Verification and Analysis of Programs in a Constructive Environment

Marco Benini

Advisor:
Prof. Mario Ornaghi
Benini Marco
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39,
20135 Milano–Italy
beninim@dsi.unimi.it
Acknowledgments

I would like to thank Mario Ornaghi and Pierangelo Miglioli for the time and the effort they put into reading, understanding, criticizing and improving my work. Also, I have to thank Alessandro Avellone, Mauro Ferrari, Dirk Nowotka and Carl Pulley for the suggestions, the discussions and the time they spent in helping me.

Without the aid of those persons this thesis would never have been written.
... once he took a piece of chalk and began to write on a blackboard; it was the time when we learned the Beauty of Logic ...

... and it was the smallest gift he gave us.

Thank you, Pierangelo

Professor Pierangelo Miglioli left us forever on August 7th, 1999.
# Contents

1 Introduction .......................... 1
   1.1 The Formal Verification Problem .............................................. 1
   1.2 What is a Constructive Logic ................................................. 3
   1.3 Constructive Verification Systems: a Survey ......................... 5

2 Design of a Verification Environment ........................................... 9
   2.1 Design Issues ................................................................. 10
   2.2 Architecture ................................................................. 17
      2.2.1 The ISABELLE Generic Theorem Prover .......................... 19
      2.2.2 A Traditional Proposal for a Constructive Verification System 21
      2.2.3 A Purely Constructive Verification System ..................... 23
   2.3 Summary ........................................................................... 26

3 The Constructive Reasoner .......................................................... 29
   3.1 The IL and E Systems ............................................................ 32
      3.1.1 The IL-T and E-T Tableau Calculi .................................. 32
   3.2 The Constructive Reasoner .................................................... 39
   3.3 Translating Tableau Proofs into Natural Deduction .................. 40
   3.4 Non-Sound Tableaus ......................................................... 58
      3.4.1 An Illustrating Example ............................................... 61
   3.5 Summary ........................................................................... 64

4 Computer Arithmetic ................................................................. 67
   4.1 Computer Arithmetic ............................................................ 68
   4.2 The Components ................................................................. 72
      4.2.1 The Logical Development ............................................... 72
      4.2.2 The Calculational Engine ............................................. 78
      4.2.3 The Decision Procedure .............................................. 82
   4.3 The Combined System ......................................................... 83
   4.4 Summary ........................................................................... 84

5 Specification Frameworks ............................................................ 87
   5.1 Specification Frameworks ....................................................... 88
      5.1.1 Closed Specification Frameworks .................................... 89
CONTENTS

5.1.2 Open Specification Frameworks ............................................ 95
5.2 Specifications ................................................................. 96
  5.2.1 Specifications for Program Synthesis .............................. 97
  5.2.2 Schemas for Program Synthesis .................................... 99
  5.2.3 Induction Principles .................................................. 101
5.3 Summary ................................................................. 104

6 Analysis of Correctness Proofs .......................................... 105
  6.1 The Collection Method .................................................. 107
    6.1.1 Intuitionistic Logic .............................................. 107
    6.1.2 The E Logic .................................................... 113
    6.1.3 □-Theories ..................................................... 114
    6.1.4 Identity ...................................................... 116
    6.1.5 Induction Principles ........................................... 116
  6.2 How to Distinguish Relevant Information .......................... 118
    6.2.1 Labelling Algorithm .......................................... 122
  6.3 Program and Specification Analysis ................................. 124
  6.4 Summary ............................................................. 126

7 Representing Object Programs .......................................... 127
  7.1 Translating Object Code into Logic ................................ 127
  7.2 Compressing the Representation ...................................... 132
  7.3 Summary ............................................................. 136

8 Theoretical Aspects ....................................................... 137
  8.1 Uniformly Constructive Formal Systems ............................ 137
  8.2 IL is Uniformly Constructive ........................................ 141
  8.3 The E Logic is Uniformly Constructive ............................. 148
  8.4 Analysis of Constructive Principles ................................ 153
    8.4.1 □-Theories ................................................... 154
    8.4.2 Identity ...................................................... 155
    8.4.3 Induction on Natural Numbers ................................ 155
    8.4.4 Descending Chain Principle ................................ 156
    8.4.5 Further Remarks .............................................. 157
  8.5 Equivalence of the E and E-T Calculi ............................... 158
    8.5.1 Soundness .................................................... 160
    8.5.2 Completeness ................................................ 169

9 Conclusions .............................................................. 181

A A Full Example .......................................................... 183
  A.1 Description .......................................................... 183
  A.2 Formalization ....................................................... 183
  A.3 Proving Correctness ................................................ 189
A.4 Concluding Remarks ............................................. 192
A.5 Source Files ..................................................... 197
    A.5.1 strcmp.thy ............................................... 197
    A.5.2 strcmpdef.thy ........................................... 201
    A.5.3 ArithSupp.thy ........................................... 203
Chapter 1

Introduction

The goal of this thesis is to analyze the use of methods from constructive logics in the field of formal verification of computer programs. We want to make evident that a particular view of constructivism, due to P. Miglioli and his group, permits to cope with the problem of verifying the correctness of computer programs. We want to show that these methods have enough expressive power to model the verification task in a deeper way with respect to the traditional approach.

Our analysis will be carried on by designing a tool to formally verify programs. Hence, the content of this thesis is double

- we will describe the design of a Constructive Verification Environment, showing, and solving, the problems involved in this task;

- we will analyze, from an unified point of view, the one of formal verification, many mathematical notions from constructive logics, to show how they can be used to verify programs.

The novelty of our thesis, as it appears from the preceding remarks, lies in the homogeneity in the foundation of our design, and in the uniformity in the supporting mathematics. Thus, the main effort has been in constructing a global framework where many ideas, tools and techniques smoothly interact. To achieve this goal, some new results have been produced, but they have to be regarded as auxiliary to the main goal.

In the following chapters we will show the results of our effort, starting from the architectural description of the Constructive Verification Environment in Chapter 2.

The rest of this introductory chapter is dedicated to a brief summary of the problem of formal verification, to a simple and intuitive introduction to our notion of constructivism, and, finally, to a survey of the existing verification environments.

1.1 The Formal Verification Problem

One of the main reasons for our lack of confidence in reliability of computer programs is the lack of mathematical theories to forecast their behaviors accurately.
Simulation and testing can be a never-ending proposition. Only the most trivial systems can be tested exhaustively. However, if computing systems are modeled in some mathematical theory, they can be studied as mathematical objects, and therefore proving their correctness becomes possible. This is the essence of formal verification.

Correctness proofs can be extremely large and tedious, and it is therefore difficult for humans to check all the proof details, and ensure that they are correct. To reduce the chance of mistakes in such proofs, the idea of mechanically proving the correctness of computer programs has been extensively studied, see the survey in (Boyer and Moore, 1985).

Unfortunately, the use of mechanical system to develop correctness proofs is not sufficient to guarantee the reliability of verified systems.

A big problem in formal verification is the quality of answers, when we perform a correctness proof for a program, the real result of our effort normally is reduced to an answer like “yes, it works” or “it does not conform to specifications”. Of course, considering the amount of time and effort spent in the correctness proof development, such an answer is, at least, unsatisfactory. Normally, the answer is accompanied by the correctness proof; unfortunately, as remarked before, the proof is unreadable for its size and complexity. It requires another expert to be understood.

In fact, as it will appear from the analysis in Chapter 2, the validity of a correctness proof depends on many factors: the formal representation of the program and of the specifications, the reliability of the theorem proving system itself, and the soundness of the tools and theories used in the correctness proof development. All these aspects have an influence on the quality of the correctness proof.

Since the reference life cycle for software production assisted by formal verification, see Figure 1.1, forces the development of the correctness proof after the program has been finished, it follows that a great time passes between the availability of a running program and its certification.

This is the main reason why the scope of formal verification is limited to safety-critical applications, to applications where a failure is not admissible (National Aeronautics and Space Administration, 1995), and in academic research projects, where time is not a big constraint.

In this thesis, we will try to develop a mechanical system where the development of correctness proof is possible. In Chapter 2 we will fix the characters of our design for such a tool. We will call it the Constructive Verification Environment.

In the perspective of the previous comments on the limitations of formal verification, we will try to analyze another direction: to make the proof content readable to non-experts. By non-experts we mean who developed the program, who performed the proof analysis, and, in general who took part in the development of the software product, but is not trained on the formal verification techniques. We will use the word analysis for characterizing this information extraction activity, and we will return on this point in Chapter 6.

A verification environment is more than a specialized theorem prover, in fact, it has to provide many facilities which make no sense in an application oriented in
mechanizing mathematics, such as the automatic translation of code into a logical representation (Chapter 7), a set of specialized theories (Chapters 3 and 4), a way to declare new theories, in particular datatypes (Chapter 5).

The design of a verification system, as we are about to do, do not reduce the knowledge an user must have to perform correctness proofs using it; simply, the right instruments are available when needed.

The main character a verification environment must possess is uniformity: it should use a minimal amount of concepts, from which all the available tools are derived. The nature of these concepts determines the power of the system, as well as its understandability. In the survey of the constructive verification environments we will present the core concepts underlying these systems.

1.2 What is a Constructive Logic

In this section, we want to introduce the notion of constructivism we will employ in the rest of this thesis. The content of this section summarizes, in a somewhat intuitive way, the idea of constructive logical system as developed by P. Miglioli and his group.

In the following, $L$ will denote a logic and $T$ a theory.

We say that the logical system $T + L$ is naively constructive iff

- if $T|_L A ∨ B$ then $T|_L A$ or $T|_L B$;
• if \( T \vdash \exists x. A(x) \) then \( T \vdash A(t) \), for some term \( t \).

Of course, a logic is naively constructive iff it has this property, the system \( \emptyset + L \), and a theory \( T \) is naively constructive, fixed a logic \( L \), iff \( T + L \) is so. When we use the word constructive for a logical system, we will intend naively constructive.

As the name suggests, a naively constructive system may defeat the standard, well-established notions of constructive system, as reported in (Beeson, 1985; Troelstra, 1977a; Troelstra and van Dalen, 1988). An example is the pathological system described in (Ferrari, 1997b).

The concept of naively constructive system permits, and this is our main interest, to give a computational reading to formulas. In fact, we are interested in something stronger: to give a computational reading to both proofs and formulas.

This part will be developed in Chapters 5, 6 and 8; but we can give an example. Let us consider the formula

\[
\phi(x) \equiv \exists y. x = 2y \lor x = 2y + 1
\]

we can prove that \( \text{PA} \models \forall x. \phi(x) \), where \( \text{PA} \) is Peano arithmetic, and \( \text{IL} \) is intuitionistic logic.

Actually, it is possible to prove that \( \text{PA} + \text{IL} \) forms a constructive system, and we will prove a stronger result in Chapter 8.

Now, from \( \text{PA} \models \forall x. \phi(x) \), we derive that, for any term \( t \), \( \text{PA} \models \phi(t) \), that is, \( \text{PA} \models \exists y. t = 2y \lor t = 2y + 1 \).

In a computational reading, \( \phi(t) \) represents a task we have to compute: given a term \( t \), find an element \( y \) such that \( t = 2y \) or \( t = 2y + 1 \).

The key point is that the derivation sign \( \models \) means “the task represented on the right hand side can be computed assuming to know how to compute the tasks on the left hand side”.

Very roughly, \( T \models \phi \) can be read as the machine \( L \) is able to compute \( \phi \) using the library functions in \( T \).

This view is very approximate, but, essentially, it gives the intuition behind the notion of constructivism we need.

It should be clear that a naively constructive system admits a computational reading of formulas because the proofs in \( T + L \) can be recursively enumerated; in our example, we must find an index for a proof of the fact \( t = 2t' \), or of the fact \( t = 2t' + 1 \), for some term \( t' \). The fact that \( \text{PA} + \text{IL} \) is naively constructive guarantees that such an index exists.

At the same time, it should be clear that, by enumerating all the possible proofs, we really use \( L \) as a machine, which computes by enumeration, but, in the definition of constructive system, we take no advantage of knowing the proof of a disjunctive (existential) statement.

\footnote{We have to remark that not any logical system admits a computational reading for formulas; for instance, in classical Peano Arithmetic, \( \text{PA} \models \neg G \lor \neg \neg G \), where \( G \) is the Gödel sentence (Gödel, 1931). If that formula admits a computational reading, it means that we can decide whether \( \text{PA} \models G \) or \( \text{PA} \models \neg G \), violating the Gödel incompleteness theorem.}
1.3. **CONSTRUCTIVE VERIFICATION SYSTEMS: A SURVEY**

Thus, we are interested in a stronger notion of constructive system, which assigns a computational reading to proofs; this notion is called *uniformly constructive formal system* (Ferrari, 1997b).

A formal treatment of this notion is too involved to be presented here, although we will return on this point in Chapters 6 and 8, but the intuition behind is very simple: every proof \( \Pi \) in a logic \( L \) establish, directly or indirectly, a set of true facts; we call this set the *information content* of \( \Pi \). We say that a system \( T + L \) is uniformly constructive iff

- if \( T \models \exists x. A(x) \), via the proof \( \Pi \), then there is a term \( t \) such that \( A(t) \) is in the information content of \( \Pi \).

An immediate consequence is that every uniformly constructive system is also naively constructive. The converse does not hold, as proven in (Ferrari, 1997b).

A non immediate consequence is that, in a uniformly constructive system \( T + L \), we have a description of the *computational model* for proofs; in fact, as we will describe in Chapter 6, the way to show that a system is uniformly constructive provides a computational interpretation for proofs, for details see (Miglioli and Ornaghi, 1978).

In the case of the logics employed in this thesis, this *computational interpretation* takes the form of an algorithm which extracts information from proofs; details will be explained in Chapter 6.

In a sense which will appear clear later, the computational interpretation of formulas and proofs provides the uniform background which links together verification, analysis and synthesis.

### 1.3 Constructive Verification Systems: a Survey

The purpose of this chapter is to show the "state of the art" on the subject of Constructive Verification Systems. To this purpose, we will describe some Verification Systems. Our choice of which systems to speak about, has been taken according to the following criterions: well-known, used in real applications, and significantly innovative.

Essentially, one way to classify the Verification Systems is based on the amount of automatic reasoning they perform: it is a matter of fact that the automation of the mathematical proving activity has not been very successful; theorem provers need guidance from the user when they have to solve non-elementary problems. Our opinion is supported by many results in literature, e.g. (Paulson, 1989), but, more than anything else, our opinion is supported by the fact that even small successes in automatic proving of difficult theorems are regarded as big gains for the field of
CHAPTER 1. INTRODUCTION

Theorem proving, see e.g. (Boyer and Moore, 1984; Paulson, 1992b; McCune and Padmanabhan, 1996). Thus, when we have to formally verify a program, it makes sense to consider interactive theorem provers, where the user has the control over proof development.

The most well-known Verification Systems are based on three theorem provers, HOL (Gordon and Melham, 1993), ISABELLE (Paulson, 1990) and PVS (Owre et al., 1996). We prefer to group these systems together because they are all based on a common set of concepts, inherited from their common ancestor, LCF (Paulson, 1987).

The real interest of these Verification Systems can be understood only by looking at the distinguishing features of whole family of provers. They have a common ancestor, the LCF theorem proving system (Plotkin, 1977; Paulson, 1987), and they inherit its main characters:

- higher order logic
- tactics and tacticals
- distinction between meta logic and object logic
- type system
- ML

In fact, all the theorem provers and the verification systems in the LCF family are built starting from the previous ideas; different ways to combine these concepts have been developed, but the basic ingredients are the same.

All these systems are coded in the ML programming language (Cousineau et al., 1986), which has been developed to program the meta level of LCF. The system itself distinguishes between an object level and a meta level; the former encodes the logical system which is used to describe a problem, and where the reasoning on the world is performed; the latter provides the system where the object level is defined and where it takes a meaning. The meta level provides a logic, the meta logic which constitutes the main inference engine of the proving system; the inference rules of the object level become formulas in the meta level logic and they may be composed using the meta level inference rules to derive new results.

To ensure the correct application and composition of object level inference rules, a type system is used; the meta level is typed, and a particular type is devoted to represent truth at the object level; a careful use of this type ensures that no incorrect derivations may be performed in the object level by acting on the meta level.

Object level inference rules are coded as tactics, that is steps in the proof space of the object level; more complex tactics can be programmed by composing in suitable ways the basic inference rules. The composition of tactics is possible, and the operators are called tacticals. Typical examples of tacticals are THEN, sequential composition of tactics allowing backtracking, DETERM, which acts like the Prolog
cut, disallowing backtracking on the tactic which is applied to, \texttt{DEPTH-FIRST}, which applies a tactic taking into account backtracking alternatives in a depth-first fashion, until a goal gets solved.

All these tools together give raise to an highly homogeneous interactive proving system, which has been successfully employed in many real verification tasks, both on hardware and on software systems, e.g., see (Gordon, 1983; Birtwistle and Graham, 1990; Chou, 1994; Harrison, 1995; Harrison, 1996).

The COQ proof assistant (Dowek et al., 1993) is, in a sense, another descendant of the LCF family, but it has a series of important differences, which make it interesting by its own. In fact, COQ was developed after LCF, and, thus, inherits all the techniques we exposed so far, but it changes an important point: its meta logic is constructive.

The meta logic of COQ is the Calculus of Inductive Constructions, an extension/specialization of the Calculus of Constructions (Coquand and Huet, 1988).

Although, this kind of constructive proving system is quite distant from the goal we will pursue, it presents important points of interests.

First of all, the meta logical system is constructive in the sense of (Martin-Löf, 1984). Thus, we have a logic where propositions are expressed by formulas, and their proofs are expressed by types; then, using the Curry-Howard isomorphism (Howard, 1980), in the same way as for typed \(\lambda\)-calculi (Barendregt, 1992), one obtains a computational interpretation of proofs as programs.

The interest of the COQ group in certified software has put in the verification systems project some novelties with respect to the standard LCF approach: as a result, the COQ system is conceived to verify programs, and to synthesize them.

This double approach is possible because of the particular meta logic COQ adopts; in fact, thanks to the widespread application of the Curry-Howard isomorphism, there is a direct correspondence between programs and their correctness proofs. We should remark that this correspondence is possible only because the Calculus of Inductive Construction is constructive.

The LEGO (Lego, 1998) proof assistant has been developed from the Edinburgh Logical Framework, another member of the LCF family. As for all other members of this family, LEGO is an interactive theorem prover which adopts the tactic based way of reasoning, and the separation between object level and meta level.

But LEGO has a distinctive feature: its meta logic is constructive in a wide sense. More specifically it implements an extension to the Calculus of Construction; this extension goes into the direction of Pure Type Systems (Barendregt, 1992).

From a simplified point of view, a Pure Type System (PTS) is a particular typed \(\lambda\)-calculus, which has a direct logical interpretation; the logic corresponding to a PTS is derived by applying a variant of the Curry-Howard isomorphism, which interpret a \(\lambda\)-term as a proposition, and a \(\lambda\)-type as a proof. A term \(t\) admits \(T\) as a type if, and only if, the proof corresponding to \(T\) proves the formula corresponding to
$t$ in the logic corresponding to the PTS. One may reverse the previous exposition, forcing the existence of a Curry-Howard isomorphism, and using it to synthesize a logic from a PTS.

The interesting part of the adoption of the PTS approach is in the computational interpretation of proofs the Curry-Howard isomorphism imposes. In fact, using the notion of reduction in the PTS, which is a subrelation of the standard $\beta\eta$-reduction of type free $\lambda$-calculus (Barendregt, 1984), one may interpret types as programs. Thus, proofs are, in a very strict and precise sense, programs.

As a consequence, like in Coq, the use of a constructive meta logic permits to develop in parallel verification and synthesis of programs.

Our goal, as exposed before, is to enlarge this view. In Coq and Lego, the key point is that proofs have a computational reading, that is, they can be interpreted as programs. This result is achieved in these systems by the Curry-Howard isomorphism, and it is possible only because the starting logics are constructive.

A part of our goal, is to take the computational interpretation of formulas and proofs as fundamental, and trying to derive a system which permits to exploit this character at its maximum; thus, we will not assume to have an isomorphism between proofs and programs, but, instead we will start from a proper definition of constructive system, which captures the computational interpretation we are interested in. Then we will try to show what can be done in these systems with respect to the formal verification problem.
Chapter 2

Design of a Verification Environment

The purpose of this chapter is to investigate the problems which arise from a traditional approach to formal verification, and to propose and discuss solutions in a constructive perspective, eventually leading to the definition of the architecture of a Constructive Verification System.

The chapter is divided into two parts: the first one is devoted to the analysis of the problems we encounter during the design of a verification system; the second part illustrates the resulting architectures.

The role of this chapter in the thesis is to describe the whole design of our project, while the following chapters are devoted to analyze and present the single components, thus here the unified view of the whole system is shown.

The discussion which follows will try to give evidence to the fact that a constructive approach to the studied problem is both natural and profitable. We have already spoken in Chapter 1 about our idea of constructivism.

In this sense, we need to cope with the view a programmer has of its own activity: he thinks to the objects he manipulates (data, variables, ... ) in a classical way, i.e., something is either true or false, but he thinks to programs in a constructive way, that is, every output must be directly constructed (computed) from the inputs.

Of course, the previous fact is a good signal for the need of our analysis, but, alone, it does not provide a motivation to adopt a constructive system as the basis for a formal verification environment.

In the following analysis, we will show how a constructive approach provides in a uniform setting a series of advantages, which are difficult to achieve in a more traditional approach.

But, and it will completely clear after Chapter 8, there is at least one point which makes a constructive approach superior: the ability to extract information from a correctness proof, and to ensure that the set of extracted information is uniform.
2.1 Design Issues

The ultimate purpose of this chapter is to design the architecture of a verification environment. In this perspective, we should provide an in-depth analysis of the basic problems in formal verification. This process will eventually generate a series of design issues forming the key infrastructure behind the implemented solutions.

The existing verification systems which are based on a constructive approach, e.g., LEGO (Lego, 1998) or Coq (Dowek et al., 1993) are far from what we intend to do with constructive mathematics. They are based on constructive type theory (Martin-Löf, 1984), which does not permit, in our opinion, to perform the analysis we must do in our thesis.

Hence, we start our analysis from the very beginning, reinterpreting the basic design decisions which underlie a verification system development. The spirit of our analysis would be to try to follow the flow of thoughts we did when we designed the solutions we are to propose.

As we already remarked in Chapter 1, the formal verification problem requires the adoption of a theorem prover.

This solution is the very standard choice underlying most verification systems. Successful verification systems can be classified into two major categories: the former adopts model-checking as the main proving technique; the latter uses interactive theorem provers (or, equivalently, logical frameworks, generic theorem provers, proof assistants) as their inference engines. The most useful source of information on verification tools is on the World Wide Web at the URL

http://www.comlab.ox.ac.uk/archive/formal-methods.html

Since constructing counter-models, and, in general, reasoning by refutation in non classical logics is difficult, because of the non uniform interpretation of negation, and, since most non classical systems have a natural deduction-like or sequent-like presentation, we stick on the logical framework paradigm.

A great amount of theorem proving systems is available nowadays; we have two possibilities: to build a new theorem prover from scratch, or to use an already developed one. Since developing a new theorem proving system constitutes a major effort (Paulson, 1992a), we need a strong motivation for this choice. Actually, we do not see such a strong need; the advantages of adopting an already developed system, exceed by far the problems: we can take advantage of the experience, of the deep testing and of the wide libraries of theories and tools a well established theorem prover possesses.

Developing a new system from scratch means to have a better tuning for our usage, but it requires, in some way, to rediscover the wheel. This fact is better understood by looking at the preliminary design documents of theorem proving systems; just as an example, we refer the reader to (Benini et al., 1993; Vaccari et al., 1993; Paulson, 1992a; Paulson, 1996).

Using a theorem prover to ensure correctness of a formal proof poses another problem:
2.1. DESIGN ISSUES

Is the theorem prover correct?

This is not an academic question in formal verification; correctness proofs are long and complex, see, e.g., Appendix A, resulting almost impossible to check by hand; hence, a great amount of confidence in the proving system is required.

An obvious solution to this problem is to choose among logical frameworks available today in the theorem proving community. A suitable choice must accomplish the following features, to cope with the problem of the prover correctness\footnote{We will denote architectural requirements as \([Rn]\).}:

[R1] The theorem prover must be generic, so we can customize it to work with non-standard logics and theories.

[R2] The theorem prover has to be in a stable development phase, i.e., mature and thoroughly tested; we prefer a prover for which a correctness argument was given. In other words, we ask for reasonably reliable prover.

Requirement [R1] cuts down the number of systems we can regard as candidates for our inference engine; requirement [R2] refines the list of possible candidates. Essentially, our choice is restricted to the following systems: COQ (Dowek et al., 1993), HOL (Gordon and Melham, 1993), ISABELLE (Paulson, 1994), NuPRL (Constable and Howe, 1990). Because our experience (Benini, 1995; Benini, 1996; Benini et al., 1998a) is wider on HOL and ISABELLE, we restrict our attention to these two theorem provers.

We should remark that the previous requirements do not suffice to our purposes; in fact, a verification system will provide many packages and tools that, although based on the theorem prover, are essential to the verification development. An error in those packages will nullify any effort to guarantee the correctness of any formal proof developed using the whole system.

A solution to this kind of problems is to develop consistency preserving techniques and to adopt them when building tools on the top of the prover. The simplest, but very efficient, way to ensure that additions to the logical system are consistent is the one adopted in the HOL system (Gordon and Melham, 1993; Harrison, 1995; Harrison, 1996): it disallows the introduction of new axioms except for definitional extensions (Barwise, 1977). Moreover, the only way to construct new theorems is to build them using inference rules and axioms; new inference rules can be coded only by composing the already derived ones. The result of this strict policy is that no inconsistent axiom or inference rule can be added to the system; the main drawback is that, in a sense, the inference system is fixed, forcing not to use instruments, techniques, ideas and theories which cannot be coded into this schema.

Hence, this kind of policy is unsatisfactory for our purposes. What we need is something less restrictive; our proposal is for a system which allows us to specify an axiomatic theory \(T\) and guarantees that, if \(T\) is consistent, then we cannot derive a contradictory consequence in \(T\) by using in whatever way the theorem proving system and the tools built on its top.
We can state this fact as another requirement on our ideal verification system:

[R3] Moreover, there should be a mechanism which ensures that the tools operating in an axiomatic theory, are not allowed to automatically derive false consequences from true assumptions.

Combining [R3] with [R2], we ask for a system where theories can be represented in a sound way, and tools and packages are, at least, as reliable as the theorem prover itself.

The flexibility of ISABELLE’s syntax and its simplicity in the definition of new theories makes it the best choice for our inference engine. Unfortunately, ISABELLE does not meet requirement [R3], but we can develop techniques and policies which force this feature on our customization of the proof assistant. Later on, in the next section and in Chapters 3 and 4 we will return on this point.

The next question which directly comes from the formal verification problem is:

How to ensure that the representation of the program is coherent with the real code?

The standard solution is to use a formal semantics of the programming language as the way to represent programs. In this respect, there are too many references in literature to give here a comprehensive view of what one can achieve. We will not go further on this line because most real programming languages lack such a formal semantics.

There is an alternative way to formally represent programs, i.e., to give them a purely operational semantics. We have got some experience on this line (Benini et al., 1998a) and we devise two subproblems:

- to ensure that the execution of the program is coherent with its formal execution, that is, the behavior as computed by the operational semantics.

- the representation of a program in this way tends to be very long, complex and uninformative.

As already stated in (Benini et al., 1998a), the former problem can be solved adopting machine language as the programming language. This choice is not restrictive since this is the format of executables.

The latter problem forces the development of hiding and abstraction techniques, which are able to compress representations without loss of information. Details of a possible approach we explored can be found in (Benini et al., 1998a).

Summarizing, we require

[R4] a formal verification system where a representation for object code can be computed automatically;

[R5] that a formal verification system provides abstraction mechanisms over code representations to keep them manageable.
The last question we think worth considering when looking at the formal verification problem is:

\textit{How to guarantee that the formal specification and the real specification are requesting the same things?}

From a mathematical point of view, there is no answer to this question because a real specification, being non formal, does not have a mathematical nature, so there is no way to ensure that its formal counterpart is equivalent; more, no notion of equivalence, comparison or whatsoever formal relation can be given and checked. So, it appears hopeless to deal with this problem.

