Nodesat, an arithmetical tactic for the Coq proof assistant

Daniel Hirschkoff

avril 1996

N^o 96-61

Nodesat, an arithmetical tactic for the Coq proof assistant

Daniel Hirschkoff

Résumé

Ce rapport décrit l'implantation de Nodesat, *une procédure de décision arithmétique pour le système d'aide à la preuve Coq, qui est développé à l'INRIA (Rocquencourt) et à l'École Normale Supérieure de Lyon. À partir d'un système de comparaisons entre expressions arithmétiques, l'algorithme construit un graphe orienté et valué dans lequel il essaye de mettre en évidence des cycles de poids strictement positif, ce qui revient à vérifier le système initial. Lorsque la procédure vérifie ce système, un terme de preuve est construit à partir de ce qui est en fait une trace des différentes étapes de l'algorithme. Il s'agit pour le système Coq d'une première tentative d'écriture de procédure de décision arithmétique, un outil qui se révèle indispensable dans le cadre du développement d'applications industrielles.*

Abstract

This report describes the implementation of Nodesat, *an arithmetical decision procedure for the Coq proof assistant, a system developped at INRIA (Rocquencourt) and at the École Normale Supérieure (Lyon). The algorithm builds a directed, valuated graph representing a system of comparisons between arithmetical expressions, and looks for a strictly positive cycle graph in order to check the validity of the system. If the procedure succeeds, a proof term is generated, which is basically a trace of the various steps of the algorithm. This attempt is a first step towards implementation of an arithmetical decision procedure for Coq, a feature that turns out very useful for industrial applications.*

Introduction

The conception of many proof checkers is based on typed λ-calculi; the underlying paradigm of such a design is that building a proof is analogous to writing a program. This idea, usually referred to as the Curry-Howard isomorphism, is a result giving two interpretations of a term in a typed λ-calculus: on one side there is an algorithm, on the other side a method to prove a specification given by the type of the term. While this beautiful result can indeed be used as the foundation of a proof-checking system, the confrontation with "realistic" proofs makes visible the fact that the complexity gap is quite big between terms used for implementation and terms used for proofs. As one wants to prove a non straightforward result, technical details often arise, even if the objects are simply and naturally implemented into the underlaying λ-calculus. This results in a waste of time, and even more importantly in a loss of understandability in the proof scripts, where the technical parts play the main rôle. To avoid that, logical frameworks that are designed towards real size proofs come with decision procedures, or in other words tools for building in an automatic way technical proofs; these decision procedures are usually concerned with topics like propositional logic, finite state model-checking, linear arithmetic or rewriting.

We describe here *Nodesat*, which is a first attempt to write an arithmetical decision procedure for the Coq proof assistant [CCF+96]. Although this tactic is not very powerful in terms of complexity of the problems treated, it can be considered as a first tool that gives Coq users an automatic way to avoid technical proofs using arithmetical results.

After sketching briefly the conception of the proof system, we describe the algorithm used by *Nodesat* and compare it to existing systems. We then discuss implementation, giving a few examples of the use of *Nodesat*, and we conclude considering improvements that can be brought to the tactic.

1 The Coq proof assistant

1.1 The system

Coq [CCF+96] is a proof assistant based on the *Calculus of Inductive Constructions*, a higher-order logic which gives to this system a strong expressiveness as well as a reasonable flexibility. Inductive definitions are introduced into the system *via* their constructors, and automatically generate elimination schemes that can be used by proofs by induction.

Coq proofs are led in a goal-directed way through the application of

tactics implementing backward reasoning: at any point in the proof a tactic can be invoked, and if it succeeds, i.e. if it matches (in some way depending on the tactic that is actually used) the current goal to be proved, it generates one or several subgoals (or even none if the proof is complete).

Example `Prop` is the type for logical properties; the `Split` tactic transforms a conjunctive goal into two subgoals corresponding to the two members of the conjunction:

```
1 subgoal

  P : Prop
  Q : Prop
  H : P
  H0 : Q
  ============================
   P/\Q

Unnamed_thm < Split.
2 subgoals

  P : Prop
  Q : Prop
  H : P
  H0 : Q
  ============================
   P

subgoal 2 is:
 Q
```

1.2 Arithmetic in Coq

Among other definitions, the type `nat` for natural integers is automatically loaded in a Coq session. It is defined in an inductive way, following Peano's axiomatisation, as follows:

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat.
```

This definition reads: "`nat` *is a new inductive type of sort* `Set` *whose constructors are* `O` *of type* `nat` *(a constant) and* `S` *of type* `nat -> nat` *(the successor function, taking a natural number and returning as a result a natural number)*".

Integers can then be defined according to Valérie Ménissier-Morain's implementation[1], as:

