HAL
archives-ouvertes.fr

Proof reconstruction (preliminary version).

Judicael Courant

▶ To cite this version:

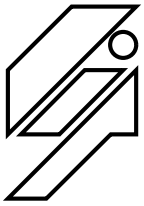
Judicael Courant. Proof reconstruction (preliminary version).. [Research Report] LIP 1996-26, Laboratoire de l'informatique du parallélisme. 1996, 2+14p. hal-02101787

HAL Id: hal-02101787
https://hal-lara.archives-ouvertes.fr/hal-02101787

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

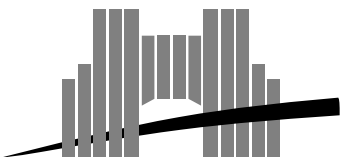
Proof reconstruction

Preliminary version

Judicaël Courant

September 96

Research Report N^{o} 96-26



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Proof reconstruction

Preliminary version

Judicaël Courant

September 96

Abstract

In the field of formal methods, rewriting techniques and provers by consistency in particular appear as powerful tools for automating deduction. However, these provers suffer limitations as they only give a (non-readable) trace of their progress and a yes/no answer where the user would expect a detailed explicit proof. Therefore, we propose a general mechanism to build an explicit proof from the running of a generic class of inductionless induction provers. We then show how it applies to Bouhoula's SPIKE prover, and give examples of proofs built by this method.

Keywords: Rewriting, theorem prover, natural deduction, inductionless induction, SPIKE, Coq

Résumé

Dans le domaine des méthodes formelles, les techniques de réécriture et les prouveurs par récurrence implicite sont des outils puissants pour automatiser le processus de preuve. Cependant, ces prouveurs donnent une trace du déroulement de la preuve peu compréhensible, alors que l'utilisateur lambda aimerait avoir une preuve explicite détaillée. Nous proposons un mécanisme général permettant de construire une preuve explicite à partir de la trace du déroulement d'une preuve pour une classe générique de démonstrateurs par récurrence implicite. Nous montrons comment ce procédé s'applique au prouveur SPIKE de Bouhoula, et donnons des exemples de preuves construites par cette méthode.

Mots-clés: Réécriture, démonstrateur, déduction naturelle, récurrence implicite, SPIKE, Coq

Proof reconstruction*

Preliminary version

Judicaël Courant

September 96

The growing need for formal methods in industry stresses the necessity for efficient tools for specifying and verifiying software. Beyond the choice of the logical framework, designing such a tool raises (at least) the two following issues:

- how can one be sure that the answer of a proof assistant is indeed correct?
- how can one have one's proofs made in an automatic way?

To address the first one, some systems such as Alf [17], Coq [6, 10] or Lego [16] require the proof of a proposition to be a formal object, built by the user and whose manual or mechanical check-up is quite simple[1]. But these systems suffer from the lack of proof automatisms, partly due to the complexity of their underlying logical framework (higher-order).

In order to address the second issue, some others systems like NQTHM [4], RRL [19], SPIKE [3] deliberately choose simpler frameworks, arguing that in many cases, equational or conditional equational reasoning plus induction is enough for solving usual problems. They use the power of rewriting techniques to achieve some complex proofs with few interaction with the user [5, 13, 11]. A significant step for automatising induction, the *inductionless induction* technique, was taken in the early eighties [9]: its principle is to simulate induction by term rewriting. The scope of this technique has considerably widened since then [12, 8, 1, 14, 18, 15, 3].

Unfortunately, this way to solve the second issue is quite far from addressing the first one, as with inductionless induction technique, the correction of the proof relies on the whole correction of the prover. As no proof structure clearly arises, no further verification is possible. On the contrary, in a prover requiring proof objects, an automatic tactic builds a proof that is later verified by the prover: even if the tactic code is buggy, no false conjecture can be proved provided that the small amount of code verifying a proof object is correct.

Our aim is to show a way to reconcile the two approaches: we want to have an inductionless induction prover run, and once it stops, then get the trace of its progress to be automatically transformed into an explicit proof in a first-order natural deduction formalism with recursion schemes. To achieve this goal, we shall first recall some useful definitions, and informally what is Bouhoula's notion of generic inference procedure [2] and give the first-order logic extension with recursion schemes we want to work with in section 1. Then, in section 2 we shall introduce our notion of K-system, which is a particular class of inference system for automatic proving. We shall prove that a conjecture justified by a run of this system is provable in our logical framework in a constructive manner, *i.e.* we shall give an algorithm

*This research was partially supported by the ESPRIT Basic Research Action Types and by the GDR Programmation cofinanced by MRE-PRC and CNRS

[1] In these systems, proof objects are terms, and the program checking a proof is only a type-checker ; of course, to keep a proof object is also interesting in these frameworks with respect to the "proofs as programs" paradigm.

that builds an explicit proof. Then in section 3, we shall see how the previous work applies to the prover SPIKE [3].

1 Terminology and framework

1.1 Terminology and notations

Let $(S, F, \mathcal{C})$ be a many sorted signature where S is a finite set of sorts, F is a finite set of function symbols and $\mathcal{C}$ is the subset of F whose elements are constructor symbols. Let X be a family of sorted variables, and $T(F, X)$ be the set of well-sorted terms and $T(F)$ the set of ground terms (*i.e.* without any variable).

Definition 1 (Clause, conditional equation) *A* clause *is a formula which states that a conjunction of some (possibly zero) atomic equalities between terms implies a disjunction of some (possibly zero) atomic propositions i.e. which is of the form* $(A_1 \wedge \ldots \wedge A_n) \Rightarrow (B_1 \vee \ldots \vee B_m)$. *If* $m = 1$, *this clause is said to be a* conditional equation.

Let $Clauses$ be the (infinite) set of all clauses.

Definition 2 (Specification) *A* specification *is a finite set* L *of clauses:* L *is a specification* $\iff L \subset Clauses$.

