Vérification des programmes logiques.

Sorin Craciunescu

Ecole Polytechnique

Thèse de Doctorat en Informatique

Preuves d'Equivalence de Programmes Logiques

Proving the Equivalence of Logic Programs

Sorin Craciunescu

Soutenue le 24 mars 2004

Directeur de thèse: François Fages

Composition du jury:

M. Gilles Dowek, président du jury
M. Gérard Ferrand, rapporteur
M. Dale Miller, rapporteur
M. François Fages, directeur de thèse
M. Mathieu Jaume

Contents

1 Introduction **9**

1.1 Logical programming languages 9

1.2 The CLP language 11

1.3 Proving the success equivalence 13

2 The CLP∀ Language **17**

2.1 Basic definitions 18

2.2 Syntax of the CLP∀ language 20

2.3 The constraint domain 23

2.4 Finite success semantics 27

2.5 Infinite success semantics 40

3 An induction based proof system for finite success equivalence **47**

3.1 Increasing approximations of predicates 48

3.2 Least fixed point semantics of CLP∀ 53

3.3 An induction-based proof system 56

3.4 Soundness of the proof system without induction 60

3.5 Soundness of the induction rule 67

3.6 A weaker condition for the soundness of the system 79

3.7 Proving other properties through success inclusion 90

4 A coinduction based proof system for infinite success equivalence **95**

4.1 Decreasing approximations of predicates 96
4.2 Greatest fixed point semantics of CLP∀ 98
4.3 A coinduction-based proof system 101
4.4 Equivalence of the two proof systems 108

5 Implementation, examples and comparison to previous work 111
5.1 Implementation of a proof assistant 112
5.2 Natural numbers 124
5.3 Lists and permutations 132
5.4 Proof of quicksort 136
5.5 Comparison with previous work 143

6 Conclusion and Future Work 149

Résumé.

Le but de ce travail est de proposer un système formel pour prouver que l'ensemble des succès d'un programme logique est inclus dans l'ensemble correspondant d'un autre programme. Cela permet de prouver que deux programmes logiques, un qui représente la spécification et un représentant l'implantation sont équivalents.

Le langage logique considéré est CLP$\forall$ qui est le langage de programmation logique avec contraines (CLP) auquel est ajouté le quantificateur universel. Nous présentons les sémantiques des succès finis et infinis et montrons qu'elles peuvent être exprimées comme le plus petit et le plus grand point fixe du même opérateur.

Un système de preuve pour l'inclusion des succès finis est présenté. Le système utilise pour les opérateurs et les quantificateurs logiques les mêmes règles que la logique du premier ordre. Pour raisonner sur les prédicats récursifs le système contient une règle d'induction. Nous prouvons la correction du système sous certains conditions.

Un système analogue pour l'inclusion des succès infinis est présenté. La règle d'induction est remplacée par une règle de coinduction. La correction est démontrée sous conditions analogues. Le deux systèmes sont équivalents sous certains conditions.

Une implantation a été réalisée sous la forme d'assistant de preuve écrit en Prolog. Le programme a environ 4000 lignes et contient des procédures simples mais efficaces de recherche de preuves. Nous présentons des exemples de preuves réalises avec ce programme parmi lesquels la preuve de correction de quicksort.

Preface

Summary. The goal of this work is to present a formal system that can be used to prove the success equivalence of logic programs. By proving success equivalence one can prove that a program representing the specification and another one representing the implementation are equivalent. This can be done by proving that the set of successes of each program is included in the corresponding set of the other one.

The language studied is CLP$\forall$ which is the Constraint Logic Programming (CLP) language to which the universal quantifier was added. The set of finite success of a program is defined by the finite success semantics of the language. We define also the set of infinite success of a program by extending the definitions for finite success to take into account infinite (non-terminating) computations. This gives the infinite success semantics of the language. We prove that the sets of finite and infinite successes are given by the least and greatest fixed point of the same operator.

In order to prove success inclusion we propose a proof system based of the first -order classical logic. The rules for logical operators and quantifiers are the same as in first-order logic. In order to reason about recursively defined predicates we add an induction rule and we prove the correctness of the system under certain conditions. This constitutes the main contribution of this work. The proof of other properties can be reduced to the proof of success inclusion.

The same proof system can be used to prove infinite success inclusion by replacing the induction rule with a coinduction rule. We prove the correctness of the system under conditions analogous to those for finite successes.

We also show that the two systems are equivalent under some conditions related to the existence of negated constraints.

The implementation of a proof assistant written in Prolog is presented. The software assists the user in building proof and has a simple but effective proof-search procedure that reduces the user's work.

Chapter structure. The first chapter is an introduction that begins by briefly presenting the concept of logic programming, then the syntax and semantics of CLP. In the last section we explain the goal of proving success inclusion.

The second chapter is dedicated to the CLP$\forall$ language. It presents its syntax, the semantics of finite and infinite successes.

The third chapter defines the induction based formal system for proving finite success inclusion. Its correctness is proved under two different conditions.

The fourth chapter defines the coinduction based system for proving infinite success inclusion. Its correctness is proved under conditions that are analogous to the ones for finite success. The equivalence of the two systems is also proved.

The fifth chapter presents the implementation of the proof assistant. Several examples of proofs realized with its help are presented, among which the correctness of *quicksort*. A comparison with previous work closes the chapter.

The sixth chapter consists in the conclusions and perspectives for future developments of this work.

Acknowledgements. I would like to thank François Fages, my thesis advisor, for his supervision and useful suggestions which helped improve this work. I would also like to thank Sylvain Soliman, Dale Miller and Slim Abdennadher for interesting discussions and suggestions about various parts of this work. Sylvain Soliman also contributed by pointing out errors in an initial version of the manuscript.

Chapter 1

Introduction

In this chapter we present briefly the logic programming languages and state our goal of proving success inclusion of logic programs.

In the first section we recall briefly the basic ideas of the Prolog language. In the second section the syntax and semantics of the CLP (Constraint Logic Programming) language is presented following [Mah99]. In the third section we state the goal of proving success inclusion of logic programs which can be used to prove success equivalence of a specification and its implementation.

1.1 Logical programming languages

The logical programming languages were introduced by Kowalski [Kow74] and Colmerauer [Col73]. They are based of the first order classical logic and have a declarative nature.

The first incarnation of a logic language was Prolog [Col73]. While initially it was conceived for human-machine interfaces and later for artificial intelligence it gradually became a full programming language. This was achieved adding non logical (imperative) capabilities and therefore sacrificing the simplicity and elegance given by its logic heritage.

In the following decades there were several developments of Prolog among which Concurrent Prolog [Sha83], Constraint Logic Programming (CLP)[Jaf87], Concurrent Constraint Programming (CCP)[Sar90], Linear Concurrent Con-

straint Programming (LCC) [FRS01].

Some of the most popular Prolog implementations are GNU-Prolog [Dia01], SWI-Prolog [Wie], SICStus Prolog [Car91], Eclipse [Che03].

The basic operation of Prolog is the unification of terms represented as $t_1 = t_2$. More complex expressions can be built using the conjunction represented by comma "," and the disjunction represented by a semicolon ";".

The syntax of (pure) Prolog is:

$$
\begin{aligned}
program &::= \epsilon \,|\, definition.program \\
definition &::= p(\vec{X}) : -exp \\
exp &::= t_1 = t_2 \,|\, p(\vec{t}) \,|\, exp,\ exp \,|\, exp;\ exp
\end{aligned}
$$

where

ϵ is the void string

t_1, t_2 are terms

$\vec{X}, \vec{t}$ are vectors of variables and terms, respectively

$p(\vec{X}) : -exp$ represents the definition of the predicate $p(\vec{X})$ which is analogous to the definition of a procedure in imperative languages. The unification can be seen as a built-in predicate.

Practical implementations of Prolog have many other built-in predicates, some of them - like *var* - having a non logical nature. A predicate can be defined by more than one construct (called clause) of the form $p(\vec{X}) : -exp$ but this is redundant as the same effect can be achieved using the disjunction ";".

The following code in an example of a Prolog program:

```
nat(X):-X=0
      ; X=s(Y), nat(Y).
```

The interpretation of this definition is: $nat(X)$ is true (or succeeds) if $X = 0$ is true or there exists a term Y such that $X = s(Y)$ and $nat(Y)$ are true.

For example $nat(0)$ succeeds because $0 = 0$ is true. Also $nat(s(0))$ succeeds because by taking Y to be 0 we have that $s(s(0)) = s(s(0))$ and $nat(0)$

are true. In general the predicate $nat(X)$ succeeds when X is a term of the form $s(s(...s(0)...))$ which represent the natural numbers written in base one.

This example introduces intuitively the notion of set of successes of a predicate. This notion will be central for the rest of this work.

1.2 The CLP language

The CLP (Constraint Logic Programming) language was introduced by Jaffar and Lassez in 1987 [Jaf87]. It is an evolution or generalization of Prolog. While in Prolog the basic operation is the unification of two terms, in CLP it is adding a constraint to the global store.

The constraints are relations over some set. The set and the constraints form the constraint system which is not fixed but can vary. Therefore we can say that there are multiple instances of CLP languages. The constraint domain X is a parameter of the CLP language, thus the notation CLP(X).

Examples of constraint domains are:

- the Herbrand domain composed of the set of terms and the unification of terms as constraint. Hence Prolog is an instance of the CLP languages
- the finite domains have finite sets of elements for example set of booleans - in this case the constraints are logic relations
- the domains of integer, rational or real numbers - the typical constraints are addition, multiplication, comparison operators, etc.
- more exotic domains like the domains of intervals, lists, etc..

An implementation of the CLP language must contain a *solver* for its domain of constraints. A *solver* is a function that maps a constraint to either *true*, *false* or *unknown* indicating that the constraint is satisfiable, unsatisfiable or that the solver cannot tell.

In the rest of the section we will present briefly the CLP language following Maher [Mah99].

The syntax of CLP is:

$$
\begin{aligned}
program &::= \epsilon \,|\, definition.program \\
definition &::= p(\vec{X}) : -exp \\
exp &::= c \,|\, p(\vec{t}) \,|\, exp, exp
\end{aligned}
$$

where

ϵ is the void string

c is a constraint

$\vec{X}$, $\vec{t}$ are vectors of variables and terms, respectively.

A literal is either a constraint c or a call $p(\vec{t})$.

The operational semantics of CLP is given in terms of transitions between states. A state is a pair $< G|c >$ where G - the current goal - is a conjunction of literals or $\circ$ and c is a constraint which represents the current store.

The initial state is of the form $< G_0|true >$ where G_0 is the initial goal. We will denote the solver function by *solv*.

Let the current state be $< L_1, .., L_n|c >$. It can be changed by the following transitions:

- if L_i is a constraint and $solv(c \wedge L_i) \neq false$ then the current state becomes $< L_1, .., L_{i-1}, L_{i+1}, .., L_n|c \wedge L_i >$
- if L_i is a constraint and $solv(c \wedge L_i) = false$ then the current state becomes $< \circ|false >$
- if L_i is a call $p(\vec{t})$ and there exists a definition $p(\vec{s}) : -B$ in the program then the current state becomes $< L_1, .., L_{i-1}, s_1 = t_1, .., s_m = t_m, B, L_{i+1}, .., L_n|c >$
- if L_i is a call $p(\vec{t})$ and there is no definition $p(\vec{s}) : -B$ in the program then the current state becomes $< \circ|false >$

For the last two transitions we suppose that the free variables in the definitions have been renamed such that they are not used in the current state.

The transition between states will represented by an arrow $\rightarrow$. If $< G_0|c > \rightarrow ... \rightarrow < \circ|c >$ and $c \neq false$ then we call an *answer* of the goal G_0 the constraint $\overline{\exists}_{vars(G_0)}c$ which represents the existential quantification of c over all its free variables with the exception of the free variables of G_0. The answers correspond to the successes of a Prolog program.

1.3 Proving the success equivalence

One of the current goals of Computer Science is reducing the number of faults ("bugs") that appear in software, especially in complex software. Different approaches have been proposed like high level languages that simplify programming, type systems that can detect common programming errors and proving formally that programs have the intended properties.

We are interested in the last one: formal proofs of programs. In this approach after the program is written one tries to prove that it has certain intended properties. In this process the faults in the program can be found and corrected. The formal proof can be checked by a computer which gives a high degree of assurance that it is correct.

In the context of logic programming one of the most useful properties to prove is the equivalence of programs: two programs are equivalent when they give the identical outputs for the same input. One of the programs - called the specification - can be written such as to be very simple and easy to understand but possibly inefficient. The other one - called the implementation - can be written to be efficient but possibly complex. The specification has a higher probability of being correct. By proving that the two programs are equivalent the number of faults in the implementation is reduced dramatically.

For logic programs proving that two programs are equivalent reduces to proving that their sets of successes are equal.

The goal of this work is to provide a proof system for proving the success inclusion. In other words we want to be able to prove formally that the set of successes of a goal G_1 is included in the set of successes of a goal G_2.

Recall the previous example of a Prolog or CLP(H) program that defines the predicate *nat* whose successes are the natural numbers. We can similarly define the predicate *int* whose successes are the integer numbers.

The two predicates are:

```
nat(X):-X=0
      ; ∃Y.(X=s(Y), nat(Y)).

int(X):-X=0
      ; ∃Y.(X=s(Y), int(Y))
      ; ∃Z.(X=p(Z), int(Z)).
```

Intuitively the set of successes $nat(X)$ is included in the set of successes of $int(X)$.

Since our goal is to prove it formally and since the Prolog (and other logic languages) use logic operators we can try using the first order logic as proof system for success inclusion. For this purpose we have made explicit that the local variables Y and Z in the previous definitions can be viewed as existentially quantified.

Translating the definitions above into logic formulae gives:

$$nat(X) \leftrightarrow X = 0 \vee \exists Y.(X = s(Y) \wedge nat(Y))$$
$$int(X) \leftrightarrow X = 0 \vee \exists Y.(X = s(Y) \wedge int(Y)) \vee \exists Z.(X = p(Z) \wedge int(Z))$$

Let $Defs$ be a first order formula, the conjunction of the two formulae above.

We try to derive

$$Defs, nat(X) \vdash int(X)$$

using the Gentzen calculus for the first order logic:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{Defs, X = 0 \vdash X = 0 \vee \ldots}(\vee R, axiom) \quad \Omega}{Defs, X = 0 \vee \exists Y.(\ldots) \vdash X = 0 \vee \exists Y.(\ldots) \vee \exists Z.(\ldots)}(\vee L)}{Defs, X = 0 \vee \exists Y.(\ldots) \vdash int(X)}(\rightarrow L, axiom)}{Defs, nat(X) \vdash int(X)}(\rightarrow L, axiom)$$

where Ω is the subtree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{Defs, X = s(Y), nat(Y) \vdash X = 0}(axiom) \quad \dfrac{Defs, nat(Y) \vdash int(Y)}{Defs, X = s(Y), nat(Y) \vdash int(Y)}(wR)}{Defs, X = s(Y), nat(Y) \vdash \exists Y.(\ldots)}(\exists R, \wedge R)}{Defs, X = s(Y), nat(Y) \vdash X = 0 \vee \exists Y.(\ldots) \vee \exists Z.(\ldots)}(\vee R, \vee R, wR, wR)}{Defs, \exists Y.(\ldots) \vdash X = 0 \vee \exists Y.(\ldots) \vee \exists Z.(\ldots)}(\exists L, \wedge L)$$

Therefore we start with $Defs, nat(X) \vdash int(X)$ and we obtain $Defs, nat(Y) \vdash int(Y)$ where Y is a fresh variable. If we continue trying to derive $Defs, nat(Y) \vdash int(Y)$ we will never be able to finish the derivation but instead we will obtain an infinite derivation.

We conclude that the rules of the first order logic are not enough for proving success inclusion. The solution we propose in chapter 3 is to add an induction rule that will allow us to close the derivation. We will prove that the corresponding proof system is correct under certain restrictions. The work presented here was presented in an initial form in [Cra01] and [Cra02].

One of the early works about program equivalence in pure Prolog is [Tam84] . The authors prove that the folding and unfolding of predicates preserves program equivalence. Moreover they study the effects of combining folding and unfolding with other useful transformations. They prove that under certain conditions those transformations can be applied in any order to any program and the resulting program is equivalent to the initial one. The program equivalence in this case is defined as the equality between the set of successful ground goals. Later works on transformations of logic programs are [Tam92], [Ben98], [Kan86], [Pet99].

Another approach to proving program properties is using a formal proof system, the approach we use in this work. This approach was used in [BGMP97], [Sta98], etc.. A more detailed comparison with these works will be presented in section 5.5.

For Linear Concurrent Constraint Programming (LCC) languages the approach proposed in [FRS01] uses the phase semantics of Linear Logic in order to prove temporal properties like safety and liveness.

Chapter 2

The CLP∀ Language

This chapter is dedicated to the CLP∀ language, its syntax and semantics. The CLP∀ language is the Constraint Logic Programming language (CLP) augmented with the universal quantifier. Like CLP, it is parametrized by the constraint domain and therefore CLP∀ can be referred to as a class of programming languages. In this work it will be referred to as a language for simplicity and because the syntax, semantics and properties of all instances considered here will be the same.

The first section presents the basic definitions used throughout the book which are the usual definitions for first order logic: terms, substitutions, etc.

The second section presents the syntax of the CLP∀ language and continues definitions for the sets of expressions (or goals), free variables of an expression, valid programs, etc..

The constraint domain is presented in section three. It is given by an entailment relation between two constraints and its extension to entailment between a conjunction and a disjunction of constraints. The entailment relation is subject to certain restrictions in order to have a meaningful semantics and to ensure that the correctness of the proof systems presented in the following chapters hold.

The fourth section contains the finite success semantics of the language. The notion of success of an expression (goal) is similar to the corresponding notion for CLP. It is defined by means of a sequent calculus. Its rules are

similar to the rules of first order logic, more so since the universal quantifier is also present in the language. Proofs of some basic properties of the semantics follow.

The fifth section presents the infinite success semantics. Intuitively an infinite success corresponds to an execution that terminates successfully (like a finite success) or doesn't terminate. To define the set of infinite successes we define the notion of infinite derivation, an extension of the notion of finite derivation. We prove for the set of infinite successes the same properties that we proved for the set of finite successes.

2.1 Basic definitions

This section contains basic definitions which will be used through the rest of the document. They are the usual definitions found in the first order logic books: the sets of variables, terms and substitutions.

A difference is the set of constraints which parallels the set of predicates. Constraints are analogous to predicates, except that unlike constraints, predicates can have definitions which allow one to reason about them. On the other hand, the properties of constraints are "encapsulated" into the relations $\vdash_D$ and $\vdash_{DM}$ defined in the next section.

We assume that all symbols defined below can be subscripted and/or superscripted.

$FunctN$ is the set of function names denoted by f, g.

$PredsN$ is the set of predicate names denoted by p, q, r.

$ConstrN$ is the set of constraint names denoted by a, b which contains $true$ and $false$ of arity 0.

$ar : FunctN \cup PredsN \cup ConstrN \rightarrow N$ is the function that gives the arity of a function, predicate or constraint name.

$Vars$ is the infinite set of variables denoted by U, V, W, X, Y, Z.

$Terms$ is the set of terms denoted by s, t which is the smallest set satisfying the following properties:

$$Terms \supseteq Vars$$
$$Terms \supseteq \{f(t_1,..,t_n) \,|\, f \in FunctN,\, ar(f) = n,\, t_1,...,t_n \in Terms\}$$

We use the upper arrow notation for vectors:
$\vec{X}$ denotes a vector $(X_1, X_2, ..., X_n)$ of variables and
$\vec{t}$ denotes a vector $(t_1, t_2, ..., t_m)$ of terms.

$ConstrA$ is the set of atomic constraints of the form $a(t_1, ..., t_n)$ where a is a constraint name of arity n and $t_1, ..., t_n$ are terms .

Constraints is a set of constraints which are first order logical formulae built from atomic constraints, the logical connectives $\neg, \wedge \vee, \rightarrow$ and the quantifiers $\exists, \forall$. They will be denoted by c, d, e. Specifically *Constraints* is the smallest set satisfying the following properties:

$$Constraints \supseteq ConstrA$$
$$Constraints \supseteq \{\neg c \,|\, c \in Constrainsts\}$$
$$Constraints \supseteq \{c_1 \wedge c_2 \,|\, c_1, c_2 \in Constrainsts\}$$
$$Constraints \supseteq \{c_1 \vee c_2 \,|\, c_1, c_2 \in Constrainsts\}$$
$$Constraints \supseteq \{c_1 \rightarrow c_2 \,|\, c_1, c_2 \in Constrainsts\}$$
$$Constraints \supseteq \{\exists c \,|\, c \in Constrainsts\}$$
$$Constraints \supseteq \{\forall c \,|\, c \in Constrainsts\}$$

Subst is the set of finite substitutions which are functions denoted by σ, τ, such that $\sigma : Vars \rightarrow Terms$ and for which the set

$$\{X \in Vars \,|\, \sigma(X) \neq X\}$$

is finite. They can be written as compositions of a finite number of substitutions σ_i that change the value of a single variable:

$$\sigma = \sigma_n \circ ... \circ \sigma_1$$

2.2 Syntax of the CLP∀ language

The CLP∀ language is the usual CLP (Constraint Logic Programming) language augmented with the universal quantifier. In this section we specify the syntax of CLP∀ and give some usual definitions like the set of expressions and the free variables of an expression. The universal quantifier is part of the language. In other works like [Bru93] the universal quantifier is used for compiling negations of predicates.

The syntax of CLP∀ in Backus Naur form (BNF) is:

$$\begin{aligned} program &::= \epsilon \mid definition.program \\ definition &::= p(\vec{X}) : -exp \\ exp &::= tell(c) \mid p(\vec{t}) \mid exp,\ exp \mid exp;\ exp \mid \exists X.exp \mid \forall X.exp \end{aligned}$$

where
ϵ is the void string
c is a constraint
$\vec{X}, \vec{t}$ are vectors of variables and terms, respectively.
A definition of the form

$$p(\vec{X}) : -exp$$

is also called "clause", $p(\vec{X})$ is called "the head of the clause" and exp is called "the body of the clause". The variables in $\vec{X}$ are called the arguments of the clause.

Exp is the set of expressions defined by the rule exp above and denoted by $E, F, G, Body$. In other words, Exp is the smallest set which satisfies the following properties:

$$\begin{aligned} &Exp \supseteq \{tell(c) \mid c \in Constraints\} \\ &Exp \supseteq \{p(t_1, ..., t_n) \mid p \in PredsN,\ ar(p) = n,\ t_1, ..., t_n \in Terms\} \end{aligned}$$

$$Exp \supseteq \{(E_1, E_2) \mid E_1, E_2 \in Exp\}$$
$$Exp \supseteq \{(E_1; E_2) \mid E_1, E_2 \in Exp\}$$
$$Exp \supseteq \{\exists X.E \mid X \in Vars,\ E \in Exp\}$$
$$Exp \supseteq \{\forall X.E \mid X \in Vars,\ E \in Exp\}$$

An atomic expression is an expression of the form $tell(c)$ or $p(t_1, ..., t_n)$.

Remark: the notation *Body* for an expression is reserved for the case where the expression is the body of a clause.

$vars : Terms \cup Constraints \cup Exp \rightarrow Vars$ is the function that gives the set of variables of a term, constraint or expression. It's defined inductively the usual way:

$$vars(X) = \{X\}$$
$$vars(f(t_1, ..., t_n)) = \bigcup_{i=1..n} vars(t_i)$$
$$vars(a(t_1, ..., t_n)) = \bigcup_{i=1..n} vars(t_i)$$
$$vars((\neg c)) = vars(c)$$
$$vars((c_1 \wedge c_2)) = vars(c_1) \cup vars(c_2)$$
$$vars((c_1 \vee c_2)) = vars(c_1) \cup vars(c_2)$$
$$vars((c_1 \rightarrow c_2)) = vars(c_1) \cup vars(c_2)$$
$$vars((\exists X.c)) = \{X\} \cup vars(c)$$
$$vars((\forall X.c)) = \{X\} \cup vars(c)$$
$$vars(tell(c)) = vars(c)$$
$$vars(p(t_1, ..., t_n)) = \bigcup_{i=1..n} vars(t_i)$$
$$vars((E_1, E_2)) = vars(E_1) \cup vars(E_2)$$
$$vars((E_1; E_2)) = vars(E_1) \cup vars(E_2)$$
$$vars((\exists X.E)) = \{X\} \cup vars(E)$$
$$vars((\forall X.E)) = \{X\} \cup vars(E)$$

$fv : Terms \cup Constraints \cup Exp \rightarrow Vars$ is the function that gives the set of free variables of a term, constraint or expression. Its definition is similar to the definition of *vars* except for the cases:

$$fv((\exists X.c)) = fv(c) - \{X\}$$
$$fv((\forall c.E)) = fv(c) - \{X\}$$
$$fv((\exists X.E)) = fv(E) - \{X\}$$
$$fv((\forall X.E)) = fv(E) - \{X\}$$

$E\{X/t\}$ denotes the expression E where all the free occurrences of X have been replaced by t. It can be defined inductively as:

$$X\{X/t\} = t$$
$$Y\{X/t\} = Y \text{ if } Y \neq X$$
$$f(t_1, ..., t_n)\{X/t\} = f(t_1\{X/t\}, ..., t_n\{X/t\})$$
$$tell(c)\{X/t\} = tell(c\{X/t\})$$
$$p(t_1, ..., t_n)\{X/t\} = p(t_1\{X/t\}, ..., t_n\{X/t\})$$
$$(E_1, E_2)\{X/t\} = (E_1\{X/t\}\,,\ E_2\{X/t\})$$
$$(E_1; E_2)\{X/t\} = (E_1\{X/t\}\,;\ E_2\{X/t\})$$
$$(\exists X.E)\{X/t\} = (\exists X.E)$$
$$(\exists Y.E)\{X/t\} = (\exists Y.E\{X/t\}) \text{ if } Y \neq X$$
$$(\forall X.E)\{X/t\} = (\forall X.E)$$
$$(\forall Y.E)\{X/t\} = (\forall Y.E\{X/t\}) \text{ if } Y \neq X$$

The cases created by the structure of constraints are similar to those for expressions and were omitted.

σE denotes the expression E in which each free occurrence of a variable X is replaced by $\sigma(X)$.

$E(X_1, ..., X_n)$ or $E(\vec{X})$ denotes a parametric expression - an expression E where some of its free variables, $X_1, ..., X_n$ have been chosen as parameters.

$E(t_1, ..., t_n)$ is the expression E where the free occurrences of its parameters have been replaced by the terms $t_1, .., t_n$. An equivalent but more cumbersome notation is $E\{X_1/t_1\}...\{X_n/t_n\}$. The parameters $X_1, .., X_n$ are to be deduced from the context.

Progs is the set valid programs defined by the rule *program* above and the following restrictions:

- a program can contain at most one definition for each predicate name (of a given arity)
- there are no free variables in the body of any clause except for its arguments.

Alternatively, the set of valid programs can be defined as the smallest set which satisfies the following properties:

$$\begin{aligned} &Progs \supseteq \{\epsilon\} \\ &Progs \supseteq \{p(X_1, ..., X_n) : -Body.P \mid P \in Progs, \\ &\qquad fv(Body) - \{X_1, ..., X_n\} = \emptyset\} \end{aligned}$$

A natural convention - borrowed from Prolog - is to consider that all free variables in the body of clause which are not arguments of the clause are existentially quantified over the entire body. This convention simplifies the writing of source code in CLP$\forall$ and other logical languages.

2.3 The constraint domain

Like CLP, the CLP$\forall$ language is parametrized by a constraint domain. In this section we introduce the relation $\vdash_D$ which, together with the set of constraint names $ConstrN$, defines the domain of constraints. We also introduce its extension $\vdash_{DM}$ to multiple constraints.

As mentioned in the section 2.1 , constraints are similar to predicates. While predicates can have definitions which characterize their properties, the properties of constraints are given by the two relations presented above, $\vdash_D$ and $\vdash_{DM}$. Therefore, constraints can be seen as "black box" predicates (their definitions being unknown), their only known properties deriving from the two relations.

$\vdash_D$ is any binary relation on *Constraints* which satisfies the following properties:

- for all $c \in$ *Constraints* we have $false \vdash_D c$, $c \vdash_D true$ and $c \vdash_D c$.
- if $c_1 \vdash_D c_2$ and $c_2 \vdash_D c_3$ then $c_1 \vdash_D c_3$.
- if $c_1 \vdash_D c_2$ then $\forall t \in Terms, \forall X \in Vars$ we have $c_1\{X/t\} \vdash_D c_2\{X/t\}$.

The first two properties imply that $\vdash_D$ is reflexive and transitive and the third that it is stable with respect to variable substitution by terms.

Remark: the first-order entailment relation $\bullet \vdash_{FO} \bullet$ between constraints defined the usual way satisfies all the properties required for $\vdash_D$.

Intuitively the relation $\vdash_D$ captures the notion that a constraint "implies" another, e.g.

$$c \vdash_D d$$

is to be interpreted as "c implies d". Some restriction must be imposed on this relation in order for the soundness of the proof systems presented in the next chapter to hold. In general it is desirable to impose as few restrictions as possible. We believe that the definition of $\vdash_D$ achieves a good balance between being simple and imposing few restrictions.

Constraints* is the set defined as

$$Constraints^* = \bigcup_{n \in \mathbf{N}^*} Constraints^n$$

where $Constraints^n = Constraints \times ... \times Constraints$ ($n > 0$ times).

$\vdash_{DM}$ is a binary relation on *Constraints**for which we will use the following notation: if

$x = (c_1, .., c_n) \in Constraints^n$ and

$y = (d_1, .., d_m) \in Constraints^m$

then we will write $x \vdash_{DM} y$ as

$$c_1, .., c_2 \vdash_{DM} d_1; ..; d_m$$

The relation is uniquely defined by the following property:

$$c_1, .., c_n \vdash_{DM} d_1; ..; d_m$$

iff for all $e \in \mathit{Constraints}$ such that $\forall i \in \overline{1,n}$, $e \vdash_D c_i$ there exists $j \in \overline{1,m}$ such that $e \vdash_D d_j$.

We now extend the notation $\vdash_{DM}$ to also stand for two unary relations on $\mathit{Constraints}^n$: $\bullet \vdash_{DM}$ and $\vdash_{DM} \bullet$ defined as follows:

- $c_1, .., c_n \vdash_{DM}$ iff $c_1, .., c_n \vdash_{DM} \mathit{false}$
- $\vdash_{DM} d_1; ..; d_m$ iff $\mathit{true} \vdash_{DM} d_1; ..; d_m$

Remark: Together, $\mathit{ConstrN}$, $\vdash_D$ and $\vdash_{DM}$ form the constraint domain D which is a parameter of the CLP$\forall$ language.

The relation $\vdash_{DM}$ is the natural generalization of $\vdash_D$ to more than one constraint:

$$c_1, .., c_n \vdash_{DM} d_1; ..; d_m$$

is to be interpreted as "the conjunction of $c_1, ..., c_n$ implies the disjunction of $d_1, ..., d_m$". Moreover, by extending $\vdash_D$ to also have meaning for the specific cases where $n = 0$ or $m = 0$ we capture the notion that a conjunction of constraints "implies" false (is false) and that a disjunction of constraints "is implied" by true (is true). The practical reason for this extension is to simplify the side condition of the rule (*tell*) introduced in the next chapter where the number of constraints in the left or right side of a sequent may be zero.

Remark: the relation $\vdash_{DM}$ doesn't have to be totally computable for the proof systems defined in the following sections to be sound. However,

being only partially computable may reduce the number of proofs that can be built in those systems.

The relation $\vdash_{DM}$ is easily computable if the constraint domain has the *Independence of Negated Constraints* property (INC) [Mah99].

An important example of constraint domain is the Herbrand domain - denoted by H - where the only constraint is the unification of terms ("="). The instance of CLP where the constraint domain is H - CLP(H) - is the language Prolog [Llo87]. Hence CLP∀(H) is Prolog extended with the universal quantifier.

The Herbrand domain has *Independence of Negated Constraints* property (INC) and therefore the relation $\vdash_{DM}$ on this domain is computable.

Example: the following program, named NI, is a CLP∀(H) program (and also a Prolog program as it doesn't use the universal quantifier):

```
nat(X):-tell(X=0)
      ; ∃Y.(tell(X=s(Y)), nat(Y)).

int(X):-tell(X=0)
      ; ∃Y.(tell(X=s(Y)), int(Y))
      ; ∃Z.(tell(X=p(Z)), int(Z)).
```

It defines two predicates *nat* and *int* which have the following intuitive meaning:

- $nat(t)$ is true iff t is either 0 or a term of the form $s(s(...(0)...))$. Hence $nat(t)$ is true if and only if t is a unary representation of a natural number.