We do not think so. Our proposal for a solution is proof analysis and interpretation; we think that, by extracting information from a correctness proof and by rephrasing it on the original program, one can provide formal evidence of the programmer's thought, and, in this empirical way, one checks consequences of formal specifications (the correctness proof) against consequences of informal ones (the programmer's belief). Consequently,

\textbf{[R6]} We require that the verification system provides a way to extract information from correctness proofs, and it is able to remap the relevant facts onto the original code.

Later, in Chapter 6, we will examine the point and the techniques to perform analysis and to remap proof information onto programs.

A non trivial problem we avoided to discuss till now is closely related to representations: we posed a series of requirements to ensure that representations are correct; but

\textit{What about feasibility?}

We have to ask for enough representative power to model programs and specifications, and we have to ask for reasoning tools.

A requirement like \textbf{[R1]} is not enough since it allows the creation of new theories, but it poses no rules on the shape of theories.

The need is for a coordinated set of theories, each one with the proper decision procedures. The distinguishing features of this set must be:

- it must provide the basic data types to represent object code;
- it must provide simplification and decision procedures with enough power to solve \textit{trivial} goals (Paulson, 1997b);
- it has to be extensible to allow the modeling of new data types.

We think that the basic data types which are absolutely needed to represent and to reason about programs are integer arithmetic, a theory of bitwise operations, and the elementary data types like lists, arrays, sequences, trees, ... Our proposal is for a
tool which allows to reason about computer arithmetic and another tool to represent abstract data types; details will follow in the next section and in Chapters 4 and 5.

Although necessary, we dislike the above discussion on problems related to feasibility of the verification task because it provides no strong insight; we would prefer to have some form of coordination among theories which guarantees [R3], [R5] and [R6], that is, a way to abstract over the code and the data structure, and to reason on the result ensuring soundness, as well as the concrete possibility of extracting information from the proofs we developed, i.e., to formally analyze them. The first problem is to relate representations of specifications and of programs; to represent in a strict way this relations gives us the instruments to analyze what we can say, which amounts to solve the feasibility problem.

In Chapter 5, we will introduce a non usual representation for programs which strongly depends on constructive features; in a sense that we will detail later, in that representation, a specification is formalized as a formula, while a program becomes a proof pattern.

In a more standard approach, in Chapter 7, we will also consider a representation for programs where a program becomes a formula. The main advantage of the second schema is a direct coding of programs into their representations, see, e.g., Appendix A. The proofs as programs representation, in this respect, is, by far, more involved.

For the moment being, we just reformulate requirement [R4]:

[R4] The representations of programs and specifications must be linked and, whenever a program is proven correct, the correctness proof must become part of this linkage; moreover the representation of object code has to be automatically computed.

The reason why we require a strong link between the formal counterparts of programs and specifications, are, essentially the following:

- It is easier to keep track of which specification describes the behavior of which proof (program).

- It is easier to automate the proof analysis task, since the program is part of the description of a formal verification task: the complete description comprises the program, the specification, the correctness proof and the framework, i.e., the set of theories and tools, used to develop the proof itself.

- It closely resembles the standard organization of the work in a formal verification group (Windyey et al., 1991; Cohn, 1988; Cohn, 1989); every member checks parts of the whole project, while correctness of the combination is proven by assembling these subparts. Making explicit the link between proofs, specifications and programs, allows us to introduce modularization.

We want to underline the importance of modularization; although it is a natural way to organize the work of a formal verification group, modularization has a great
2.1. DESIGN ISSUES

importance of its own. In fact, in our proposal for a formal verification system it plays a central role.

The great importance of modularization comes from a series of practical and theoretical issues. The first one is that, as already said, a correctness proof is long and complex, so a strong need for intermediate facts to prove clearly appears; technically, we will refer to this issue as lemmification.

Considered as the only modularization technique, lemmification appears to be insufficient and primitive. We need something more structured, from which lemmification follows naturally. Of course, our choice is restricted by the previous requirements, in particular [R1] and [R6], that is, generality and the need for analysis.

To assert requirements for a satisfactory modularization technique, we start by fixing the problem. We need to assert lemmas not just to split a complex proof into subpieces, but lemmas should have a proper meaning, or, in other words, their statements should be relevant.

In general what is relevant is a subjective judgment, but there are natural ways to introduce lemmas; in particular when a statement comes from a definite subset of theories, like an arithmetical result, or from a particular theory, like properties of boolean algebras, we can safely assume that a lemma has to be introduced. The modularization which comes from this lemma introduction strategy, is fairly good: it encapsulates, in an object oriented fashion, results from a set of theories, masquerading their proofs and the peculiar reasoning techniques for those theories; in particular, the use of decision procedures is limited to the necessary scope, and this fact results in a good memory usage, a key problem in interactive theorem proving (Paulson, 1992a).

We will solve the modularization problem mainly using specification frameworks, in the sense of (Lau and Ornaghi, 1994), a compact mathematical instrument to link a logical theory to specifications and programs, which gives its best in a constructive approach. As a requirement, we ask for an extensible set of theories such that

[R7] There is a way to inherit results from a theory so to minimize the amount of proving work.

Mathematical and implementive details about frameworks in our setting will be discussed in Chapter 5. Till now, we avoided as much as possible to speak about constructivism in our analysis for the basic requirements of our verification system. Henceforth, our requirement are neutral, that is, they do not force any particular choice for a preferred logical system. To simplify references, we summarize our requirements in Table 2.1.

We can divide the requirements into two subgroups:

- requirements on the instruments we should adopt; in this class we group [R1], [R2], [R3] and [R7].
- requirements on the methods we should employ; in this class we find [R4], [R5] and [R6].
[R1] The theorem prover must be generic.

[R2] The theorem prover must be stable.

[R3] There is a mechanism which ensures soundness of tools over the theorem prover.


[R5] Abstraction mechanisms over object code.

[R6] A tool to extract information from proofs.

[R7] Inheritance between theories.

Table 2.1: Architectural Requirements.

The reason behind this classification is that the first group does not prevent the choice of a particular logic, while any architectural solution satisfying the second group generates a new series of problems because the inner nature of the involved theoretical aspects.

In this phase it begins to appear why a constructive approach is interesting. In fact, adopting a classical approach we have one computational reading of our representations, the one which permits the coding of the operational semantics of object programs. In the constructive, we retain this computational reading of formal entities, hence we are able to verify programs in the same style as in the classical case. But in our proposal we are able to include two other computational readings.

The first one will be detailed in Chapter 5; we will show that it is possible to interpret a proof of a specification as a program. We will fix a way to write specifications, and a way to develop proofs so that parts of the proof of a specification can be directly translated into a program correctly implementing the specification. Although we can replicate the same reading in a classical setting, the theoretical frame where these results where conceived is purely constructive.

This reading has a twofold consequence: it provides a natural way to represent the link between code, its representation, its specification and the correctness proof, thus realizing [R4], and it provides a natural way to synthesize correct programs.

But there is a third computational reading of the formal aspects of verification; we can extract information from a proof, and we can, in a constructive setting, read the extraction process as a computation of the program satisfying the specification which constitutes the conclusion of the proof. We will detail the extraction method in Chapter 6 and we will show its properties in Chapter 8.

This last reading constitutes the essence of our idea of constructivism, and it is possible only in a constructive setting. From a practical point of view, we are able to ensure that the quality of the extracted information from a correctness proof is
high enough to be able to replicate the computation of the program itself.

This fact opens the door to automatic analysis of correctness proofs.

The previous remarks have a natural counterpart in the applicative field: we have shown in Chapter 1 what is the intended process for developing software when adopting formal verification.

In this respect, a formal verification environment which permits the synthesis of correct programs can be very useful, in fact, in the early stages of development, when most of the software components are specified, but no yet implemented, it is useful to have a non efficient but correct implementation of them, and this can be synthesized in our environment. In this way, the verification team can work, and the developing team has the possibility to perform traditional debugging on the whole software, knowing that some parts are formally correct by construction. When the works reaches the point where the synthesized components have to be implemented in the final release, there is already a starting code that can be modified for performances, and this fact simplifies the work of the verification team, that has to change the existing correctness proofs, without redoing them from scratch.

The ability to track which points of a correctness proof has to be changed when the code is modified, is an information coming out from a formal analysis of the correctness proof itself, hence the need for an automatic analysis of proofs. As we will discuss at length later, formal analysis of correctness proof is also a way to enhance confidence in the correct formalization of specifications, and it makes possible to inspect proofs even to non logicians, and in particular to the developing team, thus improving the quality of documentation and understanding of what the code does. The ultimate consequence is that maintenance of programs becomes easier.

Of course, nothing prevents to achieve the same results in a classical verification environment, but at the cost of a non uniform approach, because there is no unifying theoretical framework which makes plain and smooth the transition between verification, analysis and synthesis.

### 2.2 Architecture

Architecture As in the previous section we discussed problems and requirements, in this section we will discuss about architectural solutions.

To start off an analysis of solutions, we can look at the standard architecture of a verification system, and, considering the weak and strong points with respect to our requirements, we may try to design an architectural structure corresponding to our needs.

The standard architecture of a verification environment is shown in Figure 2.1. This is the reference architecture for Coq, HOL, Isabelle, NuPRL and PVS, that is, the most well-known theorem provers and formal verification systems.

The base of that architecture is a generic theorem prover, which provides support for higher order logic, either directly, by adopting it as the meta logical language
and reasoning system, or indirectly, by implementing it as an object logic. Any verification system provides a minimal amount of reasoning tools, and, in particular, a partial decision procedure for classical logic, the classical reasoner, a partial decision procedure for arithmetic, which solves simple goals on natural and integer arithmetic, and a support tool to define, at least, inductive datatypes, and to reason on their structure. Finally we suppose to have some kind of user interface which permits to interact with the reasoning system and with its tools.

As we said in the previous section, we have chosen ISABELLE as our generic theorem prover. It roughly conforms to the architecture in Figure 2.1. It provides a generic reasoning system (Paulson, 1990), based on intuitionistic Horn clauses, which results in an higher order meta logic. ISABELLE has an already developed theory for classical higher order logic (Paulson, 1997c), in a natural deduction presentation. It provides an instance for HOL (higher order logic) of the generic classical reasoner (Paulson, 1997c; Paulson, 1997b), as well as an instance of the simplifier (Paulson, 1997c). The former tool is an automatic partial decision procedure for classical logic with equality; roughly speaking, it reasons using the calculus in (Dyckhoff, 1992) with a variety of strategies to search in the proof spaces, from simple depth first and breadth first, to more sophisticated algorithms like best first and A*; for a detailed description of these search strategies we refer the reader to (Rich, 1983). The latter tool provides a generic rewriting engine which transforms formulas into formulas substituting equals by equals, according to a given set of equalities.

In the HOL object logic, many important mathematical theories are developed, like set theory (Paulson, 1992b), Peano arithmetic and many others (Paulson, 1997d). Proper expansions to the simplifier and to the classical reasoner are available inside the base system to support reasoning in these theories.

Also, in the HOL object logic a package (Paulson, 1993; Paulson, 1997a) for generating inductive structures, and to reason on them is available as part of the standard ISABELLE distribution. It supports induction on the structure of recursive definitions, as well as coinductive reasoning, and primitive recursive definition of functions and predicates. A theory for least/greatest fixed points in set theory,
2.2. ARCHITECTURE

roughly as described in (Barwise and Moss, 1996), is available, too.

Finally, ISABELLE has a text-based user interface which adopts the EMACS editor, with an appropriate mode, as a front-end to the prover.

Since we will adopt ISABELLE as our inference engine, in the next subsection we will describe its structure and its relevant features according to our purposes. For a more comprehensive exposition of the system and of the accompanying tools, we refer the reader to the official documentation, in particular to (Paulson and Nipkow, 1990; Paulson, 1994; Paulson, 1997a; Paulson, 1997b).

The choice of ISABELLE, with respect to our requirements, means to fulfill both [R1] and [R2].

2.2.1 The ISABELLE Generic Theorem Prover

In this subsection, we are interested in recalling the notions and the terminology which appears to be relevant for our exposition. The content of this subsection is based on (Kalvala, 1994), which provides a simple, easy to understand introduction to the ISABELLE theorem proving system.

The basic ISABELLE system provides a framework for developing proof systems. This framework, called Pure ISABELLE, can be instantiated with rules, axioms and syntax for many different logics, in the form of ISABELLE theories. The logics which are implemented in ISABELLE make use of a meta logic defined in Pure ISABELLE, with a syntax for theorems and a set of rules for manipulating theorems. Underlying the use of ISABELLE, then, is the concept of a meta level logic and an object level logic. The meta logic of ISABELLE is intuitionistic higher order logic.

Several instantiations of ISABELLE to particular object logics are part of the distribution, providing off-the-shelf theorem proving environment. The most interesting of these theories for our purposes is HOL, classical higher order logic.

The ISABELLE theorem proving system is written in the ML programming language (Paulson, 1996), and, comes from the tradition of LCF provers (Paulson, 1987), thus inheriting the style and many ideas from the original Logic for Computable Function theorem prover.

In the interaction with ISABELLE, object level sentences are entered as ML strings, which are then parsed into terms; usually, parsing is completely transparent to the user. Parsing depends on the language, and the syntax declared in theories. For example, in HOL, the operators \( \implies \) and \( \rightarrow \) both denote implication; the first one is used to show an inference from the antecedent to the consequent, and signifies a derivation. The latter, though, is used to describe a property in the object level.

Theorems are objects in ISABELLE that are constructed from terms, and are entered as axioms or produced by proofs.

Theories are modules that contain information on a logic, or, in the case of applications, definitions that describe the problem being modeled. Often, theories are defined as extensions of other theories. In a sense, every object logic can be regarded as an extension of Pure, the meta logic of ISABELLE.
For more complex applications, one often has to define particular \textit{types}, especially in the case of strongly typed logics as HOL. The types can be completely new ones, or they can be type \textit{constructors}. Every type may be a member of one or more type \textit{classes}, providing a way to introduce polymorphism. Types can also be parametrized by other types.

In ISABELLE, object level inference rules are theorems that contain premises and a conclusion, linked by a meta level implication. They are created through proofs or as axioms, just like any other theorem.

In object logics there are two basic ways of reasoning: forward and backward proofs. The former consists in combining theorems to create new ones; the typical way to do this is by meta level resolution.

Backward proofs are performed in ISABELLE by setting up a goal, and by applying tactics to reduce it into zero or more subgoals, until no subgoals are left. A tactic is an ML function which takes a goal and produces a sequence of sets of subgoals, which logically derive from it. The transformation which a tactic performs on a goal is guaranteed to be sound because of the implementation of the ML type system. Technically, a goal is a parameter passed to the tactic, and it has ML type \texttt{thm}, that is, it is a theorem; the result of the computation of a tactic is a sequence of theorems; since the only way to build a theorem in a theory is to derive it from meta level axioms (the object level inference rules) by means of meta level inference rules, no inconsistent result from a sound theory can be derived.

A tactic produces a sequence of possible results, allowing \textit{backtracking}; when a result is not accepted, the next element of the sequence, i.e., another possible outcome for the application of the tactic, is chosen.

Elementary tactics allow to perform various kind of resolution on the current goal, proof by assumption, as well as simple manipulations on the premises and the conclusion of a goal.

ISABELLE provides a set of generic packages, which can be instantiated for a particular theory, supplying powerful and complex tactics for proving.

One of the most important of these packages is the \textit{Simplifier}. Simplification consists in rewriting a goal using a set of equational theorems; non equational theorems are converted to equalities when added to a simplification set. The simplifier is a very powerful tool which allows to do conditional rewriting, but it suffers the possibility of infinite rewriting loops. Most simplification sets are already available within the theories supplied with the system, but new ones can be easily created.

The other most important package is the \textit{Classical Reasoner}; it provides a level of automation of proof in the form of proof search tactics such as \texttt{fast_tac}, which exploit most of the in-built rules for the predefined logical connectives. These tactics can prove a surprisingly large number of goals. If several elementary inference rules may be applicable to a subgoal, there are nodes in the proof where it must be decided which to apply, and then which choice of subgoals to explore further, with the possibility of backtracking if a proof is not found at the particular branch taken. The search method can be simple depth first or can incorporate heuristics.

Another important package is the \textit{Induction Package}; it provides a way to define
in HOL inductive types, automatically generating theorems which code induction and case analysis. It permits also to define coinductive structures, and recursive functions over a data type.

### 2.2.2 A Traditional Proposal for a Constructive Verification System

Now we will return to search for a suitable design which permits to exploit constructive techniques in formal verification.

Our goal now is to decide what additions to ISABELLE are needed to fulfill most of the requirements we fixed in Section 2.1. We will discover that some requirements are out of reach starting from this architectural design, asking for a radical change in the basic picture; nevertheless, a formal verification system based on a traditional approach can easily benefit from new versions of the ISABELLE prover, but, more important, enables us to reuse tools and methods developed independently either for ISABELLE, or for other theorem provers. This expansion effort is almost for free, since it does not require changes in the tools themselves, and we have to modify just small parts of our system, essentially the ones which have a global view of the verification process. Referring to Figure 2.2, which illustrates our architectural proposal, the only impact that additional theories and packages will have on this architecture is limited to the Proof Analyzer, which needs to know about the theory language, in particular about definitional extensions, induction principles and atomic constructors.

<table>
<thead>
<tr>
<th>User Interface</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical Reasoner</td>
<td>Simplifier</td>
</tr>
<tr>
<td>Constructive Reasoner</td>
<td>Computer Arithmetic Toolkit</td>
</tr>
<tr>
<td>Induction Package</td>
<td>Specification Frameworks</td>
</tr>
<tr>
<td>Higher Order Logic</td>
<td>Proof Analyzer</td>
</tr>
</tbody>
</table>

**Figure 2.2: An Architecture Derived From the Standard**

Our proposal for a constructive verification system based on the standard architecture is shown in Figure 2.2. The theories, tools and packages written in boldface are additions to what the standard distribution of ISABELLE already provides.

*ISABELLE does not provide a way to ensure that a theory is *internally* consistent.*
But, as remarked before, no tactic can introduce inconsistencies if, implicitly, they are not already present in the theory definition. So a standard consistency proof, done by hand, assures us that a theory is "good".

As soon as tactics are implemented inside the Pure system, there is no way to deduce false conclusions from true assumptions, so [R3] is met, from this point of view. But this is not the only way to implement tactics; ISABELLE provides the so-called oracle mechanism, which allow an external theorem prover to prove a goal, bypassing the type checking mechanism which ensures soundness of standard tactics.

Not to use oracles is a bad idea; a theorem prover external to the ISABELLE type system can be quite faster than ISABELLE tactics, since it can represent goals in a more convenient way, and it can manipulate them very efficiently by using specialized transformations which directly modify the data structures. Of course, there is no built-in mechanism to ensure that oracles are correct. We have developed a technique (Avelone et al., 1999) which allows to use an external prover, and which is able to check the result it provides, automatically. A non immediate consequence of this technique is that we can use a non sound prover, on purpose, just to push performances, discarding its wrong answers in the checking phase. The technique, applied to the constructive reasoner, and a discussion of non sound theorem proving is presented in Chapter 3.

About the formalization of object code, that is [R4], we propose a straightforward representation; its practical use is shown in (Benini et al., 1998a), and an example is discussed in Appendix A. The idea behind this representation is as follows:

- Every instruction is mapped into a formula.
- The formula associated with an instruction has the following format:

\[ \forall t. \text{pc}(t) = n \rightarrow \phi \]

where \( n \) is the address of the current instruction, and \( \phi \) states what are the values of all registers and of memory at time \( t + 1 \).

- The registers are represented as functions from time to values, with the exception of the status register which is decomposed in a series of predicates on the time domain, which are true when the flag is set; memory is represented by a function which takes two arguments, time and memory address, and returns the content of the addressed cell in the specified moment.

As we remarked before, when speaking about the consequences of a constructive approach, we need a conceptual model of the software development process, which allows us to strictly formalize specifications, and to link them to proofs, and program representations. Our solution is depicted in Figure 2.2 as the Specification Framework package, described in Chapter 5, which provides exactly what we need to meet [R4], [R7] and [R5].
2.2. ARCHITECTURE

Our information extraction techniques give their best when applied to constructive proofs, for reasons which will be clear in Chapter 8, so we prefer, whenever possible, to reason in a constructive way. In ISABELLE that means to limit the use of the Classical Reasoner, in favor of an equivalent package, the Constructive Reasoner, which does not make use of the Law of Excluded Middle.

About theories, we have another strong need; since our representation for programs makes use of functions which are supposed to compute on the domain of values, that is, bytes, words, long words, \ldots, we need a logical theory about the arithmetic a computer really employs.

In Chapter 3, this package, the Computer Arithmetic Toolkit (Benini et al., 1998b), will be presented: it is composed by a theory, in the ISABELLE sense, a specialized equational reasoner, and a partial decision procedure.

In order to extract information from a proof, we need another package, the Proof Analyzer, which makes use of ISABELLE's proof objects (Paulson, 1997e). The description of this package, mostly on the theoretical side, can be found in Chapter 6.

The design we get in this way is not best one for the purposes of the thesis, since it is based on classical higher order logic. But, from a practical point of view, it makes sense to retain the design in Figure 2.2, since it is easier to implement and it provides tools which, with a minimal effort, can be reused by others who are interested in.

Thus, our final decision is to keep this design, and to develop in parallel a purely constructive verification system, we are going to describe in the next section.

2.2.3 A Purely Constructive Verification System

Since we want to develop a purely constructive verification system, our first decision is to fix the logic. It may appear that intuitionistic logic is the best choice, since it is standard, with a rich series of results, and there is a lot of experience on. Our feeling is that intuitionistic logic is good, but it misses a fundamental point: in a programmer's view, data structures, and, in general, types and domains, are classical; we mean that a programmer thinks that an elementary truth on a domain, like an atomic formula, is true or it is false. There is no space for a third possibility, like we do not know. Essentially, this fact amount to say that we have a classical interpretation in mind, when programming, but our activity is purely constructive, since we have to build up a result in any computation. This interpretation is impossible to achieve in intuitionistic theories, since, they can be anti-classical (Troelstra, 1977b), thus opposite to the intended view we spoke above.

The weakest constructive logic whose extensions are guaranteed to be classically compatible is Kuroda logic (Gabbay, 1981), see also Chapter 8. From our point of view, Kuroda logic enjoys all the properties we need, namely, it is uniformly constructive, see Chapter 8, any theory built on its top is classically compatible and it permits to develop the specification framework approach.

But we would like to have a feature Kuroda logic has not; negation is not constructive in Kuroda logic, while we would like it to be. We mean that a negative
statement like \( \neg(P \land Q) \) may have a proof in Kuroda logic which does not make evident if \( \neg P \) holds, or if \( \neg Q \) holds; we would like that the logical system we will employ, has this character in addition to the needed properties we cited before.

In (Miglioli et al., 1982a; Miglioli et al., 1989b), a logic, called the \( \mathbb{E} \) system, which has the characters we want, is presented. In essence, the \( \mathbb{E} \) logical system permits to combine classical sentences with constructive reasoning. Developing specification frameworks in this logic, we get exactly the intended view for applicative domains, that is, we interpret data structures in a classical sense, by means of initial models. More, this logic is uniformly constructive, that is, it permits information extraction from proofs in a smooth way, and the extraction procedure is both solid, from a computational point of view, and rich, from the theoretical side. Finally, this logic has been developed with the specific purpose to make negation constructible.

So our choice will be the \( \mathbb{E} \) system, which is briefly presented in Chapter 3, and analyzed with respect to program specification, representation and synthesis in Chapter 5, while its properties from the point of view of information extraction are discussed in Chapters 6 and 8.

---

```
<table>
<thead>
<tr>
<th>User Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplifier</td>
</tr>
<tr>
<td>Constructive Reasoner</td>
</tr>
<tr>
<td>Specification Frameworks</td>
</tr>
<tr>
<td>( \mathbb{E} ) Logic</td>
</tr>
<tr>
<td>Signature Morphisms</td>
</tr>
<tr>
<td><strong>ISABELLE</strong></td>
</tr>
</tbody>
</table>
```

Figure 2.3: A Purely Constructive Architecture

The architecture we are proposing is shown in Figure 2.3. It is very similar to the one in Figure 2.2, except that we changed the logic from HOL to the \( \mathbb{E} \) system. As before, there are strong reasons to have a Constructive Reasoner, the Computer Arithmetic Toolkit, and, of course, a support for Specification Frameworks. We can benefit from the ISABELLE simplifier to add reasoning power to the system for free. Also, the Proof Analyzer is meant to perform the same task as in the previous architecture, that is, extracting information from proofs, and, eventually, remapping it onto programs, when considering a correctness proof.
2.2. ARCHITECTURE

There is a natural way to synthesize programs from specifications is the context of specification frameworks, and so we add a Program Synthesizer tool in our design.

We think that program synthesis is important in a formal verification system, if we want to apply the system on the field. In fact, when we introduce the formal verification task in the normal software development cycle, we can maximize the benefits only if we verify the software we are producing while we are writing it. In this way we have early validations of code, implying less debugging phases. Program synthesis enters into this view as an accelerator, so to build quickly right, but inefficient code, that is supposed to be substituted as soon as it becomes available a version which satisfies both the specifications and the performance requirements.

Using in an appropriate way specification frameworks, we are able to automate the process of inheriting results from one theory to another; the Signature Morphisms Manager enters in this process as the main tool acting beyond the scenes.

Apparently it seems that the architecture in Figure 2.3 is the right one, for the same reasons as for the traditional design proposal, plus the bonus of a constructive system, the E logic, with many good properties from the point of view of program verification.

This is not right. In fact, there is a weak point in this design, which lies in the fact that checking a theory to have an isoinitial model requires a semantical reasoning.

We need that the check is done in a semi automatic way, since we stated in [R3] that an axiomatic theory should provide mechanisms to ensure its soundness and the soundness of provers, i.e., decision procedures, it exports to the whole system.

Extending the verification system with new theories and decision procedures is a natural requirement, because we need to verify programs on specific domains, which are not known in advance. So expanding the knowledge domain with new specification frameworks constitutes the natural way to model programs and their environments, and, because of this fact, we need a mechanism which ensures that the extensions are done in proper way, that is, respecting the constraints on admissibility of specification frameworks.

The admissibility check involves a proof which states that a closed specification framework has an isoinitial model.

It may not be clear to the reader, at this point, the precise sense of the previous discussion, since specification frameworks and their theory will be introduced in Chapter 5, but, nevertheless, this poses a serious constraint on a possible architectural design.

The admissibility check needs to automatically ensure that

- the theory is reachable, that is, every element of the intended domain is denoted by a term.

- the theory has to satisfy some basic proof-theoretical properties, namely, it has to be complete on atomic disjunctions, i.e., for every atomic formula $A$, we must be able to prove $A \lor \neg A$. 
Hence, we need a tool, which does not interact with the user, but permits to Isabelle to perform the admissibility check.

For this very reason, our proposal for the design of a purely constructive verification system is the one shown in Figure 2.4.

![Figure 2.4: A Constructive Architecture with the Admissibility Check](image)

The right hand side of the figure shows the previous proposal, so nothing new is introduced. We think to this side as the object level of our system.

The left hand side closely resembles the standard architecture, as represented in Figure 2.1. It contains what is needed to perform the admissibility check, and it particular, it permits to check for reachability.

The novelty in this design is the $\mathbf{E}$ Logic Admissibility Checker box. It is an Isabelle theory which provides the tools which are needed to test a theory for reachability, and for atomic completeness, hence providing the necessary support to the Specification Framework Package.

### 2.3 Summary

In this chapter we analyzed the needs for building a constructive formal verification system. We ended up with two design proposal; the former is illustrated in Figure 2.2, the latter in Figure 2.4.
The former is a traditional architecture for a verification system, which can directly benefit from improvements in the area, such as the availability of new decision procedures, new theories for the ISABELLE system, and so on.

The latter is a purely constructive verification system, and it acts as our reference architecture. This design proposal allows to fully exploit the idea we are analyzing, that is, application of constructive techniques to the verification of computer programs.

In the following chapters, we will discuss the tools we are proposing to develop, and the theory behind them. In particular:

Chapter 3 We will introduce the intuitionistic first order logic IL, and the E logic; we will develop a tableau prover for both logics, which forms our Constructive Reasoner. Also, we will introduce a way to certify a tableau proof by translating it into natural deduction; this translation server a double purpose: first, it permits to use a non verified prover, ensuring that its results are certified; second, it produces proofs in a format we are able to analyze.

