Simplifying Polynomial Expressions in a Proof Assistant

Laurent Théry

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Simplifying Polynomial Expressions in a Proof Assistant

Laurent Théry

N° 5614

Juin 2005

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--5614--FR+ENG

Simplifying Polynomial Expressions in a Proof Assistant

Laurent Théry

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lemme

Rapport de recherche n° 5614 — Juin 2005 — 16 pages

Abstract: This paper presents a simple way of performing polynomial simplifications in a proof assistant. This method has been implemented and tested within the Coq prover.

Key-words: Simplification, polynomial expression, proof system, Coq

Simplifier une expression polynomiale dans un système de preuve

Résumé : Ce papier présente une façon simple d'effectuer des simplifications sur des expressions polynomiales dans un système de preuve. Cette méthode a été implémentée et testée dans le système Coq.

Mots-clés : simplification, expression polynomiale, système de preuve, Coq

1 Introduction

The work presented in this paper has been motivated by our experiments in applying formal verification in the area of computer arithmetics [3]. We discovered very quickly that the problem was not so much in defining inside the prover notions like floating-point numbers or rounding modes than in providing proof procedures allowing to manipulate these new notions efficiently. In that respect, simplification of polynomial expressions, in particular polynomial equalities and inequalities over the reals, plays a central rôle. Before this work, making the proof system perform a simple transformation such as replacing the expression

$$y(3x + y + 2) < x(3y + 1) + z(x + y)$$

by its simpler form

$$y(y + 2) < x + z(x + y)$$

was not a trivial task. One way to go to solve this problem is to extend the user-interface of the proof system in order to let the user indicate which simplification he or she wants to perform. For example, the user could pilot the simplification with the mouse device mimicking what is usually done on a blackboard:

$$y(\cancel{3x} + y + 2) < x(\cancel{3y} + 1) + z(x + y)$$

This approach, called direct-manipulation, has already been experimented in some systems (see for example [6,1]). Another way to go is to make the proof system "guess" the simplified term. It is the approach followed by this paper.

The problem of finding a simpler form of a given expression is not new and has already been studied, in another setting, for computer algebra systems. For example, Moses details some of the technical issues in performing algebraic simplifications in [7]. Most of these ideas can easily be applied to theorem proving applications. Still, we believe that the fact that we are building proofs rather than just performing symbolic computations changes slightly the perspective.

One aspect that is crucial in our application is that most of the proofs we are doing cannot be fully automated. The proof process is then mainly done in an interactive fashion. This means that the user needs to have a clear understanding of what each proof step is doing. It implies that our procedure

needs to give a precise meaning to the term simplifying. It should also try to preserve as much as possible the shape of the goal it is trying to simplify. Avoiding unnecessary shuffling of the expression makes it easier to understand the result of the simplification procedure. This is precisely these two issues that we try to address in the following. We first present the idea. Then we give the algorithm. Finally we sketch its implementation within the Coq prover.

2 Basic Idea

The idea of our procedure is very simple. We take as premises that the procedure should not change the structure of the expression. This seems a rather strong assumption. It implies in particular that we do not allow to distribute products over sums. Using distributivity could lead to a perfectly acceptable procedure. For the example given in the introduction

$$y(3x + y + 2) < x(3y + 1) + z(x + y)$$

it would first consist in fully expanding the products on both sides of the inequality

$$3xy + yy + 2y < 3xy + x + zx + zy$$

and then in cancelling identical monomials

$$yy + 2y < x + zx + zy$$

Still we believe that the simplified form we propose in the introduction

$$y(y + 2) < x + z(x + y)$$

should be favoured. As this form has preserved much of the structure of the initial formula, we get a more immediate understanding of what the simplification procedure actually did. Also, preserving the structure gives a direct meaning of what simplifying is about: a formula is simpler if it is structurally smaller.