- $int(t)$ is true iff t is either 0 or a term of the form $s(s(p(...(0)...)))$ where s and p can alternate freely. Hence $int(t)$ is true if and only if t is a unary representation of a natural number.

Remark: it is obvious that an integer may have more than one representation: for example, 2 can be represented as $s(s(0))$ or $p(s(s(s(0)))$. With a slight complication we can modify $int(t)$ to be true only when t is 0, or $s(s(...(0)...))$, or $p(p(...(0)...))$ (no alternation of p and s allowed).

2.4 Finite success semantics

This section contains the finite success semantics of the CLP∀ language. First a logical semantics is presented in the form of a sequent calculus. The finite successes of a goal are defined in relation to the (finite) derivations in this calculus. We prove some useful properties of the set of successes which will be needed in the following chapters.

An important notion for a logical language is the successes of an expression (or goal). In Prolog, the successes are substitutions of the existentially quantified variables in the goal which make the goal true. In CLP and CLP∀ the successes are constraints that "imply" the goal.

We will give the logical semantics of the CLP∀ language by a sequent calculus of the form:

$$P, c \vdash_s E$$

which should be interpreted as "the constraint c is a success (or solution) of the expression (goal) E in the context of the program P". The subscript s stands for (finite) "success".

The rules of the calculus are:

$$\frac{}{P, c_1 \vdash_s tell(c_2)}(tell)$$

$(tell)$ having the side condition: $c_1 \vdash_D c_2$

$$\frac{P, c \vdash_s E_1 \quad P, c \vdash_s E_2}{P, c \vdash_s (E_1, E_2)}(,)$$

$$\frac{P, c \vdash_s E_1}{P, c \vdash_s (E_1; E_2)}(;1) \quad \frac{P, c \vdash_s E_2}{P, c \vdash_s (E_1; E_2)}(;2)$$

$$\frac{P, c \vdash_s Body(\vec{t})}{P, c \vdash_s p(\vec{t})}(def.)$$

where $p(\vec{X}) : -Body(\vec{X})$ is a clause in the program P

$$\frac{P, c \vdash_s E(t)}{P, c \vdash_s \exists X.E(X)}(\exists)$$

$$\frac{P, c \vdash_s E(Y)}{P, c \vdash_s \forall X.E(X)}(\forall)$$

$(\forall)$ having the side condition: $Y \notin fv(c) \cup vars(E(X))$.

False is the set defined by:

$$False = \{c \in Constraints \mid c \vdash_D false\}$$

$Succ : Progs \times Exp \to \wp(Constraints)$ is the function that gives the set of successes of an expression E in the context of a program P:

$$Succ(P, E) = \{c \in Constraints \mid (P, c \vdash_s E) \, is \, derivable\} - False$$

Sometimes we will omit the first parameter of *Succ* when it is implicit from the context.

The definition of *Succ* given above has the drawback that an expression E can have two successes c_1, c_2 such that $c_1 \vdash_D c_2$ but $c_2 \nvdash_D c_1$.

For example $(X = 0 \wedge Y = 0)$ and $X = 0$ are successes of $tell(X = 0)$ and $(X = 0 \wedge Y = 0) \vdash_D X = 0$ but $X = 0 \nvdash_D (X = 0 \wedge Y = 0)$. Knowing that $X = 0$ is a success of $tell(X = 0)$ implies that $(X = 0 \wedge Y = 0)$ is also

a success of $tell(X = 0)$, therefore specifying $(X = 0 \wedge Y = 0)$ as a success can be seen as "redundant".

In practice we might be interested by the "non-redundant" successes of an expression. These successes, called core successes are given by the following function:

$SuccH : Progs \times Exp \rightarrow \wp(Constraints)$ is a function defined as follows:

$$SuccH(P, E) = \{c \in Succ(P, E) \mid \forall d \in Succ(P, E), c \vdash_D d \Rightarrow d \vdash_D c\}$$

By a slight abuse of language we may refer to the core successes simply as successes.

Example: Recall the previous example (the program NI):

```
nat(X):-X=0
      ; ∃Y.(tell(X=s(Y)), nat(Y)).

int(X):-X=0
      ; ∃Y.(tell(X=s(Y)), int(Y))
      ; ∃Z.(tell(X=p(Z)), int(Z)).
```

Using the $\vdash_s$ calculus one can derive that $X = s(0)$ is a success of the expression (goal) $nat(X)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{NI, X = s(0) \vdash_s tell(X = s(0))}(tell) \quad \dfrac{\Omega}{NI, X = s(0) \vdash_s nat(0)}}{NI, X = s(0) \vdash_s \exists Y.(tell(X = s(0)), nat(0))}(,)}{NI, X = s(0) \vdash_s tell(X = 0); \exists Y.(...)}(;2, \exists)}{NI, X = s(0) \vdash_s nat(X)}(def.)$$

where Ω is:

$$\dfrac{\dfrac{\dfrac{}{NI, X = s(0) \vdash_s tell(X = s(0))}(tell)}{NI, X = s(0) \vdash_s tell(X = 0); \exists Y.(...)}(;1)}{NI, X = s(0) \vdash_s nat(0)}(def.)$$

The predicates nat and int contain the following successes:

$Succ(NI, nat(X)) \supseteq \{X = 0, X = s(0), ...\}$
$Succ(NI, int(X)) \supseteq \{X = 0, X = s(0), ..., X = p(0), X = p(s(0)), ...\}$

The core successes of *nat* and *int* are:

$SuccH(NI, nat(X)) = \{X = 0, X = s(0), ...\}$
$SuccH(NI, int(X)) = \{X = 0, X = s(0), ..., X = p(0), X = p(s(0)), ...\}$

It's obvious that $Succ(NI, nat(X))$ is included in $Succ(NI, int(X))$. A proof system that can be used to prove success inclusion will be presented in the next chapter.

Remark: The successes of an expression can be ground or not. For example *true* is a ground success of $\exists X.(tell(X = 2))$ and $X = 2$ and $Y = 2$ are non ground successes of the same expression.

Remark: The $\vdash_s$ calculus might be used to write an interpreter of CLP∀ were it not for the rule $(\exists)$. This rule needs the interpreter to "guess" a term t which will replace the existentially quantified variable X. This problem is solved in Prolog by making X an uninstatiated term which will be progressively instantiated during the execution (proof search). The same solution can be used for CLP∀.

The next proposition gives some useful properties of the set of successes which will be used for proving the soundness of the proof system presented in the next chapter. It also confirms that the logical semantics have the expected properties (e.g. $Succ(P, (\exists X.E(X))) = \bigcup_{t \in Terms} Succ(P, E(t))$).

Proposition 2.1: For each CLP∀ program P, predicate $p(\vec{X})$ defined in P by $p(\vec{X}) : -Body(\vec{X})$, expressions E, E_1, E_2 and term t the following properties hold:

1. $Succ(P, tell(c)) = \{d \in Constraints \mid d \vdash_D c\} - False.$

2. $Succ(P, p(\vec{t})) = Succ(P, Body(\vec{t}))$.
3. $Succ(P, (E_1, E_2)) = Succ(P, E_1) \cap Succ(P, E_2)$.
4. $Succ(P, (E_1; E_2)) = Succ(P, E_1) \cup Succ(P, E_2)$.
5. $Succ(P, (\exists X.E(X))) = \bigcup_{t \in Terms} Succ(P, E(t))$.
6. $Succ(P, (\forall X.E(X))) = \bigcap_{t \in Terms} Succ(P, E(t))$.
7. if $Succ(P, E_1) \subseteq Succ(P, E_2)$ then $\forall t \in Terms$, $Succ(P, E_1\{X/t\}) \subseteq Succ(P, E_2\{X/t\})$.
8. if $Succ(P, E_1) \subseteq Succ(P, E_2)$ then $\forall \sigma \in Subst$, $Succ(P, \sigma E_1) \subseteq Succ(P, \sigma E_2)$.

Proof:

1. $Succ(P, tell(c)) = \{d \in Constraints \mid d \vdash_D c\} - False$

If $d \in Succ(P, tell(c))$ then $P, d \vdash_s tell(c)$ is derivable and $d \nvdash_D false$. The only rule at the root of the derivation tree can be $(tell)$:

$$\frac{}{P,\ d \vdash_s tell(c)}(tell)$$

with $d \vdash_D c$ hence $d \in \{d \in Constraints \mid d \vdash_D c\} - False$ and therefore

$$Succ(P, tell(c)) \subseteq \{d \in Constraints \mid d \vdash_D c\} - False$$

If $d \in \{d \in Constraints \mid d \vdash_D c\} - False$ then $d \vdash_D c$ and $d \nvdash_D false$ therefore $P, d \vdash_s tell(c)$ is derivable:

$$\frac{}{P,\ d \vdash_s tell(c)}(tell)$$

and thus

$$Succ(P, tell(c)) \supseteq \{d \in Constraints \mid d \vdash_D c\} - False$$

2. $Succ(P, p(\vec{t})) = Succ(P, Body(\vec{t}))$.

If $c \in Succ(P, p(\overrightarrow{t}))$ then $P, c \vdash_s p(\overrightarrow{t})$ is derivable. The only rule at the root of the derivation tree can be (def):

$$\frac{P, c \vdash_s Body(\overrightarrow{t})}{P, c \vdash_s p(\overrightarrow{t})}(def.)$$

where $p(\overrightarrow{X}) : -Body(\overrightarrow{X})$ is the definition of p. The same tree without the root sequent, $P, c \vdash_s p(\overrightarrow{t})$, is a derivation for $c \in Succ(P, Body(\overrightarrow{t}))$.

Therefore:

$$Succ(P, p(\overrightarrow{t})) \subseteq Succ(P, Body(\overrightarrow{t}))$$

If $c \in Succ(P, Body(\overrightarrow{t}))$ then $P, c \vdash_s Body(\overrightarrow{t})$ is derivable. If Ω is its derivation tree then $P, c \vdash_s p(\overrightarrow{t})$ is also derivable:

$$\frac{\Omega}{P, c \vdash_s p(\overrightarrow{t})}(def.)$$

Therefore:

$$Succ(P, p(\overrightarrow{t})) \supseteq Succ(P, Body(\overrightarrow{t}))$$

We conclude that $Succ(P, p(\overrightarrow{t})) = Succ(P, Body(\overrightarrow{t}))$.

3. $Succ(P, (E_1, E_2)) = Succ(P, E_1) \cap Succ(P, E_2)$.

If $c \in Succ(P, (E_1, E_2))$ then $P, c \vdash_s (E_1, E_2)$ is derivable. The only rule at the root of the derivation tree can be $(,)$:

$$\frac{P, c \vdash_s E_1 \quad P, c \vdash_s E_2}{P, c \vdash_s (E_1, E_2)}(,)$$

Hence $P, c \vdash_s E_1$ and $P, c \vdash_s E_2$ are derivable therefore $c \in Succ(P, E_1)$ and $c \in Succ(P, E_2)$ so

$$Succ(P, (E_1, E_2)) \subseteq Succ(P, E_1) \cap Succ(P, E_2)$$

If $c \in Succ(P, E_1) \cap Succ(P, E_2)$ then $c \in Succ(P, E_1)$ and $c \in Succ(P, E_2)$ so $P, c \vdash_s E_1$ and $P, c \vdash_s E_2$ are derivable. If Ω_1 and Ω_2 are the derivation trees than we can also derive $P, c \vdash_s (E_1, E_2)$:

$$\frac{\Omega_1 \quad \Omega_2}{P, c \vdash_s (E_1, E_2)}(,)$$

Therefore

$$Succ(P, (E_1, E_2)) \supseteq Succ(P, E_1) \cap Succ(P, E_2)$$

and we conclude that $Succ(P, (E_1, E_2)) = Succ(P, E_1) \cap Succ(P, E_2)$.

4. $Succ(P, (E_1; E_2)) = Succ(P, E_1) \cup Succ(P, E_2)$.

If $c \in Succ(P, (E_1; E_2))$ then $P, c \vdash_s (E_1; E_2)$ is derivable. The only rule at the root of the derivation tree can be $(;1)$ or $(;2)$. The two cases being symmetrical, we consider only the first one:

$$\frac{P, c \vdash_s E_1}{P, c \vdash_s (E_1; E_2)}(;1)$$

Hence $P, c \vdash_s E_1$ is derivable therefore $c \in Succ(P, E_1)$ so

$$Succ(P, (E_1; E_2)) \subseteq Succ(P, E_1) \cup Succ(P, E_2)$$

If $c \in Succ(P, E_1) \cup Succ(P, E_2)$ then $c \in Succ(P, E_1)$ or $c \in Succ(P, E_2)$ so either $P, c \vdash_s E_1$ or $P, c \vdash_s E_2$ is derivable. Because of the symmetry we only need to consider the first case: if Ω_1 is the derivation tree then $P, c \vdash_s (E_1; E_2)$ is also derivable:

$$\frac{\Omega_1}{P, c \vdash_s (E_1; E_2)}(;1)$$

Therefore

$$Succ(P, (E_1; E_2)) \supseteq Succ(P, E_1) \cup Succ(P, E_2)$$

and we conclude that $Succ(P,(E_1;E_2)) = Succ(P,E_1) \cup Succ(P,E_2)$.

5. $Succ(P,(\exists X.E(X))) = \bigcup_{t \in Terms} Succ(P,E(t))$.

If $c \in Succ(P,(\exists X.E(X)))$ then $P, c \vdash_s \exists X.E(X)$ is derivable. The only rule at the root of the derivation tree can be $(\exists)$:

$$\frac{P, c \vdash_s E(t_0)}{P, c \vdash_s \exists X.E(X)}(\exists)$$

for some term t_0. Hence $P, c \vdash_s E(t_0)$ is derivable which implies $c \in Succ(P,E(t_0))$ and therefore

$$Succ(P,(\exists X.E(X))) \subseteq \bigcup_{t \in Terms} Succ(P,E(t))$$

If $c \in \bigcup_{t \in Terms} Succ(P,E(t))$ then there exists a term t_0 such that $c \in Succ(P,E(t_0))$ which implies that $P, c \vdash_s E(t_0)$ is derivable. If Ω is its derivation tree then $P, c \vdash_s \exists X.E(X)$ is also derivable:

$$\frac{\Omega}{P, c \vdash_s \exists X.E(X)}(\exists)$$

Therefore

$$Succ(P,(\exists X.E(X))) \supseteq \bigcup_{t \in Terms} Succ(P,E(t))$$

6. $Succ(P,(\forall X.E(X))) = \bigcap_{t \in Terms} Succ(P,E(t))$.

If $c \in Succ(P,(\forall X.E(X)))$ then $P, c \vdash_s \forall X.E(X)$ is derivable. The only rule at the root of the derivation tree can be $(\forall)$:

$$\frac{P, c \vdash_s E(Y)}{P, c \vdash_s \forall X.E(X)}(\forall)$$

with $Y \notin fv(c) \cup vars(E(X))$ hence $P, c \vdash_s E(Y)$ is derivable.

Lemma 2.1.1: If $Y \notin fv(c)$ and $P, c \vdash_s E(Y)$ is derivable then $\forall t_0 \in Terms$, $P, c \vdash_s E(t_0)$ is derivable.

Proof: by induction over the structure of the derivation tree of $P, c \vdash_s E(Y)$.

Some of the cases are trivial. The non trivial ones are:

- $(tell)$:

$$\frac{}{P, c_1 \vdash_s tell(c_2)}(tell)$$

with $c_1 \vdash_D c_2$. By definition of $\vdash_D$ we also have $c_1\{Y/t_0\} \vdash_D c_2\{Y/t_0\}$. Since $Y \notin fv(c)$ we have $c_1\{Y/t_0\} = c_1$ therefore $c_1 \vdash_D c_2\{Y/t_0\}$ hence

$$\frac{}{P, c_1 \vdash_s tell(c_2\{Y/t_0\})}(tell)$$

- $(\exists)$:

$$\frac{P, c \vdash_s E_1(t)}{P, c \vdash_s \exists X.E_1(X)}(\exists)$$

So $E(Y) = \exists X.E_1(X)$. We have two cases:

a. $Y = X$ then $E(t_0) = (\exists X.E_1(X))\{X/t_0\} = \exists X.E_1(X) = E(Y)$. Since $P, c \vdash_s E(Y)$ is derivable, so is $P, c \vdash_s E(t_0)$.

b. $Y \neq X$ then $E(t_0) = (\exists X.E_1(X))\{Y/t_0\} = \exists X.(E_1(X)\{Y/t_0\})$ or $E(t_0) = \exists X.(E_1\{Y/t_0\})$ so we want to prove that $P, c \vdash_s \exists X.(E_1\{Y/t_0\})$ is derivable.

Since $Y \notin fv(c)$ we have by induction hypothesis that $P, c \vdash_s E_1(t)\{Y/t_0\}$ which is the same as $P, c \vdash_s E_1\{X/t\}\{Y/t_0\}$ is derivable.

If Ω is its derivation tree then we can put "on top" of Ω the following rule:

$$\frac{P, c \vdash_s E_1\{Y/t_0\}\{X/t\{Y/t_0\}\}}{P, c \vdash_s \exists X.(E_1\{Y/t_0\})}(\exists)$$

This is because $E_1\{Y/t_0\}\{X/t\{Y/t_0\}\} = E_1\{X/t\}\{Y/t_0\}$ and therefore we conclude that $P, c \vdash_s E(t_0)$ is derivable.

- (∀):

$$\frac{P, c \vdash_s E_1(Z)}{P, c \vdash_s \forall X.E_1(X)}(\forall)$$

with $Z \notin fv(c) \cup vars(E(X))$.

So $E(Y) = \forall X.E_1(X)$. We have two cases:

a. $Y = X$ then $E(t_0) = (\forall X.E_1(X))\{X/t_0\} = \forall X.E_1(X) = E(Y)$. Since $P, c \vdash_s E(Y)$ is derivable, so is $P, c \vdash_s E(t_0)$.

b. $Y \neq X$ then $E(t_0) = (\forall X.E_1(X))\{Y/t_0\} = \forall X.(E_1(X)\{Y/t_0\})$ or $E(t_0) = \forall X.(E_1\{Y/t_0\})$ so we want to prove that $P, c \vdash_s \forall X.(E_1\{Y/t_0\})$ is derivable.

Since $Y \notin fv(c)$ we have by induction hypothesis that $P, c \vdash_s E_1(t)\{Y/t_0\}$ which is the same as $P, c \vdash_s E_1\{X/Z\}\{Y/t_0\}$, is derivable.

If Ω is its derivation tree then we can put "on top" of Ω the following rule:

$$\frac{P, c \vdash_s E_1\{Y/t_0\}\{X/Z\{Y/t_0\}\}}{P, c \vdash_s \forall X.(E_1\{Y/t_0\})}(\forall)$$

This is because $E_1\{Y/t_0\}\{X/Z\{Y/t_0\}\} = E_1\{X/t\}\{Y/t_0\}$ and therefore we conclude that $P, c \vdash_s E(t_0)$ is derivable.

This concludes the proof of lemma 2.2.1.

Since $Y \notin fv(c)$ we deduce by lemma 2.1.1 that $\forall t_0 \in Terms$, $P, c \vdash_s E(Y)\{Y/t_0\}$is derivable. Since $Y \notin vars(E(X))$ we have that $E(Y)\{Y/t_0\} = E(X)\{X/Y\}\{Y/t_0\} = E(X)\{X/t_0\} = E(t_0)$ so $P, c \vdash_s E(t_0)$ is derivable.

Therefore

$$Succ(P, (\forall X.E(X))) \subseteq \bigcap_{t \in Terms} Succ(P, E(t))$$

If $c \in \bigcap_{t \in Terms} Succ(P, E(t))$ then $c \in Succ(P, E(Y))$ where Y is a variable such that $Y \notin fv(c) \cup vars(E(X))$, hence $P, c \vdash_s E(Y)$ is derivable. If its derivation tree is Ω, then $P, c \vdash_s \forall X.E(X)$ is also derivable:

$$\frac{\Omega}{P,\, c \vdash_s \forall X.E(X)}(\forall)$$

Therefore

$$Succ(P,(\forall X.E(X))) \supseteq \bigcap_{t \in Terms} Succ(P, E(t))$$

7. if $Succ(P,E_1) \subseteq Succ(P,E_2)$ then $\forall t \in Terms$, $Succ(P,E_1\{X/t\}) \subseteq Succ(P,E_2\{X/t\})$.

We will prove it by induction over the structure of the expressions E_1 and E_2.

Case $E_1 = tell(c_1)$, $E_2 = tell(c_2)$:

From the definition of $\vdash_D$ we have that $c_1 \vdash_D c_1$ so $c_1 \in Succ(P,E_1)$ hence $c_1 \in Succ(P,E_2)$ therefore $P,\, c_1 \vdash_s tell(c_2)$ is derivable. The only way to derive it is by the rule $(tell)$:

$$\frac{}{P,\, c_1 \vdash_s tell(c_2)}(tell)$$

We deduce that $c_1 \vdash_D c_2$ and from the definition of $\vdash_D$ we obtain $\forall t \in Terms$, $c_1\{X/t\} \vdash_D c_2\{X/t\}$. If $c \in Succ(P, tell(c_1\{X/t\}))$ then $c \notin False$ and $c \vdash_D c_1\{X/t\}$ by transitivity of $\vdash_D$ we have $c \vdash_D c_2\{X/t\}$and thefore $c \in Succ(P, c_2\{X/t\})$.

Hence $\forall t \in Terms$, $Succ(P,E_1\{X/t\}) \subseteq Succ(P,E_2\{X/t\})$.

The other cases are trivial.

8. if $Succ(P,E_1) \subseteq Succ(P,E_2)$ then $\forall \sigma \in Subst$, $Succ(P,\sigma E_1) \subseteq Succ(P,\sigma E_2)$.

Each finite substitution σ can be decomposed in a finite number of substitutions σ_i that change a single variable X_i into a term t_i:

$$\sigma = \sigma_n \circ ... \circ \sigma_1$$

Let $E_{1,i}$ be $\sigma_i...\sigma_1 E_1$ and $E_{2,i}$ be $\sigma_i...\sigma_1 E_2$ (with $E_{1,0} = E_1$ and $E_{2,0} = E_2$).

We use the previous property (lemma 2.1.7) n times: from $Succ(P, E_{1,i}) \subseteq Succ(P, E_{2,i})$ we deduce $Succ(P, E_{1,i+1}) = Succ(P, \sigma_{i+1}E_{1,i}) = Succ(P, E_{1,i}\{X_i/t_i\} \subseteq Succ(P, E_{2,i}\{X_i/t_i\} = Succ(P, \sigma_{i+1}E_{2,i}) = Succ(P, E_{2,i+1})$.

We conclude that $Succ(P, \sigma E_1) \subseteq Succ(P, \sigma E_2)$.

The finite success semantics of CLP∀ are conform to the semantics of CLP as presented in chapter 1 and [Mah99].

The syntax of CLP and CLP∀ differ slightly: in CLP there can be multiple clauses for a predicate unlike CLP∀. In CLP∀ the existential quantifier is explicitely written while in CLP it is implicit in the predicate definitions. In CLP the arguments of clause head can be terms while in CLP∀ they can only be variables.

Definition: The translation *TR1* between CLP and CLP∀ is a function that associates to each expression and program in CLP an expression or program in CLP∀. It is defined as follows:

•for an expression: $TR1(c) = tell(c)$ and $TR1(p(\overrightarrow{t})) = p(\overrightarrow{t})$

•for a program: for each definition of a predicate p of arity $m$whose clauses are:

$p(\overrightarrow{s_1}) : -Body_1.$

...

$p(\overrightarrow{s_n}) : -Body_n.$

we have a predicate p in the CLP∀ program whose definition is:

$p(\overrightarrow{X}) : -Body_{11}; ...; Body_{n1}$

where $\overrightarrow{X}$ is a vector of fresh variables of length m. $Body_{i1}$ is defined as $\exists \overrightarrow{Y}.(tell(s_{i1} = X_1), ..., tell(s_{im} = X_m), Body_{i2})$ where $\overrightarrow{Y}$ are all the free variables in $\overrightarrow{s_i}$ and $Body_i$ and $Body_{i2} = TR1(Body_i)$.

Proposition: If $< G|c > \rightarrow ... \rightarrow < \circ|d >, solv(d) \neq false, G' =$

$TR1(G)$, for all constraints c' we have

$$solv(c') = false \; iff \; c' \vdash_D false$$

and $\forall c_1, c_2 \in Constraints$, $c_1 \wedge c_2 \vdash_D c_1$, $c_1 \wedge c_2 \vdash_D c_2$ then

$$d \in Succ(TR1(P), (G', tell(c)))$$

Proof: We use an induction on the length n of the derivation of the answer c in CLP.

- $n = 1$:

$< G|c > \rightarrow < \circ|d >$ thefore $G = c_1$ and $d = c_1 \wedge c$. We also have $solv(d) \neq false$ hence $d \nvdash_D false$.

Thus:

$$d \in Succ(TR(P1), (tell(c_1), tell(c)))$$

because $c_1 \wedge c \vdash_D c_1$ and $c_1 \wedge c \vdash_D c$.

- $n > 1$:

$< G|c > \rightarrow < G_1|c_1 > \rightarrow^* < \circ|d >$ with $solv(d) \neq false$ and $d \in Succ(TR1(P1), (G_1', tell(c_1)))$ where $G_1' = TR1(G_1)$. Thus $d \in Succ(TR1(P1), G_1')$ and $d \in Succ(TR1(P1), tell(c_1))$ hence $d \vdash_D c_1$ and $d \nvdash_D false$.

We have the following cases:

-$G = (G_1, c_2)$ therefore the first transition is $< G_1, c_2|c > \rightarrow < G_1|c \wedge c_2 >$ hence $c_1 = c \wedge c_2$ and $solv(c_1) \neq false$. We have:

$$Succ(TR1(P1), ((G_1', tell(c_2)), tell(c))) = Succ(TR1(P1), ((G_1', (tell(c_2), tell(c)))))$$

Since $c_1 = c \wedge c_2$ and $d \vdash_D c_1$ we obtain by definition of $\vdash_D$ that $d \vdash_D c$ and $d \vdash_D c_1$ hence $d \in Succ(TR1(P1), (tell(c_2), tell(c)))$ and therefore $d \in Succ(TR1(P1), ((G_1', (tell(c_2), tell(c)))))$.

$-G = (G_2, p(\overrightarrow{t}))$ therefore the first derivation is $< G_2, p(\overrightarrow{t})|c > \rightarrow < G_2, s_1 = t_1, ..., s_k = t_k, B|c >$ where $p(s_0) : -B_0$ is the definition of p in the program $P1$ and $p(\overrightarrow{s}) : -B$ is the definition in which the free variables were renamed such that they are fresh.

We have that $G_1 = (G_2, s_1 = t_1, ..., s_k = t_k, B)$ and $c_1 = c$. By induction hypothesis, $d \in Succ(TR1(P1), (G_2', tell(s_1 = t_1), ..., tell(t_k = s_k), B'))$ where $G_2' = TR1(G_2)$ and $B' = TR1(B)$.

We have that

$$Succ(TR1(P1), (G_2', p(\overrightarrow{t}))) = Succ(TR1(P1), G_2) \cap Succ(TR1(P1), p(\overrightarrow{t}))$$

If the definition of p in the CLP∀ program is

$$p(\overrightarrow{X}) : -(\exists \overrightarrow{Y_0}.(tell(s_{01} = X_1), ..., tell(s_{0m} = X_m), B_0')); ...$$

where $\overrightarrow{Y_0}$ are all the free variables in $\overrightarrow{s_0}$ and $B_0' = TR1(B_0)$ then we obtain

$$Succ(TR1(P1), p(\overrightarrow{t})) \supseteq Succ(TR1(P1), (tell(s_1 = t_1), ..., tell(s_m = t_m), B'))$$

by substituting $\overrightarrow{Y_0}$ with the corresponding variables in $\overrightarrow{s}$ given by renaming mentioned above.

2.5 Infinite success semantics

In the previous section the notion of finite success was introduced. Its dual, presented in this section, is the infinite success.

A constraint c is a finite success of a goal E (in the context of a program P) iff there exists a derivation of

$$P, c \vdash_s E$$

In other words, a depth-first interpreter built from the rules of $\vdash_D$ calculus will find a derivation in finite time iff c is a finite success of E. If c is an

infinite success of E then the same interpreter may find a derivation or may not terminate. We define the infinite successes in a similar way to the finite ones by taking into account the finite and infinite derivations.

Finally we prove that the infinite successes have the same properties as the ones proved for finite successes.

In order to define the notion of infinite successes we will first define a sequent calculus of the form:

$$P, c \vdash_{is} E$$

which should be interpreted as "the constraint c is an infinite success of the expression (goal) E in the context of the program P". The subscript is stands for "infinite successes".

The rules of the sequent calculus are the following are identical to the ones of the sequent calculus $\vdash_s$ except for the rule $(trim)$:

$$\frac{}{P, c_1 \vdash_{is} tell(c_2)}(tell)$$

$(tell)$ having the side condition: $c_1 \vdash_D c_2$

$$\frac{P, c \vdash_{is} E_1 \quad P, c \vdash_{is} E_2}{P, c \vdash_{is} (E_1, E_2)}(,)$$

$$\frac{P, c \vdash_{is} E_1}{P, c \vdash_{is} (E_1; E_2)}(;1) \quad \frac{P, c \vdash_{is} E_2}{P, c \vdash_{is} (E_1; E_2)}(;2)$$

$$\frac{P, c \vdash_{is} Body(\vec{t})}{P, c \vdash_{is} p(\vec{t})}(def.)$$

where $p(\vec{X}) : -Body(\vec{X})$ is a clause in the program P

$$\frac{P, c \vdash_{is} E(t)}{P, c \vdash_{is} \exists X.E(X)}(\exists)$$

$$\frac{P, c \vdash_{is} E(Y)}{P, c \vdash_{is} \forall X.E(X)}(\forall)$$

$(\forall)$ having the side condition: $Y \notin fv(c) \cup vars(E(X))$.

$$\frac{}{P, c \vdash_{is} p(\overrightarrow{t})}(trim)$$

Definitions: A derivation tree in the system $\vdash_{is}$ which contains no application of the rule $(trim)$ is called a *complete derivation* otherwise it is called an *incomplete derivation.*

A branch that ends with an an application of the rule $(trim)$ is called an *incomplete branch.*

An *extendable branch* is an incomplete branch such that its $(trim)$ leaf can be replaced by a derivation of height greater than 1. An *extendable derivation* is an incomplete derivation such that all its incomplete branches are extendable. An *infinitely extendable derivation* is an extendable derivation such that the derivation resulting after the replacement of all $(trim)$ leafs by derivations whose root is not $(trim)$ is also infinitely extendable.

We will also call an infinitely extendable derivation an *infinite derivation.*

An *i-derivation* is a complete or infinitely extendable derivation.

Remark: A complete derivation in $\vdash_{is}$ is a also derivation in $\vdash_{s}$ and vice versa.

$ISucc : Progs \times Exp \rightarrow \wp(Constraints)$ is the function that gives the set of infinite successes of an expression E in the context of a program P:

$$ISucc(P, E) = \{c \in Constraints \mid (P, c \vdash_{is} E)\ has\ an\ i-derivation\} - False$$

$ISuccH : Progs \times Exp \rightarrow \wp(Constraints)$ is the function that gives the set of infinite core successes of an expression E in the context of a program P:

$$ISuccH(P,E) = \{c \in ISucc(P,E) \mid \forall d \in ISucc(P,E), c \vdash_D d \Rightarrow d \vdash_D c\}$$

Remark: The notion of infinite success is related to the notion of non-terminating computations: an expression (goal) can have infinite successes even if its execution doesn't terminate. Useful non-terminating computations appear in reactive programs which are programs that interact with an external environment. Examples of such programs are operating systems and programs that control or monitor external devices.

ISucc parallels the function *Succ* for infinite successes. Moreover all the properties of *Succ* given in proposition 2.1 hold also for *ISucc* as shown by the following:

Proposition 2.2: For each CLP∀ program P, predicate $p(\vec{X})$ defined in P by $p(\vec{X}) : -Body(\vec{X})$, expressions E, E_1, E_2 and term t the following properties hold:

1. $ISucc(P, tell(c)) = \{d \in Constraints \mid d \vdash_D c\} - False.$
2. $ISucc(P, p(\vec{t})) = ISucc(P, Body(\vec{t}))$.
3. $ISucc(P, (E_1, E_2)) = ISucc(P, E_1) \cap ISucc(P, E_2)$.
4. $ISucc(P, (E_1; E_2)) = ISucc(P, E_1) \cup ISucc(P, E_2)$.
5. $ISucc(P, (\exists X.E(X))) = \bigcup_{t \in Terms} ISucc(P, E(t))$.
6. $ISucc(P, (\forall X.E(X))) = \bigcap_{t \in Terms} ISucc(P, E(t))$.
7. if $ISucc(P, E_1) \subseteq ISucc(P, E_2)$ then $\forall t \in Terms$, $ISucc(P, E_1\{X/t\}) \subseteq ISucc(P, E_2\{X/t\})$.
8. if $ISucc(P, E_1) \subseteq ISucc(P, E_2)$ then $\forall \sigma \in Subst$, $ISucc(P, \sigma E_1) \subseteq ISucc(P, \sigma E_2)$.

