Smart matching

Andrea Asperti, Enrico Tassi

[1] Department of Computer Science, University of Bologna
asperti@cs.unibo.it
[2] Microsoft Research-INRIA Joint Center
enrico.tassi@inria.fr

Abstract. One of the most annoying aspects in the formalization of mathematics is the need of transforming notions to match a given, existing result. This kind of transformations, often based on a conspicuous background knowledge in the given scientific domain (mostly expressed in the form of equalities or isomorphisms), are usually implicit in the mathematical discourse, and it would be highly desirable to obtain a similar behaviour in interactive provers. The paper describes the superposition-based implementation of this feature inside the Matita interactive theorem prover, focusing in particular on the so called *smart application* tactic, supporting smart matching between a goal and a given result.

1 Introduction

The mathematical language has a deep contextual nature, whose interpretation often presupposes not trivial skills in the given mathematical discipline. The most common and typical example of these "logical abuses" is the implicit use of equalities and isomorphisms, allowing a mathematician to freely move between different incarnations of a same entity in a completely implicit way. Equipping ITP systems with the capability of reasoning up to equality yields an essential improvement of their intelligence, making the communication between the user and the machine sensibly easier.

Techniques for equational reasoning have been broadly investigated in the realm of automated theorem proving (see eg [7,22,10]). The main deductive mechanism is a *completion* technique [17] attempting to transform a given set of equations into a confluent rewriting system so that two terms are equal if and only if they have identical normal forms. Not every equational theory can be presented as a confluent rewriting system, but one can progressively approximate it by means of a refutationally complete method called *ordered completion*. The deductive inference rule used in completion procedures is called *superposition*: it consists of first unifying one side of one equation with a subterm of another, and hence rewriting it with the other side. The selection of the two terms to be unified is guided by a suitable *term ordering*, constraining inferences and sensibly pruning the search space.

Although we are not aware of any work explicitly focused on superposition techniques for interactive provers, the integration between fully automatic

provers (usually covering paramodulation) and interactive ones is a major research challenge and many efforts have been already done in this direction: for instance, KIV has been integrated with the tableau prover $3T^AP$ [1]; HOL has been integrated with various first order provers, such as Gandalf [15] and Metis; Coq has been integrated with Bliksem [8]; Isabelle was first integrated with a purpose-built prover [23] and more recently with Vampire [20]. The problems of these integrations are usually of two kinds: (a) there is a *technical* difficulty in the forward and backward translation of the information between systems, due to the different underlying logics (ITP systems are usually higher-order, and some of them intuitionistic); (b) there is a *pragmatical* problem in the management of the knowledge base to be used by the automatic solver, since it can be huge (so we cannot pass it at every invocation), and it grows dynamically (hence, it cannot be exported in advance).

A good point of the superposition calculus (and not the last reason for restricting the attention to this important fragment) is that point (a), in this context, becomes relatively trivial (and the translation particularly effective). As for point (b), its main consequence is that the communication between the Interactive Prover and the Problem Solver, in order to be efficient, cannot be *stateless*: the two systems must share a common knowledge base. This fact, joined with the freedom to adapt the superposition tool to any possible specific requirement of the Matita system convinced us to rewrite our own solver, instead of trying to interface Matita with some available tool. This paper discusses our experience of implementation of a (first order) superposition calculus (Section 2), its integration within the (higher-order) Matita interactive prover [5] (Section 3), and in particular its use for the implementation of a *smart application* tactic, supporting smart matching between a goal and a given results (Section 4). We shall conclude with a large number of examples of concrete use of this tactic.

2 The Matita superposition tool

One of the components of the automation support provided by the Matita interactive theorem prover is a first order, untyped superposition tool. This is a quite small and compact application (little more than 3000 lines of OCaml code), well separated by the rest of the system. It was entirely rewritten during the summer 2009 starting from a previous prototype (some of whose functionalities had been outlined in [6]), with the aim to improve both its abstraction and performance. The tool took part to the 22nd CADE ATP System Competition, in the unit equality division, scoring in fourth position, beating glorious systems such as Otter or Metis [16], and being awarded as the best new entrant tool of the competion [28].

In the rest of this section we shall give an outline, as concise as possible, of the theory and the architecture of the tool. This is important in order to understand its integration with the interactive prover.