```
Inductive Z : Set :=
  OZ : Z
| pos : nat -> Z
| neg : nat -> Z.
```

An integer (type `Z`) is either `OZ`, or `(pos n)` (denoting `n`+1), or `(neg n)` (denoting −`n`−1).

Along with these inductive types, the usual operations and relations are defined on natural numbers (`plus, mult, le, lt, ge, gt`) and on integers (`addZ, multZ, leZ, ltZ, geZ, gtZ`), as well as the canonical injection `injp` (of type `nat -> Z`).

As one can easily guess, this very primitive implementation of integers turns out tedious to handle in big proofs, since many straightforward results on integers are not at all given for free with the definitions of the objects (for example, the proof that $(\mathbb{Z}, +)$ is a group takes more than 350 lines in V. Ménissier-Morain's original implementation). Therefore, a tool for generating automatically the proofs for such straightforward results can really be useful to the user, especially if he/she is dealing with a proof where the arithmetical results are not important at all: the purpose is here to give to the effectiveness of theorem proving enough power to "catch-up" the high expressiveness of its logic.

1.3 Writing a tactic for Coq

The Coq V5.10 source CamlLight code provides user-friendly functions that can be used to write one's own tactics; these functions are used for parsing or unparsing of terms, as well as for manipulating them (applying weak head beta-reductions, for example). This makes Coq a quite *open* system, especially since this opportunity is documented in [CCF+96]. The integration of extra code to the system is quite easy, and does not make the requirement of understanding the whole system. The Coq proof assistant has already a few tactics implementing decision procedures written this way, like `Tauto` for intuitionistic propositional calculus [Muñ94] and `Linear` for *Direct Predicate Calculus* [Fil95].

[1] This implementation can be found in the Coq `contrib/Rocq/ARITH/Chinese/` directory. Another implementation defines $\mathbb{Z}$ as the symmetric completion of the semi-group $\mathbb{N}$; it can be found in the `contrib/Rocq/RATIONAL/Integer` directory; it is also used in the LEGO system [LP92], for example.

2 The decision procedure

We present here the algorithm used by *Nodesat*; it is actually the algorithm used by the *Arith* tactic of the NuPrl system [Jac94] and described in [Cha77] under the name of "the `nodesat` procedure". Roughly, this algorithm works on comparisons between arithmetical expressions by building a graph and representing these comparisons on it.

Other approaches can be considered, the mostly used being the SupInf method for Presburger formulae, as stated by Bledsoe [Ble75] and improved by Shostak [Sho77]. The NuPrl system has actually two arithmetical tactics, namely *Arith* (implementing the `nodesat` procedure) and *SupInf* (implementing Bledsoe's method); this latter method is also used by the HOL *arith* library [Gor, Bou92], as well as the PVS system [SOR93].

2.1 The `nodesat` procedure

2.1.1 Looking for a contradiction

The procedure described in [Cha77] is called the `nodesat` procedure. As will be shown in the following, it builds a graph from a list of comparisons on arithmetic expressions. Let us first state the form of the proposition that has to be checked by the algorithm we shall describe:

$$G = H_1 \wedge \cdots \wedge H_n \Rightarrow C_1 \vee \cdots \vee C_m$$

$$H_i, C_j = A \mid \neg A$$

$$A = P < Q \mid P \leq Q \mid P > Q \mid P \geq Q \mid P = Q$$

The first basic idea of the `nodesat` procedure is to add the negation of the conclusion to the system built on the hypothesis of the proposition, and to look for a contradiction. We now have thus the system:

$$H_1 \wedge \cdots \wedge H_n \wedge \neg C_1 \wedge \cdots \wedge \neg C_m \Rightarrow \bot$$

In other words, from this new system of comparisons (since the negation of a comparison on arithmetic expressions still can be formulated as a comparison, think for example of $\neg(x \leq y) \equiv (x > y)$), the algorithm will try to prove it *unsatisfiable*, i.e. that no instantiation of its variables can satisfy the constraints expressed by the comparisons. Conversely, if an instantiation that satisfies all the comparisons can be found, the system is said *satisfiable*, and the original proposition is proved false.

2.1.2 Normalisation of comparands

We now have comparisons on arithmetical expressions, and we want to work with them in order to find a contradiction. That for, we need them to look "all the same way", through a normalisation of the comparands; these are originally in the following form:

$$P = x \mid k \mid -P \mid P_1 + P_2 \mid P_1 * P_2,$$

and should finally look like:

$$P = \sum_{i_1=0\ldots n_1,\ldots,i_n=0\ldots n_n} C_{i_1\ldots i_n} x_1^{i_1} \ldots x_n^{i_n}$$

In other words, we rewrite an arithmetical expression into a sum of monoms of the various variables involved in the proposition we are considering. For example the expression $2\,(x+1) + y\,(x-3)$ is rewritten into $2 + 2.x + (-3).y + 1.xy$. Note that properties used for this rewriting of terms into normal form are the basic properties of the ring of integers.

2.1.3 Linearisation of comparands

At this point of the algorithm, we have comparisons on "normal terms", the next step is the linearisation of these terms. This is actually a key step in the `nodesat` procedure, since it can discard much information from the initial problem. What we do is basically to *linearise* all the comparands, that is to replace a term of the form $P = \sum_{i_1=0\ldots n_1,\ldots,i_n=0\ldots n_n} C_{i_1\ldots i_n} x_1^{i_1} \ldots x_n^{i_n}$ with a term of the form $c + y$ (c a constant), creating a new, fresh variable y that will mask all the non-constant part of the original normal term. This is needed by the forecoming steps of the algorithm, but can of course ruin all the efficiency of the algorithm, as can be seen in the two following examples:

- suppose we have to prove

  $$(x^2 \leq 0) \Rightarrow (x = 0)$$

  We first take the negation of the conclusion, and get the system

  $$(x^2 \leq 0) \wedge (x <> 0) \Rightarrow \bot$$

  The normalisation of terms gives then

  $$(1.x^2 \leq 0) \wedge (1.x <> 0) \Rightarrow \bot$$

(which is still contradictory), and the linearisation of comparands creates two new variables y and z, leading to the non contradictory system

$$(y \leq 0) \wedge (z <> 0) \Rightarrow \bot$$

The `nodesat` procedure will fail here, on a really simple system, because it "forgets" the relation between $y(= x^2)$ and $z(= x)$ ...