Proof: analogous to the proof or proposition 2.1.

Example: the following program called *Lst* defines two non-terminating predicates *list*0 and *list*01:

```
list0(L):-
    ∃L1.(tell(L=[0 | L1]), list0(L1)).

list01(L):-
    ∃L1.((tell(L=[0 | L1]) ; tell(L=[1 | L1])), list01(L1)).
```

Intuitively $list0(L)$ "checks" that L is an infinite list composed only of zeroes and $list01(L)$ "checks" that L is composed either of zeroes or ones.

They can be seen as models of reactive processes that monitor the state of an external device (given here by the elements of the list L) and when the state has an invalid value they trigger an action (here they fail).

We would like to say that one successes of $list0(L)$ is the infinite list composed of 0. However we cannot explicitly write this success of $list0(L)$ i.e. we cannot write

$$L = [0, ...]$$

as there are no infinite terms in the formalism.

A possible solution is to add infinite terms to the formalism. This would result in a higher complexity which in our opinion is not worth.

Another solution is to have a special constraint system which contains the following:

- a term $infl0$ representing an infinite list of elements equal to 0
- a constraint $hl(L, H, T)$ whose is signification is: H, T are the head and the tail of L. The following holds for the constraint:

$$true \vdash_D hl(infl0, 0, infl0)$$

therefore $hl(L,0,infl0) \vdash_D hl(infl0,0,infl0)$ holds too.

We can derive

$$Lst, hl(L,0,infl0) \vdash_{is} list0(L)$$

by the following derivation (we omitted the name of the program Lst):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{hl(L,0,infl0) \vdash_{is} tell(hl(L,0,infl0))}(tell) \quad \Omega}{hl(L,0,infl0) \vdash_{is} tell(hl(L,0,infl0)), list0(infl0))}(,)}{hl(L,0,infl0) \vdash_{is} \exists L_1(tell(hl(L,0,L_1)), list0(L_1))}(\exists)}{hl(L,0,infl0) \vdash_{is} list0(L)}(def.)$$

where Ω is the subtree:

$$\dfrac{\dfrac{}{hl(L,0,infl0) \vdash_{is} tell(hl(infl0,0,infl0)}(tell) \quad \dfrac{\Theta}{hl(L,0,infl0) \vdash_{is} list0(infl0)}}{hl(L,0,infl0) \vdash_{is} list0(infl0)}(def.,\exists)$$

where Θ is the infinite branch.

Thus we have

$$hl(L,0,infl0) \in ISucc(Lst, list0(L))$$

but

$$hl(L,0,infl0) \notin Succ(Lst, list0(L))$$

Another solution is to "hide" the infinite terms and "expose" only parts of them. This means that the free variables in a goal are not unified to infinite terms but only to finite parts of them.

We can add to the previous program the following predicates:

```
list0(L,N,LR):-
    N=0,LR=[],list0(L)
 ;  N=s(N1),
    L=[0 | L1],
    list0(L1, N1, R1).

list01(L,N,LR):-
```

```
    N=0,LR=[],list01(L)
;   N=s(N1),
    (tell(L=[0 | L1]) ; tell(L=[1 | L1])),
    list01(L1, N1, LR).
```

For readability we use the Prolog convention of considering the free variables that appear in the body of a predicate but not in its head as existentially quantified.

The predicate $list0(L, N, LR)$ returns in LR the list composed by the first N elements of the infinite list L whose elements are all 0. The predicate $list01(L, N, LR)$ is similar.

We have that

$$SuccH(Lst, \exists L.list0(L, N, LR)) = \emptyset$$

because $list0(L, N, LR)$ does not terminate.

However

$$ISuccH(Lst, \exists L.list0(L, N, LR)) = \{(N = 0, LR = []), (N = s(0), LR = [0])..., \}$$

We also have

$$ISucc(Lst, \exists L.list0(L, N, LR)) \subseteq ISucc(Lst, \exists L.list01(L, N, LR))$$

In conclusion the notion of infinite successes extends the notion of finite successes by taking into account the infinite derivations. More about infinite successes and their relation with finite successes will be presented in chapter 4.

Chapter 3

An induction based proof system for finite success equivalence

In this chapter a formal system for proving success inclusion (and therefore success equivalence) is presented. The goal of the system is to enable one to prove that the set of finite successes of a CLP$\forall$ expression (goal) are included in the corresponding set of another expression. The proof system is based on the Gentzen calculus for the first order logic. We add an induction rule that allows proving the success inclusion for recursively defined predicates.

The first section introduces the increasing approximations of the predicates. Intuitively the n-th degree approximation corresponds to unfolding n times the original predicate. Its successes are the successes of the initial predicate that can be obtained by at most n nested calls. We prove that the set of successes of the initial predicate is the union of the set of successes of all its approximations. Thus the increasing approximations provide a way to eliminate recursion which will be used later in the chapter.

In the second section we prove that the set of successes is equal to the least fixed point of an operator on the set of programs.

The third section presents a proof system for the inclusion of finite suc-

cesses of CLP∀ expressions (goals) . The system contains rules from the Gentzen calculus, an induction rule and a rule for the constraint domain. We illustrate the system with an example.

In the fourth section we prove the correctness of the system when the induction rule is not used. We show with an example that when the induction is used the system is not correct without additional conditions.

The fifth section introduces a sufficient condition for the correctness of the full system. The correctness proof based on the notion of finite approximations is given.

The sixth section defines a weaker condition for the correctness of the proof system and gives the corresponding proof.

In the seventh section we show that other useful properties of programs like running time and memory consumption can be expressed with success inclusion and therefore proved using the system described in this chapter.

3.1 Increasing approximations of predicates

In this section the approximations of a predicate will be defined. They are nonrecursive predicates with increasing sets of successes that are included in the set of successes of the predicate.

After introducing their definition we prove in proposition 3.1 their main property, namely that each success of the predicate is a success of an approximation and vice versa. This property, together with the fact that their definitions are nonrecursive will be used later in the chapter.

For the purpose of the following definitions we need to extend the set of predicate names with the names of predicate approximations:

$PredsNAI$ denotes the extended set of predicate names which includes $PredsN$and for each predicate name $p \in PredsN$, $PredsNAI$ contains the predicate approximations names $p^n_{inc}, n \in \mathbb{N}$. We suppose that $p^n_{inc} \notin PredsN$.

Other definitions that depend on $PredsN$ will be extended accordingly, e.g.:

ExpAI, *ProgsAI* denotes the corresponding extended set of expressions and programs.

$SuccAI : ProgsAI \times ExpAI \rightarrow \wp(Constraints)$ denotes the extension of the function *Succ*.

The definition of the sequent calculus $\vdash_s$ must be extended to take into account the extended programs and expressions.

For convenience we will henceforth refer to the elements of these extended sets (e.g. *ExpAI*, *ProgsAI*) by the same names as the elements of the original sets (e.g. "expressions" and "programs").

Definition: Let P be a program, $p_1, ..., p_k$ be all the predicates defined in P and $q_1, ..., q_l$ be all the predicates that are called in the definitions of P.

$ApxInc(P, n)$ where $n \in \mathbb{N}$ is the program obtained by adding to P, for each predicate $p \in \{p_1, ..., p_k\}$ defined as $p(\overrightarrow{X}) : -Body(\overrightarrow{X})$ in P the following definitions (by definition of $PredsNAI$ p^i_{inc}, q^i_{inc}, $i \in \overline{0,n}$ do not appear in P):

$p^0_{inc} : -tell(false)$
$p^1_{inc}(\overrightarrow{X}) : -Body(\overrightarrow{X})\{q/q^0_{inc}\}.$
....
$p^n_{inc}(\overrightarrow{X}) : -Body(\overrightarrow{X})\{q/q^{n-1}_{inc}\}.$

Here $Body(\overrightarrow{X})\{q/q^i_{inc}\}$ denotes the expression $Body(\overrightarrow{X})$ where all the calls $q(\overrightarrow{t})$ have been replaced with calls $q^i_{inc}(\overrightarrow{t})$.

$Body^i_{inc}(\overrightarrow{X})$ will henceforth be another notation for $Body(\overrightarrow{X})\{q/q^i_{inc}\}$.

p^n_{inc}, $n \in \mathbb{N}$ are called *the increasing approximations of the predicate* p.

n is called *the degree of the approximation* p^n_{inc}.

Example: for the previous CLP∀ program NI we can construct $ApxInc(NI, n)$ by adding to NI the following definitions:

```
nat^0_inc(X):-tell(false).
nat^1_inc(X):-tell(X=0)
```

```
      ; ∃Y.(tell(X=s(Y)), nat⁰_inc(Y)).
...
int⁰_inc(X):-tell(false).
int¹_inc(X):-tell(X=0)
      ; ∃Y.(tell(X=s(Y)), int⁰_inc(Y))
      ; ∃Z.(tell(X=p(Z)), int⁰_inc(Z)).
...
```

We have that

$$SuccAI(ApxInc(NI, m), nat^0(X)) = \emptyset$$

We also have

$$SuccAI(ApxInc(NI, m), nat^n(X)) \supseteq \{X = s(0), ..., X = s(s(...s(0)...))\}$$

(the set of unary representations of numbers 0 to n) for all $m \geq n$.

It's also obvious that the union of successes of the finite approximations $nat^n_{inc}(X)$ is equal to the set of successes of $nat(X)$.

The generalization of this property is stated by the following proposition:

Proposition 3.1: For each program P and predicate p defined in P the following property holds:

$$Succ(P, p(\overrightarrow{t})) = \bigcup_{n \in \mathbb{N}} SuccAI(ApxInc(P, n), p^n_{inc}(\overrightarrow{t}))$$

Proof: If $c \in Succ(P, p(\overrightarrow{t}))$ then there exists a derivation of $P, c \vdash_s p(\overrightarrow{t})$. If the height of the derivation tree is h then we can build a derivation tree for $ApxInc(P, h), c \vdash_s p^h_{inc}(\overrightarrow{t})$ by replacing $p(\overrightarrow{t})$ with $p^h_{inc}(\overrightarrow{t})$ at the root of the tree and propagating the change towards the leaves.

When propagating through the (def) rule the definition of the approximation p^n_{inc} will be used instead of the definition of p:

$$\frac{P,\, c \vdash_s Body(\overrightarrow{t_1})}{P\, c \vdash_s p(\overrightarrow{t_1})}(def.)$$

becomes

$$\frac{ApxInc(P,h),\, c \vdash_s Body_{inc}^{n-1}(\overrightarrow{t_1})}{ApxInc(P,h),\, c \vdash_s p_{inc}^{n}(\overrightarrow{t_1})}(def.)$$

Therefore

$$Succ(P, p(\overrightarrow{t})) \subseteq \bigcup_{n \in \mathbf{N}} SuccAI(ApxInc(P,n), p_{inc}^{n}(\overrightarrow{t}))$$

If $c \in \bigcup_{n\in\mathbf{N}} SuccAI(ApxInc(P,n), p_{inc}^{n}(\overrightarrow{t}))$ then for some n there exists a derivation of $ApxInc(P,n), c \vdash_s p_{inc}^{n}(\overrightarrow{t})$. We can build a derivation for $P, c \vdash_s p(\overrightarrow{t})$ by replacing $p_{inc}^{n}(\overrightarrow{t})$ with $p(\overrightarrow{t})$ at the root of the tree and propagating the change towards the leaves.

When propagating through the (def) rule the definition of p will be used instead of the definition of the approximation p_{inc}^{m}: $m > 0$ hence

$$\frac{ApxInc(P,n),\, c \vdash_s Body_{inc}^{m-1}(\overrightarrow{t_1})}{ApxInc(P,n),\, c \vdash_s p_{inc}^{m}(\overrightarrow{t_1})}(def.)$$

becomes

$$\frac{P,\, c \vdash_s Body(\overrightarrow{t_1})}{P\, c \vdash_s p(\overrightarrow{t_1})}(def.)$$

We conclude that

$$Succ(P, p(\overrightarrow{t})) \supseteq \bigcup_{n \in \mathbf{N}} SuccAI(ApxInc(P,n), p_{inc}^{n}(\overrightarrow{t}))$$

E_{inc}^{n} denotes the expression E where each predicate call $p(\overrightarrow{t})$ has been replaced with $p_{inc}^{n}(\overrightarrow{t})$.

Corollary: For each program P and expression E the following property holds:

$$Succ(P,E) = \bigcup_{n\in\mathbb{N}} SuccAI(ApxInc(P,n), E^n_{inc})$$

The increasing approximations of a predicate p have nonrecursive definitions. Their sets of successes form an increasing sequence i.e.

$$SuccAI(ApxInc(P,n), p^m_{inc}(\overrightarrow{t})) \subseteq SuccAI(ApxInc(P,n), p^{m+1}_{inc}(\overrightarrow{t}))$$

where $m+1 \leq n$. The set of successes of an approximation $p^m_{inc}(\overrightarrow{t})$ is included in the set of successes of $p(\overrightarrow{t})$:

$$SuccAI(ApxInc(P,n), p^m_{inc}(\overrightarrow{t})) \subseteq Succ(P, p(\overrightarrow{t}))$$

where $m \leq n$. Therefore we can interpret $p^m_{inc}(\overrightarrow{t})$ as "approximating" $p(\overrightarrow{t})$ with respect to the set of successes, hence the name.

The fact that the approximations have nonrecursive definitions will be used for the proof of the theorem of soundness of the proof system presented later in the chapter . By proposition 3.1 in order to prove that the successes of a recursive predicate have a certain property it suffices to prove that the successes of its approximations have the property.

Remark: Another way of defining the increasing approximations of a predicate is:

$p^1_{inc}(\overrightarrow{X}) : -Body(\overrightarrow{X})\{q/q^0_{inc}\}.$
....
$p^n_{inc}(\overrightarrow{X}) : -Body(\overrightarrow{X})\{q/q^{n-1}_{inc}\}.$

In this case $p^0_{inc}(\overrightarrow{X})$ is not defined. The drawback is that we cannot define the decreasing approximations of a predicate (see section 4.1) in a similar way.

Yet another way of defining the increasing approximations of a predicate is:

$p^0_{inc}(\vec{X}) : -p^0_{inc}(\vec{X}).$
$p^1_{inc}(\vec{X}) : -Body(\vec{X})\{q/q^0_{inc}\}.$
....
$p^n_{inc}(\vec{X}) : -Body(\vec{X})\{q/q^{n-1}_{inc}\}.$

There is a definition for $p^0_{inc}(\vec{X})$ namely $p^0_{inc}(\vec{X}) : -p^0_{inc}(\vec{X})$. It's obvious that the property

$$Succ(P, p^0_{inc}(\vec{X})) = \emptyset$$

still holds as there can be no derivation of the for $P, c \vdash_s p^0_{inc}(\vec{X})$.

This definition of increasing approximation has the drawback that the definition of $p^0_{inc}(\vec{X})$ is recursive.

In the following sections we may drop the subscript inc therefore writing p^n instead of p^n_{inc} when there is no risk of confusion.

3.2 Least fixed point semantics of CLP∀

In this section we will introduce the operator IE and will prove that the set of finite successes of an expression is given by the least fixed point of IE. In section 4.2 we will prove that the greatest fixpoint of IE gives the set of infinite successes.

$IE : Progs \times (Exp \to \wp(Constraints)) \to (Exp \to \wp(Constraints))$ is an operator defined recursively as follows: if $A : Exp \to \wp(Constraints)$ and $p(\vec{X}) : -Body(\vec{X})$ is a predicate definition then

$IE(P, A)(tell(c)) = \{d \in Constraints \mid d \vdash_D c\} - False$
$IE(P, A)((E_1, E_2)) = IE(P, A)(E_1) \cap IE(P, A)(E_2)$

$IE(P,A)((E_1;E_2)) = IE(P,A)(E_1) \cup IE(P,A)(E_2)$
$IE(P,A)(\exists X.E_1(X)) = \bigcup_{t \in Terms} IE(P,A)(E_1(t))$
$IE(P,A)(\forall X.E_1(X)) = \bigcap_{t \in Terms} IE(P,A)(E_1(t))$
$IE(P,A)(p(\vec{t})) = IE'(P,A)(Body(\vec{t}))$
where IE' is an operator defined identically to IE except that
$IE'(P,A)(p(\vec{t})) = A(p(\vec{t}))$.

We will overcharge the function name $Succ$ such that it also denotes a function
$Succ : Progs \rightarrow (Exp \rightarrow \wp(Constraints))$ such that

$$Succ(P)(E) = Succ(P,E)$$

In other words we take the liberty to use $Succ$ as either unary or binary function.

Similarly, IE will be overcharged to a unary function:

$$IE(P)(A) = IE(P,A)$$

An important property of IE is given by the following proposition:

Proposition 3.2: For each CLP$\forall$ program P we have:

$$Succ(P) = \mu A.IE(P,A)$$

where μ is the well-known notation for the least fixed point.

Proof: First we prove that $Succ(P)$ is a fixed point of $IE(P)$ i.e.

$$IE(P, Succ(P)) = Succ(P)$$

or $\forall E\, IE(P, Succ(P))(E) = Succ(P)(E)$.

We proceed by induction over the structure of expression E. We have the following cases:

- $E = tell(c)$:

$IE(P, Succ(P))(tell(c)) = \{d \in Constraints \mid d \vdash_D c\} - False = Succ(P)(tell(c))$

- $E = (E_1, E_2)$:

$IE(P, Succ(P))((E_1, E_2)) = IE(P, Succ(P))(E_1) \cap IE(P, Succ(P))(E_2) =$
$Succ(P)(E_1) \cap Succ(P)(E_2) = Succ(P)((E_1, E_2))$

- $E = (E_1; E_2)$:

$IE(P, Succ(P))((E_1; E_2)) = IE(P, Succ(P))(E_1) \cup IE(P, Succ(P))(E_2) =$
$Succ(P)(E_1) \cup Succ(P)(E_2) = Succ(P)((E_1; E_2))$

- $E = \exists X.E(X)$:

$IE(P, Succ(P))(\exists X.E(X)) = \bigcup_{t \in Terms} IE(P, Succ(P))(E(t)) =$
$\bigcup_{t \in Terms} Succ(P)(E(t)) = Succ(P)(\exists X.E(X))$

- $E = \forall X.E(X)$:

$IE(P, Succ(P))(\forall X.E(X)) = \cap_{t \in Terms} IE(P, Succ(P))(E(t)) =$
$\cap_{t \in Terms} Succ(P)(E(t)) = Succ(P)(\forall X.E(X))$

- $E = p(\vec{t})$:

$IE(P, Succ(P))(p(\vec{t})) = IE'(P, Succ(P))(Body(\vec{t})) = Succ(P)(Body(\vec{t}))$
- the last equality can be proven by a similar induction over the structure of the expression $Body(\vec{t})$.

We then prove that $Succ(P)$ is the least fixed point of $IE(P)$ i.e. $\forall A : Exp \to \wp(Constraints)$ such that $IE(P, A) = A$ we have

$$\forall E \in Exp\, Succ(P)(E) \subseteq A(E)$$

Let $A_0 : Exp \to \wp(Constraints)$ be a function defined as $\forall E \in Exp$, $A_0(E) = \emptyset$. We have $IE(P)^0(A_0) = A_0 \subseteq A$ and by monotonicity of $IE(P)$ we derive

$$IE(P)^n(A_0) \subseteq IE(P)^n(A) = A$$

Therefore

$$\bigcup_{n \in \mathbf{N}} IE(P)^n(A_0) \subseteq A$$

We also have

$$IE(P)^n(A_0)(E) = SuccAI(ApxInc(P,n), E^n_{inc})$$

We deduce

$$\bigcup_{n \in \mathbf{N}} IE(P)^n(A_0) = SuccAI(ApxInc(P)) = Succ(P)$$

thus

$$Succ(P) = \bigcup_{n \in \mathbf{N}} IE(P)^n(A_0) \subseteq A$$

which concludes the proof.

3.3 An induction-based proof system

In this section we will present a proof system which allows one to prove that the successes of an expression (goal) E are included in those of an expression F in the context of a program P. After defining the system we will give an example of proof constructed in it.

In the next section we will prove its soundness with respect to the semantics of the CLP∀ language.

The goal that we set is to be able to prove formally that for a program P and two expressions E, F we have:

$$Succ(P, E) \subseteq Succ(P, F)$$

We will use for that purpose a proof system expressed by means of a sequent calculus of the form:

$$P, \Sigma \vdash_{si} E \rhd F$$

where Σ is a multiset of elements of the form $E_1 \rhd F_1$. They are called "hypotheses" and are denoted by the letter H (possibly subscripted). The subscript "si" stands for "success inclusion".

The meaning of the sequent above is "in the context of the program P and hypotheses Σ the successes of E are included in the successes of F".

The meaning of the hypothesis $E_1 \rhd F_1$ is "in the context of the program P the successes of E_1 are included in the successes of F_1".

The elements of the multiset Σ will separated by ",". Therefore $[(E_1 \rhd F_1), \Sigma]$ denotes the multiset composed by the hypotheses $E_1 \rhd F_1$ and the elements of the multiset Σ.

To make the rules clearer we will also use Γ, Δ as symbols for CLP$\forall$ expressions.

The proof system is given by the following rules:

$$\frac{}{P,\ \Sigma \vdash_{si} E, \Gamma \rhd E; \Delta}(id.)$$

if E is an atomic expression

$$\frac{P,\ \Sigma \vdash_{si} E_n, E_1, .., E_{n-1} \rhd \Delta}{P,\ \Sigma \vdash_{si} E_1, .., E_n \rhd \Delta}(comm.\ L) \qquad \frac{P,\ \Sigma \vdash_{si} \Gamma \rhd E_n; E_1; ..; E_{n-1}}{P,\ \Sigma \vdash_{si} \Gamma \rhd E_1; ..; E_n}(comm.\ R)$$

$$\frac{P,\ \Sigma \vdash_{si} E_1, E_2, \Gamma \rhd \Delta}{P,\ \Sigma \vdash_{si} (E_1, E_2), \Gamma \rhd \Delta}(,L) \qquad \frac{P,\ \Sigma \vdash_{si} \Gamma \rhd E_1; \Delta \quad P,\ \Sigma \vdash_{si} \Gamma \rhd E_2; \Delta}{P,\ \Sigma \vdash_{si} \Gamma \rhd (E_1, E_2); \Delta}(,R)$$

$$\frac{P, \Sigma \vdash_{si} E_1, \Gamma \triangleright \Delta \quad P, \Sigma \vdash_{si} E_2, \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} (E_1; E_2), \Gamma \triangleright \Delta}(;L) \quad \frac{P, \Sigma \vdash_{si} \Gamma \triangleright E_1; E_2; \Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright (E_1; E_2); \Delta}(;R)$$

$$\frac{}{P, \Sigma \vdash_{si} tell(c_1), .., tell(c_n), \Gamma \triangleright tell(d_1); ..; tell(d_m); \Delta}(tell)$$

if $c_1, .., c_n \vdash_{DM} d_1; ..; d_m$ and $n > 0$ or $m > 0$.

$$\frac{P, \Sigma \vdash_{si} E(Y), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} \exists X.E(X), \Gamma \triangleright \Delta}(\exists L) \quad \frac{P, \Sigma \vdash_{si} \Gamma \triangleright E(t); \Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright \exists X.E(X); \Delta}(\exists R)$$

$(\exists L)$ having the side condition: $Y \notin vars(E(X)) \cup fv(\Gamma) \cup fv(\Delta)$

$$\frac{P, \Sigma \vdash_{si} E(t), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} \forall X.E(X), \Gamma \triangleright \Delta}(\forall L) \quad \frac{P, \Sigma \vdash_{si} \Gamma \triangleright E(Y); \Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright \forall X.E(X); \Delta}(\forall R)$$

$(\forall R)$ having the side condition: $Y \notin vars(E(X)) \cup fv(\Gamma) \cup fv(\Delta)$

$$\frac{P, \Sigma \vdash_{si} Body(\overrightarrow{t}), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(def.L) \quad \frac{P, \Sigma \vdash_{si} \Gamma \triangleright Body(\overrightarrow{t}); \Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright p(\overrightarrow{t}); \Delta}(def.R)$$

$(def.L)$ and $(def.R)$ having the side condition: $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

$$\frac{P, \Sigma \vdash_{si} E' \triangleright F'}{P, \Sigma \vdash_{si} E \triangleright F}(gen.)$$

if $\exists \tau$ such that $\tau E' = E$, $\tau F' = F$.

$$\frac{P, [(F \triangleright G), \Sigma] \vdash_{si} E', \Gamma \triangleright \Delta}{P, [(F \triangleright G), \Sigma] \vdash_{si} E, \Gamma \triangleright \Delta}(hyp.L)$$

if $\exists \tau$ such that $\tau F = E$, $\tau G = E'$.

$$\frac{P, [((p(\overrightarrow{t}), \Gamma) \triangleright \Delta), \Sigma] \vdash_{si} Body(\overrightarrow{t}), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(ind.)$$

$(ind.)$ having the side condition: $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

Remark: The proof system is based on the Gentzen calculus for the first order logic. The rules for the conjunction "," and the disjunction ";" are identical to the rules for the logical connectors $\wedge$ and $\vee$. The rules for the quantifiers $\exists$ and $\forall$ are also identical to their counterparts in first order logic.

The CLP$\forall$ language is parametrized by the constraint domain and the proof system accounts for it by means of the rule (*tell*): it is this rule that captures the specifics of the constraint domain.

Finally there is the induction rule (*ind*) and the (*hyp. L*) rule that allow for proofs by induction to be constructed in the system.

Remark: Some other rules can be added to the proof system, for example weakening and duplication. However all sequents that can be derived with these rules can be derived without (the rules are admissible).

Another rule that can be added is

$$\frac{}{P,\ \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(nodef.\ L)$$

if the predicate p is not defined in P.

It allows one to prove that the successes of an undefined predicate are included in the successes of any expression. An alternate point of view is that one shouldn't be able to prove anything about an undefined predicate. In general a call to an undefined predicate is a programming error, which can be caught if this rule is absent.

Example: Lets recall the previous example of CLP$\forall$ program, *NI*:

```
nat(X):-X=0
      ; ∃Y.(X=s(Y), nat(Y)).

int(X):-X=0
      ; ∃Y.(X=s(Y), int(Y))
```

```
; ∃Z.(X=p(Z), int(Z)).
```

In the previously defined proof system we can prove that:

$$NI, [] \vdash_{si} nat(X) \triangleright int(X)$$

is derivable.

Here is the derivation tree:

$$\dfrac{\dfrac{\dfrac{}{NI, [H_{ni}] \vdash_{si} tell(X=0) \triangleright tell(X=0)}(def.R,\ id.)}{NI, [H_{ni}] \vdash_{si} tell(X=0) \triangleright int(X)} \quad \dfrac{\Omega}{NI, [H_{ni}] \vdash_{si} \exists Y.(...) \triangleright int(X)}(def.R)}{\dfrac{NI, [nat(X) \triangleright int(X)] \vdash_{si} tell(X=0); \exists Y.(...) \triangleright int(X)}{NI, [] \vdash_{si} nat(X) \triangleright int(X)}(ind.)}(;L)$$

where Ω is the following subtree (the subtree Θ was omitted):

$$\dfrac{\Theta \qquad \dfrac{}{NI, [H_{ni}] \vdash_{si} nat(X), ... \triangleright int(X); ...}(hyp.L,\ id.)}{NI, [H_{ni}] \vdash_{si} \exists Y.(tell(X=s(Y)), nat(Y)) \triangleright tell(X=0); \exists Y.(tell(X=s(Y)), int(Y)); ...}(...)$$

Here H_{ni} is the hypothesis $nat(X) \triangleright int(X)$.

In the next sections we will prove the soundness of the proof system. We will be able to deduce that:

$$Succ(NI, nat(X)) \subseteq Succ(NI, int(X))$$

3.4 Soundness of the proof system without induction

In this section we will prove the soundness of the proof system defined in the previous section when the induction rule is not used. In the next sections we will give sufficient conditions for the soundness of the system when the induction rule is used.

The purpose of proof system defined in the previous section is to allow one to prove that the successes of a an expression E are included into the successes of another expression F (in the context of a program P). This

requires that the proof system is sound with respect to the semantics of CLP∀.

First we will prove its soundness for all the proofs that do not use the induction rule (*ind*):

Proposition 3.4.1: If the sequent

$$P, \Sigma \vdash_{si} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ has a derivation where the rule (*ind*) doesn't appear and

$$Succ(P, E_i) \subseteq Succ(P, F_i), \forall i \in \overline{1..p}$$

then

$$Succ(P, E) \subseteq Succ(P, F)$$

Proof: We proceed by induction over the structure of the proof for the sequent $P, \Sigma \vdash_{si} E \triangleright F$. With respect to the rule at the root of the tree we have the following cases:

- (*id.*):

$$\frac{}{P, \Sigma \vdash_{si} E, \Gamma \triangleright E; \Delta}(id.)$$

By proposition 2.1 $Succ(P, (E, \Gamma)) = Succ(P, E) \cap Succ(P, \Gamma) \subseteq Succ(P, E) \cup Succ(P, \Delta) = Succ(P, (E; \Delta))$.

- (*comm. L*):

$$\frac{P, \Sigma \vdash_{si} E_n, E_1, .., E_{n-1} \triangleright \Delta}{P, \Sigma \vdash_{si} E_1, .., E_n \triangleright \Delta}(comm.\ L)$$

By proposition 2.1 and the induction hypothesis, $Succ(P,(E_1,..,E_n)) = \bigcap_{i\in\overline{1..n}} E_i = Succ(P,(E_n,E_1,..,E_{n-1})) \subseteq Succ(P,\Delta)$.

- $(comm.\,R)$:

$$\frac{P,\,\Sigma \vdash_{si} \Gamma \triangleright E_n, E_1, .., E_{n-1}}{P,\,\Sigma \vdash_{si} \Gamma \triangleright E_1; ..; E_n}(comm.\,R)$$

By proposition 2.1 and the induction hypothesis,
$Succ(P,\Gamma) \subseteq Succ(P,(E_n,E_1,..,E_{n-1})) = \bigcup_{i\in\overline{1..n}} E_i = Succ(P,(E_1;..;E_n))$.

- $(,L)$:

$$\frac{P,\,\Sigma \vdash_{si} E_1, E_2, \Gamma \triangleright \Delta}{P,\,\Sigma \vdash_{si} (E_1,E_2), \Gamma \triangleright \Delta}(,L)$$

By proposition 2.1, $Succ(P,((E_1,E_2),\Gamma)) = Succ(P,E_1) \cap Succ(P,E_2) \cap Succ(P,\Gamma) = Succ(P,(E_1,E_2,\Gamma))$ and by induction hypothesis, $Succ(P,(E_1,E_2,\Gamma)) \subseteq Succ(P,\Delta)$.

- $(,R)$:

$$\frac{P,\,\Sigma \vdash_{si} \Gamma \triangleright E_1; \Delta \quad P,\,\Sigma \vdash_{si} \Gamma \triangleright E_2; \Delta}{P,\,\Sigma \vdash_{si} \Gamma \triangleright (E_1,E_2); \Delta}(,R)$$

By proposition 2.1, $Succ(P,((E_1,E_2);\Delta)) = (Succ(P,E_1) \cap Succ(P,E_2)) \cup Succ(P,\Delta)$ and by induction hypothesis,
$Succ(P,\Gamma) \subseteq Succ(P,(E_1;\Delta)) = Succ(P,E_1) \cup Succ(P,\Delta)$ and
$Succ(P,\Gamma) \subseteq Succ(P,(E_2;\Delta)) = Succ(P,E_2) \cup Succ(P,\Delta)$
hence $Succ(P,\Gamma) \subseteq Succ(P,((E_1,E_2);\Delta))$.

- $(;L)$:

$$\frac{P,\,\Sigma \vdash_{si} E_1, \Gamma \triangleright \Delta \quad P,\,\Sigma \vdash_{si} E_2, \Gamma \triangleright \Delta}{P,\,\Sigma \vdash_{si} (E_1;E_2), \Gamma \triangleright \Delta}(;L)$$

By proposition 2.1, $Succ(P,((E_1;E_2),\Gamma)) = (Succ(P,E_1) \cup Succ(P,E_2)) \cap Succ(P,\Gamma)$ and by the induction hypothesis
$Succ(P,(E_1,\Gamma)) = Succ(P,E_1) \cap Succ(P,\Gamma) \subseteq Succ(P,\Delta)$ and
$Succ(P,(E_2,\Gamma)) = Succ(P,E_2) \cap Succ(P,\Gamma) \subseteq Succ(P,\Delta)$
hence $Succ(P,((E_1;E_2),\Gamma)) \subseteq Succ(P,\Delta)$.

