we developed a stylesheet that expands the definitions and statements of the theorems in the query results when the number of results does not exceed a given threshold; otherwise a simple list of hyperlinks to the HELM rendering of the object is generated.

The SEARCH-ENGINE will be a fundamental component of our proof assistant prototype.

5.6 Comparison with Other Works

The MBase [69] system resolves a matching query by iterating the matching operation over each thesis in its database (which now holds about 6,600 theorems). Thanks to the clever implementation of the unification procedure, the time spent to perform the query is still very low, in the order of 0.8 – 0.9 seconds. Nevertheless, a library of just 6,600 theorems is to be considered a very small library; in fact the libraries of the Coq proof assistant and of the Mizar proof assistant already hold 40,000 theorems each and they cover just a small subset of the overall mathematical knowledge. This approach will definitely not scale to much larger libraries. To solve the problem, the MBase team is now working on a solution that is extremely close to the one proposed in this paper: the theorems will be indexed in order to avoid unifications that will surely fail. To our knowledge, no details on the indexing technology has been published so far.

The Mizar team is now developing a user-friendly query language for the Mizar library [14]. The query language is not based on unification, but it is nevertheless quite useful in retrieving interesting results from the library. To retrieve a theorem, the user combines a set of filters over the whole library, until the wanted result is found. The filters are not applied to the library itself, but to a set of indexes (metadata) automatically extracted from the library. Thus the Mizar approach is extremely close to our approach: the main difference is in the set of metadata that, in the case of Mizar, are often very peculiar to the system. For instance, in Mizar it is possible that several symbols share the same notation and there is a query to retrieve the symbols represented by a given notation. Other metadata, instead, are less refined versions of the ones we just presented.

The problem of speeding up matching and unification has already been extensively studied in the literature. In particular, several term indexing techniques have been proposed to speed up matching and unification by storing the whole library in an ad hoc data-structure particularly optimized for matching. Among the most successful structures, there are discrimination trees, substitution trees and, more recently, coded context trees [48]. The common idea behind these data-structures is to share the term structure as much as possible: two shared substructures are syntactically equal and thus they match one another. Matching and substitutions implemented over these structures can be several orders of magnitude more efficient than iterating unification over the whole library.

Fast term indexing techniques are not incompatible with our approach. In particular, they can be seen as the limit case where the collected metadata are so abundant to describe all the statements in the library and the information about their sharing. Of course, the choice of an efficient data-structure to collect the metadata is a necessary requirement to achieve the gain of fast term indexing. Even if fast term indexing techniques are surely much more performant than our implementation, our general approach has some merits. First of all, since our metadata are RDF triplets and not term contexts, they can easily be stored in a off the shelf relational database, while term indexing techniques require specialized libraries implementing complex data-structures. As a consequence, we can easily benefit from the advancements of the database community, including transparent distribution techniques. Moreover, writing Web spiders to collect new metadata and implementing new kinds of queries is a minor effort in our approach and leaves complete freedom to the implementor. On the contrary, even if the great majority of queries can be reduced to unification, unification is the only operation that is natively provided by term indexing data structures. Moreover, the experience of the Mizar group, also supported by our experiments, shows that unification is often an overkill: the results obtained by querying a very small set of computed metadata are often precise enough to skip the unification step. Finally, the interesting false matches provided by our implementation do provide useful information "for free". To retrieve the same information from term indexing data structures, additional effort must be spent.

5.7 Correctness of the Filtering Operation for First Order Matching

This is the first of two sections that describe a formal framework for the technique described in the previous chapters. The framework has been partially inspired by abstract interpretation. Indeed it is possible to cast the ideas in that framework, although the embedding is quite artificial and does not provide additional insight.

In this section we present the standard definition of first order matching and we prove the correctness of the filtering operation defined in Sect. 5.1 with respect to first order matching. Although a direct proof would be absolutely trivial, we prefer to prove a more general result based on an abstract definition of metadata structure. Then we show that the metadata introduced in Sect. 5.1 satisfies all the properties required by the metadata structure definition. We also present as an example another admissible (but less manageable) class of metadata and we introduce an order relation on metadata structures according to their precision.

We are interested only in the conclusion of mathematical statements and we restrict ourselves to a very simple language of applicative expressions (the subset of the untyped λ-calculus obtained by not considering λ-abstractions). In the next section we will slightly enlarge it also to include the hypotheses.

We will now formally present our language and the set of rigid patterns we are interested in. Expressions will be a subset of patterns. We also introduce the usual standard ordering over patterns, induced by the definition of substitution. Then we define first order matching using the introduced order.

Let a, b, c, f, g range over a non empty set of constants $\mathcal{C}$ and X, Y, Z over an enumerable set of meta-variables $\mathcal{V}$.

Definition 5.2 (Expressions)

$$E ::= c \mid (E\ E) \qquad\qquad \textit{expressions}$$

Definition 5.3 (Patterns and rigid patterns)

$$\begin{array}{ll} R ::= c \mid (R\ P) & \textit{rigid patterns} \\ P ::= X \mid c \mid (R\ P) & \textit{(general) patterns} \end{array}$$

Rigid patterns are patterns used to match the head of an application. General patterns are used to match the arguments of an application. Notice that an expression is also a (rigid) pattern, and a rigid pattern is also a general pattern. Let $\mathcal{E}, \mathcal{R}, \mathcal{P}$ be respectively the set of all the expressions, the set of all the rigid patterns and the set of all the general patterns.

Definition 5.4 (Substitutions) *A substitution is a partial function from $\mathcal{V}$ to P.*

We let σ, σ' range over substitutions.

Definition 5.5 (Substitutions application) *Let P be a pattern (either rigid or general) and let σ be a substitution. The application of σ to P (that we write $P\sigma$) is defined inductively over P as follows:*

$$X\sigma = \begin{cases} \sigma(X) & \textit{if } X \in \mathrm{Dom}(\sigma) \\ X & \textit{otherwise} \end{cases}$$

$$c\sigma = c$$

$$(P_1\ P_2)\sigma = (P_1\sigma\ P_2\sigma)$$

Definition 5.6 (Canonical ordering) *We define the canonical ordering* $\leq \subseteq \mathcal{P} \times \mathcal{P}$ *over patterns as follows:* $P \leq P' \iff \exists \sigma.\ P\sigma = P'$.

Intuitively, $P \leq P'$ means that P is "less instantiated or equivalent to" P'.

Definition 5.7 (First order matching) *Let* E *be an expression and* R *be a rigid pattern. We say that* E matches R *if* $R \leq E$.

Definition 5.8 (Matching patterns) *Let* E *be an expression and* S *a set of rigid patterns. We denote with* $\mu(S, E)$ *the set of rigid patterns in* S *that* E *matches:*

$$\mu(S, E) \stackrel{\mathrm{def}}{=} \{R \in S \mid E \textit{ matches } R\} = \{R \in S \mid R \leq E\}$$

Notice that computing the set of matching patterns for a given expression is exactly the query we were interested in the first use case (Sect. 5.1).

We are now ready to define what is an admissible metadata structure:

Definition 5.9 (Admissible metadata structure) *An* admissible metadata structure *is a triplet* $(\mathcal{M}, \sqsubseteq, \epsilon)$ *such that* $(\mathcal{M}, \sqsubseteq)$ *is a well-founded partial order and* $\epsilon : \mathcal{R} \to \mathcal{M}$ *(the* extraction function*) is a morphism of partial orders.*

Let us now formally define metadata-based filtering:

Definition 5.10 (Filtering) *Let* E *be an expression,* S *a set of rigid patterns and* $\mathbb{M} = (\mathcal{M}, \sqsubseteq, \epsilon)$ *an admissible metadata structure. The result* $\phi_{\mathbb{M}}(S, E)$ *of metadata-based filtering* S *looking for* E *is defined as follows:* $\phi_{\mathbb{M}}(S, E) \stackrel{\mathrm{def}}{=} \{R \in S \mid \epsilon(R) \sqsubseteq \epsilon(E)\}$

The correctness of the filtering operation, i.e. no matching pattern is filtered out, is a trivial property of the definition of admissible sets of metadata:

Theorem 5.11 (Filtering correctness) *For each expression* E*, set of rigid patterns* S *and admissible set of metadata* $\mathbb{M} = (\mathcal{M}, \sqsubseteq, \epsilon)$*,* $\mu(S, E) \subseteq \phi_{\mathbb{M}}(S, E)$

Proof: Let $R \in \mu(S, E)$. Then $R \leq E$ by definition of $\mu(\_,\_)$. Then $\epsilon(R) \sqsubseteq \epsilon(E)$ by definition of admissible set of metadata. Hence $R \in \phi_{\mathbb{M}}(S, E)$ by definition of $\phi(\_,\_)$. □

We are now ready to address Theorem 5.1 (page 52). Because of Theorem 5.11, to prove the correctness of Theorem 5.1, we only need to formally define the set of metadata introduced in Sect. 5.1, equip it with an ordering relation and prove that it forms an admissible metadata structure.

Definition 5.12 (Simple metadata) *Let* R *be a rigid pattern and let* $\mathcal{C}_R$ *be the multi-set of constants that occur in* R*. We define* $\epsilon(R) \subseteq \mathcal{C} \times 2^{\mathcal{C}}$ *(the simple metadata over* R*) as follow:* $\epsilon(R) \stackrel{\mathrm{def}}{=} (c, C)$ *such that*

1. $R = c$ *or* $R = (c\ P_1\ \ldots\ P_n)$ *for some finite sequence of patterns* $P_1, \ldots, P_n$

2. $C = \{a \in \mathcal{C} \mid a$ *occurs in conclusion in* $R\}$ *where* a occurs in conclusion in R *if*

   - $a = c$ *and* c *occurs at least twice in* $\mathcal{C}_R$ or
   - $a \neq c$ *and* a *occurs at least once in* $\mathcal{C}_R$

We call c *the* main conclusion *of* R*.*

Simple metadata are naturally equipped with the following partial order relation:

Definition 5.13 ($\preccurlyeq$) $(c, S) \preccurlyeq (c', S') \iff c = c' \wedge S \subseteq S'$

Fact 5.14 *$(\mathcal{C} \times 2^{\mathcal{C}}, \preccurlyeq)$ is a well-founded partial order relation*

To prove that this simple metadata forms an admissible metadata structure, we need a couple of lemmas that show that simple metadata capture an invariant of first-order matching.

Lemma 5.15 (Substitution preserves constant occurrences) *If a constant c occurs in a pattern P, then c occurs in $P\sigma$ for each substitution σ.*

Proof: The proof, by induction over the structure of P, is trivial. □

Lemma 5.16 *Let R be a rigid pattern, $(c, C) = \epsilon(R)$, σ be a substitution and $(c', C') = \epsilon(R\sigma)$. Then $c = c'$ and $C \subseteq C'$.*

Proof: If $R = c$ then $R\sigma = c$ and the proof trivially holds. Otherwise let $R = (c\ P_1\ \ldots\ P_n)$. Then $R\sigma = (c\ P_1\sigma\ \ldots\ P_n\sigma)$ and c is the main conclusion of both R and $R\sigma$. Let $\mathcal{C}_R$ be the multi-set of constants that occur in R and $\mathcal{C}'_R$ be the multi-set of constants that occur in $R\sigma$. From Lemma 5.15 we already know that $\mathcal{C}_R \subseteq \mathcal{C}'_R$. From the definition of $\epsilon(\_)$ we conclude that $C \subseteq C'$. □

Theorem 5.17 *$\mathbb{M} = (\mathcal{C} \times 2^{\mathcal{C}}, \preccurlyeq, \epsilon)$ is an admissible metadata structure.*

Proof: Since we already know that $(\mathcal{C} \times 2^{\mathcal{C}}, \preccurlyeq)$ is a well-founded partial order relation, we only have to prove that ϵ is a morphism of partial orders, i.e. we need to prove that $\forall R, R' \in \mathcal{R}. R \leq R' \Longrightarrow \epsilon(R) \preccurlyeq \epsilon(R')$.

Let R, R' be rigid patterns such that $R \leq R'$. Then there exists a substitution σ such that $R\sigma = R'$. Thus, by Lemma 5.16, we know that $\epsilon(R) \preccurlyeq \epsilon(R\sigma) = \epsilon(R')$. □

In Sect. 5.1 we already motivated our choice of $\mathcal{C} \times 2^{\mathcal{C}}$ as our set of metadata. We did not motivate our choice of pseudo-language for performing the query. It is now clear that, in order to be as generic as possible, we need a query language that allows us to compare metadata according to the ordering relation of the metadata structure. As a consequence, we cannot adopt SQL as our query language, since SQL only deals with equality and disequality over primitive data types, while our ordering relation is over metadata. MathQL Level 1 [54] is a concrete implementation of a query language with enough expressive power to compare — using the subset inclusion relation — metadata that can be expressed as lists of identifiers. Our simple metadata discussed in Sect. 5.1 are an example of metadata that enjoy this property.

Having a formal definition of metadata structure, we can also easily define when a metadata structure is *more precise* than another.

Definition 5.18 (Morphisms of metadata structures) *Let* $(\mathcal{M}_1, \sqsubseteq_1, \epsilon_1)$, $(\mathcal{M}_2, \sqsubseteq_2, \epsilon_2)$ *be two metadata structures. A function* $f : \mathcal{M}_1 \to \mathcal{M}_2$ *is a* morphism of metadata structures *when* f *is a morphism of ordered relations and* $\epsilon_2 = \epsilon_1 \circ f$.

Definition 5.19 *A metadata structure* $\mathbb{M}_1$ *is* more precise *than a metadata structure* $\mathbb{M}_2$ *(notation:* $\mathbb{M}_1 \leq \mathbb{M}_2$*) when there exists a morphism* f *from* $\mathbb{M}_1$ *to* $\mathbb{M}_2$.

Fact 5.20 $\leq$ *is a well-founded partial order between metadata structures. It has a minimum that is the metadata structure* $\mathbb{E} = (\mathcal{E}, \leq, \mathtt{id})$.

To conclude the section, we give the formal definition of the *rigid skeleton* of a pattern, together with a partial order over rigid skeletons, and we prove that rigid skeletons also form an admissible metadata structure that is more precise than the structure of simple metadata. A rigid skeleton is a pattern where at most one variable occurs, and it is a simplification of a pattern where we loose all the information about the variables that occur, replacing them with a hole. Although rigid skeletons constitute another example of admissible metadata structure, they are less suited for our purposes since as data structures they are too complex to be stored in a database and to be queried even using a rather expressive query language such as MathQL Level 1.

Definition 5.21 (Rigid skeletons) *The set* B *of rigid skeletons is defined as follows:*
$\mathcal{B} \stackrel{\mathrm{def}}{=} \{P \mid X$ *is the only variable that occurs in* P *or no variables occur in* $P\}$.

Definition 5.22 (Rigid skeleton of a pattern) *Let* P *be a pattern and let* σ_X *be the constant substitution that instantiates every variable in* $\mathcal{V}$ *with* X. *We define the* rigid skeleton $\mathcal{B}_P$ *of* P *as follows:*
$\mathcal{B}_P \stackrel{\mathrm{def}}{=} P\sigma_X$.

Definition 5.23 ($\preccurlyeq$**)** *We define* $\preccurlyeq \subseteq \mathcal{B} \times \mathcal{B}$ *as the smallest relation such that:*

$$\frac{}{X \preccurlyeq P} \qquad \frac{}{c \preccurlyeq c} \qquad \frac{P_1 \leq P_1' \quad P_2 \leq P_2'}{(P_1\ P_1') \preccurlyeq (P_2\ P'2)}$$

Fact 5.24 $\preccurlyeq$ *is a well-founded order relation*

In Sect. 5.1 we stated informally that the substitution operation preserves the rigid skeleton of a pattern. We can now reformulate formally the property of preservation of rigid skeletons and prove the theorem:

Theorem 5.25 (Preservation of rigid skeletons under substitution) *Let* P *be a pattern and* σ *be a substitution. Then* $\mathcal{B}_P \preccurlyeq \mathcal{B}_{P\sigma}$.

Proof: Let σ_X be the constant substitution that instantiates every variable in $\mathcal{V}$ with X. We proceed by induction over the structure of P:

- $P = Y$
  We have $\mathcal{B}_P = P\sigma_X = X$. From the definition of $\preccurlyeq$ we know that $\mathcal{B}_P \preccurlyeq \mathcal{B}_{P\sigma}$.

- $P = c$
  We have $P\sigma = c = P$. Hence the thesis.

- $P = (P_1\ P_2)$
  We have $P\sigma = (P_1\sigma\ P_2\sigma)$ and $\mathcal{B}_{P\sigma} = (P_1\sigma\sigma_X\ P_2\sigma\sigma_X) = (\mathcal{B}_{P_1\sigma}\ \mathcal{B}_{P_2\sigma})$. By induction hypothesis we have $\mathcal{B}_{P_1} \preccurlyeq \mathcal{B}_{P_1\sigma}$ and $\mathcal{B}_{P_2} \preccurlyeq \mathcal{B}_{P_2\sigma}$. By definition of $\preccurlyeq$ we know that $\mathcal{B}_P = (\mathcal{B}_{P_1}\ \mathcal{B}_{P_2}) \preccurlyeq (\mathcal{B}_{P_1\sigma}\ \mathcal{B}_{P_2\sigma}) = \mathcal{B}_{P\sigma}$.

$\square$

Corollary 5.26 $(\mathcal{B}, \preccurlyeq, \mathcal{B}\_)$ *is a metadata structure.*

Theorem 5.27 $\mathbb{B} \leq \mathbb{M}$ *(rigid skeletons are more precise than simple metadata)*

Proof: The extraction function ϵ of the metadata structure $\mathbb{M}$ is a morphism from $\mathbb{B}$ to $\mathbb{M}$. $\square$

5.8 Correctness of the Filtering Operation for Extended First Order Matching

In this section we formally introduce an extension of first order matching that is expressive enough for the kind of matching we needed for the use cases in Sect. 5.2 and Sect. 5.3. We also formally define the corresponding notion of filtering, based on the definition of metadata structures we gave in Sect. 5.7, and we prove the correctness of filtering in the general case and for the generic class of metadata we introduced in Sect. 5.4.

Let $\mathcal{C}$ be a non empty set of constants ranged over by a, b, c, f, g. Let $\mathcal{R}$ be an enumerable set of variables ranged over by x, y, z. Let $\mathcal{V}$ be an enumerable set of meta-variables ranged over by X, Y, Z and let $\mathcal{L}$ be another enumerable set of row meta-variables ranged over by $\square, \square_1, \square_2$. $\mathtt{Prop}$ is the type of all the propositions: we use $\mathtt{Prop}$ to express second-order quantifications. Since we need a language for theorem statements, we extend our language of applicative expressions of Sect. 5.7 by adding logical implications and universal quantifications.

Definition 5.28 (Expressions)

$$\begin{array}{ll} K ::= c \mid x \mid \mathrm{Prop} \mid (K\ K) & \textit{conclusions} \\ E ::= K \mid E \Rightarrow E \mid \forall x : E.E & \textit{expressions} \end{array}$$

Patterns for conclusions are obtained by adding to the productions of K the meta-variable occurrences X. Patterns for expressions are obtained by adding a special pattern $\Box \Rightarrow P$ that matches any possible empty list of logical implications and universal quantifications:

Definition 5.29 (Patterns)

$$\begin{array}{ll} Q ::= X \mid c \mid x \mid \mathrm{Prop} \mid (R\ Q) & \textit{(general) conclusion patterns} \\ R ::= c \mid x \mid \mathrm{Prop} \mid (R\ Q) & \textit{rigid conclusion patterns} \\ P ::= R \mid \Box \Rightarrow P \mid\ P \Rightarrow P \mid \forall x : P.P & \textit{expression patterns} \end{array}$$

Let $\mathcal{E}, \mathcal{K}, \mathcal{P}, \mathcal{R}, \mathcal{Q}$ be respectively the set of all the expressions, the set of all the conclusions, the set of all the expression patterns, the set of all the rigid conclusion patterns and the set of all the general conclusion patterns. We have $\mathcal{E} \subseteq \mathcal{P}$.

Definition 5.30 (Substitutions) *A substitution is a couple (σ_V, σ_L) where σ_V is a partial function from $\mathcal{V}$ to $\mathcal{Q}$ and σ_L is a partial function from $\mathcal{L} \times \mathcal{P}$ to $\mathcal{P}$ such that $\forall (\Box, P) \in \mathcal{L} \times \mathcal{P}$. $\mathit{is\_suffix}(P, \sigma_L(\Box, P))$ holds, where $\mathit{is\_suffix} \subseteq \mathcal{P} \times \mathcal{P}$ is the smallest relation such that:*

$$\frac{}{\mathit{is\_suffix}(P, P)} \qquad \frac{\mathit{is\_suffix}(P, P')}{\mathit{is\_suffix}(P, \Box \Rightarrow P')}$$

$$\frac{\mathit{is\_suffix}(P, P_2)}{\mathit{is\_suffix}(P, P_1 \Rightarrow P_2)} \qquad \frac{\mathit{is\_suffix}(P, P_2)}{\mathit{is\_suffix}(P, \forall x : P_1.P_2)}$$

We let σ, σ' range over substitutions.

Definition 5.31 (Substitution application) *Let P be an expression pattern and let $\sigma = (\sigma_V, \sigma_L)$ be a substitution. The application of σ to P (that we write $P\sigma$) is defined inductively over P as follows:*

$$\begin{aligned}
X\sigma &= \begin{cases} \sigma_V(X) & \textit{if } X \in \mathrm{Dom}(\sigma_V) \\ X & \textit{otherwise} \end{cases} \\
c\sigma &= c \\
x\sigma &= x \\
\mathrm{Prop}\sigma &= \mathrm{Prop} \\
(P_1\ P_2)\sigma &= (P_1\sigma\ P_2\sigma) \\
(\Box \Rightarrow P)\sigma &= \begin{cases} \sigma_L(\Box, P\sigma) & \textit{if } \Box \in \mathrm{Dom}(\sigma_L) \\ \Box \Rightarrow P\sigma & \textit{otherwise} \end{cases}
\end{aligned}$$

Example 5.32 Let

$$\sigma_V(X) = \begin{cases} (f\ x\ y) & \text{if } X = Y \\ \textit{undefined} & \text{otherwise} \end{cases} \qquad \sigma_L(\Box, P) = \begin{cases} (f\ y\ y) \Rightarrow (f\ y\ x) \Rightarrow P & \text{if } \Box = \Box_1 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

and let σ be the substitution (σ_V, σ_L). Applying σ to the expression pattern $\forall x : S.\ \forall y : S.\ \Box_1 \Rightarrow Y$ yelds $\forall x : S.\ \forall y : S.\ (f\ y\ y) \Rightarrow (f\ y\ x) \Rightarrow (f\ x\ y)$.

So far we have redefined our language of expressions, patterns and substitutions. The canonical order $\leq$ between patterns and the notion of matching are the usual one (and they are absolutely identical to the corresponding definitions in Sect. 5.7):

Definition 5.33 (Canonical ordering) *We define the canonical ordering $\leq \subseteq \mathcal{P} \times \mathcal{P}$ over patterns as follows:* $P \leq P' \iff \exists \sigma.\ P\sigma = P'$.

Intuitively, $P \leq P'$ means that P is "less instantiated or equivalent to" P'.

Definition 5.34 (Extended first order matching) *Let E be an expression and P be a pattern. We say that* E matches P *if* $P \leq E$.

In Sect. 5.7 we were interested in all the patterns that matched a given expression; we called them matching patterns. According to the duality already discussed in Sect. 5.2, we are now interested in all the expressions that match a given pattern. We call them *matching expressions*.

Definition 5.35 (Matching expressions) *Let P be a pattern and S a set of expressions. We denote with $\mu(P, S)$ the set of expressions in S that P matches:*

$$\mu(P, S) \stackrel{\text{def}}{=} \{E \in S \mid E \textit{ matches } P\} = \{E \in S \mid P \leq E\}$$

Notice the duality between this definition and the definition of matching patterns given in Sect. 5.7 (page 65).

Since we are interested in matching expressions, we need to redefine the notion of filtering that was given for matching patterns in Sect. 5.7. The two definitions are obviously dual:

Definition 5.36 (Filtering) *Let P be a pattern, S a set of expressions and $\mathbb{M} = (\mathcal{M}, \sqsubseteq, \epsilon)$ an admissible metadata structure (defined in Sect. 5.7). The result $\phi_{\mathbb{M}}(P, S)$ of metadata-based filtering S matching P is defined as follows:* $\phi_{\mathbb{M}}(P, S) \stackrel{\text{def}}{=} \{E \in S \mid \epsilon(P) \sqsubseteq \epsilon(E)\}$

The correctness of the filtering operation, i.e. no matching expression is filtered out, is a trivial property of the definition of admissible set of metadata. Alternatively, it is an application of duality to Theorem 5.11.

Theorem 5.37 (Filtering correctness) *For each pattern P, set of expressions S and admissible set of metadata $\mathbb{M} = (\mathcal{M}, \sqsubseteq, \epsilon)$,* $\mu(P, S) \subseteq \phi_{\mathbb{M}}(P, S)$

Proof: Let $E \in \mu(P, S)$. Then $P \leq E$ by definition of $\mu(\_,\_)$. Then $\epsilon(P) \sqsubseteq \epsilon(E)$ by definition of admissible set of metadata. Hence $E \in \phi_{\mathbb{M}}(P, S)$ by definition of $\phi(\_,\_)$. □

Because of Theorem 5.37, to prove the correctness of the general case (that subsumes the residual use cases), we need to define formally the generic class of metadata introduced in Sect. 5.4, equip it with an ordering relation and prove that it forms an admissible metadata structure.

Definition 5.38 (Advanced metadata) *Let* P *be a pattern and let* $\mathrm{Depth}, \mathrm{Pos}, \mathrm{MainPos}$ *be the following datatypes:*

$$
\begin{aligned}
&\mathrm{Depth} ::= n \mid *\\
&\mathrm{MainPos} ::= \mathrm{MainConclusion}(\mathrm{Depth}) \mid \mathrm{MainHypothesis}(\mathrm{Depth})\\
&\mathrm{Pos} ::= \mathrm{MainPos} \mid \mathrm{InConclusion} \mid \mathrm{InHypothesis}
\end{aligned}
$$

where n *ranges over natural numbers.*
We define $\epsilon(P) \subseteq 2^{\mathcal{C} \times \mathrm{Pos}} \times 2^{\mathrm{MainPos}} \times 2^{\mathrm{MainPos}}$ *(the* advanced metadata *over* P*) as follows:*

$$
\begin{aligned}
&\epsilon(P) = \epsilon^0_B(P)\\
&\epsilon^i_B(c) = \{(c, \mathrm{MainConclusion}(i))\}, \emptyset, \emptyset\\
&\epsilon^i_B(x) = \emptyset, \{\mathrm{MainConclusion}(i)\}, \emptyset\\
&\epsilon^i_B(\mathrm{Prop}) = \emptyset, \emptyset, \{\mathrm{MainConclusion}(i)\}\\
&\epsilon^i_B(R\ Q) = \epsilon^i_B(R) \cup \epsilon_C(Q)\\
&\epsilon^i_B(\square \Rightarrow P) = \epsilon^*_B(P)\\
&\epsilon^i_B(P_1 \Rightarrow P_2) = \epsilon^0_H(P_1) \cup \epsilon^{i+1}_B(P_2)\\
&\epsilon^i_B(\forall x : P_1.P_2) = \epsilon^0_H(P_1) \cup \epsilon^{i+1}_B(P_2)\\
&\epsilon_C(X) = \emptyset, \emptyset, \emptyset\\
&\epsilon_C(c) = \{(c, \mathrm{InConclusion})\}, \emptyset, \emptyset\\
&\epsilon_C(x) = \emptyset, \emptyset, \emptyset\\
&\epsilon_C(\mathrm{Prop}) = \emptyset, \emptyset, \emptyset\\
&\epsilon^i_H(c) = \{(c, \mathrm{MainHypothesis}(i))\}, \emptyset, \emptyset\\
&\epsilon^i_H(x) = \emptyset, \{\mathrm{MainHypothesis}(i)\}, \emptyset\\
&\epsilon^i_H(\mathrm{Prop}) = \emptyset, \emptyset, \{\mathrm{MainHypothesis}(i)\}\\
&\epsilon^i_H(R\ Q) = \epsilon^i_H(R) \cup \epsilon_K(Q)\\
&\epsilon^i_H(\square \Rightarrow P) = \epsilon^*_H(P)\\
&\epsilon^i_H(P_1 \Rightarrow P_2) = \epsilon^{i+1}_H(P_2)\\
&\epsilon^i_B(\forall x : P_1.P_2) = \epsilon^{i+1}_H(P_2)\\
&\epsilon_K(X) = \emptyset, \emptyset, \emptyset\\
&\epsilon_K(c) = \{(c, \mathrm{InHypothesis})\}, \emptyset, \emptyset\\
&\epsilon_K(x) = \emptyset, \emptyset, \emptyset\\
&\epsilon_K(\mathrm{Prop}) = \emptyset, \emptyset, \emptyset
\end{aligned}
$$

where $*+1 \stackrel{\text{def}}{=} *$ *and* $(S_1, S_2, S_3) \cup (S'_1, S'_2, S'_3) \stackrel{\text{def}}{=} (S_1 \cup S'_1, S_2 \cup S'_2, S_3 \cup S'_3)$.

The first element of the triplet is the list of constants that occur in the term, together with their position. The second element is the list of positions of bound variables that occur in sensible positions. The third element is the list of sensible positions of the occurrences of Prop. Advanced metadata are naturally equipped with the following partial order relation:

Definition 5.39 ($\preccurlyeq$)

$$(S_1, S_2, S_3) \preccurlyeq (S'_1, S'_2, S'_3) \iff \forall i \in \{1,2,3\}.\ \forall x \in S_i.\ \exists x' \in S'_i.\ x \preccurlyeq_D x'$$

where $\preccurlyeq_D$ *is the smallest relation such that:*

$$\frac{}{d \preccurlyeq_D d} \qquad \frac{d \preccurlyeq_D d'}{(c,d) \preccurlyeq_D (c,d')}$$

$$\frac{}{\mathsf{MainConclusion}(*) \preccurlyeq_D \mathsf{MainConclusion}(i)}$$

$$\frac{}{\mathsf{MainHypothesis}(*) \preccurlyeq_D \mathsf{MainHypothesis}(i)}$$

Fact 5.40 $(2^{\mathcal{C}\times \mathsf{Pos}} \times 2^{\mathsf{MainPos}} \times 2^{\mathsf{MainPos}}, \preccurlyeq)$ *is a well-founded partial order relation*

To prove that the advanced metadata form an admissible metadata structure, we need to prove that the advanced metadata capture invariants of extended first-order matching. The proof relies on a couple of simple lemmas:

Lemma 5.41 *Let* P *be a pattern. If* $\mathsf{MainHypothesis}(d)$ *or* $\mathsf{MainConclusion}(d)$ *occurs in* $\epsilon^*_T(P)$ *for* $T = B$ *or* $T = H$*, then* $d = *$.

Proof: The proof is by induction on the structure of P and it mimics the definition of ϵ^i_T. □

Lemma 5.42 *For all patterns* P, P'*, if is_suffix*(P, P') *then* $\epsilon^*_T(P) \preccurlyeq \epsilon^d_T(P')$ *for each depth* d *and for* $T = B$ *or* $T = H$.

Proof: The proof is by induction over the structure of P' and it mimics the definition of is_suffix.

- If $P' = P$ then $\epsilon^*_T(P) \preccurlyeq \epsilon^d_T(P')$ by Lemma 5.41.
- If $P' \equiv \Box \Rightarrow P''$ then $\epsilon^d_T(P') = \epsilon^*_T(P'')$ and, by induction hypothesis, we know that $\epsilon^*_T(P) \preccurlyeq \epsilon^d_T(P'')$ for each depth d. Hence the thesis.
- If $P' \equiv P_1 \Rightarrow P_2$ or $P' \equiv \forall x : P_1.P_2$, then $\epsilon^{d+1}_T(P_2) \preccurlyeq \epsilon^d_T(P')$ since $\epsilon^d_H(P') = \epsilon^{d+1}_H(P_2)$ and $\epsilon^d_B(P') = \epsilon^0_H(P_1) \cup \epsilon^{d+1}_B(P_2)$. Moreover, by induction hypothesis, we know that $\epsilon^*_T(P) \preccurlyeq \epsilon^{d+1}_T(P_2)$. We conclude the thesis using the transitivity property of $\preccurlyeq$.

□

Theorem 5.43 $\forall P, P' \in \mathcal{P}.\ P \leq P' \Rightarrow \epsilon(P) \leq \epsilon(P')$

Proof: Since $P \leq P'$, there exists a substitution $\sigma = (\sigma_V, \sigma_L)$ such that $P\sigma = P'$. The very long (twenty two cases) and technical proof is by induction over the structure of P and it mimics the definition of the function ϵ. Here we provide only the intuition about the four cases that are not trivial and that are not simple applications of the inductive hypotheses:

- $P \equiv X$ and $\epsilon \equiv \epsilon_C$ or $\epsilon \equiv \epsilon_K$
  $\epsilon_C(X) = \epsilon_K(X) = \emptyset, \emptyset, \emptyset$ and $\emptyset, \emptyset, \emptyset$ is the minimum element of the ordering relation $\preccurlyeq$. Hence the thesis.

- $P \equiv \square \Rightarrow P'$ and $\epsilon \equiv \epsilon^i_B$ or $\epsilon \equiv \epsilon^i_H$
  Let $T = B$ if $\epsilon \equiv \epsilon^i_B$ and let $T = H$ if $\epsilon = \epsilon^i_H$. If $\square \notin Dom(\sigma_L)$, then $P\sigma = \square \Rightarrow P'\sigma$ and we can apply the induction hypothesis. Otherwise $\epsilon(P) = \epsilon^*_T(P')$ and is_suffix$(P'\sigma, P\sigma)$. Applying Lemma 5.42 we get $\epsilon^*_T(P'\sigma) \preccurlyeq \epsilon^{i+1}_T(P\sigma)$. By induction hypothesis we know that $\epsilon^*_T(P') \preccurlyeq \epsilon^*_T(P'\sigma)$. The thesis follows from the transitivity property of $\preccurlyeq$.

$\square$

The main result of this section follows as a simple corollary of Theorem 5.43:

Corollary 5.44 $(2^{\mathcal{C}\times Pos} \times 2^{MainPos} \times 2^{MainPos}, \preccurlyeq, \epsilon)$ *is an admissible metadata structure*

5.9 Conclusions

In this chapter we presented a general methodology for speeding up queries by matching over large and distributed libraries of mathematical knowledge.

Our approach consists in capturing some of the matching invariants by means of metadata, which can be automatically computed by a batch process (a Web spider). Every time the user submits a pattern to be matched, we use these metadata to filter a small subset of the library items which is made of matching candidates: we compute a query over the metadata from the matching pattern; the results of the query are the matching candidates. The correctness of the filtering operation, which is the property of not loosing matching terms, is a trivial consequence of the fact that our metadata capture the matching invariants.

Depending on the kind of matching we are interested in, we need to identify a different set of metadata and we need to query them. A second outcome of this work has been to identify a reasonable and accurate set of metadata that covers several common matching procedures. They are described in Sect. 5.4.

To judge the efficiency of our approach, we implemented a prototype working over the HELM distributed library, made of about 40.000 theorems coming from the Coq proof assistant library.

The prototype has been implemented using the MathQL Level 1 query language [54] and has been turned into a Web service. A Web interface to the prototype is available following the appropriate link at the address: `http://helm.cs.unibo.it/library.html`. The prototype has been extensively used by one of the authors while developing a medium-sized Coq contribution [109]. The experiment result was extremely positive: the author was able to locate and reuse several theorems that were given in parts of the library where the author did not expect them to be. As a consequence, the library reusage factor was improved and the development effort was reduced.

Since the prototype is implemented as a Web service, it can easily be exploited by theorem provers or proof-planners to retrieve lemmas or theorems. Indeed, this is one of the usage of the SEARCH-ENGINE in the proof assistant prototype that we will describe in the second part of this dissertation.

An important feature of our filtering technology is that the filtering operation is not too strict, in the sense that wrong candidates that are "close enough" to matching terms are not dropped. These wrong candidates are very interesting, since the human user can often exploit them reducing his thesis to the found one; other times, instead, they can reveal missing assumptions in the user statement.

The interest in the wrong candidates, especially the ones that can be reduced to the user statements, is not a surprise. Indeed, several authors [40, 38] already pointed out that queries by matching are often too strict. For example, we would like the pattern $\forall x.x \geq 0$ to be able to match the statement $\forall x.0 \leq x$ too. Other typical examples are patterns involving binary operators: often we would like to match up to associative or commutative rewriting. The most general framework for this kind of query is provided by so-called "query up to isomorphisms" [40]: both the pattern and the statements in the library are reduced to a decidable normal form before matching; the reduction is such that a proof of the pattern can be automatically produced from the results of the query.

Chapter 6

Exportation Modules

The formalization of interesting and useful fragments of mathematics and computer science requires already existing libraries made of several elementary theories. In particular, we recently got more and more evidence of the fact that complex chapters of mathematics may be required even to justify the correctness of simple algorithms [60, 6].

Current tools for the development of formal mathematical knowledge, such as theorem-provers and proof assistants, have been successful in the last decade only in encoding small parts of mathematics and computer science, starting from the elementary ones. The lack of sharing between the different systems resulted in many theories formalized again and again, each time using a different tool. Moreover, since all of these systems are application-centric and use proprietary formats, the formalized knowledge is not even accessible to other systems based on the same logic. Indeed, one of the main goals of the HELM project and of this dissertation is to create document centric reasoning tools based on standard formats and aimed at maximizing information sharing and reusal.

Even if we propose to build a new generation of document centric systems, we cannot simply ignore the already existing libraries. First of all, loosing all the mathematical knowledge already formalized is not acceptable and is in clear contradiction with the aims of HELM. Secondly, developing a proof assistant prototype is already a time expensive task, and bugs could easily be introduced in the kernel. Thus, we need a big library to test the system and to be immediately able to use it for the development of non trivial results, in order to test its user friendliness and expressivity.

The solution consists in developing *exportation modules* for the existing mathematical reasoning tools. In theory, an exportation module is an extremely simple piece of code: it just has to convert the internal format of the application into an XML document that other tools can interpret. The exported XML documents must be encoded at the formal level.

The practice, though, is very different. In 1999 the author of this dissertation wrote a DTD for a variant of the Calculus of (Co)Inductive Constructions (CIC), which is the logical system of the Coq[1] proof assistant, and an exportation module from the Coq compiled library files (".vo" files) to the format. Then all the tools described in the previous chapters of this dissertations were implemented, and they were tested over the distributed library generated using the exportation module. Once the tools were developed, several wrong decisions in the design of the DTD and in the implementation of the exportation module were spotted. In 2002 we heavily changed the DTD and we reimplemented the exportation module almost from scratch. Notwithstanding this, we have already spotted some additional problems, and we will probably have to redesign the exportation module again.

In the meantime, the HELM project started a loose collaboration with some members of the NuPrL team, in order to develop an exportation module for the NuPrL system, that is based on a variant of the Martin-Löf constructive type theory. Thus a new DTD was developed and a new exportation module was written. Our previous experience helped in avoiding the most common mistakes. Notwithstanding this, the DTD was subject to major changes and the NuPrL exportation module was reimplemented at least once.

Thus, we decided to write down the principles that we had learnt in the previous four years. The paper [110] was presented at the second International Conference on Mathematical Knowledge Management. Its content is reproduced almost verbatim in this chapter.

6.1 Tips and Pitfalls

The main self-assessment criteria for an exportation module is that the exported information must be complete, general enough and unbiased, in the sense that it must not force some design choices that we would better avoid. The risk is to later find out to be unable to develop the expected tools.

Thus we must face at least the following issues: What information should be exported? What is the granularity of the generated files? How is the information described? (i.e. how should a DTD be designed)

6.1.1 What Classes of Information Should Be Exported?

Catalogue the information according to its use. When the information is required for more than one task, factorize.

Before writing the DTDs, catalogue all the information that is available inside the system and all the information you are interested in (which may or may not be already available). Since different tools may require different subsets of the whole knowledge and since you are likely to change the DTDs and the documents often, insulating the tools from changes in other parts of the

[1]Another proof assistant based on a different variant of the same calculus is Lego.

library is almost mandatory. It is not unusual to find out later that the grain was not fine enough. Example: not every operation that is applied to a theorem requires both the statement and the proof. Finding out which lemma can be applied, for instance, just requires the statement or, even better, the list of its hypotheses and the conclusion.

Sometimes some data is required to perform more than one task. In that case it is better to factorize, even if the factorized information does not seem at once to have any special meaning.

We have identified the following classes of information in the library of the Coq system:

- Definitions, inductive definitions and proof-objects as lambda-terms, according to the Curry-Howard isomorphism and the theory of CIC. Even for definitions, this is not what the user entered to the system, but the result of complex post-processing, typing and refining rules implemented in Coq. This is what is actually *proof-checked*, by type-checking the lambda-terms and testing the convertibility of their inferred type (in case of a proof, what the theorem really proves) with the expected type (the statement of the theorem). In the sequel, we will further refine theorems.

- Statements of the theorems. They are useful not only for proof-checking. First of all the user can be interested just in the statement of a theorem, that can be parsed and rendered independently of its proof. Secondly they are the only information required to answer several kinds of logic-dependent queries:

  1. which lemma can be applied in this situation?
  2. which theorem concludes an instance or a generalization of something?
  3. what can we conclude from a certain set of hypotheses?

  Note that these queries are essentially based on the notion of unification, which is an expensive operation when performed on very large sets of candidates (that must also be parsed and loaded into memory to apply unification).

  Statements of the theorems can be further divided into hypotheses and conclusion. This is not a trivial task, since this information is encoded in a lambda-term and because the constructors of the logical framework (Π-abstractions) are overloaded in Coq to mean either a dependent product, a non dependent product (function space), a logical implication or a universal quantification. Other logical frameworks such as the one of NuPrL, instead, can have several primitive or derived constructors for introducing hypotheses. Thus, to implement the logic independent queries in a general way, the separation must be performed in advance (either statically or dynamically, when needed).

- Logic independent information extracted from a statement. For example, the single notion of occurrence of a definition in a statement is useful to answer interesting queries:

  - which theorem states some property of an operator?
  - which theorem *may* be applied in a certain situation? Note that if we want to prove some fact about the multiplication of two real numbers, we are interested only on those statements where that multiplication occurs and no other uninteresting definition (let's say the "`append`" function of two lists) occurs. So we can effectively use this information to quickly filter out those theorems whose application will never succeed. Note that this filtering operation is logic independent (then it can be provided once and for all for every system), it is easily implemented using a standard relational or XML-based database and can be extremely more efficient than trying one at a time the applications, which usually involve higher-order unification of two expressions.

- Proofs. Their size is usually orders of magnitude bigger than the size of their statements. They are never rendered alone, but are an interesting part of the library for data-mining (for example to recognize similar proofs or to understand the complexity of some proofs). An usual operations on proofs that do not require additional information is proof improvement: a group of inference steps, possibly automatically found by the system, may be replaced with a shorter proof, usually human provided.

- Logic independent information extracted from a proof. As for the case of statements, the most interesting notion is that of occurrence. Given the list of occurrences it is easy to answer the following queries:

  - which proofs depend, directly or indirectly, on a given lemma or axiom?
  - which axioms does a proof depend on, either directly or indirectly?
  - which part of the library will be affected if some definition or axiom is changed?
  - what should I learn to be able to understand the following theorem?

- System dependent information related to a proof or definition. For example, in Coq and other systems there exists a notion of implicit arguments, which are those arguments that can be automatically inferred by the system if they are not provided. For example, when writing $x = \pi$ it is clear that the monomorphic equality of type $\forall T.T \rightarrow T \rightarrow \mathbf{Prop}$ is applied to the set of real numbers. This information is not necessary to proof-check a document, but may be useful to render it: information that can be inferred by a system can often be inferred by the human being and is usually omitted in an informal presentation. Finally, note that implicit arguments are really system dependent, since a more powerful system may be able to infer more information and thus consider more arguments to be implicit.

- Redundant information related to a proof or definition. A typical example is the opacity of a constant. Opaque constants are abstract data types, whose exact definition cannot be inspected. Proofs in proof-irrelevant systems are always opaque. In those systems that are not proof-irrelevant, such as Coq, all the constants may be considered transparent for the sake of proof-checking. This means that all constants may be expanded during proof-checking. Often, though, it is not necessary to expand every constant for proof-checking and knowing in advance which constants may not be expanded can make the system much more efficient. This information is essentially logically redundant, since there is an easy algorithm to make it explicit: try type-checking without any expansion and, in case of failure, backtrack and expand. Of course the computational complexity of this algorithm is, in the worst case, exponential in the number of constants that occur in the theorem being typed.

Another even more interesting example of redundant information is the types of the subexpressions of one proof, that corresponds to the conclusions of the subproofs. This information is completely logical redundant (if type-inference is decidable), but it is essential to render the term in a pseudo-natural language [34].

Since extracting the implicit information inside the system is easy, but it may be very difficult, time consuming or hardly possible to do with an external tool, the advice is to

Make implicit information explicit.

- Metadata related to definitions and theorems. These are other logic independent information such as author name or the version of the system the proof was developed in that can be useful to implement other kind of queries. Metadata can range from simple to very complex ones, like the ones needed in educational systems [71]. Usually, though, they are not provided inside the system or they are just given as unstructured comments.

- A system dependent history of the operations that lead to the creation of a definition or theorem. For example, in the case of Coq, we can export the proof-tree, which is the internal structured representation of the list of tactics (and their arguments) used to prove one theorem. This information may be useful both for rendering purposes and to replay the proof inside the system, in case we need to modify it.

- Comments. Comments provided by the users are an extremely valuable form of documentation. The problem is that they are often discharged during the lexical analysis of the input to the system; So they are likely to be unavailable inside the system itself.

- Parsing and pretty-printing rules. The existing systems only perform pretty-printing to ASCII notation, while we are interested in more advanced visual rendering. Thus we do not have any use for this information, that will not be exported.

- Tactics and decision procedures. It would be extremely interesting to put the code of tactics and decision procedures into the library. In this way it would become possible to replay the construction of a proof independently from the system that generated it and from its version. Indeed a problem we face is that every time some detail of the implementation of Coq changes, the same script produces a new slightly different proof. This represents a problem from the point of view of proof-engineers, that must continuously update the proofs every time a definition changes.

To have the tactics into the library we need first to formalize the language (and the libraries) in which they are implemented and this is surely a complex task.

Once the information classes have been identified, it is time to start developing a syntax (a DTD in the case of XML) for them. At this stage it is also important to understand what are the relations between the data belonging to the different classes. Those relations must be made explicit. Thus the third advice is

Develop one DTD for each class you are interested in. Make the links between different instances explicit in the DTD. Locate the atomic components.

Our initial expectation was to be able to heavily link at a very fine grained level the information collected in the documents belonging to the different classes. For example, it is reasonable to link:

- every type of a sub-term of a proof (inner-type) to that sub-term;
- every node of the proof-tree (that corresponds to a user or system issued tactic) to the sub-term it generated.

Still it is often the case that there is no real correspondence between those notions inside the system. For example it may happen that a tactic generates a term that is not present in the final proof, because a subsequent tactic modified or discarded it. Another possibility is that the whole proof can be generated using dirty tactics that create detours or lot of redexes that are later simplified during a clean-up stage. We consider this behaviour of the system to be a debatable design decision, since it introduces a gap between the proof the user wants to describe and the proof produced at the end. This may be especially annoying when the user is really interested in the generated proof-object and she is not given any effective tool to inspect it.

In these cases, instead of simply avoiding to provide that linking information, it is better to pinpoint to the developers of the system the problematic operations to be modified. In the worst case it may be necessary to create a branch in the system development to obtain a slightly modified tool that is used to produce the library that will be eventually exported.

The atomic components that must be located belong to the same information classes and represent the minimal referentiable units that have a precise meaning when considered as stand-alone documents[2]. Sometimes this notion is quite fuzzy. For example, what are the atomic components of a block of mutually defined functions? We may choose that the only atomic component is the whole block, and the functions are simply subparts that may be referenced starting from the whole block (using XPath paths, in the case of XML). Or we may consider each function to be atomic and make it refers to the other mutual defined functions via links. Note that some operations (rendering, proof-checking) require the whole block, while others (extraction of metadata used to answer queries) operate on one function at a time. Thus, according to our first suggestion, we should split the functions.

6.1.2 How Should Files Be Organized?

The need to clearly identify the atomic components, addressed in the previous sections, is related to our next suggestion:

Do not mix information about several components or belonging to several classes: one file for each class and for each component.

Assembling together information from different classes is perhaps the worst pitfall, for several reasons:

- The set of operations we want to perform on the data is not a fixed one. For example, we can implement several indexing tools to be able to answer to different queries. Since every tool requires and produces new information that should be put into the library, it is fundamental to be able to define new file formats (DTDs) and change existing ones. If several information classes are assembled together, this means that we will have to modify all the already developed tools to handle (usually to ignore!) the new or modified syntax. Moreover we will have to regenerate huge amounts of files distributed over the network (we are supposed to work with distributed libraries of formal mathematical knowledge).

- The amount of information we want to handle is large. For example, just the proof objects of the Coq libraries and their inner-types, once saved as compressed XML files, require about 260Mb. It is not unusual to have single theorems that, exported in XML and compressed, require 24Kb. Thus parsing (or even lexically analyze) these files is an expensive operation. Of course we want to avoid as much as possible wasting time in parsing parts of a file we are not interested in.

- It is reasonable, at least for some applications, to use a database to hold the exported information. In that case the XML representation is used just as an exchange format and mixing

[2]Of course the documents may have links to other documents that must be considered when determining their meaning. What is important here is that if some kind of context is requested, it must be explicitly stated by means of explicit links.

several data in the same file is not a problem, since the information will be disgregated inside the database. This approach, though, limits a priori the possibility of implementing other tools that, instead of connecting themselves to the database, will just work on the original files. Moreover, the database approach is surely heavier from the point of view of Web-publishing, where the most successful model is that of Web pages publishing, which is simpler and less demanding.

So far we have given suggestions on how to decide which classes of information should be exported and how files should be organized. We will now go back to the problem of designing the DTD and focus on the main difficulties that must be faced.

6.1.3 What Theory Should Be Described?

Describe, as much as possible, the theory and not the implementation.
Do not be afraid of mapping concepts from the implementation back to the theory.
Make the theory explicit.
Introduce new theories to justify the implementation.

There may be a big gap between the theory a system is based on and the way it is implemented. Proof assistants are implemented starting from a well-known theory (e.g. Martin-Löf Constructive Type Theory for NuPrL; the Calculus of Constructions for Coq). Later on, extensions to the theory are implemented and possibly proved to be correct. For example, both in the case of NuPrL and HOL, logic constructions that are derived in the original theory are made primitive in the implementation for efficiency reasons. In NuPrL this is done, for example, for the integer numbers and the usual arithmetic operations on them.

What usually happens is that after some years it is difficult to reconstruct a unified theory of all the extensions provided, while many other changes are completely undocumented. This happens not only in the kernel of the systems, which is responsible of checking that inference steps are well-applied, but also in external layers that rely on techniques (e.g. higher order unification) that are well-known in the literature and that require extensions to the representation of the proofs (e.g. metavariables that are typed holes in a proof-term).

When exporting from Coq the first time, we decided to be quite close to the implementation, even if some parts of what we exported were partially not understood. When we developed our proof-checker first and our proof assistant later, we realized that we had no clear foundation for the theory we were implementing. Moreover, having represented not the terms of the theory but their encoding in the Coq structures, the implementation of many operations were forcedly isomorphic to what was done in Coq, even if we wanted to experiment with other solutions. We will now present in details a couple of examples.

Metavariables and Existential Variables

According to the Curry-Howard isomorphism, a correct proof can be seen as a well-typed lambda-term in some lambda-calculus. Thus an incomplete, partially correct proof must be a well-typed lambda-term with holes therein. To extend the notion of well-typing to terms with holes, a hole (called *metavariable* in the literature [90]) cannot simply be a missing term, but must be associated to a sequent. So a metavariable has a given type and a typed context.

The two main operations on metavariables are instantiation and restriction. A metavariable can only be instantiated with a term that is closed w.r.t. the metavariable context and that has the expected type in that context. Instantiating a metavariable is a heavy operation, since it requires a linear visit of the whole proof-term. The other operation, restriction, deletes some hypotheses from the metavariable context. It is required, for example, to perform unification: to unify two metavariables the first step is to restrict both of them so that their contexts become equal (possibly up to a decidable convertibility relation).

How can those operations be implemented efficiently? One possibility is to implement restriction using instantiation: every time a metavariable should be restricted, a new metavariable of the right shape is generated and used to instantiate the old one. Since restriction occurs quite often, this implies that an efficient way to perform instantiation must be designed. This is the approach of Coq: restriction is reduced to instantiation and instantiation is not performed explicitly, but delayed using a new environment that maps every instantiated metavariable to the term used to instantiate it.

A completely different approach implements restriction explicitly, making each hypothesis in the sequent optional, so that it is possible to remove one hypothesis keeping trace of the fact that it was removed[3]. Nothing is done to speed up instantiation, that becomes a seldom required operation.

The two possibilities require different data-structures and representation of metavariables. Moreover, for technical reasons, in Coq there is a further distinction between full-fledged metavariables (called existential variables and used to represent holes in the type of other metavariables) and restricted metavariables that are only used to represent the open goals.

The first time we exported from Coq, we did not change the Coq representation. Later on, when implementing our own proof assistant, we found out that our choice forced us to a treatment of metavariables similar to the one of Coq. Moreover the distinction between the two kinds of metavariables did not allow any progress in the proof[4]. So we defined a different internal representation of metavariables, following the second solution, and we decided to change the DTD.

[3]This information is required for managing explicit substitutions that are needed to allow reduction of terms with metavariables.

[4]This is not a problem in Coq since the proof-tree with holes is generated on-demand only for pretty-printing and exporting purposes. Proof-trees are instead used to describe an incomplete proof and the progress on it.

Now, which is the right format for describing metavariables in the library? The only reasonable choice is to stick to the theory, where hypotheses in a metavariable context are not optional and there is no environment to delay instantiation. It is a responsibility of the systems to map back and forth between this standard and well-understood format and their internal encoding.

Sections, Variables and Discharging

In the syntax of Coq it is possible to abstract a group of definitions and theorems with respect to some assumptions. Example (in Coq syntax):

```
Section S.
 Variable A : Prop.
 Variable B : Prop.
 Definition H1 : Prop := A /\ B.
 Theorem T1 : H1 -> A.
  Proof.
  <some proof>
  Qed.
End S.
(* Here the type of H1 and T1 are different. See code fragment below *)
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True False (I,I)).
 Qed.
```

The previous fragment should be equivalent, from a logic point of view, to the following input:

```
Definition H1 : Prop -> Prop -> Prop := [A:Prop ; B:Prop]A/\B.
Theorem T1 : (A:Prop ; B:Prop)(H1 A B) -> A.
 Proof.
 <some slightly different proof>
 Qed.
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True False (I,I)).
 Qed.
```

The operation that transforms the first code fragment in the second one is called *discharging*. Discharging is implemented in an external layer of the Coq system, in such a way that the kernel of the system is given the discharged term (and the Coq kernel theory does not need to be modified).

Since, for rendering purposes, we are more interested in the first fragment, we exported the undischarged form of the theorems and definitions. The problem with that representation is that the theorem `T1` that is used in theorem `T2` is no more equal to the one defined above: its type is different! This implies two kinds of problems in developing tools:

1. While rendering `T2`, we would like to make the occurrence of `T1` an hyperlink to its definition. What we get is misleading for the reader: the theorem `T1` is shown to have a type that is not the same of its occurrence in `T2`.

2. To proof-check `T2` we need the type of the discharged form of `T1`. So we are obliged to discharge `T1` and this leads to serious problems: either we save the discharged form, as Coq does, and this goes against our initial choice; or we discharge the theorem on-the-fly when needed, and, being this an expensive procedure, we have to implement complex caching machineries[5].

The solution we are adopting now is simply to redesign the theory, replacing the notion of discharging with that of explicit named substitution. The new theory, that is also the theory we base our proof assistant prototype on, will be described in Sect. 8.3. According to the new theory, while exporting, we completely change the definition of `T2` to the following one:

```
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1[A := True ; B := False] (I,I)).
 Qed.
```

In this way we are no longer exporting the exact definition of `T2` inside Coq, but something more well-behaved for our purposes and that can be mapped back, if needed, to the Coq representation. Both of the previous problems are solved with this representation, since `T1` can be rendered as it is just adding the explicit substitution to the top and `T2` can be type-checked without discharging `T1`, by taking care of the explicit substitution in the typing rules of the system.

If we stop reflecting on the real cause of the problems given by discharging, we will easily identify another severe problem that must be faced when exporting the information: a proof assistant may be implemented in an imperative way, where there is a strong notion of time and the objects of the library change from time to time. This behaviour seems utterly incompatible with the notion of mathematical library:

A library, due to its nature, is a random access structure, both in space and time. Try to minimize the dependencies, give them the right direction, beware of "imperative" commands of the system that change already defined things. Make them immutable.

[5]This is what we implemented, but the nature of the discharging operation interfered with the usual locality reference principle of caches. As a result we got very poor and unexpected performances.

How can we face the situation in which we have to export some imperative information? For example, Coq has many other examples of this kind of information:

- A list of theorems that are considered while automatically searching a proof. Theorems can be added and removed from this list at pleasure.
- Implicit variables. Even if the system automatically chooses the variables it thinks it may infer, the user can force at any time the set of implicit variables for an object. Subsequent commands will use the new set.
- Opacity. It can be changed from transparent to opaque, to simulate abstract data types.
- Parsing and pretty-printing rules.

The situation can be described in two ways. The first one is to add the imperative command that changes the system status to the library, adding links from the command to all the other objects that are affected. The second way is to say that every command or definition of the system has an implicit dependency on the state of Coq. This dependency can be effectively made explicit. While the first solution is easier to implement, because that information is already available in the system, interpreting a command means in practice re-running all the commands in a clone of the Coq system to build the status requested. The second solution, instead, avoids rebuilding the status. Remember that, in a distributed setting, collecting all the given commands and definitions just to update them because of imperative commands is already an unaffordable operation. Managing a distributed status may be even worse. The price to pay, of course, is that objects that change are replicated again and again for each status they may have. Our experience shows that either the requested status information is minimal or there are some problems in the formulation of the theory that should be better removed, as in the case of discharging.

6.2 Further Discussion

The previous sections should make it obvious that there is a conflict between the two design principles "*what is expensive is worth saving*" and "*what is computable can be disregarded*". Indeed it is often the case that the computations that can be disregarded are also the expensive ones.

We recall that the two strong motivations for the second rule are: redundant information requires consistency checks; redundant information may require unwanted parsing of useless data. Our suggestion is to look for a compromise preferring to avoid redundant information when possible. When this is not the case, the expensive computed information should be stored in a different file, so that we can grant at least internal consistency, i.e. consistency of the data in the same file. Internal consistency is often enough for our needs. Unwanted parsing is also minimized in this way.

So far we have not tried to apply our guidelines to export information from a Computer Algebra System (CAS). Our feeling is that some of the issues we faced would be irrelevant for Computer Algebra Systems. For example, the theory behind these systems is quite uniform and better understood, just being some extension of a first-order setting. Determining the information classes to export should present more or less the same problems. Probably, though, other issues may get much more severe and new ones will certainly arise: CASs are more imperative than proof assistants and theorems prover, and a notion of library is much more difficult to define. Since the power of a CAS mainly derives from the procedures it implements and not from a library of pre-computed results, exporting a description of these procedures to the library could be compelling.

Since our experience derives from a long process of committing mistakes and finding solutions, we want to conclude with a few final advises: be ready to go back on your steps; and be always creative and flexible:

Sometimes it is better to avoid good suggestions and look for compromises!

6.3 Conclusions

In this chapter we presented a collection of simple rules-of-thumb that we learned in the previous four years from small successes and lots of mistakes. Each rule was justified with small examples; their only aim was to give a grasp on the set of problems that must be faced by a designer and implementor of an exporting procedure from a proof assistant to a distributed library of mathematical documents.

Even if many of the suggestions may appear obvious, the temptation of sticking to the internals of the system when exporting the information is great. The experiences of the HELM project strongly suggest that the design phase of the DTDs requires a lot of time and a lot of thought; but the result is worth the time spent, since wrong decisions will greatly slow-down further developments.

Only time and further experiences can have the final word on our guidelines and solutions. Still we believe we have provided at least a comprehensive list of the common problems that must be faced.

Part II

The HELM Proof Assistant

Chapter 7

Requirements and Architecture

In the first part of this dissertation, we described an architecture and several tools for the management of formal mathematical knowledge. Each tool was developed independently from the others and provides a particular service on the distributed library of mathematical theorems and definitions. A Web-interface was developed by the HELM team to give a unified access to the HELM and MoWGLI libraries and to all the available services. Moreover, since each tool was developed as a Web service, third-party developers can provide independent components to orchestrate our Web services and to provide higher-level services.

The thesis of the second part of this dissertation is that the architecture of the existing tools for mathematical reasonings, in particular proof assistants, can be greatly simplified by building the systems on top of the existing Web services for mathematical management, such as those developed in the first part of this dissertation. To prove the claim, we developed a proof assistant prototype, called GTOPLEVEL, greatly inspired by the Coq proof assistant [32]. The main characteristic of GTOPLEVEL is that it is totally compatible with the logical framework of Coq. The compatibility allows GTOPLEVEL to reuse and to augment the distributed library of HELM which was originally obtained by exporting to XML the Coq library. Due to the development of GTOPLEVEL, the HELM library is no more a static snapshot of the Coq library, but it becomes a "live" and evolving entity, raising new interesting challenges. Moreover, the kind of interaction with the library required to develop new knowledge poses completely new requirements with respect to usual Mathematical Knowledge Management activities, which are mainly only centered on digitalization, preservation, indexing, searching and rendering.

The rest of this chapter is devoted to establishing a more precise set of goals and requirements for GTOPLEVEL. The architecture of GTOPLEVEL is also sketched. In Chap. 8 we formally present the syntax, reduction, convertibility and typing rules of the Calculus of (Co)Inductive Constructions (CIC), which is the logic that, up to small variations, is shared by both Coq and GTOPLEVEL.

Chap. 9 describes the design and implementation of the kernel of GTOPLEVEL, which is the part of the system that deals with the typing rules of CIC and which is responsible to grant the correctness of the whole system. Since most of the implementation followed quite straightforwardly the theory of CIC, without trying to compete with Coq at the level of performances, the emphasis will be put on the didactic value of the system, which is one important requirement of GTOPLEVEL, and the peculiarity of the environment management: during type-checking of an object, the environment of GTOPLEVEL is not a read-only static entity; on the contrary, the minimal necessary environment is dynamically built on-demand by recursively type-checking parts of the distributed library.

Chap. 10 describes our implementation of the unification procedure for CIC terms, and the operations that are directly based over unification, which are refinement and ambiguous parsing. The implementation of unification and refinement are once again pretty usual. Ambiguous parsing, instead, is addressed with a new technique proposed by the author. Notice that there is no ambiguous parser in the Coq system, whose traditional interface is textual and ASCII-based: mathematical symbols are rarely used (since they cannot be expressed in plain ASCII) and they are never overloaded (the user has to explicitly specify what is the meaning of each symbol she is using). Ambiguous parsing, instead, plays an important role in Mathematical Knowledge Management, since the user interacts with several systems, libraries and interfaces and she cannot learn a different syntax for each of them; instead she wants to use the traditional mathematical notation she is used to, and she expects the systems to be able to understand her.

Chap. 11 describes the tactics implemented for GTOPLEVEL. Tactics are the high-level commands used by the user to progress in the proofs. All the tactics implemented so far are reimplementation of those available in Coq; the architecture is greatly simplified with respect to Coq, without any significative loss of performance or functionalities. The main element of novelty is an analysis of tactics aimed at providing metrics to compare tactics implementations. The metrics should be used to improve the quality of the generated terms.

Chap. 12 briefly describes the interface of GTOPLEVEL, which is obtained by combining together several of the components developed in the first part of this thesis.

In [111] an interesting extension of GTOPLEVEL is described: GTOPLEVEL is interfaced with H-BUGS, a network of distributed agents (called *tutors*) that try to automatically advance in the current proof, proposing asynchronous suggestions to the user. H-BUGS, that was developed as the master thesis of Stefano Zacchiroli [124], can be considered a successful validation of the architecture proposed in GTOPLEVEL: the modularity of GTOPLEVEL, its integration with the XML technologies and the simplicity of its interface easily allow both to quickly develop new tutors and to extend the system functionalities by interfacing it with more and more remotely running Web services.

7.1 Requirements

Developing a new proof assistant from scratch is a time-consuming and difficult task. For instance, the still very active development of the Coq proof assistant started in 1984[1]. Moreover, there is already a plethora[2] of proof assistants based on incompatible logics and often greatly under-developed. The abundance of proof assistants witnesses the level of interest in these tools and the existence of a whole lattice of systems characterized by different levels of expressivity, user friendliness and adequacy in solving ad hoc tasks. At the same time, this multitude of systems hinders the development of their own libraries: for real world developments, the level of coverage of the libraries, as well as their completeness in basic fields of mathematics, is at least as important as the expressivity of the logics. For instance, the system with the most complete mathematical library is Mizar, that is based on first order logic plus higher order schemes; Coq has a much more expressive logic, but a smaller mathematical library. Developing a new non trivial mathematical result in Mizar is often quicker than doing the same in Coq. Since writing a library is an extremely long task and since the existing systems are based on incompatible logics, every team developing a new system has to spend a lot of time in developing the system library before being able to practically compare the system with the alternative tools.

For all the previous reasons, our aim is not to develop a new proof assistant able to compete with the existing ones in performance. Moreover, since we do not want to have to write a new library from scratch, we need to adopt the logic of an already existing tool in order to share its library. Our aim is to provide a prototype of a new proof assistant to support our claims about the benefits of adopting a different architectural approach in the development of proof assistants. At the same time, we are also interested in other orthogonal features, such as the simplicity of its interfaces and the modularity of its design; these features are aimed at providing a canonical and easily understandable example of a proof assistant that can be used for didactics and that can be easily extended to test new ideas and new approaches. More explicitly, we commit ourselves to the following requirements:

Req1. GTOPLEVEL must be designed according to the document-centric model;

Req2. it must be able to consult and contribute to the distributed library of CIC mathematical documents of the HELM and MoWGLI projects;

[1]The original version, whose logical framework was the original Calculus of Constructions by Huet and Coquand, was called CoC. The name was changed into Coq only several year later.

[2]Among the existing higher-order proof assistants that have reached a non-neglectable number of users, there are Coq [32], PVS [104], Mizar [86], Isabelle [65], NuPrL [96], HOL-Light [64]. First order proof assistants are generally under-developed, since in the first order setting expressivity is sacrificed in favor of a greater level of automation, that naturally leads to the development of theorem-provers.

Req3. it must reuse and integrate as much as possible all the services developed in HELM and MoWGLI;

Req4. remote services must be implemented as Web services;

Req5. the system should constraint the user as much as possible in such a way that only reasonable proofs are generated; when this is not possible, the system should help the user in improving parts of the generated proofs;

Req6. the system must be developed as an acyclic graph of libraries; each new library must provide a new set of functionalities on top of the previous libraries; every library must have a coherent interface and should be usable independently of the GTOPLEVEL libraries built on top of it;

Req7. the interface of each library and of each module inside a library should be minimal and easily understandable; in particular, every function exported should have a clear and well-known theoretical description and it should be easily understandable by an undergraduate student;

Req8. the modules of the system should be as loosely coupled as possible, even at the expense of performances; the different functionalities of the system (e.g. reduction, convertibility, type-checking) should be as orthogonal as possible. For instance, changing the implementation of the reduction function to experiment new strategies should require only a local change in a module, without any consequence on the conversion test or the type-checking rules.

Requirements Req1, Req3 and Req4 allow us to use GTOPLEVEL as a test-bench for the HELM and MoWGLI architecture and they allow us to validate our claims. Thanks to the second requirement, we can reuse a huge library of lemmas and definitions and test our system on non-trivial mathematical theories. An alternative benefit of the second requirement is that there is now a tool independent from Coq to double-check the Coq proofs: the two systems can help each other by comparing their respective behavior on critical examples. Moreover, GTOPLEVEL represents a good platform to experiment new ideas before integrating them in Coq later on; the benefits of working in GTOPLEVEL and not in Coq itself should be the simplicity of the interfaces with respect to those of Coq (requirement Req7) and the fact that GTOPLEVEL's modules are loosely coupled (requirement Req8). A well known limitation of Coq, indeed, is the number of functions exported by each module, the complexity of their types and the lack of documentation. As a consequence, the learning curve for a beginner Coq developer is very steep even for a Ph.D. student. On the contrary, we want GTOPLEVEL to be easily extensible even by an undergraduate student who followed an undergraduate course in logic or an introductory course on the Curry-Howard isomorphism (requirement Req7). Finally, requirement Req5 is aimed at proving that, at

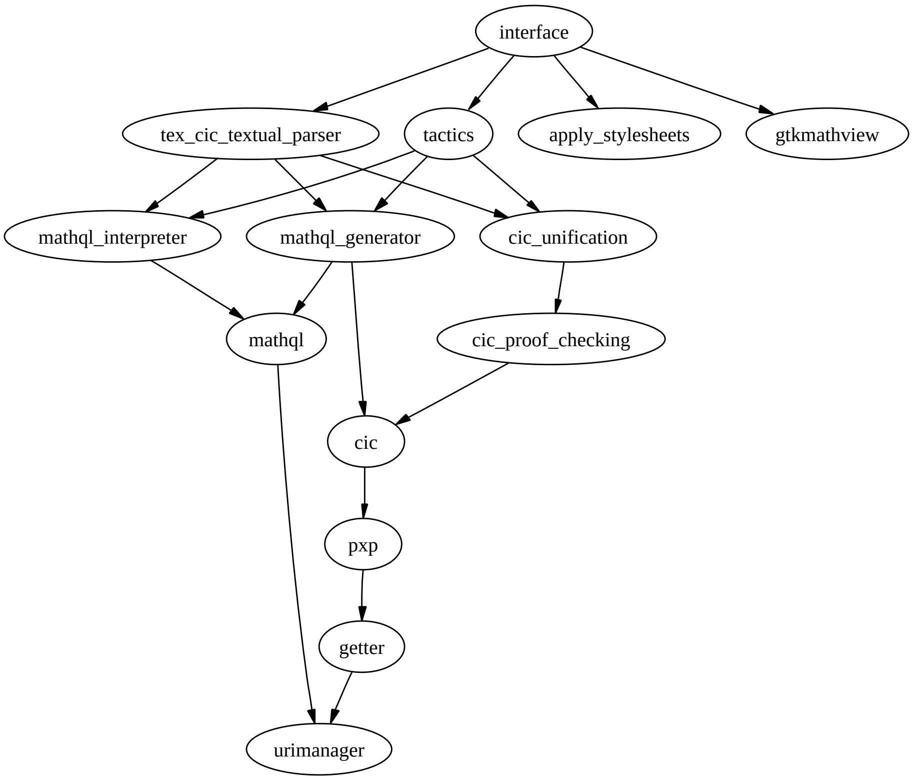

Figure 7.1: The implemented libraries.

least in some cases, it is possible to write tactics in such a way that the proofs they produce can be easily rendered in natural language, obtaining reasonable proofs. For the residual cases, the system should provide to the user direct access to the proof-object, in order to make her detect and improve the parts of the proofs she does not like. This functionality is not provided by Coq[3].

7.2 Architecture

In Fig. 7.1 we show the modules that we developed to implement GTOPLEVEL.

`urimanager` is a generic library implementing a data-structure to efficiently handle a stable set of URIs; testing the equality of two URIs in the data-structure is in O(1).

[3]Mizar users do not need this functionality since the Mizar system has no proof-object notion. Instead the user writes down a script that it is also natural enough to be considered as a mathematical proof. Since the script is completely written by hand, there is no need for a tool to improve it. Users of the systems of the Alfa family, instead, are used to interact directly with the proof-object without using any tactic. Thus the functionality is naturally provided to them by the systems.

`getter` is a stub library that forwards the requests to the HELM GETTER, that was described in Chap. 2.

`pxp` is a small library that adds a few functionalities to `PXP`, a third party library for XML parsing.

The `cic` library comprises the definition of the data-type that is the abstract syntax trees of CIC terms. Parsing functions from XML are also provided.

The `cic_proof_checking` library is the kernel of the proof-engine. It is the component responsible of the correctness of the whole system. All the bugs in the libraries the kernel does not depend on are harmless, since they will be detected by the `cic_proof_checking` library. The library will be described in details in Chap. 9, after the formal definition of CIC.

The `cic_unification` library implements unification and refinement as described in Chap. 10.

`mathql` and `mathql_interpreter` are two libraries developed by Ferruccio Guidi in his Ph.D. thesis [54]. They define and implement a low-level query language for mathematical metadata encoded in RDF. Actually, the `mathql_interpreter` is an in-process implementation of the Web service described in Chap. 5. An alternative and more loosely coupled implementation consists in a stub that just forwards the requests to the remove Web service.

`mathql_generator` is a module, developed in collaboration with Ferruccio Guidi and already described in Chap. 5, that generates queries that identify a suitable set of matching candidates given a pattern. For instance, the module is used by the `SearchPattern` tactic that looks for every theorem that can be successfully applied to progress in the proof. In order to avoid to perform unification over the whole distributed library of theorems, we generate a query using the `mathql_generator` library and we try unification only on the set of candidates that is the answer to the query.

The `tactics` library implements the core features of GTOPLEVEL as a proof-engine. Several tactics have already been implemented. Tactics are the subject of Chap. 11.

The `tex_cic_textual_parser` library implements a disambiguating parser from TEX expressions to CIC terms. The algorithm adopted for the disambiguating parser is described in Sect. 10.4.

The `apply_stylesheets` library is an in-process implementation of UWOBO, the stylesheet manager described in Chap. 3. The choice of an in-process implementation instead of a stub library is due to performance motivations. Indeed, the files that must be exchanged between UWOBO and GTOPLEVEL are whole theorems to be rendered. Since the theorems are encoded in XML, serialization and deserialization of the XML files is too heavy for interactive proving. An in-process implementation partially solves the problem. The stylesheets applied by the `apply_stylesheet` module are exactly those applied by UWOBO when the user browses the HELM Web interface [63].

The `gtkmathview` library is an OCaml binding of the GtkMathView Gtk widget for rendering MathML Presentation documents [98]. The widget also allows semantic (multiple) selection, that is the possibility of constraining the user to select only subexpressions that correspond to well-formed CIC subterms.

Finally, the library `interface` is the graphical interface for proof objects manipulation described in Chap. 12.

Other services and tools developed in HELM are also based on all the modules described so far. Moreover, we have already integrated in the graphical interface the possibility to invoke other remote Web services that are not shown in Fig. 7.1 (since they are not linked with GTOPLEVEL). The most interesting example is H-BUGS [111, 124], that is an architecture of distributed agents that suggest possible steps to advance in a proof. The H-BUGS agents, called tutors, are physically distributed Web services and some of them are implemented as clients of other HELM Web services, such as the HELM SEARCH-ENGINE.

According to the requirements for GTOPLEVEL, there is a clear distinction between the components and libraries that are logic dependent (i.e. whose implementation depends on the library `cic`) and those components that are logic independent. In particular, the library management, retrieval and rendering components are logic independent as expected:

- *Library management*: the `urimanager`, `getter` and `pxp` libraries can be used to locate and parse any document in a physically distributed library, creating the logical abstraction of a hierarchical local library.
- *Retrieval*: the `mathql` and `mathql_interpreter` libraries implement a query processor for RDF documents.
- *Rendering*: the `apply_stylesheet` library can be used to apply any (logic dependent) set of stylesheets to obtain content level documents and any set of logic independent stylesheets to obtain presentation level documents. `gtkmathview` provides rendering and interaction functionalities for MathML Presentation documents.

The functionalities that are logic dependent are indexing, proof checking and unification. The tactics of the proof engine are also obviously logic dependent. The case of indexing is particularly interesting, since it can be factorized into logic dependent and logic independent functionalities:

- *Indexing*: the `mathql_generator` library extracts the metadata associated to CIC terms and produces a set of high level constraints that are compiled to queries in the MathQL Level 1 language, as already described in Chap. 5. Only the phase of extraction of the metadata (the real indexing phase) is logic dependent.

The only library we have not discussed yet is the `tex_cic_textual_parser` library. Fig. 7.1 shows that the library depends both on logic independent libraries (that are mainly used to generate the parsing trees) and on logic dependent libraries (that are mainly used to prune the parsing trees that are not well typed). In Sect. 10.4, where we describe our algorithm for parsing ambiguous mathematical expressions, we will fix as a fundamental requirement the separation of the logic dependent part from the logic independent part.