2.1 The superposition calculus in a nutshell

Let $\mathcal{F}$ bet a countable alphabet of functional symbols, and $\mathcal{V}$ a countable alphabet of variables. We denote with $\mathcal{T}(\mathcal{F},\mathcal{V})$ the set of terms over $\mathcal{F}$ with variables in $\mathcal{V}$. A term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ is either a 0-arity element of $\mathcal{F}$ (constant), an element of $\mathcal{V}$ (variable), or an expression of the form $f(t_1,\ldots,t_n)$ where f is a element of $\mathcal{F}$ of arity n and $t_1,\ldots,t_n$ are terms.

Let s and r be two terms. $s|_p$ denotes the subterm of s at position p and $s[r]_p$ denotes the term s where the subterm at position p has been replaced by r.

A substitution is a mapping from variables to terms. Two terms s and t are unifiable if there exists a substitution σ such that $s\sigma = t\sigma$. In the previous case, σ is called a most general unifier (mgu) of s and t if for all substitution θ such that $s\theta = t\theta$, there exists a substitution τ which satisfies $\theta = \tau \circ \sigma$.

A literal is either an abstract predicate (represented by a term), or an equality between two terms. A clause $\Gamma \vdash \Delta$ is a pair of multisets of literals: the negative literals Γ, and the positive ones Δ. If $\Gamma = \emptyset$ (resp. $\Delta = \emptyset$), the clause is said to be positive (resp. negative).

A Horn clause is a clause with at most one positive literal. A unit clause is a clause composed of a single literal. A unit equality is a unit clause where the literal is an equality.

A strict ordering $\prec$ over $\mathcal{T}(\mathcal{F},\mathcal{V})$ is a transitive and irreflexive (possibly partial) binary relation. An ordering is *stable* under substitution if $s \prec t$ implies $s\sigma \prec t\sigma$ for all terms t, s and substitutions σ. A well founded monotonic ordering stable under substitution is called *reduction ordering* (see [11]). The intuition behind the use of reduction orderings for limiting the combinatorial explosion of new equations during inference, is to only rewrite big terms to smaller ones.

superposition left	**superposition right**	**equality resolution**
$\dfrac{\vdash l = r \qquad t_1 = t_2 \vdash}{(t_1[r]_p = t_2 \vdash)\sigma}$	$\dfrac{\vdash l = r \qquad \vdash t_1 = t_2}{(t_1[r]_p = t_2 \vdash)\sigma}$	$\dfrac{t_1 = t_2 \vdash}{\Box}$
if $\sigma = mgu(l, t_1|_p), t_1|_p \neq x, l\sigma \not\preceq r\sigma$ and $t_1\sigma \not\preceq t_2\sigma$		if $\exists\sigma = mgu(t_1, t_2)$.

Fig. 1. Inference rules

For efficiency reasons, the calculus must be integrated with a few additional optimization rules, the most important one being demodulation ([29]).

2.2 The main algorithm

A naive implementation of the superposition calculus could just combine (superpose) all known clauses in all (admitted) ways, and repeat that process until the desired clause (called *goal*) is resolved. To avoid useless duplication of work,

subsumption	**tautology elimination**	**demodulation**
$\frac{S \cup \{C, D\}}{S \cup \{C\}}$	$\frac{S \cup \{\vdash t = t\}}{S}$	$\frac{S \cup \{\vdash l = r, C\}}{S \cup \{\vdash l = r, C[r\sigma]_p\}}$
if $\exists \sigma, D\sigma \equiv C$		if $l\sigma \equiv C\vert_p$ and $l\sigma \succ r\sigma$

Fig. 2. Simplification rules

it is convenient to keep clauses in two distinct sets, traditionally called *active* and *passive*, with the general invariant that clauses in the active set have been already composed together in all possible ways. At every step, some clauses are selected from the passive set and added to the active set, then superposed with the active set, and consequently with themselves (*inference*). Finally, the newly generated clauses are added to the passive set (possibly after a simplification).

A natural selection strategy, resulting in a very predictable behaviour, would consist in selecting the whole passive set at each iteration, in the spirit of a breadth first search. Unfortunately the number of new equations generated at each step grows extremely fast, in practice preventing the iteratation of the main loop more than a few times.

To avoid this problem, all modern theorem provers (see e.g. [24]) adopt the opposite solution. According to some heuristics, like size and goal similarity for example, they select only *one* passive clause at each step. Not to loose completeness, some fairness conditions are taken into account (i.e. every passive clause will be eventually selected). This approach falls under the name *given-clause*

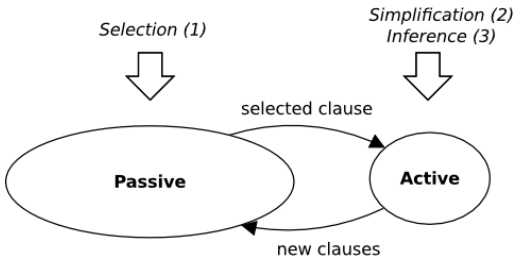

Fig. 3. given-clause loop
Numbers in parentheses reflect the steps order.