- There is an even more simple example of the disaster the linearisation can cause[2]: suppose we have the system

  $$(n > 0) \wedge (-n \geq 2) \Rightarrow \bot$$

  The linearisation gives

  $$(y > 0) \wedge (z \geq 2) \Rightarrow \bot$$

  ... which is of course not provable.

As said before, a lot of informations can get lost during this linearisation step. Still, if we succeed to find a contradiction in the linearised system, the original proposition is proved true; on the other hand, if we fail, we cannot state the original proposition as false, since we cannot know if the failure comes from a weakening of the hypothesis during the linearisation. That is the reason why this so-called decision procedure is in fact only a semi-decision procedure [3].

2.1.4 Normalisation of comparisons

After the linearisation step, we have a list of comparisons looking like

$$H = A \mid \neg A$$

$$A = P < Q \mid P \leq Q \mid P > Q \mid P \geq Q \mid P = Q$$

and that involve linearised terms, ie terms described by

$$P, Q = c \mid y \mid c + y$$

[2] Such cases are avoided in the actual implementation, due to the improvements brought by V. Ménissier-Morain. See the next section.

[3] In some cases, we can be sure that there is no loss of information due to the linearisation; we thus have a real decision procedure and answer "true" or "false" to the original goal.

We transform these comparisons into *normal comparisons* such as

$$N \;=\; P \geq Q \mid P = Q \mid P <> Q$$

What we do is in fact keep the equalities and non-equalities as they are, and transform all the inequalities into greater-than inequalities, using common theorems such as $x < y \Rightarrow y \geq x + 1$ (in this case, we need of course to rewrite a term like $x + 1$ into a "linearised form" as described above, thus invoking some kind of "relinearisation", but in this case we know it is safe, because of the form of the original terms).

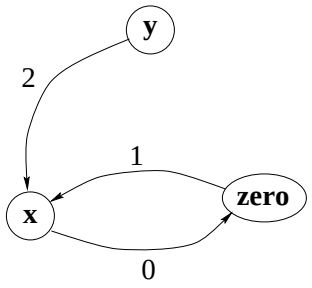
2.1.5 The graph of inequalities

Building the graph Once we have all the *normal comparisons* of *linear terms*, we can start the real procedure, namely by building the graph of inequalities, which will be the support for our reasoning. This graph is an oriented, weighted graph, whose nodes represent the variables we are considering (linearised variables, of course), and whose edges represent the information we have between these nodes, in terms of greater-than inequalities; there is an extra node for constant terms, in order to represent inequalities such as $x \geq 3$. On this graph, an edge from node x to node y, weighted by constant c, means: $x \geq c + y$. The particular case of constant terms is treated as follow: the node *zero* represents 0, and the inequality $x \geq 3$ is represented by an edge from node x to node *zero*, of weight 3.

As we want to represent our problem on this graph, the first thing we do is to "translate" our inequalities onto it, by drawing the corresponding edges (for the moment, we forget about equalities and inequalities). For example the following system of inequalities:

$$(x \geq 0) \wedge (y \geq x + 2) \wedge (1 \geq x)$$

is represented by the graph:

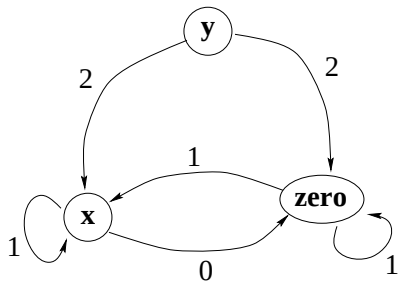


An important fact to notice about the weight of an edge is that its role is to represent the *maximum* weight between two nodes: suppose we have the two inequalities $x \geq y + 1$ and $x \geq y + 2$, we will draw an edge of weight 2 from node x to node y, since the second inequality is more restrictive (and hence gives more information) than the first.

Propagating the weights The main operation that we will perform on the graph, each time we add some new information to it, is to propagate this information; since this information is represented by the weights of the edges on this graph, this will be done through a "propagation" of the weights. That for, we use the transitivity of inequality, stated as follows: $(x \geq y + c) \wedge (y \geq z + d) \Rightarrow (x \geq z + (c + d))$, which translates on the graph to the assignment

$$A_{ij} := max(A_{ij}, max_{k=0,1,\ldots,n}(A_{ik} + A_{kj})),$$

if A_{ij} is the weight of the edge between nodes i and j. On the previous example, the inequalities $(y \geq x + 2)$ and $(x \geq 0)$ lead to the new inequality $y \geq 2$, represented by a new edge of weight 2 from y to *zero*. Similarly, by taking $i = j$ in the above assignment, we can deduce the weight for an edge from one node to itself: from inequalities $(x \geq 0)$ and $(1 \geq x)$, we deduce $(x \geq x + 1)$, that is represented by a loop on the graph. If we propagate the information while building the graph of our example, we get the following graph:



Finding a contradiction Since what we are looking for is a contradiction, we have to know what it means in terms of inequalities, and how it is represented on the graph. The `nodesat` procedure is in fact looking for cycles with a strictly positive way, since they correspond (modulo propagation) to an inequality of the form $x \geq x + c$, with $c > 0$, which is obviously absurd. We are hence looking for edges from a node to itself with weight > 0, such as edges like the one from node x to x or from node *zero* to *zero* on our example (which was trivially contradictory).

If we have such edges after construction of the graph of inequalities and propagation of weights (like in our example), a contradiction is found and the `nodesat` procedure terminates with a success; if not, we must consider the other informations we have, namely the equalities and the inequalities: this is the next step in the `nodesat` procedure.

2.1.6 Getting rid of equalities

What we want to do now is to get rid of equalities, by putting all the information they represent on the graph of inequalities. This is done by restricting the size of the graph according to these equalities, and propagating the information they give.

The way to proceed is this: suppose we have the graph of inequalities, with nodes x and y representing the corresponding variables, and we know as well the equality $x = y + k$. What we do is delete the node x and bring all informations about x on the node y: suppose we have an edge from node say z to node x of weight c, this edge "says" $z \geq x + c$, and since $x = y + k$, we can *update* the weight of the edge from z to y with weight $c+k$ (updating means: if there is no edge from z to y, then draw one with weight $c + k$, otherwise, affect to it the maximum of its weight and $c + k$). We have in effect

$$(z \geq x + c) \wedge (x = y + k) \Rightarrow (z \geq y + (c + k)).$$