7.3 Comparisons with Related Work

The motivations and requirements for the development of GTOPLEVEL are pretty different from those of the mainstream proof assistants. Indeed, at least for now GTOPLEVEL is intended to be just a prototype to experiment with the problems and solutions that arise from the integration of MKM technologies with interactive theorem proving. In particular

- We are not aiming at efficiency (as the PVS and Coq proof assistants, that are mainly used for industrial verifications of large programs).
- We are not developing a proof assistant to experiment with more than one logic (as the Isabelle generic proof assistant or, to a lesser extent, the Yarrow proof assistant [125]). As a consequence the architectural choices have been similar to those of the "traditional" proof assistants based on the Curry-Howard isomorphism (e.g. Coq and NuPrL) instead of those of the generic proof assistants (e.g. Isabelle and MetaPrL).
- We are not investigating yet variations of a particular logic or alternative solutions for its implementation (as the no longer developed Lego proof assistant [72]). Notwithstanding this, we are really interested in keeping the interfaces of the modules of our proof assistant as simple as possible, completely hiding all the implementation details in order to expose only functions characterized by a well-known theoretical specification. This requirement, together with the modularity requirement, are aimed at being able in the future to experiment with new solutions before applying them to the mainstream proof assistants.
- We are interested in the didactic value of our GTOPLEVEL for teaching students the basis of reasoning tools implementations, but we are not interested in the usage of GTOPLEVEL for teaching how to develop formal mathematics (as the PhoX proof assistant [101]).

GTOPLEVEL shares with the Alfa family of proof assistant the (basically) document centric approach, that involves direct manipulation of proof objects and the interest in natural language rendering of the generated proofs. Notwithstanding this, the proof assistants of the Alfa family can hardly be considered as mainstream, since the number of users and the number of developed libraries is actually quite small.

Attempting a comparison with the requirements of the Mizar proof assistant is not simple. Indeed, the Mizar team is probably the proof assistant development group more involved in MKM problematics since its beginning. Library management problematics, as well as rendering problematics, indexing and retrieval, and the adoption of human readable formats are the strong characteristic of Mizar. Moreover, there exist plans to provide the system with XML exportation and importation modules, although the task is pretty complex since the system is not based on the Curry-Howard isomorphism and it has no proof objects. Notwithstanding all this, there exist deep differences between the two systems. The main one is that GTOPLEVEL is meant to be an interactive system. For instance, the tactics of the system often ask the user about missing information that was not provided, suggesting the possible answers if possible. On the contrary, Mizar is a sort of compiler whose input is a mathematical proof and whose output is a list of deduction errors, that are inference steps that cannot be deduced by the system. Thus, all the requirements of interactive development, such as semantic selection, are lifted. Moreover, our interest in systems that are based on the Curry-Howard isomorphism is genuine, since proof objects in a well defined logic are the only kind of objects that allow completely independent development of document centric proof assistants.

Chapter 8

The Calculus of (Co)Inductive Constructions

8.1 The Calculus

The Curry-Howard isomorphism is a formal correspondence between several typed functional calculi — such as the λ-calculus — and the natural deduction formulation of some logics. The two simplest systems related by the isomorphism are propositional logic and the simply typed λ-calculus [53]: the types of the λ-calculus correspond to the propositions; the λ-terms correspond to proof trees; cut elimination corresponds to the image of β-reduction. A term t *inhabits* a type T if t has type T. The system is consistent if there exists a particular type $\perp$ which is not inhabited.

Logics — and consequently the corresponding calculi — can be classified according to their expressivity, which can be augmented in two ways: by introducing new objects (new types); or by augmenting the algorithmic expressivity. For instance, predicate calculus is obtained from propositional logic by introducing new objects; higher order logics are derived from first order logic by introducing new abstractions. The first possibility allows, for instance, to add directly new inductive and coinductive data types, by giving their constructors (observations in the coinductive case) and their destructors, i.e. their (co)recursive principles [52].

The second possibility allows to add new terms and types of higher and higher degree of dependency, which is reflected in the level of polymorphism of the calculus: while in the simply typed λ-calculus the only abstractions allowed are those of terms over terms (λ-abstraction), in the Barendregt's λ-cube calculi [15] types and terms can freely depend on each other.

Another characteristic used to discriminates the logical systems is *predicativity*. In predicative systems there exists a partial order among the types; abstractions over greater types are not allowed. In impredicative systems every abstraction is allowed. Predicativity is strictly linked to the notion of constructivism since, in impredicative systems, functions can be applied to types which are not yet defined when the function is made available. On the contrary, predicativity does not allow the definition of an object d whose type refers to a collection D of which d is an element, avoiding cycles creation. Even though it is possible to define impredicatives consistent systems, it is quite easy to obtain inconsistent systems[1].

Two logical frameworks that greatly influenced CIC were proposed at the beginning of the seventies. The first one, Martin-Löf's *Intuitionistic Type Theory*, is a predicative system which was developed to give a new constructive foundation to mathematics. Its main characteristic is the presence of a rather expressive system of inductive types. The second one, Girard's polymorphic λ-calculus F_ω, is a functional impredicative calculus that corresponds through the Curry-Howard isomorphism to higher order logic. Inductive types are not primitive constructions in the system, but can be simulated thanks to the polymorphism. For instance, Church's numeral are used to represent natural numbers.

In 1985 Coquand presents a first version of the *Calculus of Constructions* (CoC), obtained by combining the characteristic of both systems in a higher order extension of the language of the Automath system. The system is already very expressive, but inductive types cannot be defined directly, but only uniformly encoded using impredicativity. As a consequence, implementation efficiency is lowered and it is not possible to have strong elimination, i.e. defining a new type by cases over a given inductive term. In 1990 Luo [76] proposes a consistent extension of the system, the *Extended Calculus of Constructions*, that is impredicative over propositions and predicative over data types. The formalism is further extended in 1989 by T. Coquand and C. Pauline-Mohring with primitive inductive types, obtaining the *Calculus of Inductive Constructions* [33].

Later on, several modifications to CIC were proposed to enhance the Coq system, which is the most relevant implementation of CIC. The most important modifications are mutual inductive definitions, coinductive types and coinductive elimination principles and two new schemes of primitive operators for case analysis over inductive terms and recursive well-founded definition of functions. Since these extensions to the CIC were given independently, to the author knowledge there exists no single paper were it is possible to find the syntax, reduction and typing rules of the whole calculus, as implemented in Coq. The three most relevant works are Werner's Ph.D. dissertation [118], which proves the logical consistency of the original version of CIC, the paper of Gimenez [52] that introduces case analysis and (co)recursive definitions, and the habilitation dissertation [99] of Christine Paulin-Mohring that reviews the rules related to mutual inductive and coinductive definitions. The recent Ph.D. dissertation of Alexandre Miquel [84] is also very interesting, since it is the first to introduce set-theoretical models for restrictions of the calculus.

In the next sections we describe the syntax, reduction, convertibility and typing rules of the version of CIC that is implemented in Coq V7.4. The rules have been assembled from different sources, comprising the code of Coq. They are the base set of rules that are implemented in GTOPLEVEL.

[1]For instance, Coquand in [33] shows that the Girard paradox can be obtained in several impredicative systems.

The given rules are surely not stricter than the rules of Coq, since GTOPLEVEL can type-check the whole set of theorems exported from Coq. It may be possible for the rules to be looser than the one of Coq or even inconsistent: proving the consistency of the whole set of rules would provide enough material for another Ph.D. dissertation. Nevertheless the possibility of an inconsistent set of rules is unlikely, the most complex and less documented rules having been deeply compared with the ones implemented in the Coq system. I am also grateful to Hugo Herbelin that, through several private communications, improved a lot my understanding of the whole system.

8.1.1 The Syntax

With a small notational abuse, in the whole chapter all the sequences indexed from 1 to n will also comprise the empty sequence, unless stated otherwise.

- Let $\mathcal{V}$ be a denumerable family of variable names, $x, y, z, \ldots \in \mathcal{V}$.
- Let $\mathcal{C}$ be a denumerable family of constant names, $c, c_1, \ldots, c_n \in \mathcal{C}$.
- Let $\mathcal{I}$ be a denumerable family of (co)inductive type names, $i, i_1, \ldots, i_n \in \mathcal{I}$.
- Let $\mathcal{K}$ be a denumerable family of (co)inductive constructor names, $k, k_1, \ldots, k_n, k_1^1, \ldots, k_{n_1}^1, \ldots, k_1^m, \ldots, k_{n_m}^m \in \mathcal{K}$.

Well formed terms (represented by t,f,u,T,U,N,M) are inductively defined in Fig. 8.1.1, where $m, n_1, \ldots, n_m, j$ and l are positive integers, l is lesser or equal to the number of functions defined in its (CO)FIX constructor and $\overline{x}$ is a possibly empty sequence of identifier whose length is $|\overline{x}|$. $s, s_1, \ldots, s_n$ are served for the sorts. As usual, $\Pi x : T_1.T_2$ is abbreviated in $T_1 \to T_2$ when x is not a free variable in T_2. The inductive type i in the CASES constructor is redundant, since distinct inductive types have distinct constructors; it is given to improve clarity. Note that, in CIC, there is no syntactic distinction between terms and types.

The concrete syntax differs from the abstract syntax only from the usage of parentheses to disambiguate expressions, provided the usual convention that application is left associative and λ-abstraction is right associative.

x is bound in the bodies of the λ-abstraction, i.e. after the dots. Similarly, in the case of (co)recursive function definitions, x is bound in the body of the defined functions, i.e. after the assignment symbols. Variables occurring in the left hand side of a case analysis branch are bound in the right hand side, i.e. after the thick arrows. Free variables, substitution and α-conversion are defined as usual. Terms are syntactically identified up to α-conversion.

Some syntactic restrictions of terms will be required in the typing rules. The *arities* are the normal forms of the types of inductive types; they are ranged over by $A, B, A_1, \ldots, A_n$. The

t	$::=$	x	identifiers
	$\mid$	c	constants
	$\mid$	i	(co)inductive types
	$\mid$	k	(co)inductive constructors
	$\mid$	$\mathit{Set} \mid \mathit{Prop} \mid \mathit{Type}(j)$	sorts
	$\mid$	$t\ t$	application
	$\mid$	$\lambda x : t.t$	λ-abstraction
	$\mid$	$\mathit{let}\ x := t\ \mathit{in}\ t$	local definitions
	$\mid$	$\Pi x : t.t$	dependent product
	$\mid$	$< t >$ CASES$_i$ t OF	case analysis
		$\quad k_1\ \overline{x} \Rightarrow t \mid \ldots \mid k_n\ \overline{x} \Rightarrow t$	
		END	
	$\mid$	FIX$_l$ $\{x/n_1 : t := t\ ;\ \ldots\ ;\ x/n_m : t := t\}$	well-founded recursive definitions
	$\mid$	COFIX$_l$ $\{x : t := t\ ;\ \ldots\ ;\ x : t := t\}$	corecursive definitions

Table 8.1: CIC terms syntax

constructor types for i — ranged over by $C(i), C_1(i), \ldots, C_n(i)$ or simply by $C, C_1, \ldots, C_n$ — are the normal forms of the types of the constructors of the inductive type i. Arities and constructor types are defined as follows

$$
\begin{array}{rcl}
A & ::= & s \mid \Pi x : t.A \\
C(i) & ::= & (i\ t_1 \ldots t_h) \\
 & \mid & \Pi x : T.C(i)
\end{array}
$$

provided the following additional conditions hold:

1. every inductive type that is mutually defined with i, such as i itself, cannot occur in $t_1, \ldots, t_h$.
2. $i \neq x$
3. If i depends on k parameters and if its arity is $\Pi x_1 : T_1 \ldots \Pi x_l : T_l.s$, then $h = k + l$
4. i occurs only strictly positively in T

where i occurs only strictly positively in T if

- i and every inductive type that is mutually defined with i do not occur in T *or*

- $T = \Pi x : T_1.T_2$ and i and every inductive type mutually defined with i do not occur in T_1, while i occurs only strictly positively in T_2 *or*

- $T = (i'\ t_1 \ldots t_n)$ and i' is i or an inductive type mutually defined with i and i and every inductive type mutually defined with i do not occur in $t_1 \ldots t_n$ *or*

- $T = (i'\ t_1 \ldots t_n)$ and i' is not i nor it is mutually defined with i and i' is an inductive type that is not mutually defined with other inductive types and that depends on $m < n$ parameters $p_1, \ldots, p_m$; moreover i and every inductive type mutually defined with i does not occur in $t_{m+1}, \ldots, t_n$ and its occurs only weakly positively in $C^{i'}\{t_1/p_1\ ;\ \ldots\ ;\ t_m/p_m\}$ for each type $C^{i'}$ of every constructor of i'

where i occurs only weakly positively in C, which is the type of a constructor of i', if

- $C = (i'\ t_1 \ldots t_n)$ *or*

- $C = T \rightarrow C'$ and i occurs only weakly positively in C' and i occurs only strictly positively in T after substituting i' with a fresh variable x *or*

- $C = \Pi x : T.C'$ and i occurs only weakly positively in C' and i and every inductive type mutually defined with i do not occur in T after substituting every application of i' to the actual arguments for the inductive definition with a fresh variable y

8.1.2 Reductions and Convertibility

Before introducing the reductions rules of CIC, it is necessary to define *contexts* and *environments*.

Contexts and Environments

A *context* Γ is an ordered list of variable declarations $(x_i : T_i)$ or variable definitions $x_i : T_i := t_i$ where every x_i is unique in the list. $(x : T) \in \Gamma$ (respectively $(x : T := t) \in \Gamma$) means that $(x : T)$ (respectively $(x : T := t)$) is an element of Γ. $x \in \Gamma$ stands for $\exists T.((x : T) \in \Gamma \vee (x : T := t) \in \Gamma)$. $[]$ is the empty context.

An *environment* E is an ordered list of definitions of constants and mutually defined (co)inductive types. It is written as $[\omega_1, \ldots, \omega_n]$ where ω_j may be either $c : T := t$ (the definition of a constant c) or

$$\begin{array}{l}\Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\ \quad \{i_1 : A_1 := \{k_1^1 : C_1^1\ ;\ \ldots\ ;\ k_{n_1}^1 : C_{n_1}^1\}\ ; \\ \quad \ldots\ ; \\ \quad i_m : A_m := \{k_1^m : C_1^m\ ;\ \ldots\ ;\ k_{n_m}^m : C_{n_m}^m\} \\ \quad \}_{(\mathrm{CO})\mathrm{IND}}\end{array}$$

(the definition of a block of mutually defined types $i_1, \ldots, i_m$, that depend on the parameters $x_1, \ldots, x_h$ and whose constructors are $k_1^1, \ldots, k_{n_m}^m$). $x_1, \ldots, x_h$ are bound in $A_1, \ldots, A_m, C_1^1, \ldots, C_{n_m}^m$; $i_1, \ldots, i_m$ may occur in $C_1^1, \ldots, C_{n_m}^m$. $(\mathrm{CO})\mathrm{IND}$ must be IND for inductive types and it must be COIND for coinductive types.

The names of the constants defined in an environment must be distinct. The same holds for the names of mutually defined types and those of their constructors. The notations for the empty environment and the "belongs-to" relation are the same as those introduced for contexts.

The rules to decide whether a context or environment is well-typed and those for extending a context or environment are given in Sect. 8.1.3, since they are mutually defined with the typing rules of CIC terms. The mapping with the Coq and GTOPLEVEL concrete syntaxes should be pretty obvious.

β-reduction, δ-reduction, ι-reduction, ζ-reduction and unfolding

We define five reduction rules: β-reduction, ι-reduction, δ-reduction, ζ-reduction, and unfolding. The first three rules correspond to cut elimination for implication and inductively defined operators. The third and fourth one are pretty usual in the semantics of programming languages, and simply state that a definiendum can be replaced by its definiens. The fifth one is sometimes found in the semantics of lazy programming languages, even if in less restricted ways.

Definition 8.1 (β-reduction) *This is the standard λ-calculus reduction rule $(\lambda x : T.M)N \rhd_\beta M\{N/x\}$, provided the side condition — often left implicit — that every free variable of N must not be captured in $M\{N/x\}$.*

Definition 8.2 (ζ-reduction) *This rules states that a local definition can be simply replaced by the term in which the definition is in scope, after having replaced every definiendum with the definiens: let $x := t$ in $M \rhd_\zeta M\{t/x\}$.*

Definition 8.3 (ι-reduction) *It is similar to β-reduction for (co)inductive types. If the declaration*

$$\begin{array}{l}\Pi x_1 : U_1 \ldots \Pi x_h : U_h.\\ \quad \{i_1 : A_1 := \{k_1^1 : C_1^1 \;;\; \ldots \;;\; k_{n_1}^1 : C_{n_1}^1\}\;;\\ \quad \ldots\;;\\ \quad i_m : A_m := \{k_1^m : C_1^m \;;\; \ldots \;;\; k_{n_m}^m : C_{n_m}^m\}\\ \quad \}_{(\mathrm{CO})\mathrm{IND}}\end{array}$$

occurs in the environment E, then

$$\begin{array}{ll}< T > \mathrm{CASES}_{i_l}\ (k_j^l t_1 \ldots t_h t'_1 \ldots t'_{n_j})\ \mathrm{OF} & \\ \quad k_1^l\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_n^l\ \overline{x^n} \Rightarrow f_n & \rhd_\iota f_j\{t'_1/x_1 \;;\; \ldots \;;\; t'_{n_j}/x_{n_j}\}\\ \mathrm{END} & \end{array}$$

where $l \in \{1, \ldots, m\}$, $\forall l \in \{1, \ldots, n_j\}.|x_l| = n_l$ *and* $\overline{x^j} = (x_1, \ldots, x_{n_j})$

Definition 8.4 (δ-reduction) *This reduction step is also standard and corresponds to ζ-reduction for global definitions and definitions in the context: if the declaration* $c : T := t$ *belongs to the environment* E *or to the context* Γ, *then* $c \rhd_\delta t$.

Definition 8.5 (Unfolding) *It exists in two variants, one for recursion and one for corecursion. Let* FIX^j *be a syntactic shorthand for*

$$\mathrm{FIX}_j\ \{x_1/n_1 : T_1 := f_1\ ;\ \ldots\ ;\ x_m/n_m : T_m := f_m\}$$

and COFIX^j *a syntactic shorthand for*

$$\mathrm{COFIX}_j\ \{x_1 : T_1 := f_1\ ;\ \ldots\ ;\ x_m : T_m := f_m\}$$

$$(\mathrm{FIX}^j\ t_1 \ldots t_{n_j-1} k) \rhd_{\mathrm{FIX}} (f_j\{\mathrm{FIX}^1/x_1\ ;\ \ldots\ ;\ \mathrm{FIX}^m/x_m\}\ t_1 \ldots t_{n_j-1} k)$$

and

$$\begin{array}{l} < T > \mathrm{CASES}_i\ (\mathrm{COFIX}^j t_1 \ldots t_h)\ \mathrm{OF} \\ \quad k_1\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_n\ \overline{x^n} \Rightarrow f_n \qquad \rhd_{\mathrm{COFIX}} \\ \mathrm{END} \\ \quad < T > \mathrm{CASES}_i\ (f_j\{\mathrm{COFIX}^1/x_1\ ;\ \ldots\ ;\ \mathrm{COFIX}^m/x_m\}\ t_1 \ldots t_h)\ \mathrm{OF} \\ \qquad k_1\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_n\ \overline{x^n} \Rightarrow f_n \\ \quad \mathrm{END} \end{array}$$

Note that the two rules are restriction of the usual unfolding rule. The restrictions, which are different for the two rules, have dual roles and, combined with the GD *(Guarded by Destructors) condition and its dual* GC *(Guarded by Constructors) condition — defined respectively at page 114 and 112 — force the termination of the reduction relation, i.e. they grant the strong normalization property of CIC. The latter property implies that, after reduction, each term is in normal form. Since the normal form*[2] *of an inhabitant of an inductive type must be the head of a constructor of the type and since the inductive type* FALSE *has no constructors, then strong normalization of CIC implies logic consistency (i.e.* FALSE *is not inhabited).*

Convertibility

The convertibility relation $=_{\beta\delta\iota\zeta}$ over CIC terms is the reflexive, symmetric, transitive and context-aware closure[3] of the reduction rules $\rhd_\beta, \rhd_\delta, \rhd_\iota, \rhd_\zeta, \rhd_{\mathrm{FIX}}$ e $\rhd_{\mathrm{COFIX}}$.

The strong normalization property[4] implies the decidability of convertibility. The latter property implies decidability of type-checking, which will be defined in the next section.

[2]Weak head normal form is sufficient for the reasoning to hold.

[3]The context-aware closure is the standard one, which asserts that $=_{\beta\delta\iota\zeta}$ is a congruence relation w.r.t. the CIC operators. It is unrelated to the CIC contexts previously defined.

[4]B. Werner in [118] proves strong normalization for an older and more primitive version of CIC after extending the convertibility relation to include also η-conversion. The result is strongly based on the work of Geuvers [50]

8.1.3 The typing rules

We give the typing rules of CIC by mutually defining two judgment. The first one, $E[\Gamma] \vdash t : T$, asserts that t, in an environment E and a context Γ that depends on E, has type T. The second one, $\mathcal{WF}(E)[\Gamma]$, asserts that E is a valid environment — i.e. it comprises only well-typed constant and (co)inductive types definitions — and that Γ is a valid context in E — i.e. it comprises only variable declarations whose type is well-typed.

The Rules of the Calculus of Constructions

WF-Empty

$$\mathcal{WF}([])[[]]$$

WF-Var

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathbf{Prop}, \mathbf{Set}, \mathbf{Type}(i)\} \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma, (x : T)]}$$

WF-Const

$$\frac{E[[]] \vdash T : s \qquad E[[]] \vdash t : T \qquad c \notin E}{\mathcal{WF}(E; c : T := t)[[]]}$$

Ax-Prop

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathbf{Prop} : \mathbf{Type}(n)}$$

Ax-Set

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathbf{Set} : \mathbf{Type}(n)}$$

Ax-Acc

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad n < m}{E[\Gamma] \vdash \mathbf{Type}(n) : \mathbf{Type}(m)}$$

Var

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x : T) \in \Gamma \ \text{ or } \ (x : T := t) \in \Gamma}{E[\Gamma] \vdash x : T}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c : T := t) \in E}{E[\Gamma] \vdash c : T}$$

Prod-SP

$$\frac{E[\Gamma] \vdash T : s_1 \qquad E[\Gamma, (x : T)] \vdash U : s_2 \qquad s_1 \in \{\mathbf{Prop}, \mathbf{Set}\} \ \text{ or } \ s_2 \in \{\mathbf{Prop}, \mathbf{Set}\}}{E[\Gamma] \vdash \Pi x : T.U : s_2}$$

Prod-T

$$\frac{E[\Gamma] \vdash T : \mathbf{Type}(n_1) \qquad E[\Gamma, (x : T)] \vdash U : \mathbf{Type}(n_2) \qquad n_1 \leq n \qquad n_2 \leq n}{E[\Gamma] \vdash \Pi x : T.U : \mathbf{Type}(n)}$$

Lam

$$\frac{E[\Gamma, (x : T)] \vdash t : U \qquad E[\Gamma] \vdash \Pi x : T.U : s}{E[\Gamma] \vdash \lambda x : T.t : \Pi x : T.U}$$

LetIn

$$\frac{E[\Gamma] \vdash t_1 : T_1 \qquad E[\Gamma, (x : T_1 := t_1)] \vdash t_2 : T}{E[\Gamma] \vdash \mathtt{let}\ x := t_1\ \mathtt{in}\ t_2 : T\{t_1/x\}}$$

App

$$\frac{E[\Gamma] \vdash t : \Pi x : U.T \qquad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{u/x\}}$$

Conv

$$\frac{E[\Gamma] \vdash T_1 : s \qquad E[\Gamma] \vdash t : T_2 \qquad E[\Gamma] \vdash T_1 =_{\beta\delta\iota\zeta} T_2}{E[\Gamma] \vdash t : T_1}$$

A less tight convertibility rule is admissible for CIC. The rule is obtained from the Conv rule by replacing convertibility with an order relation $\leq_{\beta\delta\iota\zeta}$ that takes in account also cumulativity between universes:

Definition 8.6 (Of $\leq_{\beta\delta\iota\zeta}$)
Let E *be an environment,* Γ *be a context and* T_1, T_2 *two terms.* $E[\Gamma] \vdash T_1 \leq_{\beta\delta\iota\zeta} T_2$ *iff*

- $T_1 =_{\beta\delta\iota\zeta} T_2$ *or*
- $T_1 = \mathbf{Type}(i)$ *and* $T_2 = \mathbf{Type}(j)$ *and* $i \leq j$ *or*
- $T_1 = \mathbf{Prop}$ *and* $T_2 = \mathbf{Type}(i)$ *or*
- $T_1 = \mathbf{Set}$ *and* $T_2 = \mathbf{Type}(i)$ *or*
- $T_1 = \Pi x : S_1.T_1'$ *and* $T_2 = \Pi x : S_2.T_2'$ *and* $S_1 =_{\beta\delta\iota\zeta} S_2$ *and* $T_1' \leq_{\beta\delta\iota\zeta} T_2'$

In the rest of this dissertation we will not consider the extended convertibility rule that uses $\leq_{\beta\delta\iota\zeta}$, considering only the simpler case of the Conv rule that use $=_{\beta\delta\iota\zeta}$. Nevertheless, all the results presented can be improved to cover also extended convertibility.

The Rules About (Co)Inductive Types

To shorten the rules, in the following rules the symbol $\star$ stands for the following mutual declaration of (co)inductive types:

$$\begin{array}{l} \Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\ \quad \{ i_1 : A_1 := \{k_1^1 : C_1^1\ ;\ \ldots\ ;\ k_{n_1}^1 : C_{n_1}^1\}\ ; \\ \quad \ldots\ ; \\ \quad i_m : A_m := \{k_1^m : C_1^m\ ;\ \ldots\ ;\ k_{n_m}^m : C_{n_m}^m\} \\ \quad \}_{(\mathrm{CO})\mathrm{IND}} \end{array}$$

WF-Ind

$$\frac{\begin{array}{l}(E[[]] \vdash (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j : s_j)_{j=1,\ldots,m} \\ (E[[y_1 : A_1^* \, ; \; \ldots \, ; \; y_m : A_m^*]] \vdash C_l^j\{y_1/i_1 \, ; \; \ldots \, ; y_m/i_m\} : s_l^j)_{j=1,\ldots,m \, ; \, l=1,\ldots,n_j} \\ i_1, \ldots, i_m, k_1^1, \ldots, k_{n_m}^m \notin E \\ i_1, \ldots, i_m, k_1^1, \ldots, k_{n_m}^m \text{ distinct}\end{array}}{\mathcal{WF}(E, \star)[[]]}$$

where $A_j^* = \Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j$ for $j \in \{1, \ldots, m\}$
and the following syntactic restriction holds: in $C_1^1, \ldots, C_{n_m}^m$ the first h arguments of the applications whose head is one of the inductive types $i_1, \ldots, i_m$ must be $x_1, \ldots, x_h$

Ind

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \star \in E}{E[\Gamma] \vdash i_j : \Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j} \qquad \text{if } j \leq m$$

Constr

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \star \in E}{E[\Gamma] \vdash k_l^j : \Pi x_1 : U_1 \ldots \Pi x_h : U_h.C_l^j} \qquad \begin{array}{ll}\text{if} & j \leq m \\ \text{and} & l \leq n_j\end{array}$$

Case

$$\frac{\begin{array}{l}\star \in E \qquad E[\Gamma] \vdash t : (i_l \; t_1 \ldots t_h \; t_1' \ldots t_s') \\ E[\Gamma] \vdash T : A \qquad [(i_l \; t_1 \ldots t_h) : A_l\{t_1/x_1 \, ; \; \ldots \, ; \; t_h/x_h\}|A]_{i_l} \\ (E[\Gamma] \vdash f_j : \Delta\{C_j^l\{t_1/x_1 \, ; \; \ldots \, ; \; t_h/x_h\}, T, (k_j^l \; t_1 \ldots t_h)\})_{j=1,\ldots,n_l}\end{array}}{E[\Gamma] \vdash \left(\begin{array}{l} < T > \mathrm{CASES}_{i_l} \; t \; \mathrm{OF} \\ \quad k_1^l \; \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_{n_l}^l \; \overline{x^n} \Rightarrow f_{n_l} \\ \mathrm{END}\end{array}\right) : (T \; t_1' \ldots t_s' \; t)}$$

Fix

$$\frac{\begin{array}{l}(E[\Gamma] \vdash T_l : s_l)_{l=1,\ldots,m} \\ (E[\Gamma, (x_1 : T_1), \ldots, (x_m : T_m)] \vdash f_l : T_l)_{l=1,\ldots,m} \\ (GD\{x_1, \ldots, x_m, n_1, \ldots, n_m, x, [], T\})_{l=1,\ldots,m}\end{array}}{E[\Gamma] \vdash \mathrm{FIX}_j \; \{x_1/n_1 : T_1 := f_1 \, ; \; \ldots \, ; \; x_m/n_m : T_m := f_m\} : T_j}$$

where $f_j = \lambda y_1 : T_1' \ldots \lambda y_{n_j - 1} : T_{n_j-1}'.\lambda x : (i \; t_1 \ldots t_n)T$

CoFix

$$\frac{\begin{array}{l}(E[\Gamma] \vdash T_l : s_l)_{l=1,\ldots,m} \\ (E[\Gamma, (x_1 : T_1), \ldots, (x_m : T_m)] \vdash f_l : T_l)_{l=1,\ldots,m} \\ (GC_0\{x_1, \ldots, x_m, f_l\})_{l=1,\ldots,m}\end{array}}{E[\Gamma] \vdash \mathrm{COFIX}_j \; \{x_1 : T_1 := f_1 \, ; \; \ldots \, ; \; x_m : T_m := f_m\} : T_j}$$

We have not yet defined the conditions GD, GC and $[T : A|B]$, and the $\Delta\{.,.,.\}$ operator:

Definition 8.7 ($\Delta\{.,.,.\}$) *Let $\mathcal{T}$ be the set of terms of CIC. We define* $\Delta : \{C/C$ *is a constructor type for* $i\} \times \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ *by structural induction over the first argument:*

$$\begin{array}{lcl} \Delta\{(i\ t_1 \ldots t_n), T, t\} & = & (T\ t_1 \ldots t_n\ t) \\ \Delta\{(x : T)C, T, t\} & = & \Pi x : t.\Delta\{C, T, (c\ x)\} \end{array}$$

In CIC the elimination of a (co)inductive type is not always allowed. The first motivation is the distinction between computationally relevant parts of the proof (the subterms of sort SET) and those parts which have no computational content (the subterms of sort PROP). The distinction is crucial for code exportation: the computationally irrelevant subterms are completely forgot during the automatic exportation of code. Thus, eliminating a non-informative type to obtain an informative type must not be allowed, unless there is only one way in which the elimination can be performed: this last case corresponds to the elimination of a singleton type (which has only one constructor whose parameters are all non-informative).

The second motivation to forbid elimination of (co)inductive types is due to the existence of paradoxes that arise in impredicative systems where elimination is not constrained. Coquand, in [33], shows that in CIC the elimination of an impredicative type to obtain a predicative type allows to encode the Burali-Forti paradox in type theory. In [118], Werner shows how to avoid the Coquand paradox restricting to *strong elimination* the elimination of an inductive type to obtain a new type. Strong elimination is the name given to the elimination of *small* inductive types in SET, where a small inductive type is defined as follow:

Definition 8.8 (Small (co)inductive type) *A constructor of a (co)inductive type is* small *if its type is* $\Pi x_1 : T_1 \ldots \Pi x_h : T_h.(i\ t_1 \ldots t_n)$ *where every type* $T_1, \ldots, T_h$ *has type* SET *or* PROP. *A (co)inductive type is* small *when its constructors are all small.*

Definition 8.9 (The condition for the elimination of (co)inductive types) *The condition* $[T : A|B]_i$ *is the smallest relation such that:*

Prod

$$\frac{[(T\ x) : A'|B']_i}{[T : \Pi x : A.A'|\Pi x : A.B']_i}$$

Prop-Prop

$$[T : \textbf{\textit{Prop}}|T \to \textbf{\textit{Prop}}]_i$$

Prop-Singleton

$$\frac{i \textit{ is a singleton type}}{[T : \textbf{\textit{Prop}}|T \to \textbf{\textit{Set}}]_i}$$

Set-PS

$$\frac{s \in \{\textbf{\textit{Prop}}, \textbf{\textit{Set}}\}}{[T : \textbf{\textit{Set}}|T \to s]_i}$$

Set-Small (strong elimination)

$$\frac{\mathrm{i} \textit{ is a small (co)inductive type} \qquad s = \textbf{\textit{Type}}(\mathrm{i})}{[T : \textbf{\textit{Set}} | T \rightarrow s]_{\mathrm{i}}}$$

Type

$$\frac{s \in \{\textbf{\textit{Prop}}, \textbf{\textit{Set}}, \textbf{\textit{Type}}(j)\}}{[T : \textbf{\textit{Type}}(l) | T \rightarrow s]_{\mathrm{i}}}$$

The next condition that we introduce is the Guarded by Destructors condition (GD). GD is introduced to grant strong normalization when reduction involves recursive functions (which must explicitly state what is their decreasing argument). The GD condition is satisfied when the actual value of the decreasing parameter for every recursive call is "smaller" than the one of the previous iteration; in this context an argument is said to be *smaller* than another argument t when it is obtained by applying one or more destructors to t. Strong normalization derives from the combination of GD, the constraints that inhibit unfolding and the invariant that there exists no term that inhabits an inductive type and in which an infinite number of constructors occur.

Definition 8.10 (The GD(Guarded by Destructors) condition)
The condition $\mathrm{GD}\{x_1, \ldots, x_m, n_1, \ldots, n_m, x, l, T\}$ *— where* $x_1, \ldots, x_m$ *are the mutual functions recursively defined,* $n_1, \ldots, n_m$ *are the indexes of their decreasing parameters,* l *is the initially empty set of variables that are allowed as actual arguments for the decreasing parameters,* x *is the index of the recursive argument of the recursively defined function and* T *is the term that must be checked — is defined by structural induction over* T *according to the following rules*[5]*. The* GD *condition holds*

- *if* $T = \Pi z : M.N$, $\mathrm{GD}\{l, M\}$ *holds and* $\mathrm{GD}\{l, N\}$ *holds*
- *if* $T = \lambda z : M.N$, $\mathrm{GD}\{l, M\}$ *holds and* $\mathrm{GD}\{l, N\}$ *holds*
- *if* $T = (x_j\ t_1 \ldots t_{n_j-1}\ t_{n_j}\ t_{n_j+1} \ldots t_n)$ *if* $(\mathrm{GD}\{l, t_h\})_{h=1,\ldots,n}$ *holds and* $\mathrm{RS}\{x_1, \ldots, x_m, n_1, \ldots, n_m, x, l, t_j\}$ *holds for* $j \in \{1, \ldots, m\}$
- *if* $T = (t\ t_1 \ldots t_n)$ *where* $t \notin \{x_1, \ldots, x_m\}$ *and* $(\mathrm{GD}\{l, t_h\})_{h=1,\ldots,n}$ *holds*
- *if* T *is the occurrence of a variable, a constant, an inductive type, an inductive constructor or a sort*
- *if* $T = \mathrm{FIX}_j\ \{y_1/n'_1 : T_1 := f_1\ ;\ \ldots\ ;\ y_d/n'_d : T_d := f_d\}$ *and* $(\mathrm{GD}\{l, T_h\})_{h=1,\ldots,d}$ *holds and* $(\mathrm{GD}\{l, f_h\})_{h=1,\ldots,d}$ *holds*

[5] To make the presentation of the rules shorter, in every recursive call all the constant parameters — i.e. every parameter but the last two — are omitted.

- *if* $T = \text{COFIX}_j\ \{y_1 : T_1 := f_1\ ;\ \ldots\ ;\ y_d : T_d := f_d\}$ *and* $(\text{GD}\{l, T_h\})_{h=1,\ldots,d}$ *holds and* $(\text{GD}\{l, f_h\})_{h=1,\ldots,d}$ *holds*
- *if* $T = \left(\begin{array}{l} < U > \text{CASES}_i\ t\ \text{OF} \\ \quad k_1\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_d\ \overline{x^d} \Rightarrow f_d \\ \text{END} \end{array}\right)$ *and one of the following three mutual exclusive conditions hold:*
  - i *is an inductive type and* $t = (z\ t_1 \ldots t_r)$ *where* $z \in l \cup \{x\}$ *and* $\text{GD}\{l, U\}$, $\text{GD}\{l, t\}$ *and* $(\text{GD}\{l \cup \text{PR}\{k_h, \overline{x^h}\}, G'_h\})_{h=1,\ldots,d}$ *hold where* $\text{PR}\{k_h, (x_1, \ldots, x_r)\} = \{x_j|$*, the* j*-th parameter of the* k_h *constructor is recursive in one of the types that are mutually recursive with its type*$\}$ *and* $G_h = \lambda y_1 : t_1 \ldots \lambda y_{n_h} : t_{n_h}.G'_h$ *where* n_h *is the number of parameter of the* k_h *constructor.*
  - i *is a coinductive type and* $\text{GD}\{l, U\}$, $\text{GD}\{l, t\}$ *and* $(\text{GD}\{l, f_h\})_{h=1,\ldots,d}$ *hold*
  - i *is an inductive type,* $t = (z\ t_1 \ldots t_r)$ *where* $z \in l \cup \{x\}$ *does not hold and* $\text{GD}\{l, U\}$, $\text{GD}\{l, t\}$ *and* $(\text{GD}\{l, f_h\})_{h=1,\ldots,d}$ *hold*

where RS *is the following condition:*

Definition 8.11 (RS (Really Smaller) condition) *The RS condition checks that the actual parameter that instantiates the decreasing formal parameter reduces only to smaller terms. The condition* $\text{RS}\{x_1, \ldots, x_m, n_1, \ldots, n_m, x, l, T\}$ *is defined by structural induction on* T *according to the following rules*[6]*. The condition holds*

- *if* $T = (z\ t_1 \ldots t_n)$ *where* $z \in l$
- *if* $T = \lambda z : M.N$ *and* $\text{RS}\{l, M\}$ *holds and* $\text{RS}\{l, N\}$ *hold*
- *if* $T = \text{FIX}_j\ \{y_1/n'_1 : T_1 := f_1\ ;\ \ldots\ ;\ y_d/n'_d : T_d := f_d\}$ *and* $(\text{RS}\{l, f_h\})_{h=1,\ldots,d}$ *holds*
- *if* $T = \text{COFIX}_j\ \{y_1 : T_1 := f_1\ ;\ \ldots\ ;\ y_d : T_d := f_d\}$ *and* $(\text{RS}\{l, f_h\})_{h=1,\ldots,d}$ *holds*
- *if* $T = \left(\begin{array}{l} < U > \text{CASES}_i\ t\ \text{OF} \\ \quad k_1\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_d\ \overline{x^d} \Rightarrow f_d \\ \text{END} \end{array}\right)$ *and one of the three following mutual exclusive conditions hold:*
  - i *is an inductive type and* $t = (z\ t_1 \ldots t_r)$ *where* $z \in l \cup \{x\}$ *and* $(\text{RS}\{l \cup \text{PR}\{k_h, \overline{x^h}\}, G'_h\})_{h=1,\ldots,d}$ *holds where* $\text{PR}\{k_h, (x_1, \ldots, x_r)\} = \{x_j|$ *the type of* j*-th parameter of the* k_h *constructor is an inductive type*$\}$ *and* $G_h = \lambda y_1 : t_1 \ldots \lambda y_{n_h} : t_{n_h}.G'_h$ *where* n_h *is the number of parameters of the* k_h *constructor.*

[6]To reduce the size of the formulae, in the recursive call every parameter is omitted, but the last two, which are the only parameters that are not constant.

- i *is a coinductive type and* $(\mathrm{RS}\{l, f_h\})_{h=1,\dots,d}$ *holds*
- i *is an inductive type and* $t = (z\ t_1 \dots t_r)$ *where* $z \in l \cup \{x\}$ *does not hold and* $(\mathrm{RS}\{l, f_h\})_{h=1,\dots,d}$ *holds*

The RS condition was introduced in [52] in a much simpler form: T is required to be an identifier in l. The definition given here has been obtained by reverse-engineering the behavior of Coq; no correctness proof for this formulation is known to the author. An error in the definition could imply that strong normalization (and hence also logical consistency) does not hold.

The last condition required is the Guarded by Constructors condition (GC). GC ensures strong normalization even for terms having as subterms occurrences of functions defined corecursively. The condition checks that results of recursive calls occurs only as arguments of a constructor. As a consequence, the term generated by the function call "grows" at each iteration and an infinite number of constructors (observations) may occur in the term lazily obtained. The infiniteness of the term is not incompatible with strong normalization: combining the GC condition and the constraints given on the unfolding operation — that grant a lazy unfolding strategy — the infinite term is not actually generated unless it needs to be inspected and it can be inspected only a finite number of times.

Definition 8.12 (GC (Guarded by Constructors) condition)
The condition $\mathrm{GC}_h\{x_1, \dots, x_m, T\}$, *— where* $x_1, \dots, x_m$ *are the functions defined by corecursion,* T *is the term to check and* $h \in \{0, 1\}$ *is a flag, initially of value 0, that records whether a constructor has been crossed — is defined by structural induction on* T *according to the following rules*[7]*. The condition holds*

- *if* $T = (z\ t_1 \dots t_n)$ *where* $z \in \{x_1, \dots, x_m\}$ *and* $h = 1$ *and* $x_1, \dots, x_m$ *do not occur in* $t_1, \dots, t_n$
- *if* $T = (k\ t_1 \dots t_n)$ *where* k *is the constructor of a coinductive that is abstracted over* n *parameters, and for each* $j \in \{1, \dots, n\}$ *if* k *is recursive in the* j*-th parameter then* $\mathrm{GC}_1\{t_j\}$ *holds, otherwise* $x_1, \dots, x_m$ *do not occur in* t_j
- *if* $T = (t\ t_1 \dots t_n)$ *where* $t \notin \{x_1, \dots, x_m\}$*,* t *is not a constructor of a coinductive type and* $x_1, \dots, x_m$ *do not occur in* $t, t_1, \dots, t_n$
- *if* T *is a sort, a (co)inductive type or a (co)inductive constructor*
- *if* $T = \lambda z : M.N$ *and* $x_1, \dots, x_m$ *do not occur in* M *and* $\mathrm{GC}_h\{N\}$ *holds*
- *if* $T = \mathrm{FIX}_j\ \{y_1/n'_1 : T_1 := f_1\ ;\ \dots\ ;\ y_d/n'_d : T_d := f_d\}$ *and* $x_1, \dots, x_m$ *do not occur in* $f_1, \dots, f_d, T_1, \dots, T_d$

[7]To shorten the rules, the parameters of the recursive calls are omitted, since they are constant.

- *if* $\mathsf{T} = \mathsf{COFIX}_j\,\{y_1 : T_1 := f_1\ ;\ \ldots\ ;\ y_d : T_d := f_d\}$ *and* $x_1, \ldots, x_m$ *do not occur in* $f_1, \ldots, f_d, T_1, \ldots, T_d$
- *if* $\mathsf{T} = \left(\begin{array}{l} <\mathsf{U}> \mathsf{CASES}_i\ t\ \mathsf{OF} \\ \quad k_1\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_d\ \overline{x^d} \Rightarrow f_d \\ \mathsf{END} \end{array}\right)$ *and* $x_1, \ldots, x_m$ *do not occur in* U *or* t *and one of the following two mutually exclusive conditions holds:*
  - $\mathsf{U} = \lambda z : M.(i\,t_1 \ldots t_r)$ *where* i *is a coinductive type and* $(GC_h\{f_l\})_{l \in \{1,\ldots,d\}}$ *holds*
  - $\mathsf{U} \neq \lambda z : M.(i\,t_1 \ldots t_r)$ *where* i *is a coinductive type and* $x_1, \ldots, x_m$ *do not occur in* $f_1, \ldots, f_d$

8.2 Some Extensions to the Calculus

In the previous section we described CIC trying to be as close as possible to the version that is implemented in Coq. In this section we informally describe some Coq implementation details that will play an important role also in our implementation.

8.2.1 Some extensions to CIC

The topic of this section are those extensions that directly affect the logical framework and that are relevant in this dissertation.

Implicit Universe Levels

The *universe level* of a sort $\mathrm{TYPE}(i)$ is the index i. Even though in the previous presentation of CIC the universes (i.e. the sorts TYPE) are totally ordered (by the universe level i), the partial order generated by the two side-conditions of the Prod-T typing rule and by the side condition of the rule Ax-Acc would be sufficient. From the user point of view, always making explicit the universe levels would be extremely demanding, since it is not always obvious what universe level should be assigned to each type. Moreover, in order to add a new definition or to prove a new theorem, it may be necessary to interpolate a new level between two consecutive universes: given two types of sort $\mathrm{TYPE}(i)$ and $\mathrm{TYPE}(i+1)$, it is easy to give a new definition that requires a sort $\mathrm{TYPE}(j)$ that must satisfy at the same time the constraints $i < j$ and $j < i+1$. To admit the definition, one possibility is to admit rational universe levels. Another solution, that is less feasible since it requires to change the whole library already defined, consists in incrementing every universe level greater than $i+1$.

For the previous reasons, the Coq developers chose to automatically infer the graph that describes the partial order on the set of universe levels in use [35]: the user has the feeling of working in a system with just a non-indexed sort TYPE; the system keeps the acyclic graph of the used universes and checks that no cycle is introduced by a new universe level assignment. When the type-checking of a new definition or theorem tries to introduce a cycle, the definition is rejected by the Coq kernel.

This solution has several shortcomings [35], even from the point of view of the user. The main one is that, for instance, it may happen that two terms t_1 and t_2 are well-typed in the same environment E, but their conjunction is not. This is a minimal example:

$$
\begin{array}{rcll}
E & = & [& (P : \mathbf{Type} := \mathbf{Type} \to \mathbf{Type}), \\
 & & & (Q : \mathbf{Type} := \mathbf{Type} \to \mathbf{Type}), \\
 & & & (p : P := \lambda x : \mathbf{Type}.x), \\
 & & & (q : Q := \lambda x : \mathbf{Type}.x) \\
 & &] &
\end{array}
$$

$$
\begin{array}{l}
t_1 : Type := (p\ Q) \\
t_2 : Type := (q\ P)
\end{array}
$$

The fact that a cycle is created is a trivial consequence of the symmetry of the roles played by p, P and q, Q.

The most annoying problem for the user is understanding what are the declarations and theorems that are responsible for the cycles in non-trivial cases. One way to address the problem is to give the user a suitable interface to render the problematic part of the universes graph, making explicit the dependencies inferred by the system. A more radical solution [35] consists in giving explicit names to every universe level and make the user write them explicitly. Unfortunately, this solution puts more burden on the user in all the cases, the great majority[8], where the phenomenon of cyclicity does not happen.

Cast

Casts are introduced in CIC to explicitly associate an expected type to a given term. For the property of existence of a principal type, up to convertibility, for each term of CIC, a cast does not increase the expressivity of CIC. Not withstanding this, casts have several applications in Coq: for instance, they allow the user to put more information in terms containing implicit parameters

[8] Examples of situations where the user works with several universe levels at once is the study of Set Theory or Category Theory, where she needs to distinguish big sets (big categories) from small sets (small categories). Since the Coq system does not implement universe polymorphism [57], the user is forced to syntactically duplicate the two (or more) definitions and all the theorems stated on them in order to make the system generate two different TYPEs. According to the relative use of the two syntactically equal definitions, the system will impose the typing constraints on the universe levels: those definitions whose universe levels are smaller will become the small set (small categories).

that must be inferred; the additional information increases the possibility of correctly inferring the term. The Cast operator also interacts with the procedure that infers the universe levels, forcing the system to unify some type indexes.

The concrete syntax in Coq to say that a term t is casted to a type T is $t :: T$. The typing rule is pretty obvious:

Cast

$$\frac{E[\Gamma] \vdash t : T_2 \qquad E[\Gamma] \vdash T_1 : s \qquad T_1 =_{\beta\delta\iota\zeta} T_2}{E[\Gamma] \vdash (t :: T_1) : T_1}$$

Metavariables

The Curry-Howard isomorphism, that is established between complete proofs and λ-terms, can be extended to deal with non-completed proofs. λ-terms are extended with *metavariables*, which correspond to the conjectures the proof is actually based on. Metavariables, that are syntactically distinct from variables, are particular constants with a type and a context (that together form a sequent), but that do not have yet a body. To complete the proof, the user has to instantiate every metavariable with a term well-typed in the metavariable context.

The Coq syntax for metavariables occurrences in a term is $?i$, where i is an integer (an unique identifier). A new map (`metasenv`) is introduced to associate a sequent to each metavariable. An incomplete proof or definition is an object whose `metasenv` has a non empty domain. Only complete proofs and definitions can enter the environment.

In this chapter we do not show the typing rules introduced in Coq for metavariables. Instead, in Sect. 10.1 we will present our own version of CIC extended with metavariables, and we will give the substitution, reduction and typing rules for the calculus.

Implicit Terms

There is much redundant information in λ-terms. The redundant information can be exploited to write a type-inference based procedure to reconstruct some omitted subterms of a term. In Coq, for instance, the user can put question marks in place of subterms that she wants the system to infer. Sometimes the question marks themselves can be omitted.

Implicit terms can be dealt with either in the kernel (by adding them directly to CIC) or outside the kernel. In the latter case, a new type of CIC terms extended with implicit arguments is added outside the kernel and the correctness of the kernel does not depend on the correctness of the refinement algorithm (which is the algorithm that infers the replacements for the implicit terms). As a drawback of this solution, neither the efficiency of the type-checker nor the size of the terms that enter the environment are improved. While Coq adopts the approach of avoiding implicit arguments in the kernel, Necula and Lee in [94] follow the alternative approach[9]. The results that they obtain are quite impressive: both the size of the proofs and their type-checking time are in $O(n^{0.62})$ where n is the value obtained before introducing implicit terms. The constants hidden in the big-O notation are also very small.

8.2.2 Coq Implementation Details

As for the most part of the systems based on the Curry-Howard isomorphism and developed after Automath [95], bound variables are internally represented as de Brujin's indexes [91]. In Coq de Brujin's indexes are also exploited for bound constants, such as mutual references between the entities mutually defined inside a single block. For instance, the internal representation of the definition

$$
\begin{array}{l}
\Pi A : \mathbf{Prop}.\{ \\
\quad \mathsf{tree} : \mathbf{Set} := \{ \\
\quad\quad \mathsf{emptyT} : (\mathsf{tree}\ A)\ ; \\
\quad\quad \mathsf{node} : A \to (\mathsf{forest}\ A) \\
\quad \}\ ; \\
\quad \mathsf{forest} : \mathbf{Set} := \{ \\
\quad\quad \mathsf{emptyF} : (\mathsf{forest}\ A)\ ; \\
\quad\quad \mathsf{cons} : (\mathsf{tree}\ A) \to (\mathsf{forest}\ A) \to (\mathsf{forest}\ A) \\
\quad \} \\
\}
\end{array}
$$

is the following:

$$
\begin{array}{l}
\Pi \mathbf{Prop}.\{ \\
\quad \mathsf{tree} : \mathbf{Set} := \{ \\
\quad\quad \mathsf{emptyT} : (\mathbf{1}\ \mathbf{3})\ ; \\
\quad\quad \mathsf{node} : \mathbf{3} \to (\mathbf{2}\ \mathbf{3}) \\
\quad \}\ ; \\
\quad \mathsf{forest} : \mathbf{Set} := \{ \\
\quad\quad \mathsf{emptyF} : (\mathbf{2}\ \mathbf{3})\ ; \\
\quad\quad \mathsf{cons} : (\mathbf{1}\ \mathbf{3}) \to (\mathbf{2}\ \mathbf{3}) \to (\mathbf{2}\ \mathbf{3}) \\
\quad \} \\
\}
\end{array}
$$

Another implementation detail of the Coq system is the internal representation of the CASES operator, which, without any apparent reason, is provided in two different forms. The first one

[9] Necula and Lee work in a logical framework derived by LF and not in CIC. Nevertheless, the problems raised by the addition of implicit arguments are similar.

is used for the general (dependent) case, while the second one is used only for non-dependent elimination. In the non-dependent case, the type of the CASES branches — which is displayed between angular brackets — does not start with the λ-abstraction over the term the user is doing case-analysis on. Both in the dependent and non-dependent case, the redundancy between the inductive type and their constructors is resolved by λ-abstracting the branches of the CASES over the parameters of the constructors, omitting the constructors.

For instance, the following term

$$\begin{array}{l}\lambda n : nat. \\ \quad < \lambda z : nat.nat > \mathsf{CASES}_{nat}\ n\ \mathsf{OF} \\ \qquad \mathsf{O} \Rightarrow \mathsf{O} \mid (\mathsf{S}\ m) \Rightarrow m \\ \quad \mathsf{END}\end{array}$$

can be represented internally either as the fake dependent elimination

$$\begin{array}{l}\lambda nat. \\ \quad < \lambda nat.nat > \mathsf{CASES}_{nat}\ \mathbf{1}\ \mathsf{OF} \\ \qquad \mathsf{O} \mid \lambda nat.\mathbf{1} \\ \quad \mathsf{END}\end{array}$$

or as the non-dependent elimination

$$\begin{array}{l}\lambda nat. \\ \quad < nat > \mathsf{CASES}_{nat}\ \mathbf{1}\ \mathsf{OF} \\ \qquad \mathsf{O} \mid \lambda nat.\mathbf{1} \\ \quad \mathsf{END}\end{array}$$

8.2.3 The Objects Level and the Environment Management

We call *objects* every entity which can be put in the environment. In CIC, an object can be either a constant definition, introduced by the WF-Const rule, or the definition of mutually recursive types, introduced by the rule WF-Ind. Coq refines the class of constants in definitions and theorems, and it introduces two new types of object, that are axioms and variables. Axioms and variables have a type, as CIC constants have, but they do not have a body (a definiens). Thus, axioms and variables occurrences are already in δ-normal form. The rule to add them to the environment is the following:

WF-SectVar

$$\frac{E[[]] \vdash T : s \qquad c \notin E}{\mathcal{WF}(E; c : T)[[]]}$$

The difference between axioms and variables will be explained in Sect. 8.2.4. The difference between theorems and definitions is in their *opacity* status: theorems are usually *opaque*, i.e. δ-expansion is not performed over theorem occurrences. Forbidding δ-expansion approximates the principle of *proof-irrelevance* that states that there is no way to discriminate any two proofs of the same statement[10]. Definitions, instead, are usually *transparent*, i.e. normally subject to δ-expansion.

Sometimes it is useful to modify the opacity status of a term, for instance to formulate a theorem over a proof or to make a data type abstract. Starting from the version 7.1 of Coq, though, the user is not given any more the possibility to make transparent a term previously defined as opaque[11].

8.2.4 Sections and Discharging

Dowek added to Coq a machinery to structure mathematical theories into blocks (called *sections*) that comprise particular axioms (called *variables*) that are *discharged* when the block is closed. The approach is directly inspired by the usual mathematical practice: for instance, a mathematician that wants to develop group theory is likely to begin its work by assuming the usual group axioms; later on she will apply her theorems to any mathematical structure, provided that she can prove that the group axioms hold for that structure. Discharging consists of universally quantifying (i.e. Π-abstracting) over a variable c every type depending on c. As a consequence, to obtain well-typed terms, every term that depends over c must be λ-abstracted over c and every occurrence of a term discharged over c must be applied to the new variable c' bound by the new λ-abstraction. Discharged variables are no longer referred, and they can be freely removed from the environment.

The following simple example should clarify the previous informal description. Let us consider the following theory:

```
Section S.
 Variable A : Prop.
 Variable B : Prop.

 Theorem AB_A_B : (A -> B) -> A -> B.
 Proof.
```

[10] To have complete proof-irrelevance, any two terms inhabiting the same type of sort **Prop** should be convertible. [85] is a preliminary proposal to provide a proof-irrelevant version of CIC

[11] The restriction is unrelated to the consistency of the system. Indeed it is still possible for the user to add some code outside the kernel to bypass the restriction.

```
  Exact [ab:(A ->B)][a:A](ab a).
 Qed.

 Theorem A_AB_B : A -> (A -> B) -> B.
 Proof.
  Exact [a:A][ab:(A->B)](AB_A_B ab a).
 Qed.
End S.
```

We just used the Coq syntax which should be self-explicative, except for the peculiar notation used for primitive operators: "$[x : T]t$" stands for "$\lambda x : T.t$"; "$(x : T)t$" stands for "$\Pi x : T.t$"; finally "->" stands for "$\rightarrow$". When the section is closed (by the "END S" command), its content is discharged. The result is the same that would have been obtained from the following definitions:

```
Theorem AB_A_B : (A:Prop)(B:Prop)(A -> B) -> A -> B.
Proof.
 Exact [A:Prop][B:Prop][ab:(A ->B)][a:A](ab a).
Qed.

Theorem A_AB_B : (A:Prop)(B:Prop)A -> (A -> B) -> B.
Proof.
 Exact [A:Prop][B:Prop][a:A][ab:(A->B)](AB_A_B A B ab a).
Qed.
```

Sections and discharging are useful and widely used features of the Coq system. Nevertheless, they introduce mutability for the objects in the environment: environments, that in pure CIC can only be extended with new objects, are now subject to removals and in-place modifications of the objects.

A natural choice in the development of our proof assistant based on a distributed library is that of replacing the environment with a view (i.e. a subset) of the whole library. The issue was already discussed in Sect. 2. To preserve consistency, a library should be made of immutable objects: any modification cannot be performed as a local operation, but involves the updating of all the parts of the library that depend on the changed result, yielding a new version of the whole library.

As a consequence, discharging is not compatible with our view of the environment as a fragment of the whole library. To overcome the difficulty, in Sect. 8.3 we will propose an extension to CIC that allows the same expressivity as sections without requiring discharging or any other operation affecting the library when closing a section.

8.2.5 Names and Section Paths

Sections are made of an ordered list of objects definitions and nested sections. Hence sections are structured as trees, whose internal nodes are sections and whose leaves are object definitions. A Coq theory (a ".v" file) is just a section whose name is the name of the file. Thus loading several theories at once means building a forest. The most natural way to identify a leaf in a forest is using a *section path*, i.e. an unique identifier obtained appending all the section names in the root-leaf path and adding the object name at the end. Delimiters must be used to separate the names in the section path.

Unfortunately, closing a section (and discharging the objects therein) has the effect of moving every child of the section a level up in the tree, shortening its section path. When every section is closed, what is left by the discharging operation is just a flat and not very informative list of names. Moreover, closing a section can produce name clashes, as shown in the following example where the definition A given in section S2 is not valid since it will clash with the one given in S1:

```
Section S1.
 Axiom A : Prop.
End S1.

Section S2.
 Axiom A : Prop.
End S2.
```

Name clashes arise spontaneously for definitions given by different authors in different theories. Coq developers have chosen a shading politics for names belonging to different theories: only the definition in the last required section is in scope[12].

8.3 CIC with Explicit Named Substitutions

In this section we introduce a new extension to CIC which is meant to replace the discharging machinery of Sect. 8.2.4, without loosing the expressivity and user-friendliness of sections.

The section is organized as follows: after introducing the main motivations for our new extension in Sect. 8.3.1, we present the calculus in Sect. 8.3.2 and a correct translation of CIC with discharging into CIC with Explicit Named Substitutions in Sect. 8.3.3.

[12]The latest version of Coq has been extended introducing modules and functors similar to the ones of the OCaml system [31, 73]. This forced the implementors to change the naming policy. Thus the problems described in this section have been already partially fixed in Coq.

8.3.1 Motivations

In Coq it is possible to abstract a group of definitions and theorems with respect to some assumptions. Here is an example (in Coq syntax):

```
Section S.
 Variable A : Prop.
 Variable B : Prop.
 Definition H1 : Prop := A /\ B.
 Theorem T1 : H1 -> A.
  Proof.
   Exact (Cases H1 of (conj A B a b) => a end).
  Qed.
End S.
(* Here the type of H1 and T1 are different. See code fragment below *)
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True True (I,I)).
 Qed.
```

The previous fragment should be equivalent, from a logic point of view, to the following input:

```
Definition H1 : Prop -> Prop -> Prop := [A:Prop ; B:Prop]A/\B.
Theorem T1 : (A:Prop ; B:Prop)(H1 A B) -> A.
 Proof.
  Exact (Cases (H1 A B) of (conj A B a b) => a end).
 Qed.
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True True (I,I)).
 Qed.
```

The operation that transforms the first code fragment in the second one is called *discharging*. Discharging is implemented in an external layer of the Coq system, in such a way that the kernel of the system is given the discharged term (and the theory of the kernel of Coq does not need to be modified).

Since, for rendering purposes, we are more interested in the first fragment, which is the one entered by the user, it seems natural to export into the library only the undischarged form of the theorems and definitions. The problem with that representation is that the theorem `T1` that is used in theorem `T2` is no more equal to the one defined above: for instance, its type is different. This implies two kinds of problems in developing tools over the library:

1. While rendering `T2`, we would like to make the occurrence of `T1` an hyperlink to its definition. What we get is misleading for the reader: the theorem `T1` that is shown has a type that is not the same of that of its occurrence in `T2`.

2. To proof-check `T2` we need the type of the discharged form of `T1`. So we are obliged to discharge `T1` and this leads to serious problems: either we save the discharged form, as Coq does, and this goes against our initial choice; or we discharge the theorem on-the-fly when needed, and, this being an expensive procedure, we have to implement complex caching machineries[13]. Notice also that sections can be arbitrarily nested. Thus if we decided to saved the discharged form, we would be obliged to save also all the intermediate forms that are obtained by closing only the innermost sections. This solution easily leads to wasting considerable amounts of memory, especially for well-structured theories with several levels of sections nesting.

If we reflect on the real cause of the problems given by discharging, we will easily identify a severe problem that must be faced when designing a theory whose information must be exported into a library: the theory may force proof assistants into being implemented in an imperative way, where there is a strong notion of time and the objects of the library change from time to time. This behaviour seems utterly incompatible with the notion of mathematical library according to one of the golden rules that we identified in [110] and we presented in Sect. 6.

A library, due to its nature, is a random access structure, both in space and time. Try to minimize the dependencies, give them the right direction, beware of "imperative" commands of the system that change already defined things. Make them immutable.

In the next section we propose an original extension to CIC which preserves the section facility avoiding the addition of any imperative operation. The burden added to the user is also minimal, consisting in the introduction of a new kind of application that corresponds to the abstraction over a section variable.

8.3.2 The Calculus

We introduce the idea underlying our extension to CIC considering again the example presented in the previous section:

```
Section S.
 Variable A : Prop.
 Variable B : Prop.
```

[13]This is what we implemented in an experiment, but the nature of the discharging operation interfered with the usual locality reference principle of caches. As a result we got very poor and unexpected performances.

```
 Definition H1 : Prop := A /\ B.
 Theorem T1 : H1 -> A.
  Proof.
   Exact (Cases H1 of (conj A B a b) => a end).
  Qed.
End S.
(* Here the type of H1 and T1 are different. See code fragment below *)
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True True (I,I)).
 Qed.
```

It is pretty obvious that, inside the section `S`, the recursive dependencies of `H1` and `T1` over the section variables `A` and `B` represent a kind of implicit abstraction. More precisely, there are two hidden abstractions: the first one, similar to a Π-abstraction, abstracts the types of `H1` and `T1` over `A` and `B`; the second one, similar to a λ-abstraction, abstracts the bodies of `H1` and `T1` over `A` and `B`. The two implicit abstractions are converted to real Π- and λ-abstractions during discharging.

The idea is to introduce two new kinds of abstractions over variables and to use them to make the dependencies explicit even inside the section. Or even better, we can just introduce a single *named* abstraction of a whole object over one variable. The notation $\mathcal{D}efinition\ D(V_1, \ldots, V_n)$, $\mathcal{T}heorem\ T(V_1, \ldots, V_n)$, $\mathcal{I}nductive\ I(V_1, \ldots, V_n)$ and $\mathcal{V}ariable\ V(V_1, \ldots, V_n)$ is used for the named abstraction over the variables $V_1, \ldots, V_n$.

```
Section S.
 Variable A : Prop.
 Variable B : Prop.
 Definition H1(A,B) : Prop := A /\ B.
 Theorem T1(A,B) : H1 -> A.
  Proof.
   Exact (Cases H1 of (conj A B a b) => a end).
  Qed.
End S.
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1 True True (I,I)).
 Qed.
```

The types of the new parameters `A` and `B` of `H1` and `T1` can be deduced by looking in the environment for the variables in scope called `A` and `B`.

It is now clear that two different CIC constructors are used to eliminate the new abstractions inside and outside the section. Outside the section, thanks to the discharging operation, the arguments are instantiated by means of the usual CIC application. During the discharging operation, the order of definition of the variables determines the correspondence between the position of the argument and the variables that get instantiated.

There are two possible interpretations for the elimination of the abstractions inside the section: either we consider the arguments to be non-instantiated (as it happens in Coq before discharging); or we consider the arguments to be implicitly instantiated with a fresh occurrence of the variable itself (as it roughly happens in Coq after the discharging, up to the fact that a new binder is also created).

Thus, to get rid of the discharging operation, it is sufficient to introduce a new form of CIC application that subsumes the two forms of instantiation of the parameters inside and outside a section. To this aim, we can profitably use *explicit substitutions*. Moreover, since the abstractions are naturally named, we can introduce *explicit named substitutions* into the calculus, making the substitution optional: the default for a named substitution is the substitution with a new fresh occurrence of the variable itself.

```
Section S.
 Variable A : Prop.
 Variable B : Prop.
 Definition H1(A,B) : Prop := A /\ B.
 Theorem T1(A,B) : H1 -> A.
  Proof.
   Exact (Cases H1 of (conj A B a b) => a end).
  Qed.
End S.
Theorem T2 : True /\ True -> True.
 Proof.
  Exact (T1[True/A ; True/B] (I,I)).
 Qed.
```

We use the notation $[t_1/v_1 \; ; \; \ldots \; ; \; t_n/v_n]$ for the explicit substitution of the terms $t_1, \ldots, t_n$ in place of the parameters (i.e. the variables) $v_1, \ldots, v_n$. Note that, in the definition of `T1`, no explicit substitution is specified over `H1`. The intended meaning is the default substitution $[A/A \; ; \; B/B]$.

It is well known (see, for instance, [90]) that the introduction of explicit substitutions in CIC leads to several problems related to reduction and the creation of critical pairs. The problems arise from the fact that the explicit substitutions instantiate the parameters bound by the λ-abstractions, which are parts of β-redexes.

In our calculus, explicit named substitutions can only instantiate explicit named abstractions over section variables. Moreover, since explicit named abstractions occur only in the definitions of the objects, we can syntactically restrict the occurrences of explicit named substitutions to be applied only to occurrences of references to objects (i.e. only constants occurrences, variables occurrences, (co)inductive types occurrences and (co)inductive constructors occurrences can be arguments of an explicit named substitution).

As a result, we obtain a very simplified form of explicit substitutions, characterized by very simple and local modifications to the syntax, reduction and typing rules of CIC.

Terms

We extend the terms of CIC adding explicit named substitutions to the occurrences of constants, variables, inductive types and inductive constructors:

$$
\begin{array}{lcll}
t & ::= & \ldots & \\
 & | & c[\sigma] & \text{constants} \\
 & | & v[\sigma] & \text{variables} \\
 & | & i[\sigma] & \text{(co)inductive types} \\
 & | & k[\sigma] & \text{(co)inductive constructors} \\
 & & & \\
\sigma & ::= & t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n & \text{(simultaneous) explicit named substitution}
\end{array}
$$

To avoid confusion, from now on we will write explicit named substitutions using square brackets, and we will reserve the curly brackets for the usual "instantaneous" substitutions. All the substitutions we will consider in the following sections will be simultaneous.

Our instantaneous substitutions will come in two flavors: those whose domain is a set of bound variables (de Brujin's indexes) and those whose domain is the set of variables. To improve readability, we will put a small v subscript to the closing curly bracket of instantaneous substitutions of variables.

Substitution and Reduction

Substitution is naturally propagated to the terms in explicit named substitutions:

$$r[t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n]\{\sigma\} = r[t_1\{\sigma\}/v_1 \ ; \ \ldots \ ; \ t_n\{\sigma\}/v_n]$$

where r is a reference to a constant, variable, (co)inductive type or (co)inductive constructor and $\{\sigma\} = \{t'_1/x_1 \ ; \ \ldots \ ; \ t'_m/x_m\}$ is an instantaneous substitution. Notice that the domain of the explicit named substitution and the domain of the instantaneous substitution are disjoint, since the former ranges over variables and the latter ranges over bound variables.

δ-reduction is affected by explicit named substitutions: if b is the definiens for the definiendum c, then

$$c[\sigma] \triangleright_\delta b\{\sigma\}_\nu$$

Notice that, during δ-reduction, the explicit named substitutions is converted to an instantaneous substitution (whose domain is given by variable occurrences). Instantaneous substitution of variable occurrences is performed recursively over a CIC term until one of the following cases is met:

$$\begin{array}{ll} \nu_i[\sigma]\{t_1/\nu_1 \ ; \ \dots \ ; \ t_n/\nu_n\}_\nu = t_i & \text{if } \sigma \text{ is syntactically an identity substitution} \\ \nu[\sigma]\{t_1/\nu_1 \ ; \ \dots \ ; \ t_n/\nu_n\}_\nu = \nu[[\sigma] \vee_\nu \{t_1/\nu_1 \ ; \ \dots \ ; \ t_n/\nu_n\}_\nu] & \text{otherwise} \\ r[\sigma]\{t_1/\nu_1 \ ; \ \dots \ ; \ t_n/\nu_n\}_\nu = r[[\sigma] \vee_r \{t_1/\nu_1 \ ; \ \dots \ ; \ t_n/\nu_n\}_\nu] & \end{array}$$

where r ranges over constants, (co)inductive types and (co)inductive constructors occurrences and where the merge operator $\vee_r$ over explicit named substitutions is defined as follows:

$$\begin{array}{l} [t_1/\nu_1 \ ; \ \dots \ ; \ t_n/\nu_n] \vee_r \{t'_1/\nu'_1 \ ; \ \dots \ ; \ t'_n/\nu'_m\}_\nu = \\ \quad t_1\{t'_1/\nu'_1 \ ; \ \dots \ ; \ t'_n/\nu'_m\}_\nu/\nu_1 \ ; \ \dots \ ; \ t_n\{t'_1/\nu'_1 \ ; \ \dots \ ; \ t'_n/\nu'_m\}_\nu/\nu_n \ ; \\ \quad\; t'_{i_1}/\nu'_{i_1} \ ; \ \dots \ ; \ t'_{i_s}/\nu'_{i_s} \end{array}$$

where $i_1, \dots, i_s$ is the subset of $\{1, \dots, m\}$ such that, for each j in $i_1, \dots, i_s$, $\nu_{i_j} \notin \{\nu_1, \dots, \nu_n\}$ and the object r is a reference to is abstracted over the variable ν_{i_j}.

Notice that, if we always state explicitly the identity substitutions ν_i/ν_i for each uninstantiated variable ν_i, then we can simply characterize $\vee_r$ as follows:

$$[t_1/\nu_1 \ ; \ \dots \ ; \ t_n/\nu_n] \vee_r \{\sigma\}_\nu = [t_1\{\sigma\}_\nu/\nu_1 \ ; \ \dots \ ; \ t_n\{\sigma\}_\nu/\nu_n]$$

Note also that, according to the definition of $\{\_\}_\nu$, ν_i is considered to be free in $\nu_i[\sigma]$ if and only if σ is syntactically equal to an identity substitution. To understand the intuition behind this design choice, let us consider the following example:

```
Variable A: Set.
Variable op: A -> A.
Definition opop(A,op) : A -> A := [x:A](op (op x)).
Definition d1() : bool -> bool := opop[bool/A; not/op].
Definition d2() : bool -> bool := opop[bool/A; op[bool/A]/op].
```

In this example we assume the existence of a data-type `A` and a unary operator `op` over `A` and we define a new unary operator `opop` as the composition of the operator `op` with itself. Later on we instantiate `opop` twice, leading to the two definitions `d1` and `d2`. The first definition fully instantiates both `A` (with the data-type of boolean values) and `op` (as the negation) and leads to a definition that does not depend on any variable. Indeed, no variable occurs free in its body. The definition of `d2`, instead, is a bit peculiar: `d2` is a unary operator over boolean values, that is obtained by instantiating `opop` replacing `op` with a new operator `op[bool/A]`. In the next section we will see that all the terms involved in the example are well-typed, included the last term. What term does the notation `op[bool/A]` stand for? The typing rules tell us that it must be a unary operator whose type is `bool -> bool`. However, no such term exists yet, since `op` is a variable of type `A -> A` for a generic but fixed `A`. Intuitively, a solution could be interpreting `op[bool/A]` as a reference to a brand new anonymous axiom introduced in the environment. The type of the axiom would be `bool -> bool` and its declaration would be equal to the declaration of `op` except for the substitution of `A` with `bool` everywhere in the type. Unfortunately, this solution would not work since it would not respect convertibility: two different (but convertible) occurrences of `op[bool/A]` would be interpreted as two non convertible axioms[14]. To solve the problem, we can interpret `op[bool/A]` as the application `(_op bool)`, where `_op` is an axiom of type $\forall A : Set.A \to A$ (i.e. the discharged form of `op`) that is introduced once and for all when `op` is found during the translation of the environment.

The fact that we interpret `op[bool/A]` as `(_op bool)` explains why we must assume that `op` is free in `op[bool/A]`; it also explains why `op[bool/A]`$\{$`t/op`$\}_\nu$ `= op[bool/A]` for each term `t`.

Before reading the previous explanation, the careful reader was probably wondering why in the previous example the definition `d2` has an empty list of parameters — forbiding any further instantiation — instead of having the "parameter" `op[bool/A]`:

```
Definition d2(op[bool/A]) : bool -> bool :=
 opop[bool/A; op[bool/A]/op].
```

Notice that this solution would not require the introduction of any axiom `_op` in the interpretation: `op[bool/A]` would simply be interpreted as a parameter. Moreover, adding the parameter to `d2` would later allow us to instantiate the parameter with something useful. For instance:

```
Definition d3() : bool -> bool := d2[not / op[bool/A]].
```

Although this abstraction is both semantically sound and intuitively appealing, it would really complicate the semantics of the calculus, since identifiers and equality between identifiers would

[14]The problem cannot be solved by assuming that any two convertible occurrences are interpreted using the same axiom. Indeed, if we consider the definition `b(A) := C[A/A ; nat/B]` where A, B, C are variables, then `b[bool/A]` must be convertible with `C[bool/A ; nat/B]`. Unfortunately, `C[A/A ; nat/B]` would be interpreted as an axiom a_1 whose type is quantified over A, `b[bool/A]` would be reducible to $(a_1\ \mathrm{bool})$ and `C[bool/A ; nat/B]` would be interpreted as an axiom a_2 that is not convertible to $(a_1\ \mathrm{bool})$.

be replaced respectively by terms and the convertibility check (making the substitution operator and the convertibility check two mutual recursive definitions). Moreover, it would not add much to the expressivity of the calculus: we can easily reproduce the situation described above simply polluting a bit the environment by introducing a new variable, as the following example shows.

```
/* bool -> bool is the type of op[bool/A] */
Variable op_bool : bool -> bool.
Definition d2'(op_bool) : bool -> bool :=
 opop[bool/A; op_bool/op].
Definition d3'() : bool -> bool := d2'[not/op_bool].
```

Thus, in the whole dissertation, the rules of the calculus are presented in such a way that a variable to whom an explicit named substitution is applied behaves as an application whose head is the axiom that corresponds to the discharged form of the variables and whose arguments are the arguments of the explicit named substitution.

The second reduction rule that is affected — even if in a minimal way — by the introduction of explicit named substitution is ι-reduction: if the declaration

$$
\begin{array}{l}
(\nu_1, \ldots, \nu_n)\, \Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\
\quad \{i_1 : A_1 := \{k_1^1 : C_1^1 \;;\; \ldots \;;\; k_{n_1}^1 : C_{n_1}^1\}\,; \\
\quad \ldots\,; \\
\quad i_m : A_m := \{k_1^m : C_1^m \;;\; \ldots \;;\; k_{n_m}^m : C_{n_m}^m\} \\
\quad \}_{(\mathrm{CO})\mathrm{IND}}
\end{array}
$$

occurs in the environment E, then

$$
\begin{array}{ll}
< T > \mathrm{CASES}_{i_l}\ (k_j^l[\sigma]\ t_1 \ldots t_h t_1' \ldots t_{n_j}')\ \mathrm{OF} & \\
\quad k_1^l\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_n^l\ \overline{x^n} \Rightarrow f_n & \triangleright_\iota\ f_j\{t_1'/x_1 \;;\; \ldots \;;\; t_{n_j}'/x_{n_j}\} \\
\mathrm{END} &
\end{array}
$$

where $l \in \{1, \ldots, m\}$, $\forall l \in \{1, \ldots, n_j\}.|x_l| = n_l$ and $\overline{x^j} = (x_1, \ldots, x_{n_j})$.

Notice that the only modification to the corresponding rule of CIC has been the introduction of the explicit named substitution in the argument of the case analysis. Notice also that the explicit named substitution, as well as the other global parameters $x_1, \ldots, x_h$ of the mutual inductive definition block $\star$, plays absolutely no role in the reduction. Thus, essentially nothing is changed w.r.t. the corresponding rule of CIC.