Once we have taken the requirement to preserve the structure of the polynomial, we do not have much choice. Our only degree of freedom is to play with the coefficients of its monomials. More precisely, trying to find a simplified form of the formula that only differs from the original formula by the

coefficients of its monomials. On our example, it means to be able to fill with numbers the boxes of the pseudo-expression

$$\square y(\square x + \square y + \square) < \square x(\square y + \square) + \square z(\square x + \square y)$$

to obtain an equivalent simplified expression. There is of course a trivial solution: the initial expression

$$\boxed{1} y(\boxed{3} x + \boxed{1} y + \boxed{2}) < \boxed{1} x(\boxed{3} y + \boxed{1}) + \boxed{1} z(\boxed{1} x + \boxed{1} y)$$

Obviously the most zeros we can write in boxes, the better it is. We can reach the expected expression with the following assignment

$$\boxed{1} y(\boxed{0} x + \boxed{1} y + \boxed{2}) < \boxed{1} x(\boxed{0} y + \boxed{1}) + \boxed{1} z(\boxed{1} x + \boxed{1} y)$$

Applying the absorption law on this expression gives us the simplified expression given in the introduction.

3 The algorithm

Once given the idea, deriving an algorithm is straightforward. It is composed of eight steps.

Lifting coefficients

The first step consists in moving all the coefficients to the head of the different products. In order to make sure that every product starts with a coefficient we add an explicit one coefficient when needed. For example, given the following expression

$$3x2y + z$$

lifting the coefficients gives us

$$6xy + 1z$$

For the example given in the introduction, we need to consider the expression

$$(x(3y + 1) + z(x + y)) - y(3x + y + 2)$$

Lifting its coefficients does not do much. It just adds some extra one coefficients

$$(1x(3y + 1) + 1z(1x + 1y)) - 1y(3x + 1y + 2)$$

Adding variables of position

After the first step, we are sure that all products start with a coefficient. These coefficients are the only ones that can actually be modified in the simplification process. To do so, we associate to each of them a distinct variable name. These new variables are called variables of position. While creating them we also memorize the mapping that associates each of them to their initial value. In the following we use the convention that $a, b, c, \ldots$ denote variables of position while $x, y, z, \ldots$ denote real variables of the polynomial. On our example, we get

$$(ax(by + c) + dz(ex + fy)) - gy(hx + iy + j)$$

with

$$\Phi = \{a \to 1, b \to 3, c \to 1, d \to 1, e \to 1, f \to 1, g \to 1, h \to 3, i \to 1, j \to 2\}$$

Getting the list of monomials

In the third step, we apply the distributivity to get the list of all the monomials. Each monomial starts with its position part and ends with its real part. On our example we get

$$L = \{abxy, acx, dexz, dfyz, -ghxy, -giyy, -gjy\}$$

Putting together related monomials

In the list of monomials, there cannot be two monomials with identical position part (positions are unique) but two monomials can have an identical real part. Having an identical real part simply indicates a possible cancellation. In the forth step, we split the list of monomials and construct a mapping that associates each distinct real part to the list of variable parts they appear with in the list L. On our example we get

$$\begin{aligned}\mathcal{M} = \{\ & x \to \{ac\},\ y \to \{-gj\},\ xy \to \{ab, -gh\},\ xz \to \{de\},\\ & yy \to \{-gi\},\ yz \to \{df\}\}\end{aligned}$$

Getting the system of equations

Now we are ready to turn our problem into a resolution of a system of equations. Each element of the mapping $\mathcal{M}$ indicates a possible cancellation. We just need to preserve that the sum of the elements of the each list is equal to the sum of their initial value given by the mapping Φ. On our example we get

$$E \begin{cases} ac &= 1 \\ -gj &= -2 \\ ab - gh &= 0 \\ de &= 1 \\ -gi &= -1 \\ df &= 1 \end{cases}$$

Splitting the system in subsystems

Before solving the system, we first check if the system could not be split in independent subsystems, i.e. subsystems with no variable in common. In that case, solving the overall system can be more efficiently done by individually solving its subsystems and combining the solutions. On our example, we can split our system in two subsystems:

$$E_1\{a,b,c,g,h,i,j\} \begin{cases} ac &= 1 \\ -gj &= -2 \\ ab - gh &= 0 \\ -gi &= 1 \end{cases}$$

$$E_2\{d,e,f\} \begin{cases} de &= 1 \\ df &= 1 \end{cases}$$

Solving the subsystems

We are interested in solutions of the system that would maximize the number of zeros. For this, we are going to perform a heuristic search using a slightly modified version of the Davis-Putnam algorithm for satisfying boolean formulae [4]. While for boolean formulae the search is done on a binary tree, in our case we do it on a ternary tree. Each node selects an unassigned variable that

has not yet been selected by any of it ancestor nodes. Each of this node generates three subtrees. The first subtree assigns the selected variable to zero. The second one assigns the variable to its initial value. The last one leaves the variable unassigned.