- $(;R)$:

$$\frac{P,\ \Sigma \vdash_{si} \Gamma \triangleright E_1;E_2;\Delta}{P,\ \Sigma \vdash_{si} \Gamma \triangleright (E_1;E_2);\Delta}(;R)$$

By proposition 2.1, $Succ(P,((E_1;E_2);\Delta)) = Succ(P,E_1) \cup Succ(P,E_2) \cup Succ(P,\Delta) = Succ(P,(E_1;E_2;\Delta))$ and by the induction hypothesis, $Succ(P,\Gamma) \subseteq Succ(P,(E_1;E_2;\Delta))$.

- $(tell)$:

$$\frac{}{P,\ \Sigma \vdash_{si} tell(c_1),..,tell(c_n),\Gamma \triangleright tell(d_1);..;tell(d_m);\Delta}(tell)$$

if $c_1,..,c_n \vdash_{DM} d_1;..;d_m$.

If $n > 0$ and $m > 0$ then by proposition 2.1 we have
$Succ(P,(tell(c_1),..,tell(c_n),\Gamma)) = \bigcap_{i \in \overline{1..n}} Succ(P,tell(c_i)) \cap Succ(P,\Gamma)$ and
$Succ(P,(tell(d_1);..;tell(d_m);\Delta)) = \bigcup_{i \in \overline{1..m}} Succ(P,tell(d_i)) \cup Succ(P,\Delta)$

If $e \in Succ(P,(tell(c_1),..,tell(c_n),\Gamma))$ then $\forall i \in \overline{1..n}$, $e \vdash_D c_i$ and $e \nvdash_D false$. By definition of $\vdash_{DM}$ we deduce that $\exists j \in \overline{1..m}$ such that $e \vdash_D d_j$ hence $e \in Succ(P,tell(d_j))$ and therefore

$$Succ(P,(tell(c_1),..,tell(c_n),\Gamma)) \subseteq Succ(P,(tell(d_1);..;tell(d_m);\Delta))$$

If $m = 0$ then $n > 0$ and by definition of $\vdash_{DM}$ we have that $c_1,..,c_n \vdash_{DM} false$. If $d \in Succ(P,(tell(c_1),..,tell(c_n),\Gamma))$ then $\forall i \in \overline{1..n}$, $d \vdash_D c_i$ and $d \nvdash_D false$. By definition of $\vdash_{DM}$ we deduce that $d \vdash_D false$, contradiction. We deduce that d doesn't exist therefore

$$Succ(P,(tell(c_1),..,tell(c_n),\Gamma)) = \emptyset \subseteq Succ(P,(tell(d_1);..;tell(d_m);\Delta))$$

If $n = 0$ then $m > 0$ and by definition of $\vdash_{DM}$ we have that $true \vdash_{DM} d_1;..;d_m$. By definition of $\vdash_{DM}$ we deduce that $\forall d \in Constraints, d \in Succ(P,(tell(d_1);..;tell(d_m);\Delta))$, therefore

$$Succ(P,\Gamma) \subseteq Succ(P,(tell(d_1);..;tell(d_m);\Delta))$$

- $(\exists L)$:

$$\frac{P, \Sigma \vdash_{si} E(Y), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} \exists X.E(X), \Gamma \triangleright \Delta}(\exists L)$$

where $Y \notin vars(E(X)) \cup fv(\Gamma) \cup fv(\Delta)$.

By induction hypothesis, $Succ(P,(E(Y),\Gamma)) \subseteq Succ(P,\Delta)$ or $Succ(P,(E\{X/Y\},\Gamma)) \subseteq Succ(P,\Delta)$ hence, by proposition 2.1 prop. 7, we obtain that $\forall t \in Terms$, $Succ(P,(E\{X/Y\},\Gamma)\{Y/t\}) \subseteq Succ(P,\Delta\{Y/t\})$. Since $Y \notin fv(\Gamma) \cup fv(\Delta)$ we have that $\forall t \in Terms$, $Succ(P,(E\{X/Y\}\{Y/t\},\Gamma)) \subseteq Succ(P,\Delta)$.

Since $Y \notin vars(E(X))$ we deduce that $E\{X/Y\}\{Y/t\} = E\{X/t\}$ hence $\forall t \in Terms$, $Succ(P,(E\{X/t\}\},\Gamma)) \subseteq Succ(P,\Delta)$ and therefore by proposition 2.1 prop. 7

$$Succ(P,(\exists X.E(X),\Gamma)) = \bigcup_{t \in Terms} Succ(P,(E(t),\Gamma)) \subseteq Succ(P,\Delta)$$

- $(\exists R)$:

$$\frac{P, \Sigma \vdash_{si} \Gamma \triangleright E(t);\Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright \exists X.E(X);\Delta}(\exists R)$$

By proposition 2.1 and the induction hypothesis, $Succ(P,\Gamma) \subseteq Succ(P,(E(t);\Delta)) \subseteq Succ(P,(\exists X.E(X);\Delta))$.

- $(\forall L)$:

$$\frac{P,\Sigma \vdash_{si} E(t),\Gamma \triangleright \Delta}{P,\Sigma \vdash_{si} \forall X.E(X),\Gamma \triangleright \Delta}(\forall L)$$

By proposition 2.1 and the induction hypothesis, $Succ(P,(\forall X.E(X),\Gamma)) \subseteq Succ(P,(E(t),\Gamma)) \subseteq Succ(\Delta)$.

- $(\forall R)$:

$$\frac{P,\Sigma \vdash_{si} \Gamma \triangleright E(Y);\Delta}{P,\Sigma \vdash_{si} \Gamma \triangleright \forall X.E(X);\Delta}(\forall R)$$

where $Y \notin vars(E(X)) \cup fv(\Gamma) \cup fv(\Delta)$.

By induction hypothesis, $Succ(P,\Gamma) \subseteq Succ(P,(E(Y);\Delta))$ or $Succ(P,\Gamma) \subseteq Succ(P,(E\{X/Y\};\Delta))$ hence, by proposition 2.1 prop. 7, we obtain that $\forall t \in Terms$, $Succ(P,\Gamma\{Y/t\}) \subseteq Succ(P,(E\{X/Y\};\Delta)\{Y/t\})$. Since $Y \notin fv(\Gamma) \cup fv(\Delta)$ we have that $\forall t \in Terms$, $Succ(P,\Gamma) \subseteq Succ(P,(E\{X/Y\}\{Y/t\};\Delta))$.

Since $Y \notin vars(E(X))$ we deduce that $E\{X/Y\}\{Y/t\} = E\{X/t\}$ hence $\forall t \in Terms$, $Succ(P,\Gamma) \subseteq Succ(P,(E\{X/t\};\Delta))$ and therefore by proposition 2.1 prop. 7

$$Succ(P,\Gamma) \subseteq \bigcap_{t \in Terms} Succ(P,(E\{X/t\};\Delta)) = Succ(P,(\forall X.E(X);\Delta))$$

- $(def.L)$:

$$\frac{P,\Sigma \vdash_{si} Body(\overrightarrow{t}),\Gamma \triangleright \Delta}{P,\Sigma \vdash_{si} p(\overrightarrow{t}),\Gamma \triangleright \Delta}(def.L)$$

where $(p(\overrightarrow{X}) :- Body(\overrightarrow{X})) \in P$.

By proposition 2.1 and the induction hypothesis, $Succ(P,(p(\overrightarrow{t}),\Gamma)) = Succ(P,(Body(\overrightarrow{t}),\Gamma)) \subseteq Succ(P.\Delta)$.

- $(def.R)$:

$$\frac{P, \Sigma \vdash_{si} \Gamma \triangleright Body(\overrightarrow{t}); \Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright p(\overrightarrow{t}); \Delta}(def.R)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

By proposition 2.1 and the induction hypothesis,
$Succ(P, \Gamma) \subseteq Succ(P, (Body(\overrightarrow{t}); \Delta)) = Succ(P, (p(\overrightarrow{t}); \Delta))$.

- $(gen.)$:

$$\frac{P, \Sigma \vdash_{si} E' \triangleright F'}{P, \Sigma \vdash_{si} E \triangleright F}(gen.)$$

if $\exists \tau$ such that $\tau E' = E$ and $\tau F' = F$.

By induction hypothesis we have $Succ(P, E') \subseteq Succ(P, F')$ hence, by proposition 2.1 prop. 8 we deduce that

$Succ(P, E,) = Succ(P, \tau E') \subseteq Succ(P, \tau F') = Succ(P, (F)$.

- $(hyp.L)$:

$$\frac{P, [(F \triangleright G), \Sigma] \vdash_{si} E', \Gamma \triangleright \Delta}{P, [(F \triangleright G), \Sigma] \vdash_{si} E, \Gamma \triangleright \Delta}(hyp.L)$$

if $\exists \tau$ such that $\tau F = E$ and $\tau G = E'$.

By hypothesis $Succ(P, F) \subseteq Succ(P, G)$ and by proposition 2.1 prop. 8 we deduce that $Succ(P, E) = Succ(P, \sigma F) \subseteq Succ(P, \sigma G) = Succ(P, E')$, therefore by induction hypothesis we deduce that $Succ(P, (E, \Gamma)) \subseteq Succ(P, \Delta)$.

Remark: the previous proof is still valid if the program P is an extended program as the proof doesn't depend on the assumption that P is a normal (non-extended) program.

This property will be used in the next sections.

The proof system is not sound in general i.e. when the induction rule is used. The following example illustrates this.

Example: It's easy to prove that $NI, [] \vdash_{si} nat(X) \triangleright nat(s(X))$ therefore $Succ(NI, nat(X)) \subseteq Succ(NI, nat(s(X)))$. Using $nat(X) \triangleright nat(s(X))$ as hypothesis we can prove

$$NI, [nat(X) \triangleright nat(s(X))] \vdash_{si} nat(X) \triangleright tell(X = 0)$$

by the following derivation tree:

$$\dfrac{\dfrac{\dfrac{}{NI, [H_{nn}, H_1] \vdash_{si} tell(X=0) \triangleright tell(X=0)}(tell) \quad \dfrac{\Omega}{NI, [H_{nn}] \vdash_{si} \exists Y.(...) \triangleright tell(X=0)}(\exists L)}{NI, [H_{nn}, H_1] \vdash_{si} tell(X=0); \exists Y.(...) \triangleright tell(X=0)}(;L)}{NI, [H_{nn}] \vdash_{si} nat(X) \triangleright tell(X=0)}(ind.)$$

where Ω is the following subtree:

$$\dfrac{\dfrac{}{NI, [H_{nn}, H_1] \vdash_{si} tell(X = s(Y)), s(Y) = 0 \triangleright tell(X=0)}(tell)}{NI, [H_{nn}, H_1] \vdash_{si} tell(X = s(Y)), nat(s(Y)) \triangleright tell(X=0)}(hyp.\ L)}{NI, [H_{nn}, H_1] \vdash_{si} tell(X = s(Y)), nat(Y) \triangleright tell(X=0)}(hyp.\ L)$$

Here H_{nn} is the initial hypothesis $nat(X) \triangleright nat(s(X))$ and H_1 is the induction hypothesis $nat(X) \triangleright tell(X = 0)$.

Obviously the proof is incorrect as $Succ(NI, nat(X))$ is not included in $Succ(NI, tell(X = 0))$. In the following sections we will present restrictions on the proofs that insure the correctness of the system.

3.5 Soundness of the induction rule

In this section we extend the result of the previous section by giving sufficient conditions for the soundness of the system when the induction rule is used.

The main result in this section is theorem 3.5.4 which states that by imposing the condition C0 (also defined in this section) on the proofs we obtain a sound proof system.

Definitions: For a rule (*ind*) of the form

$$\frac{P, [((p(\overrightarrow{t}), \Gamma) \triangleright \Delta), \Sigma] \vdash_{si} Body(\overrightarrow{t}), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(ind.)$$

with $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$ we say that the hypothesis $p(\overrightarrow{t}), \Gamma \triangleright \Delta$ is *generated* by the call $p(\overrightarrow{t})$.

In this case the hypothesis is called *induction hypothesis*. All hypotheses that are not induction hypotheses are called *initial hypotheses*. They where present in the root sequent of the derivation tree.

For a rule $(hyp.\ L)$ of the form

$$\frac{P, [(p(\overrightarrow{t}), F \triangleright G), \Sigma] \vdash_{si} E', \Gamma \triangleright \Delta}{P, [(p(\overrightarrow{t_1}), F \triangleright G), \Sigma] \vdash_{si} p(\overrightarrow{t_2}), E, \Gamma \triangleright \Delta}(hyp.\ L)$$

with $\tau(p(\overrightarrow{t_1}), F) = (p(\overrightarrow{t_2}), E)$ and $\tau G = E'$ where $p(\overrightarrow{t_1}), F \triangleright G$ is an induction hypothesis. If the call $p(\overrightarrow{t_1})$ generated the hypothesis we say that $p(\overrightarrow{t_2})$ is its *corresponding call* in the rule $(hyp.\ L)$.

Definition: We say that a sequent of the form

$$P, \Sigma \vdash_{si} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ *is correct* if $Succ(P, E) \subseteq Succ(P, F)$ whenever $Succ(P, E_i) \subseteq Succ(P, F_i)$, $\forall i \in \overline{1..p}$.

Definition: We say that an application of a rule in a derivation tree *is correct* if the lower sequent is correct whenever the higher sequents are correct.

Definitions: If $p_1(\overrightarrow{t_1}), p_2(\overrightarrow{t_2})$ are predicate calls in a proof tree we say that $p_2(\overrightarrow{t_2})$ *is derived from* $p_1(\overrightarrow{t_1})$ if one of the following conditions is true:

- $p_2(\overrightarrow{t_2})$ is the same as $p_1(\overrightarrow{t_1})$
- the rule where $p_1(\overrightarrow{t_1})$ appears is $(def.L)$ or (ind):

$$\frac{P, \Sigma' \vdash_{si} Body_1(\overrightarrow{t_1}), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} p_1(\overrightarrow{t_1}), \Gamma \triangleright \Delta}(def.\, L)\ or\ (ind.)$$

$Body_1(\overrightarrow{t_1})$ contains $p_3(\overrightarrow{t_3})$ and $p_2(\overrightarrow{t_2})$ is derived from $p_3(\overrightarrow{t_3})$.
In this case we say that the rule *is interposed between* $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$.

- the rule where $p_1(\overrightarrow{t_1})$ appears is $(def.\, R)$:

$$\frac{P, \Sigma \vdash_{si} \Gamma \triangleright Body_1(\overrightarrow{t_1}); \Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright p_1(\overrightarrow{t_1}); \Delta}(def.\, R)$$

$Body_1(\overrightarrow{t_1})$ contains $p_3(\overrightarrow{t_3})$ and $p_2(\overrightarrow{t_2})$ is derived from $p_3(\overrightarrow{t_3})$.
In this case we say that the rule *is interposed between* $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$.

- $p_1(\overrightarrow{t_1})$ appears in the lower sequent in a rule

$$\frac{P, \Sigma \vdash_{si} ...p_1(\overrightarrow{t_1})...}{P, \Sigma \vdash_{si} ...p_1(\overrightarrow{t_1})...}$$

which leaves $p_1(\overrightarrow{t_1})$ unchanged and $p_2(\overrightarrow{t_2})$ is derived from $p_1(\overrightarrow{t_1})$ which appears in the upper sequent.

In the case of the rule $(hyp.\, L)$

$$\frac{P, [(F \triangleright G), \Sigma] \vdash_{si} E', \Gamma \triangleright \Delta}{P, [(F \triangleright G), \Sigma] \vdash_{si} E, \Gamma \triangleright \Delta}(hyp.\, L)$$

with $\tau F = E$ and $\tau G = E'$, $p_1(\overrightarrow{t_1})$ is left unchanged by the rule if it appears in Γ or Δ but not in E.

If $p_2(\overrightarrow{t_2})$ is derived from $p_1(\overrightarrow{t_1})$ which is contained in E_1 then we also say that $p_2(\overrightarrow{t_2})$ *is derived from* E_1.

Remark: The previous definition captures the notion of a predicate call $p_2(\overrightarrow{t_2})$ which is derived from another $p_1(\overrightarrow{t_1})$ by unfolding of $p_1(\overrightarrow{t_1})$: either

- $p_2(\overrightarrow{t_2})$ is $p_1(\overrightarrow{t_1})$ or
- $p_2(\overrightarrow{t_2})$ is obtained by unfolding several times $p_1(\overrightarrow{t_1})$ by one of the rules $(def.L)$, $(def.R)$,(ind) or
- $p_2(\overrightarrow{t_2})$ appears lower in the proof tree and is left unchanged by the rules in between.

Definition: If

$$\frac{P,\ [(F \triangleright G), \Sigma] \vdash_{si} E', \Gamma \triangleright \Delta}{P,\ [(F \triangleright G), \Sigma] \vdash_{si} E, \Gamma \triangleright \Delta}(hyp.L)$$

with $\tau F = E$ and $\tau G = E'$ is an application of the rule $(hyp.L)$ in a derivation then we say that a predicate call $p(\overrightarrow{t})$ *is introduced by the application* if $p_1(\overrightarrow{t_1})$ appears in E' and $p(\overrightarrow{t})$ is derived from $p_1(\overrightarrow{t_1})$.

If a predicate is introduced by an application of the rule $(hyp.L)$ then we also say that the predicate call *is introduced by the rule* $(hyp.L)$.

Definition: If $p_1(\overrightarrow{t_1}), p_2(\overrightarrow{t_2})$ are predicate calls in a proof tree and $p_2(\overrightarrow{t_2})$ is derived from $p_1(\overrightarrow{t_1})$ then the *degree distance dd* between $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$ is a natural number defined by the following conditions:

- if $p_1(\overrightarrow{t_1})$, $p_2(\overrightarrow{t_2})$ are the same then it is 0
- if the rule where $p_1(\overrightarrow{t_1})$ appears is $(def.L)$ or (ind):

$$\frac{P,\ \Sigma' \vdash_{si} Body_1(\overrightarrow{t_1}), \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} p_1(\overrightarrow{t_1}), \Gamma \triangleright \Delta}(def.L)\ or\ (ind.)$$

$p_2(\overrightarrow{t_2})$ is derived from $p_3(\overrightarrow{t_3})$ which is contained in $Body_1(\overrightarrow{t_1})$ then dd is 1+ the degree distance between $p_3(\overrightarrow{t_3})$ and $p_2(\overrightarrow{t_2})$.

- the rule where $p_1(\overrightarrow{t_1})$ appears is $(def.R)$:

$$\frac{P, \Sigma \vdash_{si} \Gamma \rhd Body_1(\overrightarrow{t_1}), \Delta}{P, \Sigma \vdash_{si} \Gamma \rhd p_1(\overrightarrow{t_1}), \Delta}(def.\ R)$$

$p_2(\overrightarrow{t_2})$ is derived from $p_3(\overrightarrow{t_3})$ which is contained in $Body_1(\overrightarrow{t_1})$ then dd is 1+ the degree distance between $p_3(\overrightarrow{t_3})$ and $p_2(\overrightarrow{t_2})$.

- $p_1(\overrightarrow{t_1})$ appears in the lower sequent in a rule

$$\frac{P, \Sigma \vdash_{si} ...p_1(\overrightarrow{t_1})...}{P, \Sigma \vdash_{si} ...p_1(\overrightarrow{t_1})...}$$

which leaves $p_1(\overrightarrow{t_1})$ unchanged and $p_2(\overrightarrow{t_2})$ is derived from $p_1(\overrightarrow{t_1})$ in the upper sequent then dd is the degree distance between $p_1(\overrightarrow{t_1})$ in the upper sequent and $p_2(\overrightarrow{t_2})$.

We will also say *distance* instead of *degree distance.*

Remark: The degree distance between $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$ is equal to the number of unfolding rules $(def.\ L)$, $(def.\ R)$,(ind) interposed between $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$.

Definition: A derivation tree is said to *satisfy the condition C0* if for each application of the rule $(hyp.\ L)$ in the derivation,

$$\frac{P, [(p(\overrightarrow{t}), F \rhd G), \Sigma] \vdash_{si} G_1, \Gamma \rhd \Delta}{P, [(p(\overrightarrow{t}), F \rhd G), \Sigma] \vdash_{si} p(\overrightarrow{t_1}), F_1, \Gamma \rhd \Delta}(hyp.\ L)$$

where $\tau p(\overrightarrow{t}) = p(\overrightarrow{t_1}), \tau F = F_1, \tau G = G_1$ and $p(\overrightarrow{t}), F \rhd G$ is an induction hypothesis generated by $p(\overrightarrow{t})$, we have that $p(\overrightarrow{t_1})$ is derived from $p(\overrightarrow{t})$.

Proposition 3.5.1: If a derivation tree satisfies the condition C0 then with the notations of the previous definition the distance between the calls $p(\overrightarrow{t})$ and $p(\overrightarrow{t_1})$ is strictly positive.

Proof: As $p(\overrightarrow{t})$ and $p(\overrightarrow{t_1})$ belong to different rules, they are different. As $p(\overrightarrow{t_1})$ derives from $p(\overrightarrow{t})$ it's obvious that the distance is strictly positive.

Definition: Let $P, \Sigma_0 \vdash_{si} p_0(\overrightarrow{t_0}), E_0 \triangleright F_0$ be a sequent that has a derivation whose root rule is (ind):

$$\frac{P, [H_0, \Sigma_0] \vdash_{si} Body_0(\overrightarrow{t_0}), \Gamma_0 \triangleright \Delta_0}{P, \Sigma_0 \vdash_{si} p_0(\overrightarrow{t_0}), \Gamma_0 \triangleright \Delta_0}(ind)$$

where H_0 is $p_0(\overrightarrow{t_0}), \Gamma_0 \triangleright \Delta_0$ and $(p_0(\overrightarrow{X}) : -Body_0(\overrightarrow{X})) \in P$.

Let the sequent $P, \Sigma_f \vdash_{si} p_f(\overrightarrow{t_f}), E_f \triangleright F_f$ be the lower sequent of a $(hyp.\ L)$ rule in the derivation and $p_f(\overrightarrow{t_f})$ is derived from $p_0(\overrightarrow{t_0})$.

There are in general $m \in \mathbb{N}$ applications of the rule (ind) different from the root one:

$$\frac{P, [H_i, \Sigma_i] \vdash_{si} Body_i(\overrightarrow{t_i}), \Gamma_i \triangleright \Delta_i}{P, \Sigma_i \vdash_{si} p_i(\overrightarrow{t_i}), \Gamma_i \triangleright \Delta_i}(ind.)$$

with $i \in \overline{1..m}$ where H_i is $p_i(\overrightarrow{t_i}), \Gamma_i \triangleright \Delta_i$ and $(p_i(\overrightarrow{X}) : -Body_i(\overrightarrow{X})) \in P$. Let $d_i > 0$ be the distance between $p_0(\overrightarrow{t_0})$ and $p_i(\overrightarrow{t_i})$.

Let n be a natural number and Σ_{new} be a multiset of hypotheses:

$$\Sigma_{new} = \{(p_i^{k_i}(\overrightarrow{t_i}), \Gamma_i \triangleright \Delta_i) \mid i \in \overline{0..m}, k_i \in \overline{0..n - d_i - 1}\}$$

The *transformation T0* of the proof tree starts at the root and propagates towards the $(hyp.\ L)$ rule:

- replace $p_0(\overrightarrow{t_0})$ with $p_0^n(\overrightarrow{t_0})$
- replace P with $ApxInc(P, n)$
- replace $\Sigma_0$with Σ' where $\Sigma' = \Sigma_0 \cup \Sigma_{new}$

To avoid writing the transformation twice, as we define it we will also prove the following:

Proposition 3.5.2: The transformation T0 has the following property: if in the sequent $P, \Sigma \vdash_{si} E_1, \Gamma \triangleright \Delta, E_1$ is replaced with E_{11} then

- E_{11} is identical to E_1 except for a single call of the form $q^k(\overrightarrow{t})$ instead of $q(\overrightarrow{t})$
- if the initial tree satisfies the condition C0 the replacing rules are correct.

Proof: The changes propagate towards the leaves of the tree. The rules are modified as follows:

- $(;L)$:

$$\frac{P, \Sigma \vdash_{si} E_1, \Gamma \triangleright \Delta \quad P, \Sigma \vdash_{si} E_2, \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} (E_1; E_2), \Gamma \triangleright \Delta}(;L)$$

If a call $p(\overrightarrow{t})$ in Γ is replaced by $p^k(\overrightarrow{t})$ and $p_f(\overrightarrow{t_f})$ is derived from Γ from the upper left sequent then the rule becomes:

$$\frac{ApxInc(P,n), \Sigma' \vdash_{si} E_1, \Gamma_1 \triangleright \Delta \quad P, \Sigma \vdash_{si} E_2, \Gamma \triangleright \Delta}{ApxInc(P,n), \Sigma' \vdash_{si} (E_1; E_2), \Gamma_1 \triangleright \Delta}(;L\_a)$$

where Γ_1 is the result of replacement of $p(\overrightarrow{t})$ in Γ is by $p^k(\overrightarrow{t})$. It's obvious that $Succ(ApxInc(P,n), (E_2, \Gamma_1)) \subseteq Succ(ApxInc(P,n), (E_2, \Gamma)) = Succ(P, (E_2, \Gamma))$.

- $(def.L)$ or (ind):

$$\frac{P, \Sigma_2 \vdash_{si} Body(\overrightarrow{t}), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(def.L)\ or\ (ind)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.
If $p(\overrightarrow{t})$ is replaced by $p^k(\overrightarrow{t})$ with $k > 0$ the rule becomes:

$$\frac{ApxInc(P,n),\ \Sigma' \vdash_{si} Body_1(\overrightarrow{t}), \Gamma \rhd \Delta}{ApxInc(P,n),\ \Sigma' \vdash_{si} p^k(\overrightarrow{t}), \Gamma \rhd \Delta}(def.\ L)$$

where $Body_1(\overrightarrow{t})$is $Body(\overrightarrow{t})$ where the call $q(\overrightarrow{t_2})$ from which $p_f(\overrightarrow{t_f})$ is derived is replaced by $q^{k-1}(\overrightarrow{t_2})$.

It's obvious that $Succ(ApxInc(P,n), p^k(\overrightarrow{t})) \subseteq Succ(ApxInc(P,n), Body_1(\overrightarrow{t}))$ therefore $Succ(ApxInc(P,n), (p^k(\overrightarrow{t}), \Gamma)) \subseteq Succ(ApxInc(P,n), \Delta)$.

If $k = 0$ then $p^0(\overrightarrow{t})$ has no definition in $ApxInc(P,n)$ therefore the rest of the tree is replaced by the rule $(nodef.\ L)$.

- $(hyp.\ L)$:

$$\frac{P,\ [p(\overrightarrow{t}), F' \rhd G', \Sigma] \vdash_{si} G, \Gamma \rhd \Delta}{P,\ [p(\overrightarrow{t}), F' \rhd G', \Sigma] \vdash_{si} p(\overrightarrow{t}), F, \Gamma \rhd \Delta}(hyp.\ L)$$

where $\tau p(\overrightarrow{t}) = p(\overrightarrow{t}), \tau F' = F, \tau G' = G$.

If $p(\overrightarrow{t})$ is replaced by $p^k(\overrightarrow{t})$ then we have two cases:

1. the hypothesis $p(\overrightarrow{t}), F' \rhd G'$ is one of the induction hypotheses $p_i(\overrightarrow{t_i}), \Gamma_i \rhd \Delta_i$.

The rule becomes:

$$\frac{ApxInc(P,n),\ \Sigma' \vdash_{si} G, \Gamma \rhd \Delta}{ApxInc(P,n),\ \Sigma' \vdash_{si} p^k(\overrightarrow{t}), F, \Gamma \rhd \Delta}(hyp.\ L)$$

By proposition 3.5.1 the distance between the call $p_i(\overrightarrow{t_i})$ that generated the induction hypothesis and the call $p(\overrightarrow{t})$ is strictly positive. Therefore the hypothesis $p^k(\overrightarrow{t}), F' \rhd G'$ belongs to Σ' and the rule is correct.

The rule $(hyp.\ L)$ "loses the approximations" meaning that in the sequent above no predicate call is replaced by an approximation and therefore the propagation of replacements with approximations stops at this rule.

2. the hypothesis $p(\overrightarrow{t}), F' \rhd G'$ is not one of the induction hypotheses $p_i(\overrightarrow{t_i}), \Gamma_i \rhd \Delta_i$.

Therefore this rule $(hyp.\,L)$ must be below the rule $(hyp.\,L)$ with an induction hypothesis (which stops the propagation of replacements with approximations). But in this case the condition C0 is not true for the derivation tree, contradiction.

Therefore this case cannot arise.

- a rule that leaves unchanged the predicate call affected by the propagation:

$$\frac{P,\ \Sigma \vdash_{si} ...p(\overrightarrow{t})...}{P,\ \Sigma \vdash_{si} ...p(\overrightarrow{t})...}$$

If the lower $p(\overrightarrow{t})$ is replaced by $p^k(\overrightarrow{t})$ then the upper $p(\overrightarrow{t})$ is replaced by $p^k(\overrightarrow{t})$ also:

$$\frac{ApxInc(P,n),\ \Sigma' \vdash_{si} ...p^k(\overrightarrow{t})...}{ApxInc(P,n),\ \Sigma' \vdash_{si} ...p^k(\overrightarrow{t})...}$$

Remark: The transformation T0 proceeds by replacing a predicate call $p_0(\overrightarrow{t_0})$ contained in the root sequent of a derivation tree at the left of $\triangleright$ with $p_0^n(\overrightarrow{t_0})$ and propagates the change towards the lower sequent $P,\ \Sigma_f \vdash_{si} p_f(\overrightarrow{t_f}), E_f \triangleright F_f$ of a rule $(hyp.\,L)$.

The propagation is straightforward - the only interesting cases appear when $(def.\,L)$ or (ind) are applied to an approximation $p^k(\overrightarrow{t})$ introduced by the propagation. In this case the call $q(\overrightarrow{t'})$ from the body of $p(\overrightarrow{t'})$ from which $p_f(\overrightarrow{t_f})$ is derived will be replaced by $q^{k-1}(\overrightarrow{t'})$.

Remark: The first property in proposition 3.5.1 proves that the definition of T0 is well-founded.

Definition: *The transformation T0a* is very similar to T0 and can be applied to the tree resulting from T0 or T0a.

Using the notations of the transformation T0, T0a is applied also to the rules between the root of the tree and another rule $(hyp.\,L)$ whose lower

sequent is of the form $P, \Sigma_f \vdash_{si} p_f(\overrightarrow{t_f}), E_f \triangleright F_f$. Its n parameter is the same as the previous transformation.

T0a proceeds exactly as T0 except that it doesn't modify the rules that were modified by a previous transformation. Specifically it modifies branches above the rule $(;L2)$ that weren't modified by previous applications of T0 or T0a.

Proposition 3.5.3: If the initial tree satisfies the condition C0 the replacing rules of a transformation T0a are correct.

Proof: The proof is very similar to the proof of proposition 3.5.2.

Definition: Using the notations of the transformation T0 and T0a *the transformation T0b* is a sequence of transformations T0, T0a,T0a,...,T0a such that all rules $(hyp.\ L)$ that use one of the induction hypotheses $p_i(\overrightarrow{t_i}), \Gamma_i \triangleright \Delta_i$ with $i \in \overline{0..m}$ are replaced by a transformation T0 or T0a.

Remark: In the rule $(;L\_a)$ above the upper right sequent is $P, \Sigma \vdash_{si} E_2, \Gamma \triangleright \Delta$ where Σ contains one or more induction hypotheses among $p_i(\overrightarrow{t_i}), \Gamma_i \triangleright \Delta_i$ with $i \in \overline{0..m}$. The purpose of the transformations T0a is to replace them with hypotheses from Σ_{new} in all $(hyp.\ L)$ rules that uses them. Therefore the induction hypothesis would no longer be used anywhere in the derivation tree and could be eliminated.

The following theorem gives a sufficient condition for the soundness of the proof system when the induction rule is used.

Theorem 3.5.4: If the sequent

$$P, \Sigma \vdash_{si} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ is derivable, the derivation tree satisfies the condition C0 and

$$Succ(P, E_i) \subseteq Succ(P, F_i),\ \forall i \in \overline{1..p}$$

then

$$Succ(P, E) \subseteq Succ(P, F)$$

Proof: We will use an induction on the structure of the derivation tree.

If the rule at the root is not (*ind*) then by proposition 3.4.1 we obtain the desired result.

If the rule at the root is (*ind*) then with the notations used int the definition of the transformation T0 and T0a let the rule at the root be:

$$\frac{P,\ [H_0, \Sigma_0] \vdash_{si} Body_0(\overrightarrow{t_0}), \Gamma_0 \triangleright \Delta_0}{P,\ \Sigma_0 \vdash_{si} p_0(\overrightarrow{t_0}), \Gamma_0 \triangleright \Delta_0}(ind.)$$

where H_0 is $p_0(\overrightarrow{t_0}), \Gamma_0 \triangleright \Delta_0$ and $(p_0(\overrightarrow{X}) : -Body_0(\overrightarrow{X})) \in P$.