(Figure 3), and its main advantage is that the passive set grows much slower, allowing a more focused and deeper inspection of the search space that consequently allows to find proofs that require a much higher number of main loop iterations.

The main drawback of this approach is that it makes the procedure way more sensible to the selection heuristics, leading to an essentially unpredictable behaviour.

2.3 Performance issues

In order to obtain a state-of-the-art tool able to compete with the best available systems one has eventually to take into account a lot of optimizations and techniques developed for this purpose during the last thirty years.

In the following we shall shortly describe the most critical areas, and, for each of them, the approach adopted in Matita.

Orderings used to orientate rewriting rules On complex problems (e.g. problems in the TPTP library with rating greater then 0.30) the choice of a good ordering for inference rules is of critical importance. We have implemented several orderings, comprising standard Knuth-Bendix (KBO), non recursive Knuth-Bendix (NRKBO), lexicographic path ordering (LPO) and recursive path ordering (RPO). The best suited ordering heavily depends on the kind of problem, and is hard to predict: our approach for the CADE ATP System Competition was to run in parallel different processes with different orderings.

On simpler problems (of the kind required for the smart application tactic of section **??**), the given-clause algorithm is less sensitive to the term-ordering, and we may indifferently choose our preferred strategy, opportunely tuning the library (we are currently relying on LPO).

Selection strategy The selection strategy currently implemented by Matita is a based on combination of age and weight. The weight is a positive integer that provides an estimation of the "complexity" of the clause, and is tightly related to the number of occurrences of symbols in it.

Since we are not interested in generating (counter) models of false statements, we renounced to be complete, and we silently drop inferred clauses that would slow down the main loop too much due to their excessive size.

Another similar optimization we did not implement but we could consider as a future development is Limited Resource Strategy [25], which basically allows the procedure to skip some inference steps if the resulting clauses are unlikely to be processed, mainly because of a lack of time.

Data structures and code optimization We adopted relatively simple data structures (like discrimination [18] trees for term indexing), and a purely functional (in the sense of functional programming) implementation of them. After some code optimisation, we reached a point where very fast functions are the most expensive, because of the number of calls (implied by the number of clauses), even if they operate on simple data structures.

Since we are quite satisfied with the actual performance, we did not invest resources in adopting better data structures, but we believe that further optimizations will probably require implementing more elaborate data structures, such as substitution [14] or context trees [13], or even adopt an indexing technique that works modulo associativity and commutativity [12], that looks very promising when working on algebraic structures.

Demodulation Another important issue for performance is demodulation: the given clause algorithm spends most of its time (up to 80%) in simplification, hence any improvement in this part of the code has a deep impact on performance. However, while reduction strategies, sharing issues and abstract machines have been extensively investigated for lambda calculus (and in general for left linear systems) less is known for general first order rewriting systems. In particular, while an innermost (eager) reduction strategy seem to work generally better than an outermost one (especially when combined with lexicographic path ordering), one could easily create examples showing an opposite behaviour (even supposing to always reduce needed redexes).

3 Integrating superposition with Matita

3.1 Library management

A possible approach to the integration of superposition with Matita is to solve all goals assuming that all equations part of the library lie in the passive set, augmented on the fly with the equations in the local context of the ongoing proof.

The big drawback of this approach is that, starting essentially from the same set of passive equations at each invocation on a different goal (differing only for the local context), the given clause algorithm would mostly repeat the same selection and composition operations over and over again. It is clear that, if we wish to superpose library equations, this operation should not be done at run time but in background, once and for all. Then we have to face a dual problem, namely to understand when stopping the saturation of the library with new equations, preventing an annoying pollution with trivial results that could have very nasty effects for selection and memory occupation. We would eventually like to have mechanisms to drive the saturation process.

A natural compromise is to look at library equations not as a passive set, but as the *active* one. This means that every time a new (unit) equation is added to the library it also goes through one main given-clause loop, as if it was the newly selected passive equation: it is simplified, composed with all existing active equations (i.e. all other equations in the library, up to simplification), and the newly created equations are added to the passive list. At run time, we shall then strongly privilege selection of local equations or goals.