Type-Checking

The following CIC type-checking rules are also affected by explicit named substitutions. As usual, to shorten the rules, the symbol $\star$ stands for the following mutual declaration of (co)inductive types:

$$
\begin{array}{l}
(\nu_1, \ldots, \nu_n)\ \Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\
\quad \{i_1 : A_1 := \{k_1^1 : C_1^1\ ;\ \ldots\ ;\ k_{n_1}^1 : C_{n_1}^1\}\ ; \\
\quad \ldots\ ; \\
\quad i_m : A_m := \{k_1^m : C_1^m\ ;\ \ldots\ ;\ k_{n_m}^m : C_{n_m}^m\} \\
\quad \}_{(\mathrm{CO})\mathrm{IND}}
\end{array}
$$

SectVar

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (\nu : T) \in E \qquad E[\Gamma] \vdash \mathcal{WF}(\nu, \sigma)}{E[\Gamma] \vdash \nu[\sigma] : T\{\sigma\}_\nu}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c : T := t) \in E \qquad E[\Gamma] \vdash \mathcal{WF}(c, \sigma)}{E[\Gamma] \vdash c[\sigma] : T\{\sigma\}_\nu}$$

Ind

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \star \in E \qquad E[\Gamma] \vdash \mathcal{WF}(i_j, \sigma)}{E[\Gamma] \vdash i_j[\sigma] : (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j)\{\sigma\}_\nu} \qquad \text{if } j \leq m$$

Constr

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \star \in E \qquad E[\Gamma] \vdash \mathcal{WF}(k_l^j, \sigma)}{E[\Gamma] \vdash k_l^j[\sigma] : (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.C_l^j)\{\sigma\}_\nu} \qquad \begin{array}{ll} \text{if} & j \leq m \\ \text{and} & l \leq n_j \end{array}$$

Case

$$\frac{\begin{array}{l} \star \in E \qquad E[\Gamma] \vdash t : (i_l[\sigma]\ t_1 \ldots t_h\ t'_1 \ldots t'_s) \\ E[\Gamma] \vdash T : A \qquad [(i_l[\sigma]\ t_1 \ldots t_h) : A_l\{\sigma\}_\nu\{t_1/x_1\ ;\ \ldots\ ;\ t_h/x_h\}|A]_{i_l} \\ (E[\Gamma] \vdash f_j : \Delta\{C_j^l\{\sigma\}_\nu\{t_1/x_1\ ;\ \ldots\ ;\ t_h/x_h\}, T, (k_j^l[\sigma]\ t_1 \ldots t_h)\})_{j=1,\ldots,n_l} \end{array}}{E[\Gamma] \vdash \left(\begin{array}{l} < T > \mathrm{CASES}_{i_l}\ t\ \mathrm{OF} \\ \quad k_1^l\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_{n_l}^l\ \overline{x^n} \Rightarrow f_{n_l} \\ \mathrm{END} \end{array} \right) : (T\ t'_1 \ldots t'_s\ t)}$$

Notice that the modifications to the case analysis rule are really straightforward: the arguments of the explicit named substitution behaves exactly as the parameters of the mutual inductive definitions block. Indeed, the whole mutual inductive definitions block $\star$ is abstracted over both the named variables $\nu_1, \ldots, \nu_n$ and the parameters $x_1, \ldots, x_h$.

Moreover, we add the following typing rules to establish the well-formedness of explicit named substitutions:

WF-ens

$$\frac{\begin{array}{c} \mathcal{WF}(E)[\Gamma] \qquad E[\Gamma] \vdash t_i : T_i\{t_1/\nu_1\ ;\ \ldots\ ;\ t_{i-1}/\nu_{i-1}\}_\nu \\ \text{the object referenced by } r \text{ is abstracted over } \nu_i : T_i \text{ for each } i \\ t_i = \nu_i \text{ or } t_i \text{ is not convertible with } \nu_i \end{array}}{E[\Gamma] \vdash \mathcal{WF}(r, t_1/\nu_1\ ;\ \ldots\ ;\ t_n/\nu_n)}$$

The last premise of the rule is used to avoid the need for the mutual recursion between substitution and convertibility. Indeed, we just defined substitution in such a way that $\nu[\sigma]\{\sigma'\}_\nu$ is a function of σ being *sintactically* equal or not to the identity substitution. The premise grants that *any* explicit named substitution that is convertible to an identity substitution is also sintactically equal to an identity substitution.

Finally, we add the following side condition to the rules WF-SectVar, WF-Const and WF-Ind:

> The object declaration *must* be abstracted over all the variables ν that occur free in the declaration, i.e. all the variables ν such that $\nu[\sigma]$ occurs in o, where σ is an identity substitution.

8.3.3 Translating CIC with discharging into CIC with Explicit Named Substitutions

The main aim of this section is to provide a translation (definition 8.13) from CIC environments obtained through discharging to environments for CIC extended with explicit named substitutions. The translation has been implemented into the XML exportation module of Coq, described in Chap. 6. As a consequence, we have been able to convert the whole library of Coq and all its contribs from core CIC to CIC with explicit named substitutions.

The main result of the section is theorem 8.14 that grants that the translation preserves both reduction and typing rules.

Let $\vdash_{\mathrm{ens}}$ be the typing relation of CIC extended with sections, variables and explicit named substitutions and let $\vdash_{\mathrm{dis}}$ be the typing relation of CIC without these extensions.

We define the translation $[\![.]\!]$ from CIC objects and terms to objects and terms of CIC extended with explicit named substitutions as follows:

Definition 8.13 *Objects are translated by removing the abstractions introduced by discharging, restoring the variables occurrences in place of the fake bound variables and recurring on the terms to introduce explicit named substitutions:*

1. *Let* $c : \Pi\nu_1 : T_1 \ldots \Pi\nu_n : T_n\{\nu_1/\nu_1\ ;\ \ldots \nu_{n-1}/\nu_{n-1}\}_\nu.T := \lambda\nu_1 : T_1 \ldots \lambda\nu_n : T_n.t$ *be the discharged definition of a constant that used to depend, directly or indirectly, over the section variables* $\nu_1 : T_1, \ldots, \nu_n : T_n$ *(in declaration order).*

$$[\![c : \Pi\nu_1 : T_1 \ldots \Pi\nu_n : T_n\{\nu_1/\nu_1\ ;\ \ldots\ ;\ \nu_{n-1}/\nu_{n-1}\}_\nu.T := \lambda\nu_1 : T_1 \ldots \lambda\nu_n : T_n.t]\!] \stackrel{\mathrm{def}}{=}$$
$$c(\nu_1, \ldots, \nu_n) : [\![T\{\nu_1/\nu_1\ ;\ \ldots\ ;\ \nu_n/\nu_n\}]\!] := [\![t\{\nu_1/\nu_1\ ;\ \ldots\ ;\ \nu_n/\nu_n\}]\!]$$

2. *Let*

$$\begin{array}{l}\Pi\nu_1 : T_1.\ldots\Pi\nu_n : T_n\{\nu_1/\nu_1\ ;\ \ldots\ ;\ \nu_{n-1}/\nu_{n-1}\}_\nu.\\ \quad\Pi x_1 : U_1\ldots\Pi x_h : U_h.\\ \qquad\{i_1 : A_1 := \{k_1^1 : C_1^1\ ;\ \ldots\ ;\ k_{n_1}^1 : C_{n_1}^1\}\ ;\\ \qquad\ldots\ ;\\ \qquad i_m : A_m := \{k_1^m : C_1^m\ ;\ \ldots\ ;\ k_{n_m}^m : C_{n_m}^m\}\\ \qquad\}_{(\mathrm{CO})\mathrm{IND}}\end{array}$$

be the discharged definition of mutual (co)inductive types that used to depend, directly or indirectly, over the section variables $\nu_1 : T_1, \ldots, \nu_n : T_n$ *(in declaration order).*

$$\begin{array}{l}[\![\Pi\nu_1 : T_1.\ldots\Pi\nu_n : T_n\{\nu_1/\nu_1\ ;\ \ldots\ ;\ \nu_{n-1}/\nu_{n-1}\}_\nu.\\ \quad\Pi x_1 : U_1\ldots\Pi x_h : U_h.\\ \qquad\{i_1 : A_1 := \{k_1^1 : C_1^1\ ;\ \ldots\ ;\ k_{n_1}^1 : C_{n_1}^1\}\ ;\\ \qquad\ldots\ ;\\ \qquad i_m : A_m := \{k_1^m : C_1^m\ ;\ \ldots\ ;\ k_{n_m}^m : C_{n_m}^m\}\\ \qquad\}_{(\mathrm{CO})\mathrm{IND}}]\!]\ \stackrel{\mathrm{def}}{=}\\ \Pi x_1 : [\![U_1\sigma]\!]\ldots\Pi x_h : [\![U_h\sigma]\!].\\ \qquad\{i_1(\nu_1,\ldots,\nu_n) : [\![A_1\sigma]\!] := \{k_1^1 : [\![C_1^1\sigma]\!]\ ;\ \ldots\ ;\ k_{n_1}^1 : [\![C_{n_1}^1\sigma]\!]\}\ ;\\ \qquad\ldots\ ;\\ \qquad i_m(\nu_1,\ldots,\nu_n) : [\![A_m\sigma]\!] := \{k_1^m : [\![C_1^m\sigma]\!]\ ;\ \ldots\ ;\ k_{n_m}^m : [\![C_{n_m}^m\sigma]\!]\}\\ \qquad\}_{(\mathrm{CO})\mathrm{IND}}\end{array}$$

where $\sigma = \{\nu_1/\nu_1\ ;\ \ldots\nu_n/\nu_n\}$.

Terms in the declaration/definition of every object are recursively translated until a term $[\![(r\ t_1\ \ldots\ t_n)]\!]$ *is met (where* r *is a reference to a constant, variable, (co)inductive type or (co)inductive constructor). Let* r *be a reference to an object explicitly abstracted over the variables in scope* $\nu_1 : T_1, \ldots, \nu_m : T_m$*. Note that the variables that are not in scope must not be considered. Then:*

$$[\![(r\ t_1\ \ldots\ t_n)]\!] =$$
$$= \begin{cases} \begin{array}{l}\lambda\nu_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/\nu_1\ ;\ \ldots\ ;\ [\![t_n]\!]/\nu_n\}_\nu\ \ldots\\ \quad\lambda\nu_m : [\![T_m]\!]\{[\![t_1]\!]/\nu_1\ ;\ \ldots\ ;\ [\![t_n]\!]/\nu_n\ ;\ \nu_{n+1}/\nu_{n+1}\ ;\ \ldots\ ;\ \nu_{m-1}/\nu_{m-1}\}_\nu.\\ \qquad r[[\![t_1]\!]/\nu_1\ ;\ \ldots\ ;\ [\![t_n]\!]/\nu_n\ ;\ \nu_{n+1}/\nu_{n+1}\ ;\ \ldots\ ;\ \nu_m/\nu_m]\end{array} & \textit{if } n \le m\\ (r[[\![t_1]\!]/\nu_1\ ;\ \ldots\ ;\ [\![t_m]\!]/\nu_m]\ [\![t_{m+1}]\!]\ \ldots\ [\![t_n]\!]) & \textit{otherwise}\end{cases}$$

□

Note that in definition 8.13 $[\nu_i/\nu_i]$ and $\{\nu_i/\nu_i\}_\nu$ are not identity named substitutions since the ν_i being replaced is the occurrence of the section variable ν_i, whereas the ν_i that replaces it is an occurrence of a λ-bound local variable (i.e. a de Brujin's index). Similarly, $\{\nu_i/\nu_i\}$ is not the identity named substitution since the v_i being replaced is a λ-bound local variable, whereas the v_i that replaces it is a section variable.

Note also that, for each term t of a discharged environment of CIC (previously extended with sections and variables), if $v[\sigma]$ occurs in $[\![t]\!]$, then σ is the identity substitution. Indeed, if v occurs in t then v was in scope before the discharging operation. Thus all the variables v depend on were also in scope (since in CIC extended with sections and variables the scopes are well-nested). Since the discharging operation replaces every reference r to an object that depends on a variable v with the application of r to a fresh λ-bound variable v, we conclude that v is applied in t to the freshly introduced λ-bound variables corresponding to the variables it used to depend on. Finally, the $[\![\_]\!]$ translation will map the application to the identity substitution.

Note that this remarkable property holds for the terms obtained converting discharged terms of CIC to terms of CIC extended with explicit named substitutions, but it does not hold in general for every well-typed term of CIC extended with explicit named substitutions.

Before proving the correctness of the proposed translation, we show a brief artificial example intended to provide the intuition that underlays the definition. Let us consider the following development in CIC extended with sections and variables:

```
Section S.
 Variable A: Set.
 Section S'.
  Variable f: A -> A.
  Definition d : (A -> A) := [a:A](f a).
 End S'.
 Definition d' : A -> A := (d [a:A]a).
End S.
Definition d'' : nat -> nat := (d' nat).
Definition e : nat -> nat := (d nat S).
```

Once discharged, the development looks like this:

```
Definition d : (A:Set)(A -> A) -> A -> A
 := [A:Set][f:A->A][a:A](f a).
Definition d' : (A:Set)A -> A
 := [A:Set](d A [a:A]a).
Definition d'' : nat -> nat
 := (d' nat).
Definition e : nat -> nat
 := (d nat S).
```

Using our definition, we can convert the discharged environment to CIC extended with explicit named substitutions:

```
Variable A: Set.
Variable f: A -> A.
Definition d(A,f) : (A -> A) := [a:A](f a).
Definition d(A)' : A -> A := d[[a:A]a / f].
Definition d''() : nat -> nat := d'[nat/A].
Definition e() : nat -> nat := d[nat/A ; S/f].
```

Despite the intuitive nature of the translation, its proof of correctness is rather technical, due to the replacement of de Brujin indexes with variables.

Theorem 8.14 *Let* E *be a discharged environment of CIC (previously extended with sections and variables, that were removed by the discharging phase). If* $\mathcal{WF}_{dis}(E)[[]]$ *then* $\mathcal{WF}_{ens}([\![E]\!])[[]]$*. Moreover, for each derivation of* $E'[\Gamma] \vdash_{dis} t : T$ *that is part of the proof of* $\mathcal{WF}_{dis}(E)[[]]$*, there exists a corresponding derivation of* $[\![E']\!][[\![\Gamma']\!]] \vdash_{ens} [\![t]\!] : T' =_{\beta\delta\iota\zeta} [\![T]\!]$ *in the proof of* $\mathcal{WF}_{ens}([\![E]\!])[[]]$ *where* $\Gamma = v_1 : T_1; \ ; \ \ldots \ ; \ v_m : T_m \ ; \ \Gamma'$ *and* $v_1, \ldots, v_m$ *are the variables that were discharged.*

Before proceeding in the proof of theorem 8.14, we need to establish the following lemma about reduction:

Lemma 8.15 *Let* E *be a well-formed, already discharged environment and* Γ *be a well-formed context in* E *such that* $\mathcal{WF}_{ens}([\![E]\!])[[\![\Gamma]\!]]$*. If* $E[\Gamma] \vdash_{dis} t \rhd^* t'$ *then* $[\![E]\!][[\![\Gamma]\!]] \vdash_{ens} [\![t]\!] \rhd^* t'' \ {}_{\beta}^{*}\!\lhd [\![t']\!]$*.*

Proof: We proceed by induction over the derivation of $E[\Gamma] \vdash_{dis} t \rhd^* t'$. The only interesting case is the one of δ-reduction of a reference to a constant:

$$(c\ t_1\ \ldots\ t_n) \rhd_\delta (E(c)\ t_1\ \ldots\ t_n)$$

Let $v_1 : T_1, \ldots, v_m : T_m$ be the variables in scope c used to depend on. We have to consider two cases:

1. Case $n \leq m$:

$$\begin{array}{rl}
[\![(c\ t_1\ \ldots\ t_n)]\!] = & \lambda v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\}_v.\ \ldots \\
& \quad \lambda v_m : [\![T_m]\!]\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\ ;\ v_{n+1}/v_{n+1}\ ;\ \ldots\ ;\ v_{m-1}/v_{m-1}\}_v. \\
& \quad\quad c[[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\ ;\ v_{n+1}/v_{n+1}\ ;\ \ldots\ ;\ v_m/v_m] \\
\rhd_\delta & \lambda v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\}_v.\ \ldots \\
& \quad \lambda v_m : [\![T_m]\!]\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\ ;\ v_{n+1}/v_{n+1}\ ;\ \ldots\ ;\ v_{m-1}/v_{m-1}\}_v. \\
& \quad\quad [\![E]\!](c)\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\ ;\ v_{n+1}/v_{n+1}\ ;\ \ldots\ ;\ v_m/v_m\}_v
\end{array}$$

Let $E(c) = \lambda v_1 : T_1. \ \ldots \ \lambda v_m : T_m\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.t$. From the definition of $[\![.]\!]$, we know $[\![t]\!] = [\![E]\!](c)\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/v_m\}_v$. Thus

$$\begin{array}{rcl}
[\![(E(c)\ t_1\ \ldots\ t_n)]\!] & = & [\![(\lambda v_1 : T_1. \ \ldots \ \lambda v_m : T_m\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.t\ t_1\ \ldots\ t_n)]\!] \\
& = & (\lambda v_1 : [\![T_1]\!]. \ \ldots \ \lambda v_m : [\![T_m]\!]\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.[\![t]\!] \\
& & \quad [\![t_1]\!]\ \ldots\ [\![t_n]\!]) \\
& \rhd_\beta^* & \lambda v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\}_v. \ \ldots \\
& & \quad \lambda v_m : [\![T_m]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v. \\
& & \quad [\![t]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\} \\
& = & \lambda v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\}_v. \ \ldots \\
& & \quad \lambda v_m : [\![T_m]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v. \\
& & \quad [\![E]\!](c)\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/v_m\}_v\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\} \\
& = & \lambda v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\}_v. \ \ldots \\
& & \quad \lambda v_m : [\![T_m]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v. \\
& & \quad [\![E]\!](c)\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_m/v_m\}_v
\end{array}$$

that closes the case.

2. Case $n > m$:

$$\begin{array}{rcl}
[\![(c\ t_1\ \ldots\ t_n)]\!] & = & (c[[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_m]\!]/v_m][\![t_{m+1}]\!]\ \ldots\ [\![t_n]\!]) \\
& \rhd_\delta & ([\![E]\!](c)\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_m]\!]/v_m\}_v[\![t_{m+1}]\!]\ \ldots\ [\![t_n]\!])
\end{array}$$

Let $E(c) = \lambda v_1 : T_1. \ \ldots \ \lambda v_m : T_m\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.t$. From the definition of $[\![.]\!]$, we know $[\![t]\!] = [\![E]\!](c)\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/v_m\}_v$. Thus

$$\begin{array}{rcl}
[\![(E(c)\ t_1\ \ldots\ t_n)]\!] & = & [\![(\lambda v_1 : T_1. \ \ldots \ \lambda v_m : T_m\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.t\ t_1\ \ldots\ t_n)]\!] \\
& = & (\lambda v_1 : [\![T_1]\!]. \ \ldots \ \lambda v_m : [\![T_m]\!]\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.[\![t]\!] \\
& & \quad [\![t_1]\!]\ \ldots\ [\![t_n]\!]) \\
& \rhd_\beta^* & ([\![t]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_m]\!]/v_m\}\ [\![t_{m+1}]\!]\ \ldots\ [\![t_n]\!]) \\
& = & ([\![E]\!](c)\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/v_m\}_v\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_m]\!]/v_m\} \\
& & \quad [\![t_{m+1}]\!]\ \ldots\ [\![t_n]\!]) \\
& = & ([\![E]\!](c)\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_m]\!]/v_m\}_v\ [\![t_{m+1}]\!]\ \ldots\ [\![t_n]\!])
\end{array}$$

that closes the case.

$\square$

Corollary 8.16 *Under the same hypotheses as lemma 8.15, if* $E[\Gamma] \vdash_{dis} t =_{\beta\delta\iota\zeta} t'$ *then* $[\![E]\!][[\![\Gamma]\!]] \vdash_{ens}$ $[\![t]\!] =_{\beta\delta\iota\zeta} [\![t']\!]$.

We now have all the required machinery to prove theorem 8.14.

Proof: The proof is by induction over the structure of E. The base case trivially holds. In the inductive step, for each object appended to E, the proof proceeds by induction over the derivation of the well-typedness of the object in the environment E. Let Γ' be the restricted context obtained from Γ by removing all the declarations that were introduced by discharging. From the definition of $[\![.]\!]$ we know that all the translated terms that occur in the translated derivations are close w.r.t. Γ'. Thus we know that the well-typedness of the terms in Γ' is equivalent to the well-typedness of the terms in Γ. We use this result to simplify the proof, by using $[\![\Gamma]\!]$ in place of $[\![\Gamma']\!]$ in the rest of the proof.

Convertibility is dealt within lemma 8.16. The most difficult is the one of an application whose head is a reference to a constant, variable, (co)inductive type or (co)inductive constructor:

$$E[\Gamma] \vdash_{dis} (r\ t_1\ \ldots\ t_n) : T \tag{8.17}$$

where r is a reference to an object that used to depend over the variables $v_1 : T_1, \ldots, v_m : T_m$.

We have to consider two cases:

1. Case $n \leq m$:
   From 8.17, using the inversion lemma of CIC [118], we can deduce

$$\begin{array}{l} E[\Gamma] \vdash_{dis} t_1 : T_1' =_{\beta\delta\iota\zeta} T_1\{v_1/v_1\ ;\ \ldots\ ;\ v_n/v_n\}_v \\ \ldots \\ E[\Gamma] \vdash_{dis} t_n : T_n' =_{\beta\delta\iota\zeta} T_n\{t_1/v_1\ ;\ \ldots\ ;\ t_{n-1}/v_{n-1}\ ;\ v_n/v_n\}_v \end{array} \tag{8.18}$$

   We need to prove

$$[\![E]\!][[\![\Gamma]\!]] \vdash_{ens} [\![(r\ t_1\ \ldots\ t_n)]\!] : T' =_{\beta\delta\iota\zeta} [\![T]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_n/v_n\}$$

   that is equivalent to

$$\begin{array}{l} [\![E]\!][[\![\Gamma]\!]] \vdash_{ens} \\ \quad (\lambda v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\}_v.\ \ldots \\ \qquad \lambda v_m : [\![T_m\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\ ;\ v_{n+1}/v_{n+1}\ ;\ \ldots\ ;\ v_{m-1}/v_{m-1}\}_v]\!]. \\ \qquad\quad r[[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_n]\!]/v_n\ ;\ v_{n+1}/v_{n+1}\ ;\ \ldots\ ;\ v_m/v_m]) \\ \quad : T' =_{\beta\delta\iota\zeta} [\![T]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_n/v_n\} \end{array}$$

From 8.18, corollary 8.16 and the inductive hypothesis we conclude that the explicit named substitution is well-typed:

$$
\begin{aligned}
&[\![E]\!][[\![\Gamma]\!]] \vdash_{ens} \\
&\quad (\lambda v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\}_v. \ \ldots \\
&\qquad \lambda v_m : [\![T_m\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v]\!]. \\
&\qquad\quad r[[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_m/v_m]) \\
&\quad : \Pi v_{n+1} : [\![T_{n+1}]\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\}_v. \ \ldots \\
&\qquad \Pi v_m : [\![T_m\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v]\!]. \\
&\qquad\quad S'\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_m/v_m\}_v
\end{aligned}
$$

where $(r(v_1, \ldots, v_m) : S') \in [\![E]\!]$.

From the definition of $[\![.]\!]$, we know

$$E(r) = \lambda v_1 : T_1. \ \ldots \ \lambda v_m : T_m\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.t$$

Thus

$$
\begin{aligned}
&E[\Gamma] \vdash_{dis} (r \ t_1 \ \ldots \ t_n) : T = \\
&\quad \Pi v_{n+1} : T_{n+1}\{t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n\}_v. \ \ldots \\
&\qquad \Pi v_m : T_m\{t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.S
\end{aligned}
$$

where $E[\Gamma] \vdash_{dis}= t : S''$ and $E[\Gamma] \vdash_{dis} S''\{t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n\} =_{\beta\delta\iota\zeta} S$. Using corollary 8.16, we can reduce the thesis to proving

$$[\![S''\{t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n\}]\!] =_{\beta\delta\iota\zeta} S'\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n \ ; \ v_{n+1}/v_{n+1} \ ; \ \ldots \ ; \ v_m/v_m\}_v$$

Or, equivalently, that

$$[\![S'']\!]\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\} =_{\beta\delta\iota\zeta} S'\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/v_m\}_v\{[\![t_1]\!]/v_1 \ ; \ \ldots \ ; \ [\![t_n]\!]/v_n\}$$

To prove the thesis, we will prove that

$$[\![S'']\!] =_{\beta\delta\iota\zeta} S'\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/v_m\}_v$$

From the definition of $[\![.]\!]$ we know that

$$[\![t]\!] = [\![E]\!](r)\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/v_m\}_v$$

Since $t : S''$, we have $[\![t]\!] : S''' =_{\beta\delta\iota\zeta} [\![S'']\!]$ using the induction hypothesis.

The thesis follows from the fact that $[\![E]\!](r) : S'$

2. Case $n > m$: We need to prove

$$[\![E]\!][\Gamma] \vdash_{ens} [\![(r\ t_1\ \ldots\ t_n)]\!] : T' =_{\beta\delta\iota\zeta} [\![T]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_n/v_n\}$$

that is equivalent to

$$[\![E]\!][[\![\Gamma]\!]] \vdash_{ens} (r[[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m]\ t_{m+1}\ \ldots\ t_n) : T' =_{\beta\delta\iota\zeta} [\![T]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_n/v_n\}$$

Following the same argument used in the previous case, from 8.17, corollary 8.16 and the inversion lemma of CIC [118] we conclude that the translated term is well-typed. Moreover,

$$[\![E]\!][[\![\Gamma]\!]] \vdash_{ens} r[[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m]\ : S'\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m\}_v :$$

where $(r(v_1, \ldots, v_m) : S') \in [\![E]\!]$.

Another consequence of the inversion lemma of CIC and of corollary 8.16 is that we can prove the thesis by showing that

$$[\![E]\!][[\![\Gamma]\!]] \vdash_{ens} r[[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m] : T'' =_{\beta\delta\iota\zeta} [\![T_0]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_n/v_n\}$$

where $E[\Gamma] \vdash_{dis} (r\ t_1\ \ldots\ t_m) : T_0$

From the definition of $[\![.]\!]$, we know

$$E(r) = \lambda v_1 : T_1.\ \ldots\ \lambda v_m : T_m\{v_1/v_1\ ;\ \ldots\ ;\ v_{m-1}/v_{m-1}\}_v.t$$

Thus $E[\Gamma] \vdash_{dis} (r\ t_1\ \ldots\ t_m) : T_0 =_{\beta\delta\iota\zeta} S''\{t_1/v_1\ ;\ \ldots\ ;\ t_m/v_m\}$
where $E[\Gamma] \vdash_{dis} = t : S''$

Using corollary 8.16, we can reduce the thesis to proving

$$[\![S''\{t_1/v_1\ ;\ \ldots\ ;\ t_m/v_m\}]\!] =_{\beta\delta\iota\zeta} S'\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m\}_v$$

Or, equivalently, that

$$[\![S'']\!]\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m\} =_{\beta\delta\iota\zeta} S'\{v_1/v_1\ ;\ \ldots\ ;\ v_m/v_m\}_v\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m\}$$

To prove the thesis, we will prove that

$$[\![S'']\!] =_{\beta\delta\iota\zeta} S'\{v_1/v_1\ ;\ \ldots\ ;\ v_m/v_m\}_v$$

From the definition of $[\![.]\!]$ we know that

$$[\![t]\!] = [\![E]\!](r)\{v_1/v_1\ ;\ \ldots\ ;\ v_m/v_m\}_v$$

Since $t : S''$, we have $[\![t]\!] : S''' =_{\beta\delta\iota\zeta} [\![S'']\!]$ using the induction hypothesis.

The thesis follows from the fact that $[\![E]\!](r) : S'$

□

8.3.4 Consistency of CIC with Explicit Named Substitutions

In the previous section we have proved that we can provide a faithful encoding of CIC extended with sections, variables and discharging into CIC extended with explicit named substitutions. Nevertheless, we still have to prove that all the interesting properties of CIC (subject reduction, strong normalization, consistency, etc.) still hold for CIC with explicit named substitutions. We prove all of these properties at once by providing an encoding of CIC plus explicit named substitutions into CIC, and showing that the encoding preserves both convertibility and typing. At a first glance, the encoding may seem just a simple variant of discharging. Indeed, this is the case if we restrict ourselves to the situation where the explicit named substitutions applied to the variables are all identity substitutions. Remember that this property holds for all the developments obtained in the previous section by means of the translation from discharged CIC. In the general case, though, we will be obliged to introduce new axioms in the environment during the translation. The axioms correspond to the discharged form of the variables. References to them are used to replace the istances of explicit named substitutions that are not identities and that are applied to variables. Another solution will be totally avoiding the problem by forbidding the application of explicit named substitutions to variables (while keeping them for axioms, constants, inductive types and inductive terms). We prefer not to do so because we want to minimize the differences between axioms and variables. Indeed, in section 8.3.5 we will argue in favor of the removal of axioms from CIC, since variables and explicit named substitutions are more expressive and seem, in certain situations, to be closer to the common mathematical practice (where several sets of equivalent axioms are usually assumed at once and they are considered interchangeable).

Definition 8.19 *Let* T_1 ; $\ldots$; T_m *be the respective types of the variables* $v_1, \ldots v_m$. *Let* σ *be the instantaneous substitution* $\{v_1/v_1 \ ; \ \ldots \ ; \ v_m/m_v\}_v$. *We translate variables and objects explicitly abstracted over* $v_1, \ldots, v_m$ *as follows:*

$$\begin{aligned}
&[\![c(v_1,\ldots,v_m) : T := t]\!] \stackrel{\text{def}}{=} \\
&\quad c : \Pi v_1 : T_1.\ldots.\Pi v_m : T_m\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.[\![T]\!]\sigma \\
&\quad := \lambda v_1 : T_1.\ldots.\lambda v_m : T_m\{v_1/v_1 \ ; \ \ldots \ ; \ v_{m-1}/v_{m-1}\}_v.[\![t]\!]\sigma
\end{aligned}$$

$$\begin{aligned}
&[\![\Pi x_1 : U_1 \ldots \Pi x_h : U_h. \\
&\qquad \{i_1(v_1,\ldots,v_m) : A_1 := \{k_1^1 : C_1^1 \ ; \ \ldots \ ; \ k_{n_1}^1 : C_{n_1}^1\} \ ; \\
&\qquad \ldots \ ; \\
&\qquad\ i_m(v_1,\ldots,v_m) : A_m := \{k_1^m : C_1^m \ ; \ \ldots \ ; \ k_{n_m}^m : C_{n_m}^m\} \\
&\qquad \}_{(\text{CO})\text{IND}}]\!] \stackrel{\text{def}}{=}
\end{aligned}$$

$$\begin{array}{l}\Pi v_1 : T_1 \ldots \Pi v_m : T_m\{v_1/v_1\ ;\ \ldots\ ;\ v_{m-1}/v_{m-1}\}_v.\\ \quad \Pi x_1 : [\![U_1]\!]\sigma \ldots \Pi x_h : [\![U_h]\!]\sigma.\\ \qquad \{i_1(v_1,\ldots,v_m) : [\![A_1]\!]\sigma := \{k_1^1 : [\![C_1^1]\!]\sigma\ ;\ \ldots\ ;\ k_{n_1}^1 : [\![C_{n_1}^1]\!]\sigma\}\ ;\\ \qquad\ \ldots\ ;\\ \qquad\ i_m(v_1,\ldots,v_m) : [\![A_m]\!]\sigma := \{k_1^m : [\![C_1^m]\!]\sigma\ ;\ \ldots\ ;\ k_{n_m}^m : [\![C_{n_m}^m]\!]\sigma\}\\ \qquad \}_{(\mathrm{CO})\mathrm{IND}}\end{array}$$

$$\begin{array}{l}[\![v(v_1,\ldots,v_m) : T]\!] \stackrel{\mathrm{def}}{=}\\ \quad \_v : \Pi v_1 : T_1 \ldots . \Pi v_m : T_m\{v_1/v_1\ ;\ \ldots\ ;\ v_{m-1}/v_{m-1}\}_v.[\![T]\!]\sigma\end{array}$$

Notice that a new axiom $\_v$ is generated for each variable v in the environment. We will use the new axiom to translate occurrences of $v[\sigma]$ where σ is not an identity substitution[15]*. The translation over terms proceeds recursively until the following cases are met:*

$$\begin{array}{lll}[\![r[t_1/v_1'\ ;\ \ldots\ ;\ t_n/v_n']]\!] \stackrel{\mathrm{def}}{=} (r\ [\![t_1]\!]\ \ldots\ [\![t_n]\!]) & \textit{for } r \in \{c,k,i\}\\ [\![v[\sigma]]\!] \stackrel{\mathrm{def}}{=} v & \textit{when } \sigma \textit{ is the identity substitution}\\ [\![v[t_1/v_1'\ ;\ \ldots\ ;\ t_n/v_n']]\!] \stackrel{\mathrm{def}}{=} (\_v\ [\![t_1]\!]\ \ldots\ [\![t_n]\!]) & \textit{otherwise}\end{array}$$

□

Theorem 8.20 *Let E be an environment of CIC extended with variables and explicit named substitutions. If $\mathcal{WF}_{ens}(E)[[]]$ then $\mathcal{WF}_{CIC}([\![E]\!])[[]]$.*

Before proceeding in the proof of theorem 8.20, we need to establish the following two lemmas about variable substitution and about reduction:

Lemma 8.21 *For each well-typed term t, explicit named substitution σ and variable v, $[\![t\{\sigma\}_v]\!] = [\![t]\!]\{[\![\sigma]\!]\}_v$*

Proof: We proceed by induction over the structure of t. The only interesting case are the references to constants, inductive types, inductive constructors and variables:

- Case $t = r[\sigma']$ when r is a reference to a constant, mutual inductive type or mutual inductive constructor.

  Let us suppose that σ' is made fully explicit (adding the substitutions v/v for each variable v that r depends on and that does not belong to the domain of σ'):

  $$\begin{array}{l}[\![r[\sigma']\{\sigma\}_v]\!]\\ \quad = [\![r[[\sigma'] \vee_r \{\sigma\}_v]]\!]\\ \quad = [\![r[\sigma'\{\sigma\}_v]]\!]\\ \quad = (r\ [\![\sigma'\{\sigma\}_v]\!])\\ \quad = (r\ [\![\sigma']\!])\{[\![\sigma]\!]\}_v\\ \quad = [\![r[\sigma']]\!]\{[\![\sigma]\!]\}_v\end{array}$$

[15] Since an empty substitution is always an identity substitution, we do not really need the axioms $\_v$ that are generated from the variables v that do not depend on any other variable.

The fourth rewriting step is justified by the induction hypothesis.

- Case $t = v[\sigma']$ when σ' is the identity substitution made fully explicit.

  From the last premise of the WF-ens rule (see 8.3.2), we know that if σ' is an identity substitution, then it is syntactically equal to an identity substitution and then v occurs free in $v[\sigma']$. Thus

  $$\begin{array}{l} [\![v[\sigma']\{\sigma\}_v]\!] \\ \quad = \begin{cases} [\![\sigma(v)]\!] & \text{if } v \in \mathrm{Dom}(\sigma) \\ [\![v[\sigma'\{\sigma\}_v] & \text{otherwise} \end{cases} \end{array}$$

  If $v \in \mathrm{Dom}(\sigma)$ then $[\![v[\sigma']]\!]\{[\![\sigma]\!]\}_v = v\{[\![\sigma]\!]\}_v = [\![\sigma(v)]\!]$.

  Otherwise the proof is identical to the proof of the previous case where r is replaced by v.

- Case $t = v[\sigma']$ when σ' is not the identity substitution.

  Let us suppose that σ' is made fully explicit.

  $$\begin{array}{l} [\![v[\sigma']\{\sigma\}_v]\!] \\ \quad = [\![v[[\sigma'] \vee_r \{\sigma\}_v]]\!] \\ \quad = [\![v[\sigma'\{\sigma\}_v]]\!] \\ \quad = (\_v\ [\![\sigma'\{\sigma\}_v]\!]) \\ \quad = (\_v\ [\![\sigma']\!])\{[\![\sigma]\!]\}_v \\ \quad = [\![v[\sigma']]\!]\{[\![\sigma]\!]\}_v \end{array}$$

  The last rewriting step is justified by the induction hypothesis.

□

Lemma 8.22 *Let* E *be a well-formed environment and* Γ *be a well-formed context in* E *such that* $\mathcal{WF}_{CIC}([\![E]\!])[[\![\Gamma]\!]]$. *If* $E[\Gamma] \vdash_{ens} t \rhd t'$ *then* $[\![E]\!][[\![\Gamma]\!]] \vdash_{CIC} [\![t]\!] \rhd^* t'' \rhd^*_\beta [\![t']\!]$.

Proof: We proceed by induction over the derivation of $E[\Gamma] \vdash_{ens} t \rhd t'$. The only interesting case is the one of a reference to a constant c: $E[\Gamma] \vdash_{ens} c[t_1/v_1\ ;\ \ldots\ ;\ t_m/v_m] \rhd_\delta t\{t_1/v_1\ ;\ \ldots\ ;\ t_m/v_m\}_v$ where $(c(v_1,\ldots,v_m) : T := t) \in E$. From the definition of $[\![.]\!]$ we know $(c := \lambda v_1 : T_1 \ldots . \lambda v_m : T_m\{v_1/v_1\ ;\ \ldots\ ;\ t_{m-1}/v_{m-1}\}_v . [\![t]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_m/v_m\}_v) \in [\![E]\!]$. Thus:

$$\begin{array}{rcl} [\![E]\!][\Gamma] \vdash_{CIC} [\![c[t_1/v_1\ ;\ \ldots\ ;\ t_m/v_m]]\!] & = & (c\ [\![t_1]\!]\ ;\ \ldots\ ;\ [\![t_m]\!]) \\ & \rhd_\delta & (\lambda v_1 : T_1 \ldots . \\ & & \quad \lambda v_m : T_m\{v_1/v_1\ ;\ \ldots\ ;\ t_{m-1}/v_{m-1}\}_v . \\ & & \quad\quad [\![t]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_m/v_m\}_v \\ & & \quad [\![t_1]\!]\ \ldots\ [\![t_m]\!]) \\ & \rhd^*_\beta & [\![t]\!]\{v_1/v_1\ ;\ \ldots\ ;\ v_m/v_m\}_v\ \{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m\} \\ & = & [\![t]\!]\{[\![t_1]\!]/v_1\ ;\ \ldots\ ;\ [\![t_m]\!]/v_m\}_v \end{array}$$

$$= \quad [\![t\{t_1/v_1\ ;\ \ldots\ ;\ t_m/v_m\}_v]\!]$$

The last rewriting step is justified by Lemma 8.21. □

Corollary 8.23 *Under the same hypotheses as lemma 8.22, if* $E[\Gamma] \vdash_{ens} t =_{\beta\delta\iota\zeta} t'$ *then* $[\![E]\!][[\![\Gamma]\!]] \vdash_{CIC} [\![t]\!] =_{\beta\delta\iota\zeta} [\![t']\!]$.

We are now ready to prove theorem 8.20.

Proof: The proof is by induction over the structure of E. The base case trivially holds. In the inductive step, for each object appended to E, the proof proceeds by induction over the derivation of the well-typedness of the object in the environment E. Convertibility is dealt with lemma 8.23. We are left with only two interesting cases.

- Case $t = v[\sigma]$ when σ is the identity substitution.

  From the hypothesis $E[\Gamma] \vdash_{ens} v[\sigma] : T$ we need to prove

  $$\begin{array}{l}[\![E]\!][v'_1 : [\![T'_1]\!]\ ;\ \ldots\ ;\ v'_r : [\![T'_r]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_{r-1}/v'_{r-1}\}_v\ ;\ [\![\Gamma]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_r/v'_r\}_v] \vdash_{CIC} \\ \quad [\![v[\sigma]]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_r/v'_r\}_v : T' =_{\beta\delta\iota\zeta} [\![T]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_r/v'_r\}_v\end{array}$$

  where $v'_1 : T'_1, \ldots, v'_r : T'_r$ are the section variables that occur free in σ, i.e. they are a subset of the section variables the object we are currently type-checking depends on.

  Since σ is an identity substitution, then v occurs free in $v[\sigma]$. Thus $v = v'_j$ for some j, $T = T'_j$ and $[\![v[\sigma]]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_r/v'_r\}_v = v'_j : [\![T'_j]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_{j-1}/v'_{j-1}\}_v$.

- Case $t = r[\sigma]$ when r is a reference to a variable and σ is not an identity substitution, or where r is a reference to a constant, (co)inductive type or (co)inductive constructor.

  Let r' be $\_v$ if r is a reference to a variable v, and let it be r otherwise. Let us suppose that $(r'(v_1, \ldots, v_m) : S) \in E$. From the hypothesis

  $$E[\Gamma] \vdash_{ens} r[t_1/v_1\ \ldots\ t_m/v_m] : T \tag{8.24}$$

  we need to prove that:

  $$\begin{array}{l}[\![E]\!][v'_1 : [\![T'_1]\!]\ ;\ \ldots\ ;\ v'_r : [\![T'_r]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_{r-1}/v'_{r-1}\}_v\ ;\ [\![\Gamma]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_r/v'_r\}_v] \vdash_{CIC} \\ \quad (r'\ [\![t_1]\!]\ \ldots\ [\![t_m]\!])\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_r/v'_r\}_v : T' =_{\beta\delta\iota\zeta} [\![T]\!]\{v'_1/v'_1\ ;\ \ldots\ ;\ v'_r/v'_r\}_v\end{array}$$

  where $v'_1 : T'_1, \ldots, v'_r : T'_r$ are the section variables that occur free in $t_1, \ldots, t_m$, i.e. they are a subset of the section variables the object we are currently type-checking depends on.

  From 8.24, using the inversion lemma of CIC extended with explicit named substitutions — whose proof is a simple extension of the proof given for CIC — we know:

  $$\begin{array}{l}E[\Gamma] \vdash_{ens} t_1 : T_1 \\ \ldots \\ E[\Gamma] \vdash_{ens} t_m : T_m\{t_1/v_1\ ;\ \ldots\ ;\ t_{m-1}/v_{m-1}\}_v\end{array} \tag{8.25}$$

From the definition of $[\![.]\!]$ we know that

$$(r' : \Pi v_1 : T_1 . \ldots \Pi v_m : T_m\{v_1/v_1 \; ; \; \ldots \; ; \; v_{m-1}/v_{m-1}\}_v).[\![S]\!]\{v_1/v_1 \; ; \; \ldots \; ; \; v_m/v_m\}_v \in [\![E]\!]$$

where $T =_{\beta\delta\iota\zeta} S\{t_1/v_1 \; ; \; \ldots \; ; \; t_m/v_m\}_v$

From 8.25, corollary 8.23 and the inductive hypothesis we conclude that the following CIC application is well-typed:

$$\begin{array}{l}[\![E]\!][v'_1 : [\![T'_1]\!] \; ; \; \ldots \; ; \; v'_r : [\![T'_r]\!]\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_{r-1}/v'_{r-1}\}_v \; ; \; [\![\Gamma]\!]\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v] \vdash_{CIC} \\ \quad (r' \; [\![t_1]\!] \; \ldots \; [\![t_m]\!])\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v : \\ \quad\quad [\![S]\!]\{v_1/v_1 \; ; \; \ldots \; ; \; v_m/v_m\}_v \, \{[\![t_1]\!]/v_1 \; ; \; \ldots \; ; \; [\![t_m]\!]/v_m\} \, \{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v\end{array}$$

which is equivalent to

$$\begin{array}{l}[\![E]\!][v'_1 : [\![T'_1]\!] \; ; \; \ldots \; ; \; v'_r : [\![T'_r]\!]\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_{r-1}/v'_{r-1}\}_v \; ; \; [\![\Gamma]\!]\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v] \vdash_{CIC} \\ \quad [\![r' \; [t_1/v_1 \; \ldots \; t_m/v_m]]\!]\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v : \\ \quad\quad [\![S]\!]\{[\![t_1]\!]/v_1 \; ; \; \ldots \; ; \; [\![t_m]\!]/v_m\}_v \, \{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v\end{array}$$

The thesis follows from the fact that

$$\begin{array}{l}[\![S]\!]\{[\![t_1]\!]/v_1 \; ; \; \ldots \; ; \; [\![t_m]\!]/v_m\}_v \, \{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v \\ \quad = [\![S\{t_1/v_1 \; ; \; \ldots \; ; \; t_m/v_m\}_v]\!]\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v \\ \quad =_{\beta\delta\iota\zeta} [\![T]\!]\{v'_1/v'_1 \; ; \; \ldots \; ; \; v'_r/v'_r\}_v\end{array}$$

which is a consequence of lemma 8.21 and corollary 8.23.

□

8.3.5 Applications of Explicit Named Substitutions

In this section we will investigate the benefits of introducing explicit named substitutions.

First of all we want to remark that in the proof of consistency of the calculus sections and scoping have played no role at all. The reader should not be surprised by this result: sections in Coq introduce a notion of time in the development of a theory. When a section is closed, a new time period starts. From now on the user will be able to add new results to its theory only outside the section, i.e. without getting any more benefit from the section/variables machinery. In other words, the theory developed inside the section is closed once and for all (unless, of course, the user discards the previously type-checked results and recompiles the input file, which is not acceptable for a distributed library in a collaborative environment). Outside the section, the user will be able to extend only the theory that was derived by the one inside the section through discharging.

Explicit named substitutions get rid of the notion of time (and scope) introduced by sections: any variable declaration becomes a global assumption. Namespace pollution is not a danger, as soon as every variable is given a fully qualified long name, as constants and axioms. The user is thus always free to extend a theory based on variables.

More concretely, the distinction between variables and axioms becomes even fuzzier: variables are special axioms that can be instantiated at any time, while axioms will never be instantiated. In our opinion, axioms should be removed from CIC in favor of variables and explicit named substitutions. The reason is that, in the usual mathematical practice, several sets of equivalent axioms are used at once: the mathematician dynamically chooses the set of axioms that is more appropriate to his own development. Explicit named substitutions easily allow to mimic this behaviour: the user defines the equivalent sets of axioms as sets of variable declarations and she also provides the proof of every axiom in a set using the axioms in the other. Later on she develops her own theory using just one set of axioms. To switch the set of axioms, she simply instantiates the axioms with the proofs she has already developed, introducing dependencies over the new set of axioms. For instance:

```
Variable times_comm: (n,m:nat) n * m = m * n.
Variable left_id: (n:nat) e * n = n.
Variable right_id: (n:nat) n * e = n.
(* The two axioms are equivalent *)
Theorem right_id_implies_left_id(right_id) : (n:nat)e * n = n.
 Proof.
  Rewrite times_comm; Exact right_id.
 Qed.

(* A theorem that uses the left identity axiom *)
Theorem t1(left_id) : T.
 Proof.
 ...
 Qed.

(* The same theorem based on the right identity axiom *)
Theorem t1'(right_id) := t1[right_id_implies_left_id / left_id].
```

In the next version of the Coq system the developers of Coq will introduce a module system [31], extremely similar to the one of the OCaml programming language [73]. It will then be possible to define module interfaces and functors. The user will be able to avoid the introduction of new axioms, simply defining them in a module interface and building her development as a functor. Changing the set of axioms means writing a functor that proves the equivalence of the two sets and building a new functor by means of functor composition.

The main difference between modules and explicit named substitutions will be the fact that with modules a whole set of declarations will be abstracted at once in the body of the functor. As a consequence, to use a theorem inside a functor the user will have to supply instantiations for every declaration in the functor domain, although the theorem could depend only on a subset of them. With explicit named substitutions, instead, the user is free to instantiate only the axioms she really cares about.

Functors and interfaces are related to the idea of a proof assistant as a compiler, and they allow to easily implement separate compilation. On the contrary, explicit named substitutions are closer to the idea of a proof assistant as a tool to contribute to a library of mathematical knowledge, which is freely browseable and extendable at any time, and closer to the usual working environment of a mathematician.

In Sect. 8.3.2 we already introduced as a form of syntactic sugaring the possibility to omit identity explicit named substitutions: if r depends on a variable v_1, then r is parsed as $r[v_1/v_1]$. This form of syntactic sugar is the one that allows to keep the concision of sectioning by omitting the instantiation of in-scope variables when calling functions that depend on the very same variable.

We can now easily introduce another form of syntactic sugaring: the user can enter explicit named substitutions in the context and in arbitrary positions in the terms, provided that we consider the substitutions to be instantaneous and we apply them at the end of the parsing phase. For example the script

```
Variable S: Set.
Variable op : S -> S -> S.

Definition neutral(S,op) : S -> Prop :=
 [e:S](n:S)(op e n) = n.

Theorem neutral_element_unique(S,op) :
 (e1,e2:S)(neutral e1) -> (neutral e2) -> e1 = e2.

S := nat ; op := mult;
Theorem neutral_equals_zero:
 (n:nat)(neutral n) -> n=0.
Proof.
 Intros n H.
 Exact (neutral_element_unique 0 n [n:nat](refl_equal nat 0) H).
Qed.
```

becomes, after the parsing phase and the application of the instantaneous explicit named substitution `S := nat ; op := mult`,

```
Variable S: Set.
Variable op : S -> S -> S.

Definition neutral(S,op) : S -> Prop :=
 [e:S](n:S)(op e n) = n.

Theorem neutral_element_unique(S,op) :
 (e1,e2:S)(neutral e1) -> (neutral e2) -> e1 = e2.

Theorem neutral_equals_zero:
 (n:nat)(neutral[nat/S ; op/mult] n) -> n=0.
Proof.
 Intros n H.
 Exact
  (neutral_element_unique[nat/S ; op/mult] 0 n
    [n:nat](refl_equal nat 0) H).
Qed.
```

Notice how this second form of syntactic sugaring corresponds to a functor instantiation in the approach based on modules. Note also that sections can be easily reintroduced in the syntax to delimit the scope of the instantaneous explicit substitution (but not the scope of the variables that is still global).

An interesting feature of explicit named substitutions over λ-abstractions is the possibility to instantiate the parameters in any order that preserves the well-typedness of the term. Example: in an environment where the variables $A : Set,\ a : A,\ H : a = a,\ B : Set,\ f : A \to B$ are declared, where the constant c depends over all of them and where g is a constant of type $A \to nat$, the following two terms are well-typed:

- $c[nat/B\ ;\ g/f]$ which is syntactic sugar for $c[A/A\ ;\ a/a\ ;\ H/H\ ;\ nat/B\ ;\ g/f]$
- $c[(refl\_equal\ A\ a)/H]$ which is syntactic sugar for
  $c[A/A; a/a; (refl\_equal\ A\ a)/H; B/B; f/f]$

The following term, instead, is not well-typed:

- $c[nat/B]$ which is syntactic sugar for $c[A/A\ ;\ a/a\ ;\ H/H\ ;\ nat/B\ ;\ f/f]$:
  f has type $A \to B$ while it is expected to have type $A \to nat$

The following term is also ill-typed

- $c[(\mathit{refl\_equal}\ \mathit{nat}\ 0)/H]$ which is syntactic sugar for
  $c[A/A\ ;\ a/a\ ;\ (\mathit{refl\_equal}\ \mathit{nat}\ 0)/H\ ;\ B/B\ ;\ f/f]$

However, we could easily implement a more intelligent parser that tries to instantiate the variables with implicit arguments, substituting them with the corresponding variable occurrence when the implicit term cannot be automatically inferred:

- for $c[(\mathit{refl\_equal}\ \mathit{nat}\ 0)/H]$ the parser would try to refine the term
  $c[?/A\ ;\ ?/a\ ;\ (\mathit{refl\_equal}\ \mathit{nat}\ 0)/H\ ;\ ?/B\ ;\ ?/f]$
  The first implicit argument would be instantiated with nat; the second one with 0; the instantiation of the last two implicit arguments would fail (but replacing the implicit argument with the corresponding variable would succeed). Thus the term would be parsed as the well-typed expression
  $c[\mathit{nat}/A\ ;\ 0/a\ ;\ (\mathit{refl\_equal}\ \mathit{nat}\ 0)/H\ ;\ B/B\ ;\ f/f]$

This characteristics of out-of-order parameter passing closely resembles the formal machinery introduced by Garrigue in [49] for the OCaml language. The relevant difference is that in OCaml every partial instantiation and every permutation of the order in which the parameters are instantiated are correct, since OCaml is not dependently typed. In our case, instead, the fact that the type of a parameter could depend on the value of a previous parameter introduces typing and convertibility constraints that allow only some of the permutations and partial instantiations. Notice also that the fact that a parameter type is dependent over previously introduced parameters is not a sufficient condition for the instantiation to be ill-typed: for example our second example above correctly instantiates H without having to instantiate A and a.

The possibility of passing parameters by name and not by position is quite appealing, especially when implicit or optional arguments are introduced in the language. Indeed this is one of the main motivations that pushed Garrigue to define labels for the OCaml language. The Coq developers introduced the possibility of referring to and instantiating parameters by name in the vernacular scripts. It is possible, for instance, to write `Apply eq_trans with x := 0.` in place of `Apply (eq_trans ? 0)`. Explicit named substitutions allow to understand this practice inside the calculus, whereas in Coq the named substitution is just a meta-level concept. As a consequence, we can benefit from optimized implementations of reduction machines that handle named substitutions directly, without having to compile them as traditional CIC applications (see 9.3.3).

Finally, Prof. Andrea Asperti has recently suggested yet another interpretation of the calculus extended with explicit named substitutions. According to this interpretation, every variable corresponds to an axiom and every object abstracted over a variable would behave as an object of the base calculus that has a λ-abstraction whose default value is the axiom. Any reference r to an object is always fully applied: if the reference occurs in an object that is abstracted over an optional argument ν and if r also has an optional argument that "corresponds" to ν, then r is applied to the ν in scope. All the other arguments of r are implicitly "applied" to their default value. For instance, let us consider the following development:

```
Variable A: Set.
Variable B(A) : A -> A.
Definition c() := B[nat/A].
Definition d(A,B) : A -> A := [x:A](B x).
Definition e() := d[nat/A]. (* Error *)
Definition f(A,B) := d.
Definition g() := d[nat/A ; S/B].
```

Its interpretation would be:

```
Axiom A: Set.
Axiom B: (X:Set default A)X -> X.
Definition c := (B ~X:nat).
(* In the definition of d B stands for its default (B ~X) *)
Definition d : (X:Set default A)(Y:X -> X default B)X -> X
 := [X:Set default A][Y:X -> X default B][x:X](Y x).
Definition e := (d ~X:nat). (* Error *)
Definition f := [X:Set default A][Y:X -> X default B](d ~X:X ~Y:Y).
Definition g := (d ~X:nat ~Y:S).
```

where we borrowed the syntax ~$label : value$ from OCaml (where it was introduced by Garrigue in [47]). Notice also that this interpretation is also preserved by simultaneous substitution. For instance, in the context above $d\{nat/A\}_\nu = d[A/A; B[A/A]/A]\{nat/A\}_\nu = d[nat/A; B[nat/A]/B]$. In the corresponding interpretation $d : \forall X : (Set \text{ default } A).\ \forall Y : (X \to X \text{ default } B).\ X \to X = \forall X : (Set \text{ default } A).\ \forall Y : (X \to X \text{ default } (B\ \tilde{}X : X)).\ X \to X$ and the interpretation of $d\{nat/A\}_\nu$ is $(d\ \tilde{}X : nat) : \forall Y : (nat \to nat \text{ default } (B\ \tilde{}X : A)).\ nat \to nat$.

Chapter 9

The Kernel

9.1 Architecture

The kernel of a proof assistant is the component that grants the correctness of the whole system. More precisely, it is responsible of managing the environment, granting the well-formedness invariant. Concretely, the environment can simply be implemented as an abstract data type with roughly only two operations: a function to fetch a definition or declaration from the environment; and a function to extend the environment with a new definition or declaration. The latter function checks against the environment the well-formedness of the definition or declaration, refusing to extend the argument with a definition or declaration that is not well-typed.

All the external layers of a proof assistant are aimed at creating the new objects that must enter the environment. Thus, a bug in an external layer of a proof assistant cannot harm the whole system, since the worst thing that can happen is the creation of a non well-typed object, that will be rejected by the kernel.

Concretely, the kernel must implement the typing judgement described in the previous chapter, together with the judgements and operations it depends on, that are lifting, substitution and conversion. Fig. 9.1 shows the organization in modules of the kernel of GTOPLEVEL, our proof assistant prototype.

The `CicEnvironment` module implements the environment data type. Lifting and substitution are implemented in the `CicSubstitution` module. The dependency over the module `CicEnvironment` is determined by the function that implements instantaneous named substitution: the substitution must retrieve the objects from the environment to know what are the named abstractions of the object. The implementation of the module is absolutely obvious, since lifting and substitution are defined as syntax oriented judgements describing an already efficient algorithm.

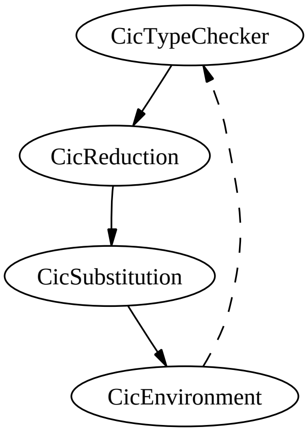

Figure 9.1: The kernel modules.

Reduction and conversion are implemented in the `CicReduction` module. Although we presented the two judgements in a syntax oriented formulation, the algorithms that corresponds to the judgements are not efficient at all. Thus we are forced to implement more complex algorithms, based on (weak head) reduction machines. In Sect. 9.2 we describe the syntax oriented judgement for checking conversion that is the one implemented in GTOPLEVEL. In Sect. 9.2 we present a new reduction machine for CIC with explicit named substitutions that is parametric in the argument passing strategy. The machine is implemented in GTOPLEVEL as a functor, whose input is a module that defines the reduction strategy.

The type checking rules described in the previous chapter were not syntax oriented. Thus they cannot be directly implemented in an efficient way (i.e. avoiding backtracking). Thus in Sect. 9.5 we present a syntax oriented version of the type checking rule, that is the version implemented in GTOPLEVEL. The main novelty of the type checking judgement is our peculiar management of the environment, that is described in Sect. 9.4 and that is aimed to blur the distinction between the environment of the proof assistant and the distributed library of documents of HELM. Briefly, we allow the type checker to ask the environment module for an object that is in the library but that is not yet in the environment. The environment module reconstructs the minimal environment extension that allows to type-check the required object. To extend the environment, the environment module `CicEnvironment` has to be allowed to invoke the type checking functions of the `CicTypeChecker` module. Since no cyclic module definitions are allowed in OCaml, the dependency of `CicEnvironment` on `CicTypeChecker` (shown in fig. 9.2 as a dotted line) is broken by making the type checker module register a callback to `CicEnvironment`.

9.2 The Conversion Algorithm

One of the peculiar features of the Calculus of (Co)Inductive Constructions is the integration of computation with reasoning by means of the convertibility rule:

Conv

$$\frac{E[\Gamma] \vdash T_1 : s \qquad E[\Gamma] \vdash t : T_2 \qquad E[\Gamma] \vdash T_1 =_{\beta\delta\iota\zeta} T_2}{E[\Gamma] \vdash t : T_1}$$

By definition of $=_{\beta\delta\iota\zeta}$, the easiest way to test whether two terms are convertible consists in reducing both of them to normal form, and check whether the two normal forms are α-convertible. Unfortunately, this approach is very inefficient since it involves reductions of pairs of terms whose convertibility can be tested with more witty algorithms avoiding their complete reduction. For instance, to check the convertibility of $n!$ with $n * (n-1)!$ we are not obliged to compute the two normal forms; we can simply perform one step of δ-reduction on the first term to obtain a couple of α-convertible terms.

Since the expensive operation involved in conversion checking is reduction, one solution is to look for heuristics that perform as few reduction steps as possible. Obtaining a complete algorithm is not easy though. In the literature [57] there exists one pretty standard algorithm that allows to check the convertibility of two terms using only weak head reduction steps, i.e. reducing only the redexes that are in head position in the term. This is also the algorithm that we implemented in GTOPLEVEL.

Formally, we define a new (almost) syntax oriented judgement, $E[\Gamma] \vdash T_1 \downarrow T_2$ that holds if and only if $T_1 =_{\beta\delta\iota\zeta} T_2$ in E and Γ. The whd function used in the judgement can be any correct implementation of weak head reduction. In Sect. 9.3 we will describe the reduction machine that we use in GTOPLEVEL to implement the whd function.

This whole section as well as the judgement proposed is based on [57].

Conv-Rel-Rel

$$\frac{}{E[\Gamma] \vdash x \downarrow x}$$

Conv-Var-Var-IdSubst

$$\frac{\sigma \text{ is syntactically the identity substitution}}{E[\Gamma] \vdash v[\sigma] \downarrow v[\sigma]}$$

Conv-Var-Var

$$\frac{\begin{array}{c}(E[\Gamma] \vdash t_i \downarrow t'_i)_{i=1,\ldots,n} \\ \text{none of the explicit named substitutions is syntactically an identity substitution}\end{array}}{E[\Gamma] \vdash v[t_1/v_1 \; ; \; \ldots \; ; \; t_n/v_n] \downarrow v[t'_1/v_1 \; ; \; \ldots \; ; \; t'_n/v_n]}$$

Conv-Const-Const

$$\frac{(E[\Gamma] \vdash t_i \downarrow t'_i)_{i=1,\ldots,n}}{E[\Gamma] \vdash c[t_1/v_1\ ;\ \ldots\ ;\ t_n/v_n] \downarrow c[t'_1/v_1\ ;\ \ldots\ ;\ t'_n/v_n]}$$

Conv-Constr-Constr

$$\frac{(E[\Gamma] \vdash t_i \downarrow t'_i)_{i=1,\ldots,n}}{E[\Gamma] \vdash k[t_1/v_1\ ;\ \ldots\ ;\ t_n/v_n] \downarrow k[t'_1/v_1\ ;\ \ldots\ ;\ t'_n/v_n]}$$

Conv-Ind-Ind

$$\frac{(E[\Gamma] \vdash t_i \downarrow t'_i)_{i=1,\ldots,n}}{E[\Gamma] \vdash i[t_1/v_1\ ;\ \ldots\ ;\ t_n/v_n] \downarrow i[t'_1/v_1\ ;\ \ldots\ ;\ t'_n/v_n]}$$

Conv-Sort-Sort

$$\frac{}{E[\Gamma] \vdash s \downarrow s}$$

Conv-Prod-Prod

$$\frac{E[\Gamma] \vdash T \downarrow T' \qquad E[\Gamma; x : T] \vdash t \downarrow t'\{x/x'\}}{E[\Gamma] \vdash \Pi x : T.t \downarrow \Pi x' : T'.t'}$$

Conv-Lambda-Lambda

$$\frac{E[\Gamma] \vdash T \downarrow T' \qquad E[\Gamma; x : T] \vdash t \downarrow t'\{x/x'\}}{E[\Gamma] \vdash \lambda x : T.t \downarrow \lambda x' : T'.t'}$$

Conv-LetIn-LetIn[1]

$$\frac{E[\Gamma] \vdash t_1 \downarrow t'_1 \qquad E[\Gamma; x := t_1] \vdash t_2 \downarrow t'_2\{x/x'\}}{E[\Gamma] \vdash \mathrm{let}\ x := t_1\ \mathrm{in}\ t_2 \downarrow \mathrm{let}\ x' := t'_1\ \mathrm{in}\ t'_2}$$

Conv-Appl-Appl

$$\frac{(E[\Gamma] \vdash t_i \downarrow t'_i)_{i=1,\ldots,n}}{E[\Gamma] \vdash (t_1\ \ldots\ t_n) \downarrow (t'_1\ \ldots\ t'_n)}$$

Conv-Cast-Cast

$$\frac{E[\Gamma] \vdash t_1 \downarrow t_2 \qquad E[\Gamma] \vdash T_1 \downarrow T_2}{E[\Gamma] \vdash (t_1 :: T_1) \downarrow (t_2 :: T_2)}$$

Conv-Case-Case[2]

$$\frac{E[\Gamma] \vdash t \downarrow t' \qquad E[\Gamma] \vdash T \downarrow T' \qquad (E[\Gamma; \overline{x^i : T^i}] \vdash f_i \downarrow f'_i\{\overline{x^i/x'^i}\})_{i=1,\ldots,n_l}}{E[\Gamma] \vdash \left(\begin{array}{l} < T > \mathrm{CASES}_{i_l}\ t\ \mathrm{OF} \\ \quad k^l_1\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k^l_{n_l}\ \overline{x^n} \Rightarrow f_{n_l} \\ \mathrm{END} \end{array}\right) \downarrow \left(\begin{array}{l} < T' > \mathrm{CASES}_{i_l}\ t'\ \mathrm{OF} \\ \quad k^l_1\ \overline{x'^1} \Rightarrow f'_1 \mid \ldots \mid k^l_{n_l}\ \overline{x'^n} \Rightarrow f'_{n_l} \\ \mathrm{END} \end{array}\right)}$$

[1] According to our definition of context, $x := t_1$ is not a valid context element, since the type of t_1 is not specified. Notice, however, that the type of a term defined in the context is never used for reduction and conversion. Thus we can simply relax our definition of context to allow definitions without a type or we can pick a dummy term for the missing information. In the implementation we chose the first solution in order to detect bugs in the implementation that would make our code use the fake type.

[2] The type T^i of the x^i constructor can either be a fake type (that is never used in reduction and conversion) or it can be the real type of x^i that is obtained by browsing the definition of i_l.

Conv-Fix-Fix

$$\frac{(E[\Gamma] \vdash T_i \downarrow T_i')_{i=1,\dots,m} \qquad (E[\Gamma\ ;\ x_1 : T_1\ ;\ \dots\ ;\ x_m : T_m] \vdash f_i \downarrow f_i'\{x_1/x_1'\ ;\ \dots\ ;\ x_m/x_m'\})_{i=1,\dots,m}}{\begin{array}{ll} E[\Gamma] \vdash & \mathrm{FIX}_j\ \{x_1/n_1 : T_1 := f_1\ ;\ \dots\ ;\ x_m/n_m : T_m := f_m\} \downarrow \\ & \mathrm{FIX}_j\ \{x_1'/n_1 : T_1' := f_1'\ ;\ \dots\ ;\ x_m'/n_m : T_m' := f_m'\} \end{array}}$$

Conv-CoFix-CoFix

$$\frac{(E[\Gamma] \vdash T_i \downarrow T_i')_{i=1,\dots,m} \qquad (E[\Gamma\ ;\ x_1 : T_1\ ;\ \dots\ ;\ x_m : T_m] \vdash f_i \downarrow f_i'\{x_1/x_1'\ ;\ \dots\ ;\ x_m/x_m'\})_{i=1,\dots,m}}{\begin{array}{ll} E[\Gamma] \vdash & \mathrm{COFIX}_j\ \{x_1 : T_1 := f_1\ ;\ \dots\ ;\ x_m : T_m := f_m\} \downarrow \\ & \mathrm{COFIX}_j\ \{x_1' : T_1' := f_1'\ ;\ \dots\ ;\ x_m' : T_m' := f_m'\} \end{array}}$$

Conv-Whd-Whd

$$\frac{whd(E[\Gamma], t_1) = t_1' \qquad whd(E[\Gamma], t_2) = t_2' \qquad E[\Gamma] \vdash t_1' \downarrow t_2'}{E[\Gamma] \vdash t_1 \downarrow t_2}$$

The judgement $E[\Gamma] \vdash t_1 \downarrow t_2$ that we have just described does not completely define a conversion algorithm, since it is not completely syntax oriented. Indeed, all the rules but Conv-Whd-Whd are syntax oriented, but the rule Conv-Whd-Whd can always be applied. The simpler way to obtain an algorithm is to introduce backtracking and make the algorithm explore every possibility. We can do better, though: the (constructive) proof of completeness of the algorithm (theorem 9.5) shows the existence of a very simple strategy that grants completeness without the need of backtracking. The strategy consists in alternating reduction steps (i.e. applications of the rule Conv-Whd-Whd) to context closure phases (where only the rules that are different to Conv-Whd-Whd are applied). As soon as no rule different from Conv-Whd-Whd can be applied, the context closure phase ends and a reduction step must be performed.

The strategy is complete, and reduction steps, that are the most expensive, are delayed if the other rules can be used to reduce the size of the term that will eventually be reduced. Notice, however, that the strategy is not optimal. Indeed, once the Conv-Whd-Whd rule is applied, the weak head normal form of both members is computed. Nevertheless, it may be possible that just performing a few of reduction steps on the two members would produce two α-convertible terms.

Moreover, the complete strategy always starts with a reduction step, whereas empirical observations show that the largest part of the conversion test involved in proof checking is on terms that are already α-convertible; these terms are nevertheless put in weak head normal form by the algorithm. The following heuristic, though, seems to work pretty well in practice: skip the initial reduction step; if conversion fails, perform the initial reduction step and repeat once. The worst case is the (extremely) unlikely case of two deep terms that are α-convertible but for a couple of terms that is at the maximum nesting level, and such that the two initial terms are convertible but no subterm of them is. In this case the heuristic performs a number of operations that is exponential in the number of operations of the complete strategy. Thanks to the law of diminishing returns, in practice the heuristic is extremely effective in the average case.

Due to the existence of the complete deterministic strategy, we prefer to adopt the terminology "conversion algorithm" in place of "conversion judgement" in the rest of the dissertation.

Theorem 9.1 (Conversion algorithm soundness) *Let* E *be an environment and* Γ *be a context and* M*,* N *be two terms. If* $E[\Gamma] \vdash M \downarrow N$ *then* $M =_{\beta\delta\iota\zeta} N$ *in the environment* E *and context* Γ*.*

Proof: The proof is by induction on the height of the derivation of $E[\Gamma] \vdash M \downarrow N$. All the cases but Conv-Whd-Whd are trivial applications of the recursion hypotheses and the context closure property of conversion. The case Conv-Whd-Whd is also trivial: by inductive hypothesis $t_1' =_{\beta\delta\iota\zeta} t_2'$; by the correctness property of the whd function $t_i \rhd^* whd(E[\Gamma], t_i)$ for $i \in \{1, 2\}$; hence the thesis. $\Box$

In order to prove the completeness of the conversion algorithm, we need first to establish a technical lemma.

Definition 9.2 (Term context associated to a weak head normal form) *Let* T *be a term and let* T_n *be its weak head normal form. With* T^* *we denote the term context obtained from* T_n *by replacing with placeholders* $\Box$ *all the subterms that have not been forced to be in weak head normal form during the reduction of* T *to* T_n*.*

Example 9.3 Let T be the term $(\lambda x.(x\ (k\ M\ x))\ (\lambda y.y))$. The weak head normal form of T is $T_n = (k\ M\ (\lambda y.y))$. The term context associated to the weak head normal form of T is $T^* = (k\ \Box\ \Box)$, since none of the arguments of k in T_n has been forced to be in weak head normal form during the reduction $T \rhd^+ T_n$.

Notice that the term context associated to a weak head normal form is independent from the reduction steps, and it could easily be inductively defined over the structure of the term.

Lemma 9.4 *For each couple of terms* M*,* N*, if if* $M =_{\beta\delta\iota\zeta} N$ *then* M^* *is* α*-convertible with* N^**.*

Proof: We proceed by induction over the structure of M^*, remembering that it must be free of weak head redexes. Since the cases are all similar, let us show as an example only the case of an application: $M^* = (M_1\ \Box)$ where M_1 is not a λ-abstraction. Since M_n is convertible with N_n and since they both cannot be redexes, we must have $N_n = (N_1\ N_2)$ and then $N^* = (N_1\ \Box)$. By induction hypothesis we know that M_1 is α-convertible with N_1. Thus M^* is α-convertible with N^*. $\Box$

We are now ready to prove the completeness of the conversion algorithm. Before doing that, though, we want to provide the intuition about the peculiar notion of term context we just introduced.

Our conversion algorithm is a slight adaptation of the one proposed in [57]. Notwithstanding this, the proof of completeness proposed here is much longer and complex. The reason is that we spotted a serious error in the proof of Harper and Pollack. Indeed, the authors of the paper proceed by induction over the shape of the two weak head normal forms M_0 and N_0 of M and N. A weak head normal form has subterms that are not in weak head normal form (like the arguments of an application, for instance). Nevertheless, the authors assume the inductive hypothesis to hold also for these subterms.

To solve the problem, we have introduced the notion of term context M^* associated to the weak head normal form M_n of M. The term context is the largest term prefix of M_n whose subterms are all guaranteed to be in weak head normal form. Substituting the terms with their term contexts, the proof of Harper and Pollack becomes correct (but it proves something different from before). The proof is also constructive. Its computational content corresponds to the context closure phases of the deterministic strategy we proposed. To complete the proof of correctness, we have to fill the gap between term contexts and terms in weak head normal form, by processing the subterms that are in the normal forms but that are not in the corresponding term context. The proof is rather technical, but its computational content is clear: it is the application of the Conv-Whd-Whd step of the deterministic strategy.

Theorem 9.5 (Conversion algorithm completeness) *) Let* E *be an environment and* Γ *be a context and* M, N *be two terms. If* $M =_{\beta\delta\iota\zeta} N$ *in the environment* E *and context* Γ*, then* $E[\Gamma] \vdash M \downarrow N$*.*

Proof: By definition of $=_{\beta\delta\iota\zeta}$, we know that M and N must be strongly normalizable and that the two normal forms of M and N must be α-convertible. By the strong normalization and confluence properties of the calculus, we also know that the normal form of M and N are unique and that any reduction strategy will terminate. Thus let us consider the two reduction sequences $M = M_0 \rhd^* M_1 \rhd^* \ldots \rhd^* M_n$ and $N = N_0 \rhd^* N_1 \rhd^* \ldots \rhd^* N_{n'}$ such that M_{i+1} and N_{i+1} are obtained respectively from M_i and N_i by locating all the subterms (that form the set $\dagger M$) that have not been forced to be in weak head normal form and by replacing them (by means of context closures rules) with their weak head normal forms (that form the corresponding set $\ddagger M$). M_n and $N_{n'}$ are respectively the normal forms of M and N.

The first step of the proof consists in proving by induction on i that M_i^* and N_i^* are α-convertible and also that for every pair of corresponding subterms $t_1 \in \ddagger M_{i-1}$ and $t_2 \in \ddagger N_{i-1}$ it is true that t_1^* is α-convertible with t_2^*.

Base case: $M_0^* = N_0^* = \Box$. The second part of the statement is not defined in the base case.

Inductive case: $M_i \rhd^* M_{i+1}$ and $N_i \rhd^* N_{i+1}$. By inductive hypothesis we know that M_i^* and N_i^* are α-convertible. By definition of M_{i+1}, M_{i+1} is obtained by M_i by replacing all the subterms in $\dagger M_i$ with the corresponding subterm in $\ddagger M_i$. The same holds for N_i. Since M and N are convertible, M_i and N_i must also be convertible. Moreover, let us consider a generic couple of corresponding subterms $s_1 \in \dagger M_i$ and $s_2 \in \dagger N_i$. Let t_1, t_2 be the weak head normal forms of s_1 and s_2. Since M_i^* and N_i^* are free of redexes, we must conclude $s_1 =_{\beta\delta\iota\zeta} s_2$ and thus also $t_1 =_{\beta\delta\iota\zeta} t_2$. By lemma 9.4 we have t_1^* is α-convertible with t_2^*. Using again the inductive hypothesis that states the α-convertibility of M_i^* with N_i^* we conclude that M_{i+1}^* is α-convertible with N_{i+1}. Hence the thesis.

So far we have proved that M_i^* is α-convertible with N_i^* and that for each i and for each couple of terms $t_1 \in \ddagger M_{i-1}$ and $t_2 \in \ddagger N_{i-1}$ we have t_1^* α-convertible with t_2^*. Thus $n = n'$, since assuming $n < n'$ (without any loss of generality) implies $M_n = M_n^* \equiv_\alpha N_n^*$ that is impossible since M_n^* has no placeholders while N_n^* must have at least one placeholder left.

We are now able to easily construct a derivation of $E[\Gamma] \vdash M_0 \downarrow N_0$. Applying once the rule Conv-Whd-Whd we can reduce it to $E[\Gamma] \vdash M_1 \downarrow N_1$. By recursion over the structure of M_1^* (that is α-convertible with N_1^*) and by using only rules that are different from Conv-Whd-Whd we can reduce the thesis to the set of new goals $E[\Gamma] \vdash s_1 \downarrow s_2$ for each couple of corresponding terms $s_1 \in \dagger M_1$ and $s_2 \in \dagger N_1$. Repeating the construction for the goals in the sets $(\dagger M_1, \dagger N_1), \ldots, (\dagger M_{n-1}, \dagger N_{n-1})$, alternating applications of the Conv-Whd-Whd rule to deal with reduction steps and applications of all the other rules to deal with contextual rules, we can complete the derivation. □

9.3 A Functor for Reduction Machines

In the previous section we specified an algorithm for testing conversion between terms assuming the existence of a correct weak head reduction function. Thus, we still have several degrees of freedom in the implementation of the weak head reduction procedure. In particular, we can either implement the function in an highly inefficient way by following closely the definition of reduction; or we can implement one or more versions of the usual reduction machines for the λ-calculus (the CAM, the SECD, Krivine's Lazy machine, etc.), modified to consider the whole CIC calculus.