We will prove by induction on $n \in \mathbb{N}$ that

$$SuccAI(ApxInc(P, n), (p_0^n(\overrightarrow{t_0}), \Gamma_0)) \subseteq Succ(P, \Delta_0)$$

$$\forall n \in \mathbb{N}, \forall i \in \overline{0..m},\ n \geq d_i \Rightarrow (SuccAI(ApxInc(P, n), (p_i^{n-d_i}(\overrightarrow{t_i}), \Gamma_i)) \subseteq Succ(P, \Delta_i))$$

and therefore by proposition 3.1 we deduce

$$Succ(P, (p_0(\overrightarrow{t}), \Gamma_0)) \subseteq Succ(P, \Delta_0)$$

Case $n = 0$: we have that

$$SuccAI(ApxInc(P, n), (p_0^0(\overrightarrow{t_0}), \Gamma_0)) = \emptyset \subseteq Succ(P, \Delta)$$

Case $n > 0$: for all i such that $d_i = n$ we have that

$$SuccAI(ApxInc(P,n),(p_i^{d_i-n}(\overrightarrow{t_i}),\Gamma_i)) = \emptyset \subseteq Succ(P,\Delta_i)$$

For the rest we apply the transformation T0b to the root of the tree. We obtain a tree where none of the induction hypotheses $p_i(\overrightarrow{t_i}),\Gamma_i \triangleright \Delta_i$ with $i \in \overline{0..m}$are used in a $(hyp.\,L)$ rule - they were replaced by the hypotheses in Σ_{new}.

The tree we obtain has an equal or lesser height than the initial tree. By induction hypothesis we obtain

$$SuccAI(ApxInc(P,n),(p_i^{n-d_i}(\overrightarrow{t_i}),\Gamma_i)) \subseteq Succ(P,\Delta_i)$$

for all i such that $d_i < n$.

Remark: The main idea of the proof is to replace an (ind) rule which introduces an induction hypothesis $p_i(\overrightarrow{t_i}),\Gamma_i \triangleright \Delta_i$with a $(def.)$ rule and to add as hypothesis $p_i^{k_i}(\overrightarrow{t_i}),\Gamma_i \triangleright \Delta_i$

for $k_i \in \overline{0..n-1}$. By applying the transformation T0b the applications of the rule $(hyp.\,L)$ that use the induction hypothesis are replaced by others that use the added hypothesis. Thus we obtain a proof tree for $p_i^n(\overrightarrow{t_i}),\Gamma_i \triangleright \Delta_i$.

Example: The incorrect proof of NI, $[nat(X) \triangleright nat(s(X))] \vdash_{si} nat(X) \triangleright tell(X=0)$ presented in the previous section does not satisfy the condition C0 as an application of rule $(hyp.\,L)$ with the initial hypothesis $nat(X) \triangleright nat(s(X))$ is interposed between the call that generates the induction hypothesis and its corresponding call in the application of rule $(hyp.\,L)$ with the induction hypothesis.

While the proof system is sound given the conditions in the previous theorem it is not complete. Moreover it can be shown that it cannot be complete by an argument using undecidability suggested by François Fages.

Suppose that we had a complete proof system for success inclusions of two

CLP∀ expressions E_1 and E_2 in the context of a program P. By enumerating all proofs we obtain an algorithm which terminates whenever $Succ(P, E_1) \subseteq Succ(P, E_2)$.

If $E_2 = tell(false)$ then $Succ(P, E_2) = \emptyset$. Suppose E_1 has a deterministic execution i.e. it has at most one success or doesn't terminate - it is sufficient for simulating a Turing machine. We can build a program that always terminates and can determine whether E_1 terminates. The program has two execution threads:

- one that executes the algorithm determining whether $Succ(P, E_1) \subseteq Succ(P, false) = \emptyset$.
- one that executes E_1

If E_1 terminates then the second thread finishes, hence the program can report that E_1 terminates. If E_1 doesn't terminate than it has no success therefore the first thread will terminate, hence the program can report that E_1 doesn't terminate.

As termination is undecidable, we have a contradiction, therefore we deduce that there can be no complete system for success inclusion.

3.6 A weaker condition for the soundness of the system

In the previous section a sufficient condition for the soundness of the proof system was given. In this section we will prove that a weaker condition is also sufficient for the soundness of the system.

We proceed by extending the definition of degree distance and then proving the same result as in the previous section for the extended definition.

Definition: Let P be a CLP∀ program. A hypothesis of the form $E_1 \triangleright E_2$ where $p_{11}(\overrightarrow{t_{11}}), ..., p_{1n}(\overrightarrow{t_{1n}})$ are the predicate calls contained in E_1 and $p_{21}(\overrightarrow{t_{21}}), ..., p_{2m}(\overrightarrow{t_{2m}})$ are the predicate calls contained in E_2 *satisfies the*

condition C1 in the context of the program P or simply *satisfies the condition C1* if there exists a set S of inequalities of the form

$$k_{1i} + c_{ij} \leq k_{2j}$$

with $c \in \mathbb{N}$, $i \in \overline{1..n}$, $j \in \overline{1..m}$ such that if the inequalities are satisfied then

$$Succ(Apx(P,l), E_1\{p_{1i}(\overrightarrow{t_{1i}})/p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})\}) \subseteq Succ(Apx(P,l), E_2\{p_{2j}(\overrightarrow{t_{2j}})/p_{2j}^{k_{2j}}(\overrightarrow{t_{2j}})\})$$

where $E_1\{p_{1i}(\overrightarrow{t_{1i}})/p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})\}$ denotes the expression E_1 where the calls $p_{1i}(\overrightarrow{t_{1i}})$, $i \in \overline{1..n}$ have been replaced with $p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})$. Here $l \geq k_{1i}, k_{2j}$ for all $i \in \overline{1..n}$, $j \in \overline{1..m}$.

Remark: For each pair (i,j) there exists in S at most one inequality of the form

$$k_{1i} + c_{ij} \leq k_{2j}$$

Proposition 3.6.1: Let

$$P, \Sigma \vdash_{si} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ be a sequent which has a derivation tree which doesn't use the rule (ind).

If each hypothesis in Σ satisfies the condition C1 then $E \triangleright F$ satisfies the condition C1.

Proof: We proceed by induction on the structure of the proof tree. The non-trivial cases for the rule at the root of the tree are:

- (id):

$$\frac{}{P,\ \Sigma \vdash_{si} E, \Gamma \triangleright E; \Delta}(id.)$$

If E is not a predicate call then $S = \emptyset$.

If E is a predicate call $p(\overrightarrow{t})$ then $Succ(P, p^{k_1}(\overrightarrow{t})) \subseteq Succ(P, p^{k_2}(\overrightarrow{t}))$ if $k_1 \leq k_2$ therefore $S = \{k_1 \leq k_2\}$

- $(, R)$:

$$\frac{P,\ \Sigma \vdash_{si} \Gamma \triangleright E_1; \Delta \quad P,\ \Sigma \vdash_{si} \Gamma \triangleright E_2; \Delta}{P,\ \Sigma \vdash_{si} \Gamma \triangleright (E_1, E_2); \Delta}(, R)$$

Let S_1, S_2 be the sets of inequalities for the upper left and upper right sequents and $S_3 = S_1 \cup S_2$.

If $p_{1i}(\overrightarrow{t_{1i}})$ is contained in Γ and $p_{2j}(\overrightarrow{t_{2j}})$ is contained in Δ then in S_3 there might be two inequalities $k_{1i} + c_1 \leq k_{2j}$ and $k_{1i} + c_2 \leq k_{2j}$.

If $c_1 < c_2$ then we eliminate the first one to obtain S otherwise we eliminate the second one.

- $(; L)$:

$$\frac{P,\ \Sigma \vdash_{si} E_1, \Gamma \triangleright \Delta \quad P,\ \Sigma \vdash_{si} E_2, \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} (E_1; E_2), \Gamma \triangleright \Delta}(; L)$$

Let S_1, S_2 be the sets of inequalities for the upper left and upper right sequents and $S_3 = S_1 \cup S_2$.

If $p_{1i}(\overrightarrow{t_{1i}})$ is contained in Γ and $p_{2j}(\overrightarrow{t_{2j}})$ is contained in Δ then in S_3 there might be two inequalities $k_{1i} + c_1 \leq k_{2j}$ and $k_{1i} + c_2 \leq k_{2j}$.

If $c_1 < c_2$ then we eliminate the first one to obtain S otherwise we eliminate the second one.

- $(def.L)$:

$$\frac{P,\ \Sigma \vdash_{si} Body(\overrightarrow{t}), \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(def.L)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

Let S_1 be the set of inequalities for the upper sequent. S is obtained from S_1 by replacing all inequalities of the form

$$k_{1i} + c_l \leq k_{2j}$$

where k_{1i} is the degree of a call contained in $Body(\overrightarrow{t})$ and $l \in \overline{1..n_1}$ with the inequality

$$k + max(1, c) - 1 \leq k_{2j}$$

where k is the degree of $p(\overrightarrow{t})$ and $c = max_{l \in \overline{1..n_1}}(c_l)$.

- $(def.R)$:

$$\frac{P, \Sigma \vdash_{si} \Gamma \triangleright Body(\overrightarrow{t}); \Delta}{P, \Sigma \vdash_{si} \Gamma \triangleright p(\overrightarrow{t}); \Delta}\,(def.R)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

Let S_1 be the set of inequalities for the upper sequent. S is obtained from S_1 by replacing all inequalities of the form

$$k_{1i} + c_l \leq k_{2j}$$

where k_{2j} is the degree of a call contained in $Body(\overrightarrow{t})$ and $l \in \overline{1..n_1}$ with the inequality

$$k_{1i} + c + 1 \leq k$$

where k is the degree of $p(\overrightarrow{t})$ and $c = max_{l \in \overline{1..n_1}}(c_l)$.

- $(hyp.L)$:

$$\frac{P, [(F \triangleright G), \Sigma] \vdash_{si} E', \Gamma \triangleright \Delta}{P, [(F \triangleright G), \Sigma] \vdash_{si} E, \Gamma \triangleright \Delta}(hyp.L)$$

if $\exists\tau$ such that $\tau F = E$ and $\tau G = E'$.

Let S_1 be the set of inequalities for the upper sequent. S_2 is obtained from S_1 by removing each inequality of the form

$$k_{1i} + c \leq k_{2j}$$

where k_{1i} is the degree of a call contained in E'.

Let $p_1(\overrightarrow{t_1})$ be contained in E (its degree is k_1), $p_1(\overrightarrow{t_2})$ is contained in F (its degree is k_2) such that $\tau p_1(\overrightarrow{t_2}) = p_1(\overrightarrow{t_1})$, $q_1(\overrightarrow{t_3})$ is contained in G (its degree is k_3), $q_1(\overrightarrow{t_4})$ is contained in E' (its degree is k_4) such that $\tau 1_1(\overrightarrow{t_3}) = q_1(\overrightarrow{t_4})$ and $q_2(\overrightarrow{t_5})$ is contained in Δ (its degree is k_5).

In S_1 there is the inequality $k_4 + c_1 \leq k_5$ and the hypothesis $F \triangleright G$ satisfies the inequality $k_2 + c_2 \leq k_3$.

By making $k_1 = k_2$ and $k_3 = k_4$ we obtain the inequality

$$k_1 + c_1 + c_2 \leq k_5$$

which will be added to S_2 to obtain S.

Remark: The condition C1 can be satisfied by sequents who have a derivation tree that use the rule (*ind*). For example in the context of the program NI presented before $nat(X) \triangleright int(X)$ satisfies the condition C1.

In general this is not true.

Example: In the following program called $NN2$:

```
nat(X):-X=0
      ; ∃Y.(X=s(Y), nat(Y)).

nat2(X):-X=0
      ; ∃Y.(X=s(Y), nat2(Y))
      ; ∃Z.(X=s(s(Z)), nat2(Z)).
```

we have two predicates $nat(X)$ and $nat2(X)$ which have the same set of

successes. The sequents $NN2, [] \vdash_{si} nat(X) \triangleright nat2(X)$ and $NN2, [] \vdash_{si} nat2(X) \triangleright nat(X)$ are derivable and the derivation trees satisfy the conditions of theorem 3.5.4.

However

$$nat2(X) \triangleright nat(X)$$

does not satisfy the condition C1. In fact for $l \geq k_1, k_2$

$$Succ(Apx(NN2, l), nat2^{k_1}(X)) \subseteq Succ(Apx(NN2, l), nat^{k_2}(X))$$

iff

$$2k_1 \leq k_2$$

The following definition extends the notion of degree distance to rules $(hyp.\ L)$ where the hypothesis used satisfies the condition C1:

Definition: If $p_1(\overrightarrow{t_1}), p_2(\overrightarrow{t_2})$ are predicate calls in a proof tree then the *degree distance dd*1 or just *distance dd*1 between $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$ is a natural number defined by the following conditions:

- if $p_2(\overrightarrow{t_2})$ is derived from $p_1(\overrightarrow{t_1})$ then it is equal to the distance dd between $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$
- if there exists a rule $(hyp.\ L)$

$$\frac{P, [p(\overrightarrow{t_3}), F' \triangleright q(\overrightarrow{t_4}), G'), \Sigma] \vdash_{si} q(\overrightarrow{t_6}), G, \Gamma \triangleright \Delta}{P, [(p(\overrightarrow{t_3}), F' \triangleright q(\overrightarrow{t_4}), G'), \Sigma] \vdash_{si} p(\overrightarrow{t_5}), F, \Gamma \triangleright \Delta}(hyp.\ L)$$

where $\tau(p(\overrightarrow{t_3}), F') = (p(\overrightarrow{t_5}), F)$, $\tau(q(\overrightarrow{t_4}), G') = (q(\overrightarrow{t_6}), G)$ such that the distance $dd1$ between $p_1(\overrightarrow{t_1})$ and $p(\overrightarrow{t_5})$ is d_1 and the distance $dd1$ between $q(\overrightarrow{t_6})$ and $p_2(\overrightarrow{t_2})$ is d_2, the hypothesis $p(\overrightarrow{t_3}), F' \triangleright q(\overrightarrow{t_4}), G'$ is an initial hy-

pothesis and satisfies the condition C1 and the inequality between k_3, the degree of $p_1(\overrightarrow{t_3})$, and k_4, the degree of $q(\overrightarrow{t_4})$ is

$$k_3 + c \leq k_4$$

then $dd1$ is $d_1 + d_2 - c$.
The rule is said to be *interposed between* $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$.

Definitions: The *transformation T1* is defined similarly to T0 between two calls $p_0(\overrightarrow{t_0})$ and $p_f(\overrightarrow{t_f})$ such that the distance $dd1$ between $p_0(\overrightarrow{t_0})$ and $p_f(\overrightarrow{t_f})$ is strictly positive. The only difference appears in the case of a rule $(hyp.\ L)$. With the notations of the previous definition if the rule is

$$\frac{P,\ [p(\overrightarrow{t_3}), F' \triangleright q(\overrightarrow{t_4}), G'), \Sigma] \vdash_{si} q(\overrightarrow{t_6}), G, \Gamma \triangleright \Delta}{P,\ [(p(\overrightarrow{t_3}), F' \triangleright q(\overrightarrow{t_4}), G'), \Sigma] \vdash_{si} p(\overrightarrow{t_5}), F, \Gamma \triangleright \Delta}(hyp.\ L)$$

and $p(\overrightarrow{t_5})$ is replaced with $p^k(\overrightarrow{t_5})$ then the rule becomes

$$\frac{ApxInc(P,n),\ \Sigma' \vdash_{si} q^{k_1}(\overrightarrow{t_6}), G, \Gamma \triangleright \Delta}{ApxInc(P,n),\ \Sigma' \vdash_{si} p^k(\overrightarrow{t_5}), F, \Gamma \triangleright \Delta}(hyp.\ L)$$

where

$$k_1 = k + c$$

The *transformations T1a* and *T1b* are defined similarly to T0a and T0b where T0 is replaced by T1.

Definition: Let P be a CLP$\forall$ program. Using the notations of the definition of condition C1, for a hypothesis of the form $E_1, \Gamma \triangleright E_2$ where $p_{11}(\overrightarrow{t_{11}}), ..., p_{1l}(\overrightarrow{t_{1l}})$ are the predicate calls contained in E_1 that satisfies the condition C1 we define S' as the subset of S that contains the inequalities

$$k_{1i} + c_{ij} \leq k_{2j}$$

with $c \in \mathbb{N}$, $i \in \overline{1..l}$, $j \in \overline{1..m}$.

We say that the hypothesis *satisfies the condition C1a in the context of the program P* or simply *satisfies the condition C1a* if whenever the inequalities from S_1 are satisfied we have

$$Succ(P, (E_1', \Gamma)) \subseteq Succ(P, E_2')$$

where $E_1' = E_1\{p_{1i}(\overrightarrow{t_{1i}})/p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})\}$ denotes the expression E_1 where the calls $p_{1i}(\overrightarrow{t_{1i}})$, $i \in \overline{1..l}$ have been replaced with $p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})$ and E_2' is E_2 where the calls $p_{2j}(\overrightarrow{t_{2j}})$ have been replaced with $p_{2j}^{k_{2j}}(\overrightarrow{t_{2j}})$ if k_{2j} appears in the inequalities contained in S'.

Proposition 3.6.2: Let

$$P, \Sigma \vdash_{si} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ be a sequent which has a derivation tree which doesn't use the rule (ind).

If each hypothesis in Σ satisfies the condition C1a then $E \triangleright F$ satisfies the condition C1a.

Proof: We use an induction over the structure of the derivation tree:

- (id):

$$\frac{}{P, \Sigma \vdash_{si} E_1, \Gamma \triangleright E_2}(id.)$$

If the common atomic E expression is contained in E_1 then $E_1 = (E, E_{11})$ and $E_2 = (E, E_{21})$.

We have two cases:

1. E is a predicate call $p(\overrightarrow{t})$ then

$$Succ(P, (E_1\{p(\overrightarrow{t})/p^{k_1}(\overrightarrow{t})\}, \Gamma)) \subseteq Succ(P, E_2\{p(\overrightarrow{t})/p^{k_2}(\overrightarrow{t})\})$$

because in S we have the inequality $k_1 \leq k_2$.

2. E is not a predicate call. Then in S we have no inequality.

If E is not contained in E_1 then $l = 0$ therefore

$$Succ(P, (E_1\{p_{1i}(\overrightarrow{t_{1i}})/p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})\}, \Gamma)) = Succ(P, (E_1, \Gamma)) \subseteq Succ(P, E_2) = Succ(P, E_2')$$

- $(, R)$:

$$\frac{P, \Sigma \vdash_{si} E_1, \Gamma \triangleright E_{21}; \Delta \quad P, \Sigma \vdash_{si} E_1, \Gamma \triangleright E_{22}; \Delta}{P, \Sigma \vdash_{si} E_1, \Gamma \triangleright (E_{21}, E_{22}); \Delta}(, R)$$

Let $S_1'S_2'$ be the sets of inequalities for the upper left and upper right sequents. Each inequality in those sets is implied by one from S'. By induction hypothesis we obtain the desired result.

- $(; L)$:

$$\frac{P, \Sigma \vdash_{si} E_{11}, \Gamma_1 \triangleright E_2 \quad P, \Sigma \vdash_{si} E_{12}, \Gamma_1 \triangleright E_2}{P, \Sigma \vdash_{si} (E_{11}; E_{12}), \Gamma_1 \triangleright E_2}(; L)$$

where $((E_{11}; E_{12}), \Gamma_1) = (E_1, \Gamma)$.

If $(E_{11}; E_{12})$ is contained in E_1 then Let S_1', S_2' be the sets of inequalities for the upper left and upper right sequents. Each inequality in those sets is implied by one from S'. By induction hypothesis we obtain the desired result.

Otherwise $(E_{11}; E_{12})$ is contained in Γ and therefore $l = 0$ hence $E_1 = E_1'$ and $E_2 = E_2'$.

- $(def. L)$:

$$\frac{P, \Sigma \vdash_{si} Body(\overrightarrow{t}), \Gamma_1 \triangleright E_2}{P, \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma_1 \triangleright E_2}(def. L)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$ and $(p(\overrightarrow{t}), \Gamma_1) = (E, \Gamma)$.

If $p(\overrightarrow{t})$ is contained in E_1 then each inequality from S'_1 - the set of inequalities for the upper sequent - is implied by one from S'. By induction hypothesis we obtain the desired result.

Otherwise $(E_{11}; E_{12})$ is contained in Γ and therefore $l = 0$ hence $E_1 = E'_1$ and $E_2 = E'_2$.

- $(def.R)$:

$$\frac{P, \Sigma \vdash_{si} E_1, \Gamma \triangleright Body(\overrightarrow{t}); \Delta}{P, \Sigma \vdash_{si} E_1, \Gamma \triangleright p(\overrightarrow{t}); \Delta}(def.R)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$ and $(p(\overrightarrow{t}); \Delta) = E_2$.

Each inequality from S'_1 - the set of inequalities for the upper sequent - is implied by one from S'. By induction hypothesis we obtain the desired result.

- $(hyp.L)$:

$$\frac{P, [(F \triangleright G), \Sigma] \vdash_{si} E'_{11}, E_{12}, \Gamma'_1, \Gamma_2 \triangleright E_2}{P, [(F \triangleright G), \Sigma] \vdash_{si} E_{11}, E_{12}, \Gamma_1, \Gamma_2 \triangleright E_2}(hyp.L)$$

where $E_1 = (E_{11}, E_{12})$, $\Gamma = (\Gamma_1, \Gamma_2)$ and $\tau F = (E_{11}, \Gamma_1)$, $\tau G = (E'_{11}, \Gamma'_1)$.

Each inequality from S'_1 - the set of inequalities for the upper sequent - is implied by one from S'. Since the hypothesis $F \triangleright G$ satisfies the condition C1a by induction hypothesis we obtain the desired result.

Definition: A derivation tree is said to satisfy the condition C1t if the following conditions are satisfied:

- the distance $dd1$ between any call that generates an induction hypothesis and the corresponding call in any rule $(hyp.L)$ that uses the induction hypothesis is strictly positive
- all the initial hypotheses used in a rule $(hyp.L)$ interposed between those calls satisfy the condition C1a

Proposition 3.6.3: If the initial tree satisfies the condition C1t the replacing rules of a transformation T1a are correct.

Proof: It is similar to the proof of proposition 3.5.3 except for the case of a rule $(hyp.\ L)$. With the notations of the definition of transformation T1 we have:

$$\frac{ApxInc(P,n),\ \Sigma' \vdash_{si} q^{k_1}(\overrightarrow{t_6}), G, \Gamma \rhd \Delta}{ApxInc(P,n),\ \Sigma' \vdash_{si} p^{k}(\overrightarrow{t_5}), F, \Gamma \rhd \Delta}(hyp.\ L)$$

where

$$k_1 = k + c$$

As the hypothesis $p(\overrightarrow{t_3}), F' \rhd q(\overrightarrow{t_4}), G'$ is an initial one it appears in Σ' and we have $k_3 + c \leq k_4$. Since the hypothesis satisfies the condition C1a we have by proposition 3.6.2

$$Succ(ApxInc(P,n), (p^k(\overrightarrow{t_3}), F')) \subseteq Succ(ApxInc(P,n), (q^{k+c}(\overrightarrow{t_4}), G'))$$

and since $k_1 = k + 1$ we obtain that

$$Succ(ApxInc(P,n), (p^k(\overrightarrow{t_5}), F)) \subseteq Succ(ApxInc(P,n), (q^{k_1}(\overrightarrow{t_6}), G))$$

Theorem 3.6.3: If the sequent

$$P, \Sigma \vdash_{si} E \rhd F$$

where $\Sigma = [E_1 \rhd F_1, .., E_p \rhd F_p]$ is derivable, the derivation tree satisfies the condition C1t and

$$Succ(P, E_i) \subseteq Succ(P, F_i),\ \forall i \in \overline{1..p}$$

then

$$Succ(P, E) \subseteq Succ(P, F)$$

Proof: The proof is similar to the proof of theorem 3.5.4 except that the transformation T1b will be used instead of the transformation T0b.

Example: The incorrect proof of

$$NI,\ [nat(X) \triangleright nat(s(X))] \vdash_{si} nat(X) \triangleright tell(X = 0)$$

presented in section 3.4 does not satisfy the condition C1t. The initial hypothesis $nat(X) \triangleright nat(s(X))$ satisfies the condition C1 in the context of the program NI:

$$Succ(Apx(NI, l), nat^{k_1}(X)) \subseteq Succ(Apx(NI, l), nat^{k_2}(s(X)))$$

if $l \geq k_1, k_2$ and $k_1 + 1 \leq k_2$.

However the distance $dd1$ between the call that generates the induction hypothesis and its corresponding one in the application of the induction hypothesis is 0 (instead of strictly positive).

3.7 Proving other properties through success inclusion

In this section we will show that other properties of logic programs can be reduced to success inclusion and therefore they can be proved using the proof system described previously.

Examples of useful properties that can be proved using these methods are the running time and complexity of programs, memory consumption, etc..

In the previous section of this chapter a formal system for proving success

inclusion was developed. While success inclusion in an important property it is not the only one of interest.

The same system can be used for proving other properties by expressing them as success inclusion. Two methods for achieving this will be presented.

The first method is to modify the code of the program such that during its execution it collects the data needed for checking the property which subsequently can be used by a predicate that checks the property.

Suppose that we want to prove that all executions of a predicate $p(\overrightarrow{t})$ verify some property *prop*. We modify the code of p and eventually the predicates that it calls such as to collect during execution the data needed for checking the property and return it in a vector of terms $\overrightarrow{t_1}$. Thus the new predicate call will be $p_1(\overrightarrow{t}, \overrightarrow{t_1})$.

We create a predicate $check\_prop(\overrightarrow{t_1})$ which succeeds when the property is satisfied for the data $\overrightarrow{t_1}$. Therefore for proving the property it is sufficient to prove that

$$p_1(\overrightarrow{t}, \overrightarrow{t_1}) \triangleright check\_prop(\overrightarrow{t_1})$$

Example: Consider the predicate $member(E, L)$ that succeeds when E is a member of the L. Its code is:

```
member(E, L):-
      tell(L=[E]),
    ;
      tell(L=[E1 | L1]),
      member(E, L1).
```

Suppose we want to prove that the number of unifications made during all its executions is smaller or equal to the number of elements of L. We modify the code of *member* by adding a parameter whose value will be the number the unifications:

```
member(E, L, No):-
      tell(L=[E]),
      No=1
    ;
      tell(L=[E1 | L1]),
      member(E, L1, No1),
      plus_nat(No1, 1, No).
```

where $plus\_nat(X, Y, Z)$ is a predicate which succeeds iff X, Y, Z are natural numbers and $Z = X + Y$.

We have only to prove that

$$member(E, L, No) \triangleright list\_len(L, Len), le\_nat(No, Len)$$

where $list\_len(L, Len)$ is a predicate that succeeds iff Len is the length of the list L and $le\_nat(X, Y)$ is a predicate that succeeds when X, Y are natural numbers and $X \leq Y$.

The method has the drawback that the code of the program must be modified and sometimes heavily so. Another method which avoids this uses a meta-interpreter of the CLP$\forall$ language. Instead of modifying the code of the predicate p and of the predicates it calls we can write a meta interpreter that provides the additional information in $\overrightarrow{t_1}$.

Suppose the interface of the meta-interpreter is the predicate $ev(Goal, Prog, \overrightarrow{Data})$ which succeeds iff $Goal$ succeeds in the context of the program $Prog$ and returns in $\overrightarrow{Data}$ the additional data needed for checking the property. $Goal$ and $Prog$ are encodings of the original goal and program into terms.

Then we only have to prove that

$$ev(p\_enc, prog\_enc, \overrightarrow{t_1}) \triangleright check\_prop(\overrightarrow{t_1})$$

where p_enc is the encoding of $p(\overrightarrow{t})$ and $prog\_enc$ is the encoding of the program that defines p.

The advantage of this method is that the original code need not be modified. However it must encoded into a term but this can be done automatically. The inconvenience is that the proof will be more complex because of the encoding and meta-interpretation.

These methods can be used for proving useful properties like the running time and complexity of programs, memory consumption, etc..

Chapter 4

A coinduction based proof system for infinite success equivalence

In this chapter we present a formal system for proving the inclusion (and therefore equivalence) of infinite successes. The system is dual to the one presented in the previous chapter in that it uses a rule for coinduction instead of induction.

The first section presents the decreasing approximations of predicates. They are similar to the increasing approximations in that they are obtained by unfolding 0,1,...,n,... times the initial definition of the predicate. We prove that the set of infinite successes of a predicate is the intersection of the set of successes of its decreasing approximations. We can therefore reason about the set of infinite successes of a recursive predicates by reasoning about the corresponding sets of its approximations whose definitions are not recursive. This property will be used later in the chapter.

The second section proves that the set of infinite successes is the greatest fixed point of the operator defined in the previous chapter. Thus the finite and infinite successes are the least and greatest fixed points of the same operator. This result reveals the duality between the finite and infinite successes which determines the duality between the proofs systems.

In the third section we prove the correctness of the proof system. We give two conditions for its correctness which are the counterparts to the conditions of correctness in the previous chapter. The proofs of correctness and the additional definitions are also symmetrical to those for the finite successes.

The fourth section defines first the negation of a constraint, expression and programs. Using this notion we prove that the two systems are equivalent. The proof is based on the left-right symmetry between the two systems.

4.1 Decreasing approximations of predicates

In this section the decreasing approximations of a predicate are presented. They are the duals of the increasing approximations presented in section 3.1. Their definitions are also nonrecursive but their sets of infinite successes are decreasing .

After introducing their definition we prove in proposition 4.1 their main property, namely that each success of a predicate is a success of all approximation and vice versa. This property, together with the fact that their definitions are nonrecursive will be used later in the chapter.

We need to extend the set of predicate names like we did for the increasing approximations:

$PredsNAD$ denotes the extended set of predicate names which includes $PredsN$and for each predicate name $p \in PredsN$, $PredsNAD$ contains the predicate approximations names $p^n_{dec}, n \in \mathbb{N}$. We suppose that $p^n_{dec} \notin PredsN$.

Other definitions that depend on $PredsN$ will be extended accordingly, e.g.:

$ExpAD$, $ProgsAD$ denotes the corresponding extended set of expressions and programs.

$SuccAD : ProgsAD \times ExpAD \rightarrow \wp(Constraints)$ denotes the extension of the function $Succ$.

The definition of the sequent calculus $\vdash_{is}$ must be extended to take into account the extended programs and expressions.

We will also refer to the elements of these extended sets (e.g. $ExpAD$, $ProgsAD$) by the same names as the elements of the original sets (e.g. "expressions" and "programs").

Definition: Let P be a program, $p_1, ..., p_k$ be all the predicates defined in P and $q_1, ..., q_l$ be all the predicates that are called in the definitions of P.

$ApxDec(P, n)$ where $n \in \mathbb{N}$ is the program obtained by adding to P, for each predicate $p \in \{p_1, ..., p_k\}$ defined as $p(\overrightarrow{X}) : -Body(\overrightarrow{X})$ in P the following definitions (by definition of $PredsNAD$ p^i_{dec}, q^i_{dec}, $i \in \overline{0,n}$ do not appear in P):

$p^0_{dec}(\overrightarrow{X}) : -tell(true).$
$p^1_{dec}(\overrightarrow{X}) : -Body(\overrightarrow{X})\{q/q^0_{dec}\}.$
....
$p^n_{dec}(\overrightarrow{X}) : -Body(\overrightarrow{X})\{q/q^{n-1}_{dec}\}.$

Here $Body(\overrightarrow{X})\{q/q^i_{dec}\}$ denotes the expression $Body(\overrightarrow{X})$ where all the calls $q(\overrightarrow{t})$ have been replaced with calls $q^i_{dec}(\overrightarrow{t})$.

$Body^i_{dec}(\overrightarrow{X})$ will henceforth be another notation for $Body(\overrightarrow{X})\{q/q^i_{dec}\}$.

p^n_{dec}, $n \in \mathbb{N}$ are called *the decreasing approximations of the predicate* p.

n is called *the degree of the approximation* p^n_{dec}.

The approximation of degree 0 of $p(\overrightarrow{X})$ has the following property:

$$ISucc(P, p^0_{dec}(\overrightarrow{X})) = Constraints - False$$

Like for increasing approximations we may also define the decreasing approximations of a predicate $p(\overrightarrow{X})$ as:

$p^0_{dec}(\overrightarrow{X}) : -p^0_{dec}(\overrightarrow{X}).$

$p^1_{dec}(\vec{X}) : -Body(\vec{X})\{q/q^0_{dec}\}.$

....