This way, we have a natural, simple but traceable syntax to drive the saturation process, by just listing in library the selected equations. As a side effect, this

approach reduces the verbosity of the library by making it unnecessary to declare (and name explicitly) trivial variants of available results that are automatically generated by superposition.

3.2 Interfacing CIC and the superposition engine

Our superposition tool is first order and untyped, while the Matita interactive prover is based on a variant of the Calculus of Inductive Construction (CIC), a complex higher-order intuitionistic logical systems with dependent types. The communication between the two components is hence far from trivial.

Instead of attempting a complex, faithful encoding of CIC in first order logic (that is essentially the approach adopted for HOL in [19]) we choose to follow a more naif approach, based on a forgetful translation that remove types and just keeps the first order applicative skeleton of CIC-terms.

In the opposite direction, we try to reconstruct the missing information by just exploiting the sophisticated inference capability of the Matita *refiner* [3], that is the tool in charge of transforming the user input into a machine understandable low-level CIC term.

Automation is thus a best effort service, in the sense that not only it may obviously fail to produce a proof, but sometimes it could produce an argument that Matita will fail to understand, independently from the fact if the delivered proof was "correct" or less.

The choice to deal with untyped first order equations in the superposition tool was mostly done for simplicity and modularity reasons. Moving towards a typed setting would require a much tighter integration between the superposition tool and the whole system, due to the complexity of typing and unification, but does not seem to pose any major theoretical problem.

The forgetful encoding Equations $r =_T s$ of the calculus of constructions are translated to first order equations by merely following the applicative structure of r and s, and translating any other subterm into an opaque constant. The type T of the equation is recorded, but we are not supposed to be able to compute types for subterms.

In spite of the fact of neglecting types, the risk of producing "ill-typed" terms via superposition rules is moderate. Consider for instance the superposition left rule (the reasoning is similar for the other rules)

$$\frac{\vdash l = r \qquad t_1 = t_2 \vdash}{(t_1[r]_p = t_2 \vdash)\sigma}$$

where $\sigma = mgu(l, t_1|_p)$ and $l\sigma \not\preceq r\sigma$. The risk is that $t_1|_p$ has a different type from l, resulting into an illegal rewriting step. Note however that l and r are usually rigid terms, whose type is uniquely determined by the outermost symbol. Moreover, $t_1|_p$ cannot be a variable, hence they must share this outermost symbol. If l is not rigid, it is usually a variable x and if $x \in r$ (like e.g. in $x = x + 0$)

we have (in most orderings) $l \preceq r$ that again rules out rewriting in the wrong direction.

This leads us to the following notion of *admissibility*. We say that an applicative term $f(x_1, \ldots, x_n)$ is *implicitly typed* if its type is uniquely determined by the type of f. We say that an equation $l = r$ is admissible if both l and r are implicitly typed, or $l \preceq r$ and r is implicitly typed. Non admissible equations are not taken into account by the superposition tool[1].

In practice, most unit equalities are admissible. A typical counter example is an equation of the kind $\forall x, y : unit.x = y$, where *unit* is a singleton type.

On the other side, non-unit equalities are often not admissible. For instance, a clause of the kind $x \wedge y = true \vdash x = true$ could be used to rewrite any term to true, generating meaningless, ill typed clauses. Extending superposition beyond the unit equality case does eventually require to take types into consideration.

3.3 (Re)construction of the proof term

Translating a first-order resolution proof into a higher-order logic natural deduction proof is a notoriously difficult issue, even more delicate in case of intuitionistic systems, as the one supported by Matita. While resolution *per se* is a perfectly constructive process, skolemization and transformation into conjunctive normal forms are based on classical principles.

Our choice of focusing on the superposition calculus was also motivated by the fact it poses less difficulties, since skolemization is not needed and thus proofs have a rather simple intuitionistic interpretation.

Our technique for reconstructing a proof term relies as much as possible on the refinement capabilities of Matita, in particular for inferring implicit types. In the superposition module, each proof step is encoded as a tuple

```
Step of rule * int * int * direction * position * substitution
```

where rule is the kind of rule which has been applied, the two integers are the two $id's$ of the composing equations (referring to a "bag" of unit clauses), direction is the direction the second equation is applied to the first one, position is a path inside the rewritten term and finally substitution is the mgu required for the rewriting step.

Every superposition step is encoded by one of the following terms:

$$eq\_ind\_l : \forall A : \textsc{Type}.\forall x : A.\forall P : A \to \textsc{Prop}.P\ x \to \forall y : A.x = y \to P\ y$$
$$eq\_ind\_r : \forall A : \textsc{Type}.\forall x : A.\forall P : A \to \textsc{Prop}.P\ x \to \forall y : A.y = x \to P\ y$$

where left ($\_l$) and right ($\_r$) must be understood w.r.t. backward application, and where P is the one hole context that represents the position in which the superposition occurred.