The tableau prover for both logic has been implemented, and it is available from the author. Its development phase is $\beta$-testing.

Chapter 4 We will introduce the Computer Arithmetic Toolkit, which provides a theory for Computer arithmetic, a specialized simplifier and a decision procedure for that theory.

This tool is publicly available on the Internet, in the HOL version, while the E logic version is in a $\beta$-testing phase.

Chapter 5 Here, we will introduce the theory behind specification frameworks, along with the design of the tools which are responsible to implement them in the Constructive Verification Environment.

In this respect, we will clarify the format of admissible program specifications, and their intended meaning.

Also, we will speak about the synthesis of programs from the proof of their specifications; this topic is directly related to the notion of specification framework, since the format of the definition of the latter, allows us to interpret proofs as programs.

At the moment being, there is no implementation of these modules, and our effort has been in designing the tools in such a way that their action is possible, by finding constraints which does not limit the expressive power of the approach, but they can be checked automatically by a theorem prover; a major design effort was put into coordinating the actions of these tools with the other packages which are present in the Constructive Verification Environment.

Chapter 6 In this chapter, we will show the Collection Method, which is the algorithm we employ to analyze formal proofs. A discussion on the implementation techniques follows.
The second part of the chapter is devoted to the analysis of correctness proofs, and, consequently to the formal analysis of programs and their specifications. The Program Analyzer has never been completely implemented, but some prototypical versions have been used to test the ideas which constitute the applicative aspects of the chapter.

Chapter 7 Here, we will show how to translate object code into a logical representation. The algorithm which performs this task has been implemented and used in practice. In fact, there is even an enhanced version, which permits to translate object code into other formal representations, namely, Temporal Logic of Actions (Lamport, 1994), and CCS processes (Milner, 1980; Milner, 1989).

Chapter 8 This chapter does not describe any tool, but it provides a series of theoretical results whose goal is to show how the various tools share a common root. The aim of this chapter is to show that the design we developed in the thesis is uniform, not only from an applicative point of view, but also from a theoretical point of view.
Chapter 3

The Constructive Reasoner

The purpose of this chapter is to illustrate the Constructive Reasoner as implemented in the Constructive Verification Environment. Referring to Chapter 2 we will describe both the implementation for the traditional design and for the purely constructive design.

We will show two versions of the Constructive Reasoner, one for intuitionistic logic, which is the intended companion for the traditional Constructive Verification System, and another version, for the E logical system, which has to be used within the purely Constructive Verification System.

The Constructive Reasoner is composed by two provers, both based on a tableau calculus. Both provers are completely trustworthy, that means, they cannot prove non-theorems. The difference between the two provers is in the way they reason. The first one adopts a sound variant of the tableau calculus to produce proofs, while the second prover adopts a non sound variant, more performant, whose results have to be checked in order to discriminate good and bad answers, and to guarantee trustworthiness.

The key idea can be synthesized in the slogan validation by translation. We assume that the ISABELLE object logic we are working on, is correct. The idea is that, whenever a prover produces a proof, it can be translated into a sequence of basic inference steps in the ISABELLE object logic, and then, executing those steps, we are able to certify the validity of the result the prover provided.

We will show an abstract algorithm to translate closed tableaus for intuitionistic first-order predicate logic into Prawitz's natural deduction calculus (Prawitz, 1965) for the same logic. More, we will introduce the E logical system, and we will describe an analogous tableau prover, based on the same certification technique, for that calculus.

The translation of proofs from tableaus into natural deduction is also important for another reason; we want to extract information from proofs. As we remarked in Chapter 2, we want to analyze correctness proofs. In Chapter 6 we will show the method we will adopt, and in Chapter 8, we will discuss its properties. For the moment being, we must say that the analysis technique has been developed for
natural deduction calculi, hence the importance of a translation algorithm.

In Chapter 8, we will prove that the tableau calculus for the $E$ logic is equivalent to its natural deduction presentation; this result is important with respect to the Constructive Reasoner because it proves that the only automatic tool we have in the pure Constructive Verification Environment to prove theorems in the pure logic, is powerful enough.

Originally, the translation problem was posed in (Wos, 1988) for resolution-based theorem provers. As P. Andrews said in (Andrews, 1991),

Substantial work has been done not only on such translations, but also on improving the structure of natural deduction proofs and translating them into natural language.

In fact, there are many other approaches which could resemble ours, but they are different in the choice of the starting calculus, as described in (Bibel et al., 1996; Buffoli, 1992; Kreitz et al., 1995; Pfenning, 1987; Sahlin et al., 1992) and in (Schmitt and Kreitz, 1995; Schmitt and Kreitz, 1996; Schmitt and Kreitz, 1998) where the authors have developed an uniform procedure for transforming classical and non classical matrix proofs into sequent style proofs. Moreover, a closely related topic is the comparison between tableau systems and resolution-based methods, see, e.g., (Ophelders and De Swart, 1993).

The distinctive character of our approach lies in the way we use the translation algorithm. Essentially, we do not require the starting tableau calculus to be sound, and we use the translation algorithm to certify the answers from an automatic tableau prover; where the answer no of the prover, when the input is, indeed, a theorem, can occur either because the implementation of the calculus is incorrect or because, to improve performances, the calculus is not sound.

Let us suppose to have an environment $E$ where we can develop proofs step by step in natural deduction, and let us suppose that this environment is correct. Now, let $P$ be an automatic theorem prover, which, given a formula $\phi$, returns $False$ if it is not able to prove $\phi$, and $True$ plus a tableau $T$ for $\phi$ if it is able to construct $T$. We do not assume that $P$ is correct, i.e., that whenever $P$ returns $True$ and $T$, $\phi$ is a theorem and $T$ is a proof for $\phi$. If $Tr$ is a translation algorithm, because it is formally correct, $Tr(T)$ is a proof in $E$ if and only if $T$ is correct, i.e., if $\phi$ is a theorem. In this way the system $E + P + Tr$ provides reliable answers, even if $P$ may be non-correct.

Later, we will show that it makes sense to use provers which are not correct on purpose, so we can avoid significant computational costs without restricting the set of provable formulas.

In the implementations, the translation algorithm constructs a tactic, i.e., a function which sequentially applies elementary inference rules; in this way, even a non-closed tableau can be converted into a partial proof. The correctness of this
variant of the translation algorithm follows from the propositions we will derive in Section 3.3.

As a minor remark, we should note that the embedding technique, see, e.g., (Harrison, 1995; Harrison, 1996), cannot be applied in our case; to embed a logical calculus, IL-T in our case, into another calculus, IL in our case, means to derive in the second calculus, IL, suitable inference rules which simulate the behavior of the first calculus, that is IL-T. But, being IL sound, it is not possible to embed the non sound variant of IL-T, hence we have not to compare our approach to the embedding technique.

We choose the tableau calculus developed in (Miglioli et al., 1997). Our choice is motivated by the problem of simplifying the search for proofs by reducing the amount of duplications, where a duplication occurs in a proof of a tableau calculus whenever a formula already used by an inference rule is used again by the same rule (for a comprehensive discussion about duplications, see (Miglioli et al., 1997)).

A quite similar problem has been taken into account on the side of sequent calculi, where the counterpart of duplication is the elimination of contractions. There, in the intuitionistic framework important results have been obtained by Dyckhoff (Dyckhoff, 1992) and, independently, by Hudelmaier (Hudelmaier, 1993), who have exhibited cut-free and contraction-free sequent calculi for Intuitionistic Propositional Logic, where a sequent calculus is contraction-free if no formula occurring in the lower sequent of an inference rule can occur in some of the upper sequents; in (Dyckhoff, 1992) also a contraction-free natural calculus is given, while in (Hudelmaier, 1993) it is shown that the involved calculus gives rise to an $O(n \log n)$-space decision procedure for Intuitionistic Propositional Logic.

The tableau calculus of (Miglioli et al., 1997) (which is a refinement of those in (Miglioli et al., 1994b; Miglioli et al., 1994c)) has improved Fitting's tableau calculus for Intuitionistic Predicate Logic (Fitting, 1969) by reducing the amount of duplications involved in its proofs.

To do so, in (Miglioli et al., 1997) the $F_c$-signed formulas have been introduced near the $F$-signed and the $T$-signed formulas of Fitting's calculus (see the discussion in (Miglioli et al., 1994c)), and the rules for implication have been refined, taking into account the ideas in Dyckhoff's paper.

Thus, the tableau calculus for Intuitionistic Predicate Logic in (Miglioli et al., 1997), is nearly optimal from the point of view of the elimination of duplications (such a calculus completely avoids duplications in the propositional framework, thus providing, for tableau calculi, a result comparable with Dyckhoff's and Hudelmaier's; on the other hand, at the predicate level, not all duplications, or mutatis mutandis, not all contractions, can be cut off).

Moreover, we have modified the tableau calculus in (Miglioli et al., 1997) in a way that the tableau rule connected with the quantifiers are not sound, since they do not handle in the proper way the binding of variables (it is a problem very close to Prolog implementations which avoid the occur-check (Lloyd, 1984)).

This variation of our tableau calculus is more efficient, since it does not require so many duplications as the sound calculus, but, of course, it may produce closed
tableaus which are not valid \textbf{IL-T} proof tables.

Our approach is to validate a closed tableau by translating it into an \textbf{IL} proof. If this is possible, we have been able to prove in an more efficient way a, possibly difficult, theorem; if this is not possible, it means that we applied in a non-sound way an expansion rule, so we are not able to judge whether the goal is a theorem.

Regarding the \textbf{E} calculus, we give a tableau version for it, \textbf{E-T}, proving its equivalence with the \textbf{E} system. We will prove in Chapter 8 that the tableau calculus \textbf{E-T} is equivalent to \textbf{E}. The \textbf{E-T} calculus is duplication free on the propositional fragment. So the situation is absolutely similar to the intuitionistic version of the Constructive Reasoner. In fact, the skeleton of the implementation is shared between the two versions of the package.

\section{The \textbf{IL} and \textbf{E} Systems}

In this section we are going to define the \textbf{IL} system and the \textbf{E} system. A natural like deduction system for each of them is shown.

The set of \textit{well formed formulas} (wffs for short) for intuitionistic first order logic is defined in the usual way, starting from the propositional connectives $\neg, \land, \lor, \rightarrow$, the quantifiers $\forall$ and $\exists$, a denumerable set $\mathcal{V}$ of individual variables, and, for any natural number $n$, a denumerable set $\mathcal{P}^n$ of $n$-ary predicate variables.

The theory \textbf{IL} denotes the set of all intuitionistic valid wffs. Our choice for a natural deduction system is the calculus Ni as shown in (Schwichtenberg and Troelstra, 1996); for reference, we report in Table 3.1 the complete set of inference rules. Since in (Troelstra, 1973b), the \textbf{Ni} calculus is proven to be sound and complete, we will refer to it as \textbf{IL}.

Similarly, the set of wffs for the \textbf{E} logic\footnote{Normally, when it is clear from the context, we will not differentiate between names for \textbf{IL} and \textbf{E}; when it is necessary to distinguish, we will prepend the identifier of the logic, like \textbf{IL}-wffs and \textbf{E}-wffs.} is defined starting from the same alphabet as for \textbf{IL}, plus a unary connective, $\Box$, whose intended meaning is to denote classical truth: $\Box A$ is true if $A$ is classically true. A natural like deduction system for the \textbf{E} logic is presented in (Miglioli et al., 1989b), and, for reference, we report in Table 3.2 the set of its inference rules.

\subsection{The \textbf{IL-T} and \textbf{E-T} Tableau Calculi}

The tableau calculus for the \textbf{IL-T} system we are going to use is defined in (Miglioli et al., 1994b; Miglioli et al., 1994c; Miglioli et al., 1997) and it uses the three signs $T$, $F$ and $F_c$. Given a wff $A$, a \textit{signed well formed formula} (swff for short) will be every expression of the kind $\mathcal{S} A$, where $\mathcal{S} \in \{ T, F, F_c \}$. By a \textit{configuration} we mean a finite sequence $S_1 | S_2 | \cdots | S_n$ (with $n \geq 1$), where every $S_j$ is a set of swffs. A set of swffs in a configuration is called \textit{node}. 

\textbf{IL-T} and \textbf{E-T} are defined with the \textit{universal elimination rule} as follows:

\begin{itemize}
  \item $T$ is \textit{true}.
  \item $T$ is the only \textit{axiom}.
  \item \textit{Universal elimination}.
\end{itemize}
### 3.1. THE **IL** AND **E** SYSTEMS

\[
\begin{array}{cccc}
A & B & \land \l 1 & A \land B \\
\hline
A \land B & \land E_i & A \land B & \land E_i \\
\hline
A & \top \l i & A \lor B \\
\hline
[\l A] & [\l B] & \vdots & \vdots \\
\hline
B & \rightarrow \l i & A \rightarrow B \\
\hline
A \rightarrow B & \rightarrow E & \bot \rightarrow E \\
\hline
A(p) & \forall x. A(x) & \forall x. A(x) \\
\hline
\forall x. A(x) & \forall E & A(t) \\
\hline
\exists x. A(x) & \exists E & B \\
\hline
\end{array}
\]

where, in (\textit{*}) and (\textit{**}), \(p\) is an eigenvariable.

**Table 3.1: Inference Rules for the First-Order Intuitionistic Logic **IL**.**

The intuitionistic tableau calculus **IL**-T is given by the rules in Table 3.3, where \(S_c\) is the certain part of \(S\), formally,

\[
S_c = \{TX \mid TX \in S\} \cup \{F_cX \mid F_cX \in S\}.
\]

**Definition 3.1.1** Let \(S\) be a set of suffixs, a tableau (or **IL** -proof-table) for \(S\) is a finite sequence of configurations \(C_1, \ldots, C_n\) such that

- \(C_1 = \{S\}\), and
- \(C_{i+1} = \{S_1, \ldots, S_n\} \cup \{N_1, \ldots, N_k\}\), where \(C_i = \{S_1, \ldots, S_n\} \cup \{M\}\) and

\[
\begin{array}{c}
\hline
M \\
\hline
N_1 \mid \ldots \mid N_k \\
\hline
\end{array}
\]

is an instance of an expansion rule as in Table 3.3.

The active wff in a node is the wff an expansion rule is applied to, the other wffs in that node are called the context. A node in the last configuration (the terminal configuration) of a tableau is called a terminal node.

A tableau is closed iff all the sets \(S_j\) of its final configuration are contradictory, where a set \(S\) is contradictory if, for some \(A\), either \(\{TA, FA\} \subseteq S\) or \(\{TA, F_cA\} \subseteq S\).
where, in (*), \( p \) is an eigenvariable, and in (**), \( P \) is any atomic formula.

Table 3.2: Inference Rules for the \( \mathbf{E} \) Logic.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, T(A \land B)$</td>
<td>$S, T(A \land B)$</td>
</tr>
<tr>
<td>$S, T(A \lor B)$</td>
<td>$S, T(A \lor B)$</td>
</tr>
<tr>
<td>$S, T(\neg A)$</td>
<td>$S, T(\neg A)$</td>
</tr>
<tr>
<td>$S, T(\forall x. A(x))$</td>
<td>$S, T(\forall x. A(x))$</td>
</tr>
<tr>
<td>$S, T(\exists x. A(x))$</td>
<td>$S, T(\exists x. A(x))$</td>
</tr>
<tr>
<td>$S, T(A \rightarrow B)$</td>
<td>$S, T(A \rightarrow B)$</td>
</tr>
<tr>
<td>$S, T((A \land B) \rightarrow C)$</td>
<td>$S, T((A \land B) \rightarrow C)$</td>
</tr>
<tr>
<td>$S, T((A \lor B) \rightarrow C)$</td>
<td>$S, T((A \lor B) \rightarrow C)$</td>
</tr>
<tr>
<td>$S, T((\forall x. A(x)) \rightarrow B)$</td>
<td>$S, T((\forall x. A(x)) \rightarrow B)$</td>
</tr>
<tr>
<td>$S, T((\exists x. A(x)) \rightarrow B)$</td>
<td>$S, T((\exists x. A(x)) \rightarrow B)$</td>
</tr>
</tbody>
</table>

Table 3.3: Expansion Rules for the IL-T Tableau Calculus.
Accordingly, we call **complementary pair** any set of swfs of the form \( \{ TA, FA \} \) or of the form \( \{ TA, FcA \} \). A proof of a wff \( B \) in \( \textbf{IL}-T \) is a closed tableau whose initial node is \( \{ FB \} \). Finally, we say that a set of swfs \( S \) is **\( \textbf{IL} \)-consistent** iff no tableau starting from \( S \) is closed.

Every rule of the \( \textbf{IL}-T \) tableau calculus is applied to a swff of a set \( S_i \) occurring in a configuration \( S_1 \mid \ldots \mid S_i \mid \ldots \); the notation \( S, T(A \land B) \) points out that the rule \( T \land \) is applied to the swff \( T(A \land B) \) of the set \( S \cup \{ T(A \land B) \} \), where \( S \) is possibly empty. We remark that all the rules of the \( \textbf{IL}-T \) calculus, excepting \( T \lor, Fexists, Fcexists, Fc\forall \) and \( T \rightarrow \forall \), are duplication-free, in the sense explained in (Avellone et al., 1997a; Avellone et al., 1997c; Dyckhoff, 1992; Miglioli et al., 1994b; Miglioli et al., 1994c; Miglioli et al., 1997).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S, T(A \land B) ) ( \frac{S, TA, TB}{T \land} )</td>
<td>( S, F(A \land B) ) ( \frac{S, FA, FB</td>
</tr>
<tr>
<td>( S, T(A \lor B) ) ( \frac{S, TA</td>
<td>S, TB}{T \lor} )</td>
</tr>
<tr>
<td>( S, T(\neg A) ) ( \frac{S, F(\neg A)}{T \neg} )</td>
<td>( S, Fc(\neg A) ) ( \frac{S, FcA}{Fc \neg} )</td>
</tr>
<tr>
<td>( S, T(\Box A) ) ( \frac{S, F(\Box A)}{T \Box} )</td>
<td>( S, Fc(\Box A) ) ( \frac{S, FcA}{Fc \Box} )</td>
</tr>
<tr>
<td>( S ) ( \frac{S, F(A \rightarrow B)}{\text{special}} )</td>
<td>( S, Fc(A \rightarrow B) ) ( \frac{S, TA, FcB}{Fc \rightarrow} )</td>
</tr>
</tbody>
</table>

Table 3.4: Expansion Rules for the \( \textbf{E}-T \) Tableau Calculus: Propositional Part.

The tableau calculus for the \( \textbf{E} \) logic is shown in Tables 3.4, 3.5, 3.6 and 3.7. It uses five signs, \( T, F, Fc, TBox \) and \( TDiamond \). The notation is, mutatis mutandis, the same we adopted for \( \textbf{IL}-T \), so we describe just the relevant notions.

The **certain part** of \( S \), with \( S \) a set of swfs, is

\[
S_c \equiv \{ TX \mid TX \in S \} \cup \{ FcX \mid FcX \in S \} \cup \\
\{ TBoxX \mid TBoxX \in S \} \cup \{ FBoxX \mid FBoxX \in S \} ;
\]

the **classical part** of \( S \) is

\[
S_0 \equiv \{ TBoxX \mid TBoxX \in S \} \cup \{ FBoxX \mid FBoxX \in S \} \cup \\
\{ TBoxX \mid TBoxX \in S \} \cup \{ FBoxX \mid FBoxX \in S \} .
\]

A tableau for \( \textbf{E} \), or \( \textbf{E} \)-proof-table, is defined from the expansion rules in Tables 3.4, 3.5, 3.6 and 3.7, in the same way as \( \textbf{IL} \) proof tables.
3.1.  **THE IL AND E SYSTEMS**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, T(\forall x. A(x))$</td>
<td>$S, T(\forall x. A(x) \rightarrow B)$</td>
</tr>
<tr>
<td>$S_c, T A(a), T(\forall x. A(x))$</td>
<td>$S, F(\forall x. A(x))$, $T((\forall x. A(x)) \rightarrow B) \mid S, F(\forall x. A(x)) \mid T B$</td>
</tr>
<tr>
<td>$S, F_c(\forall x. A(x))$</td>
<td>$S, T(\exists x. A(x))$</td>
</tr>
<tr>
<td>$S, F_c A(a)$</td>
<td>$S, T((\exists x. A(x)) \rightarrow B)$</td>
</tr>
<tr>
<td>with a new</td>
<td>$S, T(\exists x. A(x))$</td>
</tr>
<tr>
<td>$S, T((\exists x. A(x)) \rightarrow B)$</td>
<td>$S, T((\forall x. \neg A(x)) \rightarrow B)$</td>
</tr>
<tr>
<td>$S, T(\forall x. (A(x) \rightarrow B))$</td>
<td>$S, T(\exists x. (A(x) \rightarrow B))$</td>
</tr>
<tr>
<td>$S, T(\forall x. (A(x) \rightarrow B))$</td>
<td>$S, F_c(\forall x. A(x))$</td>
</tr>
<tr>
<td>$S, T(\forall x. (A(x) \rightarrow B))$</td>
<td>$S, F_c A(a), T(\forall x. A(x))$</td>
</tr>
<tr>
<td>$S, T(\forall x. A(x))$</td>
<td>$S, F_c A(a)$</td>
</tr>
<tr>
<td>$S, T(\exists x. A(x))$</td>
<td>$S, F_c A(a)$</td>
</tr>
<tr>
<td>$S, T(\exists x. A(x))$</td>
<td>$S, F_c A(a), F_c(\exists x. A(x))$</td>
</tr>
</tbody>
</table>

Table 3.5: Expansion Rules for the E-T Tableau Calculus: Predicative Part.
\[
\begin{align*}
S, T \models (A \land B) & \quad S, F \models (A \land B) \\
S, T \models A \quad S, F \models A & \quad S, T \models B \quad S, F \models B \\
S, T \models (A \lor B) & \quad S, F \models (A \lor B) \\
S, T \models \neg A & \quad S, F \models \neg A \\
S, T \models \Box A & \quad S, F \models \Box A \\
S, T \models (A \rightarrow B) & \quad S, F \models (A \rightarrow B) \\
\end{align*}
\]

Table 3.6: Expansion Rules for the $\mathbf{E}$-$\mathbf{T}$ Tableau Calculus: Classical Part.

\[
\begin{align*}
S, T \models (A \rightarrow B) & \quad S, T \models (\neg A \rightarrow B) \\
S, F \models A \quad S, F, \neg A \models A \quad S, T \models B & \quad S, F \models A \quad S, F, \neg A \models B \\
S, T \models (A \land B \rightarrow C) & \quad S, T \models ((A \land B) \rightarrow C) \\
S, T \models (A \rightarrow (B \rightarrow C)) & \quad S, T \models ((A \rightarrow B) \rightarrow C) \\
S, T \models (A \lor B \rightarrow C) & \quad S, T \models ((A \lor B) \rightarrow C) \\
S, T \models (\Box A \rightarrow B) & \quad S, T \models ((\Box A) \rightarrow B) \\
S, T \models (\neg A \rightarrow B) & \quad S, T \models ((\neg A) \rightarrow B) \\
S, T \models (A \rightarrow A) & \quad S, T \models ((A \rightarrow A) \rightarrow C) \\
S, T \models (A \land \neg B \rightarrow C) & \quad S, T \models ((A \land \neg B) \rightarrow C) \\
S, T \models (A \rightarrow (B \rightarrow C)) & \quad S, T \models ((A \rightarrow B) \rightarrow C) \\
S, T \models ((A \rightarrow B) \rightarrow C) & \quad S, T \models ((A \land \neg B) \rightarrow C) \\
\end{align*}
\]

Table 3.7: Expansion Rules for the $\mathbf{E}$-$\mathbf{T}$ Tableau Calculus: Implicational Part.
3.2. *The Constructive Reasoner*

A set $S$ of swfs is contradictory if, for some $A$, $\{TA, FA\} \subseteq S$ or $\{TA, F_eA\} \subseteq S$ or $\{TA, F_{\Box}A\} \subseteq S$ or $\{FA, F_eA\} \subseteq S$ or $\{FA, T\Box A\} \subseteq S$ or $\{FA, F_{\Box}A\} \subseteq S$ or $\{F_eA, T\Box A\} \subseteq S$ or $\{T\Box A, F_{\Box}A\} \subseteq S$. The definitions of complementary pair and closed tableau are expanded accordingly.

We say that an $E$-$T$ tableau $T$ proves $A$ iff $T$ is closed and its starting configuration is $FA | F_{\Box}A$.

### 3.2 The Constructive Reasoner

In this section, we illustrate how a reasoner adopting the IL-$T$ and the E-$T$ calculus, was implemented in *Isabelle*. Actually, we have implemented two different variants of the same prover. In fact, we have four provers: two for each logic, and two for each algorithm. The implementation provides two tactics, one for each algorithm, parametrized by the inference rules.

The tactics differ in the way they use inference rules; the first prover (let us call it $\text{T}ab$) adopts a depth-first strategy, with loop detection and heuristic ordering of candidates; it is sound for predicate logic, and complete on the propositional fragment. The second prover (we name it $\text{UnTab}$) roughly adopts the same strategy, but it is not sound.

We start describing $\text{T}ab$; it takes a goal $G$, and constructs an initial tableau for it. Internally a tableau $T$ is represented as a pair of lists of nodes: the Active ones and the Dead ones. If a node is in the active list, it means that it is a terminal node of $T$; if a node is in the dead list, then it is a terminal closed node of $T$.

The strategy is depth-first; the first active node $N$ of $T$ is chosen. Then we check if it is closed; if so, it is moved in the dead list, and we finish the expansion step. If it is not closed, then we order the swfs in $N$ according to their “safety”; then we apply an inference rule to the first formula in $N$.

We have two kinds of special cases: safe formulas and duplicating formulas. A formula is safe if applying the corresponding expansion rule to a node does not cut closing possibilities. A formula $\phi$ is duplicating, if an application of the corresponding expansion rule does not consume $\phi$.

Safe formulas in IL-$T$ are of the kind $TA, TV, T\neg, T \rightarrow \land, T \rightarrow \lor, T \rightarrow \exists, T\exists, FA, FV$ and $F_eV$; safe formulas in $E$-$T$, according to the sign and to the principal connective, are $TA, TV, T\neg, T\Box, F_e\land, F_e\lor, F_e\rightarrow, F_{\land}, F_{\lor}, F_{\rightarrow}, T_{\land}, T_{\lor}, F_{\land}, F_{\lor}, T \rightarrow \land, T \rightarrow \lor, T \rightarrow \neg, F_e\land, T\exists, T \rightarrow \forall$, and $T \rightarrow \exists\neg$. In IL-$T$, duplicating formulas have the form $T \rightarrow \lor, TV, F_e\lor$ and $F_e\exists$; in E-$T$, duplicating formulas are $T \rightarrow \lor, TV, F_e\exists, T\Box \lor$ and $F_{\Box} \exists$.

After the expansion step, which generates $M_1, \ldots, M_n$ as new nodes, if the expanded formula was not safe, the tableau $T$ is marked for backtracking; if the expanded formula $\phi$ is duplicating, and we have reached the maximum allowed duplication level, then we erase $\phi$ from $M_1, \ldots, M_n$, otherwise we decrement by one the maximum allowed duplication level. Finally, we generate a new tableau, substituting in $T$, the node $N$, with $M_1, \ldots, M_n$. 
There are two exceptional cases: when we have at least an active node, but it cannot be expanded, because it contains just atomic formulas, and it is not closed; when we have no active nodes. The former case means that the tableau will never become a proof, so we have to try other alternatives, by backtracking; of course, if there are no alternatives we have a failure. The latter case means that the tableau is a proof, so the prover stops with a success.

Being \( G \Rightarrow \phi \) the initial goal, in the case of a success, the prover tactic returns to ISABELLE a theorem of the form \( \phi \equiv \top \), which proves the goal. In the case of a failure, there are two possibilities: if no duplicating formulas were ever used, then the prover returns to ISABELLE a theorem of the form \( \phi \equiv \bot \), stating that the goal is unprovable; otherwise it signals using an ML exception that it failed to prove the goal, but its search for a proof was not exhaustive.

Essentially, \texttt{UnTab} adopts the same strategy, but it substantially differs in the choice of instantiations. In \texttt{IL-T}, the problem arises when trying to expand formulas of the kind \( \text{T}_V \), \( \text{F}_3 \) and \( \text{F}_c \); the \( a \) in Table 3.3 stands for any term, so a proper choice has to be made. The instantiating expansion rules for \texttt{Tab} calculate the set of terms in the node the expanding formulas belongs to, and generate an instance for every term. This process amounts to accept a certain amount of local duplications.