One of the aim of the developments of our own proof assistant is the creation of a test-bench that could be used to experiment and compare new and old solutions in the realm of proof assistants implementation. The system should also be used in the realm of didactics, for teaching (undergraduate) students the basics knowledge about the development of a real-world proof assistant. Thus, according to the requirement Req8 (page 94), we are interested in providing a framework to test and study the influence of the value-passing strategy over the overall reduction and type-checking time. In the next section we will describe a generalization of the Krivine lazy machine that is obtained by abstracting the parameter passing strategy (that is call-by-name for the original machine). The machine is modified in order to cover the whole CIC calculus extended with explicit named substitutions. The machine is implemented in OCaml as a functor, whose input is a module that describes the parameters passing strategy.

9.3.1 The Generalized Krivine's Abstract Machine

Before describing our generalization of the Krivine's Abstract Machine, we briefly review the original machine rules, as described in [9].

The Krivine's Abstract Machine (KAM)

Krivine proposed a very simple environment machine for lazy evaluation of the λ-calculus in [70]. The categorical soundness of the machine is proved in [9]. Sylvain Lippi proves that the KAM correctly implements head reduction[3] in [75].

In the KAM, both the term M to be evaluated and the environment ξ are represented by graphs. We have two points A and B for accessing these graphs. The first graph remains fixed during the computation, while the second one keeps growing. As usual, the recursive calls to the evaluation are implemented by a stack, where we accumulate closures, that is pairs of pointers of the form (B, A). We assume variables to be coded using De Brujin indexes.

Formally, we can give the following recursive definition of closures, environments and stacks:

Definition 9.6 (Closures, environments and stacks of the Krivine's lazy abstract machine)
Closure $\stackrel{\text{def}}{=}$ *Environment* $*$ *Term*
Environment $\stackrel{\text{def}}{=}$ *Closure List*
Stack $\stackrel{\text{def}}{=}$ *Closure List*
KAM $\stackrel{\text{def}}{=}$ *Environment* $*$ *Term* $*$ *Stack*

The reduction rules are given in 9.1 The most characterizing reduction rule is the one associated to application: when an application MN is met, we build a closure $(\xi; N)$ that is pushed on top of the stack. Thus the computation of N is not performed, but delayed to the first occurrence in head position of a variable bound to the closure: when the occurrence is met, the environment is traversed (first rule) until the closure (ξ, N) is retrieved (second rule); then the environment ξ and the term N substitute the current environment and the current term.

[3] Danos and Regnier in [36] introduce head linear reduction and claim that the reduction strategy implemented by the KAM is not really head reduction, but head *linear* reduction.

Table 9.1: Krivine's lazy abstract machine (call-by-name)

Before			After		
Environment	Code	Stack	Environment	Code	Stack
$\xi.\alpha$	n+1	S	ξ	n	S
$\xi.(\xi'; N)$	0	S	ξ'	N	S
ξ	$\lambda.M$	$\alpha.S$	$\xi.\alpha$	M	S
ξ	MN	S	ξ	M	$(\xi;N).S$

The initial state of the machine to reduce a term t is $(\emptyset, t, \emptyset)$. The final states of the machine are identified with the stuck states: $(\emptyset, n, S)$ (that can be read back as the application of an unbound variable to the terms in S) and $(\xi, \lambda.M, \emptyset)$ (that can be read back as a λ-abstraction whose body is obtained by "applying the substitution ξ" to M).

The Krivine's lazy abstract machine described by the rules in Fig. 9.1 implements a call-by-name strategy. A call-by-need version of the reduction machine can easily be obtained by making the environment graph mutable, replacing the stack with a stack of stacks and changing the second rule. However, we will not detail this version of the machine here.

The Generalized KAM (GKAM)

In this section we want to define a generalization of the KAM (called GKAM) obtained by abstracting the KAM over the argument passing strategy. In particular, the GKAM can be instantiated with several argument passing strategies, yielding, for instance, machines for the call-by-need, call-by-name and call-by-value λ-calculi. We are interested more in the genericity of the machine than in its efficiency. For instance, we do not reject non tail recursive machines. The straightforward implementation of the GKAM in OCaml consists in a functor whose input is a module describing the argument passing strategy and whose output is the corresponding KAM-like environment machine.

In the KAM, stacks and environments are lists of closures. We define the GKAM as a generalization of the KAM obtained by abstracting environments and stacks:

Definition 9.7 (Closures, environments and stacks of the GKAM)
Let EItem and SItem be two datatypes.
Environment $\stackrel{\text{def}}{=}$ *EItem List*
Stack $\stackrel{\text{def}}{=}$ *SItem List*
Closure $\stackrel{\text{def}}{=}$ *Environment* $*$ *Term*
GKAM $\stackrel{\text{def}}{=}$ *Environment* $*$ *Term* $*$ *Stack*
to_stack : Closure $\hookrightarrow$ *SItem*

to_env : SItem ↪ EItem
from_env : EItem ↪ GKAM
For the sake of stating the correctness of the GKAM, we also assume the existence of the following two read-back total functions (that will not play any role in the actual reduction):
$\mathcal{T}_s$*: SItem → Term*
$\mathcal{T}_e$*: EItem → Term*
We also assume the existence of a strict and well-founded order
$<_E \subseteq$ *Environment × Environment*
such that $\forall \xi : \mathit{Environment} . \forall \alpha : \mathit{EItem} . \xi <_E \xi.\alpha$

To make the GKAM correctly implement a reduction strategy (a triplet $(\mathtt{to\_stack}, \mathtt{to\_env}, \mathtt{from\_env})$), we need to impose some constraints over $\mathtt{to\_stack}$, $\mathtt{to\_env}$ and $\mathtt{from\_env}$. To express the constraints, we need to define the read-back function $\mathcal{T}$: GKAM → Term:

Definition 9.8 *Read-back function* $\mathcal{T}(\_,\_,\_)$

$$\mathcal{T}([\alpha_1, \ldots, \alpha_n], t, [\beta_1, \ldots \beta_m]) \stackrel{\text{def}}{=} (t\{\mathcal{T}_e(\alpha_1)/1 \; ; \; \ldots \; ; \; \mathcal{T}_e(\alpha_n)/n\} \; \mathcal{T}_s(\beta_1) \; \ldots \; \mathcal{T}_s(\beta_m)))$$

We also define the following restriction of the read-back function that is often useful:

Definition 9.9 *Read-back function* $\mathcal{T}(\_,\_)$

$$\mathcal{T}(\sigma, t) \stackrel{\text{def}}{=} \mathcal{T}(\sigma, t, \emptyset)$$

Fact 9.10 $\forall (\sigma, t), (\sigma', t') : \mathit{Closure} . \forall S : \mathit{Stack} . \mathcal{T}(\sigma, t) = \mathcal{T}(\sigma', t') \to \mathcal{T}(\sigma, t, S) = \mathcal{T}(\sigma', t', S)$

We can now state the three GKAM Correctness Conditions and the GKAM Liveliness Condition:

Definition 9.11 *GKAM Correctness Conditions:*

1. $\forall (\xi, t) : \mathit{Closure} . \mathcal{T}(\xi, t) \rhd^* \mathcal{T}_s(\mathit{to\_stack}\ (\xi, t))$
2. $\forall \alpha : \mathit{SItem} . \mathcal{T}_s(\alpha) \rhd^* \mathcal{T}_e(\mathit{to\_env}\ \alpha)$
3. $\forall \alpha : \mathit{EItem} . \mathcal{T}_e(\alpha) \rhd^* \mathcal{T}(\mathit{from\_env}\ \alpha)$

Definition 9.12 *GKAM Liveliness Condition*
hd ∘ from_env ∘ π_1 *is strictly decreasing (according to the ordering* $<_E$*)*

Table 9.2: GKAM reduction rules

Before			After		
Environment	Code	Stack	Environment	Code	Stack
$\xi.\beta$	n+1	S	ξ	n	S
$\xi.\beta$	0	S	π_1(from_env β)	π_2(from_env β)	π_3(from_env β).S
ξ	$\lambda.M$	$\alpha.S$	ξ.(to_env α)	M	S
ξ	MN	S	ξ	M	(to_stack (ξ;N)).S

From now on we assume that all the strategies considered satisfy the conditions above.

We impose no further restriction on the three computable (partial) functions from_env, to_env and to_stack. In particular, we will allow ourselves to implement them by means of instantiations of new reduction machine (i.e. recursive calls to the main function of the reduction machine).

The reduction rules of the GKAM are given in Table 9.2 Notice that all the rules are generalization of the corresponding rule for the KAM. The initial and final states of the GKAM are equal to those of the KAM. The following fundamental fact about the behaviour of the read back function on the initial states trivially holds:

Fact 9.13 *For each term* t, $\mathcal{T}(\emptyset, t, \emptyset) = t$.

Let us now consider the role of the correctness conditions. Combining the three correctness conditions together, we obtain $\forall(\xi, t)$: Closure .$\mathcal{T}(\xi, t) \rhd^*$(from_env(to_env(to_stack(ξ, t)))). Intuitively, the derived condition says that the chosen parameter passing strategy is only allowed to reduce — by means of $\mathtt{to\_stack} \circ \mathtt{to\_env} \circ \mathtt{from\_env}$ — the actual argument — the closure (ξ, t) — before substituting it for the corresponding DeBrujin index in head position. It will be no surprise that several reduction strategies can be seen as instantiations of the GKAM: for instance, the call-by-name strategy is the strategy that does not perform any reduction at all, whereas the call-by-value strategy is the strategy that maximally reduces the term (i.e. until a value is found).

The three correctness conditions state that the actual argument can be reduced by the reduction strategy not only when it is pushed on the stack (in preparation of a β-reduction), but also when it is moved to the environment (i.e. when β-reduction is actually performed) and when it is fetched from the environment (i.e. when an occurrence of the formal argument is brought in head-position). The additional freedom on the exact time when reduction is performed allows us to test several variations of the classical reduction strategies (e.g. the arguments a constant is applied to in a value can be either reduced to values or left as expressions).

We can now state and prove the correctness of the GKAM:

Theorem 9.14 *GKAM correctness*
$\forall(\xi, t, S), (\xi', t', S')$: *GKAM . if* (ξ, t, S) *evolves into* (ξ', t', S') *then* $\mathcal{T}(\xi, t, S) \rhd^* \mathcal{T}(\xi', t', S')$.

Proof: We proceed by cases on the GKAM rule that is fired:

- First rule: $\mathcal{T}(\xi.\beta, n+1) = \mathcal{T}(\xi, n)$ is trivial.

- Second rule:

$$\begin{aligned}
&\mathcal{T}(\xi.\beta, 0, S) \\
&\quad = (\mathcal{T}_e(\beta)\ \mathcal{T}_s(S)) \\
&\quad \rhd^*(\mathcal{T}(\text{from_env}\ \beta)\ \mathcal{T}_s(S)) \\
&\quad = (\pi_2(\text{from_env}\ \beta)\{\mathcal{T}_e(\pi_1(\text{from_env}\ \beta))\} \\
&\qquad\quad \mathcal{T}_s(\pi_3(\text{from_env}\ \beta))\mathcal{T}_s(S)) \\
&\quad = \mathcal{T}(\pi_1(\text{from_env}\ \beta), \pi_2(\text{from_env}\ \beta), \pi_3(\text{from_env}\ \beta).S)
\end{aligned}$$

  The second rewriting step is justified by the third GKAM Correctness Condition.

- Third rule:

$$\begin{aligned}
&\mathcal{T}(\xi, \lambda.M, \alpha.S) \\
&\quad = (\lambda.M\{\mathcal{T}_e(\xi)\}\ \mathcal{T}_s(\alpha)\ \mathcal{T}_s(S)) \\
&\quad \rhd(M\{\mathcal{T}_s(\alpha)/1; \mathcal{T}_e(\xi)\}\ \mathcal{T}_s(S)) \\
&\quad \rhd^*(M\{\mathcal{T}_e(\text{to_env}\ \alpha)/1; \mathcal{T}_e(\xi)\}\ \mathcal{T}_s(S)) \\
&\quad = \mathcal{T}(\xi.(\text{to_env}\ \alpha), M, S)
\end{aligned}$$

  The third rewriting step is justified by the second GKAM Correctness Condition.

- Fourth rule:

$$\begin{aligned}
&\mathcal{T}(\xi, MN, S) \\
&\quad = (MN\{\mathcal{T}_e(\xi)\}\ \mathcal{T}_s(S)) \\
&\quad = (M\{\mathcal{T}_e(\xi)\}\ N\{\mathcal{T}_e(\xi)\}\ \mathcal{T}_s(S)) \\
&\quad = (M\{\mathcal{T}_e(\xi)\}\ \mathcal{T}(\xi, N)\ \mathcal{T}_s(S)) \\
&\quad \rhd^*(M\{\mathcal{T}_e(\xi)\}\ \mathcal{T}_s((\text{to_stack}(\xi, N)))\ \mathcal{T}_s(S)) \\
&\quad = \mathcal{T}(\xi, M, (\text{to_stack}(\xi, N)).S)
\end{aligned}$$

  The fourth rewriting step is justified by the first GKAM Correctness Condition.

□

This correctness theorem guarantees that any move of the machine correctly performs zero or more reduction steps. It does not guarantee any liveliness property: in particular there could be an infinite sequence of machine steps that does not reduce the term and that simply corresponds to the identity function. Liveliness is granted by the following theorem, that uses the GKAM Liveliness Condition:

Theorem 9.15 *Liveliness*
There is no infinite sequence of GKAM states $(\xi_0, t_0, S_0), (\xi_1, t_1, S_1), \ldots$ *such that for each index* i (ξ_i, t_i, S_i) *moves into* $(\xi_{i+1}, t_{i+1}, S_{i+1})$ *and* $\mathcal{T}(\xi_i, t_i, S_i) = \mathcal{T}(\xi_{i+1}, t_{i+1}, S_{i+1})$

Proof: We proceed by absurdum. Let us assume that such an infinite sequence exists. From the proof of correctness of the GKAM we know that if (ξ_i, t_i, S_i) moves according to the third reduction rule, then $\mathcal{T}(\xi_i, t_i, S_i) \rhd^+ \mathcal{T}(\xi_{i+1}, t_{i+1}, S_{i+1})$. Thus there is an index h such that for each $i > h$ (xi_i, t_i, S_i) moves to $(\xi_{i+1}, t_{i+1}, S_{i+1})$ using either the first, the second or the fourth rule. Let us consider the following strict and well-founded order $<_C$ over closures (and GKAM configurations, when we ignore the last component):

$$(\xi, t) <_C (\xi', t') \text{ if } \xi <_E \xi' \text{ or if } \xi = \xi' \text{ and } t \text{ is structurally smaller than } t'$$

We now prove that the three rules in the infinite sequence are all strictly decreasing functions from continuation to continuation according to the $<_C$ ordering relation. We reach the absurdum by noting that, although the infinite sequence is strictly decreasing, the order is well-founded.

- First rule: $(\xi.\beta, n+1, S)$ moves into (ξ, n, S)
  $\xi <_E \xi.\beta$ is a requirement for every GKAM strategy.

- Second rule: $(\xi.\beta, 0, S)$ moves into $(\pi_1(\text{from\_env } \beta), \pi_2(\text{from\_env } \beta), \pi_3(\text{from\_env } \beta).S)$
  $(\text{hd} \circ \text{from\_env} \circ \pi_1)(\xi.\beta) <_E (\xi.\beta)$ because of the GKAM liveliness condition.

- Third rule: (ξ, MN, S) moves into $(\xi, M, (\text{to\_stack}(\xi, N)).S)$
  M is structurally smaller than MN.

□

Corollary 9.16 (Termination) *If* `to_stack`, `to_env` *and* `from_env` *are total functions and if a term* t *is strongly normalizable, then the execution of the GKAM on the input* $(\emptyset, t, \emptyset)$ *will terminate. Moreover, if the final state reached is* (ξ, t', S)*, then* $t \rhd^* \mathcal{T}(\xi, t', S)$ *and* $\mathcal{T}(\xi, t', S)$ *is in weak head normal form.*

Proof: The corollary is a trivial consequence of the GKAM correctness theorem and the GKAM liveliness theorem, together with the hypothesis of strong normalization of the term. □

9.3.2 Some Parameter Passing Strategies for the GKAM

In this section we will present several examples of correct parameter passing strategies for the GKAM. Many more reasonable reduction strategies can be defined as soon as constants are added to the language. In particular, strategies that evaluate the arguments (in a call-by-value fashion) until a constant is found in head position, and then stop and return the continuation (as in call-by-name) seems promising in our preliminary benchmarks on the HELM library.

In the description of the strategies, `reduce` is bound to the (generic) reduction function of the GKAM (recursively instantiated using the strategy) and $\mathcal{T}$ is bound to the (generic) read-back function (recursively instantiated using the strategy). Hence the strategies where `reduce` occurs perform non tail-recursive calls (and are thus not iterative and may be improved).

Call-by-name

SItem = Closure

EITem = Closure

to_stack = id

to_env = id

from_env(ξ, t) = $(\xi, t, \emptyset)$

$\mathcal{T}_S = \mathcal{T}$

$\mathcal{T}_E = \mathcal{T}$

$<_E$ = graph inclusion

The GKAM Correctness Conditions trivially hold. The GKAM Liveliness Condition is a consequence of an invariant that is granted by the way in which the environments are built by the GKAM reduction engine. The invariant states that for each environment $\alpha_1 \ldots\ldots \alpha_n$, the environment $\pi_1(\alpha_i)$ may have pointers only to $\alpha_1, \ldots, \alpha_{n-1}$. Hence the α_i constitute a finite d.a.g., with a natural well-founded relation.

Call-by-value, right-to-left evaluation order

SItem = GKAM

EITem = GKAM

to_stack = reduce

to_env = id

from_env = id

$\mathcal{T}_S = \mathcal{T}$

$\mathcal{T}_E = \mathcal{T}$

$<_E$ = graph inclusion

This implementation of call-by-value differs from the next variant in the evaluation order of the parameters. In particular, this strategy reduces the arguments one at a time (in right to left order) and performs β-reduction of the head only when all the arguments have been reduced. The next variant implements a more lazy strategy that evaluates the arguments from left to right and only when the argument is being consumed from a λ-abstraction. This amounts to a change in the definition of values if constants are introduced.

The GKAM Liveliness Condition is granted by the same invariant as the previous reduction strategy. The first GKAM Correctness Condition is hard to prove directly (because the machine calls the `reduce` function recursively). Nevertheless, it trivially holds *a posteriori* once we observe that, instantiating the GKAM reduction rules with our strategy, we obtain a reduction machine that is — unsurprisingly [5, 4] — extremely close to (and slightly less efficient than) the CEK reduction machine[45]: Indeed the CEK and the GKAM instantiated with the Call-by-value,

Table 9.3: CEK

Before			After		
Environment	Code	Stack	Environment	Code	Stack
$\xi.\beta$	n+1	S	ξ	n	S
$\xi.(\xi'; N)$	0	S	ξ'	N	S
ξ	$\lambda.M$	$(\mathrm{arg}(\alpha).S$	$\xi.\alpha$	M	S
ξ	MN	S	ξ	N	$\mathrm{fun}(\xi, M, S)$
ξ	V	$\mathrm{fun}(\xi', M', S')$	ξ'	M	$\mathrm{arg}(\xi, M).S'$

right-to-left evaluation order differ only in the rule that deals with application. When an application MN is met, the CEK stores somewhere (only accidentally on the stack) the continuation $(\xi, M, \mathrm{res}.S)$ (where res will be the result of the argument evaluation); then it evaluates the argument in the current environment using an "empty[4]" stack; finally, when a value is reached — that means that a λ-abstraction is found and the stack is currently empty — the continuation is resumed and the result of the evaluation of the argument is added on top of the stack.

The GKAM instantiated with the Call-by-value, right-to-left evaluation order behaves exactly in the same way. The continuation, instead of being stored in the stack, is (implicitly) remembered by the meta-language interpreter. The boring formal correspondence between the CEK and the status of the meta-level interpreter during the execution is left as an exercise to the reader.

The conclusion is that our strategy correctly implements call-by-value. Thus, a posteriori, the first GKAM Correctness Condition is satisfied, as well as the totality of the three functions.

Call-by-value, left-to-right evaluation order

SItem = Closure
EITem = GKAM
to_stack = id
to_env = reduce
from_env = id
$\mathcal{T}_S = \mathcal{T}$
$\mathcal{T}_E = \mathcal{T}$
$<_E$ = graph inclusion
The correctness of this strategy can be easily derived from the correctness of the previous one.

[4]The stack is not exactly empty, since it holds the continuation. Nevertheless, a quick inspection of the other rules shows that the continuation is never looked up and play the role of an empty-stack.

Naive Call-by-value, right-to-left evaluation order

$\mathcal{S}$Item = Term
EITem = Term
to_stack = $\mathcal{T} \circ$ reduce
to_env = id
from_env(t) = $(\emptyset, t, \emptyset)$
$\mathcal{T}_S$ = id
$\mathcal{T}_E$ = id
$\emptyset <_E \xi$, for each non-empty environment ξ

This implementation of the call-by-value strategy reads back the result term after the reduction of the actual argument, and stores the obtained term on the stack. So, in a sense, it corresponds more accurately to the traditional call-by-value machines, where the arguments are computed down to a term value. It is clearly less efficient than the first implementation, especially if constants are introduced. For instance, when the stack is not empty and has n elements, n applications are built by the read-back function; if the term happens to reach the head position, the n applications are eliminated one at a time and the stack of n elements is re-created. Moreover, explicit substitution of the environment is also performed by the read-back function, whereas the substitution could even be completely avoided when the argument does not occur in the term of the final configuration of the machine.

Call-by-need, first variant

$\mathcal{S}$Item = THUNK of type GKAM
EITem = THUNK of type GKAM
to_stack(ξ, t) = lazy[reduce(ξ, t)]
to_env = id
from_env = force
$\mathcal{T}_S$ = force $\circ\, \mathcal{T}$
$\mathcal{T}_E$ = force $\circ\, \mathcal{T}$
$<_E$ = graph inclusion

This first version of the call-by-need strategy is straightforwardly obtained from the call-by-value strategy by substituting the stack and the environment with lists of thunks. I.e. we are exploiting the laziness provided by the meta-level language. Notice, however, that in this way we are forced to evaluate the thunks when reading back the machine states. Thus, whenever constants are introduced, this strategy will perform more work than the traditional call-by-need strategy when reading back the values at the end of the computation.

The fact that the correctness and liveliness conditions are satisfied is a direct consequence of the fact that they hold for the call-by-value machines this one is based on.

Call-by-need, second variant

SItem = Closure
EITem = Ref to (Closure + GKAM)
to_stack = id
to_env(ξ, t) = ref (first (ξ, t))
from_env(α) =
    match !α with
        first (ξ, t) $\Rightarrow$
            let res = reduce(ξ, t) in
                α := second res ; res
        second res $\Rightarrow$ res
$\mathcal{T}_S = \mathcal{T}$
$\mathcal{T}_E = !\_ \circ \mathcal{T}$
$<_E$ = graph inclusion
This implementation of the call-by-need strategy does not exploit the laziness of the meta-language, but is based on imperative mutable cells (references). Concretely, it differs from the previous strategy only in the read back functions, that correctly read-back the unevaluated expression in case the term was never referenced in head position during the reduction.

9.3.3 Extending the GKAM to cover CIC plus Explicit Named Substitutions

In this section we extend the GKAM in order to cover the whole Calculus of (Co)Inductive Constructions extended with Explicit Named Substitutions. We call the new extended machine GKAM_{CIC}. In particular we add unreducible terms (i.e. opaque constants, (co)inductive types, (co)inductive constructors and sorts), transparent constants (i.e. definition occurrences), dependent products, case analysis and fixpoint and cofixpoint definitions. We also introduce a new environment (SubstitutionEnvironment) to accumulate the explicit named substitutions in scope, in order to avoid applying them instantaneously. Finally, we extend the rules to cover the case of open terms too (i.e. terms with occurrences of free variables in it).

Definition 9.17 (Closures, environments and stacks of the GKAM_{CIC})
Let EItem, SItem and SEItem be datatypes.
Environment $\stackrel{\text{def}}{=}$ *EItem List*
Stack $\stackrel{\text{def}}{=}$ *SItem List*

SubstitutionEnvironment $\stackrel{\mathrm{def}}{=}$ *SEItem List*
Closure $\stackrel{\mathrm{def}}{=}$ *Environment* $*$ *SubstitutionEnvironment* $*$ *Term*
GKAM $\stackrel{\mathrm{def}}{=}$ *Environment* $*$ *SubstitutionEnvironment* $*$ *Term* $*$ *Stack*
to_stack : Closure $\hookrightarrow$ *SItem*
to_env : SItem $\hookrightarrow$ *EItem*
to_ens : Closure $\hookrightarrow$ *SEItem*
from_env : SItem $\hookrightarrow$ *GKAM*
from_ens : SEItem $\hookrightarrow$ *GKAM*
$\mathcal{T}_s$ *: SItem* $\to$ *Term*
$\mathcal{T}_e$ *: EItem* $\to$ *Term*
$\mathcal{T}_{se}$ *: SEItem* $\to$ *Term*
We also assume the existence of a strict and well-founded order
$<_{E\times SE} \subseteq$ *(Environment* $\times$ *SubstitutionEnvironment)*^{2}
such that

1. $\forall\xi$: *Environment* $.\forall\sigma$:*SubstitutionEnvironment* $.\forall\alpha$: *EItem* .
   $(\xi, \sigma) <_{E\times SE} (\xi.\alpha, \sigma)$

2. $\forall\xi$: *Environment* $.\forall\sigma$:*SubstitutionEnvironment* $.\forall\gamma$: *SEItem* .
   $(\xi, \sigma) <_{E\times SE} (\xi, \sigma.\gamma)$

The transition rules of the GKAM_{CIC} are shown in Table 9.4 and Table 9.5. The initial state of the machine to reduce a term t is $(\emptyset, \emptyset, t, \emptyset)$. The final states of the machine are the special states $(-, -, (\xi, \sigma, t, S), -)$.

The definitions of the read-back function and the correctness and liveliness conditions generalize the corresponding definitions for the GKAM to consider also the explicit named substitutions environment:

Definition 9.18 *Read-back function* $\mathcal{T}(\_,\_,\_,\_)$

$$\mathcal{T}(-, -, (\xi, \sigma, t, S), -) \stackrel{\mathrm{def}}{=} \mathcal{T}(\xi, \sigma, t, S)$$
$$\mathcal{T}([\alpha_1, \ldots, \alpha_n], [\gamma_1/\nu_1, \ldots, \gamma_l/\nu_l], t, [\beta_1, \ldots \beta_m]) \stackrel{\mathrm{def}}{=}$$
$$(t\{\mathcal{T}_{se}(\gamma_1)/\nu_1; \ldots; \mathcal{T}_{se}(\gamma_l)/\nu_l \mid \mathcal{T}_e(\alpha_1)/1\ ;\ \ldots\ ;\ \mathcal{T}_e(\alpha_n)/n\}_m\ \mathcal{T}_s(\beta_1)\ \ldots\ \mathcal{T}_s(\beta_m)))$$

where $\{\sigma|\sigma'\}_m$ *is the simultaneous substitution mapping every de Brujin index in the domain of* σ' *to its image in* σ'*, every variable in the domain of* σ *to its image in* σ *and extending every explicit named substitutions* σ'' *mapping each variable* ν *in* $(\mathrm{Dom}(\sigma) \setminus \mathrm{Dom}(\sigma'')) \cap D$ *to* $\sigma(\nu)$*, where* D *is the set of variables the reference* r *is applied to and* r *is the reference* σ'' *is applied to.*

Notice that for each reference r to a constant or variable and for each explicit named substitution σ and σ', $c[\sigma]\{\sigma'|\emptyset\}_m = c[\sigma \vee_c \sigma']$.

Table 9.4: GKAM_{CIC} reduction rules (1/2)

Before				After			
Environment	Subst. Env.	Code	Stack	Environment	Subst. Env.	Code	Stack
$[\alpha_n\ ;\ \dots\ ;\ \alpha_1]$	σ	i (where $i \le n$)	S	π_1 (from_env α_i)	π_2 (from_env α_i)	π_3 (from_env α_i)	π_4 (from_env α_i).S
ξ	σ	i (where $i > \lvert\xi\rvert$)	S	-	-	(ξ, σ, i, S)	-
ξ	σ	$c[\sigma']$	S	$\emptyset$	σ''	b	S
ξ	σ	$c[\sigma']$	S	-	-	$(\xi, \sigma, c[\sigma'], S)$	-
ξ	σ	$i[\sigma']$	S	-	-	$(\xi, \sigma, i[\sigma'], S)$	-
ξ	σ	$k[\sigma']$	S	-	-	$(\xi, \sigma, k[\sigma'], S)$	-
ξ	σ	s	$\emptyset$	-	-	$(\emptyset, \emptyset, s, \emptyset)$	-
ξ	$\gamma_1/\nu_1\ ;\ \dots\ ;\ \gamma_n/\nu_n$	$\nu_i[\sigma']$	S	π_1 (from_ens γ_i)	π_2 (from_ens γ_i)	π_3 (from_ens γ_i)	π_4 (from_ens γ_i).S
ξ	σ	$\nu[\sigma']$	S	-	-	$(\xi, \sigma, \nu[\sigma'], S)$	-
ξ	σ	$\Pi : T.t$	$\emptyset$	-	-	$(\xi, \sigma, \Pi : T.t, \emptyset)$	-
ξ	σ	$\lambda : T.M$	$\alpha.S$	ξ, (to_env α)	σ	M	S
ξ	σ	$\lambda : T.M$	$\emptyset$	-	-	$(\xi, \sigma, \lambda : T.M, \emptyset)$	-
ξ	σ	MN	S	ξ	σ	M	(to_stack (ξ,σ,N)).S
ξ	σ	Let $:=$ M in N	S	ξ.to_env (to_stack(ξ, σ, M))	σ	N	S

Notes:

In the second rule b is the definiens for the definiendum c and $\sigma'' = [(\text{to\_ens}\ (\xi, \sigma, \_)\ \sigma')] \sqcup_c \{\sigma\}_\nu$ where $[\sigma] \sqcup_c \{\gamma_1/\nu_1\ ;\ \dots\ ;\ \gamma_m/\nu_m\}_\nu = [\sigma\ ;\ \gamma_{i_1}/\nu_{i_1}\ ;\ \dots\ ;\ \gamma_{i_n}/\nu_{i_n}]$ where $i_1, \dots, \nu_n$ is the maximal subset of $1, \dots, m$ such that $\nu\_i_j \notin Dom(\sigma)$ and c is abstracted over ν_{i_j} for each j in $1, \dots, n$.

The third rule is used when no definiens is available for the definiendum c (i.e. when c is an axiom or an opaque constant).

The eighth rule is fired only if σ' is the identity substitution. Otherwise the eighth rule is activated.

Table 9.5: GKAM$_{CIC}$ reduction rules (2/2)

Before				After			
Env.	Sub.Env.	Code	Stack	Env.	Sub.Env.	Code	Stack
ξ	σ	$< T >$ CASES$_i$ t OF $k_1\ \overline{x} \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x} \Rightarrow t_n$	S	ξ	$\sigma.((\mathtt{to\_env})\ S_r)$	t_i	S
ξ	σ	$t_0 =$ $< T >$ CASES$_i$ t OF $k_1\ \overline{x} \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x} \Rightarrow t_n$	S	-	-	(ξ, σ, t_0, S)	-
ξ	σ	$\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots\ ;\ x_m/n_m : T_m := b_m\}$	S	$\xi, F_m, \ldots, F_1$	σ	b_l	S'
ξ	σ	$t_0 = \mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots\ ;\ x_m/n_m : T_m := b_m\}$	S	-	-	(ξ, σ, t_0, S')	-
ξ	σ	$t_0 = \mathrm{COFIX}_l\ \{\overline{x : T := b}\}$	S	-	-	(ξ, σ, t_0, S)	-

Notes:

When a CASES construct is met, the argument t is reduced by istantiating a new reduction machine with environment ξ, substitution environment σ and an empty stack. Let (ξ', σ', t', S') be the computed result. If $t' = \mathrm{COFIX}_l\{\overline{x : T := b}\}$, then t' is ι-expanded into b_l, and the computation of the reduction machine is started again on the new initial state $(\xi', \sigma'.F_m.\ldots.F_1, b_l, S')$ where $F_i = \text{to\_env}(\text{to\_stack}(\mathrm{COFIX}_i\{\overline{x : T := b}\}))$. The previous step is skipped if no cofixpoint definition is found in head position in t'. Let $(\xi_f, \sigma_f, t_f, S_f)$ be the final state of the last initialized machine. If t_f is $k_i[\sigma_f']$, the first rule is fired, where $S_f = S_l.S_r$ and the length of the list S_l is the number of parameters the inductive definition i depends on. Otherwise the second rule is fired. Note that, to respect the classical definition of values for case analysis, in the second rule we should replace t with its weak-head normal form t_n (i.e. $\mathcal{T}(\xi_f, \sigma_f, t\_f, S_f)$). Unfortunately, the substitution is not possible since the term computed weak-head normal form is defined in the initial empty environment and substitution environment and not in ξ and σ. We can solve the problem by reading-back the term w.r.t. ξ and σ before performing the substituion. Thus we propose also the alternative rule that returns the configuration $(\emptyset, \emptyset, < \mathcal{T}(\xi, \sigma, T) > \mathrm{CASES}_i\ t_n\ \mathrm{OF}\ k_1\overline{x}^1 \Rightarrow \mathcal{T}(\xi, \sigma, t_1) \mid \ldots \mid k_n\overline{x}^n \Rightarrow \mathcal{T}(\xi, \sigma, t_n)\ \mathrm{END}, S)$

When a $\mathtt{FIX}_i$ construct is met, the n_i-th element of the stack (let's call it S_{n_i}) is retrieved (if present) and a new reduction machine is istantiated on the initial state $(\emptyset, \emptyset, \mathcal{T}_s(S_{n_i}), \emptyset)$; let ξ', σ', t', S' be the final state of the machine; if t' is a mutual constructor, then the third rule is fired. Otherwise — if t' is not a constructor or if the stack did not have at least n_i elements — the fourth rule is fire. In both cases (third and fourth rule applied), S' is identical to S except for the n_i-th item (if present) that is replaced by $\text{to\_stack}(\emptyset, \emptyset, \mathcal{T}(\xi', \sigma', t', S'))$.

The $F_1, \ldots, F_m$ terms in the third rule are the result of applying the $\mathtt{to\_env}(\mathtt{to\_stack}(\xi, \sigma, \_))$ function respectively to the 1-st,…,m-th function defined in the fixpoint definition.

Definition 9.19 *Read-back function* $\mathcal{T}(\_,\_,\_)$

$$\mathcal{T}(\xi,\sigma,t) \stackrel{\text{def}}{=} \mathcal{T}(\xi,\sigma,t,\emptyset)$$

The following fundamental fact about the behaviour of the read back function on the initial states trivially holds:

Fact 9.20 *For each term* t, $\mathcal{T}(\emptyset,\emptyset,t,\emptyset) = t$.

Definition 9.21 *GKAM*$_{\mathrm{CIC}}$ *Correctness Conditions:*

1. $\forall(\xi,\sigma,t) : Closure\,.\mathcal{T}(\xi,\sigma,t) \rhd^* \mathcal{T}_s(to\_stack\ (\xi,\sigma,t))$

2. $\forall\alpha : SItem\,.\mathcal{T}_s(\alpha) \rhd^* \mathcal{T}_e(to\_env\ \alpha)$

3. $\forall\alpha : EItem\,.\mathcal{T}_e(\alpha) \rhd^* \mathcal{T}(from\_env\ \alpha)$

4. $\forall(\xi,\sigma,t) : Closure\,.\mathcal{T}(\xi,\sigma,t) \rhd^* \mathcal{T}_{se}(to\_ens\ (\xi,\sigma,t))$

5. $\forall\gamma : SEItem\,.\mathcal{T}_{se}(\gamma) \rhd^* \mathcal{T}(from\_ens\ \gamma)$

Definition 9.22 *GKAM*$_{\mathrm{CIC}}$ *Liveliness Conditions*

1. $\pi_1 \circ hd \circ from\_env \circ <\pi_1,\pi_2>$ *is strictly decreasing (according to the ordering* $<_{E\times SE}$*)*

2. $\pi_2 \circ hd) \circ from\_ens \circ <\pi_1,\pi_2>$ *is strictly decreasing (according to the ordering* $<_{E\times SE}$*)*

In order to prove the correctness of the GKAM$_{\mathrm{CIC}}$ we need two technical lemmas that relate $\mathcal{T}_{se}(\_)$, $\mathcal{T}_e(\_)$, $\{\_|\_\}_m$, $\{\_\}$ and $\sqcup\_$:

Lemma 9.23 *For each constant* c *and explicit named substitution* σ',
$\forall\sigma : SubstitutionEnvironment\,.\forall\xi : Environment\,.$
$c[\sigma']\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \rhd^* c[\mathcal{T}_{se}([(to\_ens(\xi,\sigma,\_)\ \sigma')]) \sqcup_c \{\sigma\}_\nu]$

Proof:

$$\begin{aligned}
&c[\sigma']\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\quad = c[\sigma'\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\ ;\ \mathcal{T}_{se}(\sigma'')]\\
&\quad = c[(\mathcal{T}(\xi,\sigma,_)\ \sigma')\ ;\ \mathcal{T}_{se}(\sigma'')]\\
&\quad \rhd^* c[\mathcal{T}_{se}(\text{to_ens}(\xi,\sigma,_)\ \sigma')\ ;\ \mathcal{T}_{se}(\sigma'')]\\
&\quad = c[\mathcal{T}_{se}([\text{to_ens}(\xi,\sigma,_)\ \sigma'\ ;\ \sigma''])]\\
&\quad = c[\mathcal{T}_{se}([(\text{to_ens}(\xi,\sigma,_)\ \sigma')] \sqcup_c \{\sigma\}_\nu)]
\end{aligned}$$

where σ'' is the restriction of the substitution σ to the domain of the variables that c is abstracted over and that do not belong to the domain of σ'. The third rewriting step is simply an instance of the fourth GKAM$_{\mathrm{CIC}}$ correctness condition. □

Lemma 9.24 *For each term* t *and substitution* σ,

$\forall \sigma' : \mathit{SubstitutionEnvironment}\ .\ \forall \xi : \mathit{Environment}\ .$
$t\{\sigma\}\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi)\}_m \rhd^* t\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi.(\mathit{to\_env}(\mathit{to\_stack}(\xi, \sigma', \_))\ \sigma))\}_m$

Proof:

$$\begin{aligned}
& t\{\sigma\}\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi)\}_m \\
& \quad = t\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi).\sigma\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi)\}_m\}_m \\
& \quad = t\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi).(\mathcal{T}(\xi, \sigma', _)\ \sigma)\}_m \\
& \quad = t\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi).(\mathcal{T}_e(\text{to_env}(\text{to_stack}(\xi, \sigma', _)))\ \sigma)\}_m \\
& \quad \rhd^* t\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi.(\text{to_env}(\text{to_stack}(\xi, \sigma', _))\ \sigma))\}_m
\end{aligned}$$

The last rewriting step is justified by the composition of the first and second GKAM_{CIC} Correctness Conditions. □

We can now state and prove the correctness of the GKAM_{CIC}:

Theorem 9.25 *GKAM_{CIC} correctness* $\forall(\xi, \sigma, t, S), (\xi', \sigma', t', S')\ :\ \mathit{GKAM}\ .$ *if* (ξ, σ, t, S) *evolves into* (ξ', σ', t', S') *then* $\mathcal{T}(\xi, \sigma, t, S) \rhd^* \mathcal{T}(\xi', \sigma', t', S')$.

Proof: We proceed by generalized induction on the number of intermediate steps between (ξ, σ, t, S) and (ξ', σ', t', S').

If the last rule maps (ξ, σ, t, S) to $(-, -, (\xi, \sigma, t, S), -)$, then the thesis trivially holds. Otherwise, let us proceed by cases on the last GKAM rule that is fired:

- First rule:

$$\begin{aligned}
& \mathcal{T}([\alpha_n\ ;\ \ldots\ ;\ \alpha_1], \sigma, i, S) \\
& \quad = (\mathcal{T}_e(\alpha_i)\ \mathcal{T}_s(S)) \\
& \quad \rhd^*(\mathcal{T}(\text{from_env}\ \alpha_i)\ \mathcal{T}_s(S)) \\
& \quad = (\pi_3(\text{from_env}\ \alpha_i)\{\mathcal{T}_{se}(\pi_2(\text{from_env}\ \alpha_i))|\mathcal{T}_e(\pi_1(\text{from_env}\ \alpha_i))\}_m \\
& \qquad\quad \mathcal{T}_s(\pi_4(\text{from_env}\ \alpha_i))\mathcal{T}_s(S)) \\
& \quad = \mathcal{T}(\pi_1(\text{from_env}\ \alpha_i), \pi_2(\text{from_env}\ \alpha_i), \pi_3(\text{from_env}\ \alpha_i), \pi_4(\text{from_env}\ \alpha_i).S)
\end{aligned}$$

  The second rewriting step is justified by the third GKAM_{CIC} Correctness Condition.

- Third rule:

$$\begin{aligned}
& \mathcal{T}(\xi, \sigma, c[\sigma'], S) \\
& \quad = (c[\sigma']\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\ \mathcal{T}_s(S)) \\
& \quad \rhd^*(c[\mathcal{T}_{se}([(\text{to_ens}(\xi, \sigma, _)\ \sigma')]) \sqcup_c \{\sigma\}_v]\ \mathcal{T}_s(S)) \\
& \quad \rhd_\delta(b\{\mathcal{T}_{se}([(\text{to_ens}(\xi, \sigma, _)\ \sigma')]) \sqcup_c \{\sigma\}_v\}_v\ \mathcal{T}_s(S)) \\
& \quad = (b\{\mathcal{T}_{se}([(\text{to_ens}(\xi, \sigma, _)\ \sigma')]) \sqcup_c \{\sigma\}_v|\emptyset\}_m\ \mathcal{T}_s(S)) \\
& \quad = \mathcal{T}(\emptyset, [(\text{to_ens}(\xi, \sigma, _)\ \sigma')] \sqcup_c \{\sigma\}_v, b, S)
\end{aligned}$$

  The second rewriting step is a consequence of lemma 9.23, whose correctness relies on the fourth GKAM_{CIC} Correctness Condition.

- Seventh rule: trivial since a sort is a closed term.
- Eighth rule:

  Let σ be $[\gamma_1/\nu_1 \ ; \ \ldots \ ; \ \gamma_n/\nu_n]$ and σ' be the identity substitution.

$$
\begin{aligned}
&\mathcal{T}(\xi, \sigma, \nu_i[\sigma'], S)\\
&\quad = (\nu_i[\sigma']\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S))\\
&\quad = (\mathcal{T}_{se}(\gamma_i) \ \mathcal{T}_s(S))\\
&\quad \rhd^*(\mathcal{T}(\text{from\_ens } \gamma_i) \ \mathcal{T}_s(S))\\
&\quad = (\pi_3(\text{from\_ens } \gamma_i)\{\mathcal{T}_{se}(\pi_2(\text{from\_ens } \gamma_i))|\mathcal{T}_e(\pi_1(\text{from\_ens } \gamma_i))\}_m\\
&\qquad\quad \mathcal{T}_s(\pi_4(\text{from\_ens } \gamma_i)) \ \mathcal{T}_s(S))\\
&\quad = \mathcal{T}(\pi_1(\text{from\_env } \gamma_i), \pi_2(\text{from\_env } \gamma_i), \pi_3(\text{from\_env } \gamma_i), \pi_4(\text{from\_env } \gamma_i).S)
\end{aligned}
$$

  The third rewriting step is justified by the fifth GKAM_{CIC} Correctness Condition.

- Eleventh rule:

$$
\begin{aligned}
&\mathcal{T}(\xi, \sigma, \lambda.M, \alpha.S)\\
&\quad = (\lambda.M\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(\alpha) \ \mathcal{T}_s(S))\\
&\quad \rhd_\beta(M\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi); \mathcal{T}_s(\alpha)\}_m \ \mathcal{T}_s(S))\\
&\quad \rhd^*(M\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi); \mathcal{T}_e(\text{to\_env } \alpha)\}_m \ \mathcal{T}_s(S))\\
&\quad = \mathcal{T}(\xi.(\text{to\_env } \alpha), \sigma, M, S)
\end{aligned}
$$

  The third rewriting step is justified by the second GKAM_{CIC} Correctness Condition.

- Thirteenth rule:

$$
\begin{aligned}
&\mathcal{T}(\xi, \sigma, MN, S)\\
&\quad = (MN\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S))\\
&\quad = (M\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ N\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S))\\
&\quad = (M\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}(\xi, \sigma, N) \ \mathcal{T}_s(S))\\
&\quad \rhd^*(M\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s((\text{to\_stack}(\xi, \sigma, N))) \ \mathcal{T}_s(S))\\
&\quad = \mathcal{T}(\xi, \sigma, M, (\text{to\_stack}(\xi, \sigma, N)).S)
\end{aligned}
$$

  The fourth rewriting step is justified by the first GKAM_{CIC} Correctness Condition.

- Fourteenth rule:

$$
\begin{aligned}
&\mathcal{T}(\xi, \sigma, \text{Let} \ := \ M \text{ in } N, S)\\
&\quad = (\text{Let} \ := \ M \text{ in } N\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S))\\
&\quad = (\text{Let} \ := \ M\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \text{ in } N\{\uparrow \mathcal{T}_{se}(\sigma)| \uparrow \mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S))\\
&\quad = (\text{Let} \ := \ \mathcal{T}(\xi, \sigma, M) \text{ in } N\{\uparrow \mathcal{T}_{se}(\sigma)| \uparrow \mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S))\\
&\quad \rhd^*(\text{Let} \ := \ \mathcal{T}_e(\text{to\_env}(\text{to\_stack}(\xi, \sigma, M))) \text{ in } N\{\uparrow \mathcal{T}_{se}(\sigma)| \uparrow \mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S))\\
&\quad \rhd_\zeta(N\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi); \mathcal{T}_e(\text{to\_env}(\text{to\_stack}(\xi, \sigma, M)))\}_m \ \mathcal{T}_s(S))\\
&\quad = \mathcal{T}(\xi.\text{to\_env}(\text{to\_stack}(\xi, \sigma, M)), \sigma, N, S)
\end{aligned}
$$

where the lifting operator $\uparrow$ simultaneously substitutes every free de Brujin index n with $n+1$. The fourth rewriting step is a combination of the first and second GKAM_{CIC} Correctness Conditions.

- Fifteenth rule:

$$
\begin{array}{l}
\mathcal{T}(\xi, \sigma, < T > \text{CASES}_i \ t \text{ OF } k_1 \ \overline{x}^1 \Rightarrow t_1 \mid \ldots \mid k_n \ \overline{x}^n \Rightarrow t_n \text{ END}, S) \\
\quad = (< T > \text{CASES}_i \ t \text{ OF } k_1 \ \overline{x}^1 \Rightarrow t_1 \mid \ldots \mid k_n \ \overline{x}^n \Rightarrow t_n \text{ END}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \ \mathcal{T}_s(S)) \\
\quad = (< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \text{CASES}_i \ t\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m \text{ OF} \\
\qquad\qquad k_1 \ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\qquad\qquad \mid \ldots \\
\qquad\qquad \mid k_n \ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\qquad \text{END } \mathcal{T}_s(S)) \\
\quad = (< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \text{CASES}_i \ \mathcal{T}(\xi, \sigma, t) \text{ OF} \\
\qquad\qquad k_1 \ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\qquad\qquad \mid \ldots \\
\qquad\qquad \mid k_n \ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\qquad \text{END } \mathcal{T}_s(S)) \\
\quad \rhd^*(< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \text{CASES}_i \ \mathcal{T}(\xi', \sigma', \text{COFIX}_l\{\overline{x : T := b}\}, S') \text{ OF} \\
\qquad\qquad k_1 \ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\qquad\qquad \mid \ldots \\
\qquad\qquad \mid k_n \ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\qquad \text{END } \mathcal{T}_s(S)) \\
\quad \rhd_{cofix}(< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \text{CASES}_i \ \mathcal{T}(\xi', \sigma', b_l\{F_1/1, \ldots, F_m/m\}, S') \text{ OF} \\
\qquad\qquad k_1 \ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\qquad\qquad \mid \ldots \\
\qquad\qquad \mid k_n \ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\qquad \text{END } \mathcal{T}_s(S)) \\
\quad = (< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \text{CASES}_i \ (b_l\{F_1/1, \ldots, F_m/m\}\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi')\}_m \ \mathcal{T}_s(S')) \text{ OF} \\
\qquad\qquad k_1 \ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\qquad\qquad \mid \ldots \\
\qquad\qquad \mid k_n \ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\qquad \text{END } \mathcal{T}_s(S))
\end{array}
$$

$$
\begin{array}{l}
\rhd^*(< \mathrm{T}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \\
\quad\quad \mathrm{CASES}_i \\
\quad\quad\quad (b_l\{\mathcal{T}_{se}(\sigma')|\mathcal{T}_e(\xi' ; (\text{to\_env}(\text{to\_stack}(\xi', \sigma', \_))[\mathrm{F}_m ; \ldots ; \mathrm{F}_1]))\}_m \ \mathcal{T}_s(S')) \\
\quad\quad \mathrm{OF} \\
\quad\quad\quad\quad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\quad\quad\quad | \ \ldots \\
\quad\quad\quad | \ k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\quad \mathrm{END}\ \mathcal{T}_s(S)) \\
= (< \mathrm{T}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \\
\quad\quad \mathrm{CASES}_i\ \mathcal{T}(\xi'.(\text{to\_env}(\text{to\_stack}(\xi', \sigma', \_))\ [\mathrm{F}_m ; \ldots ; \mathrm{F}_1]), \sigma', b_l, S')\ \mathrm{OF} \\
\quad\quad\quad\quad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\quad\quad\quad | \ \ldots \\
\quad\quad\quad | \ k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\quad \mathrm{END}\ \mathcal{T}_s(S)) \\
\rhd^*(< \mathrm{T}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \mathrm{CASES}_i\ \mathcal{T}(\xi_f, \sigma_f, k_i[\sigma'_f], S_l.S_r)\ \mathrm{OF} \\
\quad\quad\quad\quad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\quad\quad\quad | \ \ldots \\
\quad\quad\quad | \ k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\quad \mathrm{END}\ \mathcal{T}_s(S)) \quad\quad = (< \mathrm{T}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \mathrm{CASES}_i\ (k_i[\sigma'_f]\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\ \mathcal{T}_s(S_l)\ \mathcal{T}_s(S_r))\ \mathrm{OF} \\
\quad\quad\quad\quad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^1|} \mathcal{T}_e(\xi)\}_m \\
\quad\quad\quad | \ \ldots \\
\quad\quad\quad | \ k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|} \mathcal{T}_{se}(\sigma)| \uparrow^{|\overline{x}^n|} \mathcal{T}_e(\xi)\}_m \\
\quad \mathrm{END}\ \mathcal{T}_s(S)) \\
\rhd_\iota(t_i\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_s(S_r) ; \mathcal{T}_e(\xi)\}_m\ \mathcal{T}_s(S)) \\
\rhd^*(t_i\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\text{to\_env}\ S_r) ; \mathcal{T}_e(\xi)\}_m\ \mathcal{T}_s(S)) \\
= (t_i\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi.\text{to\_env}\ S_r)\}_m\ \mathcal{T}_s(S)) \\
= \mathcal{T}(\xi.\text{to\_env}\ S_r, \sigma, t_i, S)
\end{array}
$$

where $\mathrm{F}_i = \mathrm{COFIX}_i\{\overline{x : \mathrm{T} := b}\}$.

The fourth and the nineth rewriting steps are justified by the induction hypothesis. The rewriting steps 5 – 9 are skipped — both in the proof and in the reduction rules of the machine — if the fourth step does not produce a co-fixpoint operator. Otherwise the seventh rewriting step is justified by lemma 9.24. The twelveth rewriting step is an instance of the second $\mathrm{GKAM}_{\mathrm{CIC}}$ Correctness Condition.

- Sixteenth rule (alternative formulation): The statement trivially holds with the rule we choose to adopt. The proof is slightly more envolved if we adopted the alternative formulation (discussed in the "notes" below the rule description). Notice that the alternative formulation is the only formulation that respects the classical definition of values for the case-analysis operator. The alternative formulation performs more computation than the chosen one. However, the two formulations are not comparable when used to test convertibility of two generic terms: it is possible to find terms that make any one of the two computations faster than the other.

$$
\begin{aligned}
&\mathcal{T}(\xi, \sigma, < T > \mathrm{CASES}_i\ t\ \mathrm{OF}\ k_1\ \overline{x}^1 \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x}^n \Rightarrow t_n\ \mathrm{END}, S)\\
&\quad = (< T > \mathrm{CASES}_i\ t\ \mathrm{OF}\ k_1\ \overline{x}^1 \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x}^n \Rightarrow t_n\ \mathrm{END}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\ \mathcal{T}_s(S))\\
&\quad = (< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \mathrm{CASES}_i\ t\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\ \mathrm{OF}\\
&\qquad\qquad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^1|}\mathcal{T}_e(\xi)\}_m\\
&\qquad\qquad \mid \ldots\\
&\qquad\qquad \mid k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^n|}\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathrm{END}\ \mathcal{T}_s(S))\\
&\quad = (< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \mathrm{CASES}_i\ \mathcal{T}(\xi, \sigma, t)\ \mathrm{OF}\\
&\qquad\qquad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^1|}\mathcal{T}_e(\xi)\}_m\\
&\qquad\qquad \mid \ldots\\
&\qquad\qquad \mid k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^n|}\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathrm{END}\ \mathcal{T}_s(S))\\
&\quad \rhd^*(< T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \mathrm{CASES}_i\ \mathcal{T}(\xi', \sigma', t', S')\ \mathrm{OF}\\
&\qquad\qquad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^1|}\mathcal{T}_e(\xi)\}_m\\
&\qquad\qquad \mid \ldots\\
&\qquad\qquad \mid k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^n|}\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathrm{END}\ \mathcal{T}_s(S))\\
&\quad = \mathcal{T}(\emptyset, \emptyset, < T\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m > \mathrm{CASES}_i\ \mathcal{T}(\xi', \sigma', t', S')\ \mathrm{OF}\\
&\qquad\qquad\qquad k_1\ \overline{x}^1 \Rightarrow t_1\{\uparrow^{|\overline{x}^1|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^1|}\mathcal{T}_e(\xi)\}_m\\
&\qquad\qquad\qquad \mid \ldots\\
&\qquad\qquad\qquad \mid k_n\ \overline{x}^n \Rightarrow t_n\{\uparrow^{|\overline{x}^n|}\mathcal{T}_{se}(\sigma)|\uparrow^{|\overline{x}^n|}\mathcal{T}_e(\xi)\}_m\\
&\qquad\qquad \mathrm{END}, S)
\end{aligned}
$$

- Seventeenth rule:

$$
\begin{aligned}
&\mathcal{T}(\xi, \sigma, \mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}, S_1.\alpha.S_2)\\
&\quad = (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1.\alpha.S_2))\\
&\quad = (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1)\ \mathcal{T}(\emptyset, \emptyset, \mathcal{T}_s(\alpha), \emptyset)\ \mathcal{T}_s(S_2))
\end{aligned}
$$

$$
\begin{aligned}
&\rhd^*(\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1)\ \mathcal{T}(\xi', \sigma', k_i[\sigma''], S')\ \mathcal{T}_s(S_2))\\
&= (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1)\ \mathcal{T}(\emptyset, \emptyset, \mathcal{T}(\xi', \sigma', k_i[\sigma''], S'))\ \mathcal{T}_s(S_2))\\
&\rhd^*(\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1)\ \mathcal{T}_s(\mathrm{to\_stack}(\emptyset, \emptyset, \mathcal{T}(\xi', \sigma', k_i[\sigma''], S')))\ \mathcal{T}_s(S_2))\\
&= (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1.(\mathtt{to\_stack}(\emptyset, \emptyset, \mathcal{T}(\xi', \sigma', k_i[\sigma''], S'))).S_2))\\
&= (\mathrm{FIX}_l\ \{\\
&\qquad x_1/n_1 : T_1\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m := b_1\{\uparrow^m \mathcal{T}_{se}(\sigma)|\uparrow^m \mathcal{T}_e(\xi)\}_m\ ;\\
&\qquad \ldots\ ;\\
&\qquad x_m/n_m : T_m\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m := b_m\{\uparrow^m \mathcal{T}_{se}(\sigma)|\uparrow^m \mathcal{T}_e(\xi)\}_m\\
&\quad \}\ \mathcal{T}_s(S'))\\
&\rhd_{\mathrm{FIX}}(b_l\{\uparrow^m \mathcal{T}_{se}(\sigma)|\uparrow^m \mathcal{T}_e(\xi)\}_m\{F_1/1\ ;\ \ldots\ ;\ F_m/m\}\ \mathcal{T}_s(S'))\\
&= (b_l\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\ ;\ F_m\ ;\ \ldots\ ;\ F_1\}_m\ \mathcal{T}_s(S'))\\
&= (b_l\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\ ;\ (\mathcal{T}(\emptyset, \emptyset, \_)\ [F_m\ ;\ \ldots\ ;\ F_1])\}_m\ \mathcal{T}_s(S'))\\
&\rhd^*(b_l\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\ ;\ (\mathcal{T}_e(\mathrm{to\_env}(\mathrm{to\_stack}(\emptyset, \emptyset, \_)))\ [F_m\ ;\ \ldots\ ;\ F_1])\}_m\ \mathcal{T}_s(S'))\\
&= (b_l\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi.(\mathrm{to\_env}(\mathrm{to\_stack}(\emptyset, \emptyset, \_))\ [F_m\ ;\ \ldots\ ;\ F_1]))\}_m\ \mathcal{T}_s(S'))\\
&= \mathcal{T}(\sigma, \xi.(\mathrm{to\_env}(\mathrm{to\_stack}(\emptyset, \emptyset, \_))\ [F_m\ ;\ \ldots\ ;\ F_1]), b_l, S')
\end{aligned}
$$

where $|S_1| = n_l - 1$ and $S' = S_1.(\mathtt{to\_stack}(\emptyset, \emptyset, \mathcal{T}(\xi', \sigma', k_i[\sigma''], S'))).S_2$
and $F_i = \mathrm{FIX}_i\{x_1/n_1 : T_1 := b_1\ ;\ \ldots\ ;\ x_m/n_m : T_m := b_m\}$.

The third rewriting step is justified by the induction hypothesis. The fifth rewriting step is justified by the first $\mathrm{GKAM_{CIC}}$ Correctness Condition. The eleventh rewriting step is a combination of the first and second $\mathrm{GKAM_{CIC}}$ Correctness Conditions.

- Eighteenth rule: If the length of the stack is not sufficient to retrieve the recursive argument, then the thesis trivially holds. Otherwise we proceed as we did for the previous rule:

$$
\begin{aligned}
&\mathcal{T}(\xi, \sigma, \mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}, S_1.\alpha.S_2)\\
&\quad = (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad\quad \mathcal{T}_s(S_1.\alpha.S_2))\\
&\quad = (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad\quad \mathcal{T}_s(S_1)\ \mathcal{T}(\emptyset, \emptyset, \mathcal{T}_s(\alpha), \emptyset)\ \mathcal{T}_s(S_2))
\end{aligned}
$$

$$
\begin{aligned}
&\rhd^*(\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1)\ \mathcal{T}(\xi',\sigma',t',S')\ \mathcal{T}_s(S_2))\\
&= (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1)\ \mathcal{T}(\emptyset,\emptyset,\mathcal{T}(\xi',\sigma',t',S'))\ \mathcal{T}_s(S_2))\\
&\rhd^*(\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1)\ \mathcal{T}_s(\mathrm{to\_stack}(\emptyset,\emptyset,\mathcal{T}(\xi',\sigma',t',S')))\ \mathcal{T}_s(S_2))\\
&= (\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\}\{\mathcal{T}_{se}(\sigma)|\mathcal{T}_e(\xi)\}_m\\
&\qquad \mathcal{T}_s(S_1.(\mathrm{to\_stack}(\emptyset,\emptyset,\mathcal{T}(\xi',\sigma',t',S'))).S_2))\\
&= \mathcal{T}(\xi,\sigma,\mathrm{FIX}_l\ \{x_1/n_1 : T_1 := b_1\ ;\ \ldots x_m/n_m : T_m := b_m\},\\
&\qquad S_1.(\mathrm{to\_stack}(\emptyset,\emptyset,\mathcal{T}(\xi',\sigma',t',S'))).S_2)
\end{aligned}
$$

where $|S_1| = n_l - 1$.

The third rewriting step is justified by the induction hypothesis. The fifth rewriting step is justified by the first $\mathrm{GKAM}_{\mathrm{CIC}}$ Correctness Condition.

□

The correctness theorem we just proved grants that any move of the machine correctly performs zero or more reduction steps. It does not grant any liveliness property: in particular there could be an infinite sequence of machine steps that do not reduce the term and that simply corresponds to the identity function. Liveliness is granted by the following theorem, that uses the $\mathrm{GKAM}_{\mathrm{CIC}}$ Liveliness Conditions:

Theorem 9.26 *Liveliness*
There is no infinite sequence of $\mathrm{GKAM}_{\mathrm{CIC}}$ states $(\xi_0,\sigma_0,t_0,S_0),(\xi_1,\sigma_1,t_1,S_1),\ldots$ such that for each index i (ξ_i,σ_i,t_i,S_i) moves into $(\xi_{i+1},\sigma_{i+1},t_{i+1},S_{i+1})$ and $\mathcal{T}(\xi_i,\sigma_i,t_i,S_i) = \mathcal{T}(\xi_{i+1},\sigma_{i+1},t_{i+1},S_{i+1})$

Proof: We proceed by absurdum. Let us assume that such an infinite sequence exists. From the proof of correctness of the $\mathrm{GKAM}_{\mathrm{CIC}}$ we know that if (ξ_i,σ_i,t_i,S_i) moves according to the third, seventh, eleventh, fourteenth, fifteenth or seventeenth reduction rule then $\mathcal{T}(\xi_i,\sigma_i,t_i,S_i) \rhd^+ \mathcal{T}(\xi_{i+1},\sigma_{i+1},t_{i+1},S_{i+1})$. Thus there is an index h such that for each $i > h$ (xi_i,σ_i,t_i,S_i) moves to $(\xi_{i+1},\sigma_{i+1},t_{i+1},S_{i+1})$ using either the first, the eighth or the thirteenth rule. Let us consider the following strict and well-founded order $<_C$ over closures (and GKAM configurations, when we ignore the last component):

$(\xi,\sigma,t) <_C (\xi',\sigma',t')$
if $(\xi,\sigma) <_{E\times SE} (\xi',\sigma')$
or if $(\xi,\sigma) = (\xi',\sigma')$ and t is structurally smaller than t'

We now prove that the three rules in the infinite sequence are all strictly decreasing functions from continuation to continuation according to the $<_C$ ordering relation. We reach the absurdum by noting that, although the infinite sequence is strictly decreasing, the order is well-founded.

- First rule: $([\alpha_n\ ;\ \ldots\ ;\ \alpha_1], \sigma, i, S)$ moves into $(\pi_1(\text{from\_env}\ \alpha_i), \pi_2(\text{from\_env}\ \alpha_i), \pi_3(\text{from\_env}\ \alpha_i), \pi_4(\text{from\_env}\ \alpha_i).S)$

  $[\alpha_n\ ;\ \ldots\ ;\ \alpha_i] <_{E \times SE} [\alpha_n\ ;\ \ldots\ ;\ \alpha_1]$ because of the requirements on every GKAM$_{CIC}$ strategy. Moreover $\pi_1 \circ \text{hd} \circ \text{from\_env} \circ < \pi_1, \pi_2 > ([\alpha_n\ ;\ \ldots\ ;\ \alpha_i], \sigma) <_{E \times SE} ([\alpha_n\ ;\ \ldots\ ;\ \alpha_i], \sigma)$ because of the first GKAM$_{CIC}$ Liveliness Condition.

- Eighth rule: $(\xi, [\gamma_1/\nu_1\ ;\ \ldots\ ;\ \gamma_n/\nu_n])$ moves into $(\pi_1(\text{from\_ens}\ \gamma_i), \pi_2(\text{from\_ens}\ \gamma_i), \pi_3(\text{from\_ens}\ \gamma_i), \pi_4(\text{from\_ens}\ \gamma_i).S)$

  $[\gamma_1/\nu_1\ ;\ \ldots\ ;\ \gamma_i/\nu_i] <_{E \times SE} [\gamma_1/\nu_1\ ;\ \ldots\ ;\ \gamma_n/\nu_n]$ because of the requrirements on every GKAM$_{CIC}$ strategy. Moreover $\pi_2 \circ \text{hd}) \circ \text{from\_ens} \circ < \pi_1, \pi_2 > (\xi, [\gamma_1/\nu_1\ ;\ \ldots\ ;\ \gamma_i/\nu_i]) <_{E \times SE} (\xi, [\gamma_1/\nu_1\ ;\ \ldots\ ;\ \gamma_i/\nu_i])$ because of the second GKAM$_{CIC}$ Liveliness Condition.

- Thirteenth rule: (ξ, σ, MN, S) moves into $(\xi, \sigma, M, (\text{to\_stack}(\xi, \sigma, N)).S)$
  M is structurally smaller than MN.

□

Corollary 9.27 (Termination) *If* to_stack, to_env, to_ens, from_env *and* from_ens *are total functions and if a term* t *is strongly normalizable, then the execution of the GKAM$_{CIC}$ on the input* $(\emptyset, \emptyset, t, \emptyset)$ *will terminate. Moreover, if the final state reached is* (ξ, σ, t', S)*, then* $t \triangleright^* \mathcal{T}(\xi, \sigma, t', S)$ *and* $\mathcal{T}(\xi, \sigma, t', S)$ *is in weak head normal form.*

Proof: The corollary is a trivial consequence of the GKAM$_{CIC}$ correctness theorem and the GKAM$_{CIC}$ liveliness theorem, together with the hypothesis of strong normalization of the term. □

9.4 The Environment

In Chap. 8.1 we presented the reduction and typing rules of CIC. One of the first notions introduced was the notion of environment. An environment E is an ordered list of object definitions and declarations. According to the typing rules WF-Empty, WF-Const, WF-Ind, an environment $E = [d_1, \ldots, d_n]$ is well-formed ($\mathcal{WF}(E)[]$)) when, for each $i \in \{1, \ldots, n\}$, $\mathcal{WF}([d_1, \ldots, d_{i-1}][])$ and $[d_1, \ldots, d_{i-1}][] \vdash \mathcal{WF}(d_i)$. Finally, every typing rule of CIC that retrieves a declaration or a definition from the environment has a premise that checks the well-formedness of the environment (and the current context).

The above characterization of the environments is unpractical and too strict. It is unpractical because every time we consider a reference to an object the whole environment is checked again for well-formedness. It is too strict because the linear order imposed by the nature of the data structure is artificial: the well-formedness of a declaration or definition d_i does not really depend on all the entries $d_1, \ldots, d_{i-1}$. Indeed, we can easily define a *partial* order relation $d_i \preccurlyeq_E d_j$ such that d_i precedes d_j in E if and only if it is not possible to remove one or more declarations from E in such a way that the obtained environment E' is well-formed and d_j does not belong to E'. Notice that, due to the constraints on the universes, the order does not coincide with the transitive closure of the relation "*is referenced by*" that was discussed in Chap. 4.

To solve the first problem, the usual solution is to assume the well-formedness of the environment when checking the well-formedness of a declaration or definition (see, for instance, [57]). The well-formedness of the environment is then granted by construction: the environment can be augmented only by adding well-formed declarations or definitions. This solution easily solves the computational problem of checking the same environment more than once. Moreover, it is very elegant since it breaks the mutual recursion between the rules that check the well-formedness of the environment (WF-Empty, WF-Const, WF-Ind) and the rules that infer the type of a reference (Const, Ind, Constr).

This observation is exploited in the latest versions of the Coq system to break the recursion between the module that manage the (unsafe) environments and the modules that deal with type-checking and reduction: the latter two modules depend on the first one, that does not check the well-formedness of the declarations or definitions that are appended to the current environment. Finally, another module is introduced. Its role is managing safe environments, that are an opaque data type implemented as an unsafe environment with the additional invariant of well-formedness. Since the empty environment is already well-formed, the append method of the module cheks the object that must be appended for well-formedness and, in case of success, appends it by calling the append method of the module that manages unsafe environments.

Unfortunately, this approach has the drawback of making the implicit order on the environment declarations and definitions coincide with the definition order, that is the order in which new declarations and definitions are introduced in the development. Indeed, the whole library must be seen as a single development and all the contributors must extend the single well-formed environment. Clearly, this assumption is in contrast with the distributed nature of mathematical developments we are interested in. Indeed, even the implementors of Coq have had to add the possibility of forking the development (by writing independent modules) and must handle with care the action of merging two forked developments (by requiring two modules at once). Due to the excessive strictness of the ordering relation, for instance, it may happen that two modules cannot be required at the same time to prove a theorem T, even if there exists a well-formed environment that is a subset of the union of the two modules and that allows to prove T [35].

We would like to overcome the limitations of the Coq system. In particular, we are not interested in introducing the notion of required modules, that inherently generates a notion of "library in scope". On the contrary, we prefer a paradigm where the notion of library is central and every declaration and definition is in scope. It is only when a reference is met that a well-formed environment is built on-the-fly in order to check the well-formedness of the reference. Thus an environment is, by definition, a well-formed subset of the library, and the whole library can be not well-formed. In a sense, we are trying to blur as much as possible the distinction between the distributed library and the environment of the proof assistant.

Since we need to construct a well-formed environment on-the-fly and since only the rules that deals with references require a well-formed environment, we need a way to extend a well-formed environment on-the-fly to obtain another well-formed environment that is a valid premise for checking the reference. Thus we seek a solution that is someway intermediate between the naive solution of the typing rules of Chap. 8.1 and the standard solution of [57].

In the next section we present an operational presentation of the typing rules of CIC where every typing rule has two inputs (a term t and a well-formed environment E) and an output E' (another well-formed environment) that extends E and that is the *minimal* extension of E (according to the relation $\preccurlyeq$ we just introduced) that allows the type-checking of t.

9.5 The Type-Checker

The implementation of our proof assistant's type-checker is absolutely standard, although the environment is handled in a new way, as explained in the previous section. The implementation is not complete: the only unimplemented rules are the rules that deal with the universe hierarchy. In Coq these rules are factorized out in their own module, that is responsible for updating the universe graph and checking its acyclicity (since a cycle corresponds to a logical inconsistency). The reason because we did not implement that module is that the universe hierarchy adopted so far for CIC is currently under discussion [30, 46] and it will surely change in the next release of Coq. Thus, we think that it is worth waiting for the assessment of the new proposed theories before proceding to implement the module. In the meantime, our implementation accepts inconsisten terms. In particular, the paradoxes that were found in Girard's System U [33] hold with minor modifications for our system.

The type-checking rules implemented are a syntax-oriented version of the rules we gave in Chapter 8 for the Calculus of (Co)Inductive Construction extended with casts and explicit named substitutions. In Sect. 10.1 we will extend the calculus to deal with metavariables, and we will also give the syntax-oriented version of the typing rules. For an early reference to translating type-checking rules to a syntax oriented form, the reader can consult [57]. A major difference with the approach of [57] is our peculiar handling of the environment, described in the previous section.