[1] A more liberal, but also slightly more expensive solution consists in indexing any equation and systematically try to read back each result of a superposition step in CIC, dropping it if it is not understood by the refiner.

At the end of the superposition procedure, if a proof is found, either a trivial goal has been generated, or a fact subsumes one of the active goals. In that latter case, we perform a rewriting step on the subsumed goal, so that we fall back into the previous case. Thus, when the procedure successfully stops, the selected clause is of the form $s = t$ where s and t are unifiable. We call it the meeting point, because forward steps (superposition right) and backward steps (superposition left) meet together when this trivial clause is generated, to compose the resulting proof. To generate a CIC proof term, the clauses are topologically sorted, their free variables are explicitly quantified, and nested let-in patterns are used to build the proof.

The most delicate point of the translation is closing each clause w.r.t. its free variables, since we should infer a type for them, and since CIC is an explicitly polymorphic language it is often the case that the order of abstractions does matter (e.g. variables standing for types must in general be abstracted before polymorphic variables).

The simplest solution is to generate so called "implicit" arguments leaving to the Matita *refiner* the burden of guessing them.

For instance, superposing lencat : $len\ A\ x + len\ A\ y = len\ A\ (x\underline{@}y)$ with catA : $x@(y@z) \stackrel{\leftarrow}{=} (x@y)@z$ at the underlined position and in the given direction gives rise to the following piece of code, where question marks stand for implicit arguments:

```
let clause_59 :
  ∀w :?.∀x :?.∀y :?.∀z :?.
    len w (x@y) + len w z = len w (x@(y@z))
:=
  λw :?.λz :?.λx :?.λy :?.
    eq_ind_r (List w) ((x@y)@z))
      (λhole : List w.len w (x@y) + len w z = len w hole)
      (lencat w (x@y) z)
      (x@(y@z))
      (catA w x y z) in
```

Note that w *must* be abstracted first, since it occurs in the (to be inferred) types for x, y and z. Also note the one hole context expressed as an anonymous function whose abstracted variable is named *hole*, corresponding to the position of $x\underline{@}y$ in the statement of lencat.

The interesting point is that refining is a complex operation, using e.g. hints, and possibly calling back the automation itself: the interpretation of the proof becomes hence a dialog between the system and its automation components, aimed to figure out a correct interpretation out of a rough initial trace.

A more sophisticated translation, aimed to produce a really nice, human-readable output in the form of a chain of equations, is described in [6].

4 Smart application

The most interesting application of superposition (apart from its use for solving equational goals) is the implementation of a more flexible application tactic. As a matter of fact, one of the most annoying aspects of formal development is the need of transforming notions to match a given, existing result. As explained in the introduction, most of these transformations are completely transparent to the typical mathematical discourse, and we would like to obtain a similar behaviour in interactive provers.

Given a goal $\mathcal{B}$ and a theorem t: $A \to B$, the goal is to try to match B with $\mathcal{B}$ *up to the available equational knowledge base*, in order to apply t. We call it, the *smart application* of t to G. We use superposition in the most direct

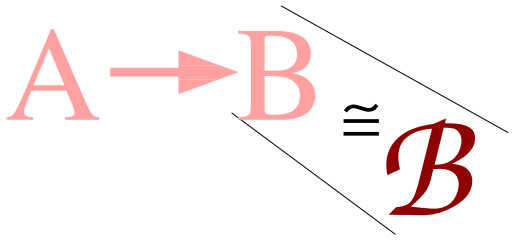

Fig. 4. Smart application

way, exploiting on one side the higher-order features of CIC, and on the other the fact that the translation to first order terms does not make any difference between predicates and functions: we simply generate a goal $B = \mathcal{B}$ and pass it to the superposition tool (actually, it was precisely this kind of operation that motivated our original interest in superposition). If a proof is found, $\mathcal{B}$ is transformed into B by rewriting and t is then normally applied.

Superposition, addressing a typically undecidable problem, can easily diverge, while we would like to have a reasonably fast answer to the smart application invocation, as for any other tactic of the system. We could simply add a timeout, but we prefer to take a different, more predictable approach. As we already said, the overall idea is that superposition right steps - realising the *saturation* of the equational theory - should be thought of as background operations. Hence, at run time, we should conceptually work as if we had a *confluent* rewriting system, and the only operation worth to do is *narrowing* (that is, left superposition steps). Narrowing too can be undecidable, hence we fix a given number of narrowing operations to apply to each goal (where the new goal instances generated at each step are treated in parallel). The number of narrowing steps can be fixed by the user, but a really small number is usually enough to solve the problem if a solution exists.