In a similar way, an edge from x to z with weight d will lead to an update of edge from y to z with weight $d - k$: at the end of this operation, we have erased the node x by using the equality $x = y+k$, which relates nodes x and y.

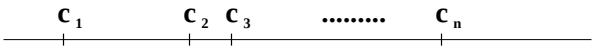
In this way, we use each equality to decrease the size of the graph, by keeping only one of the terms involved in it and transporting all the information about the removed node onto the node that stays.

Let us remember that each modification of the graph during this step is done using the *propagation* relation stated above, in order to keep for each edge the maximum weight between the nodes. We then look for a cycle of strictly positive weight. If we find one, `nodesat` succeeds; if not, we go to the next step and take in consideration the non-equalities.

2.1.7 Getting rid of non-equalities: the actual `nodesat` procedure

What is really considered as the actual `nodesat` procedure is this step on non-equalities, and it is kept as the last step since it can increase a lot the size of the algorithm, in space and in time. The idea is to take in

consideration two nodes, say x and y, and the non-equalities we have between the two corresponding variables. Suppose we have n such inequalities, such as $(x <> y + c_i)_{1 \leq i \leq n}$. This can be represented by the following drawing (we suppose that the $c_{i,\, 1 \leq i \leq n}$ are sorted):



Looking at this picture, it is easy to see that

$$(x <> y + c_1) \wedge (x <> y + c_2) \wedge \cdots \wedge (x <> y + c_n)$$

$$\equiv$$

$$(x - y < c_1) \vee (c_1 < x - y < c_2) \vee \cdots \vee (c_{n_1} < x - y < c_n) \vee (c_n < x - y)$$

In the second line of this formula, we consider the integer $x - y$, which is easy to translate on the graph of inequalities, since an edge of weight a from x to y means $(x - y \geq c)$, and an edge of weight b from y to x means $y \geq x + b$, hence $x - y \leq -b$.

Let us look at the informations we have about x and y:

- the inequalities graph says: $a \leq x - y \leq -b$ (if there is no edge corresponding to a or b, we can replace them by $-\infty$ to keep the same notation)
- the disequalities say:

  $(x - y < c_1) \vee (c_1 < x - y < c_2) \vee \cdots \vee (c_{n_1} < x - y < c_n) \vee (c_n < x - y)$

We can remark that we get from the disequalities a disjunction of inequalities, and since an inequality can be represented on the graph, we shall consider each one of the cases of the disjunction, translate the corresponding inequalities on the graph, and as usual propagate the weights then look for a strictly positive cycle. That is, in the general case, if we are in the ith disjunct, we have inequalities $c_{i-1} < x - y$ and $x - y < c_i$, i.e. $x \geq y + (c_{i-1} + 1)$ and $y \geq x + (1 - c_i)$ (canonical form for inequalities), we can update the edges between nodes x and y with them, and look for a contradiction after propagation of the weights[4].

At this point, if a contradiction is found, it means that the ith disjunct is proven contradictory. But since we are dealing with a disjunction, we have

[4]The "extreme" branches of the disjunction, $x - y < c_1$ and $c_n < x - y$, give only one inequality to update the graph with.

to prove all the branches to be contradictory[5]. If this is the case, we have proven a contradiction on the original system, and the `nodesat` procedure succeeds. If there are branches that still don't contain a contradiction, we have to go on working on them. Before explaining how this job is done, note that through the use of this disjunction we have replaced the initial problem, represented on a graph, by $n+1$ problems (n being the number of disjunctions involving x and y), and hence $n+1$ graphs: the size of the problem has dramatically grown.

The idea to keep on working on the still satisfiable branches of the disjunction is to iterate the `nodesat` algorithm, taking two new nodes instead of x and y, and using the same algorithm as above with the non-equalities concerning these nodes. There are in fact two nested loops: the loop on all pairs of nodes, containing for each pair the loop on all disequalities between the nodes of the pair.

The full `nodesat` procedure explores of course all possible pairs of nodes, and one can see that the problem grows like a tree, each level of the tree corresponding to a pair of nodes (and hence to a conjuction of disequalities, i.e. a disjunction of inequalities, as explained above); from another point of view, all nodes (or leafs) at the same depth in the tree represent two branches of the same disjunction. A leaf is reached if the corresponding situation is proven contradictory, or when all possible pairs of nodes have been explored without finding any contradiction. Let us remind that a subtree is generated by a disjunctive term, and that to prove a disjunction contradictory, one has to prove that all subterms are contradictory as well; the alternative stated above means there are two possibilities when a leaf is reached: either it can't be proved contradictory, thus all the disjunctions above (fathers in terms of trees) aren't provable contradictory, and the procedure fails, or it has been proved contradictory, and we must go on to the next subtree. To reformulate this, if one of the subtrees can't be proven contradictory, then we fail, and hence the problem was not contradictory; in this case, since we have used all the information that we got, the `nodesat` procedure fails: the linearised problem was not provable.

To prevent as much as possible the size of problem to explode, we apply the `nodesat` procedure on each strongly connected component of the graph.

[5] To prove $A \vee B \Rightarrow \bot$, one needs to prove $A \Rightarrow \bot$ and $B \Rightarrow \bot$.

3 The implementation

This work started from a CamlLight implementation of the `nodesat` procedure by Valérie Ménissier-Morain. It was actually an integration of the code into the Coq system, a process that sometimes turned out a bit unelegant or unnatural, mostly because the Coq system, which is not as flexible as the CamlLight language, forced to rewrite part of the code in order to increase the precision and the "atomicity" of some functions. As we will not enter the detail of these modifications here, we just want to stress that the reason for which the whole tactic has not been written for the Coq proof assistant from scratch is basically a lack of time. We shall describe here the main steps of this integration.

3.1 CamlLight types

Here are the important CamlLight types for the `nodesat` objects:

```
type term =
    Const of integer
  | Var of int
  | Add of term * term
  | Sub of term * term
  | Minus of term
  | Mult of term * term
;;
type comp =
  Eq of term * term | Neq of term * term
| Le of term * term | Lt of term * term | Ge of term * term | Gt of term * term
;;
type weight = None | Some of integer
;;
```

The type `comp` represents comparisons between arithmetical expressions implemented as elements of type `term`. A graph is a vector of vectors of elements of type `weight`.

For the proof, the relevant types are:

```
type arith_pf =
  PR of int * string * (term list)              (* Pattern and *)
                                                (* Rewrite *)
| RC of string * (constr list)                  (* Pattern and *)
                                                (* Rewrite with *)
                                                (* constr arguments *)
| RE of int * term * term * (arith_pf t)        (* Pattern and *)
```

```
                                                    (* Replace *)
| RN of bool *int * term * int                      (* Replace a nat term *)
                                                    (* true for >0 *)
| CC of string * term * term * (arith_pf t)         (* Cut on a *)
                                                    (* comparison *)
| CG of constr * (arith_pf t)                       (* Cut on a Coq *)
                                                    (* comparison *)
| AA                                                (* Assumption or Auto *)
                                                    (* *)
| PS of int * term * int                            (* Pattern and Simpl *)
                                                    (* *)
| AP of string * (term list)                        (* Apply a rule *)
| AG of string * (constr list)                      (* Apply with constr *)
                                                    (* arguments *)
| IN                                                (* Intro *)
| NO                                                (* null tactic *)
| DIS of (term list) * (arith_pf t) * (arith_pf t)
                                                    (* Elim for *)
                                                    (* disjunction *)
;;