The \texttt{UnTab} prover works in a different way: it generates just one instance, substituting a dummy variable for \( a \). A dummy variable is very similar to a metavariable: when checking if a node is closed, \texttt{UnTab} searches for a pair of wff \( \phi \) and \( \psi \) such that they form a complementary pair in the node, and they are unifiable in their dummy variables. If the unification process is successful, the unifier is propagated over the whole tableau. Since dummy variables gets instantiated only when a node becomes closed, it is possible to have soundness problems, like variable capturing. In Section 3.4 an example will be presented.

Obviously, the \texttt{UnTab} prover needs some mechanism which certifies its answers, so to ensure validity of what it says are theorems; on the other hand, when successful, \texttt{UnTab} is quite more efficient than \texttt{Tab}. What follows suggests a possible way to certify answers from a non sound prover. It is more general than our application requires, but the same certification technique is used in most tools developed in the Constructive Verification Environment.

### 3.3 Translating Tableau Proofs into Natural Deduction

In this section, we introduce the algorithm which translates tableaus into natural deduction proofs, and we prove its correctness. To do so, we define the concept of proof with gaps, and we use this notion to define a function \( \text{Tr} \), which maps each tableau into a proof which may contain non-specified parts, the gaps. Finally we show how we can remove these gaps when the original tableau is closed.

**Definition 3.3.1** The intuitionistic natural deduction calculus with gaps \( \text{ILG} \) is
defined from the inference rules in Table 3.1 plus
\[
\frac{\gamma_1 \ldots \gamma_n}{\alpha} \quad G.
\]

Similarly,

**Definition 3.3.2** The E calculus with gaps EG is defined from the inference rules in Table 3.2 plus
\[
\frac{\gamma_1 \ldots \gamma_n}{\alpha} \quad G,
\]
where \(\gamma_1, \ldots, \gamma_n, \alpha\) are suffixes.

A proof in these systems is a generalization of proofs in the \textsc{il} calculus (the E system, respectively); for this reason we will refer to deductions in this system as proofs with gaps (Pg) in \textsc{il} (E).

To indicate gaps into a Pg-proof, we use the notation
\[
\Xi(G_1, \ldots, G_n)
\]
where \(G_1, \ldots, G_n\) are the application of the G rule (the gaps, for short) we want to mark. When we write \(\Xi(\Pi_1, \ldots, \Pi_n)\) we intend that the gaps \(G_1, \ldots, G_n\) are substituted with the Pg-proofs \(\Pi_1, \ldots, \Pi_n\).

The intuitive meaning of a proof with gaps is that it is a “partial” proof, i.e., a proof where some parts (the gaps) are not yet developed, but their assumptions and conclusions are specified. We use a gap in a proof as a placeholder for another proof, which should be developed according to our translation algorithm.

![Figure 3.1: Derived Inference Rules for Translating Tableaus into \textsc{il}: Part I.](image-url)
### Figure 3.2: Derived Inference Rules for Translating Tableaus into IL: Part II.

<table>
<thead>
<tr>
<th>premise set</th>
<th>rule</th>
<th>conclusion set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, [A] ) ( \Gamma, [B] )</td>
<td>( A \lor B ) ( D ) ( \Gamma ) ( D )</td>
<td>( \Gamma ) ( \neg \neg A \lor \neg B ) ( D ) ( \neg \neg A \lor \neg B )</td>
</tr>
<tr>
<td>( \Gamma, [A] ) ( \Gamma, [B] )</td>
<td>( A \lor B ) ( D ) ( \Gamma ) ( D )</td>
<td>( \Gamma ) ( \neg \neg A \lor \neg B ) ( D ) ( \neg \neg A \lor \neg B )</td>
</tr>
</tbody>
</table>

### Figure 3.3: Derived Inference Rules for Translating Tableaus into IL: Part III.

<table>
<thead>
<tr>
<th>premise set</th>
<th>rule</th>
<th>conclusion set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, [A] ) ( \Gamma, [A] )</td>
<td>( \bot ) ( \bot ) ( \bot ) ( \bot )</td>
<td>( \neg \neg A ) ( \bot ) ( \bot ) ( \bot )</td>
</tr>
</tbody>
</table>

### Figure 3.4: Derived Inference Rules for Translating Tableaus into IL: Part IV.

<table>
<thead>
<tr>
<th>premise set</th>
<th>rule</th>
<th>conclusion set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, [A(p)] ) ( \exists x. A(x) ) ( D ) ( \exists x. A(x) )</td>
<td>( \bot ) ( \bot ) ( \bot ) ( \bot )</td>
<td>( \bot ) ( \bot ) ( \bot ) ( \bot )</td>
</tr>
</tbody>
</table>

with \( p \) eigenvariable.

<table>
<thead>
<tr>
<th>premise set</th>
<th>rule</th>
<th>conclusion set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, [A(a)] ), ( \neg \exists x. A(x) ) ( D ) ( \exists x. A(x) )</td>
<td>( \bot ) ( \bot ) ( \bot ) ( \bot )</td>
<td>( \bot ) ( \bot ) ( \bot ) ( \bot )</td>
</tr>
</tbody>
</table>
### 3.3. TRANSLATING TABLEAUX PROOFS INTO NATURAL DEDUCTION

![Inference Rules Table]

#### Figure 3.5: Derived Inference Rules for Translating Tableaux into IL: Part V.

**Definition 3.3.3** The natural deduction calculus **ILGD** is defined from **ILG** plus the rules in Figures 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6. We will refer to these rules as translation rules.

**Definition 3.3.4** The natural like deduction calculus **EGD** is defined from **EG** plus the rules in Figures 3.7, 3.8, 3.9, 3.10, 3.11, and 3.12. We will refer to these rules as translation rules.

In Proposition 3.3.3, we will prove that the rules in Figures 3.1-3.6 can be derived in **ILG**; hence **ILG** and **ILGD** are equivalent, because they prove the same set of wffs (but, from a practical point of view, following (Paulson, 1997d), **ILGD** is, by far, more efficient than **ILG**). The analogous result for the **E** variant is proven in Proposition 3.3.4.

The algorithm $\text{Tr}$ which underlies the application of a translation rule is:

1. If $S$ is the context of the node to which a tableau expansion rule is applied, then $\Gamma_\text{IL} = \{\phi \mid T\phi \in S\} \cup \{\neg\phi \mid F_c\phi \in S\}$ and $D_\text{IL} = \{\phi \mid F\phi \in S\}$; similarly, $\Gamma_\text{E} = \{\phi \mid T\phi \in S\} \cup \{\neg\phi \mid F_c\phi \in S\} \cup \{\square\phi \mid T\square\phi \in S\} \cup \{\neg\square\phi \mid F_c\square\phi \in S\}$ and $D_\text{E} = \{\phi \mid \neg\phi \mid F\phi \in S\}$; in the expansion rules appears $\Gamma_\Box = \{\square\phi \mid T\phi \in S\} \cup \{\neg\phi \mid F_c\phi \in S\} \cup \{\square\phi \mid T\square\phi \in S\} \cup \{\neg\square\phi \mid F_c\square\phi \in S\}$ which is easily derived from $\Gamma$ in a canonical way, applying the *special* translation rules.

2. In general, a tableau expansion rule is mapped into the translation rule with the same name, unless
   
   (a) there is no corresponding translation rule (this happens in **IL-T** for the FV and the T¬ rules); in this case, the proof with gaps is left unchanged;
where $P$ is atomic or negated

**Figure 3.6:** Derived Inference Rules for Translating Tableaus into $\mathbf{IL}$: Part VI.
Figure 3.7: Derived Inference Rules for Translating Tableaus into $\mathbf{E}$: Part I.
Figure 3.8: Derived Inference Rules for Translating Tableaus into E: Part II.

<table>
<thead>
<tr>
<th>Γ, [□A]</th>
<th>Γ, [□¬A]</th>
</tr>
</thead>
<tbody>
<tr>
<td>□A ⊥ T□</td>
<td>¬□A ⊥ F□</td>
</tr>
<tr>
<td>□A ⊥ F□</td>
<td>□¬A ⊥ F□</td>
</tr>
<tr>
<td>□□A D T□△</td>
<td>□¬A D F□△</td>
</tr>
<tr>
<td>□¬A D F□△</td>
<td>□□A D T□△</td>
</tr>
<tr>
<td>□¬A special</td>
<td>□¬A special</td>
</tr>
<tr>
<td>□¬A special</td>
<td>□¬A special</td>
</tr>
<tr>
<td>□¬A special</td>
<td>□¬A special</td>
</tr>
<tr>
<td>□¬A G D special</td>
<td>□¬A G D special</td>
</tr>
</tbody>
</table>

Figure 3.9: Derived Inference Rules for Translating Tableaus into E: Part III.

<table>
<thead>
<tr>
<th>Γ, [A]</th>
<th>Γ, [A], [¬B]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B ⊥ G</td>
<td>□¬B ⊥ G</td>
</tr>
<tr>
<td>((A → B) ⊥ G (¬(A → B)) ⊥ D F→</td>
<td></td>
</tr>
<tr>
<td>□(A → B) ⊥ G</td>
<td>□¬(A → B) ⊥ G</td>
</tr>
<tr>
<td>□¬(A → B) ⊥ G</td>
<td>□¬(A → B) ⊥ G</td>
</tr>
<tr>
<td>□¬(A → B) ⊥ G</td>
<td>□¬(A → B) ⊥ G</td>
</tr>
<tr>
<td>□¬(A → B) D F→</td>
<td>□¬(A → B) D F→</td>
</tr>
<tr>
<td>□¬(A → B) D F→</td>
<td>□¬(A → B) D F→</td>
</tr>
</tbody>
</table>
where $P$ is an atomic formula

Figure 3.10: Derived Inference Rules for Translating Tableaus into E: Part IV.
where \( p \) has to be considered an eigenvariable.

Figure 3.11: Derived Inference Rules for Translating Tableaus into \( E \): Part V.
where \( p \) has to be considered an eigenvariable.

Figure 3.12: Derived Inference Rules for Translating Tableaus into \( \textbf{E} \): Part VI.
(b) there is a \( \text{no} \lor \) or a \( \perp \) variant (e.g. \( F \land \text{no} \lor \) and \( F \land \perp \) in IL-T) and the context of the node to which the tableau rule is applied contains no \( F \)-wffs; in this case the variant rule is used.

3. When applying a translation rule, any antecedent which is not an instance of the \( G \) rule, is discharged, since it is an assumption of the gap we are filling.

We can extend \( \text{Tr} \) to tableaus as follows:

**Definition 3.3.5** The translation function \( \text{Tr} \) maps a tableau \( T \) into a proof \( \text{Tr}(T) \) in \( \text{ILGD} \); the definition is given by induction on the structure of the tableau \( T \):

- Let \( T = \{S\} \) be a tableau consisting of the initial configuration; then

\[
\text{Tr}(T) = \frac{\{\phi \mid T \phi \in S\} \cup \{\neg \phi \mid F \phi \in S\}}{\sqrt{\{\phi \mid F \phi \in S\}}} \quad \text{G}
\]

- Let \( C = \{S_1, \ldots, S_n\} \) be the last configuration of \( T \), obtained from a tableau \( T' \) and its terminal configuration \( C' = \{S'_1, \ldots, S'_m\} \) by applying a tableau rule to the suffix \( \alpha \) in the node \( S'_j = \Gamma \cup \{\alpha\} \), and generating as new nodes \( N_1, \ldots, N_k \).

Let \( \text{Tr}(T') = \Xi(G_{S'_j}) \), where \( G_{S'_j} \) is the gap corresponding to \( S'_j \). Then \( \text{Tr}(T) = \Xi(\text{Tr}(G_{S'_j})) \) and \( \text{Tr}(G_{S'_j}) \) is an application of a translation rule, according to the principal connective, the sign of \( \alpha \), and the number of \( F \)-wffs in \( S'_j \).

We observe that every node in the terminal configuration of \( T \) has a corresponding gap in \( \text{Tr}(T) \), and this correspondence defines what we intend for gap associated with a node.

In the case of the \( E \) system, the definition of \( \text{Tr} \) is similar:

**Definition 3.3.6** The translation function \( \text{Tr} \) maps a tableau \( T \) into a proof \( \text{Tr}(T) \) in \( \text{EGD} \); the definition is given by induction on the structure of the tableau \( T \):

- Let \( T = \{S\} \) be a tableau consisting of the initial configuration; calling

\[
\Gamma = \{\phi \mid T \phi \in S\} \cup \{\neg \phi \mid F \phi \in S\} \cup \{\Box \phi \mid T \Box \phi \in S\} \cup \{\Box \neg \phi \mid F \Box \phi \in S\}
\]

we define

\[
\text{Tr}(T) = \frac{\Gamma}{\sqrt{\{\phi \lor \neg \phi \mid F \phi \in S\}}} \quad \text{G}
\]

- Let \( C = \{S_1, \ldots, S_n\} \) be the last configuration of \( T \), obtained from a tableau \( T' \) and its terminal configuration \( C' = \{S'_1, \ldots, S'_m\} \) by applying a tableau rule to the suffix \( \alpha \) in the node \( S'_j = \Gamma \cup \{\alpha\} \), and generating as new nodes \( N_1, \ldots, N_k \).

Let \( \text{Tr}(T') = \Xi(G_{S'_j}) \), where \( G_{S'_j} \) is the gap corresponding to \( S'_j \). Then \( \text{Tr}(T) = \Xi(\text{Tr}(G_{S'_j})) \) and \( \text{Tr}(G_{S'_j}) \) is an application of a translation rule, according to the principal connective and the sign of \( \alpha \).
The translation function \( \text{Tr} \) can be directly mapped into a tactic in \textsc{Isabelle}, which acts by performing a deterministic sequence of inference steps, each one being an application of a translation rule. In the following, we prove that every \( \text{IL} \)-translation rule can be written as a derived rule in \( \text{IL} \), and that every \( \text{E} \)-translation rule can be written as a derived rule in \( \text{E} \); also, we prove that, whenever the \text{Tab} prover succeeds, we can translate the resulting tableau into a natural deduction proof. Moreover, we can reuse the same result for the \text{UnTab} prover, thus providing a way to certify its answers.

**Proposition 3.3.1** Let \( T \) be an \( \text{IL} \)-tableau; for any gap \( G \) in \( \text{Tr}(T) \), there is a node \( S \) in the terminal configuration of \( T \) such that \( G \) is associated with \( S \) and

\[
G \equiv \frac{\{\phi \mid T\phi \in S\} \cup \{-\phi \mid F_e\phi \in S\}}{\forall \phi \mid F\phi \in S}^G.
\]

**Proof:** By induction on the structure of the tableau \( T \). The base case, \( T = \{S\} \), is trivial. So let’s suppose \( T \) is derived from \( T' \) by expanding a terminal node \( S' \) according to the rules in Table 3.3 and let \( \text{Tr}(T') = \Xi(G_{S'}) \).

By induction hypothesis on \( T' \), for every gap \( G \) in \( \text{Tr}(T') \), there exists \( S \), terminal node in \( T' \) associated with \( G \), such that

\[
G \equiv \frac{\{\phi \mid T\phi \in S\} \cup \{-\phi \mid F_e\phi \in S\}}{\forall \phi \mid F\phi \in S}^G.
\]

By definition, \( \text{Tr}(T) = \Xi(\text{Tr}(G_{S'})) \), i.e., \( \text{Tr}(T) \) is obtained from \( \Xi(G_{S'}) \), by replacing the G-rule corresponding to the gap \( G_{S'} \) according to the translation rules. Thus, it suffices to prove that the number of gaps introduced by \( \text{Tr}(\text{Tr}(G_{S'})) \) is exactly equal to the splittings introduced by the tableau rule applied to \( S' \), and that the new gaps have the required shape.

The proof is by cases according to the translation rules. For the sake of brevity, we only treat the cases corresponding to the translation rules \( \text{T} \wedge, F_c \wedge, \) and \( F_c \land \perp \); the other cases are similar.

- Let \( S' = \{TA_1, \ldots, TA_n, FB_1, \ldots, FB_m, F_c C_1, \ldots, F_c C_k, T(H \land E)\} \) where the last configuration of \( T \) is obtained by expanding the swff \( T(H \land E) \) and let \( G_{S'} \) be equal to

\[
A_1, \ldots, A_n, \neg C_1, \ldots, \neg C_k, (H \land E) \quad \frac{B_1 \lor \ldots \lor B_m}{G_{S'}}.
\]

Then \( \text{Tr}(G_{S'}) \), corresponding to the \( \text{T} \wedge \) translation rule, is the following:

\[
A_1, \ldots, A_n, \neg C_1, \ldots, \neg C_k, H, E \quad \frac{H \land E}{B_1 \lor \ldots \lor B_m}^{G^*} \quad \frac{B_1 \lor \ldots \lor B_m}{T \land}.
\]
On the other hand, the tableau rule $T \land$ transforms the set $S'$ into the set 
$S = \{TA_1, \ldots, TA_n, FB_1, \ldots, FB_m, F_cC_1, \ldots, F_cC_k, TH, TE\}$. Hence, the gap $G'$ is associated with the set of swffs $S$ and it has the required shape.

- Let $S' = \{TA_1, \ldots, TA_n, FB_1, \ldots, FB_m, F_cC_1, \ldots, F_cC_k, F_c(H \land E)\}$, and 
  $\Gamma = \{A_1, \ldots, A_n\}$, $\Theta = \{\neg C_1, \ldots, \neg C_k\}$, and $D \equiv B_1 \lor \ldots \lor B_m$ where the last configuration of $T$ is obtained by expanding the swff $F_c(H \land E)$. Moreover, let $G_{S'}$ be equal to

$$
\Gamma \cup \Theta \cup \{-(H \land E)\} \rightarrow_{G_{S'}} D
$$

There are two cases, depending if $D$ differs from $\bot$ or not; then $\text{Tr}(G_{S'})$, corresponding to the $F_c \land$ translation rule or to the $F_c \land \bot$ translation rule, respectively, is one of the following:

$$
\begin{array}{c}
\begin{array}{c}
\Gamma \cup \Theta, [H] \rightarrow_{G_1} \bot \\
\Gamma \cup \Theta, [E] \rightarrow_{G_2} \bot
\end{array} \\
\Huge{\rightarrow_{F_c \land}}
\end{array}
$$

if $m \neq 0$)

$$
\begin{array}{c}
\begin{array}{c}
\Gamma \cup \Theta, [H] \rightarrow_{G_1} \bot \\
\Gamma \cup \Theta, [E] \rightarrow_{G_2} \bot
\end{array} \\
\Huge{\rightarrow_{F_c \land \bot}}
\end{array}
$$

if $m = 0$)

On the other hand, the $F_c \land$ rule transforms the set $S'$ into 
$S_1 = \{TA_1, \ldots, TA_n, FB_1, \ldots, FB_m, F_cC_1, \ldots, F_cC_k, F_cH\}$

and into the set 
$S_2 = \{TA_1, \ldots, TA_n, FB_1, \ldots, FB_m, F_cC_1, \ldots, F_cC_k, F_cE\}.$

Hence, the gaps $G_1$ and $G_2$ are associated with the sets of swffs $S_1$ and $S_2$, respectively, and their shapes are as required.

\[\square\]

**Proposition 3.3.2** Let $T$ be an $E$-tableau; for any gap $G$ in $\text{Tr}(T)$, there is a node $S$ in the terminal configuration of $T$ such that $G$ is associated with $S$ and

$$
G \equiv \left\{ \phi \mid T\phi \in S \right\} \cup \left\{ \neg \phi \mid F_c\phi \in S \right\} \cup \left\{ \Box \phi \mid T\phi \in S \right\} \cup \left\{ \Box \neg \phi \mid F_c\phi \in S \right\} \sqrt{\{ \phi \lor \neg \phi \mid F\phi \in S\}} \rightarrow_{G} G.
$$

**Proof:** The proof is absolutely similar to Proposition 3.3.1, hence there is no reason to repeat it here. \[\square\]
Proposition 3.3.3 The IL-translation rules can be derived in ILG.

Proof: For the sake of brevity, we treat only the cases corresponding to the translation rules $\top\land$, $\mathbf{F}_c\land$, and $\mathbf{F}_c\bot$; the other cases are similar. Hence,

\[
\frac{\Gamma, [A], [B]}{\top} \quad \frac{A \land B}{D} \quad \frac{D}{\top}
\]

becomes

\[
\frac{\top}{A \land B \quad A \land B} \quad \frac{\land E}{\top}
\]

\[
\frac{\Gamma, A}{\top} \quad \frac{\Gamma, B}{\top}
\]

Similarly,

\[
\frac{\top}{D \lor A \quad D \lor B} \quad \frac{\lor E}{D \lor (A \land B)}
\]

\[
\frac{\top}{D \lor A \quad D \lor B \quad D \lor (A \land B)} \quad \frac{\lor I}{\top}
\]

\[
\frac{\top}{D \lor (A \land B)} \quad \frac{\top}{D \lor (A \land B) \quad \lor E_2}
\]

The following rule

\[
\frac{\top}{\neg(A \land B) \quad D} \quad \frac{\top}{\neg(A \land B) \quad D} \quad \frac{\top}{\neg(A \land B) \quad D}
\]

is translated as

\[
\frac{\top}{\neg(A \land B) \quad D} \quad \frac{\top}{\neg(A \land B) \quad D} \quad \frac{\top}{\neg(A \land B) \quad D}
\]
Finally, if the conclusion is \( \bot \), as in the following rule

\[
\begin{align*}
\Gamma_1, [\neg A] & \quad \Gamma_1, [\neg B] \\
\neg (A \land B) & \quad \bot \\
\hline \\
\bot & \quad \Gamma_1, \neg A & \quad \Gamma_2, \bot \\
\hline \\
\bot & \quad G_1 \quad G_2 & \quad F_e \land \bot
\end{align*}
\]

we can expand it into a more compact form

\[
\begin{align*}
[D]_1, [E]_2 & \quad \land I \\
D \land E & \quad \neg (A \land B) \\
\bot & \quad \rightarrow E \\
\hline \\
\bot & \quad \rightarrow \neg B & \quad \land I_2 \\
\bot & \quad G_2 \\
\neg A & \quad \land I_1 \\
\bot & \quad G_1
\end{align*}
\]

\[\square\]

**Proposition 3.3.4** The E-translation rules can be derived in EG.

**Proof:** Again, the proof of this Proposition, is essentially the same as Proposition 3.3.3, hence we will not repeat it here. One can look at the soundness proof for E-T for the more interesting cases. \[\square\]

Using Propositions 3.3.1 and 3.3.3 (3.3.2 and 3.3.4, respectively) we obtain immediately the following corollary.

**Corollary 3.3.1** Let \( T \) be a tableau; then \( \text{Tr}(T) \) can be translated in a Pg-proof.

In the previous proposition we proved that every translation rule is derivable in ILG. We note that some rules, e.g., \( F_e \land \bot \), are instances of others, e.g., \( F_e \land \); the reason behind this apparent duplication of rules is that they have a shorter, more "natural" proof. Since the proof of a derived rule becomes part of the proof we generate through \( \text{Tr} \), it makes sense to have simpler and more natural proofs, so to enhance their readability.

We observe that the way of composing gaps in the expansion of a translation rule may not respect the shape of the rule itself, as in the correctness proof of \( F_e \land \). The translation rules are designed in such a way that their applications reflect the content of the corresponding node in the tableau.

Since ILGD can be embedded into ILGD, then, without loss of generality we can suppose that \( \text{Tr} \) is a function mapping every tableau \( T \) into a proof in ILG. And, of course, the same fact holds for EGD and EG.
3.3. TRANSLATING TABLEAU PROOFS INTO NATURAL DEDUCTION  55

**Definition 3.3.7** Let \( T \) be an \( \text{IL} \)-tableau and let \( \text{Tr}(T) = \Xi(G_{S_1}, \ldots, G_{S_n}) \) where all the gaps are put into evidence. The Gap closure of \( \text{Tr}(T) \), \( \Pi_c(\text{Tr}(T)) \) is the \( \text{Pg} \)-

proof obtained from \( \text{Tr}(T) \) by substituting the translation rules with their derivations according to Proposition 3.3.3 and by replacing the gaps \( G_{S_1}, \ldots, G_{S_n} \) with \( \text{Pg} \)-proofs according the following conventions:

1. if \( S_i \) is \( \text{F} \)-closed, then
   \[
   G_{S_i} = \frac{A_1, \ldots, A_n, \neg C_1, \ldots, \neg C_k, D}{B_1 \lor \ldots \lor B_m \lor D} \quad \text{G}
   \]
   is replaced with
   \[
   \frac{D}{B_m \lor D} \quad \text{\( \lor \)-I}
   \]
   \[
   \frac{B_{m-1} \lor B_m \lor D}{B_m \lor D} \quad \text{\( \lor \)-I (if \( m \neq 0 \))}
   \]
   \[
   \frac{B_1 \lor \ldots \lor B_m}{B_1 \lor \ldots \lor B_m} \quad \text{\( \lor \)-I (if \( m = 0 \))}
   \]

2. if \( S_i \) is \( \text{F}_c \)-closed, then
   \[
   G_{S_i} = \frac{A_1, \ldots, A_n, \neg C_1, \ldots, \neg C_k, D, \neg D}{B_1 \lor \ldots \lor B_m} \quad \text{G}
   \]
   is replaced with
   \[
   \frac{D, \neg D}{\bot} \quad \text{\( \rightarrow \)-E (if \( m \neq 0 \))}
   \]
   \[
   \frac{D, \neg D}{\bot} \quad \text{\( \rightarrow \)-E (if \( m = 0 \))}
   \]

3. If \( S_i \) is not closed, then the corresponding gap is left unchanged.

**Definition 3.3.8** Let \( T \) be an \( \text{E} \)-tableau and let \( \text{Tr}(T) = \Xi(G_{S_1}, \ldots, G_{S_n}) \) where all the gaps are put into evidence. The Gap closure of \( \text{Tr}(T) \), \( \Pi_c(\text{Tr}(T)) \) is the \( \text{Pg} \)-

proof obtained from \( \text{Tr}(T) \) by substituting the translation rules with their derivations according to Proposition 3.3.3 and by replacing the gaps \( G_{S_1}, \ldots, G_{S_n} \) with \( \text{Pg} \)-proofs according the following conventions:

1. if \( S_i \) is closed by \( \text{TA}, \text{FA} \), then
   \[
   G_{S_i} = \frac{\Gamma, A}{B_1 \lor \ldots \lor (A \lor \neg A) \lor \ldots \lor B_m} \quad \text{G}
   \]
   is replaced with
   \[
   \frac{A}{A \lor \neg A} \quad \text{\( \lor \)-I}
   \]
   \[
   \frac{A \lor \neg A}{B_1 \lor (A \lor \neg A)} \quad \text{\( \lor \)-I}
   \]
   \[
   \frac{B_1 \lor (A \lor \neg A) \lor \ldots \lor B_m}{B_1 \lor \ldots \lor (A \lor \neg A) \lor \ldots \lor B_m} \quad \text{\( \lor \)-I}
   \]
2. if \( S_i \) is closed by \( T A, F c A \), then
\[
G_{S_i} \equiv \frac{\Gamma, A, \neg A}{B} \quad \text{is replaced with}
\]
\[
\frac{A \quad \neg A}{B} \quad \text{Contr}
\]

3. if \( S_i \) is closed by \( T A, F \Box A \), then
\[
G_{S_i} \equiv \frac{\Gamma, A, \Box \neg A}{B} \quad \text{is replaced with}
\]
\[
\frac{A \quad [\neg A]}{A \quad \text{Contr}} \quad \frac{A \quad [\neg A]}{A \quad \text{Contr}}
\]
\[
\frac{\Box \neg A \quad \neg \Box \neg A}{B} \quad \text{Contr}
\]

4. if \( S_i \) is closed by \( F A, F c A \), then
\[
G_{S_i} \equiv \frac{\Gamma, \neg A}{B_1 \lor \ldots \lor (A \lor \neg A) \lor \ldots \lor B_m} \quad \text{is replaced with}
\]
\[
\frac{\neg A}{A \lor \neg A} \quad \lor I
\]
\[
\frac{B_i \lor (A \lor \neg A)}{A \lor \neg A} \quad \lor I
\]
\[
B_1 \lor \ldots \lor (A \lor \neg A) \ldots \lor B_m
\]

5. if \( S_i \) is closed by \( F c A, T \Box A \), then
\[
G_{S_i} \equiv \frac{\Gamma, \neg A, \Box A}{B} \quad \text{is replaced with}
\]
\[
\frac{[A] \quad \neg A}{A \quad \text{Contr}} \quad \frac{[A] \quad \neg A}{A \quad \text{Contr}}
\]
\[
\frac{\Box A \quad \neg \Box A}{B} \quad \text{Contr}
\]
6. if \( S_i \) is closed by \( T_\square A, F_\square A \), then
\[
G_{S_i} = \frac{\Gamma, \square A, \square \neg A}{B} \quad G
\]
is replaced with
\[
\begin{array}{c}
[A] [\neg A] \quad [A] [\neg A] \\
A \quad \text{Contr} \quad A \quad \text{Contr}
\end{array}
\]
\[
\begin{array}{c}
\square \neg A \\
\neg \square A \quad \text{Contr}
\end{array}
\]
\[
\begin{array}{c}
[A] \\
\neg A \quad \text{Contr}
\end{array}
\]
\[
\begin{array}{c}
\square A \\
\neg A \quad \text{Contr}
\end{array}
\]
\[
\begin{array}{c}
B
\end{array}
\]

7. If \( S_i \) is not closed, then the corresponding gap is left unchanged.

We observe that closing a gap \( G \) in the sense of the previous definitions, may cancel antecedents of \( G \) which are not essential to complete the proof. It is obvious that \( \Pi_c \) is a function from \( \text{Pg} \)-proofs to \( \text{Pg} \)-proofs. Now, we have to prove that, if \( T \) is a closed tableau and \( P \) is the result of translating it into \( \text{IlG} \), then \( \Pi_c(P) \) is, indeed, a proof in the \( \text{Il} \) system, i.e., it contains no gaps; the same fact holds for the case of \( \text{E-T} \).

