Tactics in Modern Proof-Assistants: the Bad Habit of Overkilling.

Claudio Sacerdoti Coen

Department of Computer Science
Via di Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
sacerdot@cs.unibo.it

Abstract. In this paper we will remark a common bad habit of tactic implementors in proof-assistants based on the Curry-Howard isomorphism; we name it overkilling. The wide-spreading of overkilling is due to a lack of interest in the term encoding of proofs, that leads to huge, unreadable terms. This contributes to furthermore lowering interest in the terms encoding the proofs and eventually to the concrete impossibility to create effective tools to inspect and process them.
After a general presentation of overkilling and its implications, we describe a concrete experience of fixing overkilling in the implementation of a reflexive tactic in system Coq, analyzing the gain with respect to term-size, proof-checking time and term readability.

1 Introduction

Since the development of the first proof-assistants based on the Curry-Howard isomorphism, it became clear that directly writing lambda-terms (henceforth called simply 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 terms 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 [1] in which terms are still directly written without any over-standing level of abstraction.

In particular, there are usually 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 to the system a sequence of tactics, called a script; the script is then executed to create the term encoding of the proof, which is type-checked by the kernel of the proof-assistant. 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 paper we will avoid the use of 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 as Coq [2] 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 don't add any easily accessible information to the scripts. Hence implementors have loosed interest in terms too, as far 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 leads to a polynomial asympotical reduction in the size of the terms, at the cost of an increased reduction time during retyping and compilation. Often, though, reflexive tactics could not be used; moreover, reflexive tactics suffer of the same implementative problems of 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 implementors of tactics 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 last months, though, an ever increasing number of people and projects (see, for example, HELM [4], MathWeb [5] and Formavie [3]) have been interested to switch 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 can not trust each other, hence needing retyping and making retyping time critical. Moreover, as showed by Yann Coscoy in its PhD. thesis [9] or by the Alfa interface to the Agda system [10], it is perfectly reasonable and also feasible to try to produce descriptions in natural languages of formal proofs encoded as terms. This approach, combined with the further possibility of applying the usual two-dimensional mathematic notation to the formulas that appears in the terms, is being followed by projects HELM [7], PCOQ [6] and MathWeb [5] with promising results. It must be understood, though, that the quality (in terms of naturality and readability) of this kind of 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 [11] 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 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 [12], designing proof-assistants to produce terms characterized by an high level of quality is still necessary.

In the next section we introduce a particular class of metrics for tactics evaluation. In section 3 we consider the notion of tactics equivalence and we describe one of the bad habits of tactics implementors, which is overkilling; we also provide and analyze a simple example of overkilling tactic. In the last section we describe a concrete experience of fixing overkilling in the implementation of a reflexive tactic in system Coq and we analyze the gain with respect to term-size, retyping time and term readability.

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 and a good implementation of a tactic with respect to the metric considered 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 compilers 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 1 (Of tactic). *We define a* tactic *as a function that, given a goal G (that is, a local context plus a type to inhabit) that satisfies a list of assumptions (preconditions) P, 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[1] L_i for each i in $1, \ldots, n$, returns a term $t(l)$ inhabiting G.*

Definition 2 (Of term metric). *For any goal G, a term metric μ_G is any function in $\mathbb{N}^{\{t/t \text{ inhabits } G\}}$. 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 (=term with an 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 as to be. So, in the rest of this paper, we will restrict to 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 3 (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 $\mu_G(t) = min\{\mu_G(t')/t' \text{ inhabits } G\}$.*

Using the previous notion, we can confront the behavior of two tactics on the same goal:

Definition 4. *Let τ_1 and τ_2 be two tactics both applyable 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.*

[1] We say, with a small abuse of language, that a term t inhabits a goal $G = (\Gamma, T)$ when t is of type T in the context Γ.

Note that confronting in this way "equivalent" tactics (whose definition is precised 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 confront 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 confront 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 5 (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 6 (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 7 (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.

3 Equivalent Tactics and Overkilling

We are now interested in using the metrics defined in the previous section to confront 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 confronted 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[2]. In logical frameworks with dependent types, without proof-irrelevance and with universes as CIC [13] 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 conjunction 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 for the other if a subsequent term inspects the form of the generated proof.

Put in another way, 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[3]. Convertibility, though, is a too restrictive notion that does not take in account, for example, commuting conversions. Looking for more suitable notions of equivalence is our main open issue for future work.