Success We reach a leaf in the search tree when it is not possible to create any new node. In that case, we can check if the assignment associated with the leaf is a solution. If so, in order to get the best solution, we compare it with the best solution found so far and keep the one with a major number of zeros. Two things are worth noticing. First, the search always finds a solution since it always encounters the initial assignment Φ. This is a direct consequence of the fact that each node tries as a second alternative the assignment to the initial value. Second, the assignment at a leaf may not cover all the variables since we allow nodes to leave variables unassigned.

Early Failure Detection There are two cases where it is possible to stop the search before reaching a leaf. The first one is when after an assignment at a node the corresponding partial solution already produces an unsatisfiable equation, i.e an equation $n = m$ where n and m are two different numerical constants. The second case is when after an assignment at a node the partial solution could not be completed with enough zeros to beat the current best solution.

Propagation Each time after an assignment at a node we check if the partial solution produces an equation with a single variable $nu = m$ with n and m numerical constants. In that case we can directly perform the assignment $u \leftarrow m/n$. The propagation speeds up the discovery of a solution. It is the equivalent of the unit resolution propagation of the Davis-Putnam algorithm. Of course there can be a cascade of such assignments. Note that when we are applying the simplification on polynomials with coefficients in a ring (and not a field) we also need to check that n divides m. If it is not the case, it produces an early failure.

Before showing how the search proceeds on our example, we first present a simple example to motivate our choice of a ternary search tree. The example we consider is the expression $x + x$. Adding the variables of position gives

$ax + bx$, so the system has only one equation $a + b = 2$. Figure 1 gives a graphical representation of the search. The tree starts with the variable a. Setting it to 0 and after the propagation gives us the simplified expression $0x + 2x$. Setting to 1, its initial value, and after the propagation, gives back the initial formula $x+x$. Finally, skipping the variable a, let us try to assign the second variable b. Setting b to 0 assigns the value 2 to a by progation and gives the simplified expression $2x + 0$. Setting b to 1 gives again the initial solution after propagation. Skipping b fails since there is no more variable to assign. Our search has found the 3 possible solutions: leaving the formula unchanged, cancelling the first x, cancelling the second x.

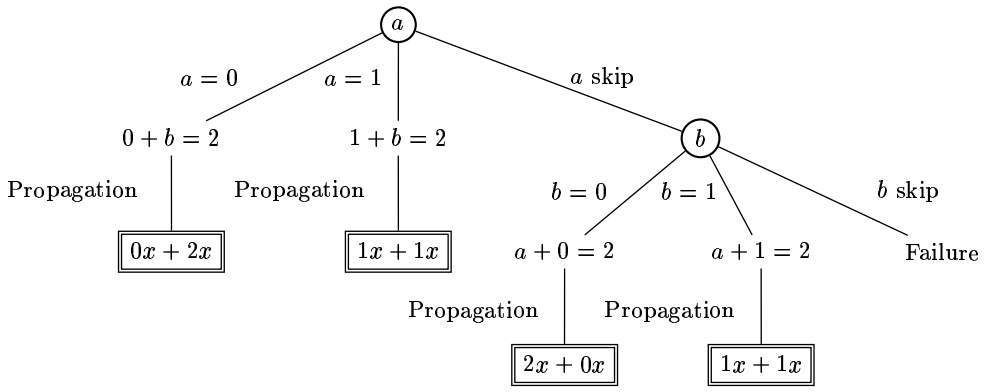

Fig. 1. Search for $x + x$

Now let us consider our example. A partial view of the search is presented in Figure 2. It shows how the search finds a first solution with 2 zeros $\{a \to 1, b \to 0, c \to 1, g \to 1, h \to 0, i \to -1, j \to 2\}$. Completing the search it is easy to check that this is the best solution indeed.