We would like to remark that in the \( \text{Il} \)-translation rules \( T\forall, F_c\forall, F_c\forall, F_c\exists \) and \( F_c\exists \) the active formula is duplicated in the gap. In fact, according to our tableau calculus \( \text{Il-T} \), the rules \( T\forall, F_c\forall \) and \( F_c\exists \) require duplicating the active formula. In order to maintain the correspondence between terminal nodes and gaps, as shown in Proposition 3.3.1, we must duplicate the active formula in the translation rule. But, in the proof obtained when translating a closed tableau, these assumptions may disappear since, as noted in a previous remark, unneeded assumptions are deleted by the gap closure operation. Similar considerations hold for the case of the \( \text{E} \) system.

**Theorem 3.3.1** Let \( T \) be a closed \( \text{Il} \)-tableau starting from a set \( S \) of suffixs. Then \( \Pi_c(\text{Tr}(T)) \) is an \( \text{Il} \)-proof of
\[
\{ \phi \mid T\phi \in S \} \cup \{ \neg \phi \mid F_c\phi \in S \} \vdash \bigvee \{ \phi \mid F\phi \in S \}
\]

**Proof:** By Propositions 3.3.1 and 3.3.3, the gaps belonging to the \( \text{Pg} \)-proof \( \text{Tr}(T) \) are those associated to the terminal nodes of \( T \). Since \( T \) is closed, all gaps in \( \text{Tr}(T) \) are replaced with proofs without gaps in \( \Pi_c(\text{Tr}(T)) \), hence, \( \Pi_c(\text{Tr}(T)) \) is a \( \text{Pg} \)-proof without gaps, that is, an \( \text{Il} \)-proof. \( \square \)

**Lemma 3.3.1** Let \( T \) be a closed \( \text{E} \)-tableau starting from a set \( S \) of suffixs. Then \( \Pi_c(\text{Tr}(T)) \) is an \( \text{E} \)-proof of
\[
\{ \phi \mid T\phi \in S \} \cup \{ \neg \phi \mid F_c\phi \in S \} \cup \{ \square \phi \mid T_\square \phi \in S \} \cup \{ \square \neg \phi \mid F_\square \phi \in S \} \vdash \bigvee \{ \phi \mid F\phi \in S \},
\]

Proof: By Propositions 3.3.2 and 3.3.4, the gaps belonging to the $P_g$-proof $\text{Tr}(T)$ are those associated to the terminal nodes of $T$. Since $T$ is closed, all gaps in $\text{Tr}(T)$ are replaced with proofs without gaps in $\Pi_e(\text{Tr}(T))$, hence, $\Pi_e(\text{Tr}(T))$ is a $P_g$-proof without gaps, that is, an $E$-proof. □

**Theorem 3.3.2** Let $T$ be a closed $E$-tableau starting from the configuration $FA | F\Box A$. Then $\Pi_e(\text{Tr}(T))$ produces two $E$-proofs which can be composed in a canonical way, to prove $A$.

Proof: From Lemma 3.3.1, applied to the subtableau for $FA$, we know that

$$\vdash A \lor \neg A ;$$

in the same way we obtain a proof for

$$\Box \neg A \vdash \bot ,$$

hence the following is a proof for $A$

\[
\begin{array}{c}
\vdash \neg A \lor \neg A ; \\
\vdash \neg A \lor \neg A \\
\vdash \neg A \\
\vdash \neg A \\
\vdash \bot \\
\vdash A \lor \neg A [A] \\
\vdash A \\
\vdash A
\end{array}
\]

□

3.4 Non-Sound Tableaus

In the previous sections we introduced $\text{IL-T}$, a tableau calculus which is sound and complete, and an algorithm, $\text{Tr}$, which maps $\text{IL-T}$ proofs to natural deduction derivations.

Our claim is that, considering an implementation $P$ of a prover for $\text{IL-T}$, we can certify its answers using an implementation of $\text{Tr}$. The key idea is that we do not need to prove that the prover $P$ and that the translation procedure are correctly implemented, i.e., no formal verification of the system is needed since we can guarantee that its answers are as reliable as the core $\text{ISABELLE}$ inference engine. We use this idea to certify answers from our $\text{UnTab}$ prover.

A natural question which arises at this point is: if our prover is not correct on purpose, because it implements a non-sound variant of $\text{IL-T}$, what happens? Does it make sense at all to study such non-sound variants?
We think that it can be very interesting to code a non-sound theorem prover and to use our translation algorithm to discriminate between good and bad answers. In this respect, we are going to describe a variant of IL-T which uses dummy variables (Ophelders, 1992; Redaeli, 1994); this modification of the original IL-T calculus is not sound in the usual sense, i.e., although not everything is provable, some proofs do not respect the usual semantics of logical operators: for an example, see subsection 3.4.1.

Let Ξ be a denumerable set of symbols, called dummy variables, and consider the tableau calculus IL-TDummy given by the expansion rules in Table 3.3 where T∀, F∃ and Fc∃ are replaced by the following:

\[
\frac{S, T(\forall x. A(x))}{S, T A(\alpha), T(\forall x. A(x))} \quad \text{T∀Dummy}
\]

\[
\frac{S, F(\exists x. A(x))}{S, FA(\alpha)} \quad \text{F∃Dummy}
\]

\[
\frac{S, Fc(\exists x. A(x))}{S, FcA(\alpha), Fc(\exists x. A(x))} \quad \text{Fc∃Dummy}
\]

where \( \alpha \in \Xi \). The rule which detects if a node is closed is: a node \( N \) in a tableau \( T \) is closed if there is a formula \( \phi \) and a formula \( \psi \) such that

- \( F\phi \in N \) or \( Fc\phi \in N \),
- \( T\psi \in N \),
- for a proper substitution \( \sigma \) of dummy variables, \( \sigma(\phi) = \sigma(\psi) \).

So a strategy for developing a tableau in IL-TDummy would be to iterate the following steps:

1. choose an open node \( N \) in the tableau \( T \); if there is no open node, return \( T \) (proof discovered);
2. check if there is an unifiable complementary pair in \( N \);
3. if so, mark \( N \) as closed, and apply the unifier over the whole tableau \( T \);
4. if not, expand \( N \) using a rule;
5. if this is not possible, by backtracking try alternatives; if there are no alternatives, return a failure.
From a very practical point of view, this expansion strategy has some advantages; in fact, we delay the choice for a witness term in the expansion rules which require one. Sometimes (most of the time, from empirical measures) this fact compensate the disadvantage of running an unification algorithm, because we have the "right" witness in the "right" moment instead of generating mostly useless instances, as required in the \texttt{IL-T} calculus. In fact, the usage of dummy variables is very similar to the use of metavariables in \textsc{Isabelle} (Paulson, 1989), although we do not take care to distinguish between free and bound occurrences.

The actual implementation of our prover for \texttt{IL-TDummy} uses an adaption of the standard Prolog unification algorithm (Lloyd, 1984), which, as it is well known, is non correct since it avoids occur-checks. This fact plus that we do not take care to respect eigenvariables during the unification process, lead us to a non sound but reasonably efficient theorem prover, which seems to be able to prove very quickly a great amount of not too complex goals. Being an instrument of a logical framework, the \texttt{IL-TDummy} prover is perfectly adequate for an interactive usage (Paulson, 1997d).

An important remark about using a non-sound calculus regards the translation function \textsc{Tr}. Since an \texttt{IL-TDummy} proof table which respects eigenvariables is a closed \texttt{IL-T} tableau, we are allowed to use \textsc{Tr} as described before to translate tableaus into tactics.

If we generate a new variant $V$ for a tableau calculus $C$ whose $V$-proof tables cannot be interpreted as those of the original calculus $C$, we are forced to define a new translation function $\tau$. In this case we must be able to prove that the $\tau$ function is correct, that is, it maps a correct tableau for $C$ into \texttt{IL}. Using the intermediate result in this proof, on the first side, we can instruct the \textsc{Isabelle} system about the facts which are needed to translate a proof in the $C$ calculus, and, on the other side, we have a general design schema for the code of a translation function. Then, we can code a theorem prover for the $V$ calculus, and, applying to its results the implemented $\tau$ function, we get answers which enjoy the same properties as for \texttt{IL-TDummy} versus \texttt{IL}.

In fact, using the same computational skeleton, we have coded a tactic for \texttt{E-T} which uses dummy variables. We can apply to it the same considerations as for \texttt{IL-Untab}. The conclusion is that, in a sense, the technique of dummy variables is independent from the logical calculus, and the certification technique based on \textsc{Tr} provides a uniform way to check results.

Sometimes we claimed that our implementation, and in particular the non sound prover, is efficient. Of course, as any application of theorem proving, there is no computational complexity measure for the performances of the prover, since it may not terminate.

Our claim is based on a series of experiments, we do not report for the sake of brevity, where we checked our provers, \texttt{Tab} and \texttt{Untab}, in the intuitionistic version, against \texttt{fast.tac} and \texttt{best.tac}, the two main tactics of the Classical Reasoner. Our comparison was restricted to intuitionistically valid formulas. We chose a set of formulas to prove requiring propositional reasoning, predicative reasoning and
3.4. NON-SOUND TABLEAUS

a special subset was formed of formulas which can be proven only not avoiding duplications. The purpose of this choice was to test every aspect of the four tactics, and to compare them on a sensible set of cases.

The result of this testing has been:

- **Tab** proves almost all of the test cases, failing when it exhausts the system resources, mainly memory; **UnTab** fails to prove half of the predicative goals, because it produces tableaus which cannot be translated into natural deduction proofs; **fast_tac** proves most of the goals, but it fails on the ones conceived for duplication; **best_tac** proves some goals, but it fails much more than **fast_tac** since it uses much more memory to compute.

- The slowest and less effective tactic is, by far, **best_tac**; it is interesting since it proves some goals where the other provers fail. The most performant tactic is **Tab**; normally, it is ten times faster than **fast_tac**, and it proves most of the goals, failing on very complex ones; we should note that on theorems requiring duplication, it is rare that it finds a proof, because it examines too many possibilities, and it tends to use too much memory to generate the right instantiations.

- The most interesting tactic is **UnTab**; it does not prove so many goals as **Tab** or **fast_tac**, but when it produces a result, it takes five times less than **Tab**, counting also the checking phase.

The set of goals **UnTab** is able to prove is not a subset of the one of **Tab**; they are able to prove the same set of propositional goals, in, more or less the same time; they substantially differ on predicative goals. The explanation of this result is that **IL-T** is a duplication-free calculus on the propositional fragment, so **Tab** and **UnTab** behaves exactly in the same way. Of course, **UnTab** has to check the tableau it constructs, but, at least in our test cases, this cost seems to be negligible.

On the predicative cases, **UnTab** proves more duplicating goals than **Tab**, using far less resources, hence it appears to be faster.

Unfortunately, we need more experimental investigation, because it is not yet clear how to guess from the format of a goal when **UnTab** will produce a proof.

3.4.1 An Illustrating Example

The formula \((\exists x. P(x) \rightarrow Q) \rightarrow (\forall x. P(x) \rightarrow Q)\) is a theorem in **IL**. With both versions of our prover, we get the same tableau.

We report the version obtained from **UnTab**, where dummy variables are de-
noted by lowercase Greek letters:
\[
\begin{align*}
\text{F}(\exists x. P(x) \rightarrow Q) & \rightarrow ((\forall x. P(x)) \rightarrow Q) \\
\text{F}(\forall x. P(x)) & \rightarrow Q, \text{T} \exists x. P(x) \rightarrow Q \\
\text{F}(\forall x. P(x)) & \rightarrow Q, \text{T} P(t) \rightarrow Q \\
\text{T} P(t) & \rightarrow Q, \text{F} Q, \forall x. P(x) \\
\text{T} P(t) & \rightarrow Q, \text{T} P(\alpha), \text{F} Q, \forall x. P(x) \\
\text{F} P(t), \text{T} P(\alpha), \text{F} Q, \forall x. P(x) | \text{T} Q, \text{T} P(\alpha), \text{F} Q, \forall x. P(x)
\end{align*}
\]
\[
\text{closed with } \alpha \equiv t.
\]

The corresponding proof in natural deduction goes as follows:
\[
\begin{align*}
\forall x. P(x) \\
P(t) \rightarrow [P(t) \rightarrow Q] \\
Q \rightarrow [P(t)] \\
Q \rightarrow [Q] \\
Q \\
\exists x. P(x) \rightarrow Q \\
(\forall x. P(x)) \rightarrow Q \\
(\exists x. P(x) \rightarrow Q) \rightarrow ((\forall x. P(x)) \rightarrow Q)
\end{align*}
\]

The formula $\neg \neg \exists y. \forall x. P(y) \rightarrow P(x)$ is a theorem in $\text{IL}$. Its proof in our tableau system require one duplication.

When we try to prove this goal with our non-sound $\text{UnTab}$ prover, which uses dummy variables, it generates an incorrectly closed tableau. Deleting unnecessary duplicated formulas, the wrong tableau is as follows:
\[
\begin{align*}
\text{F} \rightarrow \exists y. \forall x. P(y) \rightarrow P(x) \\
\text{T} \rightarrow \exists y. \forall x. P(y) \rightarrow P(x) \\
\text{F}_c \exists y. \forall x. P(y) \rightarrow P(x) \\
\text{F}_c \forall x. P(\alpha) \rightarrow P(x) \\
\text{FP}(\alpha) \rightarrow P(t) \\
\text{TP}(\alpha), \text{FP}(t)
\end{align*}
\]
\[
\text{closed with } \alpha \equiv t.
\]

It is immediate to check that this tableau could not be generated in the original calculus $\text{IL-T}$.

When we apply our translation algorithm to this tableau, we obtain an Isabelle tactic $\text{Tac}$, that means a sequence of inference steps. Applying $\text{Tac}$ to the original goal produces a failure. In order to show why Isabelle rejects this tactic, we can
write down the natural-like proof which would be the result of applying \( Tac \) to the goal:

\[
\frac{\frac{[P(t)]}{\neg \forall x. P(t) \to P(x)} \quad \frac{P(t) \to P(t)}{\bot} \quad F \rightarrow}{\bot} \quad Fc \exists
\]

\[
\frac{\neg \exists y. \forall x. P(y) \to P(x)}{F \rightarrow}
\]

This proof violates the conditions on eigenvariables in the application of the \( Fc \forall \) rule: in fact the second occurrence of \( t \) in \( P(t) \to P(t) \) should be an eigenvariable, so ISABELLE rejects this step, and the (wrong) tableau could not be certified.

For the sake of completeness, we present also the right tableau, which is generated by the version of our prover, \( Tab \), which does not make use of dummy variables; as usual, we omit duplicated formulas when not needed:

\[
F \rightarrow \neg \exists y. \forall x. P(y) \to P(x)
\]

\[
T \rightarrow \exists y. \forall x. P(y) \to P(x)
\]

\[
Fc \exists y. \forall x. P(y) \to P(x)
\]

\[
Fc \forall x. P(a) \to P(x), Fc \exists y. \forall x. P(y) \to P(x)
\]

\[
FP(a) \to P(b), Fc \exists y. \forall x. P(y) \to P(x)
\]

\[
TP(a), FP(b), Fc \exists y. \forall x. P(y) \to P(x)
\]

\[
Fc \forall x. P(b) \to P(x), TP(a), FP(b)
\]

\[
FP(b) \to P(c), TP(a), FP(b)
\]

\[
TP(b), FP(c), TP(a), FP(b)
\]

closed by \( P(b) \).

Translating this tableau into a natural-like proof is straightforward (we leave it to the reader), and ISABELLE has no problem to certify its correctness.

We want to conclude this section with another example, which shows that the prover based on dummy variables is not sound; in fact, in \( IT \), it is able to prove that \( \exists x. \forall y. x = y \) from the assumption that \( = \) is reflexive:

\[
F \exists x. \forall y. x = y, T \forall x. x = x
\]

\[
F \forall y. \alpha = y, T \forall x. x = x
\]

\[
F \alpha = z, T \forall x. x = x
\]

\[
F \alpha = z, Tz = z, T \forall x. x = x
\]

closed with \( \alpha \equiv z \)

Of course, the tableau generates a non acceptable proof in natural deduction.
3.5 Summary

Summarizing, in this chapter we have shown that

- it is possible to code a reasonably efficient theorem prover as a tactic into the ISABELLE logical framework;
- there is a correct algorithm which uniformly maps IL-tableaus into natural-like deductions; the same holds for E-T;
- such translation procedures can be used to make trustworthy non-verified provers;
- a non-sound variant of the employed tableau calculi can be used in practice to write a theorem prover, and the translation functions can be used to discriminate between good and bad answers it gives;
- the non-sound tableau prover, from empirical results, is able to tackle goals which are impossible for the sound variant, and for the standard tactics ISABELLE provides, due to complexity reasons.

The first consequence of this result is that, in order to trust the answers of an automatic theorem prover which uses the IL-T tableau calculus, we just need to prove the existence and correctness of the function Tr we developed.

Following this line, we can refine our result: in fact, there is no real need to prove that the implementation of Tr is correct; admitting the possibility that a properly closed tableau could be discarded by a wrongly constructed tactic (that amounts to potentially enlarge the incompleteness of the prover), we reduce our trust to the correctness of the ISABELLE core engine, which is responsible for the evaluation of the tactics Tr constructs. We remark that a correctness proof for the algorithm Tr is needed since we must ensure that, at least in principle, no failure is due just to the translation step.

In some sense, we cannot ask for more: in order to do some kind of theorem proving, we need an environment which provides the basic functionalities; our choice was ISABELLE, but any other logical framework would have been the same. Then we have to develop strategies and/or provers for constructing proofs in this framework, the maximum we can achieve is to limit our faith to the core of the framework, which, eventually, would be proven formally correct.

The second consequence is more subtle; what Proposition 3.3.1 says, is that the translation rules respect the tableau structure. If we change the starting tableau calculus, it is possible to generate a set of translation rules which permits to prove Proposition 3.3.1 in a similar way as we did. In order to prove Theorem 3.3.1, it is essential to establish Proposition 3.3.3, but this is possible only if the tableau calculus is sound. So, even if the tableau calculus is not sound, we have a translation function which may leave some gaps into the natural-like deduction it generates.

It may appear odd to use a non-sound calculus, but it is not. In fact, if we let the \( \alpha \) parameter in the rules for IL-T TV, F\( \exists \) and F\( \exists 3 \) to be a dummy variable which
may get instantiated by unification during the process of constructing a tableau, we
get a calculus which is not sound, since it does not handle in the proper way the
binding of variables. This variation of our tableau calculus is much more efficient,
since it does not require so many duplications as the sound calculus, but, of course,
it may produce closed tableaus which are not \textbf{IL}-T proof tables. Our approach is
to validate a closed tableau by translating it into an \textbf{IL} proof. If this is possible, we
have been able to prove in an efficient way a potentially difficult theorem; if this is
not possible, it means that we applied in a non-sound way an expansion rule, so we
are not able to judge whether the goal is a theorem.

The third consequence of our approach depends on the particular shape of the
translation rules: since we took care of avoiding unnecessary detours, our natural-
like proofs are inspectable. This is important when we want to analyze the strategies
the prover adopts.

Concluding, we would like to remark that our technique, validation by trans-
lation, can be extended to many other calculi, not limited by a tableau structure.
As we will see in Chapter 4, it can be applied to other reasoning systems and to
formalized theories so to ensure trustworthiness for a prover without performing a
formal verification of the prover itself.

Moreover, we believe that validating by translation could be a winning approach
in many situations, where it is important to have only correct answer, but complete-
ness, i.e., the ability to generate every possible answer, is not a concern.
Chapter 4

Computer Arithmetic

Computer arithmetic is the mathematical theory which underlies the way calculational machines operate on integer numbers. When dealing with object code, as we remarked in Chapter 2, computer arithmetic is the primary way a processor has to calculate.

Computers manipulate integer numbers of a finite, fixed precision, internally represented as strings of bits of fixed length. A processor’s hardware (Braun, 1963) is built to perform additions, multiplications, and other standard arithmetical operations along with logical operations like “or”, “and”, “not”, “exclusive or”, and so on.

The distinguishing features of computer arithmetic are:

- logical operations, i.e., the ability to calculate bit per bit the conjunction, the disjunction and the negation of integers. For clarity, since we deal with a logical theory, we will refer to these operations with the adjective bitwise;

- fixed precision, which means every representable number lies in a fixed, well-defined range of values and every operation must signal exceptions, when unable to provide a result which fits into that range, usually by means of carry and/or overflows bits.

The goal of this chapter is to describe a logical formulation for computer arithmetic, along with a partial decision procedure for a significant fragment, and a general simplification procedure based on equational rewriting. As described in the architectural design in Chapter 2, the Computer Arithmetic Toolkit which is the ensemble containing the logical theory for computer arithmetic, the calculational engine which performs simplification of expressions, and the decision procedure, will be presented as a single package with two interfaces, one for higher order logic, and another one for the constructive logic E.

Dropping the constraint of finite precision, computer arithmetic is actually an extension of the standard theory of integer numbers (Smoryński, 1991), i.e., it provides the usual operations on integers as well as bitwise operations. We gain finite preci-
sion by axiomatizing congruences over the modulus operator, and we gain ranges in
values by choosing particular representations for the generated quotient structure.

From another point of view, the one of formal verification (Gordon, 1986), computer
arithmetic is essential, since it is the theory we are really dealing with when we
execute a program, or when we design a digital circuit.

The outline of this chapter is as follows: in the next section, we will give an
overview of Computer Arithmetic as a mathematical theory; in Section 4.2, we
will illustrate the three components of the Computer Arithmetic Toolkit, namely
the logical theories, as represented in ISABELLE, the calculational engine and the
decision procedure; in Section 4.3 we discuss the combination of these pieces so to
describe how their interaction permits to solve complex problems.

4.1 Computer Arithmetic

Every computer works by using bits, bytes, words and so on. These are the elements,
interpreted as numbers, the Arithmetical Logical Unit, ALU, the part of a processor
which is devoted to perform calculations (Braun, 1963), is able to deal with. There
are significant differences between processors in the size of numerical types, i.e., the
precision, and in their bounds, i.e., the range of values an element can represent.
Nowadays most processors are able to deal with operations on bytes, double bytes
and quadruple bytes; the usual convention they use for the sign of a number is the
two's complement representation, and they are able to perform additions, subtrac-
tions, multiplications, divisions, bitwise conjunction, bitwise disjunction and bitwise
negation on the numerical types they provide.

We want to have a general theory, which closely resembles the way ALUs perform
calculations, but we do not want to choose a particular set of numerical types for
a specific processor. To meet this goal we have to introduce a series of numerical
types. In particular, we need a theory for natural numbers, a theory for integer
numbers, and a theory for modular arithmetic over naturals and integers. As we
have seen in Chapter 2, we have to design two versions of CAT, one for the HOL
object logic, and one for the E object logic. In the purely constructive design, both
versions are present, since we have two instances of CAT, one at the syntactical
level, the E version, and another at the semantical level, the HOL version.

To minimize our efforts and to use the already developed provers, we inherit
the Nat and Int theories of the HOL object logic; in the E version of CAT, we
build these theories from scratch. We provide an alternative formalization of both
theories, since we want a binary representation for numbers, and a formalization of
bitwise operations. The equivalence of our theories and the standard ones should
be clear, since we have just a different coding of terms. We provide a tactic which
automates the translation of results between different formalizations of the same
theory.

An important fact is that our formalization has the format of specification frame-
works, as we will detail in Chapter 5. The consequences of this fact will be discussed
in general in that chapter, although we have to remark here that

- every datatype in our verification system is treated in an uniform way, namely, via the Specification Framework approach.

- in some cases, like computer arithmetic, instead of instantiating the general schema, we prefer to develop an ad hoc theory which is more effective, especially because it allows to write more efficient decision procedures.

A practical consequence of the adoption of an alternative formalization for integers and naturals is that different tools, namely the calculational engine and the decision procedure, become available. This fact means that we can use these additional provers to better reason even in the standard ISABELLE system. In other words, CAT is conceived to be an independent package, not linked to the Constructive Verification System, but sharing the same design constraints and the same spirit, so to be smoothly integrable.

The decision procedure we provide is based on the SUP-INF method (Shostak, 1979) for proving formulas of an extension of quantifier-free Presburger arithmetic. It mainly operates on integer numbers, and an adaption to natural number reasoning is built in.

The calculational engine is used to simplify numerical expressions: it is quite more efficient than the standard ISABELLE Simplifier, since it is specialized. More, it works both on the syntactical level, reducing expressions according to the algebraic properties of integers and naturals, and on the semantical level, performing calculations whenever possible, using the numerical representation of the ML programming language.

Modular arithmetic is defined from the theories for integers and naturals, by means of a quotienting operation in HOL. It is directly formalized as a type constructor in the E logic. Since every property of modular arithmetic can be stated, modulo a uniform translation, in the usual arithmetic language, we can use the calculational engine and the decision procedure, just automating the previous cited translation into their interface. Of course, this way of computing is not efficient, but, at least for our design purposes, it suffices for a first release of the project. Since providing an efficient set of provers for modular arithmetic, specialized on the topic is not part of our investigation, we leave a patch solution in our design, which could be replaced when there is a strong need to do so. Essentially, in this stage, we do not want to design a system which is tuned for performances, but rather to have a system which permits to perform the verification task with no limitations, and we want to use it to study the applicability of constructive techniques to formal verification.

Before going into technical details, as we will do in Section 4.2.1, we will provide a general idea of the mathematical formulation of the theories for numbers, we introduced in the above paragraphs.

The theory of natural numbers in ISABELLE/HOL is, essentially, the classical Peano arithmetic, as reported in (Andrews, 1986). Considered as a data structure
declaration, it is the free term algebra generated by the constructors 0 and s, the successor function.

The theory for natural numbers we are proposing is based on a binary representation; more specifically, a bit is 0 or 1, a natural number is a string of bits; formally this structure is coded as

\[
\text{datatype Bit = false | true} \\
\text{datatype Nat = 0 | 1 | Bin(Nat, Bit)}
\]

This coding is not satisfactory because the same number can be represented in more than one way, e.g., the number 2 can be represented both as Bin(1, false) and as Bin(Bin(0, true), false); in fact, writing the two representations as strings of bits, we get “10” and “010”.