We introduce three new kind of judgments:

- $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$ means that the term t has type T in the environment E' and in the context Γ, where Φ is a list of references that are not in the domain of E, E' is an extension of E and no reference in Φ is in the domain of E'.
- $\mathtt{fetch}(\Phi, E, r, E', d)$ means that the reference r does not belong to Φ and that d is the declaration or definition in E' that r is a reference to, where E' is an extension of E such that d belongs to E' and no reference in Φ belongs to the domain of E'.
- $\langle\Phi\rangle E[\Gamma] \vdash \mathcal{WF}(r, \sigma) \bowtie E'$ means that the reference r does not belong to Φ and that the explicit named substitution σ is a well-formed explicit named substitution for r

As usual, to shorten the rules, in the following rules the symbol $\star$ stands for the following mutual declaration of (co)inductive types:

$$\begin{array}{l}(\nu_1, \ldots, \nu_n)\Pi x_1 : U_1 \ldots \Pi x_h : U_h.\\ \quad \{i_1 : A_1 := \{k_1^1 : C_1^1 \ ; \ \ldots \ ; \ k_{n_1}^1 : C_{n_1}^1\} \ ;\\ \quad \ldots \ ;\\ \quad i_m : A_m := \{k_1^m : C_1^m \ ; \ \ldots \ ; \ k_{n_m}^m : C_{n_m}^m\}\\ \quad \}_{\text{(CO)IND}}\end{array}$$

O-WF-ens

$$\frac{\begin{array}{c}\text{the object referenced by } r \text{ is abstracted over } \nu_i : T_i \text{ for each } i\\ (\langle\Phi\rangle E_i[\Gamma] \vdash t_i : T_i' \bowtie E_{i+1})_{i=1,\ldots,n}\\ (E_{n+1}[\Gamma] \vdash T_i' \downarrow T_i\{t_1/\nu_1 \ ; \ \ldots \ ; \ t_{i-1}/\nu_{i-1}\}_\nu)_{i=1,\ldots,n}\\ (t_i = \nu_i \text{ or } t_i \text{ is not convertible with } \nu_i)_{i=1,\ldots,n}\end{array}}{\langle\Phi\rangle E_1[\Gamma] \vdash \mathcal{WF}(r, t_1/\nu_1 \ ; \ \ldots \ ; \ t_n/\nu_n) \bowtie E_{n+1}}$$

O-Ax-Prop

$$\frac{}{\langle\Phi\rangle E[\Gamma] \vdash \mathbf{Prop} : \mathbf{Type} \bowtie E}$$

O-Ax-Set

$$\frac{}{\langle\Phi\rangle E[\Gamma] \vdash \mathbf{Set} : \mathbf{Type} \bowtie E}$$

O-Ax-Acc

$$\frac{}{\langle\Phi\rangle E[\Gamma] \vdash \mathbf{Type} : \mathbf{Type} \bowtie E}$$

O-Var

$$\frac{(x : T) \in \Gamma \quad \textit{or} \quad (x : T := t) \in \Gamma}{\langle\Phi\rangle E[\Gamma] \vdash x : T \bowtie E}$$

O-SectVar

$$\frac{\mathtt{fetch}(\Phi, E, v, E', (v(v_1, \ldots, v_n) : T)) \qquad \langle\Phi\rangle E'[\Gamma] \vdash \mathcal{WF}(v, \sigma) \bowtie E''}{\langle\Phi\rangle E[\Gamma] \vdash v[\sigma] : T\{\sigma\}_v \bowtie E''}$$

O-Const

$$\frac{\mathtt{fetch}(\Phi, E, c, E', (c(v_1, \ldots, v_n) : T := t)) \qquad \langle\Phi\rangle E'[\Gamma] \vdash \mathcal{WF}(c, \sigma) \bowtie E''}{\langle\Phi\rangle E[\Gamma] \vdash c[\sigma] : T\{\sigma\}_v \bowtie E''}$$

O-Ind

$$\frac{\mathtt{fetch}(\Phi, E, i_j, E', \star) \qquad \langle\Phi\rangle E'[\Gamma] \vdash \mathcal{WF}(i_j, \sigma) \bowtie E''}{\langle\Phi\rangle E[\Gamma] \vdash i_j[\sigma] : (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j)\{\sigma\}_v \bowtie E''}$$

O-Constr

$$\frac{\mathtt{fetch}(\Phi, E, k_l^j, E', \star) \qquad \langle\Phi\rangle E'[\Gamma] \vdash \mathcal{WF}(k_l^j, \sigma) \bowtie E''}{\langle\Phi\rangle E[\Gamma] \vdash k_l^j[\sigma] : (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.C_l^j)\{\sigma\}_v \bowtie E''}$$

O-Prod

$$\frac{\begin{array}{ll}\langle\Phi\rangle E[\Gamma] \vdash T : S_1 \bowtie E' & whd(E'[\Gamma], S_1) = s_1 \\ \langle\Phi\rangle E'[\Gamma, (x : T)] \vdash U : S_2 \bowtie E'' & whd(E''[\Gamma, (x : T)], S_2) = s_2\end{array}}{\langle\Phi\rangle E[\Gamma] \vdash \Pi x : T.U : s_2 \bowtie E''}$$

O-Lam

$$\frac{\langle\Phi\rangle E[\Gamma] \vdash T : S \bowtie E' \quad whd(E'[\Gamma], S) = s \quad \langle\Phi\rangle E'[\Gamma, (x : T)] \vdash t : U \bowtie E''}{\langle\Phi\rangle E[\Gamma] \vdash \lambda x : T.t : \Pi x : T.U \bowtie E''}$$

O-LetIn

$$\frac{\langle\Phi\rangle E[\Gamma] \vdash t_1 : T_1 \bowtie E' \quad \langle\Phi\rangle E'[\Gamma, (x : T_1 := t_1)] \vdash t_2 : T \bowtie E''}{\langle\Phi\rangle E[\Gamma] \vdash \mathtt{let}\ x := t_1\ \mathtt{in}\ t_2 : T\{t_1/x\} \bowtie E''}$$

O-App

$$\frac{\langle\Phi\rangle E[\Gamma] \vdash t : S \bowtie E' \quad whd(E'[\Gamma], S) = \Pi x : U.T \quad \langle\Phi\rangle E'[\Gamma] \vdash u : U' \bowtie E'' \quad E''[\Gamma] \vdash U \downarrow U'}{\langle\Phi\rangle E[\Gamma] \vdash (t\ u) : T\{u/x\} \bowtie E''}$$

O-Case

$$\frac{\begin{array}{ll}\mathtt{fetch}(\Phi, E_1, i_l, E_2, \star) & \\ \langle\Phi\rangle E_2[\Gamma] \vdash t : I \bowtie E_3 & whd(E_3[\Gamma], I) = (i_l\ t_1 \ldots t_h\ t'_1 \ldots t'_s) \\ \langle\Phi\rangle E_3[\Gamma] \vdash T : A \bowtie E_4 & [(i_l\ t_1 \ldots t_h) : A_l\{t_1/x_1\ ;\ \ldots\ ;\ t_h/x_h\}|A] \\ (\langle\Phi\rangle E_{3+j}[\Gamma] \vdash f_j : T_j \bowtie E_{4+j})_{j=1,\ldots,n_l} & \\ \multicolumn{2}{l}{(E_{4+n_l}[\Gamma] \vdash T_j \downarrow \Delta\{C_j^l\{t_1/x_1\ ;\ \ldots\ ;\ t_h/x_h\}, T, (k_j^l\ t_1 \ldots t_h)\})_{j=1,\ldots,n_l}}\end{array}}{\langle\Phi\rangle E_1[\Gamma] \vdash \left(\begin{array}{l} < T > \mathrm{CASES}_{i_l}\ t\ \mathrm{OF} \\ \quad k_1^l\ \overline{x^1} \Rightarrow f_1 \mid \ldots \mid k_{n_l}^l\ \overline{x^n} \Rightarrow f_{n_l} \\ \mathrm{END}\end{array}\right) : (T\ t'_1 \ldots t'_s\ t) \bowtie E_{4+n_l}}$$

O-Fix

$$\frac{\begin{array}{l}(\langle\Phi\rangle E_l[\Gamma] \vdash T_l : s_l \bowtie E_{l+1})_{l=1,\ldots,m} \\ (\langle\Phi\rangle E_{m+l}[\Gamma, (x_1 : T_1), \ldots, (x_m : T_m)] \vdash f_l : T'_l \bowtie E_{m+l+1})_{l=1,\ldots,m} \\ (E_{2m+1}[\Gamma, (x_1 : T_1), \ldots, (x_m : T_m)] \vdash T'_l \downarrow T_l)_{l=1,\ldots,m} \\ (\mathrm{GD}\{x_1, \ldots, x_m, n_1, \ldots, n_m, x, [], T\})_{l=1,\ldots,m}\end{array}}{\langle\Phi\rangle E_1[\Gamma] \vdash \mathrm{FIX}_j\ \{x_1/n_1 : T_1 := f_1\ ;\ \ldots\ ;\ x_m/n_m : T_m := f_m\} : T_j \bowtie E_{2m+1}}$$

where $f_j = \lambda y_1 : T'_1 \ldots \lambda y_{n_j-1} : T'_{n_j-1}.\lambda x : (i\ t_1 \ldots t_n)T$

O-CoFix

$$\frac{\begin{array}{l}(\langle\Phi\rangle E_l[\Gamma] \vdash T_l : s_l \bowtie E_{l+1})_{l=1,\ldots,m}\\ (\langle\Phi\rangle E_{m+l}[\Gamma,(x_1 : T_1),\ldots,(x_m : T_m)] \vdash f_l : T_l' \bowtie E_{m+l+1})_{l=1,\ldots,m}\\ (E_{2m+1}[\Gamma,(x_1 : T_1),\ldots,(x_m : T_m)] \vdash T_l' \downarrow T_l)_{l=1,\ldots,m}\\ (GC_0\{x_1,\ldots,x_m,f_l\})_{l=1,\ldots,m}\end{array}}{\langle\Phi\rangle E_1[\Gamma] \vdash \mathrm{COFIX}_j\ \{x_1 : T_1 := f_1\ ;\ \ldots\ ;\ x_m : T_m := f_m\} : T_j \bowtie E_{2m+1}}$$

O-Cast

$$\frac{\langle\Phi\rangle E[\Gamma] \vdash t : T_2 \bowtie E' \qquad \langle\Phi\rangle E'[\Gamma] \vdash T_1 : S \bowtie E'' \qquad E''[\Gamma] \vdash T_1 \downarrow T_2}{\langle\Phi\rangle E''[\Gamma] \vdash (t :: T_1) : T_1 \bowtie E''}$$

O-fetch-SectVar-hit

$$\frac{(v(v_1,\ldots,v_n) : T) \in E}{\mathtt{fetch}(\Phi, E, v, E, (v(v_1,\ldots,v_n) : T))}$$

O-fetch-SectVar-miss

$$\frac{\begin{array}{l}v \notin \Phi \qquad (v(v_1,\ldots,v_n) : T) \in \mathcal{L} \qquad \langle\Phi; v\rangle E[[]] \vdash T : S \bowtie E' \qquad \mathit{whd}(E'[[]], T) = s\\ \{v_1,\ldots,v_n\} \text{ is the set of all the section variables that occur free in } T\end{array}}{\mathtt{fetch}(\Phi, E, v, E'; (v(v_1,\ldots,v_n) : T), (v(v_1,\ldots,v_n) : T))}$$

O-fetch-Const-hit

$$\frac{(c(v_1,\ldots,v_n) : T := t) \in E}{\mathtt{fetch}(\Phi, E, v, E, (c(v_1,\ldots,v_n) : T := t))}$$

O-fetch-Const-miss

$$\frac{\begin{array}{l}c \notin \Phi \qquad (c(v_1,\ldots,v_n) : T := t) \in \mathcal{L}\\ \langle\Phi; c\rangle E[[]] \vdash T : S \bowtie E' \qquad \mathit{whd}(E'[[]], T) = s\\ \langle\Phi; c\rangle E'[[]] \vdash t : T' \bowtie E'' \qquad E''[[]] \vdash T' \downarrow T\\ \{v_1,\ldots,v_n\} \text{ is the set of all the section variables that occur free in } T \text{ and } t\end{array}}{\mathtt{fetch}(\Phi, E, v, E''; (c(v_1,\ldots,v_n) : T := t), (c(v_1,\ldots,v_n) : T := t))}$$

O-fetch-Ind-or-Constr-hit

$$\frac{\star \in E}{\mathtt{fetch}(\Phi, E, r, E, \star)} \qquad \begin{array}{l}\text{for } r = i_j \text{ where } j \leq m\\ \text{or } r = k_l^j \text{ where } j \leq m \text{ and } l \leq n_j\end{array}$$

O-fetch-Ind-or-Constr-miss

$$\frac{\begin{array}{l}\star \in \mathcal{L} \qquad i_1,\ldots,i_m,k_1^1,\ldots,k_{n_m}^m \notin \Phi \qquad i_1,\ldots,i_m,k_1^1,\ldots,k_{n_m}^m \notin E\\ i_1,\ldots,i_m,k_1^1,\ldots,k_{n_m}^m\ \mathtt{distinct}\\ (\langle\Phi; i_1,\ldots,i_m,k_1^1,\ldots,k_{n_m}^m\rangle E_j[[]] \vdash (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j : s_j \bowtie E_{j+1})_{j=1,\ldots,m}\\ \langle\Phi; i_1,\ldots,i_m,k_1^1,\ldots,k_{n_m}^m\rangle (E_{m+\sum_{k=1}^{j-1} n_k+l}[[y_1 : A_1^*\ ;\ \ldots\ ;\ y_m : A_m^*]] \vdash\\ \quad C_l^j\{y_1/i_1\ ;\ \ldots\ ; y_m/i_m\} : s_l^j \bowtie E_{m+\sum_{k=1}^{j-1} n_k+l+1})_{j=1,\ldots,m\ ;\ l=1,\ldots,n_j}\end{array}}{\mathtt{fetch}(\Phi, E, r, E; \star, \star)}$$

where $r = i_j$ and $j \leq m$ or $r = k_l^j$ and $j \leq m,\ l \leq n_j$
and $A_j^* = \Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j$ for $j \in \{1,\ldots,m\}$
and the following syntactic restriction holds: in $C_1^1,\ldots,C_{n_m}^m$ the first h arguments of the applications whose head is one of the inductive types $i_1,\ldots,i_m$ must be $x_1,\ldots,x_h$

Before proving the correctness and completenss of our syntax-driven typing rules, we need to prove several technical lemmas.

Lemma 9.28 *For each environment* E*, context* Γ *and term* t*, if* $\mathcal{WF}(E)[\Gamma]$ *and* $E[\Gamma] \vdash t : T$*, then for each reference* r *in* E, Γ *or* t *there exists an object declaration* o *such that* $\mathtt{fetch}(\emptyset, E, r, E, o)$.

Proof: The proof is by induction over the height of the two proofs of $\mathcal{WF}(E)[\Gamma]$ and $E[\Gamma]$. All the inductive steps but the Case step are trivial. The base cases where a reference r is encountered are the cases Const, SectVar, Ind and Constr. All the cases have a premise $o \in E$ where r is a reference to an object declared or defined in o. Thus we can apply respectively the rule O-fetch-Const-hit, O-fetch-SectVar-hit, O-fetch-Ind-or-Constr–hit or O-fetch-Ind-or-Constr–hit. Hence the thesis by inspection of the above rules. The proof of the inductive step Case is similar to the proof of the previous base cases. □

Corollary 9.29 *Under the same assumptions of lemma 9.28, if* δ *is a derivation of* $\langle\emptyset\rangle E[\Gamma]t : T' \bowtie E'$*, then every environment* E'' *that occurs in* δ *is equal to* E *and every list of references* Φ *that occurs in* δ *is empty.*

Proof: By induction over the height of δ. The inductive cases are trivial. The base cases are a consequence of lemma 9.28. □

Since we have removed all the checks upon the universe levels, we cannot establish a faithful correspondence between the operational syntax-directed rules and the typing rules we gave in Chap. 8.1. To retrieve the correspondence, from now on we consider the inconsistent variation of the rules of Chap. 8.1 obtained by replacing all the occurrences of **Type**(i) with **Type**. We are now ready to prove the main two results of this section.

Theorem 9.30 (Soundness) *For each environment* E*, context* Γ*, list of references* Φ *and term* t *such that the intersection between the domain of* E *and* Φ *is empty, if* $\mathcal{WF}(E)[\Gamma]$ *and* $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$ *then*

- $\mathcal{WF}(E')[\Gamma]$
- *the intersection between the domain of* E' *and* Φ *is empty*
- *for each couple of terms* t', T' *such that* $E[\Gamma] \vdash t' : T'$ *we also have* $E'[\Gamma] \vdash t' : T'$
- $E'[\Gamma] \vdash t : T$.

Proof: The proof is by induction over the shape of the derivation of $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$. None of the cases but O-SectVar, O-Const, O-Ind, O-Constr and O-Case browses the environment. Thus it is sufficient to apply the induction hypotheses, the Conv rule and the WF-Var rule to build a derivation for each premise of the suitable rule. We present here only a couple of cases, that are the most interesting. All the other cases are similar.

- Case O-Prod:
  By induction hypothesis we know that $\mathcal{WF}(E')[\Gamma]$ and that $E'[\Gamma] \vdash T : S_1$. Hence S_1 is well-typed. Applying the subject reduction theorem of CIC, from the hypothesis $whd(E'[\Gamma], S_1) = s_1$ we deduce that s_1 is also well-typed. Thus we can apply the Conv rule to the premise $whd(E'[\Gamma], S_1) = s_1$ and deduce that $E'[\Gamma] \vdash T : s_1$. Moreover, we can reuse the derivation of $E'[\Gamma] \vdash T : s_1$ to apply the Var rule and deduce $\mathcal{WF}(E')[\Gamma, (x : T)]$. Thus we can apply the second inductive hypothesis and conclude $E''[\Gamma, (x : T)] \vdash U : S_2$ and $E''[\Gamma] \vdash T : s_1$. Repeating the kind of reasoning that we have just done, we conclude that $E''[\Gamma, (x : T)] \vdash U : s_2$. Thus we can apply one among the rules Prod-SP and rule Prod-T, concluding that $E''[\Gamma] \vdash \Pi x : T.U : s_2$. The rest of the thesis is a trivial combination of the inductive hypotheses.

- Case O-App:
  By induction hypothesis we know that $\mathcal{WF}(E')[\Gamma]$ and that $E'[\Gamma] \vdash t : S$. Hence S is well-typed. Applying the subject reduction theorem of CIC, from the hypothesis $whd(E'[\Gamma], S) = \Pi x : U.T$ we deduce that $\Pi x : U.T$ is also well-typed. Thus we can apply the Conv rule to the premise $whd(E'[\Gamma], S) = \Pi x : U.T$ and deduce that $E'[\Gamma] \vdash t : \Pi x : U.T$. Applying the second inductive hypothesis, we conclude that $E''[\Gamma] \vdash t : \Pi x : U.T$. Repeating the same kind of reasonment for the second inductive hypothesis, we conclude that $E''[\Gamma] \vdash u : U$ Thus we can apply the rule App to conclude $E''[\Gamma] \vdash (t\ u) : T\{u/x\}$. The rest of the thesis is a trivial combination of the inductive hypotheses.

Let us face the more complex cases O-SectVar, O-Const, O-Ind, O-Constr and O-Case. They all have a premise that is a derivation of $\mathtt{fetch}(\Phi, E, r, E', d)$ for some reference r and environment E'. Using the inductive hypothesis, we can reduce the first three parts of the thesis to proving that $\mathcal{WF}(E')[]$, that the intersection between the domain of E' and Φ is empty and that for each couple of term t', T' such that $E[\Gamma] \vdash t' : T'$ we also have $E'[\Gamma] \vdash t' : T'$. We proceed by cases on the last rule of the derivation of $\mathtt{fetch}(\Phi, E, r, E')$.

If it is an "hit" rule O-fetch-SectVar-hit , O-fetch-Const-hit or O-fetch-Ind-or-Constr-hit, then $E' = E$. Hence the first three parts of the thesis. If we are not dealing with a Case, then by induction hypothesis we conclude also that $E[\Gamma] \vdash \mathcal{WF}(r, \sigma)$. Thus, choosing the suitable rule among Const, SectVar, Constr and Ind, we conclude the thesis. If we are examining a Case, instead, the rest of the proof is similar to the "simple" cases above.

Otherwise, let E'' be E' withouth the last entry. Using the inductive hypotheses, we deduce that $\mathcal{WF}(E'')[]$ and that the intersection between the domain of E'' and $\Phi; r$ is empty. Thus $r \notin E''$. Using the inductive hypotheses and the Conv rule, we can construct a derivation for each premise of the suitable environment well-formedness rule (i.e. WF-Const, WF-SectVar or WF-Ind). Hence we conclude that $\mathcal{WF}(E')[]$. Moreover the intersection between the domain of E' and Φ must be empty, since the intersection of the domain of E'' and Φ is empty. Moreover, since E' is a well-typed extension of E'' and since all the new binders introduced in E' do not belong to the domain of E'', we also know that $E''[\Gamma] \vdash t' : T'$ implies $E'[\Gamma] \vdash t' : T'$ for each couple of terms t', T'. If we are not dealing with a Case, by induction hypothesis we conclude also that $E'[\Gamma] \vdash \mathcal{WF}(r, \sigma)$. Thus, choosing the suitable rule among Const, SectVar, Constr and Ind, we conclude the thesis. If we are examining a Case, instead, the rest of the proof is similar to the "simple" cases above. □

Theorem 9.31 (Completeness) *For each environment* E*, context* Γ *and term* t*, if* $\mathcal{WF}(E)[\Gamma]$ *and* $E[\Gamma] \vdash t : T$*, then there exists an unique term* T' *such that* $\langle\emptyset\rangle E[\Gamma] \vdash t : T' \bowtie E$ *and* $E[\Gamma] \vdash T =_{\beta\delta\iota\zeta} T'$*.*

Proof: Let δ be a derivation of $E[\Gamma] \vdash t : T$. We build a derivation of $\langle\emptyset\rangle E[\Gamma] \vdash t : T'$ (for some T') by recursion on the shape of δ.

The base case is an application of one of the following axioms: Ax-Prop, Ax-Set and Ax-Acc. We build the corresponding derivation by istantiating the corresponding axiom in the syntax directed presentation.

Among the inductive cases, there are the rules that deals with the references to an object: Const, SectVar, Ind, Constr and Case. Because of corollary 9.29 and lemma 9.28, we can generate a derivation by choosing the corresponding rule and istantiating the first premise with the derivation of $\mathtt{fetch}(\emptyset, E, r, E, o)$ obtained by applying the corresponding fetch-hit rule. The other premises are generated from the recursion hypotheses.

The other interesting case is the application of the Conv rule. The theorem holds by choosing $T' = T_2$. For the derivation of $\langle\emptyset\rangle E[\Gamma] \vdash t : T_2$ we can just pick the recursive hypothesis.

All the other cases left are a trivial consequence of corollary 9.29: we just have to pick the corresponding rule in the syntax directed calculus and use the recursive hypotheses to fill in the premises. □

The soundness property grants two useful implementation properties. The first one is what we name *minimal environment reconstruction*: given a term t, if $\langle\emptyset\rangle\emptyset[[]] \vdash t : T \bowtie E$, then E is the smallest well-formed environment required to type-check t. When we integrate the module that handles the constraints about the universe levels, the inferred constraints will also be minimal, in the sense that no constraint can be relaxed and no object can be removed from the environment without losing the well-typedness of the declaration.

We could call the second property *environment extension*: in order to check a new term, definition or declaration for well-formedness, we do not have to throw away the already computed and well-formed environment. We can simply invoke the type-checking procedure passing the current environment as the initial environment. In case of success, the returned environment is the minimal extension of the initial environment that grants the well-formedness property. Unfortunately, in case of failure we are not allowed to conclude that the given term, declaration or definition is not well-typed. For instance, once the module that handles the constraints about the universe levels will be implemented, it may be the case that no well-formed extension of the current environment grants well-formedness, simply because of a conflicting set of constraints. Nevertheless, it may be the case that some of the conflicting constraints are derived from parts of the initial environment that are not necessary to type-check the given input (see [35] and 116). Thus, to conclude that the input is not well-formed, we are obliged to repeat the check starting from the empty environment.

Notice that the problem we have just noticed is not a side-effect of our special handling of the environment. Indeed, our approach is the only one that allows to solve the problem, by recostructing the minimal necessary environment. In the Coq system, for instance, when such a case is met the only solution is to factorize out of the required modules all the definitions necessary to type the incriminated definition. Basically this amounts to performing by hand the operation of environment reconstruction.

In truth, the cases where the problem arises are very uncommon. All the cases known by the author are to be found in the encoding of set theories and the theory of categories where big and small sets (or categories) are mixed together in the same development. Nevertheless, once again we feel that, in the context of mathematical knowledge management, the problem must be faced in a serious way, since it can deeply hinder the possibility of extending or reusing a development in the long term.

Finally, from the point of view of the developer, the practice of environment reconstruction paves the way to the notion of *trusted information source*. A trusted information source is a subset of the distributed library of knowledge that is closed (in the sense that it is well-formed) and that we can trust. Concretely, trusting an information source $\mathcal{L}'$ amounts to adding three new inference rules that deal with the $\mathtt{fetch}$ relation, and to slightly change the reduction rules that deal with δ and ι reduction:

O-fetch-SectVar-trust

$$\frac{(\nu(\nu_1,\ldots,\nu_n):\mathsf{T}) \in \mathcal{L}'}{\mathtt{fetch}(\Phi,\mathsf{E},\nu,\mathsf{E};(\nu(\nu_1,\ldots,\nu_n):\mathsf{T}),(\nu(\nu_1,\ldots,\nu_n):\mathsf{T}))}$$

O-fetch-Const-trust

$$\frac{(c(\nu_1,\ldots,\nu_n):\mathsf{T}:=t) \in \mathcal{L}'}{\mathtt{fetch}(\Phi,\mathsf{E},\nu,\mathsf{E};(c(\nu_1,\ldots,\nu_n):\mathsf{T}:=t),(c(\nu_1,\ldots,\nu_n):\mathsf{T}:=t))}$$

O-fetch-Ind-or-Constr-trust

$$\frac{\star \in \mathcal{L}'}{\mathtt{fetch}(\Phi, E, r, E; \star, \star)} \qquad \begin{array}{l}\text{for } r = i_j \text{ where } j \leq m \\ \text{or } r = k^j_l \text{ where } j \leq m \text{ and } l \leq n_j\end{array}$$

The rules for δ- and ι-reduction are changed by replacing the premise $(c : T := t) \in E$ (respectively $\star \in E$) with the premise $(c : T := t) \in E$ *or* $(c : T := t) \in \mathcal{L}'$ (respectively $\star \in E$ *or* $\star \in \mathcal{L}'$).

Basically, the three rules above means that a declaration or definition in a trusted information source $\mathcal{L}'$ can always be considered as well-formed. Equivalently, it says that $\mathcal{L}'$ can always be considered as a subset of any well-formed environment.

The three new rules above need to be made more complex as soon as the module that handles the universe constraints will be implemented. Indeed, the typical implementation consists in redefining the environment as a couple whose first component is the current definition of an environment and whose second component is a concrete representation of the set of constraints. Acyclic graphs are such a possible representation (the one of the Coq system), even if systems of equations are another possibility (adopted in at least some versions of the Lego system, [57]). The $\mathtt{fetch}$ relation also becomes responsible for extending the system of constraints by adding the constraints that are required to type-check the reference object. Thus, in order to obtain a sound implementation, we need to keep, for any object in a trusted information source, the minimal set of constraints imposed during its type-checking. Let $\mathcal{K}$ be the map that associates the constraints to a reference r. The three fetching rules above are changed in the following way:

O-fetch-SectVar-trust

$$\frac{(v(v_1, \ldots, v_n) : T) \in \mathcal{L}' \qquad \mathcal{WF}(\gamma \sqcup \mathcal{K}(v))}{\mathtt{fetch}(\Phi, \langle E, \gamma \rangle, v, \langle E; (v(v_1, \ldots, v_n) : T), \gamma \sqcup \mathcal{K}(v) \rangle, (v(v_1, \ldots, v_n) : T))}$$

O-fetch-Const-trust

$$\frac{(c(v_1, \ldots, v_n) : T := t) \in \mathcal{L}' \qquad \mathcal{WF}(\gamma \sqcup \mathcal{K}(v))}{\mathtt{fetch}(\Phi, \langle E, \gamma \rangle, v, \langle E; (c(v_1, \ldots, v_n) : T := t), \gamma \sqcup \mathcal{K}(v) \rangle, (c(v_1, \ldots, v_n) : T := t))}$$

O-fetch-Ind-or-Constr-trust

$$\frac{\star \in \mathcal{L}' \qquad \mathcal{WF}(\gamma \sqcup \mathcal{K}(v))}{\mathtt{fetch}(\Phi, \langle E, \gamma \rangle, r, \langle E; \star, \gamma \sqcup \mathcal{K}(r) \rangle, \star)} \qquad \begin{array}{l}\text{for } r = i_j \text{ where } j \leq m \\ \text{or } r = k^j_l \text{ where } j \leq m \text{ and } l \leq n_j\end{array}$$

where γ is the set of constraints in input, $\sqcup$ is the operation that merges two set of constraints and $\mathcal{WF}(\gamma \sqcup \gamma')$ is the new judgement that checks if the merge of two set of satisfiable constraints is satisfiable.

The version of the rules that does not handle universe constraints has already been implemented in the current prototype and it is used to speed-up proof-checking in the HELM Web service that delivers the proof-checking service: the subset of the distributed library that is provided by the site of the University of Bologna and that corresponds to the standard library of Coq (which is self-contained and also well-formed) is considered a trusted information source.

Chapter 10

Unification, Refinement and Ambiguous Parsing

In this section we describe the most important functionality that is implemented outside the kernel: *unification*. Unification (and its one-sided variant, called matching) is used to find a metavariable instantiation that makes two open terms equal. It is the basic operation that underlies the searching procedure described in Chap. 5 and the intelligent tactics that are based on the SEARCH-ENGINE. It is also used to implement *refinement*: refinement is the procedure that takes an open term (i.e. a term with metavariables therein) and find a metavariable instantiation that makes the term well-typed. Refinement is heavily used in the implementation of several tactics and also in the parsing phase, since it is used to resolve implicit arguments. In his Ph.D. dissertation [82], McBride suggests to implement refinement first, and base all the tactics (even the primitive ones) on refinement.

In Sect. 10.1 we will extend CIC with explicit named substitutions with metavariables, that represent typed placeholders for missing terms. Unification will be described in Sect. 10.2. Sect. 10.3 is devoted to the description of the refinement procedure. In Sect. 10.4 we describe a new algorithm for parsing ambiguous mathematical expressions. The algorithm is heavily based on the refinement procedure, and represents a major improvement with respect to the traditional algorithm adopted for ambiguous parsing (e.g. the one used in the Grammatical Framework [105]). The idea underlying the algorithm, however, is quite simple: we can use refinement to prune not well-typed terms even before the parsing phase is completed. Thus the search space explored by the disambiguation algorithm drastically shrinks and performances are boosted.

As usual, the last section of the chapter addresses the open problems and describes future work.

10.1 CIC with Explicit Named Substitutions and Metavariables

Traditionally, logic deals with *completely specified terms* and *finished proofs*. The inference rules of a logical system specify what *is* derivable and exclude what *could* only be derivable under additional hypotheses. In a similar way, the typing rules of a type system specify what *is* well-typed and not what *could* only be derivable under additional hypotheses or instantiations.

Nevertheless, proof assistants and theorem provers have always been built to guide the user in the construction of a proof. This naturally implies that these systems should be able to deal with incomplete proofs and incomplete information. For instance, the user could leave some hypotheses unspecified, assume the existence of a particular term that enjoys some properties and then proceed in the proof, accumulating constraints on the assumed term. Only later, once the proof is almost finished, the user can inspect the constraints and instantiate the term with a value that satisfy all the constraints. At the same time she can also instantiate the unspecified hypotheses to complete the proof.

More generally, the user expects as much automation as possible from the system. This implies the possibility of leaving part of the information unspecified, and expecting the system to be able to infer the missing information or to report the inconsistency of the provided information. We can also imagine that an interactive system that finds more than one instantiation for the unspecified information could ask the user for a refinement, proposing the different alternatives. In particular, this is what we usually expect from a parser that deals with ambiguous mathematical expressions.

To achieve all the previous goals, the proof assistants and theorem provers developers have designed systems of open terms. For instance, there exists systems of open terms for dependently typed λ-calculi [66, 78, 82, 114]. A recent work [51] of Geuvers and Jojgov describes a logic of open proofs and open terms out of a type-system of open terms. The translation between the two systems is also yet another application of the Curry-Howard isomorphism, that naturally provides a guideline for the study.

In the rest of the section we describe our extension of CIC with Explicit Named Substitutions obtained by introducing metavariables. We also detail our implementation of unification. The obtained calculus is not completely original. First of all, it seems a simplification of the system of Strecker, that was developed for the Extended Calculus of Constructions (ECC), that is another extension of the Calculus of Construction that is less complex than CIC. Moreover, the calculus is quite similar to what is implemented in the Coq system, even if some implementation differences do exist. In particular, we make no distinction between metavariables (i.e. open derivations) and existential variables (open terms). The distinction is an implementation detail of Coq that was not present in the dissertation of Munoz [90], that presents the only calculus of open proofs that was developed by the team of Coq[1]. The other significative difference w.r.t. the implementation of Coq is that metavariables are physically instantiated when an unifier is applied, whereas in Coq an instantiation map is recorded.

10.1.1 Syntax

Let $\mathcal{M}$ be an enumerable set of metavariables, ranged over by $?1, ?2, \ldots$. We extend the syntax of the terms of CIC with explicit named substitutions as follows:

$$\begin{array}{lcll} t & ::= & \ldots & \text{term} \\ & | & ?i[\rho] & \text{metavariable occurrence} \\ \rho & ::= & t \,;\, \ldots \,;\, t & \text{explicit substitution} \end{array}$$

Let t and Γ be respectively a term and a context. We will denote the set of metavariables that occur free in t (respectively in Γ) with $\mathcal{M}(t)$ (respectively $\mathcal{M}(\Gamma)$).

10.1.2 Proof Problems

We associate a sequent (i.e. a context $\Gamma_{?i}$ and a type $T_{?i}$) to each metavariable $?i$. Intuitively, the metavariable $?i$ stands for a yet unknown term of type $T_{?i}$ whose free variables can only be a subset of the domain of $\Gamma_{?i}$.

Since CIC is a dependently typed calculus, metavariables can occur in the types or the contexts of other metavariables. Roughly speaking, in Coq terminology a metavariable with such a property is called an *existential variable*. All the other metavariables are simply called metavariables. The name existential variable derives from the habit to generate such a metavariable when a proof of an existential statement is required: the existential variable becomes the witness of the existential, and it occurs in the statement of the goal that proves the property of the witness.

Following the terminology of [114], we introduce the notion of *proof problem*. A proof problem is the collection of all the sequents associated to the metavariables that occur in the current unfinished proof (or in the sequent associated to another metavariable).

In the terminology used in the implementation of Coq, a proof problem is also called a `metasenv` (metavariables environment). In the rest of the thesis we will use the two terms as synonyms, preferring the terminology proof problem in the theoretical setting and the terminology `metasenv` in the description of the implementation.

Definition 10.1 (Proof problem) *A* proof problem $\mathcal{P}$ *is a finite list of sequents of the form* $\Gamma_{?i} \vdash ?i : T_{?i}$ *where for each metavariable that occur free in* $\mathcal{P}$ *there exists a corresponding sequent in* $\mathcal{P}$.

[1]The calculus of Munoz is a calculus of explicit substitutions. Thus it differs from the one implemented in Coq, where explicit substitutions are applied only to metavariable occurrences.

We introduce the notation $\mathcal{M}(\mathcal{P})$ for the set of metavariables that occur in $\mathcal{P}$.

In the rest of the dissertation, every time we assume the existence of a proof-problem we also implicitly assume the existence of the environment where the proof-problem is defined.

Since there is no clear interpretation of a proof problem where there exists two variables $?i$ and $?j$ such that each variables occurs in the type or the context of the other (see [114] for several examples), we are obliged to introduce a strict partial order to rule out the existence of cycles in the proof problem.

Definition 10.2 (Metavariables order in a proof problem) *Let $\mathcal{P}$ be a proof problem. Let $<_{\mathcal{P}}$ be the relations defined as: $?n_1 < ?n_2$ iff $?n_1 \in \mathcal{M}(\Gamma_{?n_2}) \cup \mathcal{M}(T_{?n_2})$. Let $\ll_{\mathcal{P}}$ be the transitive closure of $<_{\mathcal{P}}$.*

Definition 10.3 (Valid proof problem) *A proof problem $\mathcal{P}$ is a* valid proof problem *if and only if $\ll_{\mathcal{P}}$ is a strict partial order (or, equivalently, if and only if $\ll_{\mathcal{P}}$ is an irreflexive relation).*

Validity is a purely syntactical property of proof problems. Well-formedness strengthen validity by adding typing constraints on the sequents that compose the proof problem.

Definition 10.4 (Well formed proof problem) *A valid proof problem $\mathcal{P}$ is a* well-formed proof problem *in an environment E if an only if for all $(\Gamma_{?i} \vdash ?i : T_{?i}) \in \mathcal{P}$ we have $E[\Gamma_{?i}] \vdash T_{?i} : s$ (or, equivalently, $\langle\emptyset\rangle E[\Gamma_{?i}] \vdash T_{?i} : S \bowtie E'$ and $E'[\Gamma_{?i}] \vdash whd(S) = s$).*

The judgement $\mathcal{WF}(\mathcal{P})$ states that the valid proof problem $\mathcal{P}$ is well-formed in E.

Having extended the syntax of our calculus, we need to adjust the definition of substitution and the typing rules of our calculus. Since no computational rules are associated to metavariables, we do not have to change the definitions of reduction and convertibility.

10.1.3 Substitution Rules

Let $\mathcal{P}$ be a valid proof-problem. Substitution is extended to metavariables as follows:

$$?i[t_1, \ldots, t_n]\{\sigma\}_\nu = ?i[t_1\{\sigma\}_\nu, \ldots, t_n\{\sigma\}_\nu]?i[t_1, \ldots, t_n]\{\sigma\} = ?i[t_1\{\sigma\}, \ldots, t_n\{\sigma\}]$$

10.1.4 Convertibility Rules

Let $\mathcal{P}$ be a valid proof problem. Even if we do not have to change the definition of convertibility, we have to add the following inference rule to the definition of the $\downarrow$ judgement (i.e. the conversion algorithm):

Conv-Meta-Meta

$$\frac{(E[\Gamma] \vdash t_i \downarrow t_i')_{i=1,\ldots,n}}{E[\Gamma] \vdash ?i[t_1 \; ; \; \ldots \; ; \; t_n] \downarrow ?i[t_1' \; ; \; \ldots \; ; \; t_n']}$$

10.1.5 Typing Rules

Let $\mathcal{P}$ be a valid proof-problem. We introduce the following new typing rule for metavariables:

WF-ExplicitSubstitution

$$\frac{\begin{array}{c}(y_1 : T_1 \; ; \; \ldots \; ; \; y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \\ (E[\Gamma] \vdash t_i : T_i\{t_1/y_1 \; ; \; \ldots \; ; \; t_{i-1}/y_{i-1}\})_{i=1,\ldots,n}\end{array}}{E[\Gamma] \vdash \mathcal{WF}(?i, [t_1, \ldots, t_n])}$$

Meta

$$\frac{(y_1 : T_1 \; ; \; \ldots \; ; \; y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \qquad E[\Gamma] \vdash \mathcal{WF}(?i, [t_1, \ldots, t_n])}{E[\Gamma] \vdash ?i[t_1, \ldots, t_n] : T_{?i}\{t_1/y_1 \; ; \; \ldots \; ; \; t_n/y_n\}}$$

Notice that the proof problem $\mathcal{P}$ occurs in the inference rules above. This means that, from now on, the whole set of type-checking rules and all the typing judgements are parameterized over a proof problem. Notice, however, that the proof problem is fixed, since it is not affected by the reduction and typing rules.

However, we have to face another problem: the definition of well-formed proof problem depends on a well-formed environment E, i.e. an environment E such that $\mathcal{WF}(E)[]$. The previous judgement, being a typing judgement, must be parameterized over a well-formed proof problem. Thus the two judgements (well-formedness of a proof problem and well-formedness of an environment) are recursively defined. Since proving properties over the two recursive judgements would be quite complex, we prefer to impose constraints over the environment E to be able to break the recursion. Thus, we add the following constraint to the rules WF-SectVar, WF-Const and WF-Ind.

no metavariable can occur in the subterms of the object declaration/definition.

The new constraint grants a global property of well-formed environments: all the terms that occur in a well-formed environment are closed.

By close inspection of the rules WF-Var, WF-SectVar, WF-Const, WF-Ind, we can now notice that a derivation of $\mathcal{WF}(E)[\Gamma]$ must have the following shape: it must be a derivation of $\mathcal{WF}(E)[]$ followed by one application of WF-Var for each variable in the domain of Γ. The derivations that constitute the premises of the rules WF-Var and the rules used in the derivation of $\mathcal{WF}(E)[]$ have other sub-derivations of $\mathcal{WF}(E)[\Gamma']$ for some Γ' that have exactly the same shape.

The previous observation suggests the possibility of stratifying the judgement $\mathcal{WF}(E)[\Gamma]$ into two new judgements:

- $\mathcal{WF}(E)$ that states the well-formedness of E
- $\mathcal{WF}(E)(\mathcal{P})[\Gamma]$ that states the well-formedness of the context Γ in a well-formed environment E and in a well-formed proof problem $\mathcal{P}$

Since no metavariables can occur in any subterm of E, we deduce that the first judgement does not depend on any proof problem. On the contrary, the second judgement is parametric in the proof problem. Thus the division is sufficient to break the recursion between well-formedness of an environment and well-formedness of a proof problem: given an environment E, a valid proof problem $\mathcal{P}$ and a context Γ, we can always check the well-formedness of E first, then the well-formedness of $\mathcal{P}$ in E and finally the well-formedness of Γ in E and $\mathcal{P}$.

In the rest of the dissertation we will assume the previous stratification over the three kind of judgements $\mathcal{WF}(\mathsf{E})$, $\mathcal{WF}(\mathsf{E})(\mathcal{P})$ and $\mathcal{WF}(\mathsf{E})(\mathcal{P})[\Gamma]$. However, in order to avoid the redefinition of all the typing rules introduced in the previous chapters, we override the notation for the old judgement:

- $\mathcal{WF}(\mathsf{E})[]$ is redefined as $\mathcal{WF}(\mathsf{E})$
- $\mathcal{WF}(\mathsf{E})[\Gamma]$ is redefined as $\mathcal{WF}(\mathsf{E})(\mathcal{P})[\Gamma]$ where the proof problem $\mathcal{P}$ (that is well-formed in E) is fixed and should be clear from the context

Actually, the restriction on the absence of metavariables in the environment could be removed, allowing to store incomplete proofs and definitions in the the distributed library. In this way, for instance, it would be possible for a user to build her own development on top of an incomplete lemma. The current solution that keeps the constraint obliges the user that wants to store an incomplete proof in the environment to replace the metavariables with their functional representation (see Sect. 10.1.7) and then instantiate the generated conjectures with axioms (or variables). The main disadvantage of the current solution is the cluttering of the environment, due to the insertion of axioms that will be (hopefully) replaced by theorems in a later stage and that, a priori, could have no clear relevance on their owns (since they were generated from subgoals of another proof). The main advantage is the significative simplification of the theory due to the stratification of the judgements about well-formedness of environments, proof problems and contexts.

Since we have only extended the set of typing rules of our calculus, in the next section we need to extend the set of inference rules of the operational presentation too, and we need to reestablish the correctness and completeness results.

10.1.6 Typing Rules (Operational Presentation)

The syntax-oriented version of the typing rules introduced for metavariables is a simple adaptation of the typing rules presented in the previous section.

o-WF-ExplicitSubstitution

$$\frac{\begin{array}{l}(y_1 : T_1 \ ; \ \dots \ ; \ y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \\ (\langle\Phi\rangle E_i[\Gamma] \vdash t_i : T'_i \bowtie E_{i+1})_{i=1,\dots,n} \\ (\langle\Phi\rangle E_{n+1}[\Gamma] \vdash T'_i \downarrow T_i\{t_1/y_1 \ ; \ \dots \ ; \ t_{i-1}/y_{i-1}\})_{i=1,\dots,n}\end{array}}{\langle\Phi\rangle E_1[\Gamma] \vdash \mathcal{WF}(?i, [t_1, \dots, t_n]) \bowtie E_{n+1}}$$

o-Meta

$$\frac{(y_1 : T_1 \ ; \ \dots \ ; \ y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \qquad \langle\Phi\rangle E[\Gamma] \vdash \mathcal{WF}(?i, [t_1, \dots, t_n]) \bowtie E'}{\langle\Phi\rangle E[\Gamma] \vdash ?i[t_1, \dots, t_n] : T_{?i}\{t_1/y_1 \ ; \ \dots \ ; \ t_n/y_n\} \bowtie E'}$$

We also add the following constraint to the rules o-fetch-SectVar-miss, o-fetch-Const-miss and o-fetch-Ind-or-constr-miss.

no metavariable can occur in the subterms of the object declaration/definition.

Lemma 9.28 and corollary 9.29 hold without any major modification. We reestablish the soundness and completeness results by inspecting the inductive cases that derive from the two new rules that we have just introduced.

Theorem 10.5 (Soundness) *For each environment E, context Γ, list of references Φ and term t such that the intersection between the domain of E and Φ is empty, if $\mathcal{WF}(E)[\Gamma]$ and $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$ then*

- $\mathcal{WF}(E')[\Gamma]$
- *the intersection between the domain of E' and Φ is empty*
- *for each couple of term t', T' such that $E[\Gamma] \vdash t' : T'$ we also have $E'[\Gamma] \vdash t' : T'$*
- $E'[\Gamma] \vdash t : T$.

Proof: The proof proceeds like the proof of theorem 9.30. The inductive case that correspond to the new inference rule o-Meta is a trivial application of the inductive hypothesis. The inductive case that corresponds to the new inference rule o-WF-ExplicitSubstitution is a trivial application of the inductive hypotheses and the Conv rule. □

Theorem 10.6 (Completeness) *For each environment E, context Γ and term t, if $\mathcal{WF}(E)[\Gamma]$ and $E[\Gamma] \vdash t : T$, then there exists an unique term T' such that $\langle\emptyset\rangle E[\Gamma] \vdash t : T' \bowtie E$ and $E[\Gamma] \vdash T =_{\beta\delta\iota\zeta} T'$.*

Proof: The proof proceeds as the proof of theorem 9.31. The inductive cases that correspond to the new inference rules o-Meta and o-WF-ExplicitSubstitution are trivial applications of the inductive hypotheses and corollary 9.29. □

10.1.7 Metatheoretical Properties

In this section we present a translation of CIC extended with explicit named substitutions and metavariables into CIC extended with explicit named substitutions. As usual, we prove that the translation preserves substitution, reduction and typing rules. As a corollary we obtain that the usual metatheoretical properties of CIC (confluence, strong normalization, subject reduction, logical consistency, etc.) hold for our calculus.

Definition 10.7 (Functional representation of metavariables) *Let* E *be a well-formed environment of CIC and let* $\mathcal{P}$ *be a well-formed proof context in* E *such that* $\mathcal{M}(\mathcal{P}) = \{?1, \ldots, ?n\}$ *and* $?1 \ll \ldots \ll ?n$*. We define the translation* $[\![\_]\!]_{\mathcal{P}}$ *and* $[\![\_]\!]$ *from contexts of CIC extended with explicit named substitutions and metavariables to contexts of CIC extended with explicit named substitutions and metavariables as follows:*

$$[\![\Gamma]\!]_{\mathcal{P}} \stackrel{def}{=} x_1 : \mathcal{F}(\Gamma_{?1}, T_{?1}) \,;\, \ldots \,;\, x_n : \mathcal{F}(\Gamma_{?n}, T_{?n}) \,;\, [\![\Gamma]\!]$$

where

- $x_1, \ldots, x_n$ *are fresh variables*
- $\mathcal{F}(y_1 : T_1 \,;\, \ldots \,;\, y_n : T_n, T) \stackrel{def}{=} \Pi y_1 : [\![T_1]\!]. \ldots \Pi y_n : [\![T_n]\!].[\![T]\!]$
- *the translation* $[\![\_]\!]$ *from terms of CIC extended with explicit named substitutions and metavariables to terms of CIC extended with explicit named substitutions is defined recursively over the structure of its argument, and behaves as the identity function until a metavariable instance is met:*

$$[\![?i[t_1 \,;\, \ldots \,;\, t_n]]\!] = (x_i \; [\![t_1]\!] \; \ldots \; [\![t_n]\!])$$

The proposed translation simply replaces any metavariable with a functional term whose type is obtained by abstracting the type of the metavariable over its context. As a consequence, explicit substitutions become simple applications. From a logical point of view, the procedure is a form of Skölemization.

Notice that the well-formedness of the context generated by the translation is granted by the validity condition: if the proof problem $\mathcal{P}$ was not valid, then there would be two metavariables $?i$ and $?j$ that occur one in the sequent associated to the other; as a consequence one variable among x_i and x_j would occur free in the generated context.

The functional representation of a proof problem is undoubtedly appealing because of its simplicity. Unfortunately, it cannot be used to represent incomplete proofs and to make progress. For instance, let us consider the proof problem $\vdash ?1 : A \to A$. One possible way to progress in the proof would be to instantiate $?1$ with a λ-abstraction, obtaining the proof problem $H : A \vdash ?2 : A$.

Unfortunately, the functional representation of the two proof problems is exactly the same. Moreover, previous experience in other systems shows that it is simpler for the user to reason about a sequent — made of hypotheses and a conclusion — than to reason about functional terms.

As promised, we will now show that the translation to the functional representation preserves substitution, reduction and typing.

Lemma 10.8 *Let E be a well-formed environment and let $\mathcal{P}$ be a well-formed proof context in E. For each term t, substitution σ and explicit named substitution σ',*

- $[\![t\{\sigma\}]\!] = [\![t]\!]\{[\![\sigma]\!]\}$
- $[\![t\{\sigma'\}_\nu]\!] = [\![t]\!]\{[\![\sigma']\!]\}_\nu$

Proof: By induction over the structure of $t\{\sigma\}$ (respectively, $t\{\sigma'\}_\nu$). □

Theorem 10.9 *Let E be a well-formed environment and let $\mathcal{P}$ be a well-formed proof context in E. For each context Γ and for each couple of terms t, t', if $E[\Gamma] \vdash t \rhd t'$ then $E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash [\![t]\!] \rhd [\![t']\!]$.*

Proof: We proceed by induction over the structure of the term t. The case of a metavariable occurrence is trivial. Indeed, since a metavariable occurrence is not a redex, the reduction must be contextual and it must occur in one of the arguments of the explicit substitution. Thus the thesis is a consequence of the inductive hypotheses.

All the other cases can be proved by means of the inductive hypotheses and lemma 10.8. □

Corollary 10.10 *Let E be a well-formed environment and let $\mathcal{P}$ be a well-formed proof context in E. For each context Γ and for each couple of terms t, t', if $E[\Gamma] \vdash t =_{\beta\delta\iota\zeta} t'$ then $E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash [\![t]\!] =_{\beta\delta\iota\zeta} [\![t']\!]$.*

Theorem 10.11 *Let E be a well-formed environment and let $\mathcal{P}$ be a well-formed proof context in E. For each context Γ and for each couple of terms t, T, if $\mathcal{WF}(E)[\Gamma]$ and $E[\Gamma] \vdash t : T$ then $\mathcal{WF}(E)[[\![\Gamma]\!]_{\mathcal{P}}]$ and $E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash [\![t]\!] : [\![T]\!]$. Equivalently, if $\mathcal{WF}(E)[\Gamma]$ and $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$ then $\mathcal{WF}(E)[[\![\Gamma]\!]_{\mathcal{P}}]$ and $\langle\Phi\rangle E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash [\![t]\!] : T' \bowtie E'$ and $E'[[\![\Gamma]\!]_{\mathcal{P}}] \vdash T' =_{\beta\delta\iota\zeta} [\![T]\!]$*

Proof: We prove only the first formulation of the thesis. The equivalent formulation holds because of theorem 10.6. The proof proceeds by induction over the height of the derivation of $\mathcal{WF}(E)[\Gamma]$ or the derivation of $E[\Gamma] \vdash t : T$. Convertibility is dealt within corollary 10.10. Only the case of a metavariable occurrence is not a trivial applications of the inductive hypotheses.

Let $t = ?i[t_1 \;;\; \ldots \;;\; t_n]$ where $(y_1 : T_1 \;;\; \ldots \;;\; y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P}$.
By hypothesis, we know that $E[\Gamma] \vdash ?i[t_1, \ldots, t_n] : T_{?i}\{t_1/y_1 \;;\; \ldots \;;\; t_n/y_n\}$. Thus $E[\Gamma] \vdash \mathcal{WF}(?i, [t_1, \ldots, t_n])$, and consequently $E[\Gamma] \vdash t_i : T_i\{t_1/y_1 \;;\; \ldots \;;\; t_{i-1}/y_{i-1}\}$ for $i \in \{1, \ldots, n\}$. By induction hypothesis we know that, for $i \in \{1, \ldots, n\}$, $E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash [\![t_i]\!] : [\![T_i\{t_1/y_1 \;;\; \ldots \;;\; t_{i-1}/y_{i-1}\}]\!]$ that, by lemma 10.8, is equivalent to $E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash [\![t_i]\!] : [\![T_i]\!]\{[\![t_1]\!]/y_1 \;;\; \ldots \;;\; [\![t_{i-1}]\!]/y_{i-1}\}$.

To conclude the thesis, we need to provide a derivation of $E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash [\![?i[t_1\ ;\ \ldots\ ;\ t_n]]\!] : [\![T_{?i}\{t_1/y_1\ ;\ \ldots\ ;\ t_n/y_n\}$ that, by lemma 10.8, is equivalent to $E[[\![\Gamma]\!]_{\mathcal{P}}] \vdash (x_i\ t_1\ \ldots\ t_n) : [\![T_{?i}]\!]\{[\![t_1]\!]/y_1\ ;\ \ldots\ ;\ [\![t_n]\!]/y_n\}$.

By definition of $[\![\_]\!]_{\mathcal{P}}$ and $[\![\_]\!]$, x_i has type $\mathcal{F}(\Gamma_{?i}, T_{?i})$ that is equivalent to $\Pi y_1 : [\![T_1]\!]. \ldots \Pi y_n : [\![T_n]\!].[\![T_{?i}]\!]$. Thus, we can construct the expected derivation by applying n times the rule App and one time the rule Var. The second premise of each Var rule is instantiated with the i-th induction hypothesis considered above. The only derivation we still have to construct is the derivation of the first premise of the Var rule, that is $\mathcal{WF}(E)[[\![\Gamma]\!]_{\mathcal{P}}]$. We already know such a derivation by induction hypothesis.s Hence the thesis. □

10.1.8 Instantiations and Proof Progress

The extension of CIC with explicit named substitutions introduced so far allows us to describe unfinished proofs and missing information. It is also possible to compute with the open terms and to type-check them. Intuitively, an open term is well-typed when no mistake has been done so far. It does not grant the existence of any inhabitant for the type of the term.

So far, we have not discussed any way to proceed in an unfinished proof. Indeed, we have not even defined yet what an unfinished proof is.

Definition 10.12 ((Well-formed) proof status) *A* (well-formed) proof status *is a tuple* $(E, \mathcal{P}, T, t)$ *such that:*

- E *is a well-formed environment*
- $\mathcal{P}$ *is a well-formed proof problem in* E
- $E[] \vdash T : s$*, i.e.* T *is a well-typed type (the thesis of the proof)*
- $E[] \vdash t : T$*, i.e.* t *is an (incomplete) proof of* T

Notice that the previous definition characterizes not only proofs (i.e. proof status where the type of T is **Prop**) but also definitions (i.e. proof status where the type of T is **Set** or **Type**). Nevertheless, we adopt the terminology proof status since dealing with incomplete proofs is the principal problem that is faced in interactive theorem proving.

Notice also that we could easily introduce the notion of incomplete definition of a block of mutual inductive types: it simply corresponds to a triplet whose first component is a well-formed environment E, whose second component is a well-formed proof problem in E and whose third component is a block of mutual inductive types that is well-typed in E and $\mathcal{P}$. All the results that we will achieve in this and the next section are smoothly extended to incomplete definitions of blocks of mutual inductive types. Nevertheless, we prefer to avoid considering this kind of incomplete objects in the rest of the dissertation, since their introduction would only obfuscate the text by duplicating several definitions and proofs, without introducing any real new insight.

Making progress in a proof means transforming a well-formed proof status into a new well-formed proof status. Let $(E, \mathcal{P}, T, t)$ be a proof status. There exist four basic proof status transformations:

1. *Weakening:* it consists in introducing a new metavariable in the proof problem component. If the sequent associated to the metavariable is well-formed in E and $\mathcal{P}$, then the obtained proof status is well-formed.

2. *Clearing:* it consists in removing from the proof problem component one or more metavariables at once. If the removed metavariables do not occur in t, T or the sequents associated to the other metavariables in P, then the obtained proof status is well-formed.

3. *Perforating:* it consists in replacing a subterm of t, T or the sequent associated to one metavariable in P with a metavariable occurrence. A necessary condition for the well-formedness of the obtained term is that the type of the replaced term must be convertible with the type of the introduced metavariable occurrence. The condition is only necessary, since the replacement could break convertibility of the replaced term with another subterm. In a dependently typed setting, this may lead to a non well-typed term. Thus, the obtained proof problem must be checked for well-formedness.

4. *Instantiation:* it consists in clearing one or more metavariables and replacing their occurrences in the original proof-problem with a given term (that, in turn, can contain other metavariable occurrences).

The rest of this section is devoted to a precise definition of the instantiation operation and its metatheoretical properties.

Definition 10.13 (Valid instantiation) *A* valid instantiation ι *is a partial function from metavariables to terms that is subject to the following* occur check *condition:* $\mathrm{Dom}(\iota) \cap \mathcal{M}(\iota(\mathrm{Dom}(\iota))) = \emptyset$

We will denote valid instantiations with $\iota, \iota_1, \iota_2, \ldots$.

Definition 10.14 (Instantiation application) *Let* t *be a term and let* ι *be a valid instantiation. We will denote with* $\iota(t)$ *the application of* ι *to* t

$\iota(t)$ *is defined by recursion over* t *and it behaves as the identity function until the case of a metavariable occurrence is met:*

$$\iota(?i[t_1\ ;\ \ldots\ ;\ t_n]) = \begin{cases} \iota(?i)\{\iota(t_1)/y_1\ ;\ \ldots\ ;\ \iota(t_n)/y_n\} & \textit{if } ?i \in \mathrm{Dom}(\iota) \\ ?i[\iota(t_1)/y_1\ ;\ \ldots\ ;\ \iota(t_n)/y_n] & \textit{otherwise} \end{cases}$$

Instantiations can also be applied to whole proof status to progress in the proof.

Definition 10.15 (Proof progress by instantiation) *Let* $(E, \mathcal{P}, t, T)$ *be a proof status and let* ι *be a valid instantiation. We denote with* $\iota(\mathcal{P})$ *the proof problem obtained applying a valid instantiation to* $\mathcal{P}$.

$$\iota(\mathcal{P}) \stackrel{def}{=} [\iota(\Gamma_{?j_1}) \vdash ?1 : \iota(T_{?j_1})\ ;\ \dots\ ;\ \iota(\Gamma_{?j_m}) \vdash ?j_m : \iota(T_{?j_m})]$$

where $\{j_1, \dots, j_m\} = \mathcal{M}(\mathcal{P}) \setminus \mathrm{Dom}(\iota)$. *We define also the application of* ι *to a whole proof status:*

$$\iota(E, \mathcal{P}, t, T) \stackrel{def}{=} (E, \iota(\mathcal{P}), \iota(t), \iota(T))$$

Not every valid instantiation is correct, in the sense that applying an instantiation to a well-formed proof status can easily yield an incorrect proof status. Thus we will now start a metatheoretical study of the interactions of instantiation with substitution, reduction and type-checking. Once again, the lemmas and definitions are modelled on the dissertation of Strecker. Nevertheless, the main result of Strecker, well-typed instantiation respects typing, does not cover CIC (and thus also not our extension to CIC). Thus we are forced to introduce a weak notion of typing that is respected by instantiation.

Lemma 10.16 (Commutativity of instantiation and substitution) *Let* t *be a term,* σ *a substitution,* σ' *a named substitution and* ι *a valid instantiation.*

- $\iota(t\{\sigma\}) = \iota(t)\{\iota(\sigma)\}$
- $\iota(t\{\sigma'\}_\nu) = \iota(t)\{\iota(\sigma')\}_\nu$

Proof: By induction over the structure of $t\{\sigma\}$ (resp. $t\{\sigma'\}_\nu$). If t is an occurrence of the metavariable $?i \in \mathrm{Dom}(\iota)$, then

$$\begin{aligned}
&\iota(?i[t_1\ ;\ \dots\ ;\ t_n]\{\sigma\})\\
&\quad = \iota(?i[t_1\{\sigma\}\ ;\ \dots\ ;\ t_n\{\sigma\}])\\
&\quad = \iota(?i)[\iota(t_1\{\sigma\})\ ;\ \dots\ ;\ \iota(t_n\{\sigma\})]\\
&\quad = \iota(?i)[\iota(t_1)\{\iota(\sigma)\}\ ;\ \dots\ ;\ \iota(t_n)\{\iota(\sigma)\}]\\
&\qquad \text{by inductive hypothesis}\\
&\quad = \iota(?i)[\iota(t_1)\ ;\ \dots\ ;\ \iota(t_n)]\{\iota(\sigma)\}\\
&\quad = \iota(?i[t_1\ ;\ \dots\ ;\ t_n])\{\iota(\sigma)\}
\end{aligned}$$

The case of a named substitution is identical.

In all the other cases the thesis can be proved by means of a straightforward application of the inductive hypotheses and the definition of substitution and instantiation. □

Theorem 10.17 (Commutativity of instantiation and reduction) *Let* E *be an environment,* $\mathcal{P}$ *be a valid proof problem,* Γ *be a context. Let* t, t' *be terms and let* ι *be a valid instantiation. If* $E[\Gamma] \vdash t \rhd t'$ *then* $E[\iota(\Gamma)] \vdash \iota(t) \rhd \iota(t')$.

Proof: We proceed by induction on the derivation of $t \rhd t'$. If t is a metavariable occurrence, then the reduction must be contextual and we can easily apply the inductive hypotheses. All the other cases are trivial application of the inductive hypotheses and lemma 10.16. □

Corollary 10.18 (Instantiation preserves convertibility) *Let* E *be an environment,* $\mathcal{P}$ *be a valid proof problem,* Γ *be a context and* t_1, t_2 *be terms. If* $E[\Gamma] \vdash t_1 =_{\beta\delta\iota\zeta} t_2$ *then* $E[\Gamma] \vdash \iota(t_1) =_{\beta\delta\iota\zeta} \iota(t_2)$.

Unsurprisingly, we cannot expect all valid instantiation to preserve well-typedness. Thus, we introduce the rather self-evident notion of well-typed instantiation.

Definition 10.19 (Well-typed instantiations) *Let* E *be a well-formed environment and let* $\mathcal{P}$ *be a well-formed proof problem in* E*. A valid instantiation* ι *is* well-typed *in* $\mathcal{P}$ *if:*

- $\iota(\mathcal{P})$ *is a valid proof problem*
- *for every* $?i \in \mathcal{M}(\iota(\mathcal{P}))$, $E[\iota(\mathcal{P}), \iota(\Gamma_{?i})] \vdash \iota(?i) : \iota(T_{?i})$
  (or, equivalently, $\langle\emptyset\rangle E[\iota(\mathcal{P}), \iota(\Gamma_{?i})] \vdash \iota(?i) : T' \bowtie E'$ *and* $E'[\iota(\mathcal{P}), \iota(\Gamma_{?i})] \vdash T' \downarrow \iota(T_{?i})$*)*

The first constraint of this definition is meant to avoid creating cycles in the metavariables dependency graph. Notice that, unless we restrict ourselves to very special classes of instantiations [114], there is no simple syntactical check over ι and $\ll_{\mathcal{P}}$ that grants the irreflexivity of $\ll_{\iota(\mathcal{P})}$. For instance, let us consider the following well-typed proof problem in $E = nat : \textbf{Set}$:

$$\begin{array}{rcl} & \vdash & ?1 : \textbf{Set} \\ x : ?1 & \vdash & ?b : nat \to nat \\ & \vdash & ?2 : nat \to nat \\ y : ?2 & \vdash & ?c : \textbf{Set} \\ z : ?c & \vdash & ?a : \textbf{Set} \end{array}$$

We have that $\ll_{\mathcal{P}}$ is the smallest relation such that $?1 \ll_{\mathcal{P}} ?b$ and $?2 \ll_{\mathcal{P}} ?c \ll_{\mathcal{P}} ?a$. Let us now consider the instantiation $\iota = \{?1 \mapsto ?a \ ; \ ?2 \mapsto ?b\}$. It is easy to see that ι preserves $\ll_{\mathcal{P}}$ (for any reasonable definition of preservation): indeed $\iota(?1) = ?a$ and for each variable $?i$ that occurs in $\iota(?1)$ — that is just $?a$ in this example — and for each variable $?j$ such that $?1 \ll_{\mathcal{P}} ?j$ we have $?j \not\ll_{\mathcal{P}} ?i$; the same property holds for $?2$. Moreover, ι respects also the second well-typedness condition. Notwithstanding the latter two observations, $\iota(\mathcal{P})$ is not valid, since $?a \ll_{\iota(\mathcal{P})} ?b \ll_{\iota(\mathcal{P})} ?c \ll_{\iota(\mathcal{P})} ?a$.

For ECC, The condition of well-typedness is sufficient (but not necessary) [114] to grant preservation of well-typedness. Moreover, the system obtained is flexible enough to allow inserting metavariable occurrences in any position (i.e. subterm) of a term.

Unfortunately, this is not the case for CIC. Indeed, a close inspection of the typing rules that deal with inductive types show that there are several places where syntactical constraints are introduced on the terms. For instance, the constraints *guarded by constructors* and *guarded by destructors* require that no recursive call appears in a given term in general position. So far, we did not specify how these constraints are extended to deal with metavariables, implicitly assuming that no special behaviour is required: a term t occurs (in general position) in $?i[t_1 \ ; \ \ldots \ ; \ t_n]$ if t occurs in one term among $t_1, \ldots, t_n$; it is free otherwise. Notice, however, that the default behaviour is unreasonably too restrictive: a metavariable that occurs in a term that is subject to the GC or GD condition cannot be parameterized over the recursive call. In other words, a metavariable that occurs inside a FIX or COFIX cannot be instantiated with a term that contains a recursive call.

Obviously, the last restriction is practically unfeasible, since it does not allow the user to interactively refine a term or proof by (co)recursion/(co)induction. One possibility to avoid the problem would be to introduce in a proof problem a new set of syntactic constraints for each metavariable. For instance, we could record for each metavariable a list of GC or GD constraints (together with their arguments) and we could redefine the notion of well-typed instantiation to consider the new constraints: an instantiation would be well-typed if the subterm satisfy the constraints. In practice, though, this would not solve all the problems: since a tactic reduce a metavariable to a new set of goals by creating an instantiation, every tactic should contain appropriate code to make the new goals “inherit” the syntactical constraints in such a way that the result instantiation is well-typed and the inherited constraints are as general as possible.

For simplicity reasons, we do not adopt here the proposed solution. On the contrary, we prefer a lax approach: we introduce a weak notion of typing for open terms that enjoy the following properties:

1. Every well-typed term is also weakly typed.
2. A closed weakly typed term is also well-typed.
3. If a term is weakly typed but it is not well-typed, than there exists two instantiations that make it respectively a well-typed term and a closed not well-typed term.

Intuitively, a well-typed open term and a weakly typed open term are both unfinished constructions that are correct so far and that could be completed in a correct way. Well-typedness means that to complete the construction in a correct way the user must only care about the inhabitation of the conjectures (goals). Weakly typedness means that the user must not only care about the inhabitation of the conjectures, but that she must also care about the inhabitant that she will choose.

From the point of view of the implementer, weak typedness is problematic, since it forces the system to type-check again every FIX and COFIX once the last metavariable occurrence in it is instantiated. For big terms, this check can be rather expensive. If instantiation preserved well-typedness, instead, type-checking could be performed incrementally, just checking the well-typedness of every instantiation before applying it to the current proof status.

Definition 10.20 (Weak typing) *Let E be a well-formed environment and $\mathcal{P}$ be a well-formed proof problem in E. We introduce the following two judgements:*

- $\mathcal{WF}_\omega[E][\mathcal{P}](\Gamma)$
  that means that the context Γ is weakly typed in E and $\mathcal{P}$
- $E[\mathcal{P},\Gamma] \vdash_\omega t : T$
  that means that Γ is a weakly typed context in E and $\mathcal{P}$ and that t is weakly typed of typed T in E, Γ and $\mathcal{P}$

The inference rules of the two judgements are equal to the typing rules given in Chap. 8.1 where all the syntactical constraints are redefined to hold inconditionally for metavariable instances.

We can now face the principal result of the section: instantiation preserves weak typing.

Theorem 10.21 (Instantiation preserves weak typing) *Let E be a well-formed environment, $\mathcal{P}$ be a well-formed proof problem in E and ι a well-typed instantiation in E and $\mathcal{P}$. For each context Γ and couple of terms t and T,*

- *if $\mathcal{WF}_\omega[E][\mathcal{P}](\Gamma)$ then $\mathcal{WF}_\omega[E][\iota(\mathcal{P})](\iota(\Gamma))$*
- *if $E[\mathcal{P},\Gamma] \vdash_\omega t : T$ then $E[\iota(\mathcal{P}),\iota(\Gamma)] \vdash_\omega \iota(t) : \iota(T)$*

Proof: We proceed by induction over the shape of the derivations of $\mathcal{WF}_\omega[E][\mathcal{P}](\Gamma)$ and $E[\mathcal{P},\Gamma] \vdash_\omega t : T$. The only case that is not a trivial application of the inductive hypothesis, lemma 10.16 and theorem 10.17 is the case of a metavariable occurrence that belongs to the domain of ι.

Let t be $?i[t_1 \; ; \; \dots \; ; \; t_n]$ such that $(y_1 : T_1 \; ; \; \dots \; ; \; y_n : T_n \vdash \; ?i : T_{?i}) \in \mathcal{P}$ and let $?i \in \mathrm{Dom}(\iota)$. Then $T = T_{?i}\{t_1/y_1 \; ; \; \dots \; ; \; t_n/y_n\}$ and $E[\mathcal{P},\Gamma] \vdash \mathcal{WF}_\omega(?i,[t_1,\dots,t_n])$. Destructuring this derivation, we obtain a derivation of $E[\mathcal{P},\Gamma] \vdash_\omega t_i : T_i\{t_1/y_1 \; ; \; \dots \; ; \; t_{i-1}/y_{i-1}\}$ for each $i \in \{1,\dots,n\}$. By induction hypothesis we obtain a derivation of $E[\iota(\mathcal{P}),\iota(\Gamma)] \vdash_\omega \iota(t_i) : \iota(T_i\{t_1/y_1 \; ; \; \dots \; ; \; t_{i-1}/y_{i-1}\})$ for each $i \in \{1,\dots,n\}$. Applying lemma 10.16 we derive

$$E[\iota(\mathcal{P}),\iota(\Gamma)] \vdash_\omega \iota(t_i) : \iota(T_i)\{\iota(t_1)/y_1 \; ; \; \dots \; ; \; \iota(t_{i-1})/y_{i-1}\} \quad \forall i \in \{1,\dots,n\} \tag{10.22}$$

By definition of well-typed instantiation, we know that

$$E[\iota(\mathcal{P}),\iota(\Gamma_{?i})] \vdash_\omega \iota(?i) : \iota(T_{?i}) \tag{10.23}$$

We need to prove $E[\iota(\mathcal{P}),\iota(\Gamma)] \vdash_\omega \iota(?i[t_1 \; ; \; \dots \; ; \; t_n]) : \iota(T_{?i}\{t_1/y_1 \; ; \; \dots \; ; \; t_n/y_n\}$, that, using again lemma 10.16, is equivalent to $E[\iota(\mathcal{P}),\iota(\Gamma)] \vdash_\omega \iota(?i)\{\iota(t_1)/y_1 \; ; \; \dots \; ; \; \iota(t_n)/y_n\} : \iota(T_{?i})\{\iota(t_1)/y_1 \; ; \; \dots \; ; \; \iota(t_n)/y_n\}$.

The thesis is obtained applying the substitution lemma of CIC — that can easily be proved also for CIC extended with explicit named substitution and metavariables — to 10.23 and 10.22. □

So far, the occur check condition imposed over valid instantiations and the second condition for well-typed instantiations have played no role. Nevertheless, they are necessary to the application of well-typed instantiations to well-formed proof problems and proof status, as the proof of the following corollary shows.

Corollary 10.24 (Instantiation preserves weakly well-formed proof status) *Let* $(E, \mathcal{P}, t, T)$ *be a weakly well-formed proof-status and let* ι *be a well-typed instantiation.* $\iota(E, \mathcal{P}, t, T)$ *is a weakly well-formed proof status.*

Proof: Since $(E, \mathcal{P}, t, T)$ is a weakly well-formed proof status by hypothesis, we know that:

- E is a well-formed environment
- $\mathcal{P}$ is a weakly well-formed proof problem in E
- $E[\mathcal{P}, \emptyset] \vdash_\omega T : s$
- $E[\mathcal{P}, \emptyset] \vdash_\omega t : T$

By theorem 10.21, we know that $E[\iota(\mathcal{P}), \emptyset] \vdash_\omega \iota(T) : s$ and $E[\iota(\mathcal{P}), \emptyset] \vdash_\omega \iota(t) : \iota(T)$. Moreover, applying theorem 10.21 to the hypothesis that $\mathcal{P}$ is weakly well-formed, we conclude that $E[\iota(\mathcal{P}), \iota(\Gamma_{?i})] \vdash_\omega \iota(T_{?i}) : s$ for each $?i \in \mathcal{M}(\iota(\mathcal{P})) \supset \mathcal{M}(\iota(\mathcal{P})) \setminus Dom(\iota)$. Thus, we just proved that if $\iota(\mathcal{P})$ is a valid proof problem, then $\iota(\mathcal{P})$ is also a well-formed proof problem.

To prove the validity of $\iota(\mathcal{P})$, we still have to check two conditions:

- *For each metavariable in* $\mathcal{M}(\iota(\mathcal{P}))$ *there exists a corresponding sequent in* $\iota(\mathcal{P})$.
  The occur check condition over ι grants that $Dom(\iota) \cap \mathcal{M}(\iota(Dom(\iota))) = \emptyset$. Let $?i \in \mathcal{M}(\iota(\mathcal{P}))$. Then either $?i \notin Dom(\iota)$ or $?i \in \mathcal{M}(\iota(Dom(\iota)))$. Hence, using the occur check condition, we conclude that in both cases $?i \notin Dom(\iota)$. Since, by hypothesis, $?i \in \mathcal{M}(\mathcal{P})$, we conclude $?i \in \mathcal{M}(\mathcal{P}) \setminus Dom(\iota)$, that is the set of metavariables that have a sequent in $\iota(\mathcal{P})$.
- $\ll_{\iota(\mathcal{P})}$ *must be a strict partial order*.
  True by definition of well-typed instantiation (first constraint).

□

The definition of weak-typedness is easily reformulated also for the operational presentation of the typing rules. This fundamental result holds also for the operational presentation.

10.2 Unification

An unification problem consists in two open terms that are well-typed in the same well-formed environment, proof problem and context. A solution to an unification problem is a well-typed instantiation that makes both terms equivalent, for a given notion of equivalence $\equiv$.

Let $\equiv$ be a fixed equivalence relation over terms in an environment, proof problem and context.

Definition 10.25 ((Weak) unification problem) *Let* E *be a well-formed environment,* $\mathcal{P}$ *be a well-formed proof problem in* E *and* Γ *be a well-formed context in* E *and* $\mathcal{P}$*. A* (weak) unification problem *is a couple of well-typed terms in* $E[\Gamma]$*.*

We will use the notation $E[\mathcal{P};\Gamma] \vdash t_1 \overset{?}{\equiv} t_2$ for the unification problem (t_1, t_2) in E, Γ and $\mathcal{P}$. Notice that, dealing with unification, we are obliged to make $\mathcal{P}$ explicit in the notation, since the proof problem is no longer fixed and it is changed by the application of a solution. In the same way, for the rest of the chapter we will write $E[\mathcal{P};\Gamma]$ in place of $E[\Gamma]$ every time the proof problem is not fixed.

Definition 10.26 (Solution to an unification problem) *Let* E *be a well-formed environment,* $\mathcal{P}$ *be a well-formed proof problem in* E *and* Γ *be a well-formed context in* E *and* $\mathcal{P}$*. Let* t_1*,* t_2 *be two well-typed terms in* $E[\Gamma]$*. A* solution *to the unification problem* $E[\mathcal{P};\Gamma] \vdash t_1 \overset{?}{\equiv} t_2$ *is a well-typed instantiation* ι *such that* $\iota(t_1) \equiv \iota(t_2)$ *in* E*,* $\iota(\mathcal{P})$ *and* $\iota(\Gamma)$*.*

Notice that we do not require the two terms of a unification problem to have the same type. On the contrary, a solution to a unification problem is also responsible for making the two types convertible. Strecker [114] observes that, in the context of dependent types, this seems to be a necessary requirement.

Having given general definitions that apply to any kind of unification, we must now address particular kinds of unification, by varying the equivalence relation $\equiv$. Previous theoretical investigations and implementation evidence shows that only very weak form of equivalence are practically necessary for interactive theorem proving. The reason is twofold. First of all, the undecidability of higher-order unification and related techniques is a well established result. Secondly, full fledged higher-order unification is never necessary to implement a tactic: since tactics deal with very limited subsets of terms and situations, they can usually be factorized in three steps. The first step is responsible for exploiting domain knowledge to reduce the higher-order unification to an essentially first order case, by performing η-expansions and term transformations that pave the way to solution by mimicking. The second step just applies a first order unification technique to find a solution. The third step applies the unifier and performs a few reduction steps to undo the expansions performed by the first step.

For instance, the `Elim` tactic of Coq is implemented in this way: given a goal P and a term t, the tactic performs η-expansion over all the subterms of t that are instantiated in the hypotheses of the elimination scheme of the type of t. If $t_1, \ldots, t_n$ is the set of all the subterms that enjoy the property, the goal generated by the first step is $(\lambda x_1 . \ldots . \lambda x_n . P\{x_1/t_1 \ ; \ \ldots \ ; \ x_n/t_n\} \ t_1 \ \ldots \ t_n)$. Notice how the knowledge over the elimination scheme properties is exploited in the expansion phase. We can now simply apply first order unification to unify the conclusion of the elimination scheme (that is $(?P \ ?1 \ \ldots \ ?n)$) with the new goal. The solution will instantiate $?P$ with $\lambda x_1 . \ldots . \lambda x_n . P\{x_1/t_1 \ ; \ \ldots \ ; \ x_n/t_n\}$ and $?i$ with t_i for each i. The second phase ends with the application of the application of the elimination scheme constant to the parameters deduced by the instantiation. All the new goals generated by the second step will be implications whose thesese are β-redexes of the form $(\lambda x_1 . \ldots . \lambda x_n . P\{x_1/t_1 \ ; \ \ldots \ ; \ x_n/t_n\} \ t'_1 \ \ldots \ t'_n)$. The third step simply reduces the β-redexes, yielding goals of the form $H_1 \Rightarrow \ldots \Rightarrow H_n \Rightarrow P\{t'_1/t_1 \ ; \ \ldots \ ; \ t'_n/t_n\}$, as expected.

Thus, we will now consider only one equivalence relation, that yields an essentially first order unification algorithm. From a practical point of view, the equivalence relation is not very useful. Indeed, it is used only to establish the soundness and completeness results of the unification judgement. Whereas soundness is a fundamental property, the completeness of the algorithm is much less interesting, since unification always plays the role of an heuristic in interactive theorem proving. Thus, we will not make the equivalence relation explicit. We just characterize it implicitly as follows: $\equiv_\alpha \subset \equiv \subset =_{\beta\delta\iota\zeta}$ (i.e. the equivalence relation identifies more terms than α-conversion, but less terms than $\beta\delta\iota\zeta$-conversion). The previous assertion can be proved formally by showing the soundness of the judgement w.r.t. $\beta\delta\iota\zeta$-conversion and its completeness w.r.t. α-conversion. Since the second result is practically completely unuseful (since α-conversion is a too weak equivalence relation), in this dissertation we only show the soundness result.

10.2.1 First-Order Unification

We are interested in a simple, operational equivalence relation at least as strong as α-convertibility. Since first order unification in a higher order setting with dependent types is always seen as a sort of necessarily incomplete heuristic [114], we just describe our algorithm and we prove its soundness with respect to convertibility.

We derive a judgement that finds a solution to an unification problem by modifying as less as possible the algorithm that decides convertibility of two terms presented in Sect. 9.2. The approach should not be surprising: indeed unification is just a sort of convertibility over open terms that, instead of producing a boolean result, may return a set of constraints (the unifier) that, if satisfied, grant convertibility of the two terms.

We will adopt the same approach in the next chapter, when we will define refinement: refinement will just be a modified version of type-checking that generates a set of constraints that, once satisfied, grant that a given open term is well-typed.

Let us now present the formalization of the unification algorithm we implemented in our prototype.

The judgement builds the solution to an unification problem by visiting the two terms that must be unified. Simplifying a bit, every time a metavariable is met in one of the two terms, a new instantiation for it is added to the solution. The new instantiation applies immediately to the terms in the domain of the solution computed so far. It also applies immediately to the proof problem and to the rest of the two terms that is still to be visited.

To formalize the algorithm, Strecker [114] follows the description we just gave literally. Thus he introduces an asymmetric composition operator $\uplus$ whose first argument is a solution and whose second argument is an instantiation of a single metavariable with a term. The operator returns a new instantiation ι that he applies immediately to the proof problem and to all the terms that he mets in the rest of the recursion.

Applying an instantiation every time a new metavariable is met is not an efficient solution. Indeed, it is similar to the naive implementation of reduction. The idea that allows to implement reduction in an efficient way is that of collecting all the substitutions in a substitution environment, that is accessed only when a bound variable is met. We apply the same idea to unification, introducing *metavariable substitution environments*, denote by Σ. The solution, however, is not novel, and it is adopted in many efficient implementations of substitutions.