5 Examples

Example 1. Suppose we wish to prove that the successor function is le-reflecting, namely

$$(*) \quad \forall n, m.Sn \leq Sm \rightarrow n \leq m$$

Suppose we already proved that the predecessor function is monotonic:

$$monotonic\_pred : \forall n, m.n \leq m \rightarrow pred\ n \leq pred\ m$$

We would like to merely "apply" the latter to prove the former. Just relying on unification, this would not be possible, since there is no way to match $pred\ X \leq pred\ Y$ versus $n \leq m$ unless *narrowing* the former. By superposing twice with the equation $\forall n.pred(Sn) = n$ we immediately solve our matching problem via the substitution $\{X := Sn, Y := Sm\}$. Hence, the smart application of *monotonic_pred* to the goal $n \leq m$ succeeds, opening the new goal $Sn \leq Sm$ that is the assumption in $(*)$.

Example 2. Suppose we wish to prove $n \leq m * n$ for all natural numbers n, m. Suppose we already proved that multiplication is left-monotonic, namely

$$monotonic\_le\_times\_l : \forall n, a, b.a \leq b \rightarrow a * n \leq b * n$$

In order to apply this result, the system has to find a suitable $?_a$ such that $?_a * n = n$, that is easily provided by the identity law for times.

Example 3. In many cases, we just have local equational variants of the needed results. Suppose for instance we proved that multiplication in le-reflecting in its right parameter:

$$le\_times\_to\_le\_times\_r : \forall a, n, m.a * n \leq a * m \rightarrow n \leq m$$

Since times is commutative, this also trivially implies the left version:

$$monotonic\_le\_times\_l : \forall a, n, m.n * a \leq m * a \rightarrow n \leq m$$

Formally, suppose to have the goal $n \leq m$ under the assumption $(H)\ n*a \leq m*a$. By applying *le_times_to_le_times_r* we obtain a new goal $?_a * n \leq ?_a * m$ that is a smart variant of H.

Example 4. Suppose we wish to prove that $(H)\ a * (Sn) \leq a * (Sm)$ implies $a*n \leq a*m$, where S is the successor function (this is a subcase in the inductive proof that the product by a positive constant a is le-reflecting). Suppose we already proved that the sum is le-reflecting in its second argument:

$$le\_plus\_to\_le\_plus\_r : \forall a, n, m.a + n \leq a + m \rightarrow n \leq m$$

By applying this result we obtain the new goal $?_a + a * n \leq ?_a + a * m$, and if we have the expected equations for times, we can close the proof by a smart application of H.

Example 5. Consider the goal $n < 2 * m$ under the assumptions $(H)\ 0 < m$ and $(H1)\ n \leq m$. Suppose that we defined $x < y$ as $x + 1 \leq y$. Morevoer, by the defining equation of times we should know something like $2 * m = m + (m + 0)$.[2]

[2] The precise shape depends by the specific equations available on times.

Hence the goal is equal to $n+1 \leq m+(m+0)$, and the idea is to use again the monotonicity of plus (in both arguments):

$$le\_plus\ n\ m : \forall a, b. n \leq m \to a \leq b \to n + a \leq m + b$$

The smart application of this term to the goal $n < 2 * m$ succeeds, generating the two subgoals $n \leq m$ and $1 \leq m + 0$. The former one is the assumption $H1$, while the latter is a *smart* variant of H.

Example 6. Suppose to have the goal $n-m \leq p$ under the assumption $n \leq p+m$. Suppose we already proved that the minus operation is monotonic

$$monotonic\_le\_minus : \forall n, a, b. a \leq b \to a - n \leq b - n$$

and also that

$$minus\_plus\_m : \forall n, m. n = (n + m) - m$$

Then the smart application of *monotonic_le_minus* would generate the following matching problem

$$(?_a - ?_n \leq ?_b - ?n) = (n - m \leq p)$$

The two (meta-)variables $?_a$ and $?_n$ get immediately unified with n and m, respectively. Hence we are reduced to look for a $?_b$ such that $?_b - m = p$. This is exactly provided by equation *minus_plus_m*, narrowing $?_b$ to $p + m$. Hence, the smart application succeeds via the substitution $\{?a := n; ?n := m; ?_b := p + m\}$, generating a new goal $n \leq p + m$ that is precisely our assumption.