The solution to this problem is to extend equality so that the two representations become identical. The following axioms, added to the standard theory of equality, have the purpose to strip down the trailing zeroes from a string of bits representation:

\[
\begin{align*}
\text{Bin}(x, \text{false}) &= 0 \leftrightarrow x = 0 \\
\text{Bin}(x, \text{true}) &= 1 \leftrightarrow x = 0 \\
\neg \text{Bin}(x, \text{true}) &= 0 \\
\neg \text{Bin}(x, \text{false}) &= 1 \\
\text{Bin}(x, a) &= \text{Bin}(y, b) \leftrightarrow x = y \land (a \leftrightarrow b)
\end{align*}
\]

The induction schema takes the form:

\[
\begin{array}{c}
[P(x)] \\
\vdots \\
P(0) \quad P(1) \quad P(\text{Bin}(x, \text{false})) \land P(\text{Bin}(x, \text{true})) \\
\hline
P(t)
\end{array}
\]

where \( x \) is an eigenvariable.

The standard operations on natural numbers are defined in the obvious way, as we will see later, by noticing that, apart from the syntax, our representation for numbers is the same as the usual notation in base 2.

The theory of integer numbers, in a binary representation is coded by two functions, Pos and Neg, from naturals to integers, and the following axioms:

\[
\begin{align*}
\text{Pos}(x) &= \text{Pos}(y) \leftrightarrow x = y \\
\text{Neg}(x) &= \text{Neg}(y) \leftrightarrow x = y \\
\text{Pos}(x) &= \text{Neg}(y) \leftrightarrow x = y \land x = 0
\end{align*}
\]

The induction schema for integers is as follows

\[
\begin{array}{c}
P(\text{Pos}(x)) \quad P(\text{Neg}(y)) \\
\hline
P(t)
\end{array}
\]

where \( x \) and \( y \) are eigenvariables.

Since Pos represents the set of positive numbers, and Neg, the set of negative numbers, the usual arithmetical operations are coded in the obvious way.
4.1. COMPUTER ARITHMETIC

We have not yet analyzed how to code bitwise operations. In a sense, these operations are critical because they strongly depend on the particular representation we choose. This dependence is less evident in the case of natural numbers, but for integers and for modular arithmetic, it becomes important.

To make clear the problem, let us suppose to define the bitwise and operation. Intuitively, it works by calculating the conjunction of bits in the same position in the binary representation of two numbers.

Let us take 5 and −7 and let us calculate the bitwise and of them. In our representation, 5 becomes \text{Pos}(5), and −7 is \text{Neg}(7), that is, using strings of bits, \text{Pos}("101") and \text{Neg}("111").

So the result of a bitwise and should be "101", that is, 5; but what about the sign? Is it \text{Pos}(5) or \text{Neg}(5)?

Since our goal is to model computer arithmetic, we adopt the convention that integer numbers are in the two complement format, so \text{Pos}(5) is really "...0101" (an infinite sequence of trailing zeroes), and \text{Neg}(7) is really "...1001" (an infinite sequence of trailing ones).

In this way bitwise operations are defined in the standard, bit per bit fashion, which, in the previous example, lead us to \text{Pos}(5) as a result.

A type for modular numbers, i.e., the elements on which modular arithmetic operates on, is defined by a quotient operation

\[ \text{ModInt}_n = \text{Int}/(\mod n). \]

The elements of \text{ModInt}_n are equivalence classes, but it is customary to use an integer number to identify a class. Normally two such representations are used: one for signed modular numbers, and another one for unsigned modular numbers. We will refer to these representations as concrete modular numbers. The diagram in Figure 4.1 shows the idea for \( n = 5 \).

\[
\text{UModInt}_5 = \{ 0, 1, 2, 3, 4 \} \\
\text{Int} \longrightarrow \text{ModInt}_5 = \{ [0], [1], [2], [3], [4] \} \\
\text{SModInt}_5 = \{ 0, 1, 2, -2, -1 \}
\]

Figure 4.1: Modular Numbers, an Example of Concrete Representations.
The function which gives a concrete representation for unsigned modular integers is \( f: \text{Int} \rightarrow \text{UModInt}_n \), where \( f(x) = x \mod n \); the function \( g: \text{Int} \rightarrow \text{SModInt}_n \), providing a concrete representation for signed modular numbers is \( g(x) = f(x) - (\text{if } 2f(x) < n \text{ then } 0 \text{ else } n) \).

Of course, \( \text{UModInt}_n \), and \( \text{SModInt}_n \) can be defined as the images of \( f \) and \( g \), respectively. Since \( f \), \( g \) are injections, we have a natural way to define arithmetical operations on modular numbers; in fact, standard results in algebra (Birkhoff and MacLane, 1967), show that, with respect to addition and multiplication, the canonical injection of integers into \( n \)-modular integers, is an homomorphism.

It is immediate to prove similar results for bitwise operations. Details will be described in the next section.

### 4.2 The Components

The concept of computer arithmetic is implemented by three major parts. A *logical theory* for arithmetic is developed in Section 4.2.1. It comprises a definition of numbers and operations on them using *Isabelle/HOL* (Paulson, 1997d) and the E theory as environments. Section 4.2.2 presents the *calculation engine* used to do calculations and simplifications on arithmetical expressions, efficiently. Last but not least, a *decision procedure* is introduced in Section 4.2.3. We have chosen Shostak's SUP-INF method (Shostak, 1979) for proving formulas of an extension of quantifier-free Presburger arithmetic.

#### 4.2.1 The Logical Development

Our goal is to code in a suitable way the numbers as previously described; we want a coordinated set of *Isabelle* theories which permit to reason on Computer Arithmetic inside HOL and inside the E logic.

Natural numbers in a binary representation can be coded by quotienting the datatype

\[
\text{datatype preNat}_2 = \text{preZero} \\
| \text{preOne} \\
| \text{preBin preNat}_2 \text{ bool}
\]

with the equivalence relation \( \sim \) defined by the following axioms:

\[
x = y \rightarrow x \sim y \tag{4.1}
\]

\[
\text{preBin } x \text{ false } \sim \text{preZero} \rightarrow x \sim \text{preZero} \tag{4.2}
\]

\[
\text{preBin } x \text{ true } \sim \text{preOne} \rightarrow x \sim \text{preZero} \tag{4.3}
\]

Although this solution is feasible in HOL, we think it is awkward to manage. We prefer a less elegant, but more effective, direct formalization by means of explicit axioms (the following is a slightly edited excerpt of the actual theory file):
4.2. THE COMPONENTS

\[
\text{types Nat2, BaseNat2;}
\]
\[
\text{consts}
\]
\[
bZero, bOne :: BaseNat2;
\]
\[
"0","1" :: Nat2;
\]
\[
\text{Bin} :: [\text{Nat2, BaseNat2 }] \to \text{Nat2}
\]
\[
\text{rules}
\]
\[
injBaseNat2 \quad \text{"bZero = bOne } \Rightarrow \text{ P"}
\]
\[
cloBaseNat2 \quad \text{"x = bZero | x = bOne"}
\]
\[
injNat2_1 \quad \text{"0 = 1 } \Rightarrow \text{ P"}
\]
\[
eq Nat2_1 \quad \text{"Bin(x,y) = 0 } \Rightarrow \text{ x = 0 & y = bZero"}
\]
\[
eq Nat2_2 \quad \text{"Bin(x,y) = 1 } \Rightarrow \text{ x = 0 & y = bOne"}
\]
\[
cloNat2 \quad \text{"EX x,y. n = Bin(x,y)"}
\]
\[
\text{indNat2} \quad \text{"[ | P(0); P(1);}
\]
\[
\text{ ALL x,y. (P(x) -- P(Bin(x,y))) | ] } \Rightarrow \text{ P(t)"
}\]

Importing the usual first order theory for equality, one can check that the preceding axioms define in an unique way an isoinitinal model for natural numbers.

We can add addition and multiplication, following the inductive structure of the binary representation; the axiomatization we are proposing is shown in Figure 4.2.

Since division and remainder are partial operations, not being defined for 0 as the dividend, we axiomatize them as follows, where \( y \neq 0 \):

\[
z = x/y \land w = x \mod y \Leftrightarrow x = w \cdot y \land w < y .
\]

In the previous formula, one should keep in mind that division and modulus are not operations, since, in a first order theory every function is total, but rather predicates. In a more explicit way, which we masquerade using the ISABELLE parser, one has, when \( y \neq 0 \),

\[
\text{Div}(x,y,z) \land \text{Mod}(x,y,w) \leftrightarrow x = w + z \cdot y \land w < y ,
\]
and \( \text{Div}(x,0,z) \leftrightarrow \bot, \text{Mod}(x,0,z) \leftrightarrow \bot \), for every \( x \) and \( z \).

In this way, we have an isoinitional theory, since there is a unique pair of partial operations satisfying the preceding equation, hence it acts like a definitional extension of the theory of natural numbers.

The definition of integers is a direct coding in ISABELLE of the representation we introduced in the previous section, where we define operations as extensions of the ones on natural numbers.

The theory file which encodes the Int2 theory declares:

\[
\text{types Int2;}
\]
\[
\text{consts}
\]
\[
\text{Pos, Neg :: Nat2 } \Rightarrow \text{ Int2;}
\]
Addition
\[ x + 0 = x \]
\[ 0 + x = x \]
\[ 1 + 1 = \text{Bin}(1, \text{bZero}) \]
\[ 1 + x = x + 1 \]
\[ \text{Bin}(x, \text{bOne}) + 1 = \text{Bin}(x + 1, \text{bZero}) \]
\[ \text{Bin}(x, \text{bZero}) + 1 = \text{Bin}(x, \text{bOne}) \]
\[ \text{Bin}(x, \text{bZero}) + \text{Bin}(z, y) = \text{Bin}(x + z, y) \]
\[ \text{Bin}(x, \text{bOne}) + \text{Bin}(z, \text{bZero}) = \text{Bin}(x + z, \text{bOne}) \]
\[ \text{Bin}(x, \text{bOne}) + \text{Bin}(z, \text{bOne}) = \text{Bin}(x + z + 1, \text{bZero}) \]

Subtraction
\[ x < y \rightarrow x - y = 0 \]
\[ x \geq y \rightarrow (z = x - y \leftrightarrow y + z = x) \]

Multiplication
\[ x \cdot 0 = 0 \]
\[ x \cdot 1 = x \]
\[ x \cdot \text{Bin}(z, \text{bZero}) = \text{Bin}(x \cdot z, \text{bZero}) \]
\[ x \cdot \text{Bin}(z, \text{bOne}) = \text{Bin}(x \cdot z, \text{bZero}) + x \]

Ordering Relations
\[ x \leq y \leftrightarrow (\exists z. y = x + z) \]
\[ x > y \leftrightarrow y < x \]
\[ x < y \leftrightarrow x < y \land \neg x = y \]
\[ x \geq y \leftrightarrow y \leq x \]

Figure 4.2: Operations and Ordering Relations on Natural Numbers.
4.2. THE COMPONENTS

rules

\begin{itemize}
\item $\text{injPos} \quad \text{"Pos}(x) = \text{Pos}(y) \Rightarrow x = y$
\item $\text{injNeg} \quad \text{"Neg}(x) = \text{Neg}(y) \Rightarrow x = y$
\item $\text{zeroInt2} \quad \text{"Pos}(x) = \text{Neg}(y) \Rightarrow x = y \land x = 0 \land y = 0$
\item $\text{indInt2} \quad \text{"}[ \quad ! ! \ x. \ P(\text{Pos}(x)) ; ! ! y. \ P(\text{Neg}(y)) \mid \quad \Rightarrow P(t)"
\end{itemize}

Operations and ordering relations are defined as in Figure 4.3.

Addition

\begin{align*}
\text{Pos}(x) + \text{Pos}(y) &= \text{Pos}(x + y) \\
x \geq y &\rightarrow \text{Pos}(x) + \text{Neg}(y) = \text{Pos}(x - y) \\
x < y &\rightarrow \text{Pos}(x) + \text{Neg}(y) = \text{Neg}(y - x)
\end{align*}

\begin{align*}
\text{Neg}(x) + \text{Pos}(y) &= \text{Pos}(y) + \text{Neg}(x) \\
\text{Neg}(x) + \text{Neg}(y) &= \text{Neg}(x + y)
\end{align*}

Multiplication

\begin{align*}
\text{Pos}(x) \cdot \text{Pos}(y) &= \text{Pos}(x \cdot y) \\
\text{Pos}(x) \cdot \text{Neg}(y) &= \text{Neg}(x \cdot y) \\
\text{Neg}(x) \cdot \text{Pos}(y) &= \text{Neg}(x \cdot y) \\
\text{Neg}(x) \cdot \text{Neg}(y) &= \text{Pos}(x \cdot y)
\end{align*}

Complement

\begin{align*}
\neg \text{Pos}(x) &= \text{Neg}(y) \\
\neg \text{Neg}(x) &= \text{Pos}(y)
\end{align*}

Subtraction

\begin{align*}
x - y &= x + (-y)
\end{align*}

Division and Remainder

\begin{align*}
z &= x / y \land w = x \mod y \quad \leftrightarrow y \neq 0 \land x = w + z \cdot y \land 0 \leq w \land w < y
\end{align*}

Ordering Relations

\begin{align*}
x \leq y &\leftrightarrow (\exists z. y = x + \text{Pos}(z)) \\
x \geq y &\leftrightarrow y \leq x \\
x < y &\leftrightarrow x \leq y \land \neg x = y \\
x > y &\leftrightarrow y < x
\end{align*}

Figure 4.3: Operations and Ordering Relations on Integer Numbers.

It is easy to verify that $\text{Int}_2$ is an isoinitial theory, and the corresponding intended model, see Chapter 5, is the usual ring of integers.
We should remark that both theories, \texttt{Nat} and \texttt{Int}, are formalized in such a way to form \textit{specification frameworks}, see Chapter 5, again, so to make uniform the approach to formalization in the whole Constructive Verification System, as required in the design issues discussed in Chapter 2.

Bitwise operations are easily defined by induction on the structure of integer two complement representation: the precise statements are given in Figure 4.4, where we adopted the standard notation, dropping the Pos and Neg; we have to remark that it is possible to adapt the \texttt{ISABELLE} external syntax so to masquerade the internal representation, and to adhere to the traditional way of writing mathematics.

It is possible to give a direct definition of them in term of Pos, Neg and the inductive structure of naturals, but it is unnecessarily complex. Bitwise conjunction and bitwise disjunction on natural numbers are defined as follows:

\[
\begin{align*}
    n \land m = p & \iff \text{Pos}(n) \land \text{Pos}(m) = \text{Pos}(p) \\
    n \lor m = p & \iff \text{Pos}(n) \lor \text{Pos}(m) = \text{Pos}(p)
\end{align*}
\]

Bitwise complement on natural numbers is undefined, since the bitwise complement of 0 would be an infinite sequence of ones, which is not a natural number in our definition.

\textbf{Conjunction}

\[
\begin{align*}
    0 \land x &= 0 \\
    -1 \land x &= x \\
    x \land 0 &= 0 \\
    x \land -1 &= x \\
    (2x + 1) \land (2y + 1) &= 2(x \land y) + 1 \\
    (2x + 1) \land 2y &= 2(x \land y) \\
    2x \land (2y + 1) &= 2(x \land y) \\
    2x \land 2y &= 2(x \land y)
\end{align*}
\]

\textbf{Negation}

\[
\begin{align*}
    \neg 0 &= -1 \\
    \neg -1 &= 0 \\
    \neg(2x + 1) &= 2(\neg x) \\
    \neg(2x) &= 2(\neg x) + 1
\end{align*}
\]

\textbf{Disjunction}

\[
\begin{align*}
    0 \lor x &= x \\
    -1 \lor x &= -1 \\
    x \lor 0 &= x \\
    x \lor -1 &= -1 \\
    (2x + 1) \lor (2y + 1) &= 2(x \lor y) + 1 \\
    (2x + 1) \lor 2y &= 2(x \lor y) + 1 \\
    2x \lor (2y + 1) &= 2(x \lor y) + 1 \\
    2x \lor 2y &= 2(x \lor y)
\end{align*}
\]

Figure 4.4: Definitions for Bitwise Operations on Integers

Modular numbers, \texttt{ModInt}, are defined in \texttt{ISABELLE} by the following axioms,
4.2. THE COMPONENTS

where Mod: Int\textsubscript{2} × Nat\textsubscript{2} → ModInt is the type constructor, and \( n \geq 2 \):

\[
\begin{align*}
\text{Mod}_n(x) &= \text{Mod}_n(y) \iff x \equiv y \mod \text{Pos}(n) \\
\text{Mod}_n(x) + \text{Mod}_n(y) &= \text{Mod}_n(x + y) \\
\text{Mod}_n(x) - \text{Mod}_n(y) &= \text{Mod}_n(x - y) \\
\text{Mod}_n(x) \cdot \text{Mod}_n(y) &= \text{Mod}_n(x \cdot y) \\
\text{Mod}_n(x)/\text{Mod}_n(y) &= \text{Mod}_n((x \mod \text{Pos}(n))/(y \mod \text{Pos}(n))) \\
\text{Mod}_n(x) \mod \text{Mod}_n(y) &= \text{Mod}_n((x \mod \text{Pos}(n)) \mod (y \mod \text{Pos}(n)))
\end{align*}
\]

In the previous definition, we should note that the last two statements, involving modulus and division have to be read as definitions of atomic formula, in the same way as for natural numbers, thus excluding from the domain all the tuples where the divisor is \( \text{Mod}_n(0) \).

The usual device to introduce a new type in ISABELLE, is to define one or more constructors, like \text{Mod}, or like \text{Pos} and \text{Neg}, and to limit the elements of that type to the ones which can be denoted from terms generated by the constructors. An easy, but important consequence of this fact, is that our theories are reachable, that is, they possess a model where every element of the universe can be denoted by a term.

The ModInt theory does not provide a definition for bitwise operations, since it depends on the concrete representation we adopt.

Concrete modular numbers are easily defined from ModInt, by using the functions introduced in the previous section; they provide definitions for bitwise operations, in the most obvious way, e.g., bitwise conjunction is defined in the theory UModInt, for unsigned modular integers, as

\[
\text{UMod}_n(x) \Delta \text{UMod}_n(y) = \text{UMod}_n((x \mod \text{Pos}(n)) \Delta (y \mod \text{Pos}(n)))
\]

In practice, the definition of instances of unsigned modular numbers and signed modular numbers reduces to provide constants for every equivalence class, that is, to introduce a name for every \( \text{Mod}_n(x) \), with \( 0 \leq x < n \), and \( n \geq 2 \). Every such a specialized theory can be generated automatically by inheriting via the Signature Morphism package the theorems from the corresponding general ISABELLE theories ModInt, UModInt and SModInt.

We should remark that, due to the way we constructed modular numbers, the computation of a modular arithmetic expressions can be reduced to a computation on integer arithmetic.

These translations between different arithmetics is automated in CAT, giving a variety of tactics which encapsulate reasoners on integer arithmetic, so that it appears that CAT provides provers and simplifiers for every numerical type it treats, although, internally this is false, since the same reasoners, the ones on binary represented integers, are always employed. The translation functions hide the way proving functionalities are implemented.

In the case of the \textbf{E} logic, our development is, essentially, the same, as we did for the traditional approach, except that we add an axiom for every relational symbol
saying that the classical and the constructive interpretation of that symbol coincide. For example, in the case of equality, we add to the theory the following axiom

$$\forall x, y. x = y \iff \forall x, y. \Box(x = y).$$

We will return on the reasons behind this addition in Chapter 5, where the Specification Framework approach is presented and discussed.

### 4.2.2 The Calculational Engine

The purpose of the Calculational Engine is to do reductions on integer expressions outside the logic.

A calculational engine works in a syntactical way by term rewriting and in a semantical way by doing actual calculations on integer numbers using the implementation programming language, i.e., SML (Paulson, 1996). This engine is defined by a function that takes the representation of a numerical expression and gives back a term that represents an equivalent, but reduced expression. We clarify next what we mean by **representation** of an expression, by **equivalent**, and by **reduced**.

The syntax of an internal language, in which the calculations and rewritings are done, is fixed by a data type. The corresponding construct `IntTerm` in ML is shown in Figure 4.5. It defines inductively a set of terms, called Int for the rest of this section.

```plaintext
datatype IntTerm = IntConstant of term * IntTerm list
  | IntValue of int
  | IntSucc of IntTerm
  | IntPred of IntTerm
  | IntComp of IntTerm
  | IntPlus of IntTerm * IntTerm
  | IntMinus of IntTerm * IntTerm
  | IntTimes of IntTerm * IntTerm
  | IntDivide of IntTerm * IntTerm
  | IntModulus of IntTerm * IntTerm;
```

Figure 4.5: The Syntax of the Calculational Engine Language.

Since the calculational engine is used to reason about integer expressions, that data type closely corresponds to the syntax for integer expressions used in Section 4.2.1. In fact, to allow a smooth combination of deduction system and calculational engine, there is an obvious mapping from integer expressions to their representations in the internal language. An advantage of this choice is that the same engine can be used in different theories, just by replacing the interface which translates an ISABELLE term into the internal language, and vice versa. See Section 4.3 for the combination of reasoners.
4.2. THE COMPONENTS

We call expressions, formed by IntTerm, integer expressions for the rest of this section, bearing in mind that those are just a representation of the expressions developed in Section 4.2.1.

The manipulation of integer expressions is done by a set of ML functions, called rewrite functions. Figure 4.6 shows such a function. The pattern matching feature of ML is used here to rewrite terms that have a certain structure and leave others unchanged.

\[
\text{fun Axiom1 (IntPlus (IntValue 0, x): IntTerm) = x} \\
| \text{Axiom1 default = default;}
\]

Figure 4.6: An Example of Rewrite Function.

A rewrite rule is gained by taking a lemma about the equivalence of integer expressions deduced from the definition of integers on the logical level and directing the equation, say from left to right, then we also translate the left- and the right-hand side to Int, using the obvious mapping where integer variables are taken to ML variables to allow pattern matching. A rewrite function is made from such rules in the obvious way. The following shows the preparing steps to implement the rewrite function of Figure 4.6. We take a lemma given by the definition of integers in Section 4.2.1.

\[
x + 0 = x
\]

That translates to the rewrite rule

\[
\text{IntPlus } (x, \text{IntValue 0}) \rightarrow x
\]

where \( x \) is a ML variable over IntTerm.

This tight relation between rewrite functions in ML and lemmas derived on the logical level gives a good justification for the claim that an integer expression is rewritten into an expression which is equivalent by means of the logical definition of integers. A formal treatment of rewriting and termination arguments is beyond the scope of this work, textbooks on term rewriting or overview articles such as (Dershowitz and Jouannaud, 1990) may be referred to for the standard proofs necessary.

An important part of our rewriting process, which, for efficiency reasons, is performed on demand, is its certification by translation. We will not give every detail here, but just the idea. Let us suppose that we are rewriting the expression \( \alpha \) into \( \beta \) by means of the rewrite function corresponding the lemma \( A = B \), where \( \alpha \) is an instance of \( A \), and \( \beta \) is an instance of \( B \). The ISABELLE system knows that \( A = B \) since this a lemma in the logical theory; in order to replicate the rewriting step on the logical level, we need to apply the substitution rule to the goal, and then to solve its first antecedent. Let the goal be \( P(\alpha) \), the instance of the substitution
rule we will apply is

\[
\frac{\alpha = \beta \quad P(\alpha)}{P(\beta)}
\]

So, after resolving the goal with this rule, we have two subgoals; the former, \( \alpha = \beta \)
is an instance of the lemma \( A = B \), so we can resolve it immediately; the latter is exactly the rewritten goal, so we have replicated on the logical level the internal rewriting step. As for the Constructive Reasoner, the Calculational Engine can generate a tactic, composed by the sequence of steps which simulate rewriting rules, and then it may use this tactic to certify its own behavior.

A crucial matter for the practical use of the calculational engine is the guarantee of termination. Since the rewrite procedure stops only if no more rules can be applied to any subterm of a given expression, it is therefore obvious that one cannot choose arbitrary lemmas to make them to rewrite rules. For instance, the unconditional application of

\[
\text{IntPlus} \ (x, y) \rightarrow \text{IntPlus} \ (y, x)
\]
is always possible if the argument expression contains addition.

The rewriting stops if every rewrite rule produces a reduced version of its argument, if applied. To express this reduction a well-founded ordering over \textbf{Int} has to be defined, and then one has to show that every rewrite rule in the system, if applied, gives a result that is strictly smaller than the argument. Provided that holds, the rewrite process must terminate, because the reduction cannot go on forever.

An ordering over terms can be conveniently defined by a homomorphism from the ground terms of \textbf{Int} to an algebra \( A \) (of the same signature) which already has a well founded ordering. Let \( \triangleright \) be such a well founded ordering on \( A \), then the monotonicity condition:

\[
f_A(\ldots x \ldots) \triangleright f_A(\ldots y \ldots) \quad \text{if} \quad x \triangleright y
\]

for all operations \( f_A \) and all \( x, y \) in \( A \) has to hold, as well.

The mapping \( \tau_1 : \textbf{Int} \rightarrow \{2, 3, \ldots \} \) explained in Figure 4.7 is a good starting point for an overall termination argument, where \( \triangleright \) is the usual well-ordering on natural numbers.

A reduction ordering \( \triangleright_{\tau_1} \) over \textbf{Int} is now given by:

\[
s \triangleright_{\tau_1} t \iff \tau_1(s) \triangleright \tau_1(t)
\]

with \( s, t \in \textbf{Int} \).

However, we need further reduction relations for coping with associative and commutative rewrite rules. Such measures are given by the mappings \( \tau_2 \) and \( \tau_3 \) sketched in Figure 4.8 and in Figure 4.9, respectively, where a suitable ordering relation \(<\) has to be defined.
4.2. THE COMPONENTS

\[
\begin{align*}
\text{IntConstant}_A(a, b) &= 2 \\
\text{IntPlus}_A(a, b) &= a + b + 1 \\
\text{IntValue}_A(a) &= 2 \\
\text{IntMinus}_A(a, b) &= a + 2^b + 1 \\
\text{IntSucC}_A(a) &= a + 4 \\
\text{IntTimes}_A(a, b) &= a \cdot b \\
\text{IntPred}_A(a) &= a + 4 \\
\text{IntDivide}_A(a, b) &= a \cdot b \\
\text{IntComp}_A(a) &= 2^a \\
\text{IntModulus}_A(a, b) &= a \cdot b \\
\end{align*}
\]

Figure 4.7: A Reduction Mapping.

\[
\begin{align*}
\text{IntPlus}_A(a, \text{IntPlus}_A(b, c)) &= 2 \\
\text{IntPlus}_A(\text{IntPlus}_A(a, b), c) &= 3 \\
&\vdots \\
\end{align*}
\]

Figure 4.8: An Associative Reduction Mapping.

\[
\begin{align*}
\text{IntPlus}_A(a, b) &= 2 \text{ if } a < b \\
\text{IntPlus}_A(a, b) &= 3 \text{ if } b < a \\
&\vdots \\
\end{align*}
\]

Figure 4.9: A Commutative Reduction Mapping.
The reduction ordering $\succ_\tau$, which we use, is built from $\tau_1$, $\tau_2$, and $\tau_3$ in the standard way of composing orderings by the lexicographical ordering of the cross-product. That is, given $s$ and $t \in \text{textbf{Int}}$, then $s \succ_\tau t$ iff $s \succ_{\tau_1} t$, or in case $s = t$, then $s \succ_{\tau_2} t$, or in case $s = t$, then $s \succ_{\tau_3} t$. The termination of the calculational engine can be easily shown with $\succ_\tau$.

The following example appeared when an actual correctness proof of a program was performed by the author in (Benini, 1998b; Benini et al., 1998b; Benini and Nowotka, 1998). It shows the effectiveness of reasoning with this rewrite engine in practice. We do not use the syntax of $\text{IntTerm}$ here to increase readability.

The expression:

$$c + ((3 + (f(x) + (-4))) + (-c))$$

is subsequently reduced to

$$c + ((f(x) + (3 + (-4))) + (-c)),$$
$$c + ((f(x) + (-1)) + (-c)),$$
$$c + (f(x) + ((-1) + (-c))),$$
$$c + (f(x) + ((-c) + (-1))),$$
$$c + ((-c) + (f(x) + (-1))),$$
$$0 + (f(x) + (-1)),$$

and finally

$$f(x) + (-1).$$

### 4.2.3 The Decision Procedure

The third part of our representation of Computer Arithmetic consists of a decision procedure for integer arithmetic with function symbols. We use Shostak's SUP-INF algorithm (Shostak, 1977; Shostak, 1979) for solving unquantified Presburger formulas which might also contain an unlimited number of function and predicate symbols.