Definition 10.27 ((Valid) metavariable substitution environment) *A* metavariable substitution environment Σ *is an ordered list* $?1 := t_1 \;;\; \ldots \;;\; ?n := t_n$ *of couples metavariable/term such that* $?1 \neq \ldots \neq ?n$. *The metavariable substitution environment is also* valid *if for each* $i \in \{1, \ldots, n\}$ *the instantiation* $\{?1 \mapsto t_1\} \circ \ldots \circ \{?i \mapsto t_i\}$ *is a valid instantiation.*

Lemma 10.28 *If* $?1 := t_1 \;;\; \ldots \;;\; ?n := t_n$ *is a valid metavariable substitution environment, then* $?i$ *does not occur in* t_j *for each* $i < j$.

Proof: By definition of valid metavariable substitution environment, $\iota \stackrel{def}{=} \{?1 := t_1 \;;\; \ldots \;;\; ?j := t_j\}$ must be a valid instantiation. By definition of valid instantiation, $\mathrm{Dom}(\iota) \cap \iota(\mathrm{Dom}(\iota)) = \emptyset$. Since $?i, ?j \in \mathrm{Dom}(\iota)$, $?i$ cannot be in $\iota(?j)$. Since $?1 \neq \ldots \neq ?n$ by definition of metavariable substitution environment, $\iota(?j) = t_j$. Thus $?i \notin t_j$. □

Since a valid metavariable substitution environment must be ordered according to the result of the previous lemma, it is not possible to extend a metavariable substitution environment simply by appending an instantiation to the end. Thus we define the following judgement that allows extension of metavariable substitution environments:

Definition 10.29 (Metavariable substitution environment extension) *Let Σ be a valid metavariable substitution environment. Let $?n \notin \mathrm{Dom}(\Sigma)$ and let t_n be a term. The judgement $\Sigma \oplus (?n := t_n) = \Sigma'$ holds iff there exists a valid metavariable substitution environment Σ' such that $\mathrm{Dom}(\Sigma') = \mathrm{Dom}(\Sigma) \cup \{?n\}$ and $\forall ?i \in \mathrm{Dom}(\Sigma).\Sigma'(?i) = \Sigma(?i)$ and $\Sigma'(?n) = t_n$.*

If $\Sigma \oplus (?n := t_n) = \Sigma'$ holds then Σ' is called a direct extension *of Σ with $?m := t_m$.*

We also say that Σ_n is an extension of Σ_1 if there exists a sequence of metavariable substitution environment $\Sigma_2, \ldots, \Sigma_{n-1}$ such that for each i Σ_{i+1} is a direct extension of Σ_i.

Since the semantics of a metavariable substitution environment is just a telescopic substitution, we can easily define the notion of application of a metavariable substitution environment.

Definition 10.30 (Application of a metavariable substitution environment)
Let $\Sigma = [?1 := t_1 \ ; \ \ldots \ ; \ ?n := t_n]$ be a metavariable substitution environment and let t be a term. The application $\Sigma(t)$ of Σ to t is defined as $\iota_n(\ldots \iota_1(t) \ldots)$ where ι_i is the instantiation $\{?i \mapsto t_i\}$.

The application of a metavariable substitution environment is extended to proof problems and proof status as usual.

The following technical lemma will be useful in the proof of soundness.

Lemma 10.31 (Extension preserves convertibility) *Let Σ be a valid metavariable substitution environment. Let Σ' an extension of Σ. For each environment E, proof problem $\mathcal{P}$, context Γ and for each couple of terms (t_1, t_2), if $E[\mathcal{P}; \Sigma(\Gamma)] \vdash \Sigma(t_1) \downarrow \Sigma(t_2)$ holds, then $E[\mathcal{P}; \Sigma'(\Gamma)] \vdash \Sigma'(t_1) \downarrow \Sigma'(t_2)$ also holds.*

Proof: The thesis is a consequence of corollary 10.18. □

Definition 10.32 (Well-typed metavariable substitution environment) *Let E be a well-formed environment and $\mathcal{P}$ a well-formed proof problem in E. A metavariable substitution environment $?1 := t_1 \ ; \ \ldots \ ; \ ?n := t_n$ is* well-typed *in E and $\mathcal{P}$ if for each $i \in \{1, \ldots, n\}$ the instantiation $\{?1 \mapsto t_1\} \circ \ldots \circ \{?i \mapsto t_i\}$ is a well-typed instantiation.*

Theorem 10.33 (The application of a metavariable substitution environment preserves weak typing)
Let E be a well-formed environment, $\mathcal{P}$ be a well-formed proof problem in E and Σ a well-typed metavariable substitution environment in E and $\mathcal{P}$. For each context Γ and couple of terms t and T, if $E[\Gamma] \vdash_\omega t : T$ then $E[\Sigma(\Gamma)] \vdash_\omega \Sigma(t) : \Sigma(T)$

Proof: Trivial by definition of well-typed metavariable substitution environment and theorem 10.21.
□

Corollary 10.34 (The application of a metavariable substitution environment preserves weakly well-formed proof status)
Let $(E, \mathcal{P}, t, T)$ be a weakly well-formed proof-status and let Σ be a well-typed metavariable substitution environment in E and $\mathcal{P}$. $\Sigma(E, \mathcal{P}, t, T)$ is a weakly well-formed proof status.

Proof: Trivial by definition of well-typed metavariable substitution environment and corollary 10.24. □

Before presenting the unification judgement, we identify syntactically a class of terms that have remarkable properties.

Definition 10.35 (Stiff terms) *A term* t *is* stiff *in an environment* E*, a proof problem* $\mathcal{P}$ *and a context* Γ *if* t *is in weak head normal form and if none of following conditions hold for the* ζ*-normal form of* t *obtained by unfolding all the definitions in* Γ*:*

- $t = ?i[\rho]$
- $t = (t_1\ t_2)$ *and* $\mathcal{M}(t_1) \neq \emptyset$
- $t = \mathsf{CASES}\ t'\ \mathsf{OF} \ldots \mathsf{END}$ *and* $\mathcal{M}(t') \neq \emptyset$
- $t = (\mathsf{Fix}_i\{\ldots\}\, t_1\ \ldots\ t_n)$ *such that the decreasing parameter of the* i*-th function defined in the block is* j *and* $\mathcal{M}(t_j) \neq \emptyset$

Theorem 10.36 *If* t *is a stiff term and* ι *an instantiation, then* $\iota(t)$ *is a stiff term and the heads of* t *and* $\iota(t)$ *are equal.*

Proof: By definition of stiff terms and by inspection of the reduction rules, applying the instantiation cannot create a weak head redex (since the next term to reduce according to the weak strategy is already in weak head normal form and moreover it is closed). Thus $\iota(t)$ is in weak head normal form and, by definition of instantiation application, its head is equal to the head of t. Moreover, $\iota(t)$ is stiff since for each closed term t', $\iota(t')$ is also closed. □

Stiff terms are important since stiffness is an invariant of instantiation. Moreover, stiff terms are already in weak head normal forms and for terms in weak head normal form the detection of convertibility is easier (since the initial reduction phase of the conversion algorithm can be skipped). As we will see in the proof of soundness of the unification judgement, the detection of stiff terms that cannot be unified is also made simpler. In particular, we will use the following fact.

Fact 10.37 *Let* E *be a well-formed environment,* $\mathcal{P}$ *a well-formed proof problem in* E*,* Γ *a well-formed context in* E *and* $\mathcal{P}$ *and* t_1 *and* t_2 *two terms in weak head normal form. If* t_1 *and* t_2 *are not instances of the same constructor of CIC with explicit named substitutions and metavariables (e.g.* t_1 *is the occurrence of a mutual inductive type whereas* t_2 *is an application), then* t_1 *and* t_2 *are not convertible.*

By combining the previous fact with theorem 10.36, we conclude that the following corollary holds:

Corollary 10.38 *Let E be a well-formed environment, $\mathcal{P}$ a well-formed proof problem in E, Γ a well-formed context in E and $\mathcal{P}$ and t_1 and t_2 two terms in weak head normal form. If t_1 and t_2 are not instances of the same constructor of CIC with explicit named substitutions and metavariables (e.g. t_1 is the occurrence of a mutual inductive type whereas t_2 is an application), then, for each metavariable substitution environment Σ, $\Sigma(t_1)$ and $\Sigma(t_2)$ are not convertible.*

Definition 10.39 (Unification judgement) *Let E be a well-formed environment, $\mathcal{P}$ be a proof problem in E and Σ a metavariable substitution environment such that $\Sigma(\mathcal{P})$ is well-formed and Σ is well-typed in $\Sigma(\mathcal{P})$. Let $\Sigma(\Gamma)$ be a well-formed context in E and $\Sigma(\mathcal{P})$ and let (t_1, t_2) be an unification problem in E, $\mathcal{P}$ and Γ. We introduce the following four judgments:*

- $\mathsf{restrict}(\mathcal{P}, ?i, \{j_1, \ldots, j_{m'}\}, (y_{h_1} : T_{h_1} \;;\; \ldots \;;\; y_{h_{m''}} : T_{h_{m''}} \vdash ?i' : T_{?i}))$
  that means that $(y_{h_1} : T_{h_1} \;;\; \ldots \;;\; y_{h_{m''}} : T_{h_{m''}} \vdash ?i' : T_{?i})$ is the (weakly) provable sequent in E and $\mathcal{P}$ obtained removing from the sequent associated to $?i$ in $\mathcal{P}$ all the hypotheses that are not in $\{j_1, \ldots, j_{m'}\}$ and all the other hypotheses that must be removed to obtain a weakly well-typed sequent.

- $\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [y_1 := t_1, \ldots, y_n := t_n], \Phi, t, \mathcal{P}', \Sigma', t')$
  that means that t' is a well-typed term in $\mathcal{P}'$ such that
  $$E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t'\{t_1/y_1 \;;\; \ldots \;;\; t_n/y_n\}) \downarrow \Sigma'(t)$$
  Moreover the only free variables in t' belong to the set $\Phi \cup \{y_1, \ldots, y_n\}$, Σ is a prefix of Σ' and $\mathcal{P}$ is a prefix of $\mathcal{P}'$.

- $E[\mathcal{P}; \Sigma; \Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}', \Sigma'$
  that means that $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t_1) \downarrow \Sigma'(t_2)$. Moreover Σ is a prefix of Σ' and $\mathcal{P}$ is a prefix of $\mathcal{P}'$.

- $E[\mathcal{P}; \Sigma; \Gamma] \vdash t_1 \Downarrow t_2 \bowtie \bot$
  Higher order unification is undecidable. This judgement states that the algorithm can decide nothing at all about the unification problem (t_1, t_2). In particular, it may be that there exists $\mathcal{P}'$ and Σ' such that $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t_1) \downarrow \Sigma'(t_2)$. On the contrary, if neither $E[\mathcal{P}; \Sigma; \Gamma] \vdash t_1 \Downarrow t_2 \bowtie \bot$ nor $E[\mathcal{P}; \Sigma; \Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}', \Sigma'$, then the t_1 and t_2 are definitely not unifiable.

The four judgements are formally defined by the following inference rules. The characterizing properties we have just given will be proved outside the definition.

restrict

$$\frac{\begin{array}{l}(y_1 : T_1 \;;\; \ldots \;;\; y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \\ V = \{h_1, \ldots, h_{m''}\} \text{ is the maximal subset of } \{j_1, \ldots, j_{m'}\} \text{ such that} \\ \quad FV(\Sigma(T_{h_1})) \cup \ldots \cup FV(\Sigma(T_{h_{m''}})) \cup FV(\Sigma(T_{?i})) \subseteq V\end{array}}{\mathsf{restrict}(\mathcal{P}, \Sigma, ?i, \{j_1, \ldots, j_{m'}\}, (y_{h_1} : T_{h_1} \;;\; \ldots \;;\; y_{h_{m''}} : T_{h_{m''}} \vdash ?i' : T_{?i}))}$$

Unif-Meta-Meta-Same-Meta

$$\frac{\begin{array}{c} ?i \notin \mathrm{Dom}(\Sigma) \qquad ?i' \notin \mathrm{Dom}(\mathcal{P}) \qquad \{h_1, \ldots, h_{n'}\} \subseteq \{1, \ldots, n\} \\ \mathrm{restrict}(\mathcal{P}_1, \Sigma_1, ?i, \{h_1, \ldots, h_{n'}\}, (y_{h_1} : T_{h_1} \ ; \ \ldots \ ; \ y_{h_{n'}} : T_{h_{n'}} \vdash ?i' : T_{?i})) \\ \mathcal{P}_2 = \mathcal{P}_1 \cup \{(y_{h_1} : T_{h_1} \ ; \ \ldots \ ; \ y_{h_{n'}} : T_{h_{n'}} \vdash ?i' : T_{?i})\} \qquad \Sigma_1 \oplus (?i := ?i'[y_{h_1} \ ; \ \ldots \ ; \ y_{h_{n'}}]) = \Sigma_2 \\ (E[\mathcal{P}_{i+1}; \Sigma_{i+1}; \Gamma] \vdash t_{h_i} \Downarrow t'_{h_i} \bowtie \mathcal{P}_{i+2}, \Sigma_{i+2})_{i=1,\ldots,n'} \end{array}}{E[\mathcal{P}_1; \Sigma_1; \Gamma] \vdash ?i[t_1 \ ; \ \ldots \ ; \ t_n] \Downarrow ?i[t'_1 \ ; \ \ldots \ ; \ t'_n] \bowtie \mathcal{P}_{n'+2}, \Sigma_{n'+2}}$$

Unif-Term-Meta-Swap

$$\frac{t \neq ?i[\sigma] \ \ \textit{for any } ?i \textit{ and } \sigma \qquad E[\mathcal{P}; \Sigma; \Gamma] \vdash ?j[\sigma] \Downarrow t \bowtie \mathcal{P}'; \Sigma'}{E[\mathcal{P}; \Sigma; \Gamma] \vdash t \Downarrow ?j[\sigma'] \bowtie \mathcal{P}'; \Sigma'}$$

Unif-Meta-Term-Hit

$$\frac{\begin{array}{c} ?i \in \mathrm{Dom}(\Sigma) \qquad \Sigma(?i) = t' \qquad (y_1 : T_1 \ ; \ \ldots \ ; \ y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \\ E[\mathcal{P}; \Sigma; \Gamma] \vdash t \Downarrow t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\} \bowtie \mathcal{P}'; \Sigma' \qquad E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash_\omega \Sigma'(t) : T \\ E[\mathcal{P}'; \Sigma'; \Gamma] \vdash T \Downarrow T_{?i}\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\} \bowtie \mathcal{P}''; \Sigma'' \end{array}}{E[\mathcal{P}; \Sigma; \Gamma] \vdash ?i[t_1, \ldots, t_n] \Downarrow t \bowtie \mathcal{P}''; \Sigma''}$$

Unif-Meta-Term-Miss

$$\frac{\begin{array}{c} ?i \notin \mathrm{Dom}(\Sigma) \qquad (y_1 : T_1 \ ; \ \ldots \ ; \ y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \\ \mathrm{abstract}(\mathcal{P}, \Sigma, \Gamma, [y_1 := t_1, \ldots, y_n := t_n], \emptyset, t, \mathcal{P}', \Sigma', t') \qquad E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma_{?i})] \vdash_\omega \Sigma'(t') : T \\ \Sigma' \oplus (?i := t') = \Sigma'' \qquad E[\mathcal{P}'; \Sigma''; \Gamma_{?i}] \vdash T \Downarrow T_{?i} \bowtie \mathcal{P}''; \Sigma''' \end{array}}{E[\mathcal{P}; \Sigma; \Gamma] \vdash ?i[t_1, \ldots, t_n] \Downarrow t \bowtie \mathcal{P}''; \Sigma'''}$$

Unif-Var-Var

$$\frac{\begin{array}{c} (E[\mathcal{P}_i; \Sigma_i; \Gamma] \vdash t_i \Downarrow t'_i \bowtie \mathcal{P}_{i+1}; \Sigma_{i+1})_{i=1,\ldots,n} \\ \textit{both explicit named substitutions are syntactically not identity substitutions} \end{array}}{E[\mathcal{P}_1; \Sigma_1; \Gamma] \vdash v[t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n] \Downarrow v[t'_1/v_1 \ ; \ \ldots \ ; \ t'_n/v_n] \bowtie \mathcal{P}'_{n+1}; \Sigma'_{n+1}}$$

Unif-Const-Const

$$\frac{(E[\mathcal{P}_i; \Sigma_i; \Gamma] \vdash t_i \Downarrow t'_i \bowtie \mathcal{P}_{i+1}; \Sigma_{i+1})_{i=1,\ldots,n}}{E[\mathcal{P}_1; \Sigma_1; \Gamma] \vdash c[t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n] \Downarrow c[t'_1/v_1 \ ; \ \ldots \ ; \ t'_n/v_n] \bowtie \mathcal{P}'_{n+1}; \Sigma'_{n+1}}$$

Unif-Constr-Constr

$$\frac{(E[\mathcal{P}_i; \Sigma_i; \Gamma] \vdash t_i \Downarrow t'_i \bowtie \mathcal{P}_{i+1}; \Sigma_{i+1})_{i=1,\ldots,n}}{E[\mathcal{P}_1; \Sigma_1; \Gamma] \vdash k[t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n] \Downarrow k[t'_1/v_1 \ ; \ \ldots \ ; \ t'_n/v_n] \bowtie \mathcal{P}'_{n+1}; \Sigma'_{n+1}}$$

Unif-Ind-Ind

$$\frac{(E[\mathcal{P}_i; \Sigma_i; \Gamma] \vdash t_i \Downarrow t'_i \bowtie \mathcal{P}_{i+1}; \Sigma_{i+1})_{i=1,\ldots,n}}{E[\mathcal{P}_1; \Sigma_1; \Gamma] \vdash i[t_1/v_1 \ ; \ \ldots \ ; \ t_n/v_n] \Downarrow i[t'_1/v_1 \ ; \ \ldots \ ; \ t'_n/v_n] \bowtie \mathcal{P}'_{n+1}; \Sigma'_{n+1}}$$

Unif-Prod-Prod

$$\frac{E[\mathcal{P}; \Sigma; \Gamma] \vdash T \Downarrow T' \bowtie \mathcal{P}'; \Sigma' \qquad E[\mathcal{P}'; \Sigma'; \Gamma, x : T] \vdash t \Downarrow t'\{x/x'\} \bowtie \mathcal{P}''; \Sigma''}{E[\mathcal{P}; \Sigma; \Gamma] \vdash \Pi x : T.t \Downarrow \Pi x' : T'.t' \bowtie \mathcal{P}''; \Sigma''}$$

Unif-Lambda-Lambda

$$\frac{E[\mathcal{P};\Sigma;\Gamma] \vdash T \Downarrow T' \bowtie \mathcal{P}';\Sigma' \qquad E[\mathcal{P}';\Sigma';\Gamma, x:T] \vdash t \Downarrow t'\{x/x'\} \bowtie \mathcal{P}'';\Sigma''}{E[\mathcal{P};\Sigma;\Gamma] \vdash \lambda x:T.t \Downarrow \lambda x':T'.t' \bowtie \mathcal{P}'';\Sigma''}$$

Unif-LetIn-LetIn[2]

$$\frac{E[\mathcal{P};\Sigma;\Gamma] \vdash t_2\{t_1/x\} \Downarrow t_2'\{t_x'/x'\} \bowtie \mathcal{P}';\Sigma'}{E[\mathcal{P};\Sigma;\Gamma] \vdash \mathtt{let}\ x := t_1\ \mathtt{in}\ t_2 \Downarrow \mathtt{let}\ x' := t_1'\ \mathtt{in}\ t_2' \bowtie \mathcal{P}';\Sigma'}$$

Unif-Appl-Appl-1

$$\frac{\begin{array}{c}(E[\mathcal{P}_i;\Sigma_i;\Gamma] \vdash t_i \Downarrow t_i' \bowtie \mathcal{P}_{i+1};\Sigma_{i+1})_{i=1,\dots,n-1}\\ E[\mathcal{P}_n;\Sigma_n;\Gamma] \vdash (t_m\ \dots\ t_n) \Downarrow t_n' \bowtie \mathcal{P}_{n+1};\Sigma_{n+1}\end{array}}{E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash (t_m\ \dots\ t_{n+1}\ t_n\ \dots\ t_1) \Downarrow (t_n'\ \dots\ t_1') \bowtie \mathcal{P}_{n+1};\Sigma_{n+1}}$$

Unif-Appl-Appl-2

$$\frac{\begin{array}{c}(E[\mathcal{P}_i;\Sigma_i;\Gamma] \vdash t_i' \Downarrow t_i \bowtie \mathcal{P}_{i+1};\Sigma_{i+1})_{i=1,\dots,n-1}\\ E[\mathcal{P}_n;\Sigma_n;\Gamma] \vdash t_n' \Downarrow (t_m\ \dots\ t_n) \bowtie \mathcal{P}_{n+1};\Sigma_{n+1}\end{array}}{E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash (t_n'\ \dots\ t_1') \Downarrow (t_m\ \dots\ t_{n+1}\ t_n\ \dots\ t_1) \bowtie \mathcal{P}_{n+1};\Sigma_{n+1}}$$

Unif-Cast-Cast

$$\frac{E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}';\Sigma' \qquad E[\mathcal{P}';\Sigma';\Gamma] \vdash T_1 \Downarrow T_2 \bowtie \mathcal{P}'';\Sigma''}{E[\mathcal{P};\Sigma;\Gamma] \vdash (t_1 :: T_1) \Downarrow (t_2 :: T_2) \bowtie \mathcal{P}'';\Sigma''}$$

Unif-Case-Case[3]

$$\frac{\begin{array}{c}E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash t \Downarrow t' \bowtie \mathcal{P}_2;\Sigma_2 \qquad E[\mathcal{P}_2;\Sigma_2;\Gamma] \vdash T \Downarrow T' \bowtie \mathcal{P}_3;\Sigma_3\\ (E[\mathcal{P}_{i+2};\Sigma_{i+2};\Gamma;\overline{x^i:T^i}] \vdash f_i \Downarrow f_i'\{\overline{x^i/x'^i}\} \bowtie \mathcal{P}_{i+3};\Sigma_{i+3})_{i=1,\dots,n_l}\end{array}}{\begin{array}{ll}E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash & \left(\begin{array}{l}<T>\ \mathrm{CASES}_{i_l}\ t\ \mathrm{OF}\\ \quad k_1^l\ \overline{x^1} \Rightarrow f_1 \mid \dots \mid k_{n_l}^l\ \overline{x^n} \Rightarrow f_{n_l}\\ \mathrm{END}\end{array}\right) \Downarrow\\ & \left(\begin{array}{l}<T'>\ \mathrm{CASES}_{i_l}\ t'\ \mathrm{OF}\\ \quad k_1^l\ \overline{x'^1} \Rightarrow f_1' \mid \dots \mid k_{n_l}^l\ \overline{x'^n} \Rightarrow f_{n_l}'\\ \mathrm{END}\end{array}\right) \bowtie \mathcal{P}_{n_l+3};\Sigma_{n_l+3}\end{array}}$$

Unif-Fix-Fix

$$\frac{\begin{array}{c}(E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash T_i \Downarrow T_i' \bowtie \mathcal{P}_2;\Sigma_2)_{i=1,\dots,m}\\ (E[\mathcal{P}_{i+1};\Sigma_{i+1};\Gamma\ ;\ x_1:T_1\ ;\ \dots\ ;\ x_m:T_m] \vdash f_i \Downarrow f_i'\{x_1/x_1'\ ;\ \dots\ ;\ x_m/x_m'\} \bowtie \mathcal{P}_{i+2};\Sigma_{i+2})_{i=1,\dots,m}\end{array}}{\begin{array}{ll}E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash & \mathrm{FIX}_j\ \{x_1/n_1:T_1 := f_1\ ;\ \dots\ ;\ x_m/n_m:T_m := f_m\} \Downarrow\\ & \mathrm{FIX}_j\ \{x_1'/n_1:T_1' := f_1'\ ;\ \dots\ ;\ x_m'/n_m:T_m' := f_m'\} \bowtie \mathcal{P}_{m+2};\Sigma_{m+2}\end{array}}$$

[2]In the rules of the conversion test $\downarrow$ we pushed the term in the context. Here we prefer this heuristic that performs one step of ζ-reduction to maximize the possibility of success of first order unification (that does not perform reduction).

[3]The type T^i of the x^i constructor can either be a fake type (that is never used in reduction and conversion) or it can be the real type of x^i that is obtained by browsing the definition of i_l.

Unif-CoFix-CoFix

$$\frac{\begin{array}{c}(E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash T_i \Downarrow T'_i \bowtie \mathcal{P}_2;\Sigma_2)_{i=1,\dots,m} \\ (E[\mathcal{P}_{i+1};\Sigma_{i+1};\Gamma\ ;\ x_1 : T_1\ ;\ \dots\ ;\ x_m : T_m] \vdash f_i \Downarrow f'_i\{x_1/x'_1\ ;\ \dots\ ;\ x_m/x'_m\} \bowtie \mathcal{P}_{i+2};\Sigma_{i+2})_{i=1,\dots,m}\end{array}}{\begin{array}{rl}E[\mathcal{P}_1;\Sigma_1;\Gamma] \vdash & \mathrm{COFIX}_j\ \{x_1 : T_1 := f_1\ ;\ \dots\ ;\ x_m : T_m := f_m\} \Downarrow \\ & \mathrm{COFIX}_j\ \{x'_1 : T'_1 := f'_1\ ;\ \dots\ ;\ x'_m : T'_m := f'_m\} \bowtie \mathcal{P}_{m+2};\Sigma_{m+2}\end{array}}$$

Unif-Conv-Conv

$$\frac{E[\Sigma(\mathcal{P});\Sigma(\Gamma)] \vdash \Sigma(t_1) \downarrow \Sigma(t_2)}{E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P};\Sigma}$$

Unif-maybe

$$\frac{\begin{array}{c}t_1 \textit{ or } t_2 \textit{ not stiff} \\ E[\Sigma(\mathcal{P});\Sigma(\Gamma)] \not\vdash \Sigma(t_1) \downarrow \Sigma(t_2) \qquad \mathcal{M}(t_1) \cup \mathcal{M}(t_2) \neq \emptyset\end{array}}{E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \bot}$$

Unif-maybe-propagation-rules

> *For each one of the previous inference rule, if a premise is of the form* $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}';\Sigma'$*, then there exists a companion rule whose premise is* $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \bot$ *and whose conclusion is also of the form* $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \bot$.

abstract-hit

$$\frac{}{\mathrm{abstract}(\mathcal{P},\Sigma,\Gamma,[y_1 := t_1,\dots,y_n := t_n],\emptyset,t_i,\mathcal{P},\Sigma,y_i)}$$

abstract-meta-not-instantiated

$$\frac{\begin{array}{l}?i \notin \mathrm{Dom}(\Sigma_1) \\ \quad \textit{let}\ \{j_1,\dots,j_{n'}\}\ \textit{be the maximal subset of}\ \{1,\dots,n\}\ \textit{such that} \\ \qquad (\mathrm{abstract}(\mathcal{P}_l,\Sigma_l,\Gamma,[\rho],\Phi,t_{j_l},\mathcal{P}_{l+1},\Sigma_{l+1},t'_{j_l}))_{l=1,\dots,n'} \\ \mathrm{restrict}(\mathcal{P}_{n'+1},\Sigma_{n'+1},?i,\{j_1,\dots,j_{n'}\},(y_{h_1} : T_{h_1}\ ;\ \dots\ ;\ y_{h_{n''}} : T_{h_{n''}} \vdash ?i' : T_{?i})) \\ \Sigma_{n'+1} \oplus (?i := ?i'[y_{h_1}\ ;\ \dots\ ;\ y_{h_{n''}}]) = \Sigma_{n'+2}\end{array}}{\begin{array}{l}\mathrm{abstract}(\mathcal{P}_1,\Sigma_1,\Gamma,[\rho],\Phi,?i[t_1,\dots,t_n], \\ \qquad \mathcal{P}_{n'+1} \cup \{y_{h_1} : T_{h_1}\ ;\ \dots\ ;\ y_{h_{n''}} : T_{h_{n''}} \vdash ?i' : T_{?i}\}, \\ \qquad \Sigma_{n'+2}, ?i'[t'_{h_1},\dots,t'_{h_{n''}}])\end{array}}$$

abstract-meta-instantiated

$$\frac{?i \in \mathrm{Dom}(\Sigma) \qquad \mathrm{abstract}(\mathcal{P},\Sigma,\Gamma,[\rho],\emptyset,\Sigma(?i[\rho']),\mathcal{P}',\Sigma',t')}{\mathrm{abstract}(\mathcal{P},\Sigma,\Gamma,[\rho],\Phi,?i[\rho'],\mathcal{P}',\Sigma',t')}$$

abstract-bound-rel

$$\frac{x \in \Phi}{\mathrm{abstract}(\mathcal{P},\Sigma,\Gamma,[\rho],\Phi,x,\mathcal{P},\Sigma,x)}$$

abstract-free-rel-to-def

$$\frac{\Gamma(x) = (x := t) \qquad \mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \emptyset, t, \mathcal{P}', \Sigma', t')}{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, x, \mathcal{P}', \Sigma', t')}$$

abstract-var

$$\frac{(\mathsf{abstract}(\mathcal{P}_i, \Sigma_i, \Gamma, [\rho], \Phi, M_i, \mathcal{P}_{i+1}, \Sigma_{i+1}, M'_i))_{i=1,\ldots,m}}{\mathsf{abstract}(\mathcal{P}_1, \Sigma_1, \Gamma, [\rho], \Phi, v[M_1/v_1\ ;\ \ldots\ ;\ M_m/v_m], \mathcal{P}_{m+1}, \Sigma_{m+1}, v[M'_1/v_1\ ;\ \ldots\ ;\ M'_m/v_m])}$$

abstract-const

$$\frac{(\mathsf{abstract}(\mathcal{P}_i, \Sigma_i, \gamma, [\rho], \Phi, m_i, \mathcal{P}_{i+1}, \Sigma_{i+1}, m'_i))_{i=1,\ldots,m}}{\mathsf{abstract}(\mathcal{P}_1, \Sigma_1, \gamma, [\rho], \Phi, c[m_1/v_1\ ;\ \ldots\ ;\ m_m/v_m], \mathcal{P}_{m+1}, \Sigma_{m+1}, c[m'_1/v_1\ ;\ \ldots\ ;\ m'_m/v_m])}$$

abstract-constr

$$\frac{(\mathsf{abstract}(\mathcal{P}_i, \Sigma_i, \gamma, [\rho], \Phi, m_i, \mathcal{P}_{i+1}, \Sigma_{i+1}, m'_i))_{i=1,\ldots,m}}{\mathsf{abstract}(\mathcal{P}_1, \Sigma_1, \gamma, [\rho], \Phi, k[m_1/v_1\ ;\ \ldots\ ;\ m_m/v_m], \mathcal{P}_{m+1}, \Sigma_{m+1}, c[m'_1/v_1\ ;\ \ldots\ ;\ m'_m/v_m])}$$

abstract-ind

$$\frac{(\mathsf{abstract}(\mathcal{P}_i, \Sigma_i, \gamma, [\rho], \Phi, m_i, \mathcal{P}_{i+1}, \Sigma_{i+1}, m'_i))_{i=1,\ldots,m}}{\mathsf{abstract}(\mathcal{P}_1, \Sigma_1, \gamma, [\rho], \Phi, i[m_1/v_1\ ;\ \ldots\ ;\ m_m/v_m], \mathcal{P}_{m+1}, \Sigma_{m+1}, c[m'_1/v_1\ ;\ \ldots\ ;\ m'_m/v_m])}$$

abstract-sort

$$\frac{}{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, s, \mathcal{P}, \Sigma, s)}$$

abstract-cast

$$\frac{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, t, \mathcal{P}', \Sigma', t') \qquad \mathsf{abstract}(\mathcal{P}', \Sigma', \Gamma, [\rho], \Phi, T, \mathcal{P}'', \Sigma'', T')}{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, (t :: T), \mathcal{P}'', \Sigma'', (t' :: T'))}$$

abstract-prod

$$\frac{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, T, \mathcal{P}', \Sigma', T') \qquad \mathsf{abstract}(\mathcal{P}', \Sigma', \Gamma, [\rho], \Phi \cup \{x\}, t, \mathcal{P}'', \Sigma'', t')}{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, \Pi x : T.t, \mathcal{P}'', \Sigma'', \Pi x : T'.t')}$$

abstract-lambda

$$\frac{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, T, \mathcal{P}', \Sigma', T') \qquad \mathsf{abstract}(\mathcal{P}', \Sigma', \Gamma, [\rho], \Phi \cup \{x\}, t, \mathcal{P}'', \Sigma'', t')}{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, \lambda x : T.t, \mathcal{P}'', \Sigma'', \lambda x : T'.t')}$$

abstract-letin

$$\frac{\begin{array}{c}\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, t_1, \mathcal{P}', \Sigma', t'_1) \\ \mathsf{abstract}(\mathcal{P}', \Sigma', \Gamma, [\rho], \Phi \cup \{x\}, t_2, \mathcal{P}'', \Sigma'', t'_2)\end{array}}{\mathsf{abstract}(\mathcal{P}, \Sigma, \Gamma, [\rho], \Phi, \mathsf{let}\ x := t_1\ \mathsf{in}\ t_2, \mathcal{P}'', \Sigma'', \mathsf{let}\ x := t'_1\ \mathsf{in}\ t'_2)}$$

abstract-appl

$$\frac{(\mathsf{abstract}(\mathcal{P}_i, \Sigma_i, \Gamma, [\rho], \Phi, t_i, \mathcal{P}_{i+1}, \Sigma_{i+1}, t'_i))_{i=1,\ldots,n}}{\mathsf{abstract}(\mathcal{P}_1, \Sigma_1, \Gamma, [\rho], \Phi, (t_1\ \ldots\ t_n), \mathcal{P}_{n+1}, \Sigma_{n+1}, (t'_1\ \ldots\ t'_n))}$$

abstract-mutcase

$$\frac{\begin{array}{c}\mathtt{abstract}(\mathcal{P}_1,\Sigma_1,\Gamma,[\rho],\Phi,T,\mathcal{P}_2,\Sigma_2,T') \qquad \mathtt{abstract}(\mathcal{P}_2,\Sigma_2,\Gamma,[\rho],\Phi,t,\mathcal{P}_3,\Sigma_3,t') \\ (\mathtt{abstract}(\mathcal{P}_{2+i},\Sigma_{2+i},\Gamma,[\rho],\Phi\cup\overline{x^i},f_i,\mathcal{P}_{3+i},\Sigma_{3+i},f'_i))_{i=1,\ldots,n_l}\end{array}}{\begin{array}{r}\mathtt{abstract}(\mathcal{P}_1,\Sigma_1,\Gamma,[\rho],\Phi,\left(\begin{array}{l}<T>\mathrm{CASES}_{i_l}\ t\ \mathrm{OF}\\ \quad k^l_1\ \overline{x^1}\Rightarrow f_1\mid\ldots\mid k^l_{n_l}\ \overline{x^n}\Rightarrow f_{n_l}\\ \mathrm{END}\end{array}\right), \\ \mathcal{P}_{3+n_l},\Sigma_{3+n_l},\left(\begin{array}{l}<T'>\mathrm{CASES}_{i_l}\ t'\ \mathrm{OF}\\ \quad k^l_1\ \overline{x^1}\Rightarrow f'_1\mid\ldots\mid k^l_{n_l}\ \overline{x^n}\Rightarrow f'_{n_l}\\ \mathrm{END}\end{array}\right))\end{array}}$$

abstract-fix

$$\frac{\begin{array}{c}(\mathtt{abstract}(\mathcal{P}_i,\Sigma_i,\Gamma,[\rho],\Phi,T_i,\mathcal{P}_{i+1},\Sigma_{i+1},T'_i))_{i=1,\ldots,m}\\ (\mathtt{abstract}(\mathcal{P}_{m+i},\Sigma_{m+i},\Gamma,[\rho],\Phi\cup\{x_1,\ldots,x_m\},f_i,\mathcal{P}_{m+i+1},\Sigma_{m+i+1},f'_i))_{i=1,\ldots,m}\end{array}}{\begin{array}{r}\mathtt{abstract}(\mathcal{P}_1,\Sigma_1,\Gamma,[\rho],\Phi,\mathrm{FIX}_j\ \{x_1/n_1:T_1:=f_1\ ;\ \ldots\ ;\ x_m/n_m:T_m:=f_m\},\\ \mathcal{P}_{2m+1},\Sigma_{2m+1},\mathrm{FIX}_j\ \{x_1/n_1:T'_1:=f'_1\ ;\ \ldots\ ;\ x_m/n_m:T'_m:=f'_m\})\end{array}}$$

abstract-cofix

$$\frac{\begin{array}{c}(\mathtt{abstract}(\mathcal{P}_i,\Sigma_i,\Gamma,[\rho],T_i,\mathcal{P}_{i+1},\Sigma_{i+1},T'_i))_{i=1,\ldots,m}\\ (\mathtt{abstract}(\mathcal{P}_{m+i},\Sigma_{m+i},\Gamma,[\rho],f_i,\mathcal{P}_{m+i+1},\Sigma_{m+i+1},f'_i))_{i=1,\ldots,m}\end{array}}{\begin{array}{r}\mathtt{abstract}(\mathcal{P}_1,\Sigma_1,\Gamma,[\rho],\mathrm{COFIX}_j\ \{x_1:T_1:=f_1\ ;\ \ldots\ ;\ x_m:T_m:=f_m\},\\ \mathcal{P}_{2m+1},\Sigma_{2m+1},\mathrm{COFIX}_j\ \{x_1:T'_1:=f'_1\ ;\ \ldots\ ;\ x_m:T'_m:=f'_m\})\end{array}}$$

The judgements proposed are not syntax oriented. Thus, to obtain an algorithm, we must either fix a strategy or implement backtracking. Unfortunately, many of the critical pairs (i.e. of the couple of judgements that can be applied when a term is considered) cannot be avoided without sacrificing completeness even more.

In GTOPLEVEL we have experimented with an algorithm that is obtained from the judgement above by fixing the following strategy. The algorithm is not complete w.r.t. the judgement. In particular, it is not even complete w.r.t. α-conversion. In practice, though, our (few) users do not strongly feel the need for a better strategy.

1. Never backtrack: if the conclusion of a rule matches the current goal, then apply that rule; otherwise look at the next rule. As a consequence, we can optimize the overall algorithm by collecting all the instantiations without using the $\oplus$ extension judgement. At the end of the algorithm the obtained metavariable substitution environment is checked for validity. In case of an invalid metavariable substitution environment, we return the unknown ($\perp$) answer.

2. Try to apply any other Unif-$*$ rule before the rules Unif-Conv-Conv and Univ-Maybe (that are mutually exclusive and whose conclusion matches any proof problem).

3. Try to apply the abstract-hit rule before any other abstract-$*$ rule.

4. Another source of backtracking and non determinism derives from the premises of the rule Unif-Meta-Meta-Same-Meta. Indeed, operationally there is no way to guess a suitable set $S = \{h_1, \ldots, h_{n'}\}$ that satisfies all the other premises without backtracking. Moreover, the set S is not even unique.

   The strategy proposed is to choose $S = \{1, \ldots, n\}$ and to return $\perp$ if the last premise cannot be inhabited (the second one is always inhabited and can be skipped, choosing $\mathcal{P}_2 = \mathcal{P}_1$ and $\Sigma_2 = \Sigma_1$).

The algorithm is quite efficient, but it is much less complete than the judgement proposed above. In practice, the algorithm is often reasonable. We should notice, though, that it seems that the greatest majority of the unification problems met in interactive theorem proving applications are quite trivial, especially if dependent types are not heavily used.

We are now going to present several lemmas that lead to the soundness proof of our unification judgment.

Lemma 10.40 (Characterization of `restrict`**)** *Let* E *be a well-formed environment,* Σ *a metavariable substitution environment and* $\mathcal{P}$ *a proof problem in* E *such that* $\Sigma(\mathcal{P})$ *is well-formed and* Σ *is a well-typed metavariable substitution environment for* $\Sigma(\mathcal{P})$*. Let* $?i$ *a metavariable in the domain of* $\mathcal{P}$ *such that* $(y_1 : T_1 \ ; \ \ldots \ ; \ y_m : T_m \vdash ?i : T_{?i}) \in \mathcal{P}$ *and* $\{j_1, \ldots, j_{m'}\}$ *a subset of the set of indexes* $\{1, \ldots, m\}$*.*

If there exists a derivation of

$$\mathtt{restrict}(\mathcal{P}, \Sigma, ?i, \{j_1, \ldots, j_{m'}\}, (y_{h_1} : T_{h_1} \ ; \ \ldots \ ; \ y_{h_{m''}} : T_{h_{m''}} \vdash ?i' : T_{?i}))$$

then

1. $E[\Sigma(\mathcal{P}), y_{h_1} : \Sigma(T_{h_1}) \ ; \ \ldots \ ; \ y_{h_{m''}} : \Sigma(T_{h_{m''}})] \vdash_\omega \Sigma(T_{?i}) : s$

2. $\{j_{h_1}, \ldots, j_{h_{m''}}\}$ *is the biggest subset of* $\{j_1, \ldots, j_{m'}\}$ *such that the previous property holds.*

Proof: Since $\Sigma(\mathcal{P})$ is by hypothesis well-formed, we know that $E[\Sigma(\mathcal{P}), y_1 : \Sigma(T_1) \ ; \ \ldots \ ; \ y_m : \Sigma(T_m)] \vdash_\omega \Sigma(T_{?i}) : s$. By inspection of the restrict derivation rules, we know that $V = \{h_1, \ldots, h_{m''}\}$ is the maximal subset of $\{j_1, \ldots, j_{m'}\}$ such that $FV(\Sigma(T_{h_1})) \cup \ldots \cup FV(\Sigma(T_{h_{m''}})) \cup FV(\Sigma(T_{?i})) \subseteq V$. By applying several times the thinning lemma of CIC extended with explicit named substitutions and metavariables, we conclude the first part of the thesis.

For the second part, we suppose by absurdum that there exists an index $j \in \{j_1, \ldots, j_{m'}\} \setminus \{j_{h_1}, \ldots, j_{h_{m''}}\}$ such that the first part of the thesis holds when we consider the set of indexes $\{j_{h_1}, \ldots, j_{h_{m''}}\} \cup \{j\}$. By definition of V, we know that there exists an unbound variable in $\Sigma(T_j)$.

Thus, by inspection of the inference rules of $\vdash_\omega$, we conclude that the sequent cannot be well-formed, reaching a contradiction. Hence the thesis. $\square$

Lemma 10.41 (Soundness of the $\mathtt{abstract}$ judgement) *Let E be a well-formed environment, $\mathcal{P}$ a proof problem in E and Σ a metavariable substitution environment such that $\Sigma(\mathcal{P})$ is a well-formed proof problem and Σ is a well-typed metavariable substitution environment for $\Sigma(\mathcal{P})$. Let $\Sigma(\Gamma)$ be a well-formed context in E and $\Sigma(\mathcal{P})$, t_i be a well-typed term in E, $\Sigma(\mathcal{P})$ and $\Sigma(\Gamma)$ for each i and finally let Φ be a set of bound variables.*

If there exists a derivation of

$$\mathtt{abstract}(\mathcal{P}, \Sigma, \Gamma, [y_1 := t_1, \ldots, y_n := t_n], \Phi, t, \mathcal{P}', \Sigma', t')$$

then

1. $\Sigma'(\mathcal{P}')$ *is a well-formed proof problem in E that extends $\Sigma(\mathcal{P})$*

2. Σ' *is a well-typed metavariable substitution environment that extends Σ*

3. $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t'\{t_1/y_1 \; ; \; \ldots \; ; \; t_n/y_n\}) \downarrow \Sigma'(t)$

4. *let t'' be the "ζ-normal form" of t' (where the ζ-normal form is computed only expanding the definitions in the context Γ, whereas the local definitions are not unfolded); the only free variables in t'' belong to the set $\Phi \cup \{y_1, \ldots, y_n\}$*

Proof: The proof is by induction over the shape of the "ζ-normal form" of t and by inspection of the rules that can be applied. The ζ-normal form of t is computed only expanding the definitions in the context Γ, whereas the local definitions are not unfolded.

The rule abstract-hit can be applied in all the cases as soon as $t = t_i$ for some t_i. If abstract-hit is applied, then the thesis trivially holds.

Otherwise, if the abstract-hit rule is not applied, then the rule applied must be the only rule that handles the case that corresponds to the shape of t. Let us analyze independently the only three interesting rules:

- abstract-meta-non-instantiated:
  Let $t = ?i[t'_1 \; ; \; \ldots \; ; \; t'_m]$ and $\mathcal{P}' = \mathcal{P}_{m'+1} \cup \{y'_{h_1} : T_{h_1} \; ; \; \ldots \; ; \; y_{h_{m''}} : T_{h_{m''}} \vdash ?i' : T_{?i}\}$ and $\Sigma' = \Sigma_{m'+2}$ and $t' = ?i'[t''_{h_1}, \ldots, t''_{h_{m''}}]$ such that

$$(\mathtt{abstract}(\mathcal{P}_l, \Sigma_l, \Gamma, [\rho], \Phi, t'_{j_l}, \mathcal{P}_{l+1}, \Sigma_{l+1}, t''_{j_l}))_{l=1,\ldots,m'} \quad (10.42)$$

$$\mathtt{restrict}(\mathcal{P}_{m'+1}, \Sigma_{m'+1}, ?i, \{j_1, \ldots, j_{m'}\}, (y'_{h_1} : T_{h_1} \; ; \; \ldots \; ; \; y'_{h_{m''}} : T_{h_{m''}} \vdash ?i' : T_{?i})) \quad (10.43)$$

$$\Sigma_{m'+1} \oplus (?i := ?i'[y'_{h_1} \; ; \; \ldots \; ; \; y'_{h_{m''}}]) = \Sigma_{m'+2} \quad (10.44)$$

where $\mathcal{P}_1 = \mathcal{P}$ and $\Sigma_1 = \Sigma$. By the inductive hypothesis over 10.42 we know that $\Sigma_{m'+1}(\mathcal{P}_{m'+1})$ is a well-formed extension of $\Sigma(\mathcal{P})$, $\Sigma_{m'+1}$ is a well-typed extension of Σ and

$$\forall l \in \{1, \ldots, m'\}.\ E[\Sigma_{m'+1}(\mathcal{P}_{m'+1}); \Sigma_{m'+1}(\Gamma)] \vdash \Sigma_{m'+1}(t''_{j_l}\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\}) \downarrow \Sigma_{m'+1}t'_{j_l}$$

By lemma 10.40, we know that

$$E[\Sigma_{m'+1}(\mathcal{P}_{m'+1}), y'_{h_1} : \Sigma_{m'+2}(T_{h_1}) \ ; \ \ldots \ ; \ y'_{h_{m''}} : \Sigma_{m'+2}(T_{h_{m''}})] \vdash \Sigma_{m'+2}(T_{?i}) : s \quad (10.45)$$

By inductive hypothesis over 10.42 we also know that $\Sigma_{m'+1}(\mathcal{P}_{m'+1})$ is a well-formed extension of $\Sigma(\mathcal{P})$. Moreover, because of 10.45, to prove that $\Sigma_{m'+1}(\mathcal{P}') = \Sigma_{m'+1}(\mathcal{P}_{m'+1} \cup \{y'_{h_1} : T_{h_1} \ ; \ \ldots \ ; \ y'_{h_{m''}} : T_{h_{m''}} \ \vdash \ ?i' \ : \ T_{?i}\})$ is also well-formed we just have to prove that $\ll_{\Sigma_{m'+1}(\mathcal{P}')}$ is irreflexive. The proof is trivial since $?i'$ does not occur in $\mathcal{M}(\Gamma_{?j}) \cup \mathcal{M}(T_{?j})$ for each $?j \in \mathrm{Dom}(\Sigma_{m'+1}(\mathcal{P}'))$.

By definition of $\Sigma_{m'+2}$, we easily conclude that $\Sigma' = \Sigma_{m'+2}$ is also a well-typed extension of Σ. In particular, $\Sigma'(?i)$ (that is defined as $?i'[y'_{h_1}, \ldots, y'_{h_{m''}}]$) has type $T_{?i}$ in the context $\Gamma_{?i}$ by construction. Moreover, checking the irreflexivity of $\ll_{\Sigma_{m'+2}(\mathcal{P}')}$ is also trivial, since $\ll_{\Sigma_{m'+1}(\mathcal{P}')}$ is irreflexive and $?i'$ does not occur in $\mathcal{M}(\Gamma_{?j}) \cup \mathcal{M}(T_{?j})$ for each $?j \in \mathrm{Dom}(\Sigma_{m'+1}(\mathcal{P}'))$.

We still have to prove the third and fourth parts of the thesis. The fourth part is a trivial application of the induction hypothesis. For the third part, we just have to observe that $\Sigma'(t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\}) = ?i'[\Sigma'(t''_{h_1}) \ ; \ \ldots \ ; \ \Sigma'(t''_{h_{m''}})]\{\Sigma'(t_1)/y_1 \ ; \ \ldots \ ; \ \Sigma'(t_n)/y_n\}$ and that $\Sigma'(t\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\}) = \Sigma'(?i[t'_1 \ ; \ \ldots \ ; \ t'_m]) = ?i'[\Sigma'(t'_{h_1}) \ ; \ \ldots \ ; \ \Sigma'(t'_{h_{m''}})]$ Thus, to construct a derivation of $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\}) \downarrow \Sigma'(t)$, we can just apply the rules Conv-Meta-Meta and inhabit its premise with the application of lemma 10.31 to the inductive hypothesis over 10.42.

- abstract-meta-instantiated:
  The thesis trivially holds since Σ' is an extension of Σ.

- abstract-bound-rel:
  The thesis trivially holds since $\mathcal{P}' = \mathcal{P}$, $\Sigma' = \Sigma$, $x \in \Phi$ and $t' = x$.

- abstract-free-rel-to-def:
  The rule will never be applied, since $t = x$ and $\Gamma(x) = (x := t')$ is an impossible case since x is a ζ-redex and t is in ζ-normal form by hypothesis.

All the other cases are just trivial applications of the inductive hypothesis. Since the cases are all very similar, we just show one case:

- abstract-lambda:
  Since $\lambda x : T.t$ is in ζ-normal form by hypothesis, also T and t are in ζ-normal form. By induction hypothesis, $\Sigma'(\mathcal{P}')$ is a well-formed proof problem that extends $\Sigma(\mathcal{P})$ and $\Sigma''(\mathcal{P}'')$ is a well-formed proof problem that extends $\Sigma'(\mathcal{P}')$ (and thus also $\Sigma(\mathcal{P})$). Moreover, Σ' is well-typed and it extends Σ and Σ'' is well-typed and it extends Σ' (and thus also Σ). A free variable of $\lambda x : T.t$ must be either a free variable of T or a free variable of t that is not x. By induction hypotheses, we know that every free variable of T belongs to $\Phi \cup \{y_1, \ldots, y_n\}$ and any free variable of t belongs to $\Phi \cup \{x\} \cup \{y_1, \ldots, y_n\}$. Thus any free variable of $\lambda x : T.t$ belongs to the set $\Phi \cup \{y_1, \ldots, y_n\}$. Finally, by induction hypothesis and lemma 10.31, we know that $E[\Sigma''(\mathcal{P}''); \Sigma''(\Gamma)] \vdash \Sigma''(T'\{t_1/y_1 \;;\; \ldots \;;\; t_n/y_n\}) \downarrow \Sigma''(T)$ and $E[\Sigma''(\mathcal{P}''); \Sigma''(\Gamma)] \vdash \Sigma''(t'\{t_1/y_1 \;;\; \ldots \;;\; t_n/y_n\}) \downarrow \Sigma''(t)$. Thus, by applying the rule Conv-Lambda-Lambda, we can build a derivation for $E[\Sigma''(\mathcal{P}''); \Sigma''(\Gamma)] \vdash \Sigma''((\lambda x : T'.t')\{t_1/y_1 \;;\; \ldots \;;\; t_n/y_n\}) \downarrow \Sigma''(\lambda x : T.t)$.

$\square$

Theorem 10.46 (Soundness of the unification judgment, part I) *Let E be a well-formed environment, $\mathcal{P}$ a proof problem and Σ a metavariable substitution environment such that $\Sigma(\mathcal{P})$ is well-formed and Σ is well-typed in $\Sigma(\mathcal{P})$. Let $\Sigma(\Gamma)$ be a well-formed context in E and $\Sigma(\mathcal{P})$, $(\Sigma(t_1), \Sigma(t_2))$ be an unification problem in E, $\Sigma(\mathcal{P})$ and $\Sigma(\Gamma)$ such that both t_1 and t_2 are well-typed.*

If there exists a derivation of $E[\mathcal{P}; \Sigma; \Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}', \Sigma'$ than

1. *$\Sigma'(\mathcal{P}')$ is a well-formed extension of $\Sigma(\mathcal{P})$*
2. *Σ' is a well-typed extension of Σ in $\Sigma'(\mathcal{P}')$*
3. *$E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t_1) \downarrow \Sigma'(t_2)$*
4. *$E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(T_1) \downarrow \Sigma'(T_2)$ where T_i is the type of t_i in E, $\Sigma'(\mathcal{P}')$ and $\Sigma'(\Gamma)$*

Proof: The proof is by structural induction over the derivation of $E[\mathcal{P}; \Sigma; \Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}', \Sigma'$. We present the cases that deal with metavariables first.

- *Case Unif-Meta-Meta-Same-Meta*:
  By lemma 10.40, we know that $E[\Sigma_1(\mathcal{P}_1), y_{h_1} : \Sigma_1(T_{h_1}) \;;\; \ldots \;;\; y_{h_{n'}} : \Sigma_1(T_{h_{n'}})] \vdash \Sigma_1(T_{?i}) : s$. By hypothesis, $\mathcal{P}_2$ is defined as $\mathcal{P}_1 \cup \{(y_{h_1} : T_{h_1} \;;\; \ldots \;;\; y_{h_{n'}} : T_{h_{n'}} \vdash ?i' : T_{?i})$ where $?i' \notin \mathcal{P}$. Thus $\Sigma_1(\mathcal{P}_2)$ is a well-formed proof problem, i.e. the sequent associated to $?i'$ is well-formed and the order $\ll_{\Sigma_1(\mathcal{P}_2)}$ is irreflexive. By lemma 10.34, if we prove that Σ_2 is well-typed in $\Sigma_1(\mathcal{P}_2)$ we conclude also that Σ_2 is well typed in $\Sigma_2(\mathcal{P}_2)$. Since Σ_2 satisfies the judgement $\Sigma_1 \oplus \{?i := ?i'[y_{h_1} \;;\; \ldots \;;\; y_{h_{n'}}]\} = \Sigma_2$, since $?i' \notin \mathcal{M}(\Gamma_{?j}) \cup \mathcal{M}(T_{?j})$ for each $?j \in \mathrm{Dom}(\Sigma_1(\mathcal{P}_2))$

and since the type of $\Sigma_2(?i) = ?i'[y_{h_1}\ ;\ \ldots\ ;\ y_{h_{n'}}]$ is equal to the type of $\Sigma_2(?i[\rho])$ in the context $\Sigma_2(\Gamma_{?i})$ by construction, we easily conclude the well-typedness of Σ_2.

To proceed in the proof, we have to consider the following inductive hypotheses that corresponds to the last premise of the rule:

$$(E[\Sigma_{i+2}(\mathcal{P}_{i+2}); \Sigma_{i+2}(\Gamma)] \vdash \Sigma_{i+2}(t_{h_i}) \downarrow \Sigma_{i+2}(t'_{h_i}))_i = 1, \ldots, n' \tag{10.47}$$

$$\Sigma_{i+2}(\mathcal{P}_{i+2}) \text{ is a well-formed extension of } \Sigma_{i+1}(\mathcal{P}_{i+1}) \text{ for each } i \tag{10.48}$$

$$\Sigma_{i+2} \text{ is a well-typed extension of } \Sigma_{i+1} \text{ in } \Sigma_{i+2}(\mathcal{P}_{i+2}) \tag{10.49}$$

We can conclude the second and third parts of the thesis by applying n' times lemma 10.34 to 10.47 and 10.48. By applying lemma 10.31 to the previous result and 10.49, we also conclude:

$$(E[\Sigma_{n'+2}(\mathcal{P}_{n'+2}); \Sigma_{n'+2}(\Gamma)] \vdash \Sigma_{n'+2}(t_{h_i}) \downarrow \Sigma_{n'+2}(t'_{h_i}))_i = 1, \ldots, n' \tag{10.50}$$

Thus we can prove the third part of the thesis by constructing a derivation applying the rule Conv-Meta-Meta to 10.50. The fourth part of the thesis is a consequence of the previous result and the typing rules for metavariable occurrences.

- *Case Unif-Meta-Term-Miss*:
  By hypothesis, we know that $?i \in \mathrm{Dom}(\Sigma)$ and that $\mathrm{abstract}(\mathcal{P}, \Sigma, \Gamma, [y_1 := t_1, \ldots, y_n := t_n], \emptyset, t, \mathcal{P}', \Sigma', t')$. Applying lemma 10.41 we obtain

  1. $\Sigma'(\mathcal{P}')$ is a well-formed proof problem in E that extends $\Sigma(\mathcal{P})$
  2. Σ' is well-typed metavariable substitution environment that extends Σ
  3. $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t'\{t_1/y_1\ ;\ \ldots\ ;\ t_n/y_n\}) \downarrow \Sigma'(t)$

  Moreover, since by hypothesis $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma_{?i})] \vdash_\omega \Sigma'(t') : T$, we deduce that t' is also a well-typed term.

  By hypothesis, the metavariable substitution environment Σ'' satisfies the judgement $\Sigma' \oplus (?i := t') = \Sigma''$. Thus, by definition of the judgement, Σ'' is valid. If we prove that Σ'' is also well-typed in $\Sigma'(\mathcal{P}')$, then we can easily prove all the four parts of the thesis by exploiting the inductive hypothesis for the last premise of the Unif-Meta-Term-Miss rule and the lemmas 10.34 and 10.31 (as we already did in the case Unif-Meta-Meta-Same-Meta).

  Since we already know that Σ'' is valid (i.e. that $\ll_{\Sigma''(\mathcal{P}')}$ is irreflexive) and that Σ' is well-typed, to prove that Σ'' is well-typed we just have to show that $E[\Sigma''(\mathcal{P}'); \Sigma''(\Gamma_{?i})] \vdash \Sigma''(t') : \Sigma''(T_{?i})$. The result follows from the hypothesis $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma_{?i})] \vdash_\omega \Sigma'(t') : T$, the inductive hypothesis $E[\Sigma''(\mathcal{P}'); \Sigma''(\Gamma_{?i})] \vdash \Sigma''(T) \downarrow \Sigma''(T_{?i})$, lemma 10.31 and the Conv typing rule of the calculus.

- *Case Unif-Meta-Term-Hit*:

  To show that the third part of the thesis holds, we have to prove that $E[\Sigma''(\mathcal{P}''); \Sigma''(\Gamma)] \vdash \Sigma''(?i[t_1, \ldots, t_n]) \downarrow \Sigma''(t)$. By hypothesis, we know that $\Sigma(?i) = t'$ and that $E[\mathcal{P}; \Sigma; \Gamma] \vdash t \Downarrow t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\} \bowtie \mathcal{P}', \Sigma'$. Thus, by inductive hypothesis, we also know that $E[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma)] \vdash \Sigma'(t) \downarrow \Sigma'(t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\})$. Moreover, by inductive hypothesis, we also know that $\Sigma''(\mathcal{P}'')$ is a well-formed extension of $\Sigma'(\mathcal{P}')$ and that Σ'' is a well-typed extension of Σ' in $\Sigma''(\mathcal{P}'')$. Thus, by lemma 10.31, we also know that $E[\Sigma''(\mathcal{P}''); \Sigma'(\Gamma)] \vdash \Sigma''(t) \downarrow \Sigma''(t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\})$. Since $\Sigma(?i[t_1, \ldots, t_n]) = t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\}$ and since Σ'' is an extension of Σ', by lemma 10.31 we deduce that $E[\Sigma''(\mathcal{P}''); \Sigma'(\Gamma)] \vdash \Sigma''(t) \downarrow \Sigma''(?i[t_1 \ ; \ \ldots \ ; \ t_n])$, that is equivalent to the third part of the thesis.

  The fourth part of the thesis can be proved exactly in the same way, considering the hypothesis $E[\mathcal{P}'; \Sigma'; \Gamma'] \vdash T \Downarrow T_{?i}\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\} \bowtie \mathcal{P}'', \Sigma''$ in place of the hypothesis $E[\mathcal{P}; \Sigma; \Gamma] \vdash t \Downarrow t'\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\} \bowtie \mathcal{P}', \Sigma'$.

  To first two parts of the thesis can be trivially derived from the composition of the inductive hypotheses.

The case Unif-Conv-Conv is trivial. All the other cases are very similar and they can be proved by exploiting the inductive hypotheses. We just show the case Unif-Prod-Prod as an example:

- *Case Unif-Prod-Prod*:

  We prove only the third part of the thesis. The proof of all the other parts is similar. By induction hypothesis we know that:

$$E[\Sigma'(\mathcal{P}', \Sigma'(\Gamma)] \vdash \Sigma'(T) \downarrow \Sigma'(T') \tag{10.51}$$

$$E[\Sigma''(\mathcal{P}'', \Sigma''(\Gamma; x : T)] \vdash \Sigma''(t) \downarrow \Sigma''(t'\{x/x'\}) \tag{10.52}$$

  Applying lemma 10.31 to 10.51 we derive

$$E[\Sigma''(\mathcal{P}'', \Sigma''(\Gamma)] \vdash \Sigma''(T) \downarrow \Sigma''(T') \tag{10.53}$$

  Thus we can build a derivation of $E[\Sigma''(\mathcal{P}''), \Sigma''(\Gamma)] \vdash \Sigma''(\Pi x : T.t) \downarrow \Sigma''(\Pi x' : T'.t')$ by applying the rule Conv-Prod-Prod to the premises 10.53 and 10.52, proving the thesis.

□

Theorem 10.54 (Soundness of the unification judgment, part II) *Let E be a well-formed environment, $\mathcal{P}$ a proof problem and Σ a metavariable substitution environment such that $\Sigma(\mathcal{P})$ is well-formed and Σ is well-typed in $\Sigma(\mathcal{P})$. Let $\Sigma(\Gamma)$ be a well-formed context in E and $\Sigma(\mathcal{P})$, $(\Sigma(t_1), \Sigma(t_2))$ be an unification problem in E, $\Sigma(\mathcal{P})$ and $\Sigma(\Gamma)$ such that both t_1 and t_2 are well-typed.*

Suppose that there exists no derivation of $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \perp$ *and there exists no derivation of* $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}',\Sigma'$ *for some proof problem* $\mathcal{P}'$ *and metavariable substitution environment* Σ'*.*

Then there exists no proof problem $\mathcal{P}'$ *and metavariable substitution environment* Σ' *such that*

1. $\Sigma'(\mathcal{P}')$ *is a well-formed extension of* $\Sigma(\mathcal{P})$
2. Σ' *is a well-typed extension of* Σ *in* $\Sigma'(\mathcal{P}')$
3. $E[\Sigma'(\mathcal{P}');\Sigma'(\Gamma)] \vdash \Sigma'(t_1) \downarrow \Sigma'(t_2)$
4. $E[\Sigma'(\mathcal{P}');\Sigma'(\Gamma)] \vdash \Sigma'(T_1) \downarrow \Sigma'(T_2)$ *where* T_i *is the type of* t_i *in* E*,* $\Sigma'(\mathcal{P}')$ *and* $\Sigma'(\Gamma)$

Proof: We proceed by induction over the structure of t_1. In every case we observe that, since by hypothesis $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \perp$ is not derivable, then at least one of the premises of the rule Unif-maybe must be not inhabited. Since all the premises correspond to decidable judgements, we conclude that at least one of the following assertions is true:

- $E[\Sigma(\mathcal{P});\Sigma(\Gamma)] \vdash \Sigma(t_1) \downarrow \Sigma(t_2)$
  Impossible since otherwise the rule Unif-Conv-Conv could be applied to derive $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P};\Sigma$ that is impossible by hypothesis.

- $\mathcal{M}(t_1) \cup \mathcal{M}(t_2) = \emptyset$
  Thus we know that t_1 and t_2 are closed terms. We can distinguish two cases: either t_1 and t_2 are convertible — that is impossible since otherwise the rule Unif-Conv-Conv could be applied — or t_1 and t_2 are not convertible. In the latter case, for each metavariable substitution environment Σ' we have $\Sigma'(t_1) = t_1$ and $\Sigma'(t_2) = t_2$ and so $E[\Sigma'(\mathcal{P}');\Sigma'(\Gamma)] \nvdash \Sigma'(t_1) \downarrow \Sigma'(t_2)$. Hence the thesis.

- *both* t_1 *and* t_2 *are stiff.*
  If t_1 and t_2 are not instances of the same constructor of CIC, then we can apply corollary 10.38 to conclude the thesis. Otherwise, by hypothesis we know that none of the Unif-*T*-*T* rules can be applied successfully. Since the rules Unif-*T*-*T* are syntax oriented, at most one rule could have been applied. Since the application of the rule is not successful, at least one of the premise must not hold. By inspection of the appropriate Unif-*T*-*T*, we notice that all the premises are of the form $E[\mathcal{P}';\Sigma';\Gamma'] \vdash t'_1 \Downarrow t'_2 \bowtie \mathcal{P}';\Sigma'$ for a couple t'_1, t'_2 of corresponding direct subterms of t_1 and t_2. Thus there is at least one premise $E[\mathcal{P}';\Sigma';\Gamma'] \vdash t'_1 \Downarrow t'_2 \bowtie \mathcal{P}';\Sigma'$ that does not hold. For the same couple of subterms t'_1, t'_2, we also know that $E[\mathcal{P}';\Sigma';\Gamma'] \vdash t'_1 \Downarrow t'_2 \bowtie \perp$ does not hold, since otherwise we could apply the corresponding Unif-maybe-propagation-rule to conclude that $E[\mathcal{P};\Sigma;\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \perp$, that is absurd by hypothesis. Thus, by inductive hypothesis, we know that there exists no metavariable substitution environment Σ'' and proof problem $\mathcal{P}''$

such that $E[\Sigma''(\mathcal{P}'');\Sigma''(\Gamma)] \vdash \Sigma''(t'_1) \downarrow \Sigma''(t'_2)$. By inspection of the conversion algorithm rules, we conclude the thesis.

□

10.3 Refinement

In this section we address the problem of *refinement*. Given a proof problem $\mathcal{P}$ and an open term t, refinement looks for the "simplest" open term t' such that t' is a weakly well-typed instantiation of t. Since the instantiation of t to t' can require the creation of new metavariables and the instantiation of some of the existing ones, a new proof problem $\mathcal{P}'$ is also returned.

Refinement is a fundamental operation for the reconstruction of missing information, and it plays an important role in the parsing phase (e.g. to infer implicit arguments) and in the implementation of several tactics. McBride [82] suggests that any primitive tactic should be implemented on top of refinement. In this dissertation we will also use refinement to implement an efficient algorithm for parsing of ambiguous expressions (see Sect. 10.4).

As unification can be seen as an extension of the convertibility test with an appropriate error recovering strategy that can constraint the value and types of the metavariables, refinement is, in the same sense, an extension of type-checking. Thus we will introduce a set of inference rules for refinement by modifying the inference rules already given in Sect. 9.5 for the operational presentation of type-checking. The novelty with respect to the standard presentation of refinement is the management of the environment, that is reconstructed on the fly during refinement. The feature is directly inherited from the operational presentation of the typing rules.

Formally, we introduce several new judgements:

- $whd(E[\mathcal{P}';\Sigma;\Gamma],t) = t'$
  that grants that $whd(E[\Sigma(\mathcal{P}');\Sigma(\Gamma),\Sigma(t)) = \Sigma(t')$.

- $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : T \bowtie E',\mathcal{P}',\Sigma'$
  that grants that $E'[\Sigma'(\mathcal{P}'),\Sigma'(\Gamma)] \vdash \Sigma'(t) : \Sigma'(T)$.

- $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : \bot$.
  Refinement is undecidable. More precisely, we are not able to decide the existence of a metavariable substitution environment that grants weakly well-typedness of the term; we are also unable to decide if all metavariable substitution environments make the term not weakly well-typed. The judgement under consideration just asserts that the problem $E'[\mathcal{P}',\Sigma,\Gamma] \vdash t : \_$ is one of the cases were we are unable to assert anything at all.

We will also grant that if there does not exist any proof problem $\mathcal{P}$ and metavariable substitution environment Σ such that $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : T \bowtie E', \mathcal{P}', \Sigma'$, then there does not exists any proof problem $\mathcal{P}'$ and metavariable substitution environment Σ' such that $E'[\Sigma'(\mathcal{P}'), \Sigma'(\Gamma)] \vdash \Sigma'(t) : \Sigma'(T)$.

As usual, to shorten the rules, in the following rules the symbol $\star$ stands for the following mutual declaration of (co)inductive types:

$$
\begin{array}{l}
(v_1, \ldots, v_n)\Pi x_1 : U_1 \ldots \Pi x_h : U_h.\\
\quad \{i_1 : A_1 := \{k_1^1 : C_1^1 \ ; \ \ldots \ ; \ k_{n_1}^1 : C_{n_1}^1\} ;\\
\quad \ldots ;\\
\quad i_m : A_m := \{k_1^m : C_1^m \ ; \ \ldots \ ; \ k_{n_m}^m : C_{n_m}^m\}\\
\quad \}_{\text{(CO)IND}}
\end{array}
$$

Definition 10.55 *Let* E *be a well-formed environment,* $\mathcal{P}$ *a proof problem in* E *and* Σ *a metavariable substitution environment such that* $\Sigma(\mathcal{P})$ *is well-formed and* Σ *is well-typed in* $\Sigma(\mathcal{P})$*. Let* Γ *be a well-formed context in* E *and* $\Sigma(\mathcal{P})$ *and let* t *and* t' *be two terms.*
$whd(E[\mathcal{P};\Sigma;\Gamma], t) = t'$ *if and only if* $whd(E[\Sigma(\mathcal{P}); \Sigma(\Gamma), \Sigma(t)) = \Sigma(t')$*.*

We do not precise any further the previous judgement. Operationally, we can implement it by applying Σ to both Γ and t, and then computing t' by means of the GKAM_{CIC} reduction machine described in Sect. 9.3.3. In this case $\Sigma(t')$ will be equal to t'.

An alternative implementation consists in extending the GKAM_{CIC} reduction machine to handle also a well-typed metavariable substitution environment Σ: the machine rule that deals with a metavariable occurrence would look for an instantiation of the metavariable in Σ, retrieving the associated value if it exists and replacing the metavariable with it. Termination of the machine is a consequence of the irreflexivity of $\ll$ and the well-typedness of Σ.