Definition 8 (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 9 (Of equivalent tactics). *We define two tactics τ_1 and τ_2 to be 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.*

Once we have the definition of equivalent tactics, we can use metrics, either deterministic or probabilistic, to confront 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:

---

[2] 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.

[3] 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.

Definition 10 (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 has been chosen because most of the time overkilling tactics are tactics that do not consider special cases, following the 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 another time 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 proofs to keep the simplest one.

3.1 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 in the case in which E_1 is convertible to E_2: 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 [9]. 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 terms 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, which is the reason for the user to apply it.

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 describe as another example a concrete experience of fixing a complex reflexive tactic in system Coq that has lead to really significant improvements in term size, retyping time and naturality.

4 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 [8]. The main idea is described in Fig. 1: 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 *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 *interp_ap* and *interp_sacs* are given to map the two forms of abstract polynomials to the concrete one. Finally, a theorem named *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 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 $E^{III} = (apolynomial\_normalize E^{II})$ could be shown to be equal to E^{IV} applying *apolynomial_normalize_ok*.

$$
\begin{tikzcd}
E^{I} \arrow[d, equal, "\text{apolynomial\_normalize\_ok}"'] & E^{II} \arrow[l, "\text{interp\_ap}"'] \arrow[d, "\text{apolynomial\_normalize}"] \\
E^{IV} & E^{III} \arrow[l, "\text{interp\_sacs}"']
\end{tikzcd}
$$

Fig. 1. Reflexion in Ring

$$
\begin{array}{rl}
E_1 = E_2 & \\
\| & \text{refl\_equal (because of } \beta\delta\iota\text{−convertibility)} \\
(\text{interp\_ap } E_1^{II}) = E_2 & \\
\| & \text{apolynomial\_normalize\_ok} \\
(\text{interp\_sacs } E_1^{III}) = E_2 &
\end{array}
$$

Fig. 2. Ring implementation (first half)

In Fig. 2 the first half of the steps taken by the Ring tactic to prove $E_1 = E_2$ are shown[4]. 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 *apolynomial_normalize_ok*.

Next, the two steps are done again on the left part of the equality, obtaining $(\mathit{interp\_sacs}\ E_1^{III}) = (\mathit{interp\_sacs}\ E_2^{III})$, that is eventually solved trying simpler tactics 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. 3: in Fig. 4 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 applications of *eqT_ind*, as are the two applications of *apolynomial_normalize_ok* and the three usage of reflexivity, i.e. the two applications of $refl\_eqT$ to justify the rewritings on the left and right members of the equality and the one that ends the proof.

[4] Here E_1 stands for E^I.

```
Coq < Goal ''0*0==0''.
1 subgoal

  ================
    ''0*0 == 0''

Unnamed_thm < Ring.
Subtree proved!
```

Fig. 3. A Coq session.

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 be simply 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:

$$
\begin{array}{l}
\lambda A : Type. \\
\ \lambda x, y : A. \\
\ \ \lambda H : (x == y). \\
\ \ \ (eqT\_ind\ A\ x\ [x : A]a == x\ (refl\_eqT\ A\ x)\ y\ H)
\end{array}
$$

In Fig. 5 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``))
```

Fig. 4. 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)))
```

Fig. 5. The λ-term created by the new implementation

4.1 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[5].

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 *interp_ap*, *interp_sacs* and *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 4 and 5, $|T|$ 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`.

Original version:

$1 + (\lvert E_1^{II}\rvert + \lvert T\rvert + 1) + \lvert E_2\rvert + \lvert E_1\rvert +$	(I rewriting Left)
$1 + \lvert E_1\rvert +$	(justification)
$1 + (\lvert E_1^{III}\rvert + \lvert T\rvert + 1) + \lvert E_2\rvert + (\lvert E_1^{II}\rvert + \lvert T\rvert + 1) +$	(II rewriting Left)
$(\lvert E_1^{II}\rvert + \lvert T\rvert + 1) +$	(justification)
$1 + (\lvert E_2^{II}\rvert + \lvert T\rvert + 1) + (\lvert E_1^{III}\rvert + \lvert T\rvert + 1) + \lvert E_2\rvert +$	(I rewriting Right)
$1 + \lvert E_2\rvert +$	(justification)
$1 + (\lvert E_2^{III}\rvert + \lvert T\rvert + 1) + (\lvert E_1^{III}\rvert + \lvert T\rvert + 1) + (\lvert E_2^{II}\rvert + \lvert T\rvert + 1) +$	(II rewriting Right)
$(\lvert E_2^{II}\rvert + \lvert T\rvert + 1) +$	(justification)
$1 + \lvert E_1\rvert =$	(reflexivity application)
$4\lvert E_1\rvert + 2\lvert E_2\rvert + 3\lvert E_1^{II}\rvert + 3\lvert E_2^{II}\rvert + 3\lvert E_1^{III}\rvert + \lvert E_2^{III}\rvert + 10\lvert T\rvert + 17$	**Total number**