Intuitively, Presburger formulas are those that can be built up from integers and variables over integers, addition, equality and inequality relations, and first-order logical connectives. The decision procedure we use here, see (Shostak, 1979), operates on an extension of the quantifier-free version of Presburger arithmetic. This extension allows an unlimited number of $n$-ary function symbols $f^{(n)}: \mathbb{Z}^n \rightarrow \mathbb{Z}$, and $n$-ary predicate symbols $P^{(n)}: \mathbb{Z}^n$, with $0 \leq n$, in each formula. These symbols are treated as undefined, i.e., they are not interpreted. A small example of this kind of formula is the following:

$$x < f(y) \land f(y) \leq (x + 1) \rightarrow (P(x) \leftrightarrow P(f(y) + (-1))).$$
4.3. **THE COMBINED SYSTEM**

More generally, Shostak's algorithm works on unquantified formulas of first-order logic which contain any function and predicate symbols over the set of integers. Those symbols are not interpreted except the function + and the predicates $<, \leq, \geq, >,$ and $\equiv.$ A more general treatment of decision procedures is given in (Shostak, 1984) and (Cyrluk et al., 1996). Eventually we plan to include these enhancements in the Constructive Verification System, but for the moment being, we limited ourselves to the plain version of the algorithm, which seems to work fine in the context of interactive theorem proving.

This decision procedure is linked to the theory of integers developed in Section 4.2.1 by a straightforward translation from logical formulas to an internal language similarly to the way described in Section 4.2.2. In addition to that, we allow multiplication with constants, for they can be rewritten as a finite sum, and also subtraction, since that can be rewritten as addition by complementing the second addend.

The decision procedure rejects terms which cannot be translated into the internal language, i.e., which are not recognized as formulas of the described extension of unquantified Presburger arithmetic. An exception is signaled in that case and the calling procedure, e.g., a proof tactic, has to cope with it.

If a formula was read in, the procedure gives a *true* or *false* as answer, depending on which conclusion it has reached.

In the case of Shostak's algorithm, we believe that the approach to validate results we used so far, cannot be applied; it appears that there is not a direct link between the logical level and the problem model which constitutes the basis of the decision procedure. In a sense, it seems that Shostak's algorithm is very *semantic*, while the validation by translation technique is very well suited for *syntactical* decision procedures.

### 4.3 The Combined System

The strength of our approach to computer arithmetic is the combination of systems that have different fields of application but which are chosen and implemented to work together in order to solve a demanding problem. Efficient interaction and a reasonable "division of labor" was a major design goal.

Stemming from the demand to reason about computer arithmetic, a logical theory was developed which fits these needs and overcomes limitations of existing solutions by extending them.

In order to make use of this theory we need a way to reason about it. The most basic way to do that, is to use basic rewriting of goals by rules gained from the equational theory induced by the definition of integers. ISABELLE provides a powerful rewrite engine, called *simplifier*, that supports the proving procedure in a sophisticated way. Nevertheless, its generality prevents the simplifier to be as efficient and useful as a specialized rewrite engine could be. Many arithmetical goals could only awkwardly or not at all be proved with that tool alone. Apart from
rewriting, actual calculations like division cannot be done in a reasonable way in Isabelle. The calculational engine is used to remedy such problems.

Simplifying arithmetical expressions by calculations and simple rewriting is a basic way of reasoning but is too "primitive" to provide efficient tactics. Shostak's decision procedure is therefore used to solve more sophisticated goals when reasoning about computer arithmetic. It should be remarked that the Computational Engine is used to aid the decision procedure directly by rewriting an integer expression into a proper format such that the decision procedure can reason about it.

Technically, the integration of the calculational engine into Isabelle is done as a simple tactic that calls an oracle which maps an Isabelle term to the internal language of the calculational engine. Given the conclusion of the current subgoal, the oracle then provides the result of the calculations as a theorem which states that the argument and the reduced expression are equivalent. The subgoal and that theorem are then resolved.

Alternatively, the calculational engine is linked to the simplifier in Isabelle. By setting up a simplification procedure the external reasoner can aid the simplifier when rewriting complex subgoals which involve arithmetical expressions.

The decision procedure is linked to the theorem prover in a similar way as the calculational engine.

Finally, we can say that a logical theory of numbers and bitwise operations provides a suitable basis for modeling Computer Arithmetic. The reasoning in that theory is made feasible by the tight coupling of a specialized rewrite engine, combined with calculation capabilities, and a powerful decision procedure.

4.4 Summary

A combined system for reasoning in computer arithmetic has been shown in this chapter. An integrated approach of logical, calculational and rewriting techniques makes it feasible to deduce complex theorems in computer arithmetic.

The content of this chapter is mainly derived from (Benini and Nowotka, 1998) and from (Benini et al., 1998b). We have to thank D. Nowotka, C. Pulley, S. Kalvala and M. Ornaghi for their suggestions, which made possible the prototyping of most of the CAT package. A version of CAT, which does not include support for modular arithmetic has been developed in the Department of Computer Science, University of Warwick by the author, D. Nowotka and C. Pulley, under the supervision of S. Kalvala, and it is available on the Web.

With respect to that work, in this chapter we axiomatized the logical side according to the requirements for specification frameworks, we considered how to render the package for the E logical system, and, last, but not the least, we designed how to cope with modular arithmetic. Actually a prototypic version of CAT, which includes everything as designed here, is developed; still, it is not available, since, at the moment of writing, there are few more technical problems to iron out before releasing it.
Future development of CAT will include enhancements to the decision procedure, as well as a specialized version of it for natural arithmetic, and for modular arithmetic. Also, a better engineering of the package, that makes CAT easier to integrate with other logics, is planned.
Chapter 5

Specification Frameworks

In this chapter we will speak about specification frameworks as a way to model the world where the verification/analysis/synthesis task takes place.

The notion of specification framework has been introduced in the context of Logic Programming and, now, it has a consolidated tradition (Flener et al., 1997a; Flener et al., 1997b; Kreitz et al., 1996).

In the design of the Constructive Verification Environment, the notion of specification framework is used to formalize the theories we use to reason about programs; this approach gives raise to a highly uniform system that permits to verify programs, to analyze their correctness proofs and to synthesize programs from specification using a very homogeneous set of common concepts. In fact, as we will show in Chapter 6, the analysis of correctness proof in our paradigm is the constructive basis which makes appealing the specification frameworks as a modeling technique.

From the point of view of the implementation, we have to warn the reader: this part of the thesis has been analyzed from a theoretical point of view, but, for the moment being, no real implementation has been developed.

Thus, our remarks about how to implement in Isabelle the tools we will introduce, are not supported by prototypical versions, but only by the experience of the author on that theorem proving system.

Finding a solution to a problem implies an analysis phase where one must build up a language used to describe the world the problem is posed in. Then one must use this language to state the properties which are supposed to be relevant to solve the problem itself. Finally, one writes down specifications; in our approach every phase of this modeling process is formalized using the formal apparatus we are going to introduce.

More specifically, the first two steps correspond to the definition of a specification framework, while the last step is realized by writing specifications in the language of the framework developed in the first two steps.

Then, one writes the program which implements the specification, and, eventually, proves it correctness. Or, alternatively, one may synthesize a formally correct
program directly from the specification. The great uniformity of our design appears here: from the representation of programs, described in Chapter 7, to the analysis of their correctness proofs, developed in Chapter 6, we work using always the same set of instruments and concepts: in a verification task, we start from a program, a specification, and a context where the program exploits its action; we model these parts in a constructive logical system, constituted by a specification framework in the E logic, we perform the verification, and, at the end, we are able to analyze the resulting proof in a formal, automatic way; in a synthesis task, we start from a specification and a context, we model these parts exactly in the same way as for the verification task, we proceed by developing a proof for the specification, and, then, we obtain a program which is guaranteed to be formally correct.

Our presentation will follow this development pattern: in the first section, we will define the notion of specification framework and we will describe its main properties; in the second section, we will define what we intend for specification of a program, by giving a computational reading to formulas in specification frameworks; in the last section, we will introduce the notion of program schema, that, in conjunction with the notion of proof schema, permits the synthesis of programs.

5.1 Specification Frameworks

The logical language we will adopt is the one of multi-sorted first-order logic (Barwise, 1977). In the following we assume the standard terminology and notations.

Before starting with the definitions, we want to remark that both the IL and the E logics are multi-sorted first-order logics, thus we will treat them together.

A Σ-structure $\mathcal{G}$, that is model on the signature $\Sigma$, is $\Sigma$-reachable if every element in the domains can be denoted by a closed term (in datatype theory this property is also referred to as no junk property (Goguen and Meseguer, 1987)).

In this context, a specification framework $\mathcal{F} = (\Sigma, \text{Ax})$ is composed of a signature $\Sigma$, and a finite or recursive set $\text{Ax}$ of $\Sigma$-axioms (Kreitz et al., 1996). We distinguish between closed and open (specification) frameworks. This distinction is formalized using isoinitial models: this concept permits to single out the intended models a framework is supposed to speak of. A formal treatment of isoinitial models is given in (Miglioli et al., 1994a; Miglioli et al., 1982a).

**Definition 5.1.1** Let $T$ be a theory and let $\mathcal{M}$ be a classical model for $T$. We say that $\mathcal{M}$ is an isoinitial model for $T$ iff, for every model $\mathcal{N}$ of $T$, there is a unique isomorphic embedding from $\mathcal{M}$ into $\mathcal{N}$.

It is immediate to show that the isoinitial model of a theory, if it exists, is unique up to isomorphisms.
5.1. SPECIFICATION FRAMEWORKS

5.1.1 Closed Specification Frameworks

Definition 5.1.2 (Closed Framework) A framework $\mathcal{F} = \langle \Sigma, Ax \rangle$ is closed if and only if there is a $\Sigma$-reachable isoinitial model $M$ for $Ax$. We call $M$ the intended model of $Ax$.

Thus, our closed frameworks are isoinitial theories, namely theories with a reachable isoinitial model. They are similar to initial theories, which axiomatize reachable initial models. The latter are quite popular in algebraic abstract data types and specifications (Goguen and Meseguer, 1985; Goguen et al., 1976). The difference is that initial models use homomorphisms, instead of isomorphic embeddings.

In Constructive Verification Environment, we want to make the user able to declare that an ISABELLE theory is a closed specification framework. To achieve this goal, we have to provide a proper extension to the syntax of ISABELLE theories, which informs the system that the theory has to be considered as a closed framework, and we need to check that the theory is, indeed, a closed specification framework.

The first part is quite easy: extending the syntax of ISABELLE theories is possible, and it has been done for other packages (Paulson, 1993; Paulson, 1997b).

The second part requires to characterize the notion of isoinitial theory in a way which permits a check. In fact, we cannot directly express the constraints of the definition in ISABELLE: we have no way to define what is a model of a theory, nor we can state what is an isomorphic embedding, and even the condition that the model is reachable cannot be expressed in ISABELLE. Thus, we need to explore other ways to characterize isoinitial theories, allowing us to check them with the formal instruments ISABELLE provides.

The starting point is to replace the condition on isomorphic embeddings with something we have some hopes to check in a theorem prover. In (Miglioli et al., 1989a), the following condition has been shown.

Definition 5.1.3 A theory $T$ is said to be atomically complete iff, for every closed atomic formula $\phi$, $T \vdash_{c.e.} \phi$ or $T \vdash_{c.e.} \neg \phi$.

Theorem 5.1.1 A theory $T$ has a reachable isoinitial model iff $T$ has a reachable model and it is atomically complete.

The preceding condition reduces the problem of checking if a theory is reachable and isoinital, to the problem of checking if it is reachable and atomically complete. Atomic completeness is a property of a syntactical nature, hence it is closer to what we want, but still it has a problem: it requires to perform a proof for every closed atomic formula.

An aspect we have not introduced till now is the link with constructive systems. In fact, to continue our analysis of specification frameworks, we need to work with constructive systems, as intuitively introduced in Chapter 1; later on, Chapter 6, we will see that the adoption of constructive formal systems as the working basis
for the verification process will open the door to the formal analysis of correctness proofs.

As we will prove in Chapter 8, \( T \) and \( E \) are constructive systems, as well as all the extensions we will show in the following.

It should be clear to the reader that a generic theory \( T \) in the \( E \) logic may be isoinitial, and, in particular, atomically complete, without implying for this fact the \( E + T \) is a constructive system, as defined in Chapter 1.

Let us suppose that \( T \) is \( E \)-constructive\(^1\), then the test for atomic completeness is reduced to prove that \( \forall x. r(x) \lor \neg r(x) \) for any relation symbol \( r \) of the signature. This formula is obviously true in \( \text{CL} \), classical logic, but not in a constructive logic; whenever we are able to prove it, we can immediately deduce that, for every term \( t \), either \( r(t) \) holds, or \( \neg r(t) \) holds.

The following fact, which can be easily proven from the results we will show in Chapter 8, permits us to design part of the solution:

**Proposition 5.1.1** If, for every relational symbol \( r \) of arity \( n \) in the signature of the \( E \)-constructive theory \( T \), one can prove in \( E \) that

\[
\forall x_1, \ldots, x_n. r(x_1, \ldots, x_n) \lor \neg r(x_1, \ldots, x_n),
\]

then \( T \) is atomically complete.

Thus, we can simply force the user to provide a tactic as part of the definition of the ISABELLE theory, which, for every defined relational symbol \( r \) with arity \( n \), solves the goal

\[
\forall x_1, \ldots, x_n. r(x_1, \ldots, x_n) \lor \neg r(x_1, \ldots, x_n).
\]

Now, we turn to the other problem, that is, to ensure that a theory has a reachable model.

The approach we adopt in this case is prescriptive: in the ISABELLE-HOL logic, we define a type \( \tau \) with a particular shape, then, via the Admissibility Checker package, see Chapter 2, we generate a set of postulates that, axiomatically, in the sense of ISABELLE, define a corresponding type \( \tau_E \) in the \( E \) logic. In the traditional Constructive Verification Framework, we proceed in the same way, deriving the axioms from the definition of \( \tau \).

The property we need to ensure that the domain represented by the type \( \tau \) is reachable is

**Proposition 5.1.2** For every finite set of constructors \( C \), i.e., constants and function symbols, and for every congruence \( \approx \), the quotient \( \approx \) is of the term algebra generated by \( C \) forms a reachable model over the signature \( C \cup \{\approx\} \).

\(^1\)In general, we will say that a theory \( T \) is \( L \)-constructive when \( T + L \) is constructive, as defined in Chapter 1.
5.1. SPECIFICATION FRAMEWORKS

Hence, when defining a new closed specification framework, we have to provide:

1. a signature \( \Sigma \), containing one or more types \( \tau_1, \ldots, \tau_n \), some constants and function symbols on these types, and some relation symbols on these types;

2. a special set of axioms, \( \text{Eq} \), on equality;

3. a set of axioms \( \mathcal{A} \) which forms the main body of the specification framework; they, together with \( \text{Eq} \) should form a constructive theory;

4. a tactic \( \text{tac} \) which proves the condition on atomic completeness for every relation symbol and for equality.

The Specification Framework package, see Chapter 2, extracts from the definition of a closed framework \( T \), the signature \( \Sigma \) and, via the Admissibility Checker, automatically constructs the sets corresponding to the types \( \tau_1, \ldots, \tau_n \) in HOL, by using the set \( \text{Eq} \) plus the theory of identity as an equivalence relation to quotient the term algebras generated by \( \Sigma \); the term algebra generation is performed by the Induction package. We call concrete representations the sets constructed in this way.

Then, the Specification Framework package analyzes the format of the \( \text{Eq} \) and \( \mathcal{A} \) axioms, checking if they give raise to a constructive theory. We will discuss later what kind of axioms are acceptable.

Finally, the Specification Framework package imports into the framework \( T \) the structural induction principle \( \text{Ind} \) as generated by the Induction package in HOL, and tries to prove, using the tactic \( \text{tac} \) the goals on atomic completeness; the tactic \( \text{tac} \) works in the logical system \( \mathbf{E} + \mathcal{A} + \text{Eq} + \text{Ind} \).

If there are no problems in this process, the framework \( T \) is accepted, and it becomes available to the user in the Constructive Verification Environment.

A small improvement we may use to easier the task of defining new closed specification frameworks, and related to the format of axioms, is contained in the following theorem from (Miglioli et al., 1989a):

**Theorem 5.1.2** For every set \( \mathcal{C} \) of constructors, i.e., constant and function symbols, the term algebra generated by \( \mathcal{C} \) is an isomorphism model for the theory

\[
T(\mathcal{C}) = \text{Identity theory} + \text{injectivity axioms} + \text{structural induction principle}.
\]

Moreover, \( T(\mathcal{C}) \) is \( \mathbf{E} \)-constructive.

Most of the times, the way of operating we have just described is not what we want: in fact, it requires that the framework to check is new; in many cases, we are interested in extending an existing framework, adding new axioms, or enlarging the signature.

In these cases, we want to have an extension mechanism which does not change the intended model except for the new symbols we may define, i.e., we want to make model-preserving extensions.
Definition 5.1.4 Let $S = \langle \Sigma, A \rangle$ be a closed specification framework, let $\Sigma'$ be a
signature containing $\Sigma$, and let $A'$ be a theory on the signature $\Sigma'$ such that $A \subseteq A'$.
Then $S' = \langle \Sigma', A' \rangle$ is an extension of $S$ iff $S'$ is a closed framework such that, being
$\mathcal{M}$ the intended model of $S$, the intended model $\mathcal{M}'$ of $S'$ is an expansion of $\mathcal{M}$, i.e.,
the interpretation of $\Sigma$ in $\mathcal{M}'$ coincides with the interpretation of $\Sigma$ in $\mathcal{M}$.

To achieve this result, which permits to reuse already defined frameworks, we
need some properties. We start our discussion on extensions by analyzing the cases
of adding new symbols and new types.

When we add a new type $\tau$, essentially, we proceed as for a new closed framework,
generating its concrete representation in HOL, and certifying the relation symbols
where $\tau$ occurs for isoinitiality, as before.

When we add a new constant or function symbol, the user may decide

1. to declare the new symbol as an extension

2. or, to declare it a a simple symbol.

In the latter case, we have to certify the whole framework, regarding it as new on
the types occurring in the definition of the new symbol, thus, the user has to provide
the necessary equality axioms, and the tactic for checking atomic completeness.

In the former case, the Specification Framework package requires a theorem
which forms the definition of the new symbol.

Lemma 5.1.1 Let $S = \langle \Sigma, A \rangle$ be a closed specification framework in the E logic, let $A$ be E-constructive, and let

$$A \vdash \forall x_1, \ldots, x_n. \exists! y. F(x_1, \ldots, x_n, y)$$

then, being $f$ a new function symbol,

$$\langle \Sigma \cup \{ f \}, A \cup \{ \forall x_1, \ldots, x_n. F(x_1, \ldots, x_n, f(x_1, \ldots, x_n)) \} \rangle$$

is an extension of $S$.

The task of the Specification Framework package is to take the theorem we use
to define the new symbol, and to generate a new axiom, as in the preceding lemma,
which codifies the definition of the new symbol.

In an analogous way we treat the addition of a relation symbol, where the property
which permits us to avoid a check for reachability and isoinitiality is:

Lemma 5.1.2 Let $S = \langle \Sigma, A \rangle$ be a closed specification framework in the E logic, let $A$ be E-constructive, and let

$$A \vdash \forall x_1, \ldots, x_n. H(x_1, \ldots, x_n) \lor \neg H(x_1, \ldots, x_n)$$

then, being $r$ a new relation symbol

$$\langle \Sigma \cup \{ r \}, A \cup \{ \forall x_1, \ldots, x_n. r(x_1, \ldots, x_n) \leftrightarrow H(x_1, \ldots, x_n) \} \rangle$$

is an extension of $S$. 
5.1. SPECIFICATION FRAMEWORKS

Similar results hold the the IL logic, thus covering extensions on the signatures of closed frameworks in both versions of the Constructive Verification Environment.

The other possibility when extending a closed specification framework, is to add new axioms. Let $\mathcal{S} = (\Sigma, A)$ be a closed specification framework and let $B$ be a set of axioms on the signature $\Sigma$, we want a mechanism which ensures that $(\Sigma, A \cup B)$ is a closed specification framework.

Our proposal is that, if $\mathcal{S}$ is an IL-framework, then $B$ has to be an Harrop theory, and, that, if $\mathcal{S}$ is an E-framework, then $B$ has to be a $\Box$-theory.

Formally, a $\Box$-theory is a set of E-Harrop formulas, and an E-Harrop formula is defined as follows

**Definition 5.1.5** A formula $\phi$ is said to be an E-Harrop iff

- $\phi \equiv \Box \psi$;
- $\phi \equiv \psi \land \theta$ and both $\psi$ and $\theta$ are E-Harrop formulas;
- $\phi \equiv \psi \rightarrow \theta$ and $\theta$ is an E-Harrop formula;
- $\phi \equiv \forall x. \psi$ and $\psi$ is an E-Harrop formula.

Our proposal is supported by the fact that $T + E$, where $T$ is a $\Box$-theory, is constructive, as proved in Chapter 8, and by the fact that the initial model is not changed, provided that $T$ is $E$-consistent. Analogous results hold for the IL logic.

There are other kind of axioms which are admissible, for example induction principles not generated via the type construction mechanism, like the Descending Chain Principle we will discuss later. For the moment being, we have decided not to support other extensions, because we think that what we provide suffices for the needs of the Constructive Verification Environment. As we said at the beginning of this chapter, this is the only part of the thesis which has not yet support from implementation, hence we have no empirical evidence of the need for other kind of axioms, since the only examples we tried fall into the categories we presented.

To conclude this subsection, we want to show that Peano Arithmetic is, indeed, a closed specification framework, as anticipated in Chapter 4; the other theories in the Computer Arithmetic Toolkit can be proven to be closed specification frameworks in a quite similar way, using the properties we have shown in this section.

The closed specification framework for Peano Arithmetic is shown in Figure 5.1. It is the result of a construction which goes as follows:

- First, we declare as a closed framework $\text{PA}_0$, including the theory of identity, as follows

  **Framework $\text{PA}_0$**
  
  **SORTS**: $\mathbb{N}$
  
  **FUNCTIONS**: $0 : [] \rightarrow \mathbb{N}$
  
  $s : [\mathbb{N}] \rightarrow \mathbb{N}$
  
  **RELATIONS**: $=: [\mathbb{N}, \mathbb{N}]$
Framework PA

Sorts : \[ \mathbb{N} \]
Functions : \[ 0 : [\mathbb{N}] \to \mathbb{N} ; \]
\[ \mathbf{s} : [\mathbb{N}] \to \mathbb{N} ; \]
\[ +, \cdot : [\mathbb{N}, \mathbb{N}] \to \mathbb{N} ; \]

Relations : \[ = : [\mathbb{N}, \mathbb{N}] \]

Axioms : \[
\forall x. -0 = \mathbf{s}(x) ; \quad \forall x, y. \mathbf{s}(x) = \mathbf{s}(y) \implies x = y ; \\
\forall x. x + 0 = x ; \quad \forall x, y. x + \mathbf{s}(y) = \mathbf{s}(x + y) ; \\
\forall x. x \cdot 0 = 0 ; \quad \forall x, y. x \cdot \mathbf{s}(y) = x + x \cdot y ; \\
H(0) \land \forall x. (H(x) \to H(\mathbf{s}(x))) \to \forall x. H(x) 
\]

Figure 5.1: The Closed Framework PA, for Peano Arithmetic.

since the intended model of \( \text{PA}_0 \) is the term algebra generated by the constructors, we are guaranteed that this is a closed specification framework. The Specification Manager provides us a type \( \mathbb{N} \) and some additional axioms characterizing it:

\[
\forall x. -0 = \mathbf{s}(x) \\
\forall x, y. \mathbf{s}(x) = \mathbf{s}(y) \implies x = y \\
H(0) \land \forall x. (H(x) \to H(\mathbf{s}(x))) \to \forall x. H(x) 
\]

• Then we can extend \( \text{PA}_0 \) to \( \text{PA} \), by enlarging the signature with two new function symbols, + and \( \cdot \), and adding a series of axioms describing their behavior

\[
\forall x. x + 0 = x \\
\forall x, y. x + \mathbf{s}(y) = \mathbf{s}(x + y) \\
\forall x. x \cdot 0 = 0 \\
\forall x, y. x \cdot \mathbf{s}(y) = x + x \cdot y 
\]

Being \( \text{E-Harrop} \) formulas these axioms are acceptable, if we are able to prove atomic completeness, that reduces to provide a tactic for solving the goal

\[
\forall x, y. x = y \lor \neg x = y . 
\]

Moreover, these axioms form a model-preserving extension of the isoinitial model for \( \text{PA}_0 \).

There is a small imprecision regarding \( \text{E-Harrop} \) formulas in the previous description: in the definition we do not mention atomic formulas, while we use them in the definition of \( \text{PA} \).

In Chapter 2 and in Chapter 4, we required that atomic formulas has to be interpreted in a classical way. The purpose of using an isoinitial semantics for our theories is exactly in this direction; but we need, in the case of the \( \text{E} \) logic, to force this fact, because we have an explicit constructor for classical truth, the \( \Box \) operator.
Hence, when generating the theory corresponding to a closed specification framework in the $\mathbf{E}$ logic, we automatically generate axioms of the form

$$\forall x_1, \ldots, x_n. \Box r(x_1, \ldots, x_n) \rightarrow r(x_1, \ldots, x_n)$$

for every relation symbol $r$ of arity $n$ in the signature.

We will prove in Chapter 8 that adding these axioms to an $\mathbf{E}$-theory does not change its constructive character, and it is immediate to check that it does not change its character with respect to isoinitiality. Thus, there is no need to distinguish between atomic formulas and boxed atomic formulas since they are equivalent ($A \rightarrow \Box A$ is a theorem in $\mathbf{E}$).

### 5.1.2 Open Specification Frameworks

Differently from a closed framework, an open framework depends on some parameters and characterizes a class of isoinitial theories. A parametric signature is a signature $\Sigma(P)$ where some symbols, the ones occurring in the list $P$, are put into evidence as parameters, see, e.g., (Miglioli et al., 1994a). A parametric theory $\text{Th}(P)$ over $\Sigma(P)$ is any $\Sigma(P)$-theory.

We can write $\text{Th}(P) = \mathcal{C}_P \cup \text{Ax}$, where $\mathcal{C}_P$ is the set of constraints, that is, axioms containing only parametric symbols and $\text{Ax}$ is the set of internal axioms, containing at least a non-parametric symbol. The internal axioms are intended to formalize the defined symbols of $\Sigma(P)$, while the constraints represent requirements to be satisfied by actual parameters.

**Definition 5.1.6 (Open Framework)** Let $P$ be a set of parameters, let $\Sigma(P)$ be a parametric signature and let $T(P) = \mathcal{C}_P \cup \text{Ax}$ be a parametric theory. The structure $\mathbb{F}(P) = \langle \Sigma(P), T(P) \rangle$ is an open specification framework iff, for every closed framework $\mathbb{C} = \langle \Sigma_C, A_C \rangle$ such that $P \subseteq \Sigma_C$ and $\mathbb{C} \models C_P$, $\mathbb{F}(\mathbb{C}) = \langle \Sigma_C \cup \Sigma(P), A_C \cup A \rangle$ is a closed specification framework. We call instance of $\mathbb{F}(P)$ with $\mathbb{C}$, the closed framework $\mathbb{F}(\mathbb{C})$. The intended models of $\mathbb{F}(P)$ are the intended models of all its instances.

One may change the definition by noticing that it is not necessary to require the $P \subseteq \Sigma_C$, but asking for something less, in particular, that there is a signature morphism preserving the parameters $P$, details can be found in (Miglioli et al., 1994a). In our design, we adopt the strict definition.

About the implementation of open specification frameworks, we have to distinguish two functions we want to provide: their creation and their instantiation.

Creating a new open specification framework is very similar to creating a new closed framework: we need to declare a special kind of ISABELLE theory where we define the signature, the parameters and the axioms, dividing them into constraints and internal axioms. Then the Specification Framework package has to check that the declared theory is, indeed, an open framework.
This checking requires two steps: first, we must check that the constraints contain only parametric symbols, thus ensuring that the syntax of the declared open framework is right; second, we must check that the instantiation of an open framework with a generic closed framework satisfying the constraints, is again a closed framework.

We are assured that an instance is a closed framework if the internal axioms form an Harrop theory in $\mathbf{IL}$, or a $\Box$-theory in $\mathbf{E}$, as immediately follows from the properties we introduced in the previous subsection.