In the following, let us assume that L is a specification and that $\prec$ is a transitive irreflexive relation on terms that is noetherian, monotonic ($t \succ t'$ implies $w[x \leftarrow t] \succ w[x \leftarrow t']$ provided x appears in w), and satisfies the proper subterm property (whenever t' is a proper subterm of t, $t \succ t'$). Furthermore, let $\prec_c$ be a well-founded ordering on clauses which is an extension of $\prec$, in the sense that if C is a clause such that x occurs in C, and t and t' are two terms such that $t \prec t'$, then $C[x \leftarrow t] \prec_c C[x \leftarrow t']$ (see for instance [2]) where $[x \leftarrow t]$ denotes the substitution associating t to x.

Definition 3 (Inductive theorem) *A clause* C *is an inductive theorem of the specification* L *if it is true in the initial model of* L. *We shall denote this by* $L \models_{ind} C$.

Definition 4 (Constructor substitutions) *A* constructor substitution, *is a substitution whose image are terms built upon variables and constructors only (i.e. terms belonging to* $T(\mathcal{C}, X)$*).*

Definition 5 (Cover set) *A cover set* CV *[19, 18] for a specification* L *(more precisely : for a signature* $(S, F, \mathcal{C})$ *is a set of terms containing constructors as only function symbols, such that for every term* u *in* $T(\mathcal{C}, X)$, *there exists* t *in* CV *and a substitution* σ *such that* $t\sigma \equiv u$. *A substitution* τ *whose image is a subset of* CV *is called a* CV *substitution, and if* v *is a term,* $v\tau$ *is called a* CV *instance of* v *(notice that a* CV *substitution is necessarily a constructor substitution).*

In the following, let us assume that CV is a cover set for L.

In the following, in order to prove that some conjectures are inductive theorems of the specification L, we shall consider an inference system D whose data structure is a pair (E, H), where E and H are two finite clause sets, *i.e.* a relation $\vdash_D$ between couples of finite clauses sets. The elements of E will be clauses to be proved and the elements of H will play the role of induction hypothesis. When a run starts, H is empty, and at each step, a conjecture of E is rewritten and is possibly added to H as a new induction hypothesis.

Definition 6 (Correction of an inference system) *D is said* correct *if and only if for every specification L, every set of clauses E_0, whenever we get the run $(E_0, \emptyset) \vdash_D (E_1, H_1) \ldots \vdash_D (E_{m-1}, H_{m-1}) \vdash_D (\emptyset, H_m)$, we have $L \models_{ind} E_0$.*

An important class of correct inference systems are I-systems, defined by Bouhoula in his PhD thesis [2, section 4.3]. Informally, this notion relies on the idea that there are essentially three types of operations one can apply in an inference system:

generate is the operation of taking one clause C in E, putting it in H, simplifying its CV instances using clauses in H and E, then putting these simplified instances in E;

simplify is the operation of replacing one clause C in E by simpler ones which are equivalent if smaller instances of other clauses in $E \cup H$ are verified;

delete consists of removing one clause from E provided it is a consequence of smaller instances of clauses in E or H.

1.2 Logical framework

Unfortunately, the notion of I-system relies on purely semantical basis. So, as we want to build explicit proof of conjectures, we shall define a different class of correct inference systems expressing the same ideas as I-systems, but on a more syntactical basis.

For this purpose, we must first describe the logical system within we shall give an explicit proof. The logical framework we want to use is a first order natural deduction system extended with recursion schemes.

We shall consider judgements of type $\Gamma \vdash_{rec} \mathcal{F}$, where $\mathcal{F}$ is a first-order formula, and Γ is a list of first-order formulas. Let F be the set of function symbols. In I-systems an order on clauses is needed to establish their correction. In our system we also require that for each $(C_1, C_2) \in Clauses^2$, there exist two k-ary predicates $\sqsubset_{C_1,C_2}$ and $\sqsubseteq_{C_1,C_2}$, where k is the sum of the number of free variables in C_1 and the number of free variables in C_2, and two binary predicates $<_s$ and $\leq_s$ for each $s \in S$ representing well-founded orders on clauses. We let the set of predicates $\mathcal{P}$ be the set of these predicates plus the equality predicates.

We will denote $\sqsubset_{C_1,C_2} \vec{x}\vec{y}$, $\sqsubseteq_{C_1,C_2} \vec{x}\vec{y}$, $<_s xy$ and $\leq_s xy$ respectively by $\vec{x} \sqsubset_{C_1,C_2} \vec{y}$, $\vec{x} \sqsubseteq_{C_1,C_2} \vec{y}$, $x <_s y$, and $x \leq_s y$, where bold-font variables $\vec{x}$ and $\vec{y}$ denote terms vectors, and the length of $\vec{x}$ and $\vec{y}$ are respectively the numbers of free variables of C_1 and C_2.

We assume the usual rules of first-order natural deduction with equality, plus a set of rules Ax giving properties about the predicates $\sqsubset_{C_1,C_2}$, $\sqsubseteq_{C_1,C_2}$, $<_s$ and $\leq_s$. For instance, these rules can make these predicates represent a polynomial ordering (see Appendix A), or lexicographic path ordering... Indeed, in order to ensure that the chosen relations represent ordering relations, we only require the following rules to be admissible in our system:

$$\frac{\Gamma \vdash_{rec} \vec{x} \sqsubset_{C_1,C_2} \vec{y}}{\Gamma \vdash_{rec} \vec{x} \sqsubseteq_{C_1,C_2} \vec{y}} \quad \Gamma \vdash_{rec} \vec{x} \sqsubseteq_{C,C} \vec{x}$$

$$\frac{\Gamma \vdash_{rec} \vec{x} \sqsubset_{C_1,C_2} \vec{y} \quad \Gamma \vdash_{rec} \vec{y} \sqsubseteq_{C_2,C_3} \vec{z}}{\Gamma \vdash_{rec} \vec{x} \sqsubset_{C_1,C_3} \vec{z}}$$

$$\frac{\Gamma \vdash_{rec} \vec{x} \sqsubseteq_{C_1,C_2} \vec{y} \quad \Gamma \vdash_{rec} \vec{y} \sqsubset_{C_2,C_3} \vec{z}}{\Gamma \vdash_{rec} \vec{x} \sqsubset_{C_1,C_3} \vec{z}}$$