Definition 10.56 *Let* E *be a well-formed environment,* $\mathcal{P}$ *a proof problem in* E *and* Σ *a metavariable substitution environment such that* $\Sigma(\mathcal{P})$ *is well-formed and* Σ *is well-typed in* $\Sigma(\mathcal{P})$*. Let* t *be a term. The judgements* $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : T \bowtie E', \mathcal{P}', \Sigma'$ *and* $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : \bot$*, together with their companion judgement of well-formedness, are completely defined by the following inference rules.*
R-WF-ExplicitSubstitution

$$
\frac{\begin{array}{l}
(y_1 : T_1 \ ; \ \ldots \ ; \ y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P}\\
(\langle\Phi\rangle E_i[\mathcal{P}_i, \Sigma_i, \Gamma] \Vdash_\omega t_i : T_i' \bowtie E_{i+1}, \mathcal{P}_{i+1}, \Sigma_{i+1})_{i=1,\ldots,n}\\
(\langle\Phi\rangle E_{n+1}[\mathcal{P}_{n+i}, \Sigma_{n+i}, \Gamma] \vdash T_i' \Downarrow T_i\{t_1/y_1 \ ; \ \ldots \ ; \ t_{i-1}/y_{i-1}\} \bowtie \mathcal{P}_{n+i+1}, \Sigma_{n+i+1})_{i=1,\ldots,n}
\end{array}}{\langle\Phi\rangle E_1[\mathcal{P}_1, \Sigma_1, \Gamma] \Vdash_\omega \mathcal{WF}(?i, [t_1, \ldots, t_n]) \bowtie E_{n+1}, \mathcal{P}_{2n+1}, \Sigma_{2n+1}}
$$

R-Meta

$$
\frac{(y_1 : T_1 \ ; \ \ldots \ ; \ y_n : T_n \vdash ?i : T_{?i}) \in \mathcal{P} \qquad \langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(?i, [t_1, \ldots, t_n]) \bowtie E', \mathcal{P}', \Sigma'}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega ?i[t_1, \ldots, t_n] : T_{?i}\{t_1/y_1 \ ; \ \ldots \ ; \ t_n/y_n\} \bowtie E', \mathcal{P}', \Sigma'}
$$

R-WF-ens

$$\frac{\begin{array}{c}\textit{the object referenced by } r \textit{ is abstracted over } v_i : T_i \textit{ for each } i\\ (\langle\Phi\rangle E_i[\mathcal{P}_i, \Sigma_i, \Gamma] \Vdash_\omega t_i : T_i' \bowtie E_{i+1}, \mathcal{P}_{i+1}, \Sigma_{i+1})_{i=1,\ldots,n}\\ (E_{n+1}[\mathcal{P}_{n+i}, \Sigma_{n+i}, \Gamma] \vdash T_i' \Downarrow T_i\{t_1/v_1\ ;\ \ldots\ ;\ t_{i-1}/v_{i-1}\}_v \bowtie \mathcal{P}_{n+i+1}, \Sigma_{n+i+1})_{i=1,\ldots,n}\\ (t_i = v_i \textit{ or } t_i \textit{ is not convertible with } v_i)_{i=1,\mathrm{ldots},n}\end{array}}{\langle\Phi\rangle E_1[\mathcal{P}_1, \Sigma_1, \Gamma] \Vdash_\omega \mathcal{WF}(r, t_1/v_1\ ;\ \ldots\ ;\ t_n/v_n) \bowtie E_{n+1}, \mathcal{P}_{2n+1}, \Sigma_{2n+1}}$$

R-Ax-Prop

$$\frac{}{\langle\Phi\rangle E[\mathcal{P}; \Sigma; \Gamma] \Vdash_\omega \textbf{Prop} : \textbf{Type} \bowtie E, \mathcal{P}, \Sigma}$$

R-Ax-Set

$$\frac{}{\langle\Phi\rangle E[\mathcal{P}; \Sigma; \Gamma] \Vdash_\omega \textbf{Set} : \textbf{Type} \bowtie E, \mathcal{P}, \Sigma}$$

R-Ax-Acc

$$\frac{}{\langle\Phi\rangle E[\mathcal{P}; \Sigma; \Gamma] \Vdash_\omega \textbf{Type} : \textbf{Type} \bowtie E, \mathcal{P}, \Sigma}$$

R-Var

$$\frac{(x : T) \in \Gamma \quad \text{or} \quad (x : T := t) \in \Gamma}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega x : T \bowtie E, \mathcal{P}, \Sigma}$$

R-SectVar

$$\frac{\mathrm{fetch}(\Phi, E, v, E', (v(v_1, \ldots, v_n) : T)) \qquad \langle\Phi\rangle E'[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(v, \sigma) \bowtie E'', \mathcal{P}', \Sigma'}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega v[\sigma] : T\{\sigma\}_v \bowtie E'', \mathcal{P}', \Sigma'}$$

R-Const

$$\frac{\mathrm{fetch}(\Phi, E, c, E', (c(v_1, \ldots, v_n) : T := t)) \qquad \langle\Phi\rangle E'[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(c, \sigma) \bowtie E'', \mathcal{P}', \Sigma'}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega c[\sigma] : T\{\sigma\}_v \bowtie E'', \mathcal{P}', \Sigma'}$$

R-Ind

$$\frac{\mathrm{fetch}(\Phi, E, i_j, E', \star) \qquad \langle\Phi\rangle E'[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(i_j, \sigma) \bowtie E'', \mathcal{P}', \Sigma'}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega i_j[\sigma] : (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.A_j)\{\sigma\}_v \bowtie E'', \mathcal{P}', \Sigma'}$$

R-Constr

$$\frac{\mathrm{fetch}(\Phi, E, k_l^j, E', \star) \qquad \langle\Phi\rangle E'[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(k_l^j, \sigma) \bowtie E'', \mathcal{P}', \Sigma'}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega k_l^j[\sigma] : (\Pi x_1 : U_1 \ldots \Pi x_h : U_h.C_l^j)\{\sigma\}_v \bowtie E'', \mathcal{P}', \Sigma'}$$

R-Prod

$$\frac{\begin{array}{ll}\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega T : S_1 \bowtie E', \mathcal{P}', \Sigma' & \mathrm{whd}(E'[\mathcal{P}', \Sigma', \Gamma], S_1) = s_1\\ \langle\Phi\rangle E'[\mathcal{P}', \Sigma', \Gamma, (x : T)] \Vdash_\omega U : S_2 \bowtie E'', \mathcal{P}'', \Sigma'' & \mathrm{whd}(E''[\mathcal{P}'', \Sigma'', \Gamma, (x : T)], S_2) = s_2\end{array}}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \Pi x : T.U : s_2 \bowtie E'', \mathcal{P}'', \Sigma''}$$

R-Prod-maybe-1

$$\frac{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega T : S_1 \bowtie E', \mathcal{P}', \Sigma' \qquad \mathrm{whd}(E'[\mathcal{P}', \Sigma', \Gamma], S_1) \textit{ is not stiff}}{\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \Pi x : T.U : \bot}$$

R-Prod-maybe-2

$$\frac{\begin{array}{ll}\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega T : S_1 \bowtie E',\mathcal{P}',\Sigma' & whd(E'[\mathcal{P}',\Sigma',\Gamma],S_1) = s_1 \\ \langle\Phi\rangle E'[\mathcal{P}',\Sigma',\Gamma,(x:T)] \Vdash_\omega U : S_2 \bowtie E'',\mathcal{P}'',\Sigma'' & whd(E''[\mathcal{P}'',\Sigma'',\Gamma,(x:T)],S_2) \textit{ is not stiff}\end{array}}{\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \Pi x:T.U : \bot}$$

R-Lam

$$\frac{\begin{array}{c}\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega T : S \bowtie E',\mathcal{P}',\Sigma' \quad whd(E'[\mathcal{P}',\Sigma',\Gamma],S) = s \\ \langle\Phi\rangle E'[\mathcal{P}',\Sigma',\Gamma,(x:T)] \Vdash_\omega t : U \bowtie E'',\mathcal{P}'',\Sigma''\end{array}}{\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \lambda x:T.t : \Pi x:T.U \bowtie E'',\mathcal{P}'',\Sigma''}$$

R-Lam-maybe

$$\frac{\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega T : S \bowtie E',\mathcal{P}',\Sigma' \quad whd(E'[\mathcal{P}',\Sigma',\Gamma],S) \textit{ is not stiff}}{\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \lambda x:T.t : \bot}$$

R-LetIn

$$\frac{\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega t_1 : T_1 \bowtie E',\mathcal{P}',\Sigma' \quad \langle\Phi\rangle E'[\mathcal{P}',\Sigma',\Gamma,(x:T_1:=t_1)] \Vdash_\omega t_2 : T \bowtie E'',\mathcal{P}'',\Sigma''}{\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \mathtt{let}\ x := t_1\ \mathtt{in}\ t_2 : T\{t_1/x\} \bowtie E'',\mathcal{P}'',\Sigma''}$$

R-App

$$\frac{\begin{array}{c}\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : S \bowtie E',\mathcal{P}',\Sigma' \quad whd(E'[\mathcal{P}';\Sigma';\Gamma],S) = \Pi x:U.T \\ \langle\Phi\rangle E'[\mathcal{P}';\Sigma';\Gamma] \Vdash_\omega u : U' \bowtie E'',\mathcal{P}'',\Sigma'' \quad E''[\mathcal{P}'';\Sigma'';\Gamma] \vdash U \Downarrow U' \bowtie \mathcal{P}''',\Sigma'''\end{array}}{\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega (t\ u) : T\{u/x\} \bowtie E'',\mathcal{P}''',\Sigma'''}$$

R-App-maybe

$$\frac{\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : S \bowtie E',\mathcal{P}',\Sigma' \quad whd(E'[\mathcal{P}';\Sigma';\Gamma],S) \textit{ is not stiff}}{\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega (t\ u) : \bot}$$

R-Case

$$\frac{\begin{array}{ll}fetch(\Phi,E_1,i_l,E_2,\star) & \\ \langle\Phi\rangle E_2[\mathcal{P}_1,\Sigma_1,\Gamma] \Vdash_\omega t : I \bowtie E_3,\mathcal{P}_2,\Sigma_2 & whd(E_3[\mathcal{P}_2,\Sigma_2,\Gamma],I) = (i_l\ t_1 \dots t_h\ t'_1 \dots t'_s) \\ \langle\Phi\rangle E_3[\mathcal{P}_2,\Sigma_2,\Gamma] \Vdash_\omega T : A \bowtie E_4,\mathcal{P}_3,\Sigma_3 & [(i_l\ t_1 \dots t_h) : A_l\{t_1/x_1\ ;\ \dots\ ;\ t_h/x_h\}|A] \\ (\langle\Phi\rangle E_{3+j}[\mathcal{P}_{2+j},\Sigma_{2+j}\Gamma] \Vdash_\omega f_j : T_j \bowtie E_{4+j},\mathcal{P}_{3+j},\Sigma_{3+j})_{j=1,\dots,n_l} & \\ \multicolumn{2}{l}{(E_{4+n_l}[\mathcal{P}_{3+n_l},\Sigma_{3+n_l},\Gamma] \Vdash T_j \Downarrow \Delta\{C^l_j\{t_1/x_1\ ;\ \dots\ ;\ t_h/x_h\},T,(k^l_j\ t_1 \dots t_h)\} \bowtie \mathcal{P}_{4+n_l},\Sigma_{4+n_l})_{j=1,\dots,n_l}}\end{array}}{\langle\Phi\rangle E_1[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \left(\begin{array}{l}<T> \mathrm{CASES}_{i_l}\ t\ \mathrm{OF} \\ \quad k^l_1\ \overline{x^1} \Rightarrow f_1 \mid \dots \mid k^l_{n_l}\ \overline{x^n} \Rightarrow f_{n_l} \\ \mathrm{END}\end{array}\right) : (T\ t'_1 \dots t'_s\ t) \bowtie E_{4+n_l},\mathcal{P}_{4+n_l},\Sigma_{4+n_l}}$$

R-Case-maybe

$$\frac{\begin{array}{ll}fetch(\Phi,E_1,i_l,E_2,\star) & \\ \langle\Phi\rangle E_2[\mathcal{P}_1,\Sigma_1,\Gamma] \Vdash_\omega t : I \bowtie E_3,\mathcal{P}_2,\Sigma_2 & whd(E_3[\mathcal{P}_2,\Sigma_2,\Gamma],I) \textit{ is not stiff}\end{array}}{\langle\Phi\rangle E_1[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \left(\begin{array}{l}<T> \mathrm{CASES}_{i_l}\ t\ \mathrm{OF} \\ \quad k^l_1\ \overline{x^1} \Rightarrow f_1 \mid \dots \mid k^l_{n_l}\ \overline{x^n} \Rightarrow f_{n_l} \\ \mathrm{END}\end{array}\right) : \bot}$$

R-Fix

$$\dfrac{\begin{array}{l}(\langle\Phi\rangle E_l[\mathcal{P}_l,\Sigma_l,\Gamma] \Vdash_\omega T_l : s_l \bowtie E_{l+1},\mathcal{P}_{l+1},\Sigma_{l+1})_{l=1,\dots,m}\\ (\langle\Phi\rangle E_{m+l}[\mathcal{P}_{m+l},\Sigma_{m+l},\Gamma,(x_1:T_1),\dots,(x_m:T_m)] \Vdash_\omega f_l : T'_l \bowtie E_{m+l+1},\mathcal{P}_{m+l+1},\Sigma_{m+l+1})_{l=1,\dots,m}\\ (E_{2m+1}[\mathcal{P}_{2m+l},\Sigma_{2m+l},\Gamma,(x_1:T_1),\dots,(x_m:T_m)] \vdash T'_l \Downarrow T_l \bowtie \mathcal{P}_{2m+l+1},\Sigma_{2m+l+1})_{l=1,\dots,m}\end{array}}{\begin{array}{l}\langle\Phi\rangle E_1[\mathcal{P}_1,\Sigma_1,\Gamma] \Vdash_\omega\\ \quad \mathrm{FIX}_j\,\{x_1/n_1 : T_1 := f_1\,;\ \dots\,;\ x_m/n_m : T_m := f_m\} : T_j \bowtie E_{2m+1},\mathcal{P}_{3m+1},\Sigma_{3m+1}\end{array}}$$

where $f_j = \lambda y_1 : T'_1 \dots \lambda y_{n_j-1} : T'_{n_j-1}.\lambda x : (i\,t_1 \dots t_n)T$

R-CoFix

$$\dfrac{\begin{array}{l}(\langle\Phi\rangle E_l[\mathcal{P}_l,\Sigma_l,\Gamma] \Vdash_\omega T_l : s_l \bowtie E_{l+1},\mathcal{P}_{l+1},\Sigma_{l+1})_{l=1,\dots,m}\\ (\langle\Phi\rangle E_{m+l}[\mathcal{P}_{m+l},\Sigma_{m+l},\Gamma,(x_1:T_1),\dots,(x_m:T_m)] \Vdash_\omega f_l : T'_l \bowtie E_{m+l+1},\mathcal{P}_{m+l+1},\Sigma_{m+l+1})_{l=1,\dots,m}\\ (E_{2m+1}[\mathcal{P}_{2m+l},\Sigma_{2m+l},\Gamma,(x_1:T_1);\dots;(x_m:T_m)] \vdash T'_l \Downarrow T_l \bowtie \mathcal{P}_{2m+l+1},\Sigma_{2m+l+1})_{l=1,\dots,m}\end{array}}{\begin{array}{c}\langle\Phi\rangle E_1[\mathcal{P}_1,\Sigma_1,\Gamma] \Vdash_\omega\\ \mathrm{COFIX}_j\,\{x_1 : T_1 := f_1\,;\ \dots\,;\ x_m : T_m := f_m\} : T_j \bowtie E_{2m+1},\mathcal{P}_{3m+1},\Sigma_{3m+1}\end{array}}$$

where $f_j = \lambda y_1 : T'_1 \dots \lambda y_{n_j-1} : T'_{n_j-1}.\lambda x : (i\,t_1 \dots t_n)T$

R-Cast

$$\dfrac{\begin{array}{c}\langle\Phi\rangle E[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega t : T_2 \bowtie E',\mathcal{P}',\Sigma' \qquad \langle\Phi\rangle E'[\mathcal{P}',\Sigma',\Gamma] \Vdash_\omega T_1 : S \bowtie E'',\mathcal{P}'',\Sigma''\\ E''[\mathcal{P}'',\Sigma'',\Gamma] \vdash T_1 \Downarrow T_2 \bowtie \mathcal{P}''',\Sigma'''\end{array}}{\langle\Phi\rangle E''[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega (t :: T_1) : T_1 \bowtie E'',\mathcal{P}''',\Sigma'''}$$

Additional $\perp$ *Generation Rules*

For each one of the previous inference rule, if the i*-th premise is of the form* $E[\mathcal{P},\Sigma,\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}',\Sigma'$*, then there exists a companion rule whose first* $(i-1)$ *premises are not changed, whose* i*-th premise is* $E[\mathcal{P},\Sigma,\Gamma] \vdash t_1 \Downarrow t_2 \bowtie \perp$ *and whose conclusion is of the form* $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : \perp$.

$\perp$ *Propagation Rules*

For each one of the previous inference rule, if a premise is of the form $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : T \bowtie E',\mathcal{P}',\Sigma'$*, then there exists a companion rule whose premise is* $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : \perp$ *and whose conclusion is also of the form* $\langle\Phi\rangle E[\mathcal{P};\Sigma;\Gamma] \Vdash_\omega t : \perp$.

Notwithstanding the huge number of inference rules, the proof of soundness of the refinement judgment is not very complex. However, since refinement is based on unification, we cannot expect it to be complete w.r.t. any interesting restriction of well-typedness.

The following lemmas are needed in the soundness proof.

Lemma 10.57 *Let E be a well-formed environment, $\mathcal{P}$ a proof problem in E and Σ a metavariable substitution environment such that $\Sigma(\mathcal{P})$ is well-formed and Σ is well-typed in $\Sigma(\mathcal{P})$. Let $\Sigma(\Gamma)$ be a well-formed context in E and $\Sigma(\mathcal{P})$ and t be a term. For all well-typed extensions Σ' of Σ, $\mathsf{whd}(\mathsf{E}[\mathcal{P};\Sigma;\Gamma],\mathsf{t}) \rhd^* \mathsf{whd}(\mathsf{E}[\mathcal{P};\Sigma';\Gamma],\mathsf{t})$.*

Proof: Trivial by theorem 10.17 and the soundness of the whd function. $\square$

Lemma 10.58 *Let E and E' be two well-formed environments, $\mathcal{P}$ a well-formed proof problem in E, Γ a well-formed context in E and $\mathcal{P}$, t and T two terms such that $\langle\Phi\rangle\mathsf{E}[\mathcal{P},\Gamma] \vdash_\omega \mathsf{t} : \mathsf{T} \bowtie \mathsf{E}'$. For each well-formed proof problem $\mathcal{P}'$ that extends $\mathcal{P}$, $\langle\Phi\rangle\mathsf{E}[\mathcal{P}',\Gamma] \vdash_\omega \mathsf{t} : \mathsf{T} \bowtie \mathsf{e}'$.*

Proof: Since t and T are well-typed in $\mathcal{P}$ and Γ, $(\mathcal{M}(\Gamma) \cup \mathcal{M}(\mathsf{t}) \cup \mathcal{M}(\mathsf{T})) \cap (\mathcal{M}(\mathsf{P}') \setminus \mathcal{M}(\mathsf{P})) = \emptyset$. By inspection of the inference rules of the operational typing judgement, we conclude that the only parts of the proof problem that are fetched are the sequents associated to the variables in the set $\mathcal{M}(\Gamma) \cup \mathcal{M}(\mathsf{t}) \cup \mathcal{M}(\mathsf{T})$. Hence the thesis. $\square$

Theorem 10.59 (Refinement soundness) *Let E be a well-formed environment, $\mathcal{P}$ a proof in E, Σ a metavariable substitution environment such that $\Sigma(\mathcal{P})$ is well-formed and Σ is well-typed in $\Sigma(\mathcal{P})$. Let Φ be a list of identifiers and t a term.*

1. *If $\langle\Phi\rangle\mathsf{E}[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \mathsf{t} : \mathsf{T} \bowtie \mathsf{E}',\mathcal{P}',\Sigma'$ is derivable then $\langle\Phi\rangle\mathsf{E}[\Sigma'(\mathcal{P}'),\Sigma'(\Gamma)] \vdash_\omega \Sigma'(\mathsf{t}) : \Sigma'(\mathsf{T}) \bowtie \mathsf{E}'$ is derivable. Moreover, $\Sigma'(\mathcal{P}')$ is a well-formed extension of $\Sigma(\mathcal{P})$ and Σ' is a well-typed extension of Σ in $\Sigma'(\mathcal{P}')$.*

2. *If there exists no derivation of $\langle\Phi\rangle\mathsf{E}[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \mathsf{t} : \bot$ and if there exists no environment E', proof problem $\mathcal{P}'$ that extends $\mathcal{P}$, metavariable substitution environments Σ and Σ' and term T such that $\langle\Phi\rangle\mathsf{E}[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \mathsf{t} : \mathsf{T} \bowtie \mathcal{P}',\Sigma'$ is derivable, then there exists no environment E', proof problem $\mathcal{P}'$, metavariable substitution environment Σ and term T such that $\langle\Phi\rangle\mathsf{E}[\Sigma'(\mathcal{P}'),\Sigma'(\Gamma)] \vdash_\omega \Sigma'(\mathsf{t}) : \Sigma'(\mathsf{T}) \bowtie \mathsf{E}'$ is derivable and for each subterm $?\mathsf{i}[\rho]$ of t whose local context is Γ', $?\mathsf{i}[\Sigma'(\rho)]$ is well-typed in E', $\Sigma'(\mathcal{P}')$ and $\Sigma'(\Gamma;\Gamma')$.*

Proof:

1. We prove the first part of the statement by induction over the derivation of $\langle\Phi\rangle\mathsf{E}[\mathcal{P},\Sigma,\Gamma] \Vdash_\omega \mathsf{t} : \mathsf{T} \bowtie \mathsf{E}',\mathcal{P}',\Sigma'$.

   Since all the cases are similar, we just show as an example the cases R-Prod and R-Appl.

- *Case R-Prod*: $\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \Pi x : T.U : s_2 \bowtie E'', \mathcal{P}'', \Sigma''$
  By inspection of the R-Prod rule, we know:

$$\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega T : S_1 \bowtie E', \mathcal{P}', \Sigma' \tag{10.60}$$

$$whd(E'[\mathcal{P}', \Sigma', \Gamma], S_1) = s_1 \tag{10.61}$$

$$\langle\Phi\rangle E'[\mathcal{P}', \Sigma', \Gamma; (x : T)] \Vdash_\omega U : S_2 \bowtie E'', \mathcal{P}'', \Sigma'' \tag{10.62}$$

$$whd(E''[\mathcal{P}'', \Sigma'', \Gamma; (x : T)], S_2) = s_2 \tag{10.63}$$

  By induction hypothesis, from 10.60 and 10.62 we know:

$$\begin{array}{c}\langle\Phi\rangle E[\Sigma'(\mathcal{P}'), \Sigma'(\Gamma)] \vdash_\omega \Sigma'(T) : \Sigma'(S_1) \bowtie E' \\ \text{and } \Sigma'(\mathcal{P}') \text{ well-formed and } \Sigma' \text{ well-typed}\end{array} \tag{10.64}$$

$$\begin{array}{c}\langle\Phi\rangle E'[\mathcal{P}'', \Sigma''(\Gamma; (x : T))] \vdash_\omega \Sigma''(U) : \Sigma''(S_2) \bowtie E'' \\ \text{and } \Sigma''(\mathcal{P}'') \text{ well-formed and } \Sigma'' \text{ well-typed}\end{array} \tag{10.65}$$

  From 10.60, 10.65, theorem 10.33 and lemma 10.58 we get:

$$\langle\Phi\rangle E[\Sigma''(\mathcal{P}''), \Sigma''(\Gamma)] \vdash_\omega \Sigma''(T) : \Sigma''(S_1) \bowtie E' \tag{10.66}$$

  Moreover, by definition of whd, we know:

$$whd(E'[\Sigma'(\mathcal{P}'); \Sigma'(\Gamma), \Sigma'(S_1)) = \Sigma'(s_1) \tag{10.67}$$

$$whd(E''[\Sigma''(\mathcal{P}''); \Sigma''(\Gamma; (x : T)), \Sigma''(S_2)) = \Sigma''(s_2) \tag{10.68}$$

  From 10.67, 10.62 and lemma 10.57 we get:

$$whd(E'[\Sigma(\mathcal{P}''); \Sigma''(\Gamma), \Sigma''(S_1)] = \Sigma''(s_1) \tag{10.69}$$

  We can construct a derivation of the first part of the thesis by applying the O-Prod rule to 10.66, 10.69, 10.65, 10.68.

- *Case R-App*: $\langle\Phi\rangle E[\mathcal{P}; \Sigma; \Gamma] \Vdash_\omega (t\ u) : T\{u/x\} \bowtie E'', \mathcal{P}''', \Sigma'''$
  By inspection of the R-App rule, we know:

$$\langle\Phi\rangle E[\mathcal{P}; \Sigma; \Gamma] \Vdash_\omega t : S \bowtie E', \mathcal{P}', \Sigma' \tag{10.70}$$

$$whd(E'[\mathcal{P}'; \Sigma'; \Gamma], S) = \Pi x : U.T \tag{10.71}$$

$$\langle\Phi\rangle E'[\mathcal{P}'; \Sigma'; \Gamma] \Vdash_\omega u : U' \bowtie E'', \mathcal{P}'', \Sigma'' \tag{10.72}$$

$$E''[\mathcal{P}''; \Sigma''; \Gamma] \vdash U \Downarrow U' \bowtie \mathcal{P}''', \Sigma''' \tag{10.73}$$

  By the soundness of the unification judgement (theorem 10.46) we have:

$$E''[\Sigma'''(\mathcal{P}'''), \Sigma'''(\Gamma)] \vdash \Sigma'''(U) \downarrow \Sigma'''(U') \text{ and } \Sigma''', \mathcal{P}''' \text{ well-formed} \tag{10.74}$$

Mimicking the proof of the previous case, we conclude:

$$E[\Sigma''(\mathcal{P}'''), \Sigma'''(\Gamma)] \vdash_\omega \Sigma'''(t) : \Sigma'''(S) \bowtie E' \tag{10.75}$$

$$whd(E'[\Sigma'(\mathcal{P}'''), \Sigma'''(\Gamma)], \Sigma'''(S)) = \Sigma'''(\Pi x : U.T) \tag{10.76}$$

$$E[\Sigma'''(\mathcal{P}'''), \Sigma'''(\Gamma)] \vdash_\omega \Sigma'''(u) : \Sigma'''(U') \bowtie E' \tag{10.77}$$

$$\tag{10.78}$$

We can construct a derivation of the first part of the thesis by applying the O-Appl rule to 10.75, 10.76, 10.77, 10.74.

2. We prove the second part of the statement by induction over the structure of t.

   Since the judgement $\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega t : T \bowtie \mathcal{P}', \Sigma'$ is syntax oriented, there is at most one rule that could have been applied to conclude the derivation. Moreover, since there exists no derivation, at least one of the premises must be not inhabited.

   The judgement $\langle\Phi\rangle E[\mathcal{P}', \Sigma'(\Gamma)] \vdash_\omega \Sigma'(t) : \Sigma'(T) \bowtie E$ is also syntax oriented. Moreover, the last inference rule that is the only one that can be applied when $\Sigma'(t)$ is considered is equal to the last inference rule that can be applied when t is considered for all t that is not a metavariable occurrence.

   Finally, each inference rule of the type inference judgement corresponds to exactly one rule of the type checking judgement.

   All the previous observations lead us to consider just the following two cases:

   (a) $t = ?i[\rho]$

   (b) $t \neq ?i[\rho]$

   Let us consider the second case first. Let H be the premise that is not inhabited. By inspection of the inference rules, we know that H must be one of the following judgements:

   - H is $\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega t : T \bowtie E', \mathcal{P}', \Sigma'$
     By induction hypothesis we know that for all $\mathcal{P}'$ that extends $\mathcal{P}$ and for all Σ' that extends Σ there is no derivation of $\langle\Phi\rangle E[\Sigma'(\mathcal{P}'), \Sigma'(\Gamma)] \vdash_\omega \Sigma'(t) : \Sigma'(T) \bowtie E'$. Thus it is impossible to inhabit the premise that corresponds to H in the inference rule of the $\vdash_\omega$ judgement.
   - H is $whd(E'[\mathcal{P}', \Sigma', \Gamma], T) = t$
     By inspection of the rules of the $\Vdash_\omega$ judgement, we know that t should have been (but it is not) a sort, a Π-abstraction or the application of an inductive type. We also know definitely that t is stiff, since otherwise the corresponding "maybe" rule would have been fired. In any case, by theorem 10.38 we also know that for each metavariable

substitution environment Σ', the head of the weak head normal form of $\Sigma'(T)$ must be equal to the head of T. In particular, the weak head normal form of $\Sigma'(T)$ cannot become a sort (respectively a Π-abstraction or the application of an inductive type). Thus it is impossible to inhabit the premise that corresponds to H in the inference rule of the $\vdash_\omega$ judgement.

- H is $E[\mathcal{P}, \Sigma, \Gamma] \vdash t_1 \Downarrow t_2 \bowtie \mathcal{P}', \Sigma'$
  By theorem 10.54 we know that for each metavariable substitution environment Σ' that extends Σ and for each proof problem $\mathcal{P}'$ that extends $\mathcal{P}$ the judgement $E[\mathcal{P}', \Sigma'(\Gamma)] \vdash \Sigma'(t_1) \downarrow \Sigma'(t_2)$ is not inhabited. Thus it is impossible to inhabit the premise that corresponds to H in the inference rule of the $\vdash_\omega$ judgement.

- H is $\mathtt{fetch}(\Phi, E, c, E', d)$
  Since c is either a reference to a constant, a reference to a mutual inductive type or a reference to a mutual inductive type constructor, for each metavariable substitution environment Σ' we have $\Sigma'(c) = c$. Thus, if H is not inhabited, it cannot be inhabited also in the corresponding rule of the $\vdash_\omega$ judgement.

To conclude the thesis, we are left with the case of the application of refinement to a metavariable occurrence: $t = ?i[\rho]$. Since the R-Meta rule cannot be applied by hypothesis, we know that either $?i \notin \mathcal{P}$ or $\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(?i, [\rho]) \bowtie E', \mathcal{P}', \Sigma'$ is not inhabited.

- *Case* $?i \notin \mathcal{P}$:
  Let Σ' be a generic well-typed metavariable substitution environment. If $?i \notin \Sigma'$, then $\Sigma'(?i[\rho]) = ?i[\Sigma'(\rho)]$ and $?i$ is still not defined in $\Sigma'(\mathcal{P})$. Thus $\Sigma'(?i[\rho])$ is not well-typed (i.e. the first premise of the o-Meta rule is not inhabited). Otherwise, $?i \in \Sigma'$ but $?i \notin \Sigma'(\mathcal{P})$. This is not possible since Σ' is well-typed by hypothesis.

- *Case* $\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(?i, [\rho]) \bowtie E', \mathcal{P}', \Sigma'$ *not inhabited*:
  Let Σ' be a generic well-typed metavariable substitution environment in $\Sigma'(\mathcal{P}')$. If $?i \notin \Sigma'$, then $\Sigma'(?i[\rho]) = ?i[\Sigma'(\rho)]$. Thus, we can analyze the premises of the R-WF-ExplicitSubstitution rule as we did in the case $t \neq ?i[\rho]$ and conclude that the rule o-WF-ExplicitSubstitution cannot be applied. Thus the rule o-Meta cannot be applied. Hence the thesis.

  Otherwise let $?i \in \Sigma'$. Thus $\Sigma'(?i[\rho]) = \Sigma'(?i)\{\Sigma'(\rho)\}$. By hypothesis, we know that $\langle\Phi\rangle E[\mathcal{P}, \Sigma, \Gamma] \Vdash_\omega \mathcal{WF}(?i, [\rho]) \bowtie E', \mathcal{P}', \Sigma'$. Thus we can analyze the premises of the R-WF-ExplicitSubstitution rule as we did in the case $?t \neq ?i[\rho]$ and conclude that $\Sigma'(\rho)$ is not a well-typed substitution for $?i$ in $\Sigma'(\mathcal{P}')$. Notice, however, that it may be the case that $\Sigma'(?i)\{\Sigma'(\rho)\}$ is a well-typed term[4]. □

[4]For instance, if $(x : \mathrm{bool} \vdash ?1 : nat) \in \mathcal{P}$ and if $\Sigma' = [?1 := 2]$, then Σ' is well-typed in $\mathcal{P}$ (since $E[x : \mathrm{bool}] \vdash 2 : nat$)

10.4 An Efficient Algorithm for Ambiguous Parsing

Traditionally, mathematicians are used to very ambiguous notational conventions.

Sometimes the ambiguity is solved by looking at the context: for instance, if f is known to be a scalar value, than f^{-1} is probably the inverse value $1/f$; if f is a function, instead, then f^{-1} is probably the inverse function of f.

More often, the ambiguity is solved by picking the only interpretation that makes sense: for instance, since ϕ^2 is applied to x in the expression $\phi^2(x)$, then ϕ^2 must be a function, that is probably $f \circ f$. Indeed, interpreting $\_^2$ as the square of the scalar ϕ would not allows us to parse (x) is any "reasonable" way. Notice, however, that replacing ϕ with sin we obtain $\sin^2 x$ that is always parsed as $(\sin x)^2$ since $\sin \circ \sin$ usually is unlikely to be the intended meaning.

Moreover, mathematicians sometimes use utterly ambiguous expressions statements, being naturally able to recognize at a first glance the only interpretation that holds. For instance, knowing that A and B are propositional formulae, it is not unusual to find in a textbook of logic the statement $A \vee B = B \vee A$. Of course, any mathematician presented with the statement would interpret it as $(A \vee B) = (B \vee A)$ where the equality sign stands for logical equivalence. Nobody would peek the alternative interpretation $A \vee (B = B) \vee A$, that could have been equally reasonable, but that is generally false. A computer scientist could object that only one of the two previous interpretation is correct, since one among the disjunction and the equality sign must have a higher precedence than the other. Notice, however, that in $x = y \vee x = z$ the precedence of the equality sign is lower than the precedence of disjunction, that is in contrast with the interpretation of $A \vee B = B \vee A$ as an equality of disjunctions.

Computer scientists have easily overcome the ambiguity problems in the practice of programming. Indeed, a programmer is used to learn a new programming language from scratch, and having to learn ad hoc syntactical rules does not bother anyone. Unfortunately, the problem arises again when we have to deal with natural language processing, since the natural language is ambiguous.

Notice that the case of natural language processing is generally speaking much more complex than the one of mathematics. Indeed, since the human ability of attributing a semantics to a sentence is deeply involved in the parsing of the sentence, we can expect a machine to be in trouble in recognizing the only parsing trees that actually do have a meaning. On the contrary, the

and $\Sigma'(?1[3/x]) = 2\{3/x\} = 2$ that is also well-typed, although $?1[3/x]$ is not well-typed. This observation should not be surprising: it is well-known that non well-typed λ-terms can reduce to well-typed terms. Since metavariable instantiation also involves a bit of reduction (since the explicit substitution becomes an instantaneous one), here we face the same phenomenon. Notice that, if we apply the transformation $[\![\_]\!]$ to the functional representation defined in Sect. 10.1.7, the problem would be detected by the type-checking judgement.

domain of logic and mathematics is much more restricted, and the task of attributing a meaning to a mathematical expression is much easier to achieve.

Before addressing the topic of this section, that is *parsing of ambiguous mathematical statements*, we should wonder whether this activity is of any relevance in the domain of proof assistants and Mathematical Knowledge Management tools.

As a programmer has to learn the syntax of a programming language before using it, we could easily imagine a mathematician to have to learn the syntax of a proof assistant before using it. Claims that mathematicians refuse to use proof assistants because of their complex syntax are totally ridiculous: mathematicians refuse to use proof assistants because they see no real benefit for their activity, while the burden of the formal approach is evident. Notwithstanding this fact, motivations for trying to recognize ambiguous mathematical grammars do exist.

First of all, different mathematicians adopt different notations for the same concept. The context is usually more than sufficient to understand the notation used, even if it is not explicitly stated by the author. In the context of MKM, though, there are several situations where the user has to write or read a mathematical formula outside any notion of context. For instance, let us consider a search engine for mathematical results, as the one described in Chap. 5. The user has to enter a statement to retrieve all the proofs of the statement. To do that, though, she has to pick a notation and she has to use symbols and constants that have not been given any meaning by a context. Thus the search engine has to be able to parse the ambiguous sentence, disambiguate it and perform the search up to the notational conventions.

Secondly, a feature that is usually provided to the user by a proof assistant is that of defining new notations, that are stored in the context. In all these systems, though, sooner or later it will happen that a user is willing to merge together two contexts where the same notation is overloaded for different concepts. Since these systems do not handle overloading, that is the easiest form of ambiguity, one of the two notations is actually masked and the user has to redefine it, creating a break point in the notational uniformity of the whole development. For instance, the Mizar library committee has already faced several times these kind of problems, that were solved by changing one of the two notations and updating the whole Mizar library.

Clearly, the solution of always picking a fresh notation for new concepts is not really a long term solution, especially in the case of notations that are heavily overloaded in mathematics, as the "+" and "*" symbols. The Coq team, for instance, has recently improved the parser of Coq to exploit the context in a better way, allowing the automatic discrimination of overloaded symbols thanks to the notion of *scope*. A scope is a region of text were only some parsing rules are activated. For instance, there exist a scope were "*" is interpreted as the multiplication between Peano numbers and another one were it is interpreted as the product of two types. A syntactical device is given to the user to change the current scope, even in the middle of a formula. More interestingly, though, scopes are associated to the arguments of the constants, so that when an argument of type **Set** is met the scope that handles "*" as a type product is opened.

The solution of the Coq team partially solves the problem of overloading. Often, though, it is not powerful enough, since scopes are chosen by exploiting the context only in a minimal way. Thus the problem of parsing ambiguous mathematical expressions seems to be still actual even in the limited context of a proof assistant.

Finally, one of the topics of Mathematical Knowledge Management is the retrieval, conservation, digitalization and enhancement of already existing mathematical knowledge, that is retrieved by means of optical character recognition techniques (OCR) or, if the material is more recent, by extraction from LaTeX, PostScript or PDF documents. Since the notation used in these documents is ambiguous, the enhancement phase — if not even the digitalization phase — must address the problem of ambiguous parsing. Moreover, since the amount of legacy mathematical documents is enormous, we should try to minimize as much as possible the amount of disambiguation work left to the human.

Let us now fix a minimal set of requirements for an algorithm for parsing of ambiguous mathematical expressions.

10.4.1 Requirements

Traditionally, compilers are implemented as a pipeline whose first two phases, lexing and parsing, are followed by the phase of semantic analysis. The grammars used for programming language are not ambiguous grammars. A grammar is said to be ambiguous when there exists an expressions E such that there exists two different parsing trees for E. Thus, the output of the parsing phase is either a parsing error or the unique parsing tree for the expression given in input. The subsequent phase of semantic analysis completes the parsing phase by associating occurrences of bound variables to their binders and it also associates other informations, such as types, to the nodes of the abstract tree. Semantic analysis can also fail, detecting an error in the input.

Ambiguous grammars are considered in the context of natural language processing. In these cases, the output of the parsing phase is no longer a single tree, but a set of parsing trees.

Ambiguous grammars are just one of the different aspects of ambiguity. We say that a parsing tree is ambiguous when the semantic analysis phase can associate to the tree more than one semantics — whatever the semantics is. Thus the output of the semantic analysis is a set of semantics for each parsing tree in the input set.

We say that an algorithm performs ambiguous parsing when it associates to an expression E the set of all the semantics for E. We are interested in ambiguous parsing algorithms that satisfy the following requirements:

1. The algorithm should be *compositional*, in the sense that the parsing phase and the semantic analysis phase should be as separate as possible.

2. The algorithm should be *efficient*, in the sense that semantic errors must be detected as early as possible, preventing further parsing and semantic analysis of already incorrect interpretations.

The two requirements are clearly in contrast. For instance, let us consider the following ambiguous expression recognized by a non ambiguous grammar: $(1+2)*3^{-1} = 1$. The expression is possibly ambiguous — at least from the point of view of a user of a proof assistant — since each symbol in the expression have several possible interpretations:

1. Each number in the expression but "-1" can be at least a natural, integer, rational, real or complex number.

2. "-1" can be either the application of the unary operator "-" to 1, or the integer, rational, real or complex number -1.

3. Equality can be either Leibniz equality — the smallest reflexive relation, generally undecidable — or the decidable equality relation over natural, integer or rational numbers, or the undecidable negation of the notion of apartness for real and complex numbers (in the context of constructive mathematics).

4. The algebraic operators "+" and "*" can be defined over natural, integers, rationals, reals or complex numbers. They are surely defined over other sets of data types, such as sets, types and lattices.

5. Implicit coercions can be inserted around sub-expressions to inject a number of one type in a "bigger type". For instance, the number 3 could stand for $(\mathtt{to\_Q}(S(S(SO))))$ where $(S(S(SO)))$ is the third Peano integer and $\mathtt{to\_Q}$ is the embedding function that maps each Peano integer to the corresponding rational number.

6. Since different representations of the same concept may have different computational properties, several representations of, say, natural numbers may be available. For instance, it is well-known that Peano integers are well suited for proofs by induction, while unbound strings of bits allow for exponentially faster implementations of the usual arithmetical operations.

The simpler compositional algorithm would just perform parsing first, returning a huge set of parsing trees, one for each combination of the possible interpretations for each symbol. For instance, one possible output would be the tree whose root is equality over reals numbers and whose second argument is the Peano number 1. Semantic analysis would then proceed at pruning out the most part of the combinations. For instance, every tree whose root is equality over real numbers and whose second argument is not a real number would be pruned. Thus, the algorithm would not be efficient, since several trees would be parsed and analyzed even if looking at the root and its second argument already provides enough information to prune them out.

On the contrary, the most efficient top-down algorithm would generate every possible interpretation for the root, and for each interpretation would call itself on the two arguments remembering what the expected type of the argument is. The expected type would be used to pick for the second argument of the root just the only interpretation that respects the type. Corresponding efficient bottom-up algorithms can be easily built as well. Clearly, the efficient algorithm described is not compositional, since it performs type-checking already during the parsing phase, and just on a fragment of the tree. Thus the parsing and semantic analysis phases should be interleaved and they should either be made mutual recursive or implemented as coroutines.

The proposed example was made artificially more complex than what it should have been. Indeed, the expression considered was ambiguous, but all the interpretations could have shared the same unambiguous grammar. Thus it would have been possible to adopt a smarter (and more classical) parser that returned the only unambiguous parsing tree, and let the semantic phase deal with the problem of associating the interpretation to each symbol and constant. Notice, however, that the modification does not make the example less interesting. Indeed, replacing the parsing phase with the resolution phase, the same remarks as above apply again: the first algorithm would be non efficient, but compositional — in the sense that the resolution and typing phase do not need to depend on each other. On the contrary the second algorithm would be efficient but not compositional.

Moreover, there exists more complex cases where ambiguous grammars are required. For instance, the expression $A \vee B = B \vee A$ given in the previous section admits at least four different parsing trees: $((A \vee B) = B) \vee A$, $(A \vee B) = (B \vee A)$, $A \vee (B = (B \vee A))$ and $A \vee (B = B) \vee A$. Moreover, the last case can be ramified further if we consider the possibility of a binary operator $\vee$, either left or right associative, and of an n-ary operator defined elsewhere in the library. If we are willing to deal also with these cases and if we are looking for an efficient algorithm, it seems that we must resolve the ambiguities already during the parsing phase.

In the next section we propose a compositional and yet efficient algorithm for parsing ambiguous expressions. Despite its simplicity, to the author knowledge the algorithm, based on refinement, is new. Its novelty was also confirmed by private communication with Aarne Ranta, who is the main author of Grammatical Framework (GF) [105], that represents the state of the art in the domain of grammar formalisms[5].

[5]GF is a grammar formalism based on type theory. It consists of a special-purpose programming language, a compiler

10.4.2 The Algorithm

The property of being compositional is granted by the existence of two unrelated components. The first one, called *parser*, performs "lazily" the two phases of parsing and resolution, by returning a pair whose first component is a description of the choices it has to make and whose second component is a function that produces a parsing tree given in input an oracle (called *interpretation*) that guides its choices. The second one, called *disambiguator*, "drives the parser" — or better drives its output — by feeding the second component of the parsre output with a set of interpretations. The interpretations are built from the description of the choices returned by the parser. The disambiguator uses the refinement algorithm to progressively refine the produced interpretations avoiding those choices that lead to non well-typed terms. Thus the disambiguator is the component that grants efficiency.

Let us analyze more precisely the basic idea presented in the previous paragraph.

Let $\mathcal{T}$ be the set of terms (abstract syntax trees) that is the output of the disambiguation algorithm. Let $\mathcal{S}$ be a set of identifiers or symbols that can have more than one interpretation (that we call *choice*). Finally, let $\mathcal{D}$ be a set of human readable descriptions of the choices associated to the symbols.

Definition 10.79 (Interpretation) *Interpretations, interpretation domain and interpretation codomain are mutually defined in the following way:*

- *An* interpretation domain *is a finite set whose elements are couples* $(\mathtt{s}, \mathtt{l})$ *where* $\mathtt{s}$ *is an element of* $\mathcal{S}$ *(the symbol to be interpreted) and* $\mathtt{l}$ *(the list of possible choices) is a list of couples* $(\mathtt{d}, \mathtt{t})$ *where* $\mathtt{t}$ *is a function from interpretations to* $\mathcal{T}$ *and* $\mathtt{d}$ *is the human readable description of the choice* $\mathtt{t}$*.*

- *An* interpretation codomain *is a set whose elements are either the element "?" (implicit term) or function from interpretations to* $\mathcal{T}$*.*

- *An* interpretation *is a partial function from an interpretation domain to an interpretation codomain, subject to the following condition: for each element* $\mathtt{i} = (\mathtt{s}, [(\mathtt{d}_1, \mathtt{t}_1)\ ;\ \ldots\ ;\ (\mathtt{d}_n, \mathtt{tn})]$ *of the interpretation domain, if the interpretation applied to* $\mathtt{i}$ *returns a function* $\mathtt{t}$*, then* $\mathtt{t} \in \{\mathtt{t}_1, \ldots \mathtt{t}_n\}$*.*

The previous definition is probably more readable when written in ML notation:

```
type symbol;;
type description;;
type interpretation_domain_item =
```

of the language and a generic grammar processor. Among the tasks implemented in the generic grammar processor, there are parsing, translation and type checking. Among the checks that can be performed on the grammars there are well-formedness checks and verification of semantic properties of expressions.

```
   Symbol of symbol * (description * (interpretation -> term)) list
and interpretation_codomain_item =
   Implicit
 | Term of (interpretation -> term)
and interpretation =
 interpretation_domain_item -> interpretation_codomain_item
;;
```

Intuitively, an interpretation domain item describes all the possible ways to interpret a given symbol. An interpretation codomain item is either an implicit term (refusal of choosing an interpretation for the symbol) or just one interpretation among the possible ones. Thus, an interpretation is just a kind of oracle that, given a set of choices and a set of answers for each choice, associates to each choice either an answer among the one proposed or the "no choice" term.

We let D range over interpretation domains, ϕ range over interpretations and $t_\uparrow$ over functions from interpretations to terms.

Let us define the parser.

Definition 10.80 (Parser) *A* parser *is a function that maps its input (a string or a stream of tokens) into a couple* $(D, t_\uparrow)$ *where* D *is an interpretation domain and* $t_\uparrow$ *is a function from interpretations that are totally defined over* D *to terms.*

Operationally, the parser can be obtained mechanically from a traditional parser for non-ambiguous grammars, either top down, bottom up, or general, by lifting its domain functions from interpretations to terms. An example (in OCamlYacc) should clarify the previous statement. Let us consider the production that parses a cast expression $(t1 :: t2)$:

```
expr :
 LPAREN expr CAST expr RPAREN
  { Cast ($2,$4) }
```

Applying the mechanical lifting transformation, the production becomes:

```
expr :
 LPAREN expr CAST expr RPAREN
  { let (dom1,mk_expr1) = $2 in
    let (dom2,mk_expr2) = $4 in
     (union dom1 dom2),
     function interp -> Cast ((mk_expr1 interp),(mk_expr2 interp))
  }
```

The output domain is the union of the domains of the subexpressions. The output function is a λ-abstraction over an interpretation, that becomes the argument of all the subexpressions (that are now lifted and need an interpretation in input).

The interesting case is the case of ambiguous parsing, i.e. the case when the parser must make a choice. The solution is simply to add all the possible choices to the output domain, and make the interpretation function perform the choice. Example:

```
expr :
 expr EQ expr
  { let (dom1,mk_expr1) = $1 in
    let (dom2,mk_expr2) = $3 in
     (union (union dom1 dom2)
      (EQ,
        ["decidable equality over natural numbers",
           function interp ->
            APPL [eq_nat ; (mk_expr1 interp) ; (mk_expr2 interp)] ;
         "leibniz equality",
           function interp ->
            APPL [eq ; IMPLICIT ; (mk_expr1 interp) ; (mk_expr2 interp)]
        ])
     ),
     function interp ->
      match (interp EQ) with
         Implicit -> IMPLICIT
       | Term mk_term -> mk_term interp
  }
```

If the oracle `interp` given in input satisfies our definition of interpretation, then `(interp EQ)` will be either `Implicit` (refusal to choose) or

```
 Term (function interp ->
  APPL [eq\_nat ; (mk\_expr1 interp) ; (mk\_expr2 interp)]
```

or

```
 Term (function interp ->
   APPL [eq ; IMPLICIT ; (mk\_expr1 interp) ; (mk\_expr2 interp)]
```

From the last example it should be clear in what sense the parsing phase proceeds "lazily" and also in what sense interpretations behave as oracles for the parsers.

The last and more interesting component is the disambiguator.

Definition 10.81 *A* disambiguator *is a function whose input is a couple* $(D, t_\uparrow)$ *where* D *is an interpretation domain and* $t_\uparrow$ *a function from interpretations totally defined over* D *to terms. The output of the disambiguator is the set* $\{t \mid \exists\phi$ *of domain* D *s.t.* $t = t_\uparrow(\phi)$ *and* t *is well-typed*$\}$

All the couples parser, disambiguator that implement the previous definitions are clearly compositional. We will now present an efficient algorithm that satisfy the specification for a disambiguator. The idea of the algorithm is to build the set of interpretations that produce well-typed terms by progressive refinements, starting from the interpretation that always refuses to choose and progressing towards an interpretation that always returns some lifted term. At each refinement step, the function $t_\uparrow$ is applied to obtain an open term, that is immediately feed to the refinement function. The refinement function acts as a sort of validator, rejecting partial instantiations that already produce ill-typed open terms. It may also provide positive information, in the sense that it may automatically instantiate several metavariables, constraining the set of future choices.

Let $\mathtt{refine}$ be a partial function from terms to terms extended with the symbol $\perp$ such that:

- If $\mathtt{refine}(t_1) = t_2$ then t_2 is a (weakly) well-typed term obtained by instantiating some of the metavariables that occur in t_1.
- If $\mathtt{refine}(t_1)$ is not defined, then there exists no (weakly) well-typed term t_2 that can be obtained by instantiating some of the metavariables that occur in t_1.

Notice that if $\mathtt{refine}(t) = \perp$ then no information is actually provided: the term could either be refinable or not. Notice also that the above definition of the $\mathtt{refine}$ function is a major simplification of the one presented in the previous chapter. For instance, the function should also have an additional input and an additional output that are proof problems. Nevertheless, the above simplification is sufficient to understand the algorithm proposed, without blurring the description with additional details. This point is further discussed in the remarks (Sect. 10.4.3).

The simplest version of the algorithm, the one that does not exploit the positive information returned by the refine operation, looks as follows (in pseudo-code):

$$
\begin{array}{l}
\mathtt{disambiguate}(D, t_\uparrow) = \\
\quad \mathtt{let}\ \mathtt{update}(\phi, s, t'_\uparrow)(s') = \begin{cases} \mathtt{Term}(t'_\uparrow) & \text{if } s' = s \\ \phi(s) & \text{otherwise} \end{cases} \\
\quad \mathtt{let}\ \phi_0 = \lambda\_.\mathtt{Implicit} \\
\quad \mathtt{let}\ \Phi = \begin{cases} \{\phi_0\} & \text{if } \mathtt{defined}(\mathtt{refine}(t_\uparrow(\phi_0))) \\ \emptyset & \text{otherwise} \end{cases} \\
\quad \mathtt{for\ each}\ (s, l) \in D \\
\qquad \Phi := \\
\qquad\quad \{\phi' \mid \phi \in \Phi \wedge t'_\uparrow \in l \wedge \phi' = \mathtt{update}(\phi, s, t'_\uparrow) \wedge \mathtt{defined}(\mathtt{refine}(t_\uparrow(\phi')))\} \\
\quad \mathtt{return}\ \Phi
\end{array}
$$

The algorithm presented above is surely efficient according to our definition of efficiency, in the sense that at each iteration of the for loop the set of interpretations $\{\phi' \mid \phi \in \Phi \wedge t'_\uparrow \in l \wedge \phi' = \mathtt{update}(\phi, s, t'_\uparrow)\}$ is immediately pruned by means of the check $\mathtt{defined}(\mathtt{refine}(t_\uparrow(\phi')))$.

The overall computational cost of the algorithm, though, can still be improved. Indeed, at each iteration of the for loop, $t_\uparrow$ is applied to an interpretation ϕ' that can differ from ϕ only by instantiating more implicit arguments. Thus, the term returned by $t_\uparrow(\phi)$ must be a prefix of the term returned by $t_\uparrow(\phi')$. This suggests that the cost of the computation of $t_\uparrow(\phi')$ could be greatly cut down by changing $t_\uparrow$ to exploit the knowledge about the partial result $t_\uparrow(\phi)$. In the same way, when $\mathtt{refine}(t_\uparrow(\phi))$ is defined and different from $\perp$, its value can easily be exploited in the refinement of $t_\uparrow(\phi')$. Combining the two optimizations, we can easily make the cost of the two operations at each iteration neglectable, without sacrificing the property of being compositional.

It is not easy to quantifying the exact gain obtained by our algorithm over the straightforward solution of first generating all the parsing trees and them type-check them. The reason is that the actual computational complexity of the refinement operation for real world terms cannot be easily estimated. On the contrary, the number of times the refinement operation will be called is known: it is the number of well-typed prefixes of the possible parsing trees. In practice, we have been using this algorithm for quite a long time in the HELM SEARCH-ENGINE, with good results: the time spent in the disambiguation phase is always neglectable, and the number of well-typed prefixes is almost linear in the number of well-typed parsing trees. The tests are not conclusive though, since in the implementation we are not yet dealing with some forms of ambiguity induced, for instance, by implicit coercions.

10.4.3 Concluding Remarks

The algorithm for parsing ambiguous mathematical expressions presented in this section must be considered as a source of inspiration for experimenting with more intelligent solutions, and not as the ultimate solution. Its value consists in showing that refinement is the key to efficient parsing of ambiguous expressions in a restricted domain where a typing discipline can be imposed over the syntax trees, and types can be used to solve the ambiguity.

Implementing the algorithm means fixing lots of details that were intentionally ignored in this exposition to make it simpler. For instance, an implicit argument is actually a stack of fresh metavariables (each metavariable being the type of the previous one, up to the third and last metavariable that is of type $\mathbf{Type_i}$ for some i). Thus, generating a new implicit argument means extending the proof problem. Thus, we should replace every term in the last section with a couple $(\mathcal{P}, t)$ where $\mathcal{P}$ is a proof-problem and t is a term whose metavariables are defined in $\mathcal{P}$.

Moreover, the non simplified refine operation does not operate on terms, but on couples `metasenv`/term. The set of unifiers computed during the refinement operations gives extremely useful information to the refinement of the current interpretation. Indeed, if an implicit argument is in the domain of an instantiation, then the instantiation describes the only possible choice that leads to well-typed terms. This information should not be wasted as in the proposed algorithm. Indeed, it is actually partially exploited in the implementation.

Another interesting topic that was completely skipped in the presentation was the construction of the interpretation domain by the parser. There exists at least two sources of information that can be exploited to construct the list of possible choices for each symbol. The first one is related to the ambiguities that arise from the resolution of an identifier. Indeed, it is not unusual to have several objects in a distributed library that are given the same name (e.g. "reflexive_property", "domain" or simply "P"). Thus to every identifier we can associate several objects. The parser needs to retrieve for each identifier the list of all the objects whose name is equal to the identifier. The task can easily be performed making the parser consult the SEARCH-ENGINE (described in Chapt. 5). Since the SEARCH-ENGINE can also be consulted at any later time, in the implementation we delay its invocation to the disambiguator, greatly reducing the size of the closure returned by the parser and improving the overall computation time.

The second source of information is represented by the notational files that describe the notation associated to the objects of the library. These are the XML files described in Sect. 3.3 that are fed to the meta-stylesheets that produce the XslT stylesheets that render formal documents according to the usual mathematical notation. The task of automatically generating from these files also the productions of the disambiguating parser would be an easy task for a master student.

Finally, we have not so far addressed the behaviour of the system in the case where the disambiguating parser returns more than one well-typed term. Depending on the processing required on the terms, the system may either proceed in parallel on all the generated terms, or it can ask the user to choose the term (or the corresponding interpretation) she was interested in. In the latter case the system can identify all the choices that are still ambiguous, and present to the user the description d of the choices. The description should be accurate enough to allow the user to easily identify the wanted term.

To conclude, ambiguous parsing is a clear example of the kind of interactions that we foster between MKM tools and mathematical reasoning tools: search engines need to integrate parsers of ambiguous expressions; ambiguous parsers can be implemented on top of a proof assistant, and they can consult search engines and notational documents to perform the disambiguation.

Chapter 11

Tactics

Since the development of the first proof assistants based on the Curry-Howard isomorphism, it became clear that directly writing lambda-terms (henceforth simply called terms) was a difficult, repetitive, time-expensive and error-prone activity; hence the introduction of meta-languages to describe procedures that are able to automatically generate the low-level terms. Nowadays, almost all the proof assistants using term description of proofs have many levels of abstractions, i.e. meta-languages, to create the terms, with the only remarkable exception of the ALF family [7] in which terms can directly be manipulated without any over-standing level of abstraction.

In particular, there usually are at least two levels, that of tactics and that of the language in which the whole system and hence the tactics are written; once the tactics are implemented, to do a proof the user enters a sequence of tactic applications, called a script; the script is then executed to create the term encoding of the proof, which is type-checked by the proof assistant kernel. Writing a script interactively is much simpler than writing the term by hands; once written, though, it becomes impossible to understand a script without replaying it interactively. For this reason, we can hardly speak of the language of tactics as a high level language, even if it is "compiled" to the language of terms.

To avoid confusion, in the rest of this chapter we will avoid the term "proof", using only "script" and "term"; moreover, we will avoid also the terms "proof-checking" and "type-checking", replacing them with "retyping" to mean the type-checking of an already generated term; finally, we will use the term "compiling" to mean the execution of a script. Usually, compilation ends with the type-checking of the generated term; hence the choice of "retyping" for the type-checking of an already generated term.

A long term consequence of the introduction of tactics has been the progressive lowering of interest in terms. In particular, users of modern proof assistants such as Coq [32] may even ignore the existence of commands to show the terms generated by their scripts. These terms are usually very huge and quite unreadable, so they do not add any easily accessible information to the scripts. Hence implementors have lost interest in terms too, as long as their size and compilation time are small enough to make the system response-time acceptable. When this is not the case, it is sometimes possible to trade space with time using reflexive tactics, in which the potentiality of complex type-systems to speak about themselves are exploited: reflexive tactics usually lead to a significant reduction in the size of the terms, at the cost of an increased reduction time during retyping and compilation. Often, though, reflexive tactics cannot be used; moreover, reflexive tactics suffer of the same implementation problems as other tactics and so could be implemented in such a way to create huger than needed terms, even if asymptotically smaller than the best ones created without reflection.

The low level of interest in terms of the tactic developers often leads to naive implementations (“If it works, it is OK”) and this to low-quality terms, which:

1. are huger than needed because particular cases are not taken into account during tactic development,
2. require more time than needed for retyping due to their size,
3. are particularly unreadable because they don’t correspond to the “natural” way of writing the proof by hand.

To cope with the second problem, retyping is usually avoided allowing systems to reload saved terms without retyping them and using a checksum to ensure that the saved file has not been modified. This is perfectly reasonable accordingly to the traditional application-centric architecture of proof assistants in which you have only one tool integrating all the functionalities and so you are free to use a proprietary format for data representation.

In the HELM project we are interested in switching from the application-centric model to the newer content-centric one, in which information is stored in standard formats (that is, XML based) to allow different applications to work on the same set of data. As a consequence, term size really becomes an important issue, due to the redundancy of standard formats, and retyping is needed because the applications cannot trust each other, hence needing retyping and making retyping time critical. Moreover, since we are interested in producing descriptions in natural languages of formal proofs (see Chap. 3), we really care about the naturality and readability of the proofs: the quality (in terms of naturality and readability) of the proofs rendering heavily depends on the quality of terms, making also the third characteristic of low-quality terms a critical issue.

A totally different scenario in which term size and retyping time are critical is the one introduced by Necula and Lee [93] under the name Proof Carrying Code (PCC). PCC is a technique that can be used for safe execution of untrusted code. In a typical instance of PCC, a code receiver establishes a set of safety rules that guarantee the safe behavior of programs, and the code producer creates a formal safety proof that proves, for the untrusted code, adherence to the safety rules. Then, the proof is transmitted to the receiver together with the code and it is retyped before code execution. While very compact representation of the terms, highly based on type-inference and unification, could be used to reduce the size and retyping time [94], designing proof assistants to produce terms with a high level of quality is still necessary.

In the next section we formally define the notions of tactic and tactical. We also give an overview of our implementation of a tactic library. In order to judge the quality of our tactics, we introduce a particular class of metrics for tactics evaluation in Sect. 11.2. In Sect. 11.3 we consider the notion of tactic equivalence and we describe one of the bad habits of tactic developers, overkilling. We also provide and analyze a simple example of overkilling tactic and we describe a more complex concrete experience in fixing overkilling in the implementation of a reflexive tactic in the Coq system, analyzing the gain obtained with respect to term-size, retyping time and term readability. Sect. 11.4 is an overview of all the tactics and tacticals already implemented for GTOPLEVEL. The conclusions and comparison with related work follow as usual.

11.1 Tactics and Tacticals

Traditionally, tactics are defined as follows.

Definition 11.1 (Of tactic) *A* tactic *is a partial function that, given a goal* G *(that is, a local context plus a type to inhabit), returns a couple* (L, t) *where* $L = L_1, \ldots, L_n$ *is a (possible empty) list of proof-obligations (i.e. goals) and* t *is a function that, given a list* $l = l_1, \ldots, l_n$ *of terms such that* l_i *inhabits* L_i *for each* i *in* $1, \ldots, n$*, returns a term* $t(l)$ *inhabiting* G*.*

Nevertheless, to handle existential variables we need a slightly more complex definition: since goals are defined in a proof problem (or metasenv), we need to add a new argument to each tactic that is a metasenv. Moreover, since a tactic can also instantiate one or more existential variables, we also need to make it return an instantiation.

Notice that, according to the previous definition of tactic, a tactic can only act on the current open goal and, with the already discussed extension, on the metasenv. It can not modify parts of the proof that have already been developed. On the contrary, we prefer to allow tactics to inspect and modify all the parts of the proof under development. For instance, we want to admit a tactic that looks in the proof for subproofs of facts similar to the current goal; if a subproof matching the criteria is found, the subproof can be generalized and assumed as a lemma, that is used to prove the current goal and to replace the original proof of the similar goal.

Thus, we obtain the following alternative definition of tactic, that is also the definition our implementation is based on.

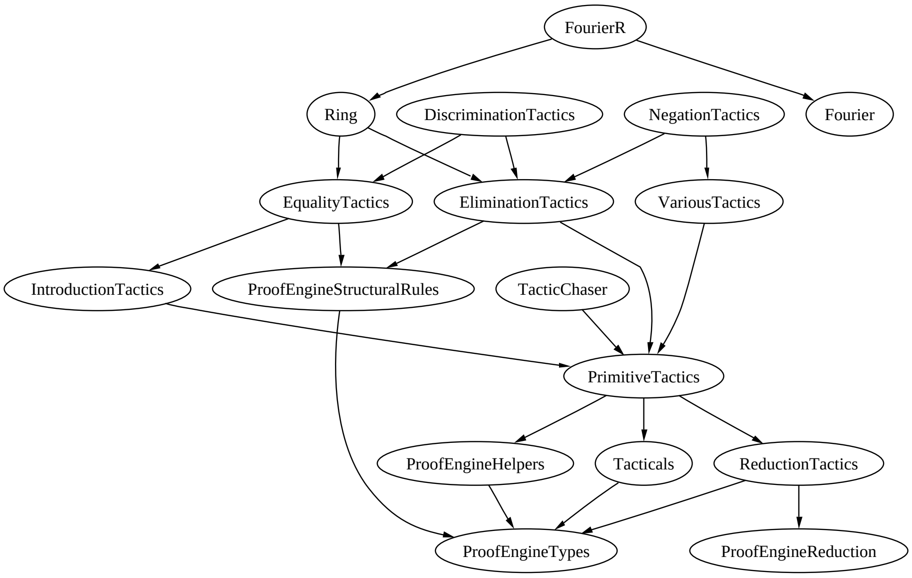

Figure 11.1: The `tactics` library.

Definition 11.2 (Of tactic, alternative definition) *A* tactic *is a partial function that, given a well-formed proof status* $(\mathsf{E}, \mathcal{P}, \mathsf{T}, \mathsf{t})$ *and a metavariable index* $\mathsf{i} \in \mathsf{Dom}(\mathcal{P})$*, returns a well-formed proof status* $(\mathsf{E}, \mathcal{P}', \mathsf{T}, \mathsf{t}')$.

Notice that the components E (the environment) and T (the statement of the theorem) of the input and output of the tactic must be the same. Thus a tactic is only allowed to change the (possibly incomplete) proof t' and the set of open goals and existential variables declarations $\mathcal{P}$. A tipical tactic — a tactic that can be defined according to the traditional definition — simply returns a proof problem $\mathcal{P}'$ that is obtained from the proof problem $\mathcal{P}$ in input by removing the current goal i and adding new metavariables declarations that are the new generated subgoals. It also returns a new proof t' that is obtained by the proof t in input by replacing every occurrence of the current goal i with a term that has in it occurrences of the new metavariables.

In our prototype tactics are implemented in the `tactics` library, whose modules organization is shown in Fig. 11.1. All the modules but `ProofEngineTypes`, `ProofEngineReduction` and `ProofEngineHelpers` are made only of tactics. The three modules which do not include any tactic are utility modules. In the rest of this section we will analyze only the utility modules. Sect. 11.4 is devoted to the description of all the tactics implemented in the other modules.

The module `ProofEngineTypes` provides no function, but it is largely the most important module, since it defines the type of a tactic. The following types are exported from `ProofEngineTypes`:

1. **proof:** an *incomplete proof* is a tuple made of an URI, a `metasenv` (also called proof problem), the body of the proof and its statement:

```
type proof =
 UriManager.uri * Cic.metasenv * Cic.term * Cic.term
```

2. **goal:** a goal is simply the index of a metavariable

```
type goal = int
```

3. **status:** the current proof status is the couple made by the current proof and the current goal:

```
type status = proof * goal
```

4. **tactic:** a tactic is a partial function that maps a proof status (i.e. proof + current goal) to a new proof. The process must close the current goal, may close any other goal and may open an unbound number of new goals (whose identifiers are also returned by the tactic):

```
type tactic = status:status -> proof * goal list
```

   In case of failure, a tactic must raise the `Fail` exception. It is important that a tactic does not raise any other exception, since the `Fail` exception is the one that is caught to perform backtracking.

5. **mk_fresh_name_type:** this is the type of functions that returns a fresh variable name in a given context. Beside the context, it has two additional arguments that can be used to choose a name that is meaningful to the user: a suggested base name (`Anonymous` is used to provide no base name) and the type of the variable (which can be used to determine the base name when no suggestion is given). An additional constraint that is not captured by the type definition is that the output cannot be `Anonymous`.

```
type mk_fresh_name_type =
 Cic.context -> Cic.name -> typ:Cic.term -> Cic.name
```

A tactics with an argument is implemented as a curried function whose domain is the type of the argument and whose codomain is the `tactic` type defined above. Tactics having several arguments are implemented in a similar way.

Higher order tactics, i.e. tactics having at least one argument of type `tactic`, are named *tacticals*. For example the tactical `try_tactics` takes as input a list of tactics L and behaves like the first one that does not fail (if such a tactic exists); it fails if every tactic in L fails. Its type is the following:

```
try_tactics:
  ProofEngineTypes.tactic list -> ProofEngineTypes.tactic
```

The module `ProofEngineReduction` includes several reduction, substitution and conversion functions which are not necessary in the proof engine kernel, but that are required to implement tactics:

- `alpha_equivalence` tests two terms for α-equivalence:

  ```
  val alpha_equivalence: Cic.term -> Cic.term -> bool
  ```

- `replace`, `replace_lifting` and `replace_lifting2` are three functions that simultaneously replace in a given term t several sub-terms s_i with other ones. The functions are parameterized over the equality function used to look for the s_i inside t by comparing with every s_i every sub-term of t.

  ```
  val replace :
    equality:(Cic.term -> 'a -> bool) ->
    what:'a list -> with_what:Cic.term list -> where:Cic.term ->
    Cic.term
  val replace_lifting :
    equality:(Cic.term -> Cic.term -> bool) ->
    what:Cic.term list -> with_what:Cic.term list -> where:Cic.term ->
    Cic.term
  val replace_lifting2 :
    int -> equality:(Cic.term -> Cic.term -> bool) ->
    what:Cic.term list -> with_what:Cic.term list -> where:Cic.term ->
    Cic.term
  ```

  The three functions are all implemented using structural recursion over the argument `where` and they differ only in their behavior when they recur under a binder: `replace` simply recurs; `replace_lifting` lifts all the `what` and `with_what` arguments; `replace_lifting2` lifts only the `with_what` arguments. Thus, for example, the following are the results of the application of the three functions to the arguments `what`=$x_{[\mathtt{Rel}(1)]}$, `with_what`= $y_{[\mathtt{Rel}(2)]}$ and `where`=$\lambda z.(z_{[\mathtt{Rel}(1)]} + x_{[\mathtt{Rel}(2)]})$ in the context $[y; x]$:

  - `replace what with_what where` $= \lambda z.(x_{[\mathtt{Rel}(2)]} + x_{[\mathtt{Rel}(2)]})$
  - `replace_lifting what with_what where` $= \lambda z.(z_{[\mathtt{Rel}(1)]} + y_{[\mathtt{Rel}(3)]})$
  - `replace_lifting2 what with_what where` $= \lambda z.(y_{[\mathtt{Rel}(3)]} + x_{[\mathtt{Rel}(2)]})$

The existence of the three variants may seem surprising. The existence of the `replace_lifting` function together with the other functions is justified by the fact that there are two different ways to identify a sub-term: the first one is to give a predicate that is satisfied only by the wanted sub-term; the second one is to give a physical pointer to the sub-term. In the former case the lifting is usually necessary, since the predicate needs to distinguish, say, an occurrence of the first variable in the context (which is $Rel(1 + d)$ where d is the number of binders in the path occurrence–root) from the occurrence of the last bound variable (which is simply $Rel(1)$). In the latter case, instead, the lifting must *not* be performed, since we need to preserve the physical structure of the formula in order to locate the wanted sub-term.

The difference between the `replace_lifting` and the `replace_lifting2` functions, instead, is justified by the fact that the `with_what` argument can be given either in the current context (and then needs to be lifted when passing a binder) or in the context where it will be substituted. Neither one of the two functions is easily definable in terms of the other one.

- `reduce` and `simpl` have the same signature as the kernel `whd` reduction function. The former implements full $\beta\delta\iota\zeta$-reduction. The latter is quite a complex reduction function that tries to fold the ζ-expanded recursive definitions after having performed $\beta\delta\iota\zeta$-reduction, backtracking the reduction when the definitions cannot be folded back. The exact algorithm will be given in Sect. 11.4 when we present the `Simpl` tactic.

The last utility module, `ProofEngineHelpers`, provides several utility functions to create and instantiate metavariables:

- `identity_relocation_list_for_metavariable` returns the relocation list $[Some\ Rel(1)\ ; \ldots ;\ Some\ Rel(n)]$ where n is the length of the parameter of the function, which is the environment where the variable is defined.

```
val identity_relocation_list_for_metavariable :
  'a option list -> Cic.term option list
```

- `new_meta` returns the first metavariable index that is higher than the index of every metavariable in the `proof` argument. Thus not only the result of `new_meta` is a fresh metavariable index, but also every higher index is.

```
val new_meta : proof:ProofEngineTypes.proof -> int
```

- `subst_meta_in_proof` and `subst_meta_and_metasenv_in_proof` are two utility functions used by primitive tactics to modify a proof status by replacing every occurrence of one or more metavariables with a given term. The metavariables environment is modified by removing the instantiated metavariable and by extending it with a new list of fresh metavariables declarations (which represent the new sub-goals). Due to the complexity of their interfaces, the two functions should be used only in the implementation of primitive tactics. Complex tactics (i.e. tactics that are implemented by composition of other lower level tactics) should never rely on these functions and they should use primitive tactics to perform the metavariables instantiation.

The most general function is `subst_meta_and_metasenv_in_proof`:

```
val subst_meta_and_metasenv_in_proof :
  ProofEngineTypes.proof ->
  int -> (Cic.term -> Cic.term) -> Cic.metasenv ->
  ProofEngineTypes.proof * Cic.metasenv
```

The first argument is the proof to be modified; the second argument is the index of the current goal. Together, the first two arguments form the current proof status. The third argument is the transformation function that performs the instantiation of the metavariables. The function is applied to any node in the proof term and in the metavariables environment. The fourth argument is the new metavariables environment of the proof before the instantiation of the metavariables. The output of the function is a couple whose first argument is the new proof status and whose second argument is the new metavariables environment *after* the metavariables substitution.

The function is quite unsafe, since it does not perform any consistency check over the new metavariables environment which is user provided. No check is performed either over the metavariables instantiation function, which can freely change the proof. This lack of checks can easily lead to programming errors. This is the reason why the function should be used only when implementing primitive tactics.

Most primitive tactics instantiate only one metavariable at a time (i.e. they are used to close only one goal, the current one). To handle this slightly simplified case, the most efficient `subst_meta_in_proof` function is provided:

```
val subst_meta_in_proof :
  ProofEngineTypes.proof ->
  int -> Cic.term -> Cic.metasenv ->
  ProofEngineTypes.proof * Cic.metasenv
```

Even though the arguments are similar, the simplified function is quite different from `subst_meta_and_metasenv_in_proof`: first of all, the third argument is no more an instantiation function, but simply the term that must replace the current goal; moreover, the fourth argument is not a new metavariable environment, but only a metavariable environment extension that will be appended to the current metavariable after having removed the current goal.

The last function exported from `ProofEngineHelpers` is `mk_fresh_name`:

```
val mk_fresh_name : ProofEngineTypes.mk_fresh_name_type
```

Tactics usually need to obtain several fresh variable names. When the role of the variables is clear to the user, the best approach is probably to ask the name to the user. Other times, instead, asking the name to the user is non-sensical: a typical situation is given by complex tactics that create big proofs made of several elimination steps, which introduce new fresh variables for each hypothesis. If the user is totally unable to predict which elimination will be made and in what context, she cannot determine a significant name for the new variables. Moreover, the user may not care at all about the proof generated by a tactic and the names of the bound variables in it. In all the cases where the tactic has to choose a fresh name without asking it to the user, the `mk_fresh_name` function can provide the name.

The choice between asking a fresh name to the user and automatically generating it depends on the context of application of a tactic. An elimination tactic, for example, should ask the names to the user when it is directly invoked by the user. Instead, it should generate the names when it is invoked in the implementation of a higher level tactic. Thus, all the tactics that we have implemented and that need fresh names are generalized over a parameter of type `ProofEngineTypes.mk_fresh_name_type`, that can be instantiated either with `mk_fresh_name_type` or with a GUI callback to make the user enter the name. The latter callback may also invoke `mk_fresh_name` to give the user a default choice, in case she is unable to figure out a suggestive name.

In Sect. 11.4 we will describe every tactic implemented in GTOPLEVEL. Before that, in the following two sections, we introduce the notion of *overkilling* and several metrics that can be used to compare several tactics implementation and to judge tactics enhancements. The second section ends with a case study, performed using the Coq system, where we improved a tactic implementation to make it less overkilling without breaking backward compatibility. Even though these sections are logically stand-alone (it describes material presented in [108]), we prefer to put them before the description of the implemented tactics since they provides a good vocabulary and some insights to understand both the typical tactics implementation difficulties and the possibility of enhancements.

11.2 From Metrics for Terms Evaluation to Metrics for Tactics Evaluation

The aim of this section is to show how metrics for term evaluation could induce metrics for tactic evaluation. Roughly speaking, this allows us to valuate tactics in terms of the quality of the terms produced. Even if we think that these kinds of metrics are interesting and worth studying, it must be understood that many other valuable forms of metrics could be defined on tactics, depending on what we are interested in. For example, we could be interested on compilation time, that is the sum of the time required to generate the term and the retyping time for it. Clearly, only the second component could be measured with a term metric while a good implementation of a tactic with respect compilation time metric could effectively decide to sacrifice term quality (and hence retyping time) to minimize the time spent to generate the term. The situation is very close to the one already encountered in compiler implementation, where there is always a compromise, usually user configurable, between minimizing compiling time and maximizing code quality.

The section is organized as follows: first we recall the definition of tactic and we introduce metrics on terms; then we give the definition of some metrics induced by term metrics on tactics.

Definition 11.3 (Of term metric) *For any goal* G*, a term metric* μ_G *is any function in* $\mathbb{N}^{\{t/t \textit{ inhabits } G\}}$. *Two important class of term metrics are functional metrics and monotone metrics:*

1. **Functional metrics:** *a metric* μ_G *is* functional *if for each term context (a term with a hole)* $C[]$ *and for all terms* t_1,t_2 *if* $\mu_G(t_1) = \mu_G(t_2)$ *then* $\mu_G(C[t_1]) = \mu_G(C[t_2])$*. An equivalent definition is that a metric* μ_G *is functional if for each term context* $C[]$ *the function* $\lambda t.\mu_G(C[\mu_G^{-1}(t)])$ *is well defined.*

2. **Monotone metrics:** *a metric* μ_G *is* monotone *if for each term context* $C[]$ *and for all terms* t_1,t_2 *if* $\mu_G(t_1) \leq \mu_G(t_2)$ *then* $\mu_G(C[t_1]) \leq \mu_G(C[t_2])$*. An equivalent definition is that a metric* μ_G *is functional if for each term context* $C[]$ *the function* $\lambda t.\mu_G(C[\mu_G^{-1}(t)])$ *is well defined and monotone.*

Typical examples of term metrics are the size of a term, the time required to retype it or even an estimate of its "naturality" (or simplicity) to be defined somehow; the first two are also examples of monotone metrics and the third one could probably be defined to be monotone. So, in the rest of this chapter, we will consider only monotone metrics, even if the following definitions also work with weaker properties for general metrics. Here, however, we are not interested in defining such metrics, but in showing how they naturally induce metrics for tactics.

Once a term metric is chosen, we get the notion of a best term (not unique!) inhabiting a goal:

Definition 11.4 (Of best terms inhabiting a goal) *the term* t *inhabiting a goal* G *is said to be a best term inhabiting* G *w.r.t. the metric* μ_G *when* $\mu_G(t) = \min\{\mu_G(t')/t' \textit{ inhabits } G\}$.

Using the previous notion, we can compare the behavior of two tactics on the same goal:

Definition 11.5 *Let τ_1 and τ_2 be two tactics both applicable to a goal G such that $\tau_1(G) = (L_1, t_1)$ and $\tau_2(G) = (L_2, t_2)$. We say that τ_1 is better or equal than τ_2 for the goal G with respect to μ_G if, for all l_1 and l_2 lists of best terms inhabiting respectively L_1 and L_2, $\mu_G(t_1(l_1)) \leq \mu_G(t_2(l_2))$ holds.*

Note that comparing in this way "equivalent" tactics (whose definition is made precise in the next section) gives us information on which implementation is better; doing the same thing on tactics that are not equivalent, instead gives us information about what tactic to apply to obtain the best proof.

A (functional) metric to compare two tactics only on a particular goal has the nice property to be a total order, but is quite useless. Hence, we will now define a bunch of different tactic metrics induced by term metrics that can be used to compare the behavior of tactics when applied to a generic goal. Some of them will be *deterministic* partial orders; others will be total orders, but will provide only a *probabilistic* estimate of the behavior. Both kinds of metrics are useful in practice when rating tactics implementation.

Definition 11.6 (Of locally deterministic better or equal tactic) *τ_1 is a locally deterministic (or locally uniform) better or equal tactic than τ_2 w.r.t. μ (and in that case we write $\tau_1 \leq_\mu \tau_2$ or simply $\tau_1 \leq \tau_2$), when for all goals G satisfying the preconditions of both tactics we have that τ_1 is better or equal than τ_2 w.r.t. the metric μ_G.*

Definition 11.7 (Of locally deterministic better tactic) *τ_1 is a locally deterministic (or locally uniform) better tactic than τ_2 w.r.t. μ (and in that case we write $\tau_1 <_\mu \tau_2$ or simply $\tau_1 < \tau_2$), when $\tau_1 \leq_\mu \tau_2$ and exists a goal G satisfying the preconditions of both tactics such that τ_1 is better (but not equal!) than τ_2 w.r.t. the metric μ_G.*

Definition 11.8 (Of locally probabilistic better or equal tactic of a factor K) *τ_1 is said to be a tactic locally probabilistic better or equal of a factor $0.5 \leq K \leq 1$ than τ_2 w.r.t. μ and a particular expected goals distribution when the probability of having τ_1 better or equal than τ_2 w.r.t. the metric μ_G is greater or equal to K when G is chosen randomly according to the distribution.*

The set of terms being discrete, you can note that a deterministically better or equal tactic is a tactic probabilistically better or equal of a factor 1.

To end this section, we can remark the strong dependence of the $\leq$ relation on the choice of metric μ, so that it is easy to find two metrics μ_1, μ_2 such that $\tau_1 <_{\mu_1} \tau_2$ and $\tau_2 <_{\mu_2} \tau_1$. Luckily, tough, the main interesting metrics, term size, retyping time and naturality, are in practice highly correlated, though the correlation of the third one with the previous two could be a bit surprising. So, in the following section, we will not state what is the chosen term metric; you may think as any of them or even at some kind of weighted mean.

11.3 Equivalent Tactics and Overkilling

We are now interested in using the metrics defined in the previous section to compare tactics implementation. Before doing so, though, we have to identify what we consider to be different implementations of the same tactic. Our approach consists in identifying every implementation with the tactic it implements and then defining appropriate notions of equivalence for tactics: two equivalent tactics will then be considered as equivalent implementations and will be compared using metrics.