When we want to use an open framework, we want to instantiate its parameters so to have a closed framework where we inherit the whole set of theorems proven in the open framework.

To model instantiation, we use the theory inheritance mechanism of $\textsc{Isabelle}$: when we declare a new closed framework that is generated by instantiating the open framework $\mathbb{F}(P)$ with the closed framework $\mathbb{C}$, we start by declaring a new $\textsc{Isabelle}$ theory inheriting from both $\mathbb{F}(P)$ and $\mathbb{C}$, and stating that it is an instance of $\mathbb{F}(P)$ via $\mathbb{C}$, providing the $\textsc{Isabelle}$ theorems, proven in $\mathbb{C}$, which satisfy the constraints of $\mathbb{F}(P)$.

The Specification Framework package, in this case, has to check that the theorems have the right format to cover the whole set of constraints of $\mathbb{F}(P)$, and it has to check that the signature of $\mathbb{C}$ contains $P$, the parameters of $\mathbb{F}(P)$; this step is performed using the Signature Morphism manager, which has the knowledge on how to cope with the $\textsc{Isabelle}$ representation of signatures. The role of the Signature Morphism manager, for the current stage of the design, is to treat the signatures of $\textsc{Isabelle}$ theories, in particular frameworks. In the future, we think to enlarge it so to manage also morphisms between signatures, thus allowing the \emph{large} definition for open specification frameworks.

As an example of open framework (Figure 5.2), we show a characterization for lists; this presentation differs from the standard algebraic description of lists, but it has the advantage to model direct access to elements.

The intended models of $\textsc{List}(\text{Elem}, \prec)$ are the usual list structures with a partial ordering $\prec$ on the (parametric) element type. Natural numbers, the function $\text{nocc}(x, L)$ (number of occurrences of $x$ in $L$) and $\text{nth}(L, i, a)$ ($a$ occurs in $L$ at position $i$) have been introduced in this framework to make possible to reason about lists as a structured aggregation of elements, and, having direct access to elements through the $\text{nth}$ function, to make easier to write down specifications.

We point the attention of the reader to the constraint $\forall x, y. x \prec y \lor \neg x \prec y$, which makes sense considering that the $\mathbf{E}$ system is constructive, see Chapter 8.

5.2 Specifications

In this section we want to introduce a computational reading for a subclass of formulas of the $\mathbf{E}$ language. The interpretation we want to show is based on the idea that a formula may be read as a specification, that is, as a declaration of a task to
perform, using a specification framework.

For instance, the specification $\exists z. z \cdot z = x \lor \square(\neg \exists z. z \cdot z = x)$ in the context of the framework PA expresses, according to our interpretation, the task of deciding whether $x$ is a perfect square or not, and in the former case it also express the task of computing the square root of $x$; while, in the same context, the specification $\square(\exists z. z \cdot z = x) \lor \square(\neg \exists z. z \cdot z = x)$ expresses the task of deciding whether $x$ is a perfect square or not, without requiring the computation of the square root.

The use of specifications of this kind in the Constructive Verification Environment is double: first, we adopt this intended reading of specifications when we verify programs, and, as we will discuss in Chapter 6, this leads us to formal analysis; second, we use these specifications for synthesizing programs, as we will show in the next section.

### 5.2.1 Specifications for Program Synthesis

We consider specifications of the form $\Gamma \Rightarrow \phi$, where $\Rightarrow$ is meta-implication, and $\Gamma$ and $\phi$ are formulas in the E language with no occurrences of $\forall$ and $\rightarrow$, except under the scope of a $\square$ operator, and with negations only on atomic formulas.\(^2\)

\(^2\)This last requirement is not restrictive, since, having De Morgan’s Laws, we can adopt the standard technique to move negations “inside”
order features. Moreover, this restriction permits to simplify the formal understanding of specifications; in any case, the semantics as used here is perfectly compatible with the general semantics of evaluation forms, as given in (Miglioli et al., 1989b), see also Chapters 3 and 8.

From the point of view of ISABELLE, a specification of this form, is just a goal to prove, presented in the standard form (Paulson, 1997e). We remark that this interpretation holds even for the most general form of specifications, with no restrictions on the occurrences of connectives.

Reading a specification as a computational requirement, it states a correctness requirement on a program. Its precise meaning can be stated using the semantics of evaluation forms, explained in (Miglioli et al., 1989b). Here we give this semantics in a simplified form, oriented to our computational interpretation of constructive proofs.

To this aim, we associate with every formula \( \theta \) the set of its free individual variables \( \text{Var}_\theta \), and a set \( V_\theta \) of evaluation variables. An assignment of \( V_\theta \) codifies an evaluation form of \( \theta \), i.e., a possible explanation (witness) of its truth (Miglioli et al., 1989b).

We inductively define \( V_\theta \), together with its (informal) meaning, as follows:

- if \( \theta \) is a literal or a \( \Box \)-formula, \( V_\theta \) is empty, since we do not require any explanation of its truth;

- if \( \theta \equiv \alpha \lor \beta \), then \( V_\theta = V_\alpha \cup V_\beta \cup \{tv_\theta\} \), where \( tv_\theta \) is a new boolean variable; in an isoinitial model \( \mathfrak{M} \) and assignment \( J \) of \( \text{Var}_\theta \), its meaning is "if \( tv_\theta \) is false then \( \alpha \) is true else \( \beta \) is true";

- if \( \theta \equiv \alpha \land \beta \), then \( V_\theta = V_\alpha \cup V_\beta \); its meaning is recursively explained by the meaning of \( \alpha \) and \( \beta \) through \( V_\alpha \) and \( V_\beta \);

- if \( \theta \equiv \exists x. \alpha \), then \( V_\theta = V_\alpha \cup \{x_\theta\} \), where \( x_\theta \) is a new variable with the sort of \( x \); in an isoinitial model \( \mathfrak{M} \) and an assignment \( J \) of \( \text{Var}_\theta \), the meaning of an assignment \( x_\theta = t \) is "\( \exists x. \alpha \) is true because \( \alpha \) is true by assigning \( x \) to \( t \)".

For a set \( \Gamma \) of formulas, \( \text{Var}_\Gamma \) and \( V_\Gamma \) are defined as the unions of \( \text{Var}_\alpha \) and \( V_\alpha \), for \( \alpha \in \Gamma \).

Now, in a closed framework \( \mathbb{F} \) with isoinitial model \( \mathfrak{M} \), a specification \( \Gamma \Rightarrow \phi \) is interpreted as the following requirement: for any assignment \( J \) of the individual sequent variables \( \text{Var}_\Gamma \cup \text{Var}_\phi \) and \( I \) of \( V_\Gamma \), if \( I \) and \( J \) make \( \Gamma \) true in \( \mathfrak{M} \), then we want to compute an assignment \( I' \) of \( V_\phi \), such that \( I' \) and \( J \) make \( \phi \) true\(^3\) in \( \mathfrak{M} \).

For example, in the \( \text{PA} \) framework, let us consider the sequent

\[
\exists z. z + z = x \lor z + sz = x \Rightarrow \exists z. z + z = sx \lor z + sz = sz = sx
\]

\(^3\)A formal definition of what we intend for "to make true" is omitted for conciseness. It follows in the obvious way from the definition of \( V_\phi \) and is informally explained through an example.
There is just one sequent variable, $x$; let
\[
V \exists x. x + z = x + x + x = x \implies \{z_1, tv_1\} \quad \text{and} \\
V \exists x. x + z = x + x + x = x \implies \{z_2, tv_2\}
\]

An assignment that makes the antecedent $\Gamma$ true is, e.g., $x = s^0$, $z_1 = s^0$, $tv_1 = \text{false}$; the correct output assignment is $z_2 = s^0$, $tv_2 = \text{true}$. So a correct procedure is the following:
\[
\text{if } tv_1 \text{ then } z_2 := z_1 \quad \text{else } z_2 := s^0 \quad tv_2 := \neg tv_1
\]

Correctness in an open framework requires correctness in all its closed instances; now, the program for computing the evaluation of the output formula is an open program, i.e., it may contain holes and uninterpreted functions.

The requirement is correct reusability: the open program is correct if, when it gets instantiated in a closed instance, it becomes a complete and correct program.

5.2.2 Schemas for Program Synthesis

In this section, we want to introduce the way we follow to synthesize programs from proofs. Our approach is based on proof schemas, a special kind of inference rules, which are derived in the $E$ system plus a specification framework satisfying some basic computational properties. In our view, we start from a specification formula, as described in the previous section, and we prove it in a framework using proof schemas; the result, as we will see, is that the proof can be compiled into a program, which correctly realizes the specification.

The intuitive idea behind a proof schema $S_{\text{proof}}$ is that it represents, at the same time, both a derived inference rule in a framework expressed in the $E$ logic, and a partially specified program, that is, a program schema. Using schemas when deducing a specification goal permits to extract from the resulting proof a program which correctly implements the starting specification. For a complete account, see (Avellone et al., 1998a); here we will just recall the fundamental inference rule, dischargeability, which assures computational completeness, i.e., that every program may eventually be derived.

Since it is quite common to consider specification of the form $\Gamma \Rightarrow \phi \lor \Box \neg \phi$, we find convenient to introduce a shortening notation: $\Gamma \Rightarrow [\phi]$.

Let us consider a closed framework $\mathbb{F} = \langle \Sigma, \text{Th} \rangle$ with isoinitial model $\mathcal{G}$, and a specification of the kind $\Delta(x) \Rightarrow [\exists z. \psi(x, z)]^4$.

The computability of the specification implies that the set of all possible elements $a$ satisfying $\Delta(x)$ in the isoinitial model $\mathcal{G}$ can be divided into two sets,
\[
D^+ = \{a \mid \mathcal{G} \models \Delta(x/a) \cup \exists z. \psi(x/a, z)\}
\]

An underlined variable like $\underline{x}$ stands for a tuple of variables $x_1, \ldots, x_k$. 

\[4\text{ An underlined variable like } \underline{x} \text{ stands for a tuple of variables } x_1, \ldots, x_k. \]
and

\[ D^- = \{ a \mid \mathcal{S} \models \Delta(x/a) \cup \{ \square \exists z. \psi(x/a, z) \} \} . \]

Now, let us suppose that there exist \( n + m \) sets of wffs \( \Gamma_1^+(x), \ldots, \Gamma_n^+(x) \) and \( \Gamma_1^-(x), \ldots, \Gamma_m^-(x) \) such that:

(F1) \( a \in D^+ \) iff there exists \( \Gamma_i^+(x) \) (\( 1 \leq i \leq n \)) such that \( \mathcal{S} \models \Gamma_i^+(x/a) \);

(F2) \( a \in D^- \) iff there exists \( \Gamma_j^-(x) \) (\( 1 \leq j \leq m \)) such that \( \mathcal{S} \models \Gamma_j^-(x/a) \).

(F3) For \( 1 \leq i \leq n \), \( \Delta(x), \Gamma_i^+(x) \models \psi(x, t(x)) \) for an appropriate term \( t(x) \);

(F4) For \( 1 \leq j \leq m \), \( \Delta(x), \Gamma_j^-(x) \models \square \exists z. \psi(x, z) \).

If we are able to satisfy conditions (F1)-(F4), then we have “reduced” the problem of solving the specification to the problem of deciding, given a possible input \( a \) satisfying the preconditions \( \Delta(x) \), which set \( \Gamma_i^+(x/a) \) gets satisfied in the isoinitial model. Obviously, the problem has been “reduced” if the wffs occurring in these sets are “simpler” than the wff representing the specification. If we can state that the set whose elements are \( \Gamma_i^+(x) \), is dischargeable then we are able to generate a proof for \( \phi \) in the \( \mathbf{E} \) system which does not depend on any \( \Gamma_i^+(x) \). Technical details can be found in (Avelone et al., 1998a). We remark that the computation which checks if a set is dischargeable can be used to construct that proof; this algorithm is the basis for implementing the Dischargeability Rule in \textsc{Isabelle} as a tactic.

The Dischargeability Rule can be interpreted as a program pattern; every element of the sets \( \Gamma_i^+(x) \) is a test or a function we should compute in a case analysis structure which follows the proof pattern. Again, details can be found in (Avelone et al., 1998a).

To illustrate how this rule works in practice, let us consider an example. Let us suppose to work in the framework of total orderings, and let us define

\[
\min(a, b, c, m) = m \leq a \land m \leq b \land m \leq c \land (m = a \lor m = b \lor m = c) ;
\]

we want to synthesize a program satisfying the specification \( \exists m. \min(x, y, z, m) \), that is, a program to compute the minimum element in a set of three elements.

Using the framework, it is not difficult to prove the following facts:

- \( \{ x \leq y, x \leq z \} \Rightarrow \min(x, y, z, x) \)
- \( \{ x \leq y, \neg x \leq z \} \Rightarrow \min(x, y, z, z) \)
- \( \{ \neg x \leq y, y \leq z \} \Rightarrow \min(x, y, z, y) \)
- \( \{ \neg x \leq y, \neg y \leq z \} \Rightarrow \min(x, y, z, z) \)
The family of sets whose members are the preconditions of these facts, constitutes a dischargeable set, so we can use it to construct the following proof, where $\theta \equiv \exists m, \min(x, y, z, m)$:

\[
\begin{array}{ccc}
[x \leq y] & [x \leq y] & [\neg x \leq y] & [\neg x \leq y] \\
[x \leq z] & [\neg x \leq z] & [y \leq z] & [\neg y \leq z] \\
\vdots & \vdots & \vdots & \vdots \\
[x \leq z] & \min(x, y, z, x) & \theta & \min(x, y, z, z) & \min(x, y, z, z)
\end{array}
\]

\[
\begin{array}{cc}
[y \leq z] & \theta & \theta \\
\end{array}
\]

The synthesized program schema is

\[
\text{if } x \leq y \text{ then } \\
\quad \text{if } x \leq z \text{ then } m := x \\
\quad \text{else } m := z \\
\text{else } \\
\quad \text{if } y \leq z \text{ then } m := y \\
\quad \text{else } m := z
\]

### 5.2.3 Induction Principles

In general, proving a specification in a framework involves some inductive reasoning. Here, we will study two induction principles, **DESCENDING CHAIN** and **DIVIDE ET IMPERA**, showing how they act as schemas, i.e., how they can be interpreted as proof patterns. While other principles are studied in (Avellone et al., 1998a), here our focus is on lifting computational structures, specifically loops, on the logical level.

#### Descending Chain Principle

The Descending Chain Principle has been introduced in the context of program synthesis in (Miglioli et al., 1994a; Miglioli and Ornaghi, 1981a; Miglioli and Ornaghi, 1981b) as the counterpart of while loops. Here we describe this principle applied to a specification of the kind $\Delta(x) \Rightarrow \exists z. \psi(x, z)$. The principle can be extended to other specification forms. The DCH rule is:

\[
\Delta(x) \quad \Delta(x), [A(x, y)] \\
\vdash_{\pi_1} \vdash_{\pi_2} \\
\exists z. A(x, z) \quad (\exists z. A(x, z) \land z < y) \lor \exists z. \psi(x, z) \\
\vdash_{\text{DCH}} \exists z. \psi(x, z)
\]

where $y$ (the parameter of the rule) does not occur free in $\psi$ and in the other undischarged assumptions of the proof $\pi_2$, so $y$ acts as an eigenvariable.

For the principle to be valid, the framework $\mathbb{F}(P)$ must satisfy the condition that the relation symbol $\prec$ must be interpreted in any intended model of $\mathbb{F}(P)$ as a well founded order relation.
$A(x, y)$ is called the invariant. This induction principle corresponds to a repeat-until loop which computes a decreasing sequence of values, with respect to $\prec$, approximating the solution; the solution is reached at the end of the cycle. Hence, the program schema corresponding to the DCH rule is:

$$
P_1; \quad \text{repeat} \quad P_2; \quad \text{until (not } t\theta);$$

where $P_1$ and $P_2$ are the sub-programs obtained by translating the proofs $\pi_1$ and $\pi_2$ respectively, and $t\theta$ is the variable storing the boolean value associated with the formula $\theta \equiv (\exists z. A(\overline{x}, z) \land z \prec y) \lor \exists z. \psi(\overline{x}, z)$, as defined in the interpretation of specifications. For a more detailed discussion, see (Avellone et al., 1998a).

**Divide et Impera**

A frequently used solving method consists of partitioning the inputs into smaller instances and then to solve the original problem combining the result obtained from the solution of the problem applied to the small instances. This solution strategy is called *Divide et Impera*, and, in many cases, it permits to obtain efficient algorithms.

Considering a specification $\Delta(x) \Rightarrow \exists z. \phi(x, z)$, the *Divide et Impera* method is described as

1. If $dg(x) \leq C$ for a fixed value $C$, the solution can be directly computed;
2. If $dg(x) > C$, the generated algorithm has to do the following steps
   
   (a) $x$ is partitioned in $y_1, \ldots, y_n$ such that, for all $j$, $dg(y_j) < dg(x)$;
   (b) The specifications $\Delta(y_j) \Rightarrow \exists z_j. \phi(y_j, z_j)$ are recursively solved;
   (c) The witnesses $t_j$, for the specifications solved in the previous step, are combined to obtain the solution for $x$.

where $dg$ is a measure function on the input space.

The proof schema which encodes the previous computation pattern is

<table>
<thead>
<tr>
<th>$\Delta(x)$, $[\text{dg}(y) \leq C]$</th>
<th>$\Delta(x)$, $[\text{PART}(y, y_1, \ldots, y_n)]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists z. \phi(y, z)$</td>
<td>$\exists z_1. \phi(y_1, z_1)$, \ldots, $\exists z_n. \phi(y_n, z_n)$</td>
</tr>
</tbody>
</table>

The program corresponding to the schema can be summarized as follows:

**Procedure** $F(x)$

begin

if $dg(x) \leq C$ then return $P_1(x)$
else
begin
\[ P_2(x, y_1, \ldots, y_n); \]
for \( j = 1 \) to \( n \) do \( z_j := F(y_j); \)
\[ P_3(x, z_1, \ldots, z_n); \]
return \( z; \)
end
end

where \( P_1, P_2 \) and \( P_3 \) are the programs synthesized from \( \pi_1, \pi_2 \) and \( \pi_3 \), respectively, and \( \text{Part} \) is the predicate which represents the input partition procedure.

This proof schema is valid only in a specification framework \( F = (\Sigma, \text{Th}) \) where it is possible to prove that

\[
\text{Th} \vdash \forall x, y_1, \ldots, y_n. \text{Part}(x, y_1, \ldots, y_n) \land \text{dg}(x) > C \rightarrow \bigwedge_{1 \leq j \leq n} \text{dg}(y_j) < \text{dg}(x)
\]

As an example, let us consider the specification \( \exists x. \minArray(A, x) \) in the open framework which axiomatizes arrays; it can be described as a direct generalization of \( \text{List}(\text{Elem}, <) \). The \( \minArray \) predicate is defined as

\[
\minArray(A, x) \equiv (\exists n. \text{nth}(A, n, x)) \land \Box(\forall i. \exists y. \text{nth}(A, i, y) \rightarrow x \leq y)
\]

To compute this specification in a correct way, we want to take advantage of the \( \min \) predicate synthesized using the Dischargeability Rule. A way to encode this idea, is the following: let us fix \( C = 3 \) and \( \text{dg}(A) = \text{size}(A) \); our partition strategy is to divide the array into three pieces of the same size, and it is immediate to write a \( \text{Part} \) predicate which encodes this requirement.

Applying the \textit{Divide et Impera} proof schema to these definitions, the proof \( \pi_3 \) is essentially the same as the proof we gave for \( \min \) in Section 5.2.2; the proof \( \pi_2 \) has to be derived from the framework; proof \( \pi_1 \) is by cases on \( \text{dg}(A) \). The resulting program schema is

**Procedure** \( \minArray(A) \)
begin
if \( \text{size}(A) \leq 3 \) then return \( P_1(A) \)
else
begin
\[ P_3(A, A_1, A_2, A_3); \]
for \( j = 1 \) to \( 3 \) do \( z_j := \minArray(A_j); \)
min \( (z_1, z_2, z_3, z) \);
return \( z; \)
end
end

We observe that the preceding program uses the already synthesized \( \min \) procedure, that the program is parametric in \( P_1 \) and \( P_2 \), and it is correct if \( P_1 \) (\( P_2 \)) has a correctness proof which matches \( \pi_1 \) (\( \pi_2 \)) in the previous schema.
5.3 Summary

We have presented the basics of the theory of specification frameworks, developed using the notion of iso-initial models. The result of this presentation is a design for the Specification Framework package, the Signature Morphism package and the E Logic Admissibility Checker, the modules of the Constructive Verification Environment which are responsible of modeling theories.

We need this kind of modeling because we want to interpret formulas as specifications, that is, we want to assign them a computational interpretation, as we discussed in this chapter. The reason for this need lies in the way we plan to synthesize programs, but also, as it will appear clear in Chapters 6 and 8, in the way we use to extract information from correctness proofs.

Finally, we have shown how to synthesize programs, using the notion of proof schema as the natural counterpart to program schema. Our study permits to develop proofs in the specification framework approach exploiting the computational content of specifications, and then, interpreting the proofs as programs in a generic programming language: of course, being the Constructive Verification Environment oriented to object code verification, our target language will be an assembly language, and specifically, we plan to use the MC68000 assembly.
Chapter 6

Analysis of Correctness Proofs

It is common experience that a proof contains many true facts which, together, concur to establish the truth of the conclusion. In a correctness proof, many of these facts are strictly related to the verified program.

Our conception of analysis is to extract the truth content from a proof. A general way to think a proof is to imagine it as a sequence of steps linking the hypothesis to the conclusion. During this process, a series of true facts must be established, and a set of implicit, but obvious, consequences are derived.

Informally, the whole set of true facts, either explicit or implicit, which are established in the proof development forms its truth content. Of course, this is not a definition, since we have not described what we mean as implicit information, and we have not fixed any system which specifies what are the proof steps we accept.

The outline of this chapter is as follows: in the next section we will introduce the Collection Method, that is our main information extraction tool; then we will show how the Collection Method can be used to distinguish relevant information in a correctness proof. In the last section, we will discuss a non obvious consequence of information extraction applied to the formal analysis of correctness proofs, namely the analysis of informal specifications.

The goal of this introduction is to provide the intuition behind the formal instruments we will develop. For this reason we are not very precise now, but we prefer to give the flavor of our analysis methodology. In this respect, we feel free not to choose any formal logic in this moment, but to discuss what aspects of this choice could be relevant for our purposes and why.

Our analysis starts from a formal proof $\Pi$: every fact which is true because we can exhibit a subproof of $\Pi$ for it, is part of the information content of $\Pi$. We can derive more facts by combining subproofs of $\Pi$; again, their conclusions are part of the information content of $\Pi$. Every instance of eigenvariables of a subproof of $\Pi$ is again something which, intuitively we have proven, hence it should be part of the information content of $\Pi$, as well.

The core of the Collection Method is to provide a family of algorithms, all based on the same structure, which construct a set $I$ of formulas which is a subset of the
information content of a proof, and, at the same time, it contains enough formulas to give a complete account of the reasons why the proof is true, in a constructive perspective.

The last sentence needs an explanation: as we will show precisely in Chapter 8, a set of formulas can be evaluated with respect to itself; if it contains a conjunction then it must contain both conjuncts, if it contains a disjunction it must contain at least one of the disjuncts, and so on.

The Collection Method constructs the minimal set, closed under evaluation, containing the whole set of facts which can be directly extracted from the initial proof, i.e., the facts which can be derived from a proof by looking at its subproofs.

Since the goal of this thesis is to analyze to what extent constructive methods can be employed in formal verification, the Collection Method is a perfect candidate to analyze. In our practical experience, it performs quite well, since it is structured in a way which permits to easily tune its computation following the functional paradigm of lazy evaluation.

As we have seen in Chapter 5, it is important, in the context of analysis of correctness proofs, to exploit the constructive character of the logical system, since deciding disjunctions and existentials gives a way to use proofs as programs. The Collection Method can be thought as a logical machine which computes proofs (Miglioli et al., 1988; Miglioli et al., 1991; Miglioli et al., 1982a; Miglioli et al., 1982b; Miglioli and Ornaghi, 1978; Miglioli and Ornaghi, 1981a; Miglioli and Ornaghi, 1981b; Miglioli and Ornaghi, 1982).

Once we know what is the information content of a correctness proof, our goal is to filter this enormous amount of information to retain just the relevant parts; if our notion of relevant is formalized as the statements which directly say something about the program, it is possible to perform this filtering operation on the fly, that is, during the lazy construction of the set containing the information content of the correctness proof. We will show this application in Section 6.2.

An interesting consequence of the simple application we will describe, is that it permits to an expert on the program to check if the formalized specifications make sense, that is, if they adhere to the informal ones, at least in the view of the developers.

This consequence, which is, of course, not formal, has a great importance in enhancing the confidence in formal verification in real applications.

As remarked in Chapter 1, a weak point in a traditional approach to formal verification of software programs is the high expertise required to read the result, and the non clear link between specifications as perceived by software developers and formal verifiers; with our technique, the gap between these two worlds is no more insurmountable, as we will discuss in Section 6.3.
6.1 The Collection Method

The Collection Method is a way to extract information from a proof in a given formal system. Fixed the formal system, it permits to construct an algorithm which extracts the information content of a finite set of proofs.

In fact, it provides a family of algorithms based on the same principle, but quite different in their implementations and performances.

In this section we will show two instances of the Collection Method, and some formal constructions of the algorithms which embody the method itself. In particular, we will show how the Collection Method can be applied to first-order intuitionistic logic and to first-order intuitionistic Peano arithmetic.

6.1.1 Intuitionistic Logic

In this part, we will show the Collection Method applied to a logic, namely first-order intuitionistic logic. The same schema we are going to investigate can be applied to most logics.

The formal system, i.e., the logical calculus we adopt for intuitionistic logic is based on the language whose connectives are $\neg$, $\land$, $\lor$, $\rightarrow$, $\forall$, $\exists$, a set of variables denoted by lowercase roman letters, and a set of uninterpreted function and predicate symbols.

Intuitionistic logic is based on minimal logic where no notion of falsity or negation is given; the calculus $\text{IL}$ for intuitionistic first order logic has been presented in Chapter 3, in Table 3.1.

The calculus $\text{IL}$ can be given in an equivalent way by deleting the $\bot E$ rule and adding explicit rules for negation:

\[
\begin{array}{c}
[A] \\
\vdots \\
C \\
\vdots \\
\neg C \\
\hline
\neg A
\end{array}
\]

and

\[
\begin{array}{c}
\neg A \\
A \\
\hline
B
\end{array}
\]

where $B$ is atomic; the general rule, where $B$ is not atomic, can be derived in the system.

We recall that the classical logic $\text{CL}$ is just $\text{IL}$ plus the classical rule

\[
\begin{array}{c}
[\neg B] \\
\vdots \\
B \\
\vdots \\
A \\
\hline
A
\end{array}
\]
Subproof Generation

The first notion we are interested in, is the subproof relation.

**Definition 6.1.1** We say that \( \Gamma \vdash A \) is a subproof of \( \Delta \), notation \( \vdash A \prec \Delta \), where \( \Gamma \vdash \Delta \) and \( \vdash A \) are proofs in the IL calculus, iff one of the following cases applies.

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);

- \( \Delta \vdash A \) and \( \vdash A \) are identical, modulo \( \alpha \)-conversion (Barendregt, 1984);
6.1. THE COLLECTION METHOD

\[ \Delta \vdash x. B_1(x) \quad \text{and} \quad \Gamma \not\vdash \Delta \]
\[ B \quad \Gamma \vdash x. B_1(x) \]

\[ \Delta \vdash B_1(t) \quad \text{and} \quad \Gamma \not\vdash \Delta \]
\[ B \quad \exists x. B_1(x) \]

\[ \Delta \vdash \Delta_1 \quad \Delta_2[B_1(p)] \quad \text{and} \quad \Gamma \not\vdash \Delta \]
\[ B \quad \exists x. B_1(x) \]

\[ \Delta \vdash \Delta_1 \quad \Delta_2 \quad \text{and} \quad \Gamma \not\vdash \Delta \]
\[ B \quad \not\vdash B_1 \quad B \quad A \]

**Definition 6.1.2** Let \( \Gamma \vdash A \) be a proof, we define Subproof\( \Gamma \vdash A \) = \( \{ \Delta \mid \