Example 7. Let us make an example inspired by the theory of programming languages. Suppose to have a typing relation $\Gamma \vdash M : N$ stating that in the environment Γ the term M has type N. If we work in De Bruijn notation, the weakening rule requires lifting[3]

$$\text{weak} : \Gamma \vdash M : N \to \Gamma, A \vdash \uparrow^1(M) : \uparrow^1(N)$$

Suppose now we have an axiom stating that $\vdash * : \Box$ where $*$ and $\Box$ are two given sorts. We would like to generalize the previous result to an arbitrary (legal) context Γ. To prove this, we have just to apply weakenings (reasoning by induction on Γ). However, the normal application of *weak* would fail, since the system should be able to guess two terms M and N such $\uparrow^1(M) = *$ and $\uparrow^1(N) = \Box$. If we know that for any constant c, $\uparrow^1(c) = c$ (that comes from the definition of lifting) we may use such an equation to enable the smart application of *weak*.

Performance In Figure 5 we give the execution times for the examples of smart applications discussed in the previous section (in bytecode). Considering these times, it is important to stress again that the smart application tactics does not

[3] The lifting operation $\uparrow^n(M)$ is meant to relocate the term M under n additional levels of bindings: in other words, it increases by n all free variables in M.

take any hint about the equations it is supposed to use to solve the matching problem, but exploits all the equations available in the (imported sections of the) library.

The important point is that smart application is fast enough to not disturb the interactive dialog with the proof assistant, while providing a much higher degree of flexibility than the traditional application.

example	applied term	execution time
1	$momonotonic\_pred$	$0.16s$.
2	$momonotonic\_le\_times\_l$	$0.23s$.
3	$H : a * n \leq a * m$	$0.22s$.
4	$H : a * (Sn) \leq a * (Sm)$	$0.15s$.
5	$le\_plus\ n\ m$	$0.57s$.
6	$momonotonic\_le\_minus$	$0.14s$.
7	$weak$	$0.15s$.

Fig. 5. Smart application execution times

6 Related works and systems

Matita was essentially conceived as a light version of COQ [9], sharing the same foundational logic (the Calculus of Inductive Constructions) and being partially compatible with it (see [4] for a discussion of the main differences between the two systems at kernel level). Hence, COQ is also the most natural touchstone for our work. The `auto` tactic of COQ does not perform rewriting; this is only done by a couple of specialized tactics, called `auto rewrite` and `congruence`. The first tactic carries out rewritings according to sets of oriented equational rules explicitly passed as arguments to the tactic (and previously build by the user with suitable vernacular commands). Each rewriting rule in some base is applied to the goal until no further reduction is possible. The tactic does not perform narrowing, nor any form of completion. The `congruence` tactic implements the standard Nelson and Oppen congruence closure algorithm [21], which is a decision procedure for *ground* equalities with uninterpreted symbols; the COQ tactic only deals with equalities in the local context. Both COQ tactics are sensibly weaker than superposition that seems to provide a good surrogate for several decision procedures for various theories, as well as a simple framework for composing them (see e.g [2]).

Comparing the integration of superposition in Matita with similar functionalities provided by Isabelle is twofold complex. Isabelle's support for equational reasoning is both delegated to external tools for full scale automation and implemented internally by the so called simplifier. The quantitative comparison with the external tools Isabelle is interfaced with has been made during the ATP

System Competition where Matita ranked fourth. A qualitative comparison is less relevant, since all tools adopt some variation of the superposition calculus and are thus able to tackle the same class of problems.
Of more interest is the comparison with the internal simplifier Isabelle employs. The integration of this tool with the library is manual: only lemmas explicitly labelled and oriented by the user are taken into account by the simplifier. Moreover, these lemmas are only used to demodulate and are not combined together to infer new rewriting rules. Nevertheless, a pre-processing phase allows the user to label theorems whose shape is not an equation. For example a conjunction of two equations is interpreted as two distinct rewriting rules, or a negative statement $\neg A$ is understood as $A = False$. Finally the simplifier is able to take into account guarded equations as long as their premises can be solved by the simplifier itself.
Anyway, the main difference from the user's perspective comes from a deep reason that has little to do with the simplifier or any other implemented machinery. Since Isabelle is based on classical logic, co-implication can be expressed as an equality. The whole library Isabelle is equipped with is built exploiting this fact. CIC allows the same line of reasoning only if setoid-rewriting is employed, but it requires a non trivial machinery Matita still lacks (i.e. the proof of a rewriting step requires the system to find a proof that the context behaves as a morphism). Any comparison of the two systems with respect to equational reasoning is thus inherently biased, since problems encountered in one system would look artificial or trivial when transposed into the other one.