type var_type =
  ZINT of constr
| INJ of constr;;
```

The type `arith_pf` is used for a proof step, each step corresponding (roughly) to a Coq tactic with its arguments when necessary. The proof generated by *Nodesat* is a queue of elements of type `arith_pf`; once we know that the `nodesat` procedure succeeds, we remove the elements from the queue one after another and translate them in order to build the Coq proof term.

The type `var_type` is used to implement the variables collected during the parsing, the constructor `ZINT` is used for variables of type `Z`, while the constructor `INJ` is used for variables of type `nat` (see below for more details).

Nodesat works with integers, as they are implemented in the files of the subdirectory `Zlemmas` of the tactic. The actual lemmas used by *Nodesat* are in the subdirectory `Rules`; they are in fact mainly *ad hoc* statements of the results proved in the `Zlemmas` files, expressed as the tactic needs them.

3.2 Following the proof steps

Let us sketch the various steps of the algorithm and the kind of proof terms they generate. As the type `pr_type` shown above suggests, the proof steps correspond very closely to Coq tactics (or in some cases *tacticals*): *Node-*

sat could probably be able to give as a result a proof script in a Coq file, instead of generating the proof term and submit it directly to the system.

It is important to notice that the actual proof term coming from a trace of the execution of `nodesat` is built only when the tactic succeeds, otherwise the proof so far is forgotten; this translation from objects of type `arith_pf` into Coq λ-terms is a quite long process, so we can say that if the user has to wait, this means that `nodesat` has succeeded and the proof will terminate, in other words if the user has to wait, he should be full of hope (or of fear of a bug in *Nodesat*!) ...

Parsing *Nodesat* begins with applying the tactic `Intros` to the current subgoal. Then it builds the list of the variables occurring in the conclusion of the goal, and taking them as *relevant variables*, looks in the context for comparisons on integer or natural expressions containing these *relevant variables*, called *relevant comparisons*. For each *relevant comparison*, *Nodesat* updates the list of *relevant variables* with the variables occurring in this comparison, and goes on building the list of *relevant comparisons*. At the end of this process, the tactic has a list of comparisons in the context and a conclusion, and the `nodesat` procedure can begin.

Remark about the `nat` type: An important feature of *Nodesat* is to work with objects of type `nat`, i.e. natural numbers. The idea is to apply to each variable the natural injection `injp` from `nat` to `Z`, adding to the system an inequality expressing the fact that a natural number is always positive. More precisely, when the parser has to deal with a comparison of two expressions of type `nat`, say for example `(le E1 E2)`, it applies the corresponding injection lemma, here

```
Lemma inj_le: (n1,n2:nat) (leZ (injp n1) (injp n2)) -> (le n1 n2).
```

As a result, we now get a comparison on integers (`(injp n1)` and `(injp n2)`); we then "propagate" the `injp` construct inside each expression, that is for example if `n1` is `(plus m1 m2)`, we replace `(injp n1)` with `(addZ (injp m1) (injp m2))`, and so on, until we reach terms like `(injp n)`, with `n` a variable of type nat: at this point, we add the inequality `(geZ (injp n) OZ)`, which is easily proved, to the system, and we store the variable `(injp n)` in the list of parsed integer variables (if it has not been already parsed before). The file `Naturals.v` contains the lemmas used for the treatment of terms of type `nat`.

The current version of *Nodesat* does not treat conjunctive hypotheses and disjunctive conclusions, as the `nodesat` algorithm does: the user has to

split them "manually" in order to use the tactic; however, a pre-treatment of the goal in order to avoid this should'nt be very difficult to implement.

Negating the conclusion To try to prove a contradiction by taking the negation of the conclusion and add it to the comparisons from the context, *Nodesat* has to use a decidability lemma.

That for, the file `Decidable.v` contains the following Coq statements:

```
Definition decidable := [P:Prop] (P\/~P).

Lemma repl_but:  (G: Prop) (decidable G) -> (~G -> False) -> G.
```

Decidability is also proved for each comparand on integer and natural expressions, which allows to use the lemma `repl_but` to replace for example a goal with a conclusion of the form `(leZ a b)` with a goal where the conclusion is `False` and the hypothesis `~(leZ a b)` is added to the context.

Normalising and simplifying The CamlLight function `cring` in the file `terms.ml` is used to normalise the arithmetic expressions; each time a rewriting rule is used by this function, a "Pattern and Rewrite" term is added to the proof, in order to trace exactly the modifications that are brought to the term. The `Pattern` tactic is used to isolate the subterm of the expression to which the rewriting rule is applied, and to preserve other possible occurences of this subterm from being modified. When a comparison is treated by the `nodesat` procedure, it is first brought into the conclusion of the Coq subgoal by using the `Cut` tactic[6], then all the `Rewrite` tactics are applied in order to normalise the terms of the comparison, and the new form of the comparison is reintroduced in the context by `Intro`.

Example Suppose we have the hypothesis `(leZ (neg 0) (addZ x (pos 0)))` (i.e. $-1 \leq x + 1$, `x` is of course of type `Z` here), we have something like:

```
  x : Z
  H : (leZ (neg 0) (addZ x (pos 0)))
  ============================
   False
```

We first bring `H` into the conclusion of the goal:

```
toto < Cut (leZ (neg 0) (addZ x (pos 0))); Try Assumption.
1 subgoal
```

[6] The `Generalize` tactic would be more clever here.