Defining two tactics as equivalent when they can solve exactly the same set of goals generating the same set of proof-obligations seems quite natural, but is highly unsatisfactory if not completely wrong. The reason is that, for equivalent tactics, we would like to have the *property of substitutivity*, that is substituting a tactic for an equivalent one in a script should give back an error-free script[1]. In logical frameworks with dependent types, without proof-irrelevance and with universes as CIC though, it is possible for a term to inspect the term of a previous proof, behaving in a different way, for example, if the constructive proof of a disjunction is made proving the left or right side. So, two tactics, equivalent w.r.t. the previous definition, that prove $A \vee A$ having at their disposal an hypothesis A proving the first one the left and the second one the right part of the conjunction, could not be substituted one with the other if a subsequent term inspects the form of the generated proof.

In other terms, it seems quite reasonable to derive equivalence for tactics from the definition of an underlying equivalence for terms. The simplest form of such an equivalence relation is convertibility (up to proof-irrelevance) of closed terms and this is the relation we will use in this section and the following one. In particular, we will restrict ourselves to CIC and hence to $\beta\delta\iota$-convertibility[2]. Convertibility, though, is too a restrictive notion that does not take into account, for example, commuting conversions. Thus considering equivalence up to isomorphisms [40] could be an interesting scenario. Notice, however, that replacing convertibility with an alternative equivalence relation requires a major modification of the kernel of the proof assistant.

Definition 11.9 (Of terms closed in a local environment) *A term* t *is* closed *in a local environment* Γ *when* Γ *is defined on any free variable of* t.

Definition 11.10 (Of equivalent tactics) *Two tactics* τ_1 *and* τ_2 *are* equivalent *(and we write* $\tau_1 \approx \tau_2$*) when for each goal* $G = (\Gamma, T)$ *and for each list of terms closed in* Γ *and inhabiting the proof-obligations generated respectively by* τ_1 *and* τ_2*, we have that the result terms produced by* τ_1 *and* τ_2 *are* $\beta\delta\iota$*-convertible.*

[1]A weaker notion of substitutivity is that substituting the term generated by a tactic for the term generated by an equivalent one in a generic well-typed term should always give back a well-typed term.

[2]The Coq proof assistant introduces the notion of *opaque* and *transparent* terms, differing only for the possibility of being inspected. Because the user could change the opacity status at any time, the notion of convertibility we must conservatively choose for the terms of Coq is $\beta\delta\iota$-convertibility after having set all the definitions as transparent.

Once we have the definition of equivalent tactics, we can use metrics, either deterministic or probabilistic, to compare them. In particular, in the rest of this section and in the following one we will focus on the notion of deterministically overkilling tactic, defined as follows:

Definition 11.11 (Of overkilling tactics) *A tactic τ_1 is (deterministically) overkilling w.r.t. a metric μ when there exists another tactic τ_2 such that $\tau_1 \approx \tau_2$ and $\tau_2 <_\mu \tau_1$.*

Fixing an overkilling tactic τ_1 means replacing it with the tactic τ_2 which is the witness of τ_1 being overkilling. Note that the fixed tactic could still be overkilling.

The name overkilling was chosen because most of the time overkilling tactics are tactics that do not consider special cases, following a general algorithm. While in computer science it is often a good design pattern to prefer general solutions to ad hoc ones, this is not a silver bullet: an example comes again from compiler technology, where ad hoc cases, i.e. optimizations, are greatly valuable if not necessary. In our context, ad hoc cases could be considered either as optimizations, or as applications of Occam's razor to keep the simplest proof.

A Simple Example of Overkilling Tactic

A first example of overkilling tactic in system Coq is *Replace*, that works in this way: when the current goal is $G = (\Gamma, T)$, the tactic "Replace E_1 with E_2." always produces a new principal proof-obligation $(\Gamma, T\{E_2/E_1\})$ and an auxiliary proof-obligation $(\Gamma, E_1 = E_2)$ and uses the elimination scheme of equality on the term $E_1 = E_2$ and the two terms that inhabit the obligations to prove the current goal.

To show that this tactic is overkilling, we will provide an example in which the tactic fails to find the best term, we will propose a different implementation that produces the best term and we will show the equivalence with the actual one.

The example consists in applying the tactic to the case of E_1 and E_2 being convertible terms: the tactic proposes to the user the two proof-obligations and then builds the term as described above. We claim that the term inhabiting the principal proof-obligation also inhabits the goal and, used as the generated term, is surely smaller and quicker to retype than the one that is generated in the implementation; moreover, it is also as natural as the previous one, in the sense that the apparently lost information has simply become implicit in the reduction and could be easily rediscovered using type-inference algorithms as the one described in Coscoy's thesis [34]. So, a new implementation could simply recognize this special case and generate the better term.

We will now show that the terms provided by the two implementations are $\beta\delta\iota$-convertible. Each closed term in $\beta\delta\iota$-normal form inhabiting the proof that E_1 is equal to E_2 is equal to the only constructor of equality applied to a term convertible to the type of E_1 and to another term convertible to E_1; hence, once the principle of elimination of equality is applied to this term, we can first apply β-reduction and then ι-reduction to obtain the term inhabiting the principal proof-obligation in which E_1 has been replaced by E_2. Since E_1 and E_2 are $\beta\delta\iota$-convertible by hypothesis and for the congruence properties of convertibility in CIC, we have that the generated term is $\beta\delta\iota$-convertible to the one inhabiting the principal proof-obligation.

This example may seem quite stupid because, if the user is already able to prove the principal proof-obligation and because this new goal is totally equivalent to the original one, the user could simply redo the same steps without applying the rewriting at all. Most of the time, though, the convertibility of the two terms could be really complex to understand, greatly depending on the exact definitions given; indeed, the user could often be completely unaware of the convertibility of the two terms. Moreover, even in the cases in which the user understands the convertibility, the tactic has the important effect of changing the form of the current goal in order to simplify the task of completing the proof: this is the motivation of the user for applying the tactic, and also the justification of the naturality of the proof step.

The previous example shows only a very small improvement in the produced term and could make you wonder if the effort of fixing overkilling and more in general if putting more attention to terms when implementing tactics is really worth the trouble. In the next section we report another example describing a concrete experience of fixing a complex reflexive tactic in the Coq system that lead to really significant improvements in term size, retyping time and naturality.

11.3.1 Fixing Overkilling: a Concrete Experience

Coq provides a reflexive tactic called *Ring* to do associative-commutative rewriting in ring and semi-ring structures. The usual usage is, given the goal $E_1 = E_2$ where E_1 and E_2 are two expressions defined on the ring-structure, to prove the goal reducing it to proving $E_1' = E_2'$ where E_i' is the normal form of E_i. In fact, once obtained the goal $E_1' = E_2'$, the tactic also tries to apply simple heuristics to automatically solve the goal.

The actual implementation of the tactic by reflexion is quite complex and is described in [25]. The main idea is described in Fig. 11.2: first of all, an inductive data type to describe abstract polynomial is made available. On this abstract polynomial, using well-founded recursion in Coq, a normalization function named $\mathit{apolynomial\_normalize}$ is defined; for technical reasons, the abstract data type of normalized polynomials is different from the one of un-normalized polynomials. Then, two interpretation functions, named $\mathit{interp\_ap}$ and $\mathit{interp\_sacs}$ are given to map the two forms of abstract polynomials to the concrete one. Finally, a theorem named $\mathit{apolynomial\_normalize\_ok}$ stating the equality of the interpretation of an abstract polynomial and the interpretation of its normal form is defined in Coq using well-founded induction. The

$$\begin{tikzcd}
E^{I} \arrow[d, equal, "\text{apolynomial_normalize_ok}"'] & E^{II} \arrow[l, "\text{interp_ap}"'] \arrow[d, "\text{apolynomial_normalize}"] \\
E^{IV} & E^{III} \arrow[l, "\text{interp_sacs}"']
\end{tikzcd}$$

Figure 11.2: Reflexion in Ring

$$\begin{array}{rl}
E_1 = E_2 & \\
\| & \text{refl_equal (because of } \beta\delta\iota\text{-convertibility)} \\
(\text{interp_ap } E_1^{II}) = E_2 & \\
\| & \text{apolynomial_normalize_ok} \\
(\text{interp_sacs } E_1^{III}) = E_2 &
\end{array}$$

Figure 11.3: Ring implementation (first half)

above machinery could be used in this way: to prove that E^{I} is equal to its normal form E^{IV}, the tactic computes an abstract polynomial E^{II} that, once interpreted, reduces to E^{I}, and such that the interpretation of $E^{III} = (\mathit{apolynomial\_normalize} E^{II})$ could be shown to be equal to E^{IV} applying $\mathit{apolynomial\_normalize\_ok}$.

In Fig. 11.3 the first half of the steps taken by the Ring tactic to prove $E_1 = E_2$ are shown[3]. The first step is replacing $E_1 = E_2$ with $(\mathit{interp\_ap}\ E_1^{II}) = E_2$, justifying the rewriting using the only one constructor of equality due to the $\beta\delta\iota$-convertibility of $(\mathit{interp\_ap}\ E_1^{II})$ with E_1. The second one is replacing $(\mathit{interp\_ap}\ E^{II})$ with $(\mathit{interp\_sacs}\ E^{III})$, justifying the rewriting using $\mathit{apolynomial\_normalize\_ok}$.

Next, the two steps are done again on the right part of the equality, obtaining $(\mathit{interp\_sacs}\ E_1^{III}) = (\mathit{interp\_sacs}\ E_2^{III})$, that is eventually solved trying simpler tactics such as *Reflexivity* or left to the user.

The tactic is clearly overkilling, at least due to the usage of rewriting for convertible terms. Let's consider as a simple example the session in Fig. 11.4: in Fig. 11.5 the λ-term created by the original overkilling implementation of Ring is shown. Following the previous explanation, it should be easily understandable. In particular, the four rewritings are clearly visible as appli-

[3]Here E_1 stands for E^{I}.

```
Coq < Goal ``0*0==0``.
1 subgoal

  ================
     ``0*0 == 0``

Unnamed_thm < Ring.
Subtree proved!
```

Figure 11.4: A Coq session.

cations of $\mathsf{eqT\_ind}$, as the two applications of $\mathsf{apolynomial\_normalize\_ok}$ and the three usage of reflexivity, i.e. the two applications of $\mathsf{refl\_eqT}$ to justify the rewritings on the left and right members of the equality and the one that ends the proof.

Let's start the analysis of overkilling in this implementation:

Always overkilling rewritings: as already stated, four of the rewriting steps are always overkilling because the rewritten term is convertible to the original one due to the tactic implementation. As proved in the previous section, all these rewritings could simply be removed obtaining an equivalent tactic.

Overkilling rewritings due to members already normalized: it may happen, as in our example, that one (or even both) of the two members is already in normal form. In this case the two rewriting steps for that member could be simply removed obtaining an equivalent tactic as shown in the previous section.

Rewriting followed by reflexivity: after having removed all the overkilling rewritings, the general form of the λ-term produced for $E_1 = E_2$ is the application of two rewritings (E_1' for E_1 and E_2' for E_2), followed by a proof of $E_1' = E_2'$. In many cases, E_1' and E_2' are simply convertible and so the tactic finishes the proof with an application of reflexivity to prove the equivalent goal $E_1' = E_1'$. A smaller and also more natural solution is just to rewrite E_1' for E_1 and then proving $E_1' = E_2$ applying the lemma stating the symmetry of equality to the proof of $E_2 = E_2'$. The equivalence to the original tactic is trivial by $\beta\iota$-reduction because the lemma is proved exactly doing the rewriting and then applying reflexivity:

$$\begin{array}{l}\lambda A : \mathsf{Type}.\\ \quad \lambda x, y : A.\\ \quad\quad \lambda H : (x == y).\\ \quad\quad\quad (\mathsf{eqT\_ind}\ A\ x\ [x : A]a == x\ (\mathsf{refl\_eqT}\ A\ x)\ y\ H)\end{array}$$

In Fig. 11.6 is shown the λ-term created by the same tactic after having fixed all the overkilling problems described above.

```
Unnamed_thm < Show Proof.
LOC:
Subgoals
Proof:
(eqT_ind R
  (interp_ap R Rplus Rmult ``1`` ``0`` Ropp (Empty_vm R)
    (APmult AP0 AP0)) [r:R]``r == 0``
  (eqT_ind R
    (interp_sacs R Rplus Rmult ``1`` ``0`` Ropp (Empty_vm R)
      Nil_varlist) [r:R]``r == 0``
    (eqT_ind R
      (interp_ap R Rplus Rmult ``1`` ``0`` Ropp (Empty_vm R) AP0)
      [r:R]
       ``(interp_sacs R Rplus Rmult 1 r Ropp (Empty_vm R) Nil_varlist)
       == r``
      (eqT_ind R
        (interp_sacs R Rplus Rmult ``1`` ``0`` Ropp (Empty_vm R)
          Nil_varlist)
        [r:R]
         ``(interp_sacs R Rplus Rmult 1 r Ropp (Empty_vm R)
           Nil_varlist) == r`` (refl_eqT R ``0``)
        (interp_ap R Rplus Rmult ``1`` ``0`` Ropp (Empty_vm R) AP0)
        (apolynomial_normalize_ok R Rplus Rmult ``1`` ``0`` Ropp
          [_,_:R]false (Empty_vm R) RTheory AP0)) ``0``
      (refl_eqT R ``0``))
    (interp_ap R Rplus Rmult ``1`` ``0`` Ropp (Empty_vm R)
      (APmult AP0 AP0))
    (apolynomial_normalize_ok R Rplus Rmult ``1`` ``0`` Ropp
      [_,_:R]false (Empty_vm R) RTheory (APmult AP0 AP0))) ``0*0``
  (refl_eqT R ``0*0``))
```

Figure 11.5: The λ-term created by the original overkilling implementation

```
Unnamed_thm < Show Proof.
LOC:
Subgoals
Proof:
(sym_eqT R ``0`` ``0*0``
  (apolynomial_normalize_ok R Rplus Rmult ``1`` ``0`` Ropp
    [_,_:R]false (Empty_vm R) RTheory (APmult AP0 AP0)))
```

Figure 11.6: The λ-term created by the new implementation

A Quantitative Analysis of the Gain Obtained

Let's now try a quantitative analysis of the gain with respect to term size, retyping time and naturality, considering the two interesting cases of no member or only one member already in normal form[4].

Part I: Term Size.
Terms metric definition: given a term t, the metric $|.|$ associates to it its number of nodes $|t|$.

Notation: $|T|$ stands for the number of nodes in the actual parameters given to $\mathit{interp\_ap}$, $\mathit{interp\_sacs}$ and $\mathit{apolynomial\_normalize\_ok}$ to describe the concrete (semi)ring theory and the list of non-primitive terms occurring in the goal to solve. In the example in figures 11.5 and 11.6, $|T|$ is the number of nodes in [R Rplus Rmult "1" "0" Ropp (Empty_vm R)]. $|R|$ stands for the number of nodes in the term which is the carrier of the ring structure. In the same examples, $|R|$ is simply 1, i.e. the number of nodes in R.

Fig. 11.7 is a quantitative analysis of the improvement for the term size. While the overall space complexity of the terms generated by the new implementation is asymptotically equal to the one of the old implementation, all the constants involved are much smaller, but for the one of E'_2 (the two normal forms) that before was 0 and now is equal to 2. Is it possible to have goals for which the new implementation behaves worst than the old one? Unfortunately, yes. This happens when the size of the two normal forms E'_1 and E'_2 is far larger than $(E^{II}_1 + |T| + 1)$ and $(E^{II}_2 + |T| + 1)$. This happens when the number of occurrences of non-primitive terms is much higher than the number of non-primitive terms and the size of them is big. More formally, if m is the number of non-primitive terms, d the average size and n the number of occurrences, the new implementation creates bigger terms than the previous one if

$$n \log_2 m + md < nd$$

where the difference between the two members is great enough to hide the gain achieved lowering all the other constants. The logarithmic factor in the previous formula derives from the implementation of the variable maps as a tree and the representation of occurrences with the path inside the tree to retrieve the term.

To fix the problem, for each non-primitive term occurring more than once inside the normal forms, we can use a *let ... in* local definition to bind it to a fresh identifier; then we replace every occurrence of the term inside the normal forms with the appropriate identifier[5]. In this way, the

[4]If the two members are already in normal form, the new implementation simply applies once the only constructor of the equality to one of the two members. The tactic is also implemented to do the same thing also when the two members are not yet in normal forms, but are already convertible. We omit this other improvement in our analysis.

[5]This has not yet been implemented in Coq.

Original version:

$1 + (\vert E_1^{II}\vert + \vert T\vert + 1) + \vert E_2\vert + \vert E_1\vert +$	(I rewriting Left)
$1 + \vert E_1\vert +$	(justification)
$1 + (\vert E_1^{III}\vert + \vert T\vert + 1) + \vert E_2\vert + (\vert E_1^{II}\vert + \vert T\vert + 1) +$	(II rewriting Left)
$(\vert E_1^{II}\vert + \vert T\vert + 1) +$	(justification)
$1 + (\vert E_2^{II}\vert + \vert T\vert + 1) + (\vert E_1^{III}\vert + \vert T\vert + 1) + \vert E_2\vert +$	(I rewriting Right)
$1 + \vert E_2\vert +$	(justification)
$1 + (\vert E_2^{III}\vert + \vert T\vert + 1) + (\vert E_1^{III}\vert + \vert T\vert + 1) +$ $(\vert E_2^{II}\vert + \vert T\vert + 1) +$	(II rewriting Right)
$(\vert E_2^{II}\vert + \vert T\vert + 1) +$	(justification)
$1 + \vert E_1\vert =$	(reflexivity application)
$4\vert E_1\vert + 2\vert E_2\vert + 3\vert E_1^{II}\vert + 3\vert E_2^{II}\vert + 3\vert E_1^{III}\vert + \vert E_2^{III}\vert +$ $10\vert T\vert + 17$	**Total number**

New version, both members not in normal form:

$1 + \vert R\vert + \vert E_1\vert + \vert E_2'\vert +$	(Rewriting Right)
$1 + \vert T\vert + \vert E_2^{II}\vert +$	(justification)
$1 + \vert R\vert + \vert E_2'\vert + \vert E_1'\vert + \vert E_2\vert +$	(Symmetry application)
$1 + \vert T\vert + \vert E_1^{II}\vert =$	(justification)
$2\vert E_1\vert + \vert E_2\vert + \vert E_1^{II}\vert + \vert E_2^{II}\vert + 2\vert E_2'\vert + 2\vert T\vert +$ $2\vert R\vert + 4$	**Total number**

New version, only the first member not in normal form:

$1 + \vert R\vert + \vert E_1\vert + \vert E_2'\vert +$	(Rewriting)
$1 + \vert T\vert + \vert E_2^{II}\vert =$	(justification)
$\vert E_1\vert + \vert E_2'\vert + \vert E_2^{II}\vert + \vert T\vert + \vert R\vert + 2$	**Total number**

Figure 11.7: Term size improvement

above inequality becomes

$$n \log_2 m + md < n + md$$

that is never satisfied.

Here it is important to stress how the latest problem was easily overlooked during the implementation and has been discovered only during the previous analysis, strengthening our belief in the importance of this kind of analysis for tactic implementations.

In the next two paragraphs we will consider only the new implementation with the above fix.

Part II: Retyping Time.
Terms metric definition: given a term t, the metric $|.|$ associates to it the time $|t|$ required to retype it.

Due to lack of space, we will omit a detailed analysis such as the one given for terms size. Nevertheless, we can observe that the retyping time required is surely smaller because all the type-checking operations required for the new implementation are already present in the old one, but for the type-checking of the two normal forms, that have lesser complexity than the type-checking of the two abstract normal forms, and the *let ... in* definitions that have the same complexity as the type-checking of the variable map. Moreover, the quite expensive operation of computing the two normal forms is already done during proof construction.

In section 11.3.1 we present some benchmarks to give an idea of the real gain obtained.

Part III: Naturality.
The idea behind the *Ring* tactic is to be able to prove an equality showing that both members have the same normal form. This simply amounts to show that each member is equal to the same normal form, that is exactly what is done in the new implementation. Indeed, every step that belonged to the old implementation and has been changed or removed to fix overkilling used to lead to some unnatural steps:

1. The fact that the normalization is not done on the concrete representation, but passing through two abstract ones that are interpreted on the concrete terms is an implementation detail that was not hidden as it should be.

2. Normalizing a member of the equality that is already in normal form, is illogical and so unnatural. Hence it should be avoided, but it was not.

3. The natural way to show $A = B$ under the hypothesis $B = A$ is just to use the symmetric property of equality. Instead, the old implementation rewrote B with A using the hypothesis and proved the goal by reflexivity.

	Term size	Re-typing time	Parsing time
Old implementation	20.27Mb	4.59s	2.425s
New implementation	12.99Mb	2.94s	2.210s
Percentage reduction	35.74%	35.95%	8.87%

Table 11.1: Some benchmarks

4. Using local definitions (*let ... in*) as abbreviations rises the readability of the proof by shrinking its size removing subexpressions that are not involved in the computation.

Some Benchmarks

To understand the actual gain in term size and retyping time on real-life examples, we have done some benchmarks on the whole set of theorems in the standard library of Coq that use the Ring tactic. The results are shown in table 11.1.

Term size is the size of the term disk dumps. Re-typing time is the user time spent by Coq in proof-checking already parsed terms. The reduction of term size implies also a reduction in Coq parsing time, that is difficult to compute because Coq files do not hold single terms, but whole theories. Hence, the parsing time shown is really the user time spent by Coq to parse not only the terms we are interested on, but also all the terms in their theories and the theories they depend on. So, this last measure greatly under-estimates the actual gain.

Every benchmark has been repeated 100 times under different load conditions on a 600Mhz Pentium III bi-processor equipped with 256Mb RAM. The timings shown are mean values.

The lesson learned by this concrete experience is that the more complex the tactic, the greater the probability of it being overkilling and the benefits of removing that overkilling.

In the next section we describe all the implemented tactics for GTOPLEVEL, underlying their overkilling aspect when present.

11.4 The Implemented Tactics

The actual list of tactics and tacticals implemented is not particularly interesting by itself, since all the tactics are pretty standard and they have already been implemented in other proof assistants such as Coq. We list them here very briefly only to give an idea of the status of the prototype.

The primitive tactics, that are the most difficult, have been implemented by Prof. Asperti and the author of this dissertation. The large majority of the remaining tactics have been implemented by undergraduate or master students.

To implement a tactic, it is not important to have a deep knowledge of the underlying theory. On the other hand, the developer needs to know quite well the interfaces provided by the main modules of the proof assistant. Indeed, this is the main difficulty that a tactic developer must face when he tries to extend the Coq system: Coq is made of lots of modules that provide a lot of functions; most of the functions either expose implementation details or they are logically redundant, existing only for the sake of optimizations. Thus it is difficult to quickly reach a level of knowledge sufficient for development. On the contrary, our effort of minimization of the complexity of the module interfaces seems to be fruitful: so far no student with just a background in undergraduate logic and basic theoretical computer science has faced serious difficulties in understanding the interfaces.

Tacticals

The following tacticals have been implemented:

`try_tactics`

Input: a list `l` of tactics and a callback function `f`.

Behaviour: first of all, all the tactics in `l` are tried; if more than one tactic succeeds, then the `f` is invoked over the list of tactics that succeeds. The value returned by `f` is the index of the tactic that is applied. The tactical fails if no tactic in `l` can be applied successfully.

Note: to reduce overkilling, the callback function can be implemented to choose the tactic that produces the best term among the tactics that solve the goal.

`thens`

Input: a tactic `t` and a list of tactics `l`.

Behaviour: `t` is applied. If `t` is successful and generates $|l|$ new goals, then each tactic in `l` is applied to the corresponding new goal. If `t` or any tactic in `l` fails, or if `t` does not generate $|l|$ new goals, then the tactical fails.

`then_`

Input: two tactics `t` and `c`.

Behaviour: `t` is applied. Then `c` is applied to each new goal generated by `t`. The tactical fails if `t` fails or if any application of `c` fails.

`repeat_tactic`

Input: a tactic `t`.

Behaviour: t is applied once. If it fails, the tactical fails. Otherwise t is repeated again over each goal generated until it can not be applied any more.

do_tactic

Input: an integer n and a tactic t.

Behaviour: when n is 0, the tactical succeeds; otherwise t is applied once and $(\mathrm{do\_tactic}(n-1)t)$ is applied on each generated goal. If t fails, the tactical fails.

try_tactic

Input: a tactic t.

Behaviour: the tactical behaves as t if t succeeds. Otherwise it succeeds without changing the proof status.

solve_tactics

Input: a tactic t.

Behaviour: the tactical behaves as t if t succeeds and no new goal is opened. Otherwise the tactical fails.

Tactics Dealing with Structural Rules

These tactics change the current goal by applying a structural rule of the sequent calculus:

clearbody

Input: none.

Behaviour: a definition into the context of the current goal is weakened into a declaration, if this does not break well-formedness of the goal.

clear

Input: none.

Behaviour: a declaration into the context that does not occur in the current goal is dropped. The correctness of the operation is granted by the thinning lemma of the calculus.

Reduction Tactics

whd_tac

Input: a subterm[6] t of the current goal.

Behaviour: t is replaced with its weak head normal form.

reduce_tac

Input: a subterm t of the current goal.

Behaviour: t is replaced with its normal form.

simpl_tac

Input: a subterm t of the current goal.

Behaviour: t is replaced with its simplified form. The simplified form is computed using the following algorithm:

1. Perform $\beta\iota\zeta$-head-reduction until a δ-redex is obtained.
2. Attempt δ-reduction. If the residual is the application of a mutual recursive definition that is λ-abstracted w.r.t. zero or more variables and if the recursive definition can be ι-reduced, then reduce it and apply the whole algorithm again to the new redex; then apply step 3 to the result of the recursive simplification. Otherwise, if the recursive definition cannot be reduced, then do not even perform the original δ-reduction, and return the δ-redex. Otherwise, if the δ-residual is not the application of a mutual recursive definition that is λ-abstracted w.r.t. zero or more variables, then reduce it and return directly the result, without performing the third step of the algorithm.
3. Fold the application of the constant to the arguments that did not change at every iteration, i.e. to the actual arguments of the λ-abstractions that preceded the application of the function mutually defined.

fold_tac

Input: a term t and a reduction function f.

Behaviour: f is applied to t to compute its normal form. The normal form is then searched in the current goal and it is substituted with t.

[6] In the rest of this section, "a subterm of the current goal" and "a subterm of the current sequent" means a physical pointer to a subterm of the current value. In our implementation, the physical pointer is generated by means of semantic selection[92]: the graphical user interface allows to highlight using the mouse only subexpressions that correspond to valid subterms; the selection associated to the highlighted expression is a physical pointer to the corresponding subterm. The physical pointer allows to disambiguate between different occurrences of a term.

Primitive Tactics

apply_tac

Input: a term t.

Behaviour: if the conclusion of the type of t is unifiable with the conclusion of the current goal, then the goal is closed by applying t, and new goals that ask to prove the premises in the type of t are generated.

exact_tac

Input: a term t.

Behaviour: It solves the current goal if the type of t is convertible with the current goal conclusion.

intros_tac

Input: a list of identifiers and a callback to generate identifiers.

Behaviour: all the implications and universal quantifications in the conclusion of the current goal are assumed. The identifiers for the new set of hypotheses are taken from the list of identifiers. If the provided identifiers are not sufficient, fresh identifiers are automatically generated using the callback function.

cut_tac

Input: a term t.

Behaviour: two new goals are generated. The first one asks to prove t. The second one asks to prove the old goal under the additional hypothesis t.

letin_tac

Input: a term t, an optional identifier and a callback function to generate a fresh identifier.

Behaviour: a new local definition whose definiens is t is assumed in the context. The definiendum is either the provided identifier or a fresh identifier generated by the callback if none is provided.

change_tac

Input: a term t.

Behaviour: the conclusion of the current goal is substituted with t if the two terms are convertible.

Introduction Tactics

constructor_tac

Input: a callback function `f`.

Behaviour: if the goal conclusion is not an inductive type or an application of an inductive type, it fails. Otherwise it tries to apply the constructors of the inductive type. If more than one constructor can be applied, the list of all the possible choices are given in input to `f`, that must return the constructor to apply.

exists_tac Synonym of `constructor_tac`. Intended to be used for proving existential statements.

split_tac Synonym of `constructor_tac`. Intended to be used for proving conjunctions.

left_tac Synonym of `constructor_tac`. The callback function always chooses the first constructor. Intended to be used for proving disjunctions.

right_tac Synonym of `constructor_tac`. The callback function always chooses the second constructor. Intended to be used for proving disjunctions.

Elimination Tactics

elim_type_tac

Input: a term `t`.

Behaviour: first of all `(cut_tac t)` is applied. Then the new hypotheses introduced in the main goal are eliminated.

decompose_tac

Input: a term `t` and a callback function `f`.

Behaviour: the type of `t` is analyzed to identify the maximal subtree made of applications of inductive types. Then the list of inductive types found is fed to `f`, that must return a sublist of its input. Then a sequence of eliminations, simplifications and introductions are issued to recursively decompose the type of `t` by repeating the elimination of the head of the current goal, the simplification of the goals obtained and by introducing as many new hypotheses as possible, before issuing the recursive call over the hypotheses introduced. Recursion is not performed if the type of the hypotheses introduced does not belong to the list returned by `f`.

Equality Tactics

rewrite_tac

Input: a term t whose type is an equality.

Behaviour: the left hand side of the equality is replaced with the right hand side in the current goal conclusion.

Notes: very often the term t is the application of a lemma that states the equality. One possibility to reduce the size of the term generated by the rewriting step is to automatically generate once and for all a new lemma that corresponds to the instantiation of the elimination lemma of equality. For instance, given a lemma L that states that $\forall n, m : nat. n + m = m + n$, we can automatically generate a new lemma $rewrite_L$ that states that $\forall n, m : nat. \forall P : nat \to \textbf{Prop}. (P\ (n + m)) \to (P\ (m + n))$. The new lemma is provable by definition of Leibniz equality. Its proof is simply the application of the elimination principle of Leibniz equality to P, $n + m$, $m + n$ and the application of L to n and m.

This simple technique for reducing the size of the term is already quite effective, since n and m occur only once in the generated proof term, whereas they occur twice if we do not adopt the trick. For instance, when we deal with equations, the gain is remarkable since we can avoid writing twice a long mathematical expression.

The technique can be applied also to the next three tactics.

rewrite_simpl_tac It is the composition of `rewrite_tac` with simplification.

rewrite_back_tac

Input: a term t whose type is an equality.

Behaviour: the right hand side of the equality is replaced with the left hand side in the current goal conclusion.

rewrite_back_simpl_tac It is the composition of `rewrite_back_tac` with simplification.

replace_tac

Input: a term t_1 and a subterm t_2 of the current goal.

Behaviour: All the occurrences of t_1 in the current goal are replaced with t_2, and a new goal $t_1 = t_2$ is generated.

reflexivity_tac

Input: none

Behaviour: it tries to solve a goal whose conclusion is an equality by applying the reflexive property.

symmetry_tac

Input: none

Behaviour: it swaps the left hand side with the right hand side of the current goal conclusion.

transitivity_tac

Input: a term t.

Behaviour: if the current goal conclusion is an equality $t_1 = t_2$, then it replaces the current goal with two new equalities $t_1 = t$ and $t = t_2$.

Various Tactics

assumption_tac

Input: a callback function f.

Behaviour: it tries to solve the current goal by applying all the assumptions in the goal context. If more than one assumption can solve the goal, the list of all the assumptions that solve the goal is passed to f, that must return the index of the hypothesis to apply.

generalize_tac

Input: a list of subterms of the current goal conclusion and a callback function to generate a fresh identifier.

Behaviour: first of all the tactic checks whether all the subterms are convertible. Then a new identifier H is generated using the call back function. Finally the current goal conclusion C is reduced to the new goal $\forall H : T.C'$ where T is the type of the subterms in the list and C' is obtained by substituting in C all the subterms in the list with H.

Negation Tactics

absurd_tac

Input: a term t.

Behaviour: the current goals is reduced to proving both t and $\neg t$.

contradiction_tac

Input: none.

Behaviour: the current goal is reduced to proving `False`.

Discrimination Tactics

`injection_tac`

Input: a term t whose type is an equality $t_1 = t_2$ where the type of t_i is an inductive type.

Behaviour: the tactic is based on the injectivity of every constructor of an inductive type. The only possibility for two inhabitants of an inductive type to be equal is that both terms start with the same constructor, and that the arguments of the constructors are pairwise equal. The tactic compares the two terms t_1 and t_2 to find the maximal common prefix made of equal constructors only. For instance, the maximal common prefix of `(Node Empty (Node Empty Empty))` and `(Node (Node Empty Empty) (Node (Node Empty Empty) Empty))` is `(Node` ☐ `(Node` ☐ `Empty))` where ☐ is a placeholder for a missing term. Then it reduces the goal to a new one obtained by adding to the context a new hypothesis for each term placeholder ☐, that states the equality of the terms that match the placeholder in t_1 and t_2. The tactic fails if t_1 and t_2 are syntactically equal.

Note: the current implementation that is very close to the one of Coq can be improved. Indeed, to peel off the outermost constructor, a big term (a projection) based on case analysis is generated. Term size can be decreased by automatically generating ad hoc injection lemmas for each constructor of each inductive type.

`discriminate_tac`

Input: a term t whose type is an equality $t_1 = t_2$ where the type of t_i is an inductive type.

Behaviour: the tactic is based on the following property: if k_1 and k_2 are different constructors of an inductive type, then they are provably different, as it is their application to any equal or different list of arguments. The tactic compares the two terms t_1 and t_2 to find the maximal common prefix made of equal constructors only. Then it looks for a couple of subterms of t_1 and t_2 that match one term placeholder in the prefix and whose head subterms are different constructors. If none is found, the tactics fails. Otherwise the goal is solved.

Notes: the current implementation that is very close to the one of Coq can be improved. Indeed, to peel off the outermost constructor, a big term (a sort of characteristic function in **Prop**) based on case analysis is generated. Term size can be decreased by automatically generating ad hoc discrimination lemmas for each couple of constructors of each inductive type.

Ring and Fourier Tactics

`ring_tac`

Input: none.

Behaviour: it solves the current goal if the conclusion is an equality of two expressions over real numbers that are provably equal using only the axioms of the ring of the real numbers.

Notes: techniques to reduce the size and retyping time of this tactic have already been described in this chapter.

`fourier_tac`

Input: none.

Behaviour: it solves linear inequations on real numbers using Fourier's method. The implementation is heavily based on the code of the homonymous tactic for the Coq system, developed by Loïc Pottiers.

Notes: the current implementation is highly overkilling. The problem is inherited from the original implementation in Coq. A precise analysis of the sources of overkilling and the relative solutions, similar to that already done for the ring tactic, is planned for the next future.

11.4.1 Conclusions

Naive ways of implementing tactics lead to low quality terms that are difficult to inspect and process. To improve the situation, we have shown how metrics defined for terms naturally induce metrics for tactics and tactics implementation and we advocate the usage of such metrics for tactics evaluation and implementation. In particular, metrics could be used to analyze the quality of an implementation or could be used at run time by a tactic to choose what is the best way to proceed.

To safely replace a tactic implementation with another one, it is important to define when two tactics are equivalent, i.e. when they generate equivalent terms. The more natural equivalence relation for terms that can be adopted is $\beta\delta\iota$-convertibility, but in many situations it seems too strong. Weaker forms of term equivalence and the equivalence relations they induce on tactics have not been studied yet. Notwithstanding this, it already seems that proof-irrelevance, η-conversion and commuting conversions are interesting ingredients for the definition of a suitable equivalence relation.

Several tactics have already been implemented for GTOPLEVEL, but only in a few cases the implementation has been optimized to reduce overkilling. Indeed, detecting and removing overkilling is a long task, especially for common sequences of tactics, that can vary depending on the user. Only using the prototype for a long period of time it will be possible to detect overkilling patterns and solutions, that could consist in constraining the user by providing her only with more high level tactics, hiding the existing low level ones.

Chapter 12

A Graphical Interface for Proof Objects Manipulation

In the previous three chapters we described the first three architectural layers of GTOPLEVEL. The first level is the kernel, the only critical part of the system, responsible for managing the well-formed environment of well-formed definitions and declarations. The second level performs unification, refinement and ambiguous parsing. These are all non critical activities, since a bug in these modules will be detected by the kernel once the non well-typed definition will try to enter the environment. The second level completes the set of basic logical tools — substitution, convertibility, type-checking, unification and refinement — that are required to implement all the functionalities of GTOPLEVEL that are not already provided by the knowledge management tools of MoWGLI. Finally, the third layer consists in the implementation of the various tactics of the GTOPLEVEL, that constitute a subset of the Coq tactic set. The implementation of tactics relies both on the functionalities provided by the previous two layers and on the SEARCH-ENGINE functionalities described in Chap. 5.

In this chapter we complete our description of GTOPLEVEL by sketching the functionalities directly provided to the user and the organization of the graphical user interface.

12.1 Requirements

We briefly recall the requirements for GTOPLEVEL that directly affect the user interface (see also Sect. 7.1).

Req1. GTOPLEVEL must be designed according to the document-centric model;

Req2. it must be able to consult and contribute to the distributed library of CIC mathematical documents of the HELM and MoWGLI projects;

Req3. it must reuse and integrate as much as possible the services developed in HELM and MoWGLI;

Req4. remote services must be implemented as Web services;

Req5. the system should constraint the user as much as possible in such a way that only reasonable proofs are generated; when this is not possible, the system should help the user in improving parts of the generated proofs.

The first requirement represents the greatest difference with the Coq system. The Coq system is definitely application-centric: the only product of the interaction with the system is the script, that is just a registration of the sequence of commands given to the system. To interpret and reconstruct the meaning associated to the script, the user must replay the script using Coq itself, or it must (dynamically) augment it with informations obtained by the execution in Coq to be able to use an external tool such as PCoq or CtCoq to understand it [115, 21, 20].

All the systems based on scripts were born as application centric systems. Recently, though, there has been a number of proposals [117, 121, 16, 59] to adopt a verbose set of tactics, similar to the commands of the Mizar tool, that completely describe the steps that the system will take. The system limits itself to checking the correctness of the steps and to filling in the "obvious" missing steps. The systems that adopt this approach are someway borderline: they can be considered document centric as soon as there exists a clear specification of the algorithm adopted to fill in the missing steps. Whenever this specification does not exist, the produced document can be checked only by the system that produced it, making the system essentially application centric. This is at least the case of the Mizar system itself.

The alternative to adopting the scripts are the systems that allow — somebody would say force — the user to interact with the underlying proof object. To the author knowledge, only a few systems have tried this approach seriously. The main ones are those of the Alfa family [7], where the user is shown the proof object with metavariables therein and she can select a metavariable to refine and then provide a replacement for it. The proof object can be presented in several ways: as a natural deduction tree, in natural language (thanks to the interfacing with Ranta's Grammatical Framework [56]), as λ-terms. As soon as the format used by these systems to save the proof is made public, these systems can definitely be characterized as document-centric systems, even if no third parties tools have been developed to manipulate the produced document.

According to Req1, we are interested in a document centric tool, where the user can constantly interact with the proof developed so far. Thanks to the straightforward integration with the rendering tools developed in HELM and MoWGLI, we can require that proof must be presented in natural language, without forcing any non trivial implementation effort. Semantic selection [92] must allow the user to select well-formed subexpressions of the proof-object, and apply transformations over them. For instance, we require the system to provide at least a *local undo* function, that replaces the selected sub-proof with a new metavariable.

Since having to instantiate metavariables by hand without any form of automation is annoying and not very effective, the graphical interface must provide the user with some of the developed tactics (but no tactical). The requirement Req5 is our motivation for constraining the set of tactics directly available to the user. Indeed, several tactics should be considered too low-level, since they are useful building blocks for other tactics, but they may be easily combined to create λ-terms that do not have a clearly recognizable shape that allows to interprete them in natural language. For instance, we require any application of an elimination principle to be followed by the introduction of the generated implications. This avoids the case of several elimination principles followed by introduction only at the end, that generates an unnatural pattern in the λ-term (a commuting conversion redex) that yields an unnatural presentation in natural language.

One of the problems of directly interacting with the proof in natural language is that it is not always easy to understand which hypotheses belong to the scope of one metavariable [1], nor to understand which hypotheses have been restricted by unification. Thus we require also the possibility of viewing the sequent associated to the current goal. We also require semantic multiple selection over the sequent, in order to easily specify the parameters of the tactics, avoiding a complex textual syntax to identify subterms (as the `Pattern` tactic of Coq). Thanks to the rendering technology of HELM, the latter requirements can immediately be fulfilled.

We will achieve the second part of Req5 by allowing the user to directly manipulate the proof object: the user is constantly aware of the proof that is generated by the tactics, and she can choose to change the order of applications of the tactics to obtain the proof she has in mind. Otherwise, she can just proceed quickly using the tactics and locally undo the generated parts of the proof that can be improved by hand. Moreover, we can also implement additional operations on the proof object, similar to those presented in [82] and [102], to rearrange the proof by applying commuting conversions to some parts of the proof, or by "taking out" and making lemmas of interesting subproofs. All these operations can be implemented more or less easily on top of semantic selection and direct rendering of the proof object.

Requirements 3 and 4 are extremely easy to enforce: indeed several Web services implemented in HELM and MoWGLI have already been exploited in the implementation of the first three layers of the proof engine. Among them, there is the GETTER for the management of the distributed library and the SEARCH-ENGINE for resolving identifiers to URIs and for finding applicable theorems. The stylesheets developed to render the proof objects in natural language, combined with a widget to render MathML Presentation, already provide a perfect interface for rendering mathematical expressions and proofs, also enabling semantic selection and folding/unfolding of subexpressions. The SEARCH-ENGINE and the tool for generating and inspecting the dependency graph of the objects in the library are natural candidates for inclusion in GTOPLEVEL. They provide longly missed functionalities that are currently not available to the mainstream proof assistants, with the only recent exception of the Mizar system, whose development team has been interested in mathematical knowledge management topics since the beginning.

To test the modularity of GTOPLEVEL and the architecture based on Web services, we are interested in interfacing the proof engine with remote mathematical reasoning tools. This part has not been fully developed so far. A promising testbench, though, has been implemented by Stefano Zacchiroli in his master thesis [124]. Zacchiroli has implemented H-BUGS, that is a network of distributed agents (called *tutors*) that try to automatically advance in the current proof, proposing asynchronous suggestions to the user. H-BUGS was modelled on the pre-existing ΩMEGA-Ants system, developed for the ΩMEGAproof planner [19, 18]. The main difference between ΩMEGA-Ants and H-BUGS is that the latter adopts standard Web services technology, whereas the former, developed before the standardization of Web services, is based on a proprietary solution. A description of H-BUGS can be found in the research paper [111]. The main achievement of the work has been proving that integration with the XML technologies easily allows both to quickly develop new tutors and to extend the system functionalities by interfacing it with more and more remotely running Web services.

To fulfill Req2, we require GTOPLEVEL to be able to submit completed proofs to the library of HELM, automatically publishing them over the Web, advertising the existence of the new proofs to a GETTER and extracting and publishing the metadata required by the SEARCH-ENGINE. We also require the system to support browsing the existing distributed library of HELMand modifying existing results by locally undoing parts of the proofs, redoing them and saving the new proof with a new URI.

12.2 Implementation

GTOPLEVEL has been implemented as a Gtk application[1]. A screenshot of the main window can be seen in Fig. 12.1. The widget on the left hand side shows an unfinished proof. Indeed, the widget is an instance of GtkMathView [98], a Gtk widget for rendering MathML Presentation developed by Luca Padovani. What is shown in the screenshot is the rendering of the proof obtained by applying the stylesheets that have been developed in the MoWGLI project.

The widget allows several form of interactions. First of all it is possible to click on a subproof to collapse (fold) or expand (unfold) it. Thus proofs can be inspected at different level of details. Moreover, it allows *semantic (multiple) selection*, i.e. it allows selection only of expressions that are the rendering of well-formed subterms of the proof object. The feature can be seen on the top window on the right, that shows the current goal. The two subexpressions $1 + n_0$ and $n_0 + 0$ are highlighted in cyan since they are currently selected. It would have not been possible, for instance, to select $1 + n_0$ on the right hand side, since the right hand side is the rendering of the mathematical expression $1 + (n_0 + 0)$.

[1] `http://www.gtk.org`

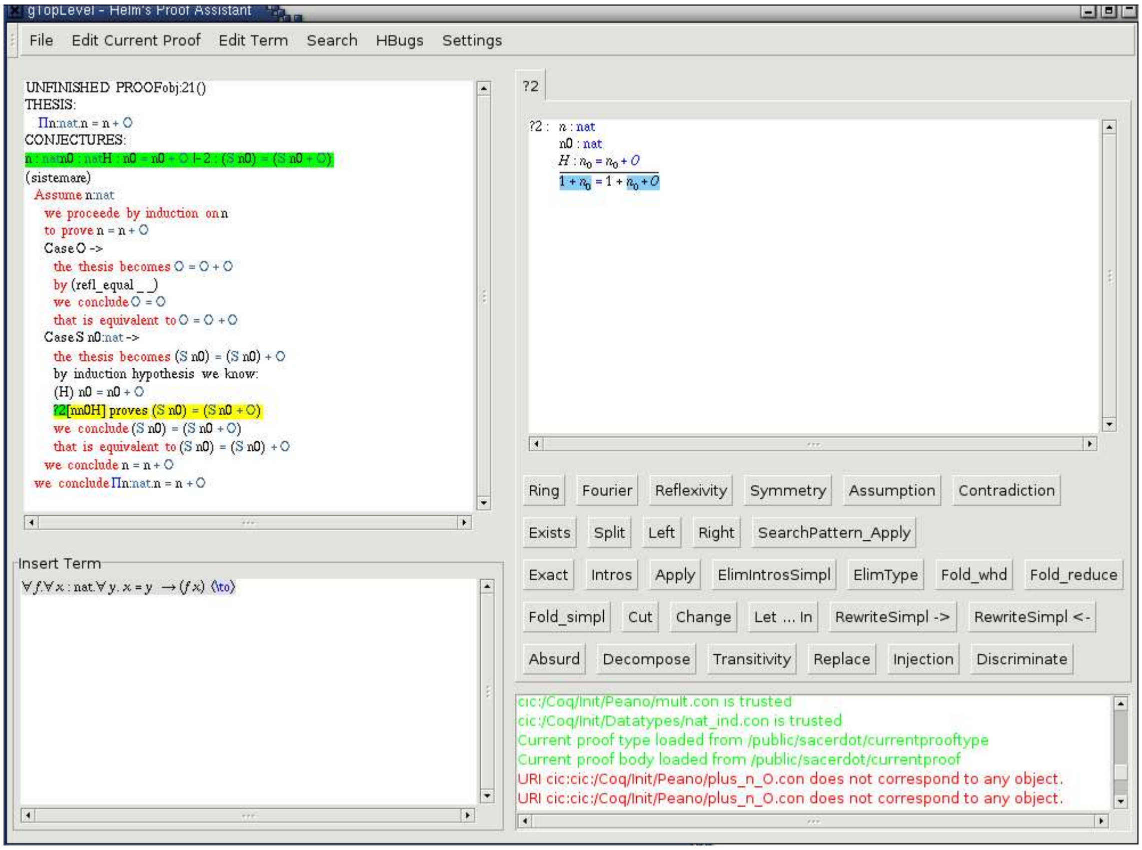

Figure 12.1: GTOPLEVEL main window

The widget in the left-bottom corner allows input of mathematical expressions in TEX-like syntax. The expressions are automatically converted in MathML Presentation and rendered during the typing phase. For instance, the image shows the token <\to>. This means that the user has entered the (incomplete) TEX macro \to. As soon as the user enters a character that ends the macro according to the TEX syntax (e.g. a blank), the token will be automatically converted to the "$\rightarrow$" symbol and rendered as such. The same thing happened with the two <\forall> tokens. As soon as an operation is invoked over the expression, the disambiguating parser developed in Sect. 10.4 will start processing the expression, contacting to SEARCH-ENGINE to resolve the identifiers. In case of multiple well-typed parsing trees, the user interface will propose to the user a menu where she can choose the wanted interpretation.

All the features described so far have been implemented in a few lines of code, simply putting together all the widgets and tools already developed in HELM and MoWGLI (as the MathML renderer GtkMathView or the widget EdiTEX for editing mathematical formulas in TEX syntax, both developed by Luca Padovani).

Some of the tactics available to the user are shown in a palette just under the current goal. These are the tactics that do not act on a selected region of the goal: they either take no input (as the Fourier or Assumption tactics) or they read their input (an expression) from the window in the left-bottom corner. The tactics that operate on the current goal selection can be activated from the contextual menu that is opened by clicking the right button of the mouse over the selection, as usual in Gtk interfaces. Notice, among the others, the two tactics `SearchPattern_Apply` and `ElimIntrosSimpl`. None of them are available in the Coq system. The first one, `SearchPattern_Apply`, invokes the SEARCH-ENGINE to look for all the theorems that could be applied to the current goal. Once the set of candidates is located, each candidate is tried by applying it to the current goal. Once the set of good candidates is computed, they are presented to the user in a new window where the user can graphically inspect their exact definition, and choose a theorem that she wants to actually apply. Thus the tactic looks a bit like the `Auto` tactic of Coq, but it does not proceed recursively and it looks in the whole distributed library, whereas `Auto` only consults a subset of the required modules in scope. The second one, `ElimIntrosSimpl`, applies in sequence the Coq tactics `Elim`, `Intros` and `Simpl`. This corresponds to the most common usage pattern of these tactics. Moreover, it forces the user to avoid commuting conversions (actually expansions) due to an elimination principle whose subgoals all start with another elimination principle whose subgoals all start with λ-abstractions that really "belong" to the outer elimination principle.

Among the actions available from the menu, we find:

- `File menu`:
  - `New block of (co)inductive definitions`: opens a wizard to define a new block of (co)inductive definitions, check its well-formedness and insert it into the library
  - `New proof or definition`: starts a new proof or interactive definition
  - `Reopen a finished proof`: browses a proof in the distributed library. By performing local undo over parts of the proof, the user can actually modify it.
  - `Load an unfinished proof` and `Save an unfinished proof`: used to load and save unfinished proof on the disk, without publishing them in the distributed library. Unpublished proof are not accessible by other users.
  - `Clear/Dump/Restore the environment`. Thanks to our algorithm for automatic reconstruction of environments, the user can always start in an empty environment and directly begin working on one proof. Since the environment is checked during the reconstruction, though, the performance of the system is degraded during the reconstruction phase. Thus, the user can, using this set of commands, save and restore from

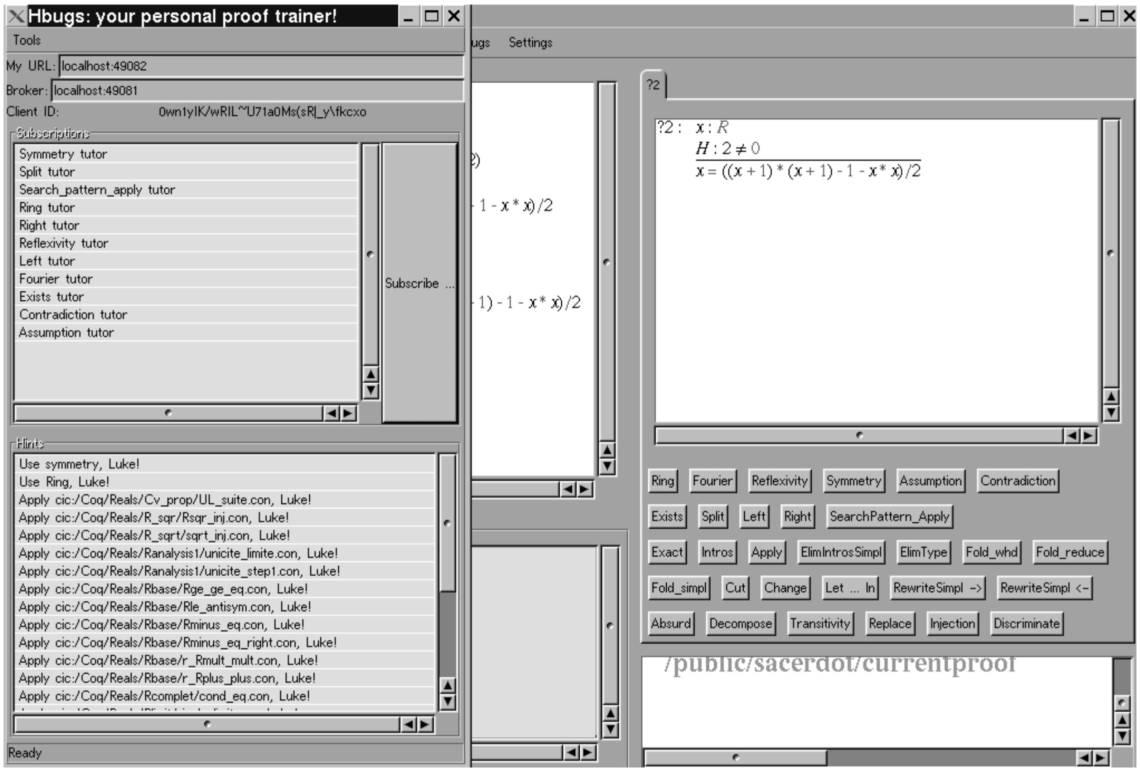

Figure 12.2: The H-BUGS suggestion window

the disk a "default" environment that is checked for well-formedness before saving it and it is not checked again when it is restored. For instance, if the user has to stop the system before completing a proof, it can be a good idea to dump the environment before quitting the application, in order to find exactly the same environment the next time GTOPLEVEL is lunched.

- `Edit current proof`: here we find the operations available over the proof object, such as local undo or focusing (i.e. changing the current goal). In the future other operations will be implemented, such as those required to rearrange the proof and already described in the requirements section.

- `Search`: this menu provides access to the SEARCH-ENGINE service. For performance reasons, the service can be compiled in-process instead of being remotely contacted as a Web service. We provide both an high level interface with automatically generated queries (see Chap. 5) and a direct access to the database by means of the MathQL Level 1 query language, developed by Ferruccio Guidi in his Ph.D. dissertation [54].

- `hbugs`: this menu allows to start the H-BUGS subsystem [111]. Fig. 12.2 is a snapshot of the H-BUGS interface in action. On the right hand side the current goal is shown. On the left hand side, there is the H-BUGS window. The upper part shows information about the broker in use and the tutors available. Among them, there are tutors that deal with expensive tactics, both in time and in space, such as the `Search_Pattern_Apply` tutor, the `Ring` tutor and the `Fourier` tutor. Since the tutors can run on a remote machine, they do not subtract resources to the user machine, still providing useful suggestions as soon as tactics that can be applied are discovered. So far, the most part of the residual tutors are only useful as real tutors for beginners. Indeed, they deal with trivial tactics whose applicability is evident to the expert user. For beginners, instead, they represent useful hints since they make the task of understanding when a tactic can be applied easier. The graphical interface also helps beginners to concentrate on learning how to prove things instead of wasting energies trying to remember the syntax and pre-conditions of the available tactics.

The lower part of the H-BUGS window shows the hints sent by the tutors and related to the current goal. Notice that several hints suggest to apply one theorem of the distributed library among the only ones that can be successfully applied. Among them, there is the theorem that suggests to multiply both members with the same scalar. The user can inspect the statement of the theorem by single-clicking over the suggestion, and she can apply the theorem by double-clicking on it. Notice that, without the tutor, the user would be obliged to either remember by heart the name of the theorem to apply or to use by hand the SEARCH-ENGINE to retrieve it.

12.3 Comparisons with Related Work

Our graphical interface does not propose any innovative functionality or interaction modality. Indeed, the part of the interface that deals with the current proof object is very similar and provides functionalities already provided by the interface of the Alfa system. The part of the interface that deals with the current goals, instead, is similar to many other interfaces for theorem provers, as PCoq [100] and ProofGeneral [12], just to cite a few of them. Semantic selection and advanced form of interaction with the goal displayed were already introduced in the CtCoq system [115], that is the former implementation of the PCoq interface.

The real element of novelty is having assembled the interface using generic components, as the GtkMathView widget and an in-process implementation of UWOBO. These components can be reused without any major modification in other proof assistants. Moreover, the functionalities of the interface have been augmented by interfacing it with remotely running Web services, that are also generic and logic independent components.

To be fair, we have to say that at least some of our generic components were also present in the CtCoq system already several years ago. Nevertheless, since the project predates the XML and Web services technologies, no simple integration of the components with other systems were possible. Indeed, the solution adopted was dual: CtCoq [20] was just the instantiation to Coq of the Centaur system [24], that could be instantiated to provide interfaces to other components. For instance, experiments were also tried with Lego and Isabelle. With minor modifications, this is also the approach adopted now by the Proof General generic interface to proof assistants [12], that provides an emacs package to interface emacs with several other proof assistants, providing a loosely coupled uniform interface to the systems.

The main advantage of generic interfaces is that they provide a real uniform interface to different systems, reducing the time necessary to adapt to a new interface when the user changes her proof assistant. The disadvantage is constituted by the fact that the generic interfaces are based on the intersection of the functionalities of the systems, providing the other functionalities in non standard way. Moreover, the interfaces are really loosely coupled and often they just communicate with the proof assistant by exchanging strings. For instance, it may not be possible for the interface to activate and deactivate some commands in response to the changes in the proof engine state, since no functionality to access the state is present.

Our generic components, on the contrary, can be adopted both in the creation of loosely coupled generic interfaces and in the creation of ad-hoc tightly coupled interfaces. The Lemme team is also working on porting the tools developed for PCoq to XML technologies, trying to benefits from the XML-related standards without loosing the features and the know-how acquired in the developments of their own technology.

When the HELM project was started, it was not clear whether XML technologies provided a good framework for the development of MKM tools that could be reused in proof assistants. More generally, some of the XML technologies, as XslT, were new and their "concrete" expressive power was unknown. In the last four years the HELM team has heavily stressed the majority of these technologies, often spotting weaknesses in the existing implementation and contributing to their specification. For instance, the HELM team has developed the GNU implementation of DOM Level 2 [29] that is used as the in-process communication format for XML documents and that defines also a system of events that are suitable for implementing reactive widgets to be used for editing; it has participated in the MathML working group and it has developed one of the first fully compliant widget for MathML Presentation; it has developed very complex XslT stylesheets and it has applied it to huge XML documents, creating scenarios that were too complex to be addressed with the existing XslT interpreter implementations[2]. It is now time to judge the results

[2]Either the XslT engine went out of memory, or it was extremely slow, in the order of several minutes. Indeed, in the last few years we have tried all the existing engines, always choosing the more performant one. Only recently, though, the first performant implementations in C have started to appear. Much can still be done, though. One of the motivations for the very slow improvement in these tools is given from the simple stylesheets and small documents that represent the

obtained and to draw future scenarios.

The first consideration that comes to our mind is that although these proposed standards have rapidly become widespread and well known, only a few of their potentiality has been explored so far. Indeed, the community of programmers has easily adopted XML as the standard language for encoding data, but it is using the more high level standards, as XQuery and XslT, only in the cases that are very simple and that do not require any complex programming skill. In the majority of the cases, instead of writing a stylesheet the programmers write a traditional program. As a consequence, the tools that process these standards are not optimized (since they only deal with easy tasks) and the programmers are not really skilled in the usage of the stylesheets. For the time being, all this hinders the main characteristic of a standard, that is to be well known and to have efficient and well tested implementations.

More concretely, for the time being XslT technology is too slow for processing huge terms (e.g. proof objects) in an interactive theorem proving scenario, where the user expects an immediate answer from the system, at least in the case of simple operations. Indeed, we are now experiencing a delay of at least a few seconds for each tactic that is applied, that is too annoying for the user. XslT technology is improving and it seems possible to study progressive solutions that allow to apply stylesheets over documents that describe only the changes applied to the input document in order to obtain the changes to be applied to the output document. Moreover, several of the serious limitation of XslT will be lifted in the next version of the standard. Thus, in a couple of years we expect to be able to adopt XslT technology without having to worry for performances.

Whereas XslT and XQuery technologies have not been adopted extensively, Web services are on their way for completely replacing COM and Corba components in the development of physically distributed systems. Thus, we really believe in the implementation of MKM tools as Web services, and in the implementation of proof assistants as both Web service clients and servers. The only serious problem with Web service technology that could become a show-stopper in the domain of MKM is the time required to encode and decode the information for its transmission: indeed, we often need to send whole proofs or parts of them between the distributed components of the MKM network; since the proofs can be huge documents, especially if they are identified with proof objects, the necessary delay can easily be greater than the time necessary to perform the logical operation over them, especially if a textual format as XML is used for the transmission.

To conclude, we now have the first prototype of a proof assistant based on MKM technologies. For its development new MKM services have been created, new problems have been addressed and new solutions have been proposed. The experience has surely been fruitful, since it has

large majority of real world input documents; thus only little pressure is applied for more performant tools.

shown that the integration of interactive theorem proving with Mathematical Knowledge Management can really produce significants benefits on the two subjects, at least since it opens new possibilities that can now be compared in practice to the traditional solutions.

A prototype is always a beginning and not a conclusion. Indeed, we are now ready to capitalize on our experience. Obtaining a real proof assistant out of the prototype means first of all to change several requirements and then to be prepared to face compromises. We are anyway pretty confident that the architectural choices we made are valid, and we already have a bunch of totally independent small components that can be easily replaced and that can be assembled to quickly test new implementations and different architectures. Now the fun begins.

References

[1] Jean-Raymond Abrial and Dominique Cansell. Click'n Prove: Interactive proofs within set theory. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLS 2003*, volume LNCS, 2758. Springer-Verlag, 2003.

[2] Peter Aczel. On relating type theories and set theories. *Lecture Notes in Computer Science*, 1657, 1999.

[3] A. A. Adams. Digitisation, representation and formalisation: Digital libraries of mathematics. In J.H. Davenport A. Asperti, B. Buchberger, editor, *Proceedings of Mathematical Knowledge Management 2003*, volume LNCS, 2594, pages 1–16. Springer-Verlag, 2003.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical Report BRICS RS-03-14, BRICS, March 2003.

[5] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. Technical Report BRICS RS-03-13, BRICS, March 2003.

[6] M. Agrawal, N. Kayal, and N. Saxena. PRIMES in P. `http://www.cse.iitk.ac.in/users/manindra/primality.ps`, August 2002.

[7] The Alfa proof editor. `http://www.math.chalmers.se/~hallgren/Alfa/`.

[8] Arith1 OpenMath Content Dictionary. The OpenMath Society, `http://www.openmath.org/cocoon/openmath/cd/arith1.ocd`, 2002.

[9] Andrea Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, 2(1):23–59, January 1992.

[10] Andrea Asperti, Ferruccio Guidi, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. Mathematical knowledge management in helm. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, May 2003.

[11] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. XML, stylesheets and the re-mathematization of formal content. In *Electronic Proceedings of EXTREME 2001*.
http://www.idealliance.org/papers/extreme03/html/2001/Asperti01/HELM2001Asperti01.html.

[12] David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, January 2000.

[13] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. Towards an evolutionary formal software-development using CASL. In *WADT99 Selected Papers Volume*, volume LNCS, 1827. Springer-Verlag, 2000.

[14] Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in MML. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003*, volume LNCS, 2594, pages 119–132. Springer-Verlag, 2003.

[15] H. Barendregt. Lambda Calculi with Types. In Abramsky, Samson and others, editor, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[16] Henk Barendregt. Towards an interactive mathematical proof language. In Fairouz Kamareddine, editor, *Thirty Five Years of Automath*, pages 25–36. Kluwer, 2003.

[17] Christoph Benzmüller. The CALCULEMUS research training network – a short overview. In Therese Hardin and Renaud Rioboo, editors, *Calculemus 2003*, pages 1–16. Aracne Editrice S.R.L., 2003. ISBN 88-7999-545-6.

[18] Christoph Benzmüller, Mateja Jamnik, Manfred Kerber, and Volker Sorge. Agent based mathematical reasoning. *Electronic Notes in Theoretical Computer Science, Elsevier*, 23(3):21–33, 1999.

[19] Christoph Benzmüller and Volker Sorge. OANTS – an open approach at combining interactive and automated theorem proving. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A.K.Peters, 2000.

[20] Yves Bertot. The ctcoq system: Design and architecture. *Formal Aspects of Computing*, 11:225–243, 1999.

[21] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25:161–194, 1998.

[22] Frédéric Blanqui. *Type Theory and Rewriting*. PhD thesis, Université Paris XI, 2001.

[23] Jorge Luis Borges. The library of Babel. In *Ficciones*. Grove Press, 1942.

[24] P. Borras, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press.

[25] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, LNCS*, volume 1281, pages 515–529. Springer-Verlag, 1997.

[26] The CALCULEMUS project.
`http://www.calculemus.net/`.

[27] O. Caprotti, D. P. Carlisle, and A. M. Cohen. *The OpenMath Standard*. The OpenMath Esprit Consortium.

[28] Olga Caprotti, Herman Geuvers, and Martin Oostdijk. Certified and portable mathematical documents from formal contexts. In Bruno Buchberger and Olga Caprotti, editors, *On-Line Proceedings of the First International Conference on Mathematical Knowledge Management, MKM 2001*. The Electronic Library of Mathematics (EMIS), 2001.

[29] Paolo Casarini and Luca Padovani. The Gnome DOM Engine. *Markup Languages: Theory & Practice*, 3(2):173–190, April 2002.

[30] Laurent Chicli. *Sur la formalisation des mathématiques dans le Calcul des Constructions Inductives*. PhD thesis, Université de Nice - Sophia Antipolis, 2003.

[31] Jacek Chrząszcs. *Modules in Type Theory with Generative Definitions*. PhD thesis, Uniwersytet Warszawski and Université de Paris Sud, 2003.

[32] The Coq proof-assistant.
`http://coq.inria.fr`.

[33] Thierry Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*, Cambridge, 1986. MA. IEEE press.

[34] Yann Coscoy. *Explication textuelle de preuves pour le Calcul des Constructions Inductives*. PhD thesis, Université de Nice-Sophia Antipolis, 2000.

[35] Judicaël Courant. Explicit universes for the Calculus of Constructions. In Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, volume 2410 of *Lecture Notes in Computer Science*, pages 115–130, Hampton, VA, USA, August 2002. Springer-Verlag.

[36] Vincent Danos and Laurent Regnier. How abstract machines implement head linear reduction. 2003.

[37] James H. Davenport. MKM from book to computer: A case study. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003*, volume LNCS, 2594, pages 17–29. Springer-Verlag, 2003.

[38] David Delahaye and Roberto di Cosmo. Information retrieval in a Coq proof library using type isomorphisms. In *Proceedings of TYPES 99*, volume LNCS. Springer-Verlag, 1999.

[39] Dewey decimal classification.
`http://www.oclc.org/dewey`.

[40] Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[41] Pietro Dilena. Generazione automatica di stylesheet per notazione matematica. Master's thesis, University of Bologna, 2003.

[42] Document Object Model (DOM) Level 2 Specification. Version 1.0. W3C Candidate Recommendation 10 May 2000,
`http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510/`.

[43] Extensible Markup Language (XML). Version 1.0. W3C Recommendation 10 February 1998,
`http://www.w3.org/TR/REC-xml`.

[44] R. Fateman, T. Tokuyasu, B. Berman, and N. Mitchell. Optical character recognition and parsing of typeset mathematics. *Journal of Visual Communication And Image Representation*, 7(1):2–15, March 1996.

[45] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the lambda calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V., (North-Holland) Amsterdam, 1987.

[46] Luís Cruz Filipe and Bas Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLS 2003*, volume LNCS, 2758. Springer-Verlag, 2003.

[47] Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. Technical Report RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, October 1995.

[48] Harald Ganzinger, Robert Nieuwehuis, and Pilar Nivela. Fast term indexing with coded context trees. journal of automated reasoning. *Journal of Automated Reasoning*, To appear.

[49] Jacques Garrigue. *Label-Selective Lambda-Calculi and Transformation Calculi*. PhD thesis, University of Tokyo, Department of Information Science, March 1995.

[50] Herman Geuvers. *Logics and Type Systems*. Ph.D. dissertation, Catholic University Nijmegen, 1993.

[51] Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CLS 2002*, volume LNCS, 2471, pages 537–552. Springer-Verlag, January 2002.

[52] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Types'94: Types for Proofs and Programs*, volume 996 of *LNCS*. Springer-Verlag, 1994. Extended version in LIP research report 95-07, ENS Lyon.

[53] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[54] Ferruccio Guidi. *Searching and Retrieving in Content-Based Repositories of Formal Mathematical Knowledge*. PhD thesis, University of Bologna, March 2003. Technical Report UBLCS 2003-06.

[55] Ferruccio Guidi and Claudio Sacerdoti Coen. Querying distributed digital libraries of mathematics. In Therese Hardin and Renaud Rioboo, editors, *Calculemus 2003*, pages 17–30. Aracne Editrice S.R.L., 2003. ISBN 88-7999-545-6.

[56] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *LPAR'2000*, volume 1955 of *LNCS/LNAI*, 2000.

[57] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[58] John Harrison. The QED Manifesto. In *Automated Deduction - CADE 12*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 238–251. Springer-Verlag, 1994.

[59] John Harrison. A Mizar Mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.

[60] John Harrison. Real numbers in real applications. Technical Report NASA/CP-2002-211736, NASA, 2002. in Proceedings of the Workshop on Formalizing Continuous Mathematics, FCM 2002.

[61] Michiel Hazewinkel. Dynamic stochastic models for indexes and thesauri, identification clouds, and information retrieval and storage. MKM Net Report.

[62] The HELM project.
`http://helm.cs.unibo.it`.

[63] The HELM on-line library.
`http://helm.cs.unibo.it/library.html`.

[64] The HOL Light proof-assistant.
`http://www.cl.cam.ac.uk/users/jrh/hol-light/`.

[65] The Isabelle proof-assistant.
`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

[66] G. I. Jojgov. Systems for open terms: An overview. Technical Report CSR 01-03, Technische Universiteit Eindhoven, 2001.

[67] Jutting van L. S. B. *Checking Landau's "Grundlagen" in the AUTOMATH System*. Ph d. thesis, Eindhoven University of Technology, 1994. Useful summary in Nederpelt, Geuvers and de Vrijier, 1994, 701-732.

[68] Michael Kohlhase and Romeo Anghelache. Towards collaborative content management and version control for structured mathematical knowledge. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003*, volume LNCS, 2594, pages 147–161. Springer-Verlag, 2003.

[69] Michael Kohlhase and Andreas Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation*, 32(4):365–402, 2001.

[70] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available on-line at `http://www.logique.jussieu.fr/~krivine`, 1985.

[71] Learning Object Metadata standard. Learning Technology Standards Committee of IEEE, `http://www.learninglab.de/elan/kb3/lexikon/metadaten-standards/docs/LOM_1484_12_1_v1_Final_Draft.pdf`, 2002.

[72] The Lego proof-assistant.
`http://www.dcs.ed.ac.uk/home/lego/`.

[73] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[74] Library of congress classification scheme.
`http://www.loc.gov`.

[75] Sylvain Lippi. *Théorie et pratique des réseaux d'interaction (Interaction nets)*. PhD thesis, Université Aix-Marseille 2, 2002.

[76] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[77] Nicolas Magaud. Changing Data Representation within the Coq System. In *TPHOLs'2003*, volume LNCS, 2758. Springer-Verlag, 2003.

[78] Lena Magnusson. *The Implementation of ALF – a Proof Editor based on Martin-Löf Monomorphic Type Theory with Explicit Substitutions*. PhD thesis, Chalmers University of Technology / Göteborg University, 1995.

[79] Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation 21 February 2001,
`http://www.w3.org/TR/MathML2`.

[80] Mathematical Subject Classification, American Mathematical Society.
`http://www.ams.org/msc`.

[81] The MathWeb project.
`http://www.mathweb.org`.

[82] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[83] Erica Melis, Jochen Bäudenbender, George Goguadze, Paul Libbrecht, and Carsten Ullrich. Knowledge representation and management in ActiveMath. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):47–64, May 2003.

[84] Alexandre Miquel. *Le Calcul des Constructions Implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, 2001.

[85] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevant model of CC. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs: International Workshop, TYPES 2002*, volume LNCS, 2646, pages 240–258. Springer-Verlag, 2003.

[86] The Mizar proof-assistant.
`http://mizar.uwb.edu.pl/`.

[87] The MONET project.
`http://monet.nag.co.uk/cocoon/monet/index.html`.

[88] MoWGLI project deliverables.
`http://mowgli.cs.unibo.it/html_no_frames/deliverables/index.html`.

[89] The MoWGLI Proposal, HTML version.
`http://mowgli.cs.unibo.it/html_no_frames/project.html`.

[90] César Munoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

[91] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[92] Hanane Naciri and Luca Padovani. MathML rendering/browsing engine. MoWGLI Report D4a, 2003.

[93] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[94] George C. Necula and Peter Lee. Efficient Representation and Validation of Proofs. In *Proceedings of the 13th Annual symposium on Logic in Computer Science,*, Indianapolis, 1998.

[95] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 1994. ISBN-0444898220.

[96] The NuPRL proof-assistant.
`http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

[97] OMDoc: An open markup format for mathematical documents (version 1.1).
`http://www.mathweb.org/omdoc/omdoc.ps`.

[98] Luca Padovani. *MathML Formatting*. PhD thesis, University of Bologna, February 2003. Technical Report UBLCS 2003-03.

[99] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supŕieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[100] The PCoq project.
`http://www-sop.inria.fr/lemme/pcoq`.

[101] The PhoX proof-assistant.
`http://www.lama.univ-savoie.fr/~RAFFALLI/phox.html`.

[102] Olivier Pons. *Conception et réalisation d'outils d'aide au développement de grosse théories dans les systèmes de preuves interactifs*. PhD thesis, Conservatoire National des Arts et Métiers, 1999.

[103] Progress report: Building interactive digital libraries of formal algorithmic knowledge. Cornell University,
`http://www.cs.uwyo.edu/~nuprl/documents/cornell_slides.pdf`, May 2002.

[104] The PVS specification and verification system.
`http://pvs.csl.sri.com/`.

[105] Aarne Ranta. Grammatical framework: A type-theoretical grammar formalism. *Journal of Functional Programming*, To appear. Manuscript made available in September 2002.

[106] Resource Description Framework (RDF) model and syntax specification. W3C Recommendation 22 February 1999,
`http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[107] Claudio Sacerdoti Coen. Progettazione e realizzazione con tecnologia XML di basi distribuite di conoscenza matematica formalizzata. Master's thesis, University of Bologna, 2000.

[108] Claudio Sacerdoti Coen. Tactics in modern proof-assistants: the bad habit of overkilling. In *Supplementary Proceedings of the 14th International Conference TPHOLS 2001*, pages 352–367, 2001.

[109] Claudio Sacerdoti Coen. A constructive proof of the soundness of the encoding of random access machines in a linda calculus with ordered semantics. In *Theoretical Computer Science: 8th Italian Conference, ICTCS 2003*, volume LNCS, 2841, pages 37–57. Springer-Verlag, 2003.

[110] Claudio Sacerdoti Coen. From proof-assistans to distributed libraries of mathematics: Tips and pitfalls. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003*, volume LNCS, 2594, pages 30–44. Springer-Verlag, 2003.

[111] Claudio Sacerdoti Coen and Stefano Zacchiroli. Brokers and Web-services for automatic deduction: a case study. In Therese Hardin and Renaud Rioboo, editors, *Calculemus 2003*, pages 43–57. Aracne Editrice S.R.L., 2003. ISBN 88-7999-545-6.

[112] Giovanni Sambin. Some points in formal topology. To appear in Theoretical Computer Science, 2002.

[113] Irene Schena. *Towards a Semantic Web for Formal Mathematics*. PhD thesis, University of Bologna, 2002.

[114] Martin Strecker. *Construction and Deduction in Type Theories*. PhD thesis, Universität Ulm, 1998.

[115] Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. Technical Report Inria Research Report 1684, INRIA, May 1992.

[116] Universal resource identifiers in WWW. RFC 1630, CERN, June 1994.

[117] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–184, 1999.

[118] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, May 1994.

[119] Benjamin Werner. Sets in types, types in sets. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, LNCS*, volume 1281, pages 530–546. Springer-Verlag, 1997.

[120] Freek Wiedijk. A new implementation of Automath. *Journal of Automated Reasoning*, 29:365–387, 2002.

[121] Freek Wiedijk. Formal proof sketches. In Wan Fokkink and Jaco van de Pol, editors, *7th Dutch Proof Tools Day, Program + Proceedings*, 2003. CWI, Amsterdam.

[122] XML Path Language (XPath). version 1.0. W3C Recommendation, 16 November 1999, `http://www.w3.org/TR/xpath`.

[123] XSL Transformations (XSLT). Version 1.0. W3C Recommendation, 16 November 1999, `http://www.w3.org/TR/xslt`.

[124] Stefano Zacchiroli. Web services per il supporto alla dimostrazione interattiva. Master's thesis, University of Bologna, 2003.

[125] Jan Zwanenburg. The proof-assistant yarrow. Technical Report Computing Science Report CS-98-11, Eindhoven University of Technology, 1998.