Producing the simplified expression

Once a solution is found, to get the simplified expression we simply need to substitute the variables of position in the expression produced by the second step with their associated values in the solution. Removing unnecessary 0 and

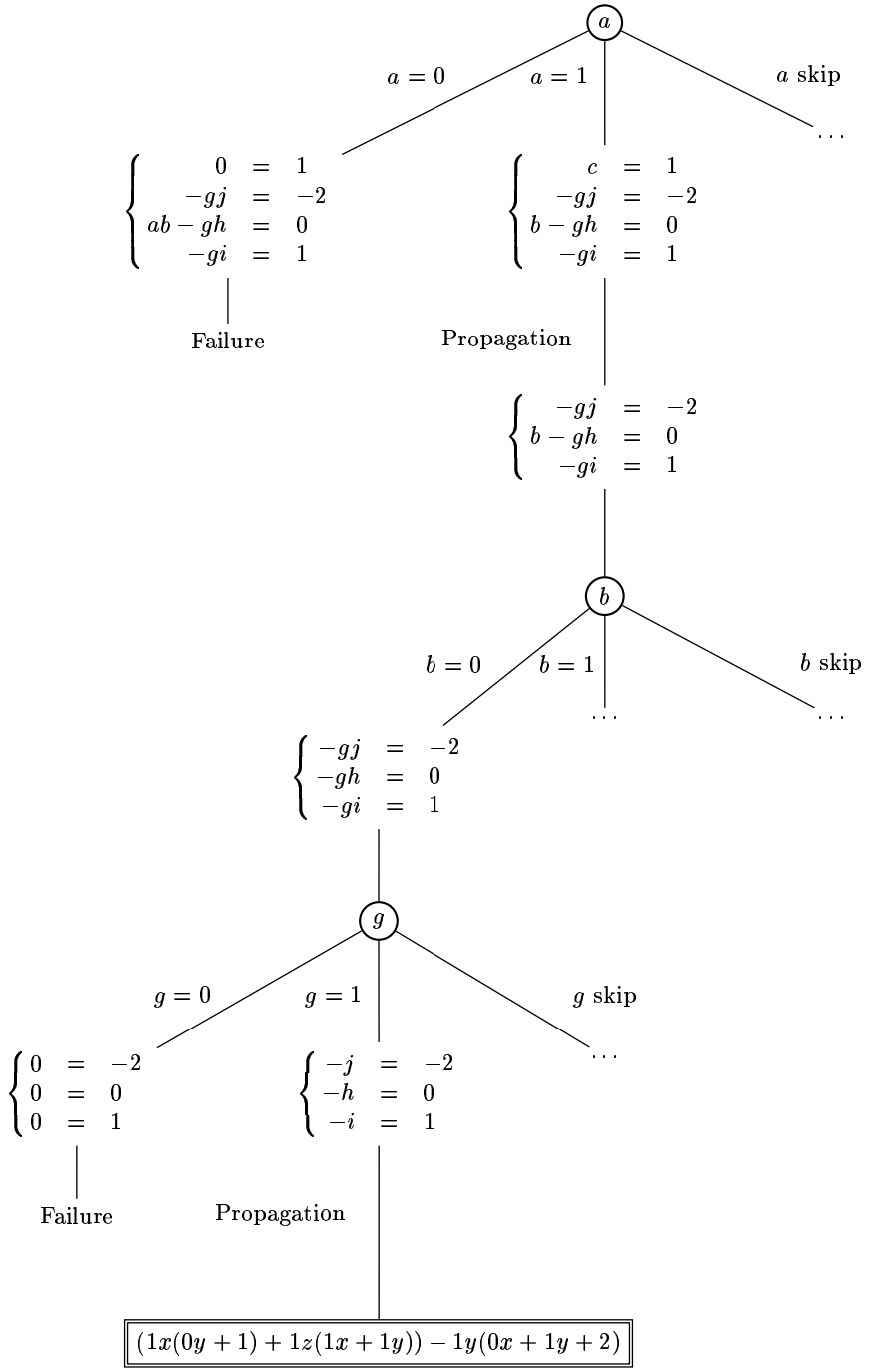

Fig. 2. Search for $(x(3y+1)+z(x+y))-y(3x+y+2)$

1 gives the final simplified expression. On our expression we get the expected expression