$p^n_{dec}(\vec{X}) : -Body(\vec{X})\{q/q^{n-1}_{dec}\}.$

This definition is identical for the two kinds of approximations.

Example: For the previous program *Lst* we can construct $ApxDec(Lst, n)$ by adding to *Lst* the following definitions:

```
list0^0_dec(L):-tell(true).
list0^1_dec(L):-
    ∃L1.(tell(L=[0 | L1]), list^0_dec0(L1)).
...
list01^0_dec(L):-tell(true).
list01^1_dec(L):-
    ∃L1.((tell(L=[0 | L1]) ; tell(L=[1 | L1])), list01^0_dec(L1)).
...
```

A similar property to the one stated in proposition 3.1 holds for the decreasing approximations:

Proposition 3.1: For each program P and predicate p defined in P the following property holds:

$$ISucc(P, p(\vec{t})) = \bigcap_{n \in \mathbf{N}} SuccAD(ApxDec(P, n), p^n_{dec}(\vec{t}))$$

Proof: The proof is similar to the one of proposition 3.1.

4.2 Greatest fixed point semantics of CLP∀

In this section we will prove a result that parallels the one in the section 3.2 namely that the set of infinite successes of an expression is given by the greatest fixed point of the operator IE.

Together, the two results expose the symmetry between the finite and infinite successes. This symmetry will appear further is section 4.4.

Similarly to *Succ* we will use the function name *ISucc* to also denote a function

$ISucc : Progs \rightarrow (Exp \rightarrow \wp(Constraints))$ such that:

$$ISucc(P)(E) = ISucc(P, E)$$

Thus *ISucc* can be used either as a unary or binary function.

Proposition 4.2: For each CLP∀ program P we have:

$$ISucc(P) = \nu A.IE(P, A)$$

where ν is the well-known notation for the greatest fixed point.

Proof: First we prove that $ISucc(P)$ is a fixed point of $IE(P)$ i.e.

$$IE(P, ISucc(P)) = ISucc(P)$$

or $\forall E\, IE(P, ISucc(P))(E) = ISucc(P)(E)$.

We proceed by induction over the structure of expression E. We have the following cases:

- $E = tell(c)$:

$IE(P, ISucc(P))(tell(c)) = \{d \in Constraints \mid d \vdash_D c\} - False = ISucc(P)(tell(c))$

- $E = (E_1, E_2)$:

$IE(P, ISucc(P))((E_1, E_2)) = IE(P, ISucc(P))(E_1) \cap IE(P, ISucc(P))(E_2) =$
$ISucc(P)(E_1) \cap ISucc(P)(E_2) = ISucc(P)((E_1, E_2))$

- $E = (E_1; E_2)$:

$IE(P, ISucc(P))((E_1; E_2)) = IE(P, ISucc(P))(E_1) \cup IE(P, ISucc(P))(E_2) = ISucc(P)(E_1) \cup ISucc(P)(E_2) = ISucc(P)((E_1; E_2))$

- $E = \exists X.E(X)$:

$IE(P, ISucc(P))(\exists X.E(X)) = \bigcup_{t \in Terms} IE(P, ISucc(P))(E(t)) = \bigcup_{t \in Terms} ISucc(P)(E(t)) = ISucc(P)(\exists X.E(X))$

- $E = \forall X.E(X)$:

$IE(P, ISucc(P))(\forall X.E(X)) = \cap_{t \in Terms} IE(P, ISucc(P))(E(t)) = \cap_{t \in Terms} ISucc(P)(E(t)) = ISucc(P)(\forall X.E(X))$

- $E = p(\overrightarrow{t})$:

$IE(P, ISucc(P))(p(\overrightarrow{t})) = IE'(P, ISucc(P))(Body(\overrightarrow{t})) = ISucc(P)(Body(\overrightarrow{t}))$ - the last equality can be proven by a similar induction over the structure of the expression $Body(\overrightarrow{t})$.

We then prove that $ISucc(P)$ is the greatest fixed point of $IE(P)$ i.e. $\forall A : Exp \to \wp(Constraints)$ such that $IE(P, A) = A$ we have

$$\forall E \in Exp\, ISucc(P)(E) \supseteq A(E)$$

Let $A_1 : Exp \to \wp(Constraints)$ be a function defined as $\forall E \in Exp$, $A_1(E) = Constraints$. We have $IE(P)^0(A_1) = A_1 \supseteq A$ and by monotonicity of $IE(P)$ we derive

$$IE(P)^n(A_1) \supseteq IE(P)^n(A) = A$$

Therefore

$$\bigcap_{n \in \mathbb{N}^*} IE(P)^n(A_1) \supseteq A$$

We also have for $\forall n \in \mathbb{N}^*$:

$$IE(P)^n(A_1)(E) = ISuccAD(ApxDec(P, n), E^n_{dec})$$

As $IE(P)^0(A_1) = A_1 = 1_{Constraints}$ we have

$$\bigcap_{n \in \mathbf{N}^*} IE(P)^n(A_1) = \bigcap_{n \in \mathbf{N}} IE(P)^n(A_1)$$

and therefore from

$$\bigcap_{n \in \mathbf{N}} IE(P)^n(A_1) = ISuccAD(ApxDec(P)) = ISucc(P)$$

we obtain

$$ISucc(P) = \bigcap_{n \in \mathbf{N}^*} IE(P)^n(A_1) \supseteq A$$

which concludes the proof.

4.3 A coinduction-based proof system

In this section we will present a proof system for inclusion of infinite successes which is dual to the one presented in chapter 3 for inclusion of finite successes. A coinduction rule replaces the induction rule of the previous system.

The condition for the soundness of the new system are dual to the conditions for the previous one. The proofs of soundness are also dual.

In the previous chapter we have given a formal system for proving the inclusion of finite successes. In this chapter we want to prove formally that for a program P and two expressions E, F we have:

$$ISucc(P, E) \subseteq ISucc(P, F)$$

We will use for that purpose a proof system expressed by means of a sequent calculus of the form:

$$P, \Sigma \vdash_{isi} E \triangleright F$$

where Σ is a multiset of elements of the form $E_1 \triangleright F_1$. They are called "hypotheses" and are denoted by the letter H (possibly subscripted). The subscript *isi* stands for "infinite success inclusion".

The meaning of the sequent above is "in the context of the program P and hypotheses Σ the infinite successes of E are included in the infinite successes of F".

The meaning of the hypothesis $E_1 \triangleright F_1$ is "in the context of the program P the infinite successes of E_1 are included in the infinite successes of F_1".

The elements of the multiset Σ will separated by ",". Therefore $[(E_1 \triangleright F_1), \Sigma]$ denotes the multiset composed by the hypotheses $E_1 \triangleright F_1$ and the elements of the multiset Σ.

The rules of the proof system are identical to the rules of the system $\vdash_{si}$ presented in the previous chapter except that the rules $(hyp.\ L)$ and (ind) are replaced by the following rules:

$$\frac{P,\ [(F \triangleright G), \Sigma] \vdash_{isi} \Gamma \triangleright E'; \Delta}{P,\ [(F \triangleright G), \Sigma] \vdash_{isi} \Gamma \triangleright E; \Delta}(hyp.\ R)$$

if $\exists \tau$ such that $\tau G = E$, $\tau F = E'$.

$$\frac{P,\ [(\Gamma \triangleright ((p(\overrightarrow{t}); \Delta)), \Sigma] \vdash_{isi} \Gamma \triangleright Body(\overrightarrow{t}); \Delta}{P,\ \Sigma \vdash_{isi} \Gamma \triangleright p(\overrightarrow{t}); \Delta}(coind.)$$

$(coind.)$ having the side condition: $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

Example: Recall the program *Lst* given in section 2.5:

```
list0(L):-
    ∃L1.(tell(L=[0 | L1]), list0(L1)).

list01(L):-
    ∃L1.((tell(L=[0 | L1]) ; tell(L=[1 | L1])), list01(L1)).
```

We can prove that

$$ISucc(Lst, list0(L)) \subseteq ISucc(Lst, list01(L))$$

$$Lst, [] \vdash_{isi} list0(L) \triangleright list01(L)$$

by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{Lst,\ [H] \vdash_{isi} tell(L = [0 \mid L2]), list0(L2) \triangleright tell(L = [0 \mid L2])}\ \Omega}{Lst,\ [list0(L) \triangleright list01(L)] \vdash_{isi} \exists L1.(...) \triangleright \exists L1.((tell(L = [0 \mid L1]); ...)}(\exists L, \exists R, ; R)}{Lst,\ [] \vdash_{isi} list0(L) \triangleright list01(L)}(coind.)$$

where H is the hypothesis $list0(L) \triangleright list01(L)$ and Ω if the following subtree:

$$\dfrac{}{Lst,\ [H] \vdash_{isi} tell(L = [0 \mid L2]), list0(L2) \triangleright tell(L = [0 \mid L2]), list01(L2)}(hyp.\ R,\ id)$$

The soundness of the proof system $\vdash_{isi}$ is obtained in a similar way to the soundness of the system $\vdash_{si}$, by imposing on the proofs addition conditions which are the duals of the conditions imposed for the system $\vdash_{si}$.

All the definitions and properties for the finite successes given in sections 3.5 and 3.6 have their counterpart for the infinite successes. They are obtained easily by replacing the rules $(hyp.\ L)$ and (ind) with the rules $(hyp.\ R)$ and $(coind)$ and by taking into account the left-right symmetry between the two proofs systems.

In the rest of this section we give some of the most important definitions and properties for the system $\vdash_{isi}$ which parallel those given for the system $\vdash_{si}$ in sections 3.5 and 3.6.

Definitions: If $p_1(\overrightarrow{t_1}), p_2(\overrightarrow{t_2})$ are predicate calls in a proof tree of the system $\vdash_{isi}$ we say that $p_2(\overrightarrow{t_2})$ *is R-derived from* $p_1(\overrightarrow{t_1})$ if one of the following conditions is true:

- $p_2(\overrightarrow{t_2})$ is the same as $p_1(\overrightarrow{t_1})$
- the rule where $p_1(\overrightarrow{t_1})$ appears is $(def.\ R)$ or $(coind)$:

$$\frac{P, \Sigma' \vdash_{isi} \Gamma \triangleright Body_1(\overrightarrow{t_1}); \Delta}{P, \Sigma \vdash_{isi} \Gamma \triangleright p_1(\overrightarrow{t_1}); \Delta}(def.\ R)\ or\ (coind.)$$

$Body_1(\overrightarrow{t_1})$ contains $p_3(\overrightarrow{t_3})$ and $p_2(\overrightarrow{t_2})$ is R-derived from $p_3(\overrightarrow{t_3})$.

In this case we say that the rule *is R-interposed between* $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$.

- the rule where $p_1(\overrightarrow{t_1})$ appears is $(def.\ L)$:

$$\frac{P, \Sigma \vdash_{isi} Body_1(\overrightarrow{t_1}), \Gamma \triangleright \Delta}{P, \Sigma \vdash_{isi} p_1(\overrightarrow{t_1}), \Gamma \triangleright \Delta}(def.\ L)$$

$Body_1(\overrightarrow{t_1})$ contains $p_3(\overrightarrow{t_3})$ and $p_2(\overrightarrow{t_2})$ is R-derived from $p_3(\overrightarrow{t_3})$.

In this case we say that the rule *is R-interposed between* $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$.

- $p_1(\overrightarrow{t_1})$ appears in the lower sequent in a rule

$$\frac{P, \Sigma \vdash_{isi} ...p_1(\overrightarrow{t_1})...}{P, \Sigma \vdash_{isi} ...p_1(\overrightarrow{t_1})...}$$

which leaves $p_1(\overrightarrow{t_1})$ unchanged and $p_2(\overrightarrow{t_2})$ is R-derived from $p_1(\overrightarrow{t_1})$ which appears in the upper sequent.

In the case of the rule $(hyp.\ R)$

$$\frac{P, [(F \triangleright G), \Sigma] \vdash_{isi} \Gamma \triangleright E'; \Delta}{P, [(F \triangleright G), \Sigma] \vdash_{isi} \Gamma \triangleright E; \Delta}(hyp.\ R)$$

with $\tau G = E$ and $\tau F = E'$, $p_1(\overrightarrow{t_1})$ is left unchanged by the rule if it appears in Γ or Δ but not in E.

If $p_2(\overrightarrow{t_2})$ is R-derived from $p_1(\overrightarrow{t_1})$ which is contained in E_1 then we also say that $p_2(\overrightarrow{t_2})$ *is R-derived from* E_1.

The previous definition are very similar to the ones given in section 3.5 for the notion of a predicate call being derived from another predicate call in a proof tree. We only replaced (*ind*) with (*coind*), (*def.L*) with (*def.R*) and (*hyp.L*) with (*hyp.R*).

To reduce duplication, for the following definitions and properties we will only specify the differences between them and their finite success correspondent.

Definition: If

$$\frac{P, [(F \triangleright G), \Sigma] \vdash_{isi} \Gamma \triangleright E'; \Delta}{P, [(F \triangleright G), \Sigma] \vdash_{isi} \Gamma \triangleright E; \Delta}(hyp.R)$$

with $\tau G = E$ and $\tau F = E'$ is an application of the rule (*hyp.R*) in a derivation of the proof system $\vdash_{isi}$ then we say that a predicate call $p(\overrightarrow{t})$ *is R-introduced by the application* if $p_1(\overrightarrow{t_1})$ appears in E' and $p(\overrightarrow{t})$ is R-derived from $p_1(\overrightarrow{t_1})$.

If a predicate call is R-introduced by an application of the rule (*hyp.R*) then we also say that the predicate call *is R-introduced by the rule* (*hyp.R*).

Definition: The *R-degree distance ddR* between two predicate calls $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$ in a derivation of the proof system $\vdash_{isi}$ is defined similarly to the notion of degree distance *dd* except that (*ind*) is replaced with (*coind*) and (*def.L*) with (*def.R*).

Definition: A derivation tree is said to *satisfy the condition C0R* if for each application of the rule (*hyp.R*) in the derivation,

$$\frac{P, [(p(\overrightarrow{t}), F \triangleright G), \Sigma] \vdash_{si} G_1, \Gamma \triangleright \Delta}{P, [(p(\overrightarrow{t}), F \triangleright G), \Sigma] \vdash_{si} p(\overrightarrow{t_1}), F_1, \Gamma \triangleright \Delta}(hyp.R)$$

where $\tau p(\overrightarrow{t}) = p(\overrightarrow{t_1}), \tau F = F_1, \tau G = G_1$ and $p(\overrightarrow{t}), F \triangleright G$ is an induction hypothesis generated by $p(\overrightarrow{t})$, we have that $p(\overrightarrow{t_1})$ is R-derived from $p(\overrightarrow{t})$.

Definition: We define the *transformations T0R, T0aR and T0bR* in

a similar way to the transformations T0, T0a and T0b except that we replace finite successes with infinite successes, increasing approximations with decreasing ones, (*ind*) with (*coind*), (*hyp. L*) with (*hyp. R*), etc..

The following theorem is the counterpart of theorem 3.5.4 for infinite successes:

Theorem 4.3.1: If the sequent

$$P, \Sigma \vdash_{isi} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ is derivable, the derivation tree satisfies the condition C0R and

$$ISucc(P, E_i) \subseteq ISucc(P, F_i),\ \forall i \in \overline{1..p}$$

then

$$ISucc(P, E) \subseteq ISucc(P, F)$$

Proof: The proof is similar to the proof of theorem 3.5.4. We use the transformation T0bR instead of T0b. The propositions 3.4.1 and 3.5.1-3.5.3 have their counterparts for infinite successes that have analogous proofs.

We can prove similar results to the ones in section 3.6 for infinite successes, namely that a weaker condition analogous to the condition C1t is sufficient for the soundness of the proof system $\vdash_{isi}$.

Definition: Let P be a CLP$\forall$ program. A hypothesis of the form $E_1 \triangleright E_2$ where $p_{11}(\overrightarrow{t_{11}}), ..., p_{1n}(\overrightarrow{t_{1n}})$ are the predicate calls contained in E_1 and $p_{21}(\overrightarrow{t_{21}}), ..., p_{2m}(\overrightarrow{t_{2m}})$ are the predicate calls contained in E_2 *satisfies the condition C1R in the context of the program P* or simply *satisfies the condition C1R* if there exists a set S of inequalities of the form

$$k_{1i} + c_{ij} \leq k_{2j}$$

with $c \in \mathbb{N}$, $i \in \overline{1..n}$, $j \in \overline{1..m}$ such that if the inequalities are satisfied then

$$ISucc(P, E_1\{p_{1i}(\overrightarrow{t_{1i}})/p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})\}) \subseteq ISucc(P, E_2\{p_{2j}(\overrightarrow{t_{2j}})/p_{2j}^{k_{2j}}(\overrightarrow{t_{2j}})\})$$

where $E_1\{p_{1i}(\overrightarrow{t_{1i}})/p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})\}$ denotes the expression E_1 where the calls $p_{1i}(\overrightarrow{t_{1i}})$, $i \in \overline{1..n}$ have been replaced with $p_{1i}^{k_{1i}}(\overrightarrow{t_{1i}})$.

Definition: If $p_1(\overrightarrow{t_1}), p_2(\overrightarrow{t_2})$ are predicate calls in a proof tree then the *degree distance* $dd1R$ or just *distance* $dd1R$ between $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$ is a natural number defined by the following conditions:

- if $p_2(\overrightarrow{t_2})$ is R-derived from $p_1(\overrightarrow{t_1})$ then it is equal to the distance ddR between $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$
- if there exists a rule $(hyp.\ R)$

$$\frac{P, [p(\overrightarrow{t_3}); F' \triangleright q(\overrightarrow{t_4}); G'), \Sigma] \vdash_{isi} \Gamma \triangleright q(\overrightarrow{t_6}); G; \Delta}{P, [(p(\overrightarrow{t_3}); F' \triangleright q(\overrightarrow{t_4}); G'), \Sigma] \vdash_{isi} \Gamma \triangleright p(\overrightarrow{t_5}); F; \Delta}(hyp.\ R)$$

where $\tau(p(\overrightarrow{t_3}); F') = (q(\overrightarrow{t_6}); G)$, $\tau(q(\overrightarrow{t_4}); G') = (p(\overrightarrow{t_5}); F)$ such that the distance $dd1R$ between $p_1(\overrightarrow{t_1})$ and $p(\overrightarrow{t_5})$ is d_1 and the distance $dd1R$ between $q(\overrightarrow{t_6})$ and $p_2(\overrightarrow{t_2})$ is d_2, the hypothesis $p(\overrightarrow{t_3}); F' \triangleright q(\overrightarrow{t_4}); G'$ is an initial hypothesis and satisfies the condition C1R and the inequality between k_3, the degree of $p_1(\overrightarrow{t_3})$, and k_4, the degree of $q(\overrightarrow{t_4})$ is

$$k_3 + c \leq k_4$$

then $dd1$ is $d_1 + d_2 + c$.
The rule is said to be *R-interposed between* $p_1(\overrightarrow{t_1})$ and $p_2(\overrightarrow{t_2})$.

Definitions: The conditions *C1aR, C1tR* and transformations *T1R, T1aR, T1bR*

are defined similarly to the conditions C1a, C1t and transformations T1,T1a,T1b by using the left-right symmetry of the proofs systems $\vdash_{si}$ and $\vdash_{isi}$.

Theorem 4.3.2: If the sequent

$$P, \Sigma \vdash_{isi} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ is derivable, the derivation tree satisfies the condition C1t and

$$ISucc(P, E_i) \subseteq ISucc(P, F_i),\ \forall i \in \overline{1..p}$$

then

$$ISucc(P, E) \subseteq ISucc(P, F)$$

Proof: The proof is similar to the proof of theorem 3.6.3 using the symmetry between the proof systems $\vdash_{si}$ and $\vdash_{isi}$.

The proof uses the counterparts for infinite successes of the properties stated in propositions 3.6.1-3.6.3 which hold also by symmetric proofs.

4.4 Equivalence of the two proof systems

In this section we begin by introducing the notion of negation of a constraint, then we define the negations of an expression and of a program.

Next, we show that the two proofs system for finite and infinite successes are equivalent in the sense that a proof for successes inclusion exists in one system if and only if a proof of inclusion exists in the other. We use the negation for obtaining the program and the expressions for the second proof.

Definition: we say that the relation $\vdash_D$ *has the negation property* if for each constraint c there exists a constraint $\overline{c}$ such that for all constraints $c_1, .., c_n, d_1, ..., d_m$ with $n, m \in \mathbb{N}$ we have

$$\overline{c_1}, .., \overline{c_n} \vdash_D \overline{d_1}; ..; \overline{d_m}$$

if and only if we have

$$d_1, .., d_m \vdash_D c_1; ..; c_n$$

Remark: The previous definition introduces the notion of negation of constraints. The negated constraints do not posses all the properties associated with negation but the condition imposed in the definition is sufficient for the main proof of this section (proposition 4.4.1).

Definitions: Let P be a CLP$\forall$ program and suppose the relation $\vdash_D$ has the negation property. We define the negation $\overline{P}$ of a program P as a program which contains the predicate non_p iff P contains the predicate p (we suppose there is no name-clash). If p is defined by:

$$p(\vec{X}) : -Body(\vec{X})$$

then non_p is defined by:

$$non\_p(\vec{X}) : -\overline{Body(\vec{X})}$$

We suppose that all the variables in $Body(\vec{X})$ are either quantified or appear in $p(\vec{X})$. If this is not the case the remaining variables are treated as they were existentially quantified at the scope of the clause body.

The negation $\overline{E}$ of an expression E is defined as follows:

$$\overline{A, B} = \overline{A}; \overline{B} \qquad \overline{A; B} = \overline{A}, \overline{B}$$

$$\overline{tell(c(\vec{t}))} = tell(\overline{c}(\vec{t}))$$

$$\overline{\forall X.E(X)} = \exists X.\overline{E}(X) \qquad \overline{\exists X.E(X)} = \forall X.\overline{E}(X)$$

$$\overline{p(\vec{t})} = non\_p(\vec{t})$$

We define the negation $\overline{\Sigma}$ of a multiset Σ of hypotheses $E_i \triangleright F_i$ as the multiset containing the elements $\overline{F_i} \triangleright \overline{E_i}$.

Proposition 4.4.1: There exists a derivation for

$$P, \Sigma \vdash_{si} E \triangleright F$$

iff and only if there exists a derivation for

$$\overline{P}, \overline{\Sigma} \vdash_{isi} \overline{F} \triangleright \overline{E}$$

Proof: We use an induction on the structure of the proof tree, the two proof systems being symmetrical with respect to negation and left-right inversion.

Chapter 5

Implementation, examples and comparison to previous work

In this chapter we present an implementation of a proof assistant for the inclusion of success and several examples of proofs realized with it. A comparison with existing work is also given.

The first section describes the implementation of the proof assistant. It is a Prolog program of about 4000 lines of code. We describe the main data structures and algorithms it uses. The main user commands are presented and an example of interactive session for building a proof is given.

The second section contains some examples of proofs about predicates over the natural numbers.

The third section contains a more complex example about permutations defined as lists. We prove that if the list L_2 is a permutation of the list L_1 then L_1 is a permutation of L_2.

The fourth section contains the proof of equivalence between an implementation of the algorithm *quicksort* and the specification of sorting a list of natural numbers. We prove that the set of successes of *quicksort* and its specification are equal by proving inclusion in both directions.

The fifth section contains a comparison of this work with previous ones.

5.1 Implementation of a proof assistant

In this section we present the implementation of proof assistant for the inclusion of success written in Prolog.

We give an overview of its data structures and its algorithms. We also present an elementary proof search procedure that is reduces the user's work by applying automatically the "simple" rules of the proof system.

Finally we present an example of interactive session between the user and the proof assistant.

We have implemented the proof system for inclusion of finite successes. The program is written in Prolog and we tested it under SWI-Prolog [Wie]. It has about 4000 lines of code and it implements a text user interface.

The program is a proof assistant i.e. it is an interactive program where the user inputs the formula to be proven and issues commands that instruct the system to build parts of the proof. The proof is built from the root towards the leafs. The system verifies the proof using the rules of the calculus $\vdash_{si}$ and the condition C0. A simple interface between the main system and constraint solvers was created. A constraint solver for the term unification (Herbrand domain) is available by default.

Data structures. The main data structure used in the system represents the proof being built. It is a Prolog term of the form

```
pr(PCurrBr, RootId, BrsList, Ext1)
```

where

```
PCurrBr is the current proof branch (or arc)
RootId  is the identifier of the root branch
BrsList is a list of proof branches (or arcs)
Ext1    is a list of additional terms
```

The whole proof tree is not memorized but only the leaves of its branches. They are stored in the list *BrsList*.

The identifier of the root branch is stored in *RootId* and the current branch (the one which the user commands act upon) is stored in *PCurrBr*.

A proof branch is stored in a term of the form:

```
prb(Id-PId, GoalI, GoalC, PElems, Fork)
```

where

```
Id      is the identifier of the branch
PId     is the identifier of the parent branch
           or 'n' if the branch is the root branch
GoalI   is (H->L->R) the initial goal where L,R are sequents
GoalC   is (H->L->R) the current goal where L,R are sequents
PElems  is a list of proof elements (added by base tactics)
Fork    memorizes whether the branch forks or not and whether
           the branch was closed or not
```

The field *fork* also memorizes the identifiers of the branches that fork from the current one.

Among the data stored in a proof branch there are the initial and current sequents *GoalI* and *GoalC* which are terms of the form *H->L->R* where *H,L,R* are the hypotheses, the left and the right expressions of the sequent $H \vdash_{si} L \triangleright R$.

Ext1 is a list of extensions to the initial data structure memorizing a proof. It is of the form:

```
[GenPairs | _]
```

where

```
GenPairs is a list of pairs (FatherId, Id)
```

Id is the identifier of the call that derives from the call with identifier *FatherId*.

Its role will be clear later when we will describe the algorithm used to check condition C0.

A sequent $H \vdash_{si} L \triangleright R$ is represented as tree lists - one for each of its components: *H,L* and *R*.

Each list contains elements of the form:

```
[Id, ElemDat, FatherId]
```

where

```
Id        is an element identifier
ElemDat   is an element data (an encoded term)
FatherId  is an identifier of the element from which
             this one derives
          For hypotheses it is
             0  if it is an initial hypothesis or
             Id-Pos if it is an induction hypothesis where
                Id is the id of the generating call
                Pos is its position in the sequent
```

The field *FatherId* is 0 if the current element doesn't derive from any other element. The role of the fields *Id* and *FatherId* is also related to the verification of condition C0.

Algorithms. The implementation is structured around the interaction with the user. The user inputs the sequent to prove together with the initial hypotheses. The system displays the current branch, waits for a command from the user, executes it and display the new current branch. When all the branches have been closed, the proof is finished.

The data structure presented above that memorizes the state of the proof is manipulated only by a small number of predicates which are called "*base tactics*". They implement the rules of the proof system. Other tactics can manipulate the proof state only by calling the base tactics. This ensures that the proof is correct.

For storing the terms is a sequent a data structure is used where to each variable is associated an index similar to the de Brujin indexes. Thus all the

basic operations in Prolog like the unification, testing the equality of terms, etc. have been rewritten.

We will present in detail how the verification of the condition C0 is done. The algorithm uses some fields of the data structures presented above.

The goal of the algorithm is to ensure that when the rule $(hyp.\, L)$ is applied with an induction hypothesis the call that generated the induction hypothesis is the one form which the corresponding call in the current sequent derives.

For this goal each component of the sequent has associated two indexes:

- *Id* which is an unique identifier
- *FatherId* which is the identifier of the component from which this one derives

We will write $E : Id : FId$ to denote that the component (or expression) E has the identifier Id and FId is its father identifier.

Thus we could write the $(,L)$ such that the identifiers are explicit:

$$\frac{P,\, \Sigma \vdash_{si} E_1 : Id2 : Id1, E_2 : Id3 : Id1, \Gamma \triangleright \Delta}{P,\, \Sigma \vdash_{si} (E_1, E_2) : Id1 : FId1, \Gamma \triangleright \Delta}(, L)$$

The components of the root sequent have the field *FatherId* equal to 0 as they do not derive from any other component.

The pairs of identifiers *Id* and *FatherId* are stored in the field *GenPairs*.

When $(hyp.\, L)$ is applied with an induction hypothesis the programs checks that the corresponding call derives from the generating call by using the pairs. Also the sequent components (or expressions) introduced by $(hyp.\, L)$ have the field *FatherId* set to 0 which insures that no application of $(hyp.\, L)$ is interposed between the generating and the corresponding call.

Two simple tactics for automation of proof search have been implemented and tested with good results.

The first one is called *det* (from "deterministic") and it applies is order the rules (id), $(tell)$, $(,L)$, $(;R)$, $(\forall R)$, $(\exists R)$ as much as possible. Normally, whenever one of these rules can be applied it is a good idea to apply it right away.

The second one is called *a0* (from "automation") and it applies in order the tactic *det* and the rules $(;L)$, $(,R)$, $(\forall L)$, $(\exists R)$ as much as possible. For the rule $(\exists R)$ the existentially quantified variable is replaced with a variable that can be instantiated later (like Prolog does). The instantiation strategy is a very simple "greedy" one: the first occurrence of the variable that can be instantiated usefully is thus instantiated. The code for $(\forall L)$ is still experimental.

In practice the tactic *a0* reduces the size of the proofs that must be manually input by the user from 4 to 5 times. However a certain "proof discipline" must be observed when using this tactic because of its "greedy" strategy.

User interface. As previously explained the user inputs the sequent to prove and then issues commands that the proof assistant carries that modify the state of the proof until the proof is complete.

The interface is a text based one - an illustration is given below:

```
Branch no. 3:

Hypotheses:
12   0  nat_max(A_13_u,B_14_u,C_15_u)->le_nat(A_13_u,C_15_u)
9    0  lt_nat(A_9_u,B_10_u)->nat(B_10_u)
6    0  lt_nat(A_5_u,B_6_u)->nat(A_5_u)
3    0  lt_nat(A_1_u,B_2_u)->le_nat(A_1_u,B_2_u)

Left -> Right:
1    0  list(A_18_u)
->
2    0  permute_list(A_18_u,A_18_u)

?>ind(1).
```

Where the hypotheses are given under the word "Hypotheses" and the left

and right part of the sequent are under "Left->Right" and are separated by "->".

"Branch no." indicates the identifier of the current active branch.

"?>" is the command prompt. In the previous example the user has input the command "ind(1)".

The variables are displayed as

```
VarName_Index_Type
```

where "Index" is the index of the variable (similar to the de Brujin index) and "Type" can be "e" or "u" which indicates whether the variable has been introduced by the existential or universal quantifier.

For each hypothesis and each component of the left and right parts of the sequent are displayed the identifiers *Id* and *FatherId*.

Some of the commands available to the user are:

- det: applies the rules (*id*), (*tell*), $(, L)$, $(; R)$, $(\forall R)$, $(\exists R)$
- dupl([Ids]): duplicates the expressions given by the indexes in the list Ids
- ol(Ind): applies $(; L)$ to the expression of index Ind
- ar(Ind): applies $(, R)$ to the expression of index Ind
- ul(Ind): applies $(\forall L)$ to the expression of index Ind
- er(Ind): applies $(\exists R)$ to the expression of index Ind
- dl(Ind): applies $(def.\, L)$ to the expression of index Ind
- dr(Ind): applies $(def.\, R)$ to the expression of index Ind
- ind(Ind): applies (*ind*) to the expression of index Ind
- hl(Inds, IdH): applies $(hyp.\, L)$ to the expressions given by the list Inds using the hypothesis given by IdH
- a0: applies the tactic a0

There are also commands that receive commands as arguments and compose them to form composed commands. Some examples are:

- C1,C2 : executes the command C1 then C2
- C1;C2 : executes the command C1; if it fails then executes C2

- r(C) : repeats command C as many times as possible

An example of interactive session between the user and the proof assistant is given below.

The goal is to build a proof of

$$NI, [] \vdash_{si} nat(X) \triangleright int(X)$$

where NI is the following program:

```
nat(X):-
          X=0
        ; X=s(Y), nat(Y).

int(X):-
          X=0
        ; X=s(Y), int(Y)
        ; X=p(Y), int(Y).
```

At the beginning the user screen is:

```
Branch no. 3:

Hypotheses:

Left -> Right:
1   0  nat(A_1_u)
->
2   0  int(A_1_u)

?>ind(1).
```

The user enters the command ind(1) and after the execution the following screen is displayed:

```
Branch no. 3:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)
```

```
Left -> Right:
4   1  ex(A_5_e,B_1_u=0;B_1_u=s(A_5_e),nat(A_5_e))
->
2   0  int(B_1_u)

?>det.
```

We observe that the command has added the induction hypothesis to the hypotheses list.