7 Conclusions

We described in this paper the "smart" application tactic of the Matita interactive theorem prover. The tactics allow the backward application of a theorem to a goal, where matching is done up to the data base of all equations available in the library. The implementation of the tactics relies on a compact superposition tool, whose architecture and integration within Matita have been described in the first sections. The tool is already performant (it was awarded best new entrant tool at the 22nd CADE ATP System Competition) but many improvements can still be done for efficiency, such as the implementation of more sophisticated data structures for indexes (we currently use discrimination trees).

Another interesting research direction is to extend the management of equality to setoid rewriting [27]. Indeed, the current version of the superposition tool just works with an intensional equality, and it would be interesting to try to figure out how to handle more general binary relations. The hard problem is proof reconstruction, but again it seems possible to exploit the sophisticated capabilities of the Matita refiner [3] to automatically check the legality of the rewriting operation (i.e. the monotonicity of the context inside which rewriting has to be performed), exploiting some of the ideas outlined in [26].

One of the most promising uses of smart application is inside the backward-based automation tactic of Matita. In fact, smart application allows a smooth

integration of equational reasoning with the prolog-like backward applicative mechanisms that, according to our first experimentations looks extremely promising. As a matter of fact, the weakest point of smart application is that it does not relieve the user form the effort of finding the "right" theorems in the library or of guessing/remembering their names (although it allows to sensibly reduce the need of variants of a given statement in the repository). A suitably constrained automation tactic could entirely replace the user in the quest of candidates for the smart application tactic. Since searching is a relatively expensive operation, the idea is to ask the automation tactic to return an explicit trace of the resulting proof (essentially, a sequence of smart applications) to speed-up its re-execution during script development.

Acknowledgements We would like to thank Alberto Griggio and Maxime Dénès for their contribution to the implementation of the superposition tool.

References

1. Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integrating automated and interactive theorem proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series, No. 9*, pages 97–116. Kluwer, Dordrecht, 1998.
2. Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. Uniform derivation of decision procedures by superposition. In Laurent Fribourg, editor, *Computer Science Logic (CSL), 15th International Workshop*, volume 2142 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2001.
3. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.
4. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the Calculus of Inductive Constructions. *Sadhana*, 34(1):71–144, 2009.
5. Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.
6. Andrea Asperti and Enrico Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In *Calculemus/MKM*, pages 146–160, 2007.
7. Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.
8. Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *J. Autom. Reasoning*, 29(3-4):253–275, 2002.
9. The Coq proof-assistant. `http://coq.inria.fr`, 2009.
10. Anatoli Degtyarev and Andrei Voronkov. Equality reasoning in sequent-based calculi. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 611–706. Elsevier and MIT Press, 2001.

11. Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.
12. Nachum Dershowitz, Jieh Hsiang, N. Alan Josephson, and David A. Plaisted. Associative-commutative rewriting. In *IJCAI*, pages 940–944, 1983.
13. Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. Fast term indexing with coded context trees. *J. Autom. Reasoning*, 32(2):103–120, 2004.
14. Peter Graf. Substitution tree indexing. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings*, volume 914 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 1995.
15. Joe Hurd. Integrating gandalf and hol. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–322. Springer, 1999.
16. Joe Hurd. First-order proof tactics in higher-order logic theorem provers. Technical Report NASA/CP-2003-212448, Nasa technical reports, 2003.
17. Donald Knuth and P. Bendix. Simple word problems in universal algebras. *Computational problems in Abstract Algebra (J. Leech ed.)*, pages 263–297, 1970.
18. W. McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
19. Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.
20. Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.
21. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.
22. Robert Nieuwenhuis and Alberto Rubio. Paramodulation-based thorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 471–443. Elsevier and MIT Press, 2001. ISBN-0-262-18223-8.
23. Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.
24. Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2-3):91–110, 2002.
25. Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003.
26. Claudio Sacerdoti Coen and Enrico Tassi. A constructive and formal proof of Lebesgue's dominated convergence theorem in the interactive theorem prover Matita. *Journal of Formalized Reasoning*, 1:51–89, 2008.
27. Matthieu Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.
28. Geoff Sutcliffe. The 4th ijcar automated theorem proving system competition - casc-j4. *AI Commun.*, 22(1):59–72, 2009.
29. Larry Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *J. ACM*, 14(4):698–709, 1967.