```
x : Z
H : (leZ (neg O) (addZ x (pos O)))
============================
 (leZ (neg O) (addZ x (pos O))) -> False
```

Then the normalisation gives the inequality $-1 \leq 1 + 1.x$ (we skip here several "Pattern and Rewrite" proof steps):

```
x : Z
H : (leZ (neg O) (addZ x (pos O)))
============================
 (leZ (neg O) (addZ (pos O) (multZ (pos O) x))) -> False
```

... and we can introduce the normalised comparison into the context.

The simplifications of the comparisons (for example replacing $x+y \geq y+z$ by $x \geq z$) are done by using the `Cut` tactic on the simplified comparison, and for the generated subgoal by applying the corresponding lemmas (in this example the lemma would look like `Lemma simpl_l_r : (x,y,z:Z) (geZ (addZ x y) (addZ y z)) -> (geZ x z).`). Note that all properties used so far, for the normalisation (mainly `Rewrite` tactic) and for the simplification (mainly `Apply` tactic) involve basically only the properties of the ring of integers (those properties that are proved in the files of the `Zlemmas` subdirectory, actually).

In the implementation of the `nodesat` procedure by Valérie Ménissier-Morain, some refinements are added to the plain algorithm in order to increase its strength. These improvements are mainly concerned with the normalisation of terms and comparisons, looking for sensible simplifications to do before the linearisation in order to reduce the loss of information during this step.

The linearisation step As far as the proof is concerned, the linearisation step does not involve any proof term at all: what *Nodesat* does is only rename internally some terms into fresh variables, and we keep track of this in order to build the Coq proof terms, but no trace of this step is kept in the proof. In other words, if *Nodesat* creates a new "linear" variable T to denote the monom $x.y$, what we need to know is only that ulterior proof terms involving T have to be built with $x.y$ replacing T.

The graph The structure itself of the graph is not actually visible in the Coq proof. What we keep during the `nodesat` procedure is an exact image of what the edges of the graph say in the context: each time an edge is updated, the corresponding inequality is added to the context. In particular, when the propagation rule stated above is used to add a new edge, the tactic *Nodesat* applies a `Cut` on the new resulting comparison, and the lemma corresponding to the propagation rule is applied to the generated subgoal in order to prove it.

The `nodesat` procedure succeeds whenever a strictly positive cycle can be found in the graph; it can then be reduced to an edge from a node to itself with a strictly positive weight: when this occurs, *Nodesat* uses a lemma of the form `(z,c:Z) (gtZ c OZ) -> (geZ z (addZ z c)) -> False` to terminate the proof.

The `nodesat` procedure This part of the algorithm basically uses the tools described above for the manipulation of the graph. The only important lemma that is used here is the disjunction lemma, stating that $x <> y$ implies $(x < y) \vee (x \geq y)$. The Coq tactic used with this lemma is of course `Elim`, and this is done each time a new subtree is generated by "splitting" a disjunction.

3.3 Examples of the use of *Nodesat*

We give here a few examples of how *Nodesat* works on common arithmetical goals; we assume that the reader is familiar with the Coq syntax.

Rewriting

```
1 subgoal

  n : nat
  m : nat
  p : nat
  ============================
   (plus m (plus (S p) n))=(S (plus (plus n m) p))

Unnamed_thm < Nodesat.
Subtree proved!
```

Hiding terms

```
1 subgoal

  f : Z->nat
  ============================
   (z:Z)(le O (f z))

Unnamed_thm < Nodesat.
Subtree proved!
```

Unprovable goals

```
1 subgoal

  ============================
   (le (S O) O)

Unnamed_thm < Nodesat.
Nodesat: that seems false...
1 subgoal

  ============================
   (le (S O) O)

Unnamed_thm <
```

Failure of *Nodesat*

```
1 subgoal

  ============================
   (x:nat)(le O (mult x x))

Unnamed_thm < Nodesat.
I can't prove that (either a linearisation problem or a bug...)
1 subgoal

  ============================
   (x:nat)(le O (mult x x))

Unnamed_thm <
```

This is a typical case where the procedure fails because of the linearisation step, that replaces `(mult x x)` with a new fresh variable.

4 Conclusion and future work

Nodesat is not very efficient in terms of time (when the tactic succeeds, the actual construction of the proof can take a minute long or even more) and of space (printing a proof term generated by *Nodesat* reveals a big amount of irrelevant proof steps, coming mostly from the normalisation process). However, it is a first step towards automation of reasoning about numbers, a feature that definately has to be part of a system designed for applications, as Coq turns to. Actually, as the system becomes more complex, the proofs done in Coq are more and more concerned with "real size" applications, such as hardaware checking [CGJ] or protocol verification [BGLH+95], rather than in various branches of pure mathematics, as it happened firstly with the task of showing the system's expressiveness. Even in this field, a big amount of work has to be done on real numbers implementation (following the ideas of [Hir92], for example) and it turns out that for such proofs (be they about industrial problems or about mathematical analysis), an arithmetical decision procedure such as *Nodesat* is very useful.

Nodesat can be improved in many ways. A first modification is the ability to treat Coq's `pred` and `minus` constructs, that behave in a very particular way because of their types: `pred` is of type `nat -> nat`, and thus `(pred 0)` has to be an absurd value (actually it is `0`), and `minus` is of type `nat -> nat -> nat`, `(minus 0 n)` returning `0` for any `n`. In the examples below, we can see how the tactic succeeds when these constructs can be erased by simplification, but fails otherwise.