After the command det we have:

```
Branch no. 3:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
6   1  A_1_u=0;A_1_u=s(C_5_u),nat(C_5_u)
->
2   0  int(A_1_u)

?>ol(6).
```

After the command ol(6) two new branches are created and the first one is automatically selected:

```
Branch no. 9:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
7   1  A_1_u=0
->
2   0  int(A_1_u)

?>det.
```

After the command det the screen is:

```
Branch no. 9:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
->
2   0  int(0)

?>dr(2).
```

After the command dr(2) the screen becomes :

```
Branch no. 9:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
->
11   0  ex(A_9_e,0=0;0=s(A_9_e),int(A_9_e);0=p(A_9_e),int(A_9_e))

?>er(11).
```

After the command er(11) the screen becomes:

```
Branch no. 9:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
->
12   0  0=0;0=s(A_9_e),int(A_9_e);0=p(A_9_e),int(A_9_e)

?>det.
```

After the command det the current branch is closed and a new one is automatically selected:

```
Branch no. 10:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
8   1  A_1_u=s(B_5_u),nat(B_5_u)
->
2   0  int(A_1_u)

?>det.
```

After the command det we obtain:

```
Branch no. 10:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
16   1  nat(A_5_u)
->
2   0  int(s(A_5_u))

?>hl([16], 5).
```

The command hl([16], 5) applies the induction hypothesis of index 5 to the expression of index 16:

```
Branch no. 10:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
18   0  int(A_5_u)
->
2   0  int(s(A_5_u))

?>dr(2).
```

After the command dr(2) the situation is:

```
Branch no. 10:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
18   0  int(A_5_u)
->
19   0  ex(B_11_e,s(A_5_u)=0;s(A_5_u)=s(B_11_e),int(B_11_e)
       ;s(A_5_u)=p(B_11_e),int(B_11_e))

?>er(19).
```

After the command er(19) the screen is:

```
Branch no. 10:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
18   0  int(A_5_u)
->
20   0  s(A_5_u)=0;s(A_5_u)=s(D_11_e),int(D_11_e)
       ;s(A_5_u)=p(D_11_e),int(D_11_e)

?>det.
```

After the command det we obtain:

```
Branch no. 10:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
18   0  int(A_5_u)
->
23   0  s(A_5_u)=s(C_11_e),int(C_11_e)
```

```
24   0  s(A_5_u)=p(C_11_e),int(C_11_e)

?>ar(23).
```

After the command det we obtain:

```
Branch no. 27:
Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)
Left -> Right:
18   0  int(A_5_u)
->
25   0  s(A_5_u)=s(C_11_e)
24   0  s(A_5_u)=p(C_11_e),int(C_11_e)

?>det.
```

After the command det we have:

```
Branch no. 28:

Hypotheses:
5   1-0  nat(A_1_u)->int(A_1_u)

Left -> Right:
18   0  int(A_5_u)
->
26   0  int(A_5_u)
24   0  s(A_5_u)=p(A_5_u),int(A_5_u)

?>det.
```

The command det closes the last branch and the system acknowledges it:

```
The proof is: [ind(1), det, ol(6), det, dr(2), er(11), det, det,
hl([16], 5), dr(2), er(19), det, ar(23), det, det]
```

Using the tactic a0 the proof can be shortened significantly: the command sequence is

```
[ind(1), a0, dr(2), a0, hl([16], 5), dr(2), a0]
```

5.2 Natural numbers

In this section we present a few examples of proofs of success inclusion concerning predicates whose arguments are natural numbers.

The following definitions are predicates over that set of natural numbers. The natural numbers are represented here in unary base i.e. the number n is represented as $s(s(...s(0)...))$ (n times).

```
lt_nat(X, Y):-                    % X<Y
    Y=s(X),
    nat(X)
  ;
    Y=s(Z),
    lt_nat(X, Z).

le_nat(X, Y):-                    % X<=Y
    X=Y,
    nat(X)
  ;
    lt_nat(X, Y).
```

The predicates $lt\_nat(X,Y), le\_nat(X,Y)$ succeed when X, Y are natural numbers such that $X < Y$ and $X \leq Y$ respectively.

First we will prove that

```
(lt_nat(X1, X2),
lt_nat(X2, X3))
->
lt_nat(X1, X3)
```

or in the usual mathematical notation

$$(X_1 < X_2) \wedge (X_2 < X_3) \Rightarrow (X_1 < X_3)$$

We have:

```
Branch no. 3:
```

```
Hypotheses:

Left -> Right:
1   0  lt_nat(A_1_u,B_2_u),lt_nat(B_2_u,D_3_u)
->
2   0  lt_nat(A_1_u,D_3_u)

?>a0.
```

After the command a0 we obtain:

```
Branch no. 3:

Hypotheses:

Left -> Right:
4   0  lt_nat(A_1_u,B_2_u)
5   0  lt_nat(B_2_u,D_3_u)
->
2   0  lt_nat(A_1_u,D_3_u)

?>ind(5).
```

The command ind(5) generates an induction hypothesis:

```
Branch no. 3:

Hypotheses:
7   5-1  lt_nat(A_1_u,B_2_u),lt_nat(B_2_u,D_3_u)->lt_nat(A_1_u,D_3_u)

Left -> Right:
4   0  lt_nat(A_1_u,B_2_u)
6   5  ex(C_16_e,D_3_u=s(B_2_u),nat(B_2_u)
       ;D_3_u=s(C_16_e),lt_nat(B_2_u,C_16_e))
->
2   0  lt_nat(A_1_u,D_3_u)

?>a0.
```

After the command a0 we obtain:

```
Branch no. 11:

Hypotheses:
7   5-1  lt_nat(A_1_u,B_2_u),lt_nat(B_2_u,D_3_u)->lt_nat(A_1_u,D_3_u)

Left -> Right:
14   5  nat(A_2_u)
4   0  lt_nat(B_1_u,A_2_u)
->
2   0  lt_nat(B_1_u,s(A_2_u))

?>dr(2).
```

The command dr(2) unfolds the call to *lt_nat*:

```
Branch no. 11:

Hypotheses:
7   5-1  lt_nat(A_1_u,B_2_u),lt_nat(B_2_u,D_3_u)->lt_nat(A_1_u,D_3_u)

Left -> Right:
4   0  lt_nat(A_1_u,B_2_u)
14   5  nat(B_2_u)
->
15   0  ex(D_37_e,s(B_2_u)=s(A_1_u),nat(A_1_u)
        ;s(B_2_u)=s(D_37_e),lt_nat(A_1_u,D_37_e))

?>a0.
```

The command a0 closes the current branch and selects a new one:

```
Branch no. 12:

Hypotheses:
7   5-1  lt_nat(A_1_u,B_2_u),lt_nat(B_2_u,D_3_u)->lt_nat(A_1_u,D_3_u)

Left -> Right:
24   5  lt_nat(A_2_u,B_16_u)
4   0  lt_nat(C_1_u,A_2_u)
->
2   0  lt_nat(C_1_u,s(B_16_u))
```

```
?>hl([4,24],7).
```

The command hl([4,24],7) applies the induction hypothesis:

```
Branch no. 12:

Hypotheses:
7   5-1  lt_nat(A_1_u,B_2_u),lt_nat(B_2_u,D_3_u)->lt_nat(A_1_u,D_3_u)

Left -> Right:
27   0  lt_nat(A_1_u,B_16_u)
->
2   0  lt_nat(A_1_u,s(B_16_u))

?>dr(2).
```

We need to unfold the call at the right by using the command dr(2):

```
Branch no. 12:

Hypotheses:
7   5-1  lt_nat(A_1_u,B_2_u),lt_nat(B_2_u,D_3_u)->lt_nat(A_1_u,D_3_u)

Left -> Right:
27   0  lt_nat(A_1_u,B_16_u)
->
28   0  ex(C_62_e,s(B_16_u)=s(A_1_u),nat(A_1_u)
        ;s(B_16_u)=s(C_62_e),lt_nat(A_1_u,C_62_e))

?>a0.
```

so that we can close the proof with the command a0.

Another example is the proof of

```
(le_nat(X1, X2),
lt_nat(X2, X1))
->
fail
```

where $fail$ can be defined as $tell(false)$ or can be left without definition (in both cases $Succ(fail) = \emptyset$).

For this proof we will first define a new predicate $lt\_nat\_2(X, Y)$ which is equivalent to $lt\_nat(X, Y)$ but has a different definition which facilitates the proof:

```
lt_nat_2(X, Y):-                % X<Y
(
    X=0,
    Y=s(Y1),
    nat(Y1)
  ;
    X=s(X1),
    Y=s(Y1),
    lt_nat_2(X1, Y1)
).
```

Then we will prove the following

```
lt_nat(X, Y)
->
lt_nat_2(X, Y)
```

which we will use as hypothesis for the first proof.

In order to save space we will omit the lower part of the screen which contains the user command and the upper part of the screen which contains the hypotheses. We will indicate the user commands and the hypotheses used or generated.

We will use the following initial hypothesis:

```
nat(X)
->
lt_nat_2(X, s(X))
```

which can be proved simply using the proof assistant - we omit this less interesting proof to save space.

In other words we will derive the sequent:

$$[nat(X) \triangleright lt\_nat\_2(X, s(X))] \vdash_{si} lt\_nat(X, Y) \triangleright lt\_nat\_2(X, Y)$$

We have:

```
1   0  lt_nat(A_6_u,B_7_u)
->
2   0  lt_nat_2(A_6_u,B_7_u)
```

We use induction on *lt_nat* and a0 which creates two branches (commands ind(1), a0), the first being:

```
12   1  nat(A_6_u)
->
2   0  lt_nat_2(A_6_u,s(A_6_u))
```

We apply the initial hypothesis mentioned above which allows us to close the current branch and pass on to the next one (commands hl([12], 6), a0):

```
16   1  lt_nat(A_9_u,B_15_u)
->
2   0  lt_nat_2(A_9_u,s(B_15_u))
```

We apply the induction hypothesis to obtain (command hl([16], 5)):

```
18   0  lt_nat_2(A_9_u,B_15_u)
->
2   0  lt_nat_2(A_9_u,s(B_15_u))
```

In order to prove this we will again use induction (commands ind(18), a0). The first created branch is:

```
30   18  nat(A_20_u)
->
2   0  lt_nat_2(0,s(s(A_20_u)))
```

which we close by unfolding at the right (commands dr(2), a0). The next branch is:

```
30   18  nat(A_20_u)
->
45   0  nat(s(A_20_u))
```

which we can also close simply by unfolding at the right (commands dr(45), a0). The next branch is:

```
30   18  nat(A_20_u)
->
65   0  nat(s(A_20_u))
57   0  lt_nat_2(C_27_e,s(A_20_u))
```

which we close by unfolding at the right of the call *nat* (commands dr(65), a0). The next branch is:

```
79   18  lt_nat_2(A_21_u,B_20_u)
->
2   0  lt_nat_2(s(A_21_u),s(s(B_20_u)))
```

By unfolding to the right (commands dr(2), a0) we obtain first the branch:

```
79   18  lt_nat_2(A_21_u,B_20_u)
->
90   0  lt_nat_2(A_21_u,s(B_20_u))
```

which we close by using the induction hypothesis (commands hl([79], 20), a0). Then we have the branch:

```
79   18  lt_nat_2(A_21_u,B_20_u)
->
100   0  nat(s(B_20_u))
90   0  lt_nat_2(A_21_u,s(B_20_u))
```

which we close using the induction hypothesis again (commands hl([79], 20), a0) thus finishing the proof.

Now we can prove the inclusion we set to prove namely:

$$[lt\_nat(X,Y) \triangleright lt\_nat\_2(X,Y)] \vdash_{si} le\_nat(X_1,X_2), lt\_nat(X_2,X_1) \triangleright fail$$

We have:

```
1   0  le_nat(A_13_u,B_14_u),lt_nat(B_14_u,A_13_u)
->
2   0  fail
```

The tactic a0 eliminates the conjunction at the left:

```
4   0  le_nat(A_13_u,B_14_u)
5   0  lt_nat(B_14_u,A_13_u)
->
2   0  fail
```

We unfold *le_nat* (commands dl(4), a0) to obtain to branches, the first one being:

```
12   0  nat(A_13_u)
5   0  lt_nat(A_13_u,A_13_u)
->
2   0  fail
```

We apply the initial hypothesis $lt\_nat(X,Y) \triangleright lt\_nat\_2(X,Y)$ to obtain (command hl([5], 9)):

```
12   0  nat(A_13_u)
14   0  lt_nat_2(A_13_u,A_13_u)
->
2   0  fail
```

We apply induction on *lt_nat_2* (command ind(14),a0):

```
30   14  lt_nat_2(A_22_u,A_22_u)
12   0  nat(s(A_22_u))
->
2   0  fail
```

We unfold *nat* (commands dl(12), a0):

```
38   0  nat(A_28_u)
30   14  lt_nat_2(A_28_u,A_28_u)
->
2   0  fail
```

and we apply the induction hypothesis $nat(A), lt\_nat\_2(A, A) \triangleright fail$ to close the branch and pass on to the last one (commands hl([38, 30], 16),a0):

```
8   0  lt_nat(A_13_u,B_14_u)
5   0  lt_nat(B_14_u,A_13_u)
->
2   0  fail
```

By applying twice the initial hypothesis $lt\_nat(X, Y) \triangleright lt\_nat\_2(X, Y)$ (commands hl([5], 9), hl([8], 9)) we obtain:

```
43   0  lt_nat_2(A_14_u,B_13_u)
45   0  lt_nat_2(B_13_u,A_14_u)
->
2   0  fail
```

After using induction (commands ind(43), a0):

```
57   43  nat(A_33_u)
45   0  lt_nat_2(s(A_33_u),0)
->
2   0  fail
```

we unfold twice lt_nat_2 to obtain (commands dl(45), a0, dl(45), a0):

```
87   0  lt_nat_2(A_46_u,B_45_u)
74   43  lt_nat_2(B_45_u,A_46_u)
->
2   0  fail
```

where we can apply the last induction hypothesis and finish the proof (commands hl([74, 87], 47), a0).

The complete proofs are given in the appendix.

5.3 Lists and permutations

This section contains examples of proofs of success inclusion of predicates that operate on lists.

Consider the following code:

```
permute_list(L1, L2):-
    (
        L1=[],
        L2=[]
      ;
        L1=[E | L11],
        permute_list(L11, L22),
        insert_into_list(E, L22, L2)
    ).

insert_into_list(E, L1, L2):-
    (
        L2=[E | L1]
      ;
        L1=[E1 | L11],
        insert_into_list(E, L11, L22),
        L2=[E1 | L22]
    ).
```

The predicate $permute\_list(L_1, L_2)$ succeeds if the L_1, L_2 are lists and L_2 is a permutation of L_1. It calls the predicate $insert\_into\_list(E, L_1, L_2)$ which succeeds if L_2 is the list L_1 into which the element E was inserted.

We want to prove that if L_2 is a permutation of L_1 then L_1 is a permutation of L_2:

$$Succ(permute\_list(L_1, L_2)) \subseteq Succ(permute\_list(L_2, L_1))$$

For this purpose we need to prove first the following lemma:

```
1   0  permute_list(A_12_u,B_13_u),insert_into_list(C_14_u,A_12_u,E_15_u)
->
2   0  permute_list(E_15_u,[C_14_u|B_13_u])
```

The tactic a0 eliminates the conjunction at the left:

```
4   0  permute_list(A_12_u,B_13_u)
5   0  insert_into_list(C_14_u,A_12_u,E_15_u)
->
2   0  permute_list(E_15_u,[C_14_u|B_13_u])
```

We apply induction on *insert_into_list* and then the tactic a0 creates two branches (commands ind(5), a0) the first being:

```
4   0  permute_list(A_12_u,B_13_u)
->
2   0  permute_list([C_14_u|A_12_u],[C_14_u|B_13_u])
```

We apply the following lemma lemma which is trivial to prove

```
permute_list(L1, L2)
->
permute_list([E | L1], [E | L2])
```

which allows us to close the branch (commands hl([4], 6), a0). The second branch is:

```
19   5  insert_into_list(A_14_u,B_30_u,C_31_u)
4   0  permute_list([D_29_u|B_30_u],F_13_u)
->
2   0  permute_list([D_29_u|C_31_u],[A_14_u|F_13_u])
```

First we unfold the left *permute_list* (commands dl(4), a0):

```
19   5  insert_into_list(A_14_u,B_95_u,C_31_u)
33   0  permute_list(B_95_u,E_96_u)
34   0  insert_into_list(F_94_u,E_96_u,H_13_u)
->
2   0  permute_list([F_94_u|C_31_u],[A_14_u|H_13_u])
```

then we apply the induction hypothesis (command hl([33, 19], 7))

```
34   0  insert_into_list(A_94_u,B_96_u,C_13_u)
37   0  permute_list(D_31_u,[E_14_u|B_96_u])
->
2   0  permute_list([A_94_u|D_31_u],[E_14_u|C_13_u])
```

By unfolding the right *permute_list* (commands dr(2), a0) we obtain two branches the first one being:

```
37   0  permute_list(A_31_u,[B_14_u|C_96_u])
34   0  insert_into_list(D_94_u,C_96_u,F_13_u)
->
49   0  insert_into_list(D_94_u,[B_14_u|C_96_u],[B_14_u|F_13_u])
```

We close this branch by applying the trivial lemma

```
insert_into_list(E1, L1, L2)
->
insert_into_list(E1, [E2 | L1], [E2 | L2])
```

(commands hl([34], 3), a0). The second branch is identical

```
37   0  permute_list(A_31_u,[B_14_u|C_96_u])
34   0  insert_into_list(D_94_u,C_96_u,F_13_u)
->
49   0  insert_into_list(D_94_u,[B_14_u|C_96_u],[B_14_u|F_13_u])
```

hence we can close it by applying the same lemma (commands hl([34], 3), a0).

We are now ready to prove our initial inclusion:

```
1   0  permute_list(A_18_u,B_19_u)
->
2   0  permute_list(B_19_u,A_18_u)
```

We apply induction on *permute_list* (commands ind(1), a0). Two branches are generated the first one being:

```
->
2   0  permute_list([],[])
```

which we close by unfolding at the right (commands dr(2), a0). The second branch is:

```
43   1  permute_list(A_25_u,B_26_u)
44   1  insert_into_list(C_24_u,B_26_u,E_19_u)
->
2   0  permute_list(E_19_u,[C_24_u|A_25_u])
```

By applying the induction hypothesis (command hl([43], 5)) we obtain

```
44   1  insert_into_list(A_24_u,B_26_u,C_19_u)
46   0  permute_list(B_26_u,E_25_u)
->
2    0  permute_list(C_19_u,[A_24_u|E_25_u])
```

We can now apply the lemma proved above to close the proof (commands hl([46, 44], 9), a0).

The complete proofs are given in the appendix.

5.4 Proof of quicksort

In this section we will present a larger example of proof about the sorting algorithm *quicksort*. We first present the code of the algorithm and the specification of list sorting for natural numbers. Then we prove with the help of the proof assistant that *quicksort* and the specification have the same successes, therefore are equivalent.

The code in Prolog for the well-known algorithm *quicksort* is given below.

```
quicksort_nat(L, S):-
    (
        L=[],
        S=[]
      ;
        L=[E | L1],
        quicksort_nat_1(E, L1, P1, Q1),
        quicksort_nat(P1, P),
        quicksort_nat(Q1, Q),
        append_list(P, [E | Q], S)
    ).

quicksort_nat_1(E, L, P, Q):-
    (
        L=[],
        P=[],
        Q=[],
```

```
        nat(E)
    ;
        L=[E1 | L1],
        (
            le_nat(E1, E),
            P=[E1 | P1],
            quicksort_nat_1(E, L1, P1, Q)
          ;
            lt_nat(E, E1),
            Q=[E1 | Q1],
            quicksort_nat_1(E, L1, P, Q1)
        )
).
```

The predicate $quicksort\_nat(L, S)$ succeeds when L is list of natural numbers and S is the same list sorted.

The specification of sorting a list of natural numbers is given below:

```
sort_list_nat(L, LS):-
        permute_list(L, LS),
        sorted_list_nat(LS).

sorted_list_nat(L):-
    (
        L=[]
      ;
        L=[E],
        nat(E)
      ;
        L=[E1, E2 | L1],
        le_nat(E1, E2),
        sorted_list_nat([E2 | L1])
    ).
```

The predicate $sort\_list\_nat(L, LS)$ succeeds if L is a list of natural numbers and LS is the same list sorted.

Its definition specify that LS is a permutation of L and that LS is a sorted list of natural numbers which is checked by the predicate $sorted\_list\_nat(LS)$.

This is a possible high-level specification of sorting a list which has the advantage of being simple and intuitive but it is not efficient as the running time is exponential in the length of the list.

On the contrary *quicksort_nat*(L, S) is an much more efficient algorithm but it's correctness is much less obvious. We will combine the simplicity of the specification with the efficiency of the implementation by proving that they are equivalent.

We first prove that

$$Succ(quicksort\_nat(L, LS)) \subseteq Succ(sort\_list\_nat(L, LS))$$

At the beginning we have:

```
1   0  quicksort_nat(A_103_u,B_104_u)
->
2   0  sort_list_nat(A_103_u,B_104_u)
```

By applying the commands dr(2), dupl([1]) we obtain:

```
1   0  quicksort_nat(A_103_u,B_104_u)
5   0  quicksort_nat(A_103_u,B_104_u)
->
4   0  permute_list(A_103_u,B_104_u),sorted_list_nat(B_104_u)
```

Then we apply induction on *quicksort_nat*, we use the hypothesis

```
quicksort_nat(L, LS)
->
permute_list(L, LS)
```

and the tactic a0 to obtain (the commands are ind(1), ar(4), hl([5], 39), a0):

```
5   0  quicksort_nat([],[])
->
9   0  sorted_list_nat([])
```

This can be proved easily by unfolding at the right (the commands are dr(9), a0). We obtain:

```
5   0  quicksort_nat([A_111_u|B_112_u],C_104_u)
35   1  quicksort_nat_1(A_111_u,B_112_u,F_113_u,G_114_u)
37   1  quicksort_nat(F_113_u,I_115_u)
39   1  quicksort_nat(G_114_u,K_116_u)
40   1  append_list(I_115_u,[A_111_u|K_116_u],C_104_u)
->
9   0  sorted_list_nat(C_104_u)
```

After duplicating the calls *quicksort_nat* we use the induction hypothesis to obtain (the commands are dupl([39, 37]), hl([37, 41], 7)):

```
42   1  quicksort_nat(A_114_u,B_116_u)
5   0  quicksort_nat([C_111_u|D_112_u],E_104_u)
35   1  quicksort_nat_1(C_111_u,D_112_u,H_113_u,A_114_u)
39   1  quicksort_nat(A_114_u,B_116_u)
40   1  append_list(L_115_u,[C_111_u|B_116_u],E_104_u)
45   0  permute_list(H_113_u,L_115_u),sorted_list_nat(L_115_u)
->
9   0  sorted_list_nat(E_104_u)
```

We use again the induction hypothesis (command hl([39, 42], 7)):

```
45   0  permute_list(A_113_u,B_115_u),sorted_list_nat(B_115_u)
40   1  append_list(B_115_u,[E_111_u|F_116_u],G_104_u)
35   1  quicksort_nat_1(E_111_u,I_112_u,A_113_u,K_114_u)
5   0  quicksort_nat([E_111_u|I_112_u],G_104_u)
48   0  permute_list(K_114_u,F_116_u),sorted_list_nat(F_116_u)
->
9   0  sorted_list_nat(G_104_u)
```

By applying the hypothesis

```
quicksort_nat_1(E, L, P, Q)
->
(list_elems_le(P, E),
list_elems_ge(Q, E))
```

and

```
(list_elems_le(L1, E), permute_list(L1, L2))
->
list_elems_le(L2, E)
```

we obtain (commands hl([35], 57), det, hl([55, 53], 51)):

```
56   0  list_elems_ge(A_114_u,B_111_u)
54   0  sorted_list_nat(C_115_u)
52   0  sorted_list_nat(D_116_u)
51   0  permute_list(A_114_u,D_116_u)
5    0  quicksort_nat([B_111_u|H_112_u],I_104_u)
40   1  append_list(C_115_u,[B_111_u|D_116_u],I_104_u)
59   0  list_elems_le(C_115_u,B_111_u)
->
9    0  sorted_list_nat(I_104_u)
```

Finally by applying hypothesis

```
(list_elems_ge(L1, E), permute_list(L1, L2))
->
list_elems_ge(L2, E),
```

and

```
(sorted_list_nat(L1),
list_elems_le(L1, E),
sorted_list_nat(L2),
list_elems_ge(L2, E),
append_list(L1, [E | L2], L3))
->
sorted_list_nat(L3)
```

we obtain (commands hl([56, 51], 54), hl([54, 59, 52, 62, 40], 48))

```
5   0  quicksort_nat([A_111_u|B_112_u],C_104_u)
68   0  sorted_list_nat(C_104_u)
->
9   0  sorted_list_nat(C_104_u)
```

which allows us to finish the proof (command a0).

The proofs of the lemmas used above are given in the appendix.

To prove the reciprocal

$$Succ(sort\_list\_nat(L, LS)) \subseteq Succ(quicksort\_nat(L, LS))$$

we must build a derivation of:

```
1   0  sort_list_nat(A_190_u,B_191_u)
->
2   0  quicksort_nat(A_190_u,B_191_u)
```

First we unfold *sort_list_nat* and apply the following lemma

```
permute_list(L1, L2)
->
permute_list(L2, L1)
```

to obtain (commands dl(1), a0, hl([5], 12)):

```
6   0  sorted_list_nat(A_191_u)
8   0  permute_list(A_191_u,C_190_u)
->
2   0  quicksort_nat(C_190_u,A_191_u)
```

We apply induction on *sorted_list_nat* and obtain two branches (commands ind(6), a0), the first one being

```
8   0  permute_list([],A_190_u)
->
2   0  quicksort_nat(A_190_u,[])
```

We close it by unfolding at the left and right (commands dl(8), a0), dr(2), a0). We obtain two new branches the first one being:

```
81   6  nat(A_197_u)
8   0  permute_list([A_197_u],C_190_u)
->
2   0  quicksort_nat(C_190_u,[A_197_u])
```

We close it by applying the lemma

```
(nat(E),
permute_list([E], L2))
->
quicksort_nat(L2,[E]),
```

(commands hl([81, 8], 18), a0) which leaves us with one branch:

```
8   0  permute_list([A_198_u,B_199_u|C_200_u],D_190_u)
87   6  le_nat(A_198_u,B_199_u)
88   6  sorted_list_nat([B_199_u|C_200_u])
->
2   0  quicksort_nat(D_190_u,[A_198_u,B_199_u|C_200_u])
```

By unfolding *permute_list* and duplicating *sorted_list_nat* (commands dl(8), a0, dupl([88, 101])) we obtain

```
102   0  insert_into_list(A_222_u,B_224_u,C_190_u)
101   0  permute_list([D_199_u|E_200_u],B_224_u)
87   6  le_nat(A_222_u,D_199_u)
88   6  sorted_list_nat([D_199_u|E_200_u])
103   0  permute_list([D_199_u|E_200_u],B_224_u)
104   6  sorted_list_nat([D_199_u|E_200_u])
->
2   0  quicksort_nat(C_190_u,[A_222_u,D_199_u|E_200_u])
```

to which we can apply the induction hypothesis (command hl([88, 101], 10))

```
104   6  sorted_list_nat([A_199_u|B_200_u])
103   0  permute_list([A_199_u|B_200_u],E_224_u)
87   6  le_nat(F_222_u,A_199_u)
102   0  insert_into_list(F_222_u,E_224_u,J_190_u)
107   0  quicksort_nat(E_224_u,[A_199_u|B_200_u])
->
2   0  quicksort_nat(J_190_u,[F_222_u,A_199_u|B_200_u])
```

By applying the lemmas

```
(sorted_list_nat([E1 | L1]),
le_nat(E2, E1))
->
list_elems_ge([E1 | L1], E2)
```

and

```
(permute_list(L1, L2),
list_elems_ge(L1, E))
->
list_elems_ge(L2, E)
```

(commands hl([104, 87], 93), hl([103, 110], 96)) we obtain

```
102   0  insert_into_list(A_222_u,B_224_u,C_190_u)
107   0  quicksort_nat(B_224_u,[E_199_u|F_200_u])
113   0  list_elems_ge(B_224_u,A_222_u)
->
2   0  quicksort_nat(C_190_u,[A_222_u,E_199_u|F_200_u])
```

and we can close the proof by applying the lemma

```
(insert_into_list(E1, L11, L1),
list_elems_ge(L11, E1),
quicksort_nat(L11, L22))
->
quicksort_nat(L1, [E1 | L22])
```

(commands hl([102, 113, 107], 84), a0).

The proofs of the lemmas used above are given in the appendix.

5.5 Comparison with previous work

In this section we give an overview of a few existing proof systems and we compare them with the proof systems for finite and infinite successes defined in the previous chapters.

The proof systems presented previously allow one to prove success equivalence of logic programs and therefore to prove that an implementation is success equivalent to its specification. Moreover the implementation and specification are written in the same logic language CLP$\forall$, which is an extension of CLP.

The proof systems are based on the Gentzen calculus for classical first order logic and they are relatively simple. They use a unique induction/coinduction rule for all proofs and do not require the addition of new axioms for new predicates. They do not impose restrictions on the programs which can be reasoned about.

There are several proof systems that can be used for reasoning about programs among which Coq[Bar], HOL[Gor93], Isabelle[Pau93], ACL2[KMM00], LPTP[Sta98], $FO\lambda^{\Delta N}$[McD00].

Some of the most popular are the proof systems based on the theory of types, e.g. Coq and HOL. The theory of types originates in the attempts of formalization of mathematics [Chu40]. A main feature is the fact that proofs can be expressed in the same language as propositions by the Curry-Howard isomorphism. An expression of the form

$$P : T$$

has the meaning that P is the proof of the proposition T or alternatively that the type of P is T.

Inductive types can be defined in the language, for example the set of natural numbers can be defined as an inductive type as follows:

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

The system doesn't have an unique induction rule for reasoning about recursively defined objects. Instead for each inductive type the system generates automatically a set of axioms that are added to the current ones. For example for the type above the system generates the Peano's induction:

```
nat_ind
     : (P:(nat->Prop))(P 0)->((n:nat)(P n)->(P (S n)))
     ->(n:nat)(P n)
```

Another class of recursively defined objects are the definitions of functions using the least fixed point:

```
Fixpoint plus [n:nat] : nat -> nat :=
    [m:nat]Cases n of
              0    => m
            | (S p) => (S (plus p m))
            end.
```

To guarantee the correctness of the system the system imposes restrictions on the recursive definitions, for example that the last argument must be strictly decreasing.

Although this restriction insures the correctness of the proof system, it doesn't allow one to reason directly about programs. Instead one can reason about programs indirectly by encoding them as terms and introducing axioms about them.

A proof system that provides for reasoning directly about programs is LPTP [Sta98]. The language used is full Prolog i.e. Prolog with non-logical features such as the *cut* operation and the *var* predicate. The proof system models the operation of a stack-based Prolog interpreter and has a large number of axioms (>60).

The proof system presented in [BP92] and [BGMP97] is the closest to the ones presented here. It allows one to prove that a program P satisfies a property ψ in the context of a set of predicate definitions D written

$$D.P\ sat\ \psi$$

In the terminology used in this work that corresponds to saying that an expression (goal) E satisfies a property ψ in the context of a program P. Therefore we call expression (goal) what they call program and program what they call set of definitions.

The language used is Concurrent Constraint Programming (CCP) which is an extension of CLP where the construct $ask(c) \rightarrow E$ (where c is a constraint and E an expression) has been added. There is no universal quantifier in this language.

The proof system is given as a sequent calculus. The rules are:

$$P.stop\ sat\ true\ \ (C0)$$

$$P.tell(c)\ sat\ c\ \ (C1)$$

$$\frac{P.E\ sat\ \phi}{P.ask(c) \rightarrow E\ sat\ c \rightarrow \phi}(Cl1)$$

$$\frac{P.E_1\ sat\ \phi \quad P.E_2\ sat\ \psi}{P.E_1 \oplus E_2\ sat\ \phi \vee \psi}(Cl2)$$

$$\frac{P.E_1\ sat\ \phi \quad P.E_2\ sat\ \psi}{P.E_1 \parallel E_2\ sat\ \phi \wedge \psi}\ (C3)$$

$$\frac{P.E\ sat\ \phi}{P.\exists X.E\ sat\ \exists X.\phi}\ (C4)$$

$$\frac{P \setminus \{p\}.p(X)\ sat\ \phi \vdash P \setminus \{p\}.Body\ sat\ \psi}{P.p(X)\ sat\ \exists X.\phi}\ (C5)$$

where $(p(X) : -Body) \in P$

$$\frac{P.\exists X.(p(X) \parallel tell(x = y))\ sat\ \phi}{P.p(y)\ sat\ \phi}\ (C6)$$

$$\frac{E\ sat\ \phi \quad T \models \phi \rightarrow \psi}{E\ sat\ \psi}\ (C7)$$

We made slight modifications in the original rules like replacing P (program) with E (expression) and D (set of definitions) with P (program) such that the rules use the same notation as used in this work.