New version, both members not in normal form:

$1 + \lvert R\rvert + \lvert E_1\rvert + \lvert E_2'\rvert +$	(Rewriting Right)
$1 + \lvert T\rvert + \lvert E_2^{II}\rvert +$	(justification)
$1 + \lvert R\rvert + \lvert E_2'\rvert + \lvert E_1'\rvert + \lvert E_2\rvert +$	(Symmetry application)
$1 + \lvert T\rvert + \lvert E_1^{II}\rvert =$	(justification)
$2\lvert E_1\rvert + \lvert E_2\rvert + \lvert E_1^{II}\rvert + \lvert E_2^{II}\rvert + 2\lvert E_2'\rvert + 2\lvert T\rvert + 2\lvert R\rvert + 4$	**Total number**

[5] 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.

New version, only the first member not in normal form:	
$1 + \|R\| + \|E_1\| + \|E'_2\| +$	(Rewriting)
$1 + \|T\| + \|E_2^{II}\| =$	(justification)
$\|E_1\| + \|E'_2\| + \|E_2^{II}\| + \|T\| + \|R\| + 2$	**Total number**

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 greatly huger than $(E_1^{II} + |T| + 1)$ and $(E_2^{II} + |T| + 1)$. 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, being m 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 map of variables to non-primitive terms 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[6]. In this way, the above inequation 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 fixing.

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 as the one given for terms size. Nevertheless, we can observe that the retyping time required is surely

[6] This has not yet been implemented in Coq.

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 fewer complexity than the type-checking of the two abstract normal forms, and the *let ... in* definitions that have the same complexity of 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 4.2 we present some benchmarks to give an idea of the real gain obtained.

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 step:

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 implementative detail that was not hidden as much as possible 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.
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.

4.2 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 1.

Term size is the size of the disk dump of the terms. Re-typing time is the user time spent by Coq in proof-checking already parsed terms. The reduction of the terms 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 on which we are interested, but also all the terms in their theories and the theories on which they depend. 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.

	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 1. Some benchmarks

5 Conclusions and Future Work

Naive ways of implementing tactics lead to low quality terms that are difficult to inspect and process. To improve the situation, we show 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. generate equivalent terms. In this work, the equivalence relation chosen for terms has simply been $\beta\delta\iota$-convertibility, that in many situations seems too strong. Hence, an important future work is the study of weaker forms of term equivalence and the equivalence relations they induce on tactics. In particular, it seems that proof-irrelevance, η-conversion and commuting conversions must all be considered in the definition of a suitable equivalence relation.

References

1. The ALF family of proof-assistants: `http://www.cs.chalmers.se/ComputingScience/Research/Logic/implementation.mhtml`
2. The Coq proof-assistant: `http://coq.inria.fr/`
3. The Formavie project: `http://http://www-sop.inria.fr/oasis/Formavie/`
4. The HELM project: `http://www.cs.unibo.it/helm/`
5. The MathWeb project: `http://www.mathweb.org/`
6. The PCoq project: `http://www-sop.inria.fr/lemme/pcoq/`
7. A.Asperti, L.Padovani, C.Sacerdoti Coen, I.Schena. Towards a library of formal mathematics. Panel session of the 13th International Conference on Theorem Proving in Higher Order Logics (TPHOLS'2000), Portland, Oregon, USA.
8. S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, TACS'97, volume 1281. LNCS, Springer-Verlag, 1997.

9. Y.Coscoy. *Explication textuelle de preuves pour le Calcul des Constructions Inductives*, PhD. Thesis, Université de Nice-Sophia Antipolis, 2000.
10. T. Hallgreen, Aarne Ranta. An Extensible Proof Text Editor. Presented at LPAR2000.
11. G. Necula, P. Lee. Safe Kernel Extensions Without Run-Time Checking. Presented at OSDI'96, October 1996.
12. G. Necula, P. Lee. Efficient Representation and Validation of Proofs. Presented at LICS'98, June 1998
13. B. Werner. *Une Théorie des Constructions Inductives*, PhD. Thesis, Université Paris VII, May 1994.