$$\frac{\Gamma \vdash_{rec} \vec{x} \sqsubseteq_{C_1,C_2} \vec{y} \quad \Gamma \vdash_{rec} \vec{y} \sqsubseteq_{C_2,C_3} \vec{z}}{\Gamma \vdash_{rec} \vec{x} \sqsubseteq_{C_1,C_3} \vec{z}}$$

generate: $(E \cup \{C\}, H) \vdash_D (E \cup \{C_1, \ldots, C_n\}, H \cup \{C\})$ and

$$\{\forall \vec{x}(\vec{x} \sqsubset_{C',C} \vec{y} \Rightarrow C'(\vec{x})) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \vdash_{rec} C(\vec{y})$$

simplify: $(E \cup \{C\}, H) \vdash_D (E \cup \{C_1, \ldots, C_n\}, H)$ and

$$\{\forall \vec{x}(\vec{x} \sqsubseteq_{C',C} \vec{y} \Rightarrow C'(\vec{x})) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \vdash_{rec} C(\vec{y})$$

delete: $(E \cup \{C\}, H) \vdash_D (E, H)$ and

$$\{\forall \vec{x}(\vec{x} \sqsubseteq_{C',C} \vec{y} \Rightarrow C'(\vec{x})) \mid C' \in E \cup H\} \vdash_{rec} C(\vec{y})$$

where $\vec{y}$ denotes new variables.

Figure 1: Conditions for D to be a K-system

However, we want this relation on clauses to be well-founded. Therefore we also require the following rule to be admissible:

$$\frac{\Gamma, \vec{y} \sqsubseteq_{C,C} \vec{t}, \forall \vec{z}(\vec{z} \sqsubset_{C,C} \vec{y} \Rightarrow C(\vec{z})) \vdash_{rec} C(\vec{y})}{\Gamma \vdash_{rec} C(\vec{t})} C \in Clauses \text{ and } y \text{ not free in } \Gamma$$

We also add rules allowing to prove something about x by inspection of the several possible cases of its constructors in Ax:

$$\vdash_{rec} \forall_s x(\exists \vec{t_1}\ x = C_1(\vec{t_1})) \vee \ldots \vee (\exists \vec{t_k}\ x =_s C_k(\vec{t_k})) \tag{1}$$

where $C_1, \ldots, C_k$ are the constructors of the sort s.

Finally, as we want to work on the given specification L, we also add the axiom $\vdash_{rec} \forall \vec{x}\ C(\vec{x})$ for every C in the specification L, and we require that every clause C such that $\vdash_{rec} C$ is provable is an inductive theorem of L.

2 K-systems

2.1 Definition

As we have explained our logical framework, we can now express formally what class of inference system we shall consider.

We say that an inference system D is a *K-system* if and only if, whenever $(E, H) \vdash_D (E', H')$, we are in one of the three cases given in Figure 1. This notion is distinct of Bouhoula's notion of I-system, since our conditions are not semantical ones. Therefore, even if the two notions express similar ideas, they are different.

The point is to know whether every K-system is correct or not.

2.2 Correction

Assume D is a K-system, and $(E_0, \emptyset) \vdash_D \ldots \vdash_D (\emptyset, H_m)$. We shall prove in this section that for every clause C in E_0 we have $\vdash_{rec} C$. Moreover, the proof of this fact is constructive: we propose an algorithm giving the skeleton of a proof of $\vdash_{rec} C$ from the run $(E_0, \emptyset) \vdash_D \ldots \vdash_D (\emptyset, H_m)$. We mean by "skeleton" of a proof of the judgement $\Delta \vdash_{rec} P$ a tree such that the root of the tree is the judgement $\Delta \vdash_{rec} P$ and such that each node is labelled by a judgement $\Phi \vdash_{rec} Q$ which can be deduced

in our framework from the proofs of the judgements $J_1, \ldots, J_k$ labelling its sons, *i.e.* the following rule is admissible in $\vdash_{rec}$:

$$\frac{J_1 \ \ldots \ J_k}{\Phi \vdash_{rec} Q}$$

The idea behind this algorithm is the following: if a clause C is replaced by $C_1, \ldots, C_n$ in E, this means we can build a skeleton of proof of C by building skeletons for $C_1, \ldots, C_n$, and then making these skeletons be the sons of a root labelled by $\vdash_{rec} C$. But this only works in very simple cases, since if C is (indirectly) used to prove itself then this algorithm loops forever. So, in this case, we have to apply an induction scheme, and make sure that the well-foundness properties of the inference system guarantee that we can use the induction hypothesis instead of looping. Therefore, we keep the induction hypothesis in the judgements $\Gamma \vdash_{rec} C$, plus those stating inequalities between terms, allowing us to conclude that the induction hypothesis can be used.

Let us now describe more precisely the algorithm:

Algorithm 1 (Proof reconstruction)
We assume we have as an input the run $(E_0, \emptyset) \vdash_D \ldots \vdash_D (\emptyset, H_m)$ *of* D *on* $(E_0, \emptyset)$. *Notice we have* $\bigcup_{i=0}^{n} H_i \subset \bigcup_{i=0}^{n} E_i$. *Proof reconstruction is done by calling the function* recurse_clauses, *defined as follows:*

Function recurse_clauses
Input:

- C, *a clause belonging to* $\bigcup_{i=0}^{n} E_i$;
- G, *a (finite) set of clauses (corresponding to induction hypothesis:* $G \subset \bigcup_{i=0}^{n} H_i$*)* ;
- J, *a judgement* $\Gamma \vdash_{rec} D$ *(which is what we want to prove ;* D *is a renaming of* C*).*

Output:
A tree labelled with judgements.
Method:

- *if* C *is in* G, *then return the tree whose root is labelled with* J *and having only one son, labelled with* $\Gamma \vdash_{rec} \vec{x} \sqsubseteq_{C,C} \vec{y}$ *where* $\vec{x}$ *and* $\vec{y}$ *are the variables such that* $D = C(\vec{x})$ *and* $\forall \vec{z}(\vec{z} \sqsubset_{C,C} \vec{y} \Rightarrow C(\vec{z}))$ *belongs to* Γ.
- *else, if* C *disappears from* E *between* i *and* $i+1$ *by the use of the* ***generate*** *rule (i.e.* $C \in E_i$, *but* $C \notin E_{i+1}$*), and* $D = C(\vec{x})$ *then take some new variables* $\vec{u}$, $\vec{y}$ *and* $\vec{z}$, *let* $\Gamma' = \Gamma, \vec{y} \sqsubseteq_{C,C} \vec{x}, \forall \vec{z}(\vec{z} \sqsubset_{C,C} \vec{y} \Rightarrow C(\vec{z}))$ *and call recursively* recurse_clauses *on* $(C', G \cup \{C\}, (\Gamma', \vec{u} \sqsubset_{C',C} \vec{y} \vdash_{rec} C'(\vec{u})))$, *for* $C' \in E_i \cup H_i \cup \{C\} \cup \{C_1, \ldots C_n\}$, *let* $T_1, \ldots T_k$ *be the resulting trees. Then return the tree whose root is labelled by* $\Gamma \vdash_{rec} C(\vec{x})$ *and has one only son labelled by* $\Gamma' \vdash_{rec} C(\vec{y})$ *itself having* $T_1, \ldots, T_k$ *as sons ;*
- *if not, then, with* $D = C(\vec{x})$, *call recursively* recurse_clauses *on* $(C', G, (\Gamma, \vec{y} \sqsubseteq_{C',C} \vec{x} \vdash_{rec} C'(\vec{y})))$ *for* $C' \in H_i \cup E'$, *where* E' *is* $E_i \cup \{C_1, \ldots, C_n\}$ *if* C *disappears with the simplification rule and* E_i *otherwise, and* $\vec{y}$ *is a vector of new variables.*

Initialisation:
call recurse_clauses *on* $(C, \emptyset, \vdash_{rec} C(\vec{x}))$.

sorts :
$nat, list, bool$
constructors :
$0 : nat$
$s : nat \rightarrow nat$
$Nil : list$
$Cons : nat \times list \rightarrow list$
$True : bool$
$False : bool$
functions
$length : list \rightarrow nat$
$insert : nat \times list \rightarrow list$
$<=: nat \times nat \rightarrow bool$
axioms
$0 <= x = True$
$s(x) <= 0 = False$
$s(x) <= s(y) = x <= y$
$length(Nil) = 0$
$length(Cons(x, y)) = s(length(y))$
$insert(x, Nil) = Cons(x, Nil)$
$x <= y = True => insert(x, Cons(y, z)) = Cons(x, Cons(y, z))$
$x <= y = False => insert(x, Cons(y, z)) = Cons(y, insert(x, z))$
$x <= x = True$
$x <= y = True \vee x <= y = False$
$x <= y = True \vee y <= x = True$
$x <= y = False \vee y <= z = False \vee x <= z = True$

Figure 2: A specification example

Theorem 1 *If this algorithm terminates, it gives a skeleton of a proof of* $\vdash_{rec} C$.

Proof 1 *See Appendix B.*

Theorem 2 *This algorithm always terminates.*

Proof 2 *See Appendix B.*

As a consequence, every K-system is a correct system, and we can find a proof in $\vdash_{rec}$ of each conjecture justified by a K-system.

2.3 Example

Let us consider the specification given in Figure 2 which is part of the specification of an insertion sort working on integer lists. We want to study the conjecture $length(insert(x, y)) = s(length(y))$.

Here is how a mathematician would prove this conjecture:

By induction on x_2, it is enough to prove $length(insert(x_1, x_2)) = s(length(x_2))$ under the hypothesis $\forall x'_1 \forall x'_2\ x'_2 <_{list} x_2 \Rightarrow length(insert(x'_1, x'_2)) = s(length(x'_2))$ where $<_{list}$ is the well-founded ordering on lists defined by $\forall x, y\ y <_{list} Cons(x, y)$ plus the transitivity rule.

Then, either $x_2 = Nil$ or $x_2 = Cons(x_3, x_4)$:

- in the first case, our conjecture is verified since $length(insert(x_1, Nil)) = length(Cons(x1, Nil)) = s(length(Nil))$;

- in the second case, as we have $x<=y = True \vee x<=y = False$ in L, we can consider the two following cases:
  - if $x_1<=x_3 = True$ then $insert(x_1, Cons(x_3, x_4)) = Cons(x_1, (Cons(x_3, x_4)))$ (by L) and $length(insert(x_1, Cons(x_3, x_4))) = s(length(Cons(x_3, x_4)))$
  - if $x_1<=x_3 = False$, then $insert(x_1, Cons(x_3, x_4)) = Cons(x_3, insert(x_1, x_4))$ (by L) and $length(insert(x_1, Cons(x_3, x_4))) = s(length(insert(x_1, x_4)))$, and $s(length(Cons(x_3, x_4))) = s(s(length(x_4)))$. But since $x_4 <_{list} x_2$, by induction hypothesis, $length(insert(x_1, x_4)) = s(length(x_4))$, and the conjecture is also true if $x_1<=x_3 = False$.□

We shall compare later this "natural" proof to the one we shall compute with our algorithm from the run of a K-system.

However, in order to have a K-system run on this example, we first need to choose which rules we put in Ax. An acceptable choice is to take a polynomial ordering, as explained in Appendix A. Here is now the possible run of such a K-system:

$$\left\{ \begin{array}{lcll} E_0 & = & \{ & length(insert(x1, x2)) = s(length(x2)) \quad \} \\ H_0 & = & & \emptyset \end{array} \right.$$

By the **generate** rule (the following is correct since $x1 <= x3 = False \vee x1 <= x3 = True \in L$):

- put $length(insert(x1, x2)) = s(length(x2))$ in H_1;
- put $s(0) = s(0)$ in E_1, which is equivalent to $length(insert(x1, x2)) = s(length(x2))[x2 \leftarrow Nil]$ in the initial model of L;
- put $x1 <= x3 = True => s(s(length(x4))) = s(s(length(x4)))$ which is equivalent to $length(insert(x1, x2)) = s(length(x2))[x2 \leftarrow Cons(x3, x4)]$ for every instance of $x1$, $x3$ such that $x1 <= x3 = True$ in E_1;
- put $x1 <= x3 = False => s(length(insert(x1, x4))) = s(s(length(x4)))$ whose any instance such that $x1 <= x3 = False$ is equivalent to the same instance of $length(insert(x1, x2)) = s(length(x2))[x2 \leftarrow Cons(x3, x4)]$ in E_1;

$$\left\{ \begin{array}{lcll} E_1 & = & \{ & s(0) = s(0), \\ & & & x1 <= x3 = True => s(length(x4)) = s(length(x4)), \\ & & & x1 <= x3 = False => s(length(insert(x1, x4))) = s(s(length(x4))) \quad \} \\ H_1 & = & \{ & length(insert(x1, x2)) = s(length(x2)) \quad \} \end{array} \right.$$

By the **delete** rule, remove the first conjecture, which is trivially true:

$$\left\{ \begin{array}{lcll} E_2 & = & \{ & x1 <= x3 = True => s(length(x4)) = s(length(x4)), \\ & & & x1 <= x3 = False => s(length(insert(x1, x4))) = s(s(length(x4))) \quad \} \\ H_2 & = & \{ & length(insert(x1, x2)) = s(length(x2)) \quad \} \end{array} \right.$$

By the **delete** rule also remove the first of the remaining conjectures:

$$\left\{ \begin{array}{lcll} E_3 & = & \{ & x1 <= x3 = False => s(length(insert(x1, x4))) = s(s(length(x4))) \quad \} \\ H_3 & = & \{ & length(insert(x1, x2)) = s(length(x2)) \quad \} \end{array} \right.$$

By **simplify** the remaining conjecture becomes:

$$\left\{ \begin{array}{lcll} E_4 & = & \{ & x1 <= x3 = False => length(insert(x1, x4)) = s(length(x4)) \quad \} \\ H_4 & = & \{ & length(insert(x1, x2)) = s(length(x2)) \quad \} \end{array} \right.$$

By **delete**, since the conjecture in E_4 is subsumed by the clause of H_4:

$$\begin{cases} E_5 = \emptyset \\ H_5 = \{ \; length(insert(x1, x2)) = s(length(x2)) \; \} \end{cases}$$

Then E_5 is empty, which means that the conjecture in E_0 is an inductive theorem of L. $\square$

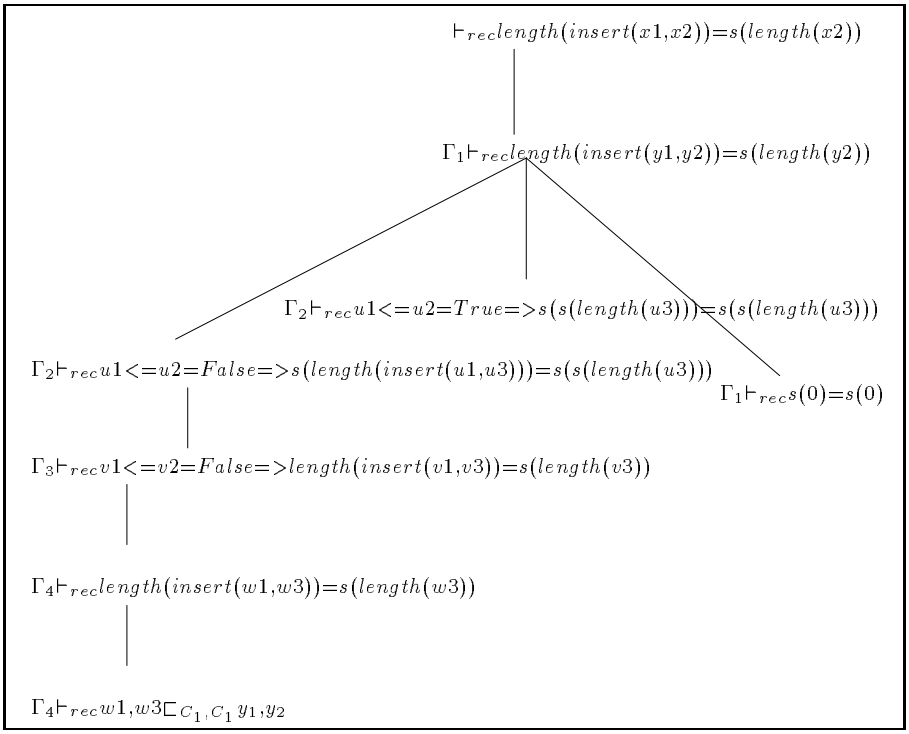


Figure 3: Proof skeleton built by the algorithm

It is clear that this run is not immediately human-readable since no proof structure arises. Let us now see what gives our algorithm: let us call recurse_clauses on $(length(insert(x1, x2)) = s(length(x2)), \emptyset, \vdash_{rec} length(insert(x1, x2)) = s(length(x2)))$. We get the proof skeleton given in Figure 3, where

$$\begin{aligned} \Gamma_1 &= y1, y2 \sqsubseteq_{C_1, C_1} x1, x2, \; \forall z1, z2 (z1, z2 \sqsubset_{C_1, C_1} y1, y2 \Rightarrow C_1(z1, z2)) \\ \Gamma_2 &= \Gamma_1, \; u1, u2, u3 \sqsubset_{C_2, C_1} y1, y2 \\ \Gamma_3 &= \Gamma_2 \; v1, v2, v3 \sqsubseteq_{C_4, C_3} u1, u2, u3 \\ \Gamma_4 &= \Gamma_3, \; w1, w3 \sqsubseteq_{C_1, C_4} v1, v2, v3 \end{aligned}$$

and

$$\begin{aligned} C_1(x1, x2) &= length(insert(x1, x2)) = s(length(x2)) \\ C_2(x1, x2, x3) &= x1 <= x2 = True => s(s(length(x3))) = s(s(length(x3))) \\ C_3(x1, x2, x3) &= x1 <= x2 = False => s(length(insert(x1, x3))) = s(s(length(x3))) \\ C_4(x1, x2, x3) &= x1 <= x2 = False => length(insert(x1, x3)) = s(length(x3)) \end{aligned}$$

We can notice that the structure of the computed skeleton is very similar to the proof we gave above, though a little more complicated. This suggests that some optimizations could be added to our algorithm, in order to introduce new variables only when it is necessary.

3 Application to SPIKE

Our notion of K-system is based on relations between first-order formulas. But in usual inference systems, only clauses are manipulated, relations defining the inference system only involve clauses and a *meta-level* ordering. Therefore, it seems interesting to define another class of inference system involving only such relations. So, we say that an inference system D is an *M-system* if and only if, whenever $(E, H) \vdash_D (E', H')$, we are in one of the three cases given in Figure 4, where $\vdash$ denotes deduction in first-order logic.

generate $(E \cup \{C\}, H) \vdash_D (E \cup \{C_1, \ldots, C_n\}, H \cup \{C\})$ and there exists a partition $\{K_\sigma \mid \sigma\ CV$ substitution$\}$, of $\{1, \ldots m\}$ such that for every CV substitution σ $\{\forall \vec{x} C'(\vec{x}) \mid C' \in L\} \cup \{t\theta \mid t \in E \cup H \cup \{C\}\ \theta$ constructor substitution and $t\theta \prec_c C\sigma\} \vdash (C\sigma) \iff (\wedge_{k \in K_\sigma} C_k)$ and $\forall k \in K_\sigma\ C_k \prec_c C\sigma$;

simplify $(E \cup \{C\}, H) \vdash_D (E \cup \{C_1, \ldots, C_n\}, H)$ and $\{\forall \vec{x} C'(\vec{x}) \mid C' \in L\} \cup \{t\theta \mid t \in E \cup H\ \theta$ constructor substitution and $t\theta \preceq_c C\} \vdash C \iff (C_1 \wedge \ldots \wedge C_n)$ and $\forall k \in \{1, \ldots, n\}\ C_k \prec_c C$

delete $(E \cup \{C\}, H) \vdash_D (E, H)$ and $\{\forall \vec{x} C'(\vec{x}) \mid C' \in L\} \cup \{t\theta \mid t \in E \cup H\ \theta$ constructor substitution and $t\theta \preceq_c C\} \vdash C$

Figure 4: Conditions for D to be an M-system

Theorem 3 *Assume D is an M-system, and that for each pair of clauses (C_1, C_2) and for every constructor substitutions σ and θ, whenever $C_1\theta \prec_c C_2\sigma$, we have $\vdash_{rec} \vec{x}\theta \sqsubset_{C_1,C_2} \vec{y}\sigma$ and whenever $C_1\theta \preceq_c C_2\sigma$, we have $\vdash_{rec} \vec{x}\theta \sqsubseteq_{C_1,C_2} \vec{y}\sigma$, where $\vec{x}$ and $\vec{y}$ are the variables of C_1 and C_2. Then, D is a K-system.*

Proof 3 *See Appendix B.*

SPIKE is a theorem prover whose inference system D_S is an I-system. We shall not describe it here precisely: the interested reader may refer to [3, 2] for a complete survey.

Unfortunately, D_S is not a K-system nor an M-system. But we may restrict the application of rules in order to get an M-system D'_S: we only have to restrict rules to apply only when they meet the requirements to be an M-system. The only thing we have to do is to forbid this inference system to use non-constructor instances of clauses in order to apply its **simplify**, **delete** or **generate** rules. D'_S is weaker than D_S since every conjecture that can be proved by D'_S can also be proved by D_S, but practically, most of the time, D_S and D'_S behaves the same way, so that in order to get an explicit proof of a conjecture, you can make SPIKE run without even modifying it, then apply algorithm 1. On Figures 5, 6 are some examples (from Bouhoula's thesis [2]) that SPIKE solves and that our algorithm translates whithout any difficulty. Notice that the first one uses mutually recursive functions, hence – according to Bouhoula – it is not automatically solvable by NQTHM.

sorts:	**axioms:**
$bool, action, state$	$or(True, x) = True$
constructors:	$or(x, True) = True$
$True :\rightarrow bool$	$or(False, False) = False$
$False :\rightarrow bool$	$F_1(do(a, s)) = or(F_2(s), F_3(s))$
$Nop :\rightarrow action$	$F_2(do(a, s)) = F_3(s)$
$S_0 :\rightarrow state$	$F_3(do(a, s)) = F_2(s)$
$do : action \times state \rightarrow state$	$F_1(S_0) = True$
functions:	$F_3(S_0) = True$
$or : bool \times bool \rightarrow$	$F_2(S_0) = False$
$F_1 : state \rightarrow bool$	**conjecture**
$F_2 : state \rightarrow bool$	$F_1(x) = True$
$F_3 : state \rightarrow bool$	

Figure 5: Situation invariant problem

4 Conclusion and future work

Properties required for inference systems to be I-systems are only semantical. We proposed a different class of inference systems, keeping the same ideas as I-systems, but expressing them in a first-order logical framework. This allowed us to present a method for building an explicit proof of a theorem from a trace of the progress of a prover by consistency. As far as we know, this approach is original. Since the inference systems of provers by consitency are generally only based on clauses, we defined a better-suited class of inference systems (M-systems), which are in fact particular cases of K-systems (under a few restrictions). This approach is powerful enough to succeed on several examples with such a modern prover as SPIKE. We think it to be a promising way to make provers by consistency safely cooperate with other provers, as even if bugs remain in the design or implementation of the interface between them or the prover by consistency, a wrong proof is rejected by the back-end prover.

A toy implementation of this method (allowing only boolean specification about naturals) to interface SPIKE with Coq has been done [7], but it has to be completed in order to work with realistic examples (a good challenging one could be the Gilbreath's card trick). Part of this implementation could also be used to add to SPIKE interface the capability to explain the proofs it makes. However, further work could be necessary to extend enough the class of system this method handles. In this respect, the study of interaction between the order and positions where induction is done could be fruitful.

References

[1] L. Bachmair. Proof by consistency in equational theories. In *Proceedings of the third IEEE Symposium on Logic in Computer Science*, pages 228–233, 1988.

[2] A. Bouhoula. *Preuves automatiques par récurrence dans les théories conditionnelles.* Thèse, Université de Nancy I, march 1994.

[3] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Spike: an automatic theorem prover. Technical Report 1636, INRIA, 1992.

sorts:
$nat, list$
constructors:
$0 : nat$
$s : nat \to nat$
$Nil : list$
$Cons : nat \times list \to list$
functions:
$+ : nat \times nat \to nat$
$: nat \times nat \to nat$
$power : nat \times nat \to nat$
$poinsum : list \times list \to list$
$bin : nat \to nat$
$seq : nat \to nat$
axioms
$x + 0 = x$
$x + s(y) = s(x + y)$
$0 * x = 0$
$s(x) * y = (x * y) + y$
$x * (y + z) = (x * y) + (x * z)$
$power(0, x) = s(0)$
$power(s(n), y) = y * power(n, y)$
$pointsum(Nil, z) = z$
$pointsum(z, Nil) = z$
$pointsum(cons(x_1, z_1), cons(x_2, z_2)) = cons(x_1 + x_2, pointsum(z_1, z_2))$
$bin(0) = cons(s(0), Nil)$
$bin(s(x)) = pointsum(cons(0, bin(x)), bin(x))$
$seq(x, Nil) = 0$
$seq(x, cons(y, z)) = y + (x * seq(x, z))$
lemma
$seq(x_1, pointsum(x_2, x_3)) = seq(x_1, x_2) + seq(x_1, x_3)$
conjecture
$power(x_1, s(x_2)) = seq(x_2, bin(x_1))$

Figure 6: binomial coefficients

[4] R. S. Boyer and J. S. Moore. *A computational Logic Handbook*. Academic Press, 1988.

[5] Bishcop C. Brock and Warren A. Hunt. Report on the formal specification and partial specification on the viper. Technical Report 46, Computational Logic, January 1990.

[6] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual Version 5.10. Technical Report 0177, INRIA, July 1995.

[7] J. Courant. Explicitation de preuves par récurrence implicite. Mémoire de DEA (Unpublished), June 1994.

[8] L. Fribourg. A strong restriction of the inductive completion procedure. In *Proceedings 13th International Colloquium on Automata, Languages and Programming*, volume 226 of *LNCS*, pages 105–115. Springer-Verlag, 1986.

[9] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, October 1982.

[10] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq Proof Assistant, A Tutorial. Technical Report 0178, INRIA, July 1995.

[11] Warren A. Hunt. System verification. *Journal of automated reasoning*, 5(4), December 1989.

[12] J.-P. Jouannaud and E. Kounalis. Proof by induction in equational theories without constructors. In *Proceedings of the first IEEE Symposium on Logic in Computer Science*, pages 358–366, Cambridge (Mass., USA), 1986.

[13] Matt Kaufmann. An extension of the boyer-moore theorem prover to support first-order quantification. *Journal of Automated Reasoning*, 9:355–372, December 1992.

[14] E. Kounalis and M. Rusinowitch. A mechanization of conditional reasoning. In *First International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 1990.

[15] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In *Proceedings of the American Association for Artificial Intelligence Conference*, pages 240–245, Boston, July 1990. AAAI Press and MIT Press.

[16] Zhaohui Luo and Randy Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, 1992.

[17] Lena Magnusson. The new implementation of alf. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[18] U. S. Reddy. Term rewriting induction. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction*, volume 449 of *LNCS*, pages 162–177, Kaiserlautern (Germany), 1990. Springer-Verlag.

[19] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. Lusk and R. Overbeek, editors, *Proceedings 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181, Argonne (Ill., USA), 1988. Springer-Verlag.

A Simulating a polynomial ordering in our logical framework

We can simulate a polynomial ordering on clauses in the following way:

- we add to Ax the axioms of first-order arithmetic;
- then we define an interpretation in *nat* for each sort s other than *nat*, *i.e.* for each sort s other than *nat*, we add a new function symbol μ_s and the following axiom:
$$\Gamma \vdash_{rec} \mu_s(t) <_{nat} \mu_s(t') \iff t <_s t'$$
- for each clause C we can add a n-ary function μ_C where n is the number of free variables in C ; and we add the rule schema:
$$\Gamma \vdash_{rec} \mu_{C_1}(\vec{t_1}) <_{nat} \mu_{C_2}(\vec{t_2}) \iff C_1(\vec{t_1}) \sqsubset_{C_1,C_2} C_2(\vec{t_2})$$
- for each n-ary symbol function f of the specification, we choose a natural c_f and n naturals $d_{f,1}, \ldots, d_{f,n}$, and we add the rules:
$$\vdash_{rec} \mu_{\wedge_{i=1}^{h} a_i =_{s_i} b_i \Rightarrow \vee_{i=1}^{k} c_i =_{s'_i} d_i}(\vec{t}) = \sum_{i=1}^{k} \nu(c_i)(\vec{t}) + \nu(d_i)(\vec{t})$$

  where ν is defined on terms as follows:
$$\begin{aligned} \nu(x) &= \mu_s(x) \text{ where } s \text{ is the sort of } x \\ \nu(f(\vec{t})) &= c_f + \sum_k d_{f,k}\nu(\vec{t}_k) \end{aligned}$$

  where $\vec{t}_k$ denotes the k-th element of vector $\vec{t}$ and
$$\vdash_{rec} \mu_s(C(x)) = c_C + \sum_k d_{C,k}\mu(\vec{t}_k) \text{ where } C \text{ is a constructor}$$

For our example, section 2.3, we can choose the following coefficients:

$$\begin{array}{rclrclrcl} c_{True} &=& 1 & & & & & & \\ c_{False} &=& 1 & & & & & & \\ c_{length} &=& 2 & d_{length,1} &=& 2 & d_{insert,2} &=& 2 \\ c_{insert} &=& 2 & d_{insert,1} &=& 2 & d_{insert,2} &=& 2 \\ c_{<=} &=& 1 & d_{<=,1} &=& 1 & d_{<=,2} &=& 1 \\ c_{Nil} &=& 0 & & & & & & \\ c_{Cons} &=& 1 & d_{Cons,1} &=& 1 & d_{Cons,2} &=& 1 \end{array}$$

B Proofs of given theorems

Proof 4 (of theorem 1) *We prove this theorem by induction: the second and the third cases are easy. Indeed, for the third one, from* $\Gamma, \vec{y} \sqsubseteq_{C',C} \vec{x} \vdash_{rec} C'(\vec{y})$ *(where the* $\vec{y}$ *are not free in* Γ*) one can deduce* $\Gamma \vdash_{rec} \forall \vec{y}\ \vec{y} \sqsubseteq_{C',C} \vec{x} \Rightarrow C'(\vec{y}))$*, and the application of the cut rule to these proofs and to the proof of* $\{\forall \vec{y}\ \vec{y} \sqsubseteq_{C',C} \vec{x} \Rightarrow C'(\vec{y})) \mid C' \in E' \cup H\} \vdash_{rec} C(\vec{x})$*, which is a property of any* I' *system.*

Let us now prove the first case.

Notice first that if recurse_clauses *called with parameters* $(C, G, \Gamma \vdash_{rec} C(\vec{x}))$ *calls itself recursively with parameters* $(C', G', \Gamma' \vdash_{rec} D')$ *then* $G \subset G'$*, there exists* Δ

such that $\Gamma' \equiv \Gamma\Delta$, *there exists* $\vec{y}$ *such that* $D' \equiv C'(\vec{y})$, *and* $\Gamma' \vdash_{rec} \vec{y} \sqsubseteq_{C',C} \vec{x}$. *Moreover, if* C *disappears with the* ***generate*** *rule, then* $\Gamma' \vdash_{rec} \vec{y} \sqsubset_{C',C} \vec{x}$.

Then, we can deduce that the following proposition is verified during the computation: when recurse_clauses *is called with* $(C, G, \Gamma \vdash_{rec} C(\vec{x}))$, *then, for each* $C' \in G$, *we have* $\Gamma \vdash_{rec} \forall \vec{y}\ \vec{y} \sqsubseteq_{C',C} \vec{x} \Rightarrow C'(\vec{t})$ *; indeed this is true at initialisation time since* G *is empty, and this property is preserved during the computation.*

As a particular case, when recurse_clauses *is called with* $(C, G, \Gamma \vdash_{rec} C(\vec{x}))$, *if* C *is in* G, *then* $\Gamma \vdash_{rec} C(\vec{x})$.

Proof 5 (of theorem 2) *By absurdity: otherwise, one branch would be infinite. Since* $\bigcup_{i=0}^{n} E_i$ *is finite, this branch would define a cycle on clauses. Since a branch terminates on clauses* C *such that* $C \in G$, *none of these clauses in the cycle disappears with the* ***generate*** *rule nor can belong to one of the* H_i. *But if* recurse_clauses *called with* C *recursively calls itself with* C' *where neither* C *nor* C' *belong to one of the* H_i, *then this means that* C' *disappears at a step after* C. *Therefore, the existence of a cycle is absurd.*

Proof 6 (of theorem 3) *Let us first inspect the case of* ***generate***.

For every CV substitution σ, *every* C' *in* $E \cup H \cup C$, *and every* θ *constructor substitution such that* $C'\theta \prec_c C\sigma$, *we have*

$$\vdash_{rec} \vec{z}\theta \sqsubset_{C',C} \vec{x}\sigma$$

Moreover, for every k *in* K_σ, *we have*

$$\vdash_{rec} \vec{x} \sqsubset_{C_k,C} \vec{x}\sigma$$

As we have $\{\forall \vec{x} C'(\vec{x}) \mid C' \in L\} \cup \{t\theta \mid t \in E \cup H \cup \{C\} \cup \{C_1, \ldots C_n\}\ \theta$ *constructor substitution and* $t\theta \prec_c C\sigma\} \vdash_{rec} (C\sigma)$ *we have*

$$\{\forall \vec{z}(\vec{z} \sqsubset_{C',C} \vec{x}\sigma \Rightarrow C'(\vec{z}) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \vdash_{rec} C(\vec{x}\sigma)$$

We deduce then

$$\vdash_{rec} \{\forall \vec{z}(\vec{z} \sqsubset_{C',C} \vec{x}\sigma \Rightarrow C'(\vec{z}) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \Rightarrow C(\vec{x}\sigma)$$

Then, by multiple application of axiom 1, we have

$$\vdash_{rec} \{\forall \vec{z}(\vec{z} \sqsubset_{C',C} \vec{x} \Rightarrow C'(\vec{z}) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \Rightarrow C(\vec{x})$$

So we can conclude

$$\{\forall \vec{z}(\vec{z} \sqsubset_{C',C} \vec{x} \Rightarrow C'(\vec{z}) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \vdash_{rec} C(\vec{x})$$

The case of ***simplify*** (resp. ***delete***) *is similar, though simpler. For every constructor substitution* θ *and every* C' *in* $E \cup H \cup \{C_1, \ldots, C_n\}$ (resp. $E \cup H$) *such that* $C'\theta \prec_c C$, *we have:*

$$\vdash_{rec} z\theta \sqsubseteq_{C',C} x\sigma$$

And for k *in* $\{1, \ldots n\}$, *we have* $\vdash_r ecx \sqsubset_{C_k,C} x$.

As $\{\forall \vec{x} C'(\vec{x}) \mid C' \in L\} \cup \{t\theta \mid t \in E \cup H \cup \{C\} \cup \{C_1, \ldots C_n\}\ \theta$ *constructor substitution and* $t\theta \preceq_c C\} \vdash_{rec} C$ (resp. $\{\forall \vec{x} C'(\vec{x}) \mid C' \in L\} \cup \{t\theta \mid t \in E \cup H \cup \{C\}\ \theta$ *constructor substitution and* $t\theta \preceq_c C\} \vdash_{rec} C$) *we conclude:*

$$\{\forall \vec{z}(\vec{z} \sqsubseteq_{C',C} \vec{x} \Rightarrow C'(\vec{z}) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \vdash_{rec} C(\vec{x})$$

(resp. $\{\forall \vec{z}(\vec{z} \sqsubseteq_{C',C} \vec{x} \Rightarrow C'(\vec{z}) \mid C' \in E \cup H \cup \{C_1, \ldots, C_n\}\} \vdash_{rec} C(\vec{x})$)