$$(x + z(x + y)) - y(y + 2)$$

Note that in order to simplify inequality such as

$$y(3x + y + 2) < x(3y + 1) + z(x + y)$$

the trick is to first move everything to the right side of the inequality

$$0 < (x(3y + 1) + z(x + y)) - y(3x + y + 2)$$

We then apply our algorithm on this right side and get

$$0 < (x + z(x + y)) - y(y + 2)$$

Now as our algorithm preserves the structure, we are sure that the simplified expression has a substraction as top operator[1]. So we can put the right part of the substraction to the left of the inequality

$$y(y + 2) < x + z(x + y)$$

3.1 Implementation

The implementation is a naïve transposition of the algorithm in a functional setting. It has been directly encoded inside the logic of Coq. To do so, we first define a type *expr* that represents a free algebra parametrized over an arbitrary set of coefficients C

Inductive $expr : Set :=$

$\quad \oplus : expr \to expr \to expr$
$\mid \quad \ominus : expr \to expr \to expr$
$\mid \quad \otimes : expr \to expr \to expr$
$\mid \quad \ominus : expr \to expr$
$\mid \quad c : C \to expr$
$\mid \quad v : \mathbb{N} \to expr.$

[1] This is almost true. As after injecting the solution we perform some trivial simplifications, an expression like $x - 0$ could be reduced into x. An easy fix is to prohibit such a simplification on the top operator if it is a subtraction

Variables are indexed by natural numbers. We require that C contains a zero and a one, some associated operations for addition, opposite, multiplication, division and some tests for equality to zero, equality to one and divisibility. The first step of the algorithm is represented by the function

$$lift\_const : expr \to expr$$

Steps 2 and 3 are performed in a single function

$$make\_list : expr \to env \to \mathbb{N} \to (list\ mon * env * \mathbb{N})$$

where the type *env* represents the mapping from variable of positions to value and the type *mon* represents the monomial composed of a sign (-1 or 1), a variable part and a real part

Definition $env := (list\ (\mathbb{N} * C))$.

Definition $mon\_pos := list\ \mathbb{N}$.

Definition $mon\_exp := list\ \mathbb{N}$.

Definition $mon := (C * mon\_pos * mon\_exp)$.

Natural numbers are also used to index variables of position. The result of the function *make_list* is a triple composed of the list of monomials, the initial assignment of variables of position, and the next free index for a variable of position.
Step 5 is represented by the function

$$make\_groups : (list\ mon) \to list\ (mon\_exp * list\ factor)$$

It returns a mapping from the monomial parts to the corresponding list of factors with
Definition $factor := (C * mon\_pos)$.

Steps 5 and 6 are performed in a single function

$$make\_system : env \to (list\ group) \to list\ ((list\ \mathbb{N}) * list\ equation)$$

It returns a list of subsystems. A subsystem is composed of a list of all the variable of positions it contains and a list of equations with

Definition $equation := C * list\ factor$.

The first argument of *make_system* is the initial value of the variables of position that is used to generate the first element of each equation by evaluating the associated list of factors.
Step 7 is the more elaborated one. It is represented by the function

$$search\_best : ((list\ \mathbb{N}) * list\ equation) \to env$$

The only limitation in Coq when writing recursive functions is to have an argument that structurally decreases in all the recursive calls. This limitation prevents infinite computation. While all the other functions have such an argument, the direct implementation of the search does not satisfy this condition. The problem comes from the propagation step where we do not know from the arguments of the function the depth of the recursive calls it will require. A standard work-around is to add an extra argument just to ensure termination. For the propagation, we can simply bound the depth of recursive calls by the number of variables of position.
The last step is represented by the function

$$make\_expr : expr \to env \to expr$$

that replaces the constant in the expression by its corresponding value in the solution. Combining all these functions in a single function we get

$$simpl : expr \to expr$$

Now that we have our algorithm implemented, we need to define two functions to convert from our domain of application to our free algebra and back. For example, given the expression over the real

$$|x|(e^x + e^y) - |x|e^x$$

the first function *r_to_expr* should turn it to an expression of type *expr*

$$v_1 \otimes (v_2 \oplus v_3) \ominus v_1 \otimes v_2$$

keeping the association

$$\{v_1 \to |x|,\ v_2 \to e^x,\ v_3 \to e^y\}$$

Now we can applied the simplification getting

$$v_1 \otimes v_3 - 0$$

The second function *expr_to_r* now uses the association to get back an expression over the real

$$|x|e^y - 0$$

While it is always possible to write the second function inside the logic of Coq, we need to use the meta-language (the tactic language) of Coq for the first function since the polynomial expression of the domain of application is usually not inductively defined.

You are not yet done since if e is the expression we want to simplify and e' its simplified form, we still need Coq to accept the statement $e = e'$. As the algorithm of simplification is writing in Coq, a canonical way to do this is to prove once and for all that the simplification algorithm is correct. This is an instance of the so-called reflective method [2]. We could obviously do so but as we have seen, the algorithm keeps on changing data representations and this type of algorithm is known to be difficult to prove correct. So the proof would not be far less easy than its implementation. We have taken an alternative approach using the algorithm just as an oracle. To assert the final equality statement, we use an already existing decision procedure over polynomial expressions called `ring` and based on the reflective method. Doing so, every run of our simplification algorithm is checked by the decision procedure.

Using Coq itself to provide the oracle is somewhat atypical, one usually uses an external oracle as in [8] or the tactic language. As a matter of fact the first version of the tool was using a version of the algorithm directly written in Ocaml. We believe that our new implementation is far more flexible. First, it leaves open both the possibilities of proving the code correct and extracting it to get a Ocaml code that can be efficiently compiled. Second, it provides a more portable implementation of our tactic since most of what the tactic does is done inside the logic of Coq. Finally, performing a fair amount of computation inside the logic of Coq is not a problem. The spectacular progress of the evaluation mechanism inside the logic of Coq as described in [5] ensures a computing power that is more than sufficient for our need.

4 Running example

The code of the tactic is available at

`http://www-sop.inria.fr/lemme/Laurent.Thery/polsimpl.zip`

After loading it

```
Require Import PolSimpl.
```

one gets a tactic `pol_simpl` that has been specialized to perform polynomial simplifications over the reals and the integers. For example, if the goal to solve is the following

```
test < 1 subgoal

  x : R
  y : R
  z : R
  ============================
   x * (z + y + 2) < y * (3 * x + 1)
```

Applying the `pol_simpl` tactic produces the following subgoal 1 subgoal

```
test < pol_simpl.
1 subgoal

  x : R
  y : R
  z : R
  ============================
   x * (z - 2 * y + 2) < y
```

5 Conclusions

In this paper we have presented a simplification procedure for polynomial expressions. Our initial motivation was to equip the Coq system with a simplification procedure that would perform the "obvious" simplifications automatically. We are aware that what is presented here is just a first step. Inequalities

we are dealing with in our application are not just polynomial expressions but also contain functions like exponentials, absolute values. More specific knowledge has to be added regarding these functions in order to perform appropriate simplifications. Still, we believe that our procedure has some nice properties. It has a clear semantics with a rather direct implementation. We believe it provides a good comprise between performing simplifications and preserving the structure of the formulae.

From the implementation point of view, our procedure is interesting. We took advantage of the fact that the equality over polynomial expressions is decidable. While in the reflective approach the code is proved correct once and for all, here we use the decision procedure to check the result of the simplication every time. This is the price to pay for not proving the correctness of our simplification procedure. Writing the oracle inside the logic still makes sense. It leaves open the possibility to use the reflexive method and it increases portability. For example, a reimplementation of the logic of Coq could benefit of our procedure without having to implement the meta-language.

References

1. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 141–160, Sendai, Japan, 1994.
2. Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *TACS'97*, volume 1281 of *LNCS*, pages 515–529, Sendai, Japan, 1997.
3. Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, number 2152 in LNCS, pages 169–184, Edinburgh, Scotland, 2001.
4. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
5. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming*, ACM SIGPLAN, pages 235–246, Pittsburgh, PA, Usa, 2002.
6. Paracomp Inc. *Milo User's Guide*, 123 Townsend St., suite 310, San Francisco, 1988.
7. Joel Moses. Algebraic simplification: a guide for the perplexed. *Commun. ACM*, 14(8):527–537, 1971.
8. Laurent Théry. Stålmarck's Algorithm in Coq: A Three-Level Approach. Technical Report 4353, INRIA, 2002.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399