The operators $\|$ and $\oplus$ correspond to the conjunction "," and respectively disjunction ";" in CLP$\forall$.

$P \setminus \{p\}$ is the program P without the definition of p.

$\Phi \vdash E \, sat \, \phi$ where Φ is a set of statements $E' \, sat \, \psi$ stands for the existence of a proof of $E \, sat \, \phi$ using the rules of proof system with additional axioms the statements of Φ.

The set of properties is defined by the grammar

$$\phi ::= c \mid u(\vec{X}) \mid \phi_1 \wedge \psi \mid \neg\phi \mid \exists X.\phi$$

where $u(\vec{X})$ is a user defined predicate.

This proof system provides for reasoning directly about programs - here CCP programs. However the languages for programs and properties are different therefore one can only prove that a program satisfies a property but not the reciprocal. The language of properties as first defined in the paper doesn't contain recursively defined properties.

Sequents of the form

$$P_1 \, sat \, \phi_1, ..., P_n \, sat \, \phi_n \vdash P \, sat \, \phi$$

can be derived using the system. Their meaning is: $P \, sat \, \phi$ can be derived using the axioms and the rules of the system (given above) and $P_i \, sat \, \phi_i$, $i \in \overline{1..n}$ as additional axioms. This facility is similar to the initial hypotheses in our systems but less powerfull.

As the languages of properties and programs are different one cannot prove that $P_1 \, sat \, P_2$ and $P_2 \, sat \, \phi$.

Later in the paper properties defined by the least fixed point are added:

$$\phi ::= p(X) \mid \mu p(X).\phi$$

A rule for the least fixed point is given (Scott induction rule) which using our notations is:

$$\frac{\psi\{p/\phi\} \rightarrow \phi}{\mu p(X).\psi \rightarrow \phi}$$

This rule is equivalent with a restriction of the rule (*ind*) with the condition C0: instead of $p(\vec{t}), \Gamma \triangleright \Delta$ we have $p(\vec{t}) \triangleright \Delta$. The authors do not do not explicitely define a proof system nor prove its correctness.

The rules (*ind*) and (*hyp. L*) separate the introduction of an induction hypothesis and its application. The Scott induction rule doesn't separate them. The rule C5 makes the separation but the initial hypotheses can be used only as axioms. Thus both rules are not stronger than (*ind*) with the condition C0.

Another proof system is the one described in [McD00] which uses higher order λ terms. It contains a rule which expresses the induction principle of the natural numbers. This rules allows reasoning about derivation trees whose length is a natural number, hence about programs.

Chapter 6

Conclusion and Future Work

The goal of this work was creating a formal system for proving the success equivalence of logic programs. The main use of such a system is proving the equivalence between two programs one being the specification and the other bing the implementation.

To achieve this goal a proof system for finite success inclusion was proposed. The programs are written in CLP$\forall$ - a logic language obtained by adding the universal quantifier to CLP. The rules of the system for logical operators and quantifiers are those of the first order logic which makes it more accessible to the users. The system contains a rule for the constraint domain. It also contains an induction rule for reasoning about recursive or mutually recursive predicates. This induction rule is unique and does not depend of the program considered. The proof system is correct under certain conditions. We give two such conditions and we prove their correctness.

An implementation of the system was realized. It consists in a proof assistant - a program that assists the user in building proofs. The proof assistant contains tactics that automate the elementary parts of the proof construction and thus reduce the user's work significantly. Various examples of proofs have been built using the proof assistant. The most complex of them is proving the equivalence between an implementation of *quicksort* and its specification.

A proof system dual to the one for the finite successes is proposed for

proving the inclusion of infinite success. It uses a coinduction rule instead of an induction one. We prove its correctness in a similarly way to the first system.

The main result of this work is defining the two proof systems and proving that they are correct. A side result is the duality between the finite and infinite successes which are given by the least and greatest fixed point of the same operator. A consequence is the symmetry between the induction and the coinduction rules.

There are several directions for future work. An important property in first order logic (and other logics) is the cut elimination property. An analogous property for the proof systems presented would be the elimination of the initial hypotheses i.e. if all the initial hypotheses of a proof are provable then there exists a proof without initial hypotheses. It is not known whether this property holds.

Another direction of research is to find conditions of correctness for the proof systems that are weaker than the conditions presented. This would increase the number of valid proofs and possibly facilitate the users' work for building proofs.

On the implementation side it would be useful to have stronger algorithms for automatic proof search. This would also facilitate proof construction and increase the practical applicability of the proof assistants based of these systems.

Appendix: Example Proofs

This appendix contains the complete proofs of the examples given in chapter 5.

The format of proofs is the following:

%IdH

```
Exp_1
->
Exp_2
    [Commands]
```

where

IdH is the index of the hypothesis (the one used in the command hl(Inds, IdH))

Commands is the sequence of commands that generates the proof.

Proofs for Section 5.2

The complete proofs for the examples given in section 5.2 are:

```
%3
    (lt_nat(E1, E2),
    lt_nat(E2, E3))
    ->
    lt_nat(E1, E3),
        [a0, ind(5), a0, dr(2), a0, hl([4, 24], 7), dr(2), a0],

%6
```

```
    nat(X)
    ->
    lt_nat_2(X, s(X)),
        [ind(1), a0, dr(2), a0, dr(25), a0, dr(41), a0, dr(2), a0, hl([49], 5),
        a0, hl([49], 5), a0],

%9
    lt_nat(X, Y)
    ->
    lt_nat_2(X, Y),
        [ind(1), a0, hl([12], 6), a0, hl([16], 5), ind(18), a0, dr(2), a0,
        dr(45), a0, dr(65), a0, dr(2), a0, hl([79], 20), a0, hl([79], 20), a0],

%12
    (le_nat(E1, E2),
    lt_nat(E2, E1))
    ->
    fail,
        [a0, dl(4), a0, hl([5], 9), ind(14), a0, dl(12), a0, hl([38, 30], 16),
        a0, hl([5], 9), hl([8], 9), ind(43), a0, dl(45), a0, dl(45), a0,
        hl([74, 87], 47), a0]
```

Proofs for section 5.3

The complete proofs for the examples given in section 5.3 are:

```
%3
    insert_into_list(E1, L1, L2)
    ->
    insert_into_list(E1, [E2 | L1], [E2 | L2]),
        [dr(2), a0],

%6
    permute_list(L1, L2)
    ->
    permute_list([E | L1], [E | L2]),
        [dr(2), a0, dr(15), a0, dr(15), a0],

%9
    (permute_list(L12, L21),
```

```
    insert_into_list(E, L12, L1))
    ->
    permute_list(L1, [E | L21]),
        [a0, ind(5), a0, hl([4], 6), a0, dl(4), a0, hl([33, 19], 7), dr(2), a0,
               hl([34], 3), a0, hl([34], 3), a0],

%12
    permute_list(L1, L2)
    ->
    permute_list(L2, L1),
        [ind(1), a0, dr(2), a0, hl([43], 5), hl([46, 44], 9), a0]
```

Proofs for section 5.4

The complete proofs for the examples in section 5.4 are given below.

The proof of correctness of quicksort uses several additional definitions given below:

```
quicksort_nat_1_1(E, L, Q):-
    quicksort_nat_1(E, L, _P, Q).


append_list_2(L1, L2, L3, L4):-
    append_list(L2, L3, LX),
    append_list(L1, LX, L4).


ins_app1(E, L1, L2, L3):-
    append_list(L1, L2, LY),
    insert_into_list(E, LY, L3).


ins_app2(E1, E2, L1, L2, L3):-
    append_list(L1, [E1 | L2], LY),
    insert_into_list(E2, LY, L3).


ins_ins(E1, E2, L1, L3):-
```

```
    insert_into_list(E2, L1, LY),
    insert_into_list(E1, LY, L3).


list_elems_lt(L, E):-
    (
        L=[],
        nat(E)
      ;
        L=[E1 | L1],
        lt_nat(E1, E),
        list_elems_lt(L1, E)
    ).


list_elems_le(L, E):-
    (
        L=[]
      ;
        L=[E1 | L1],
        le_nat(E1, E),
        list_elems_le(L1, E)
    ).


list_elems_gt(L, E):-
    (
        L=[],
        nat(E)
      ;
        L=[E1 | L1],
        lt_nat(E, E1),
        list_elems_gt(L1, E)
    ).


list_elems_ge(L, E):-
    (
        L=[],
        nat(E)
```

```
      ;
        L=[E1 | L1],
        le_nat(E, E1),
        list_elems_ge(L1, E)
    ).


lt_nat_2(X, Y):-
    (
        X=0,
        Y=s(Y1),
        nat(Y1)
      ;
        X=s(X1),
        Y=s(Y1),
        lt_nat_2(X1, Y1)
    ).


p_aux_1(E1, E2, P1, Q1, L2):-
    list_elems_ge(P1, E1),
    quicksort_nat_1(E2, L2, P2, Q1),
    insert_into_list(E1, P1, P2).


app_app(E1, E2, L1, L11, L21, L3):-
    append_list(L11, [E2 | L21], L4),
    append_list(L1, [E1 | L4], [E1 | L3]).


q1_q1(E1, E2, L11, Q1, P2, Q2):-
    quicksort_nat_1(E1, L11, P2, Q3),
    quicksort_nat_1(E2, Q3, Q2, Q1).
```

The proof of

$$quicksort\_nat(L, LS) \triangleright sort\_list\_nat(L, LS)$$

is:

```
%3
    lt_nat(X, Y)->le_nat(X,Y),
        [dr(2), det],
%6
    lt_nat(X, Y)->nat(X),
        [ind(1), a0, hl([14], 5), a0],
%9
    lt_nat(X, Y)->nat(Y),
        [ind(1), a0, dr(2), a0, hl([22],5), dr(2), a0],
%12
    nat_max(X, Y, Z)->le_nat(X, Z),
        [dl(1), a0, hl([9], 3), a0, hl([13], 3), dr(2), a0, dl(16), a0,
        hl([26], 9), a0],

%15
    list(L)->permute_list(L, L),
        [ind(1), dr(2), a0, hl([39], 5), a0, hl([39], 5), a0, dr(50), a0,
        dr(50), a0],

%18
    append_list(_X, _Y, Z)->list(Z),
        [ind(1), a0, hl([19], 5), dr(2), a0],

%21
    (append_list(L1, L2, LT1), append_list(LT1, L3, L4))
    ->
    append_list_2(L1, L2, L3, L4),
        [a0, ind(4), a0, dr(2), a0, dr(22), a0, hl([5], 18), a0, hl([5], 18),
        a0, hl([5], 18), a0, dl(5), a0, hl([71,85], 7), dl(89), a0, dr(2),
        a0, dr(97), a0],

%24
    (insert_into_list(E, L1, LX), append_list(LX, L2, L3))
    ->
    ins_app1(E, L1, L2, L3),
        [a0, ind(4), a0, dl(5), a0, dr(2), a0, dr(32), a0, dl(5), a0,
        hl([43,57],7), dl(61), dr(2), a0, dr(68), a0, dr(69), a0],

%27
    insert_into_list(E1, L1, L2)
```

```
    ->
    insert_into_list(E1, [E2 | L1], [E2 | L2]),
        [ind(1), a0, dr(2), a0, dr(25), a0, dr(2), a0, hl([37], 5), a0],

%30
    (insert_into_list(E, L1, L2), list(L2))
    ->
    list(L1),
        [a0, ind(4), a0, dl(5), a0, dl(5), a0, dr(2), a0, hl([26, 36], 7),
       a0],

%33
    (insert_into_list(E2, L2, LX), append_list(L1, [E1 | LX], L3))
    ->
    ins_app2(E1, E2, L1, L2, L3),
        [a0, ind(5), a0, dr(2), a0, dr(21), a0, dl(18), a0, dr(40), a0,
        hl([4, 51], 30), a0, dl(18), a0, dr(73), a0, hl([4, 84], 30), a0,
        hl([4], 27), a0, hl([4, 101], 7), dl(105), a0, dr(2),
        a0, dr(112), a0, hl([109], 27), a0],

%36
    (insert_into_list(E1, L1, LX), insert_into_list(E2, LX, L3))
    ->
    ins_ins(E1, E2, L1, L3),
        [a0, ind(4), a0, dl(5), a0, dr(2), a0, dr(25), a0, dr(26), a0,
        dr(47), a0, dr(2), a0, dr(64), a0, dl(5), a0, dr(2), a0, dr(87),
        a0, hl([75], 27), dl(98), a0, hl([75, 111], 7), dl(115), a0,
        hl([118], 27), dr(2), a0, hl([119], 27), a0],

%39
    quicksort_nat(L, LS)
    ->
    permute_list(L, LS),
        [ind(1), a0, dr(2), a0, hl([48], 5), hl([50], 5), ind(46), a0,
        dl(53), a0, dl(55), a0, dl(51), a0, hl([103], 15), a0,
        dl(53), a0, hl([135, 51], 24), dl(138), a0, hl([134, 121, 141, 55], 57),
        dl(147), a0, hl([161, 142], 36), dl(164), a0, dr(2), a0, dr(179), a0,
        dr(179), a0, dl(55), a0, hl([232, 51], 33), dl(235), a0,
        hl([53, 218, 238, 231], 57), dl(244), a0, hl([258, 239], 36), dl(261),
        a0, dr(2), a0, dr(276), a0, dr(276), a0],
```

```
%41
    (sorted_list_nat(L),list_elems_ge(L, E))
    ->
    sorted_list_nat([E | L]),
        [a0, ind(4), a0, dl(5), a0, dr(2), a0, dl(5), a0, dr(2), a0, dr(89), a0,
        dl(5), a0, dr(2), a0, dr(152), a0,  dr(152), a0],

%45
    (sorted_list_nat([E1 | L1]),
    list_elems_le([E1 | L1], E),
    sorted_list_nat(L2),
    list_elems_ge(L2, E),
    append_list(L1, [E | L2], L3))
    ->
    sorted_list_nat([E1 | L3]),
        [a0, ind(11), a0, hl([8, 10], 42), dl(6), a0, dr(2), a0,
        dl(4), a0, dl(6), a0, dr(2), a0, hl([78, 89, 8, 10, 58], 13), a0,
        hl([78, 89, 8, 10, 58], 13), a0],

%48
    (sorted_list_nat(L1),
    list_elems_le(L1, E),
    sorted_list_nat(L2),
    list_elems_ge(L2, E),
    append_list(L1, [E | L2], L3))
    ->
    sorted_list_nat(L3),
        [a0, ind(11), a0, hl([8, 10], 42), a0, hl([4, 6, 8, 10, 30], 45), a0],

%51
    (list_elems_le(L1, E), permute_list(L1, L2))
    ->
    list_elems_le(L2, E),
        [a0, ind(5), a0, dl(4), a0, hl([31, 19], 7), ind(20), a0, dr(2), a0,
        dl(34), a0, hl([30, 59, 71], 36), dr(2), a0],

%54
    (list_elems_ge(L1, E), permute_list(L1, L2))
    ->
```

```
    list_elems_ge(L2, E),
        [a0, ind(5), a0, dl(4), a0, hl([33, 19], 7), ind(20), a0, dr(2), a0,
        dl(36), a0, hl([32, 61, 75], 38), dr(2), a0],

%57
    quicksort_nat_1(E, L, P, Q)
    ->
    (list_elems_le(P, E),
    list_elems_ge(Q, E)),
        [ind(1), a0, dr(20), a0, dr(21), a0, hl([63], 5), a0, dr(64), a0,
        hl([63], 5), a0, hl([92], 5), a0, hl([92], 5), a0, dr(94), a0, dr(114), a0,
        dr(114), a0],

%60
    quicksort_nat(L, LS)
    ->
    sort_list_nat(L, LS),
        [dr(2), dupl([1]), ind(1), ar(4), hl([5], 39), a0, dr(9), a0,
        dupl([39, 37]), hl([37, 41], 7), hl([39, 42], 7), hl([35], 57), det,
        hl([55, 53], 51), hl([56, 51], 54), hl([54, 59, 52, 62, 40], 48), a0]
```

The proof of

$$sort\_list\_nat(L, LS) \triangleright quicksort\_nat(L, LS)$$

is (the proof marked as "external" is given above):

```
%3
    insert_into_list(E1, L1, L2)
    ->
    insert_into_list(E1, [E2 | L1], [E2 | L2]),
        [dr(2), a0],

%6
    permute_list(L1, L2)
    ->
    permute_list([E | L1], [E | L2]),
        [dr(2), a0, dr(15), a0, dr(15), a0],
```

```
%9
    (permute_list(L12, L21),
    insert_into_list(E, L12, L1))
    ->
    permute_list(L1, [E | L21]),
        [a0, ind(5), a0, hl([4], 6), a0, dl(4), a0, hl([33, 19], 7), dr(2),
        a0, hl([34], 3), a0, hl([34], 3), a0],
%12
    permute_list(L1, L2)
    ->
    permute_list(L2, L1),
        [ind(1), a0, dr(2), a0, hl([43], 5), hl([46, 44], 9), a0],
%15
    nat(E)
    ->
    quicksort_nat([E], [E]),
        [dr(2), a0, dr(17), a0, dr(17), a0, dr(291), a0, dr(291), a0,
        dr(389), a0, dr(389), a0, dr(390), a0, dr(506), a0, dr(515), a0,
        dr(532), a0, dr(541), a0, dr(390), a0, dr(564), a0, dr(573), a0,
      dr(590), a0, dr(599), a0],

%18
    (nat(E),
    permute_list([E], L2))
    ->
    quicksort_nat(L2,[E]),
         [a0, dl(5), a0, dl(18), a0, dl(19), a0,  hl([4], 15), a0],

%21
    append_list(L1, L2, L3)
    ->
    append_list([E | L1], L2, [E | L3]),
         [dr(2), a0],
%24
    (list_elems_ge(L1, E1),
    le_nat(E1, E2),
    quicksort_nat_1(E2, L1, P1, Q1))
    ->
    list_elems_ge(P1, E1),
        [a0, ind(7), a0, dl(4), a0, hl([44, 6, 31], 9), dr(2), a0, dl(4),
```

```
        a0, hl([78, 6, 65], 9), a0],
%27
    (quicksort_nat_1(E1, L11, P1, Q1),
    le_nat(E2, E1))
    ->
    quicksort_nat_1(E1, [E2 | L11], [E2 | P1], Q1),
        [dr(2), a0],
%30
    (quicksort_nat_1(E1, L11, P1, Q1),
    lt_nat(E1, E2))
    ->
    quicksort_nat_1(E1, [E2 | L11], P1, [E2 | Q1]),
        [dr(2), a0],
%33
    (list_elems_ge(L1, E1),
    le_nat(E1, E2),
    insert_into_list(E1, L1, L2),
    quicksort_nat_1(E2, L1, P1, Q1))
    ->
    quicksort_nat_1_1(E2, L2, Q1),
        [det, ind(9), a0, dl(8), a0, dr(2), a0, dr(35), a0, dr(58), a0, dr(58),
        a0, dr(304), a0, dr(425), a0, dr(546), a0, dr(546), a0, dl(4), a0, dl(8),
        a0, hl([790, 787], 27), hl([814, 6], 27), dr(2), a0, hl([803, 6, 822, 790], 11),
        dl(828), a0, hl([830, 787], 27), dr(2), a0, dl(4), a0, dl(8), a0,
        hl([839, 836], 30), hl([863, 6], 27), dr(2), a0, hl([852, 6, 871, 839], 11),
        dl(877), a0, hl([879, 836], 30), dr(2), a0],
%36
    nat(X)
    ->
    lt_nat_2(X, s(X)),
        [ind(1), a0, dr(2), a0, dr(25), a0, dr(41), a0, dr(2), a0, hl([49], 5), a0,
        hl([49], 5), a0],
%39
    lt_nat(X, Y)
    ->
    lt_nat_2(X, Y),
        [ind(1), a0, hl([12], 36), a0, hl([16], 5), ind(18), a0, dr(2), a0, dr(45),
        a0, dr(65), a0, dr(2), a0, hl([79], 20), a0, hl([79], 20), a0],
%42
    (le_nat(E1, E2),
```

```
    lt_nat(E2, E1))
    ->
    fail,
        [a0, dl(4), a0, hl([5], 39), ind(14), a0, dl(12), a0, hl([38, 30], 16),
        a0, hl([5], 39), hl([8], 39), ind(43), a0, dl(45), a0, dl(45), a0,
        hl([74, 87], 47), a0],

%45
    (quicksort_nat_1(E1, L1, P1, Q1),
    quicksort_nat_1(E1, L1, P2, Q2))
    ->
    (P1=P2, Q1=Q2),
        [det, ind(4), a0, dl(5), a0, dl(5), a0, dl(5), a0, hl([72, 99], 7),
        a0, hl([69, 106], 42), a0, dl(5), a0, hl([72, 136], 7), a0,
        hl([69, 143], 42), a0, dl(5), a0, hl([178, 151], 42), a0,
        hl([154, 189], 7), a0, dl(5), a0, hl([215, 151], 42), a0,
        hl([154, 226], 7), a0],

%48
    (insert_into_list(E1, L1, L2),
    quicksort_nat_1(E2, L1, P1, Q1),
    quicksort_nat_1(E2, L2, P2, Q1))
    ->
    insert_into_list(E1, P1, P2),
        [a0, ind(4), a0, dl(7), a0, hl([6, 37], 45), a0, dr(2), a0,
        hl([52, 6], 45), a0, dl(6), dl(7), a0, hl([60, 103, 95], 9),
        hl([107], 3), a0, hl([92, 110], 42), a0,  hl([125, 117], 42),
        a0, hl([60, 135, 120], 9), a0],

%51
    (list_elems_ge(L1, E1),
    le_nat(E1, E2),
    insert_into_list(E1, L1, L2),
    quicksort_nat_1(E2, L1, P1, Q1))
    ->
    p_aux_1(E1, E2, P1, Q1, L2),
        [det, dupl([4, 6, 8, 9, 9]), hl([4, 6, 9], 24), hl([14, 13, 8, 10], 33),
        dl(23), det, dupl([25]), hl([12, 11, 26], 48), dr(2), a0],
```

```
%54
    (quicksort_nat_1(E, L1, P1, Q1),
    quicksort_nat(P1, P),
    quicksort_nat(Q1, Q),
    append_list(P, [E | Q], S))
    ->
    quicksort_nat([E | L1], S),
        [dr(2), a0],

%57
    nat(E)
    ->
    le_nat(E, E),
        [dr(2), a0],

%60
    lt_nat(X, Y)->le_nat(X,Y),
        [dr(2), det],

%63
    (append_list(L1, L2, LT1), append_list(LT1, L3, L4))
    ->
    append_list_2(L1, L2, L3, L4),
        external,

%66
    (append_list(L1, [E1 | L11], [E1 | L2]),
    append_list(L2, [E2 | L21], L3))
    ->
    (append_list(L1, [E1 | L11], [E1 | L2]),
    append_list([E1 | L2], [E2 | L21], [E1 | L3])),
        [det, hl([5], 21), a0],


%69
    (append_list(L1, [E1 | L11], [E1 | L2]),
    append_list(L2, [E2 | L21], L3))
    ->
    app_app(E1, E2, L1, L11, L21, L3),
        [a0, hl([4,5], 66), a0, hl([9, 10], 63), dl(13), a0, dl(16), dr(2), a0],
```

```
%72
    (lt_nat(E1, E2),
    lt_nat(E2, E3))
    ->
    lt_nat(E1, E3),
        [a0, ind(5), a0, dr(2), a0, hl([4, 24], 7), dr(2), a0],

%75
    (quicksort_nat_1(E2, L11, P1, Q1),
    quicksort_nat_1(E1, P1, P2, Q2),
    lt_nat(E1, E2))
    ->
    q1_q1(E1, E2, L11, Q1, P2, Q2),
        [a0, ind(4), a0, dl(6), a0, dr(2), a0, dr(41), a0, dr(42), a0,
        dl(6), a0, hl([278, 299, 7], 9), dl(303), a0, dr(2), a0, dr(310),
        a0, hl([278, 362, 7], 9), dl(366), a0, dr(2), a0, dr(373), a0,
        dr(374), a0, dupl([7]), hl([446, 6, 7], 9), dl(451), a0, dr(2),
        a0, hl([447, 443], 72), dr(458), a0, dr(459), a0],

%78
     (list_elems_ge(L11, E1),
     quicksort_nat(L11, L22))
     ->
     quicksort_nat([E1 | L11], [E1 | L22]),
        [a0, ind(5), a0, dl(4), a0, hl([28], 15), a0, dl(4), a0, dl(52),
        a0, dupl([60, 35]), hl([60], 57), dupl([64]), hl([53, 64, 35], 24),
        hl([69, 37], 7), hl([61, 65], 27), hl([40], 21), dr(2), a0,
        dupl([53, 35, 56]), hl([103], 60), hl([105, 107, 104], 24), dupl([111]),
        hl([112, 37], 7), dl(115), a0, dupl([56]), hl([136, 40], 69), dl(140),
        a0, hl([35, 131, 137], 75), dl(148), a0, hl([152, 135, 39, 143], 54),
        hl([151, 56], 30), hl([160, 133, 157, 144], 54), a0],

%81
     (list_elems_ge(L1, E1),
     le_nat(E1, E2))
     ->
     list_elems_ge([E2 | L1], E1),
        [a0, dr(2), a0],
```

```
%84
     (insert_into_list(E1, L11, L1),
     list_elems_ge(L11, E1),
     quicksort_nat(L11, L22))
     ->
     quicksort_nat(L1, [E1 | L22]),
        [a0, ind(7), a0, dl(4), a0, dl(6), a0, hl([38], 15), a0, dl(6), a0,
        dl(4), a0, hl([47, 49, 51, 52], 54), hl([65, 64], 81), hl([81, 78], 78),
        a0, hl([65, 64, 87, 47], 51), dl(93), a0, hl([99, 96, 49], 9),
        hl([52], 21), hl([98, 103, 51, 105], 54), a0],

%87
     le_nat(E1, E2)
     ->
     nat(E1),
        [dl(1), a0, ind(6), a0, hl([21], 12), a0],

%90
     (le_nat(E1, E2),
     le_nat(E2, E3))
     ->
     le_nat(E1, E3),
        [a0, dl(4), dl(5), dr(2), a0, hl([22, 12], 72), a0, hl([22, 12], 72), a0],

%93
     (sorted_list_nat([E1 | L1]),
     le_nat(E2, E1))
     ->
     list_elems_ge([E1 | L1], E2),
        [a0, ind(4), a0, dr(2), a0, dr(32), a0, hl([5], 87), a0, hl([5], 87), a0,
        hl([5], 87), a0, dupl([5]), hl([5, 68], 90), hl([69, 73], 7), dr(2), a0],

%96
     (permute_list(L1, L2),
     list_elems_ge(L1, E))
     ->
     list_elems_ge(L2, E),
        [a0, ind(4), a0, dl(5), a0, hl([19, 33], 7), ind(20), a0, hl([36, 32], 81), a0,
        dl(36), a0, hl([32, 51, 65], 38), dr(2), a0],
```

```
%99
     sort_list_nat(L, LS)
     ->
     quicksort_nat(L, LS),
        [dl(1), a0, hl([5], 12), ind(6), a0, dl(8), a0, dr(2), a0,
        hl([81, 8], 18), a0, dl(8), a0, dupl([88, 101]), hl([88, 101], 10),
        hl([104, 87], 93), hl([103, 110], 96), hl([102, 113, 107], 84), a0]
```

Bibliography

[Acz77] Aczel, P.: An Introduction to Inductive Definitions. Barwise K., Ed.. Handbook of Mathematical Logic. North Holland, 1977.

[Bar] Barras, B. et al: The Coq Proof Assistant. Reference Manual. http://coq.inria.fr/doc/main.html.

[Ben98] Bensaou, N., Guessarian, I.: Transforming Constraint Logic Programs. Theoretical Computer Science, 206:81-125, 1998.

[BGMP97] de Boer, F.S., Gabbrielli, M., Marchiori, E., Palamidessi, C: Proving Concurrent Constraint Programs Correct. ACM-TOPLAS 19(5):685-725, 1997.

[BP92] de Boer, F.S., Palamidessi, C.: A process algebra for concurrent constraint programming. In Proc. Joint Int'l Conf. and Symp. on Logic Programming, The MIT Press, 463-477, 1992.

[Bru93] Bruscoli, P., Levi, F., Levi, G., Meo, M. C.: Compilative Constructive Negation in Constraint Logic Programs. In Sophie Tyson, editor, Proceedings of the Nineteenth International Colloquium on Trees in Algebra and Programming, CAAP '94, volume 787 of Lecture Notes in Computer Science, pages 52-67, Berlin, 1994. Springer-Verlag.

[Car91] Carlsson, M.: The SICStus Emulator. SICS Technical Report T91:15, Swedish Institute of Computer Science, 1991.

[Che03] Cheadle, A. M., et al: ECLiPSe: An Introduction. IC-Parc, Imperial College London, Technical Report IC-Parc-03-1,2003.

[Chu40] Church, A.: A Formulation of the Simple Theory of Types. Journal of Symbolic Logic, 5:56-68, 1940.

[Col73] Colmerauer, A., Kanoui, H., Roussel, P., Pasero, R., Un Systeme de Communication Homme-Machine en Français. Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[Cra01] Craciunescu, S.: Preuves de Programmes Logiques par Induction et Coinduction (Proofs of Logic Programs by Induction and Coinduction). In Actes de 10e Journees Francophones de Programmation en Logique et de Programmation par Contraintes, (JFPLC'2001), Paris, France.

[Cra02] Craciunescu, S.: Proving the Equivalence of CLP Programs. In Proc. of International Conference on Logic Programming , Copenhagen, Denmark 2002 (ICLP 2002), LNCS 2401, Springer 2002.

[Dia01] Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System". Journal of Functional and Logic Programming (JFLP), Vol. 2001, No. 6, October 2001.

[FRS01] Fages, F., Ruet, P., Soliman, S.: Linear concurrent constraint programming: operational and phase semantics. Information and Computation no. 164, 2001.

[Gor93] Gordon, J. C., Melham, T. F. (eds.): Introduction to HOL. Cambridge University Press 1993 ISBN 0-521-441897.

[Jaf87] Jaffar, J., Lassez, J-L.: Constraint Logic Programming. Proceedings of Principles of Programming Languages, Munich, 111-119, 1987.

[Kan86] Kanamori, T., Fujita, H.:Formulation of Induction Formulas in Verification of Prolog Programs. In International Conference on Automated Deduction (CADE), pp. 281-299, 1986.

[KMM00] Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach Kluwer Academic Publishers, June, 2000. (ISBN 0-7923-7744-3).

[Kow74] Kowalski, R. A.: Predicate Logic as a Programming Language. In Information Processing 74, Stockholm, North Holland, 1974, 569-574.

[Llo87] Lloyd, J. W.: Foundations of Logic Programming. Springer Verlag GmbH & Co.KG, Berlin, Heidelberg, 1987.

[Mah99] Maher, M.J.: Adding Constraints to Logic-based Formalisms, in: The Logic Programming Paradigm: a 25 Years Perspective, K.R. Apt, V. Marek, M. Truszczynski and D.S. Warren (Eds.), Springer-Verlag, Artificial Intelligence Series, 313-331, 1999.

[McD00] McDowell, R., Miller, D.: Cut-Elimination for a Logic with Definitions and Induction. Theoretical Computer Science, 232: 91 - 119,2000.

[Pau93] Paulson, L.C.: The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993. ftp://ftp.cl.cam.ac.uk/ml/ref.dvi.gz.

[Pet99] Pettorossi, A., Proietti, M.: Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs. Journal of Logic Programming, 41(2-3):197-230, 1999.

[Sar90] Saraswat, V., Rinard, M.: Concurrent constraint programming. ACM Symposium on Principles of Programming Languages 1990, San Francisco.

[Sha83] Shapiro, E.Y.: A Subset of Concurrent Prolog and its Interpreter, Technical Report TR-003, ICOT Tokyo, 1983.

[Sta98] Stärk, R.F.: The theoretical foundations of LPTP (a logic program theorem prover). Journal of Logic Programming, 36(3):241-269, 1998.

[Tam84] Tamaki, H., Sato, T.: Unfold/fold Transformation of Logic Programs. In Proceedings of the Second International Conference on Logic Programming, ed., Sten- Ake Tarnlund, pp. 127-138, Uppsala, 1984.

[Tam92] Tamaki, H., Sato, T.: Equivalence Preserving First-Order Unfold/Fold Transformation Systems. Theoretical Computer Science, 105:57-84, 1992.

[Wie] Wielemaker, J.: SWI-Prolog Reference Manual. Available at www.swi-prolog.org.