```
1 subgoal

  ============================
   (n:nat)(le (pred (S (S 0))) n)->(gt n 0)

Unnamed_thm < Nodesat.
Subtree proved!
```

... but ...

```
1 subgoal

  n : nat
  ============================
   (minus n 0)=n

Unnamed_thm < Nodesat.
Parsing of minus and pred constructs will be available soon...
```

```
1 subgoal

  n : nat
  ============================
   (minus n O)=n
```

The solution to deal with that is to consider, for each term of the type (`pred E`) with `E` a non constant expression, the case where `E` is equal to zero, and the case where `E`> 0. The size of the problem thus grows (for the (`minus E E'`) case, we have to consider two possibilities as well: `E` $\leq$ `E'` and `E` $>$ `E'`), and each proof has to be done "twice". A first version of *Nodesat* with the ability of treating `pred` and `minus` is in preparation, but still has some bugs.

Moreover, if we want to prove arithmetical lemmas automatically in the Coq system, the issue of efficiency is very important: already existent systems suggest that the point is not really to have complex procedures, that can handle a large number of problems, but rather to solve common problems quickly and with proofs as simple as possible. This task was already the main motivation of Shostak's improvements of the Bledsoe method in [Sho77], and is clearly expressed in the design of HOL's *arith* library [Bou92]. This suggests that an arithmetical tactic for the Coq proof assistant should eventually be totally rewritten, remembering that the role of such a tool is to supply quick answers to simple common problems that arise very often in technical proofs.

Another important point about the Coq system is that its design does not really allow the presence of "black boxes", i.e. tools that check a formula without giving a proof term for it (technically, this compels to define an axiom for formulas proved with such a *black box*, an axiom that should be considered in this case as a "commonly admitted result"). This is the chosen solution for the PVS proof system, that is defined in [Rus] as a system with an "impure" interpretation for its tactics (as opposed to HOL), using, to solve decidable problems, decision procedures that do not return any proof term. Of course, the improvement in effectiveness and rapidity is noticeable and gives PVS the status of an effective proof system with a wide range of industrial applications; however, such an option is not possible in Coq, where the intent is rather to consider an arithmetical proof in no way simpler or less informative than any other proof: Coq finds its place among the so-called *fully-expansive* theorem provers, as opposed to *partially-expansive* ones, and a Coq proof has thus always to be given in its full extent. Such a choice becomes quite critical if the applications come from fields where security and/or safety are very important issues, i.e. systems like nuclear powerstation controllers or defense systems, as stressed in [Bou92]. A fully-

expansive approach can also lead to implement for example exact arithmetic, as it is done for ML in [MMW95].

References

[BGLH+95] D. Bolignano, J. Goubault-Larrecq, G. Huet, D. Le Métayer, P. Lescanne, and S. Philipakis. The VIP Project - Verified Internet Protocols. description available at `http://pauillac.inria.fr/~huet/vip.html`, Décembre 1995.

[Ble75] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *Advance Papers 4th Joint Conference on Artificial Intelligence*. Tbilisi, Georgia, U.S.S.R., September 1975.

[Bou92] R.J. Boulton. The HOL arith Library. Technical report, University of Cambridge, Computer Laboratory, Cambridge, England, July 1992.

[CCF+96] C. Cornes, J. Courant, JC. Filliâtre, E. Gimenez, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual*. Projet Coq, INRIA Rocquencourt / CNRS - ENS Lyon, 1996.

[CGJ] S. Coupet-Grimal and L. Jakubiec. Vérification formelle de circuits avec Coq. LIM URA CNS 1787 - Université de Provence - Marseille.

[Cha77] T. Chan. *An algorithm for checking PL/CV arithmetic inferences*. TR77-326, Cornell University, Ithaca, New York, 1977.

[Fil95] J.C. Filliâtre. A decision procedure for direct predicate calculus. Technical report, LIP-ENS-Lyon, 1995.

[Gor] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer edition, 1987.

[Hir92] D. Hirschkoff. Construction de l'ensemble des réels en Coq. Rapport de stage scientifique E.N.P.C., 1992.

[Jac94] P.B. Jackson. *The Nuprl Proof Development System, Version 4.1 Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 1994.

[LP92] Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual*. Department of Computer Science, University of Edinburgh, May 1992.

[MMW95] Valérie Ménissier-Morain and Pierre Weis. An exact arithmetic package for ML. *Science for Computer Programming*, 1995. To appear.

[Muñ94] C. Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.

[Rus] J. Rushby. Design choices in PVS. Computer Science Laboratory - SRI International - Menlo Park CA USA.

[Sho77] R.E. Shostak. On the sup-inf method for proving Presburger formulae. *Journal of the ACM*, 24(4):529–543, October 1977.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker Reference Manual*. Computer Science Laboratory - SRI International, Menlo Park CA 94025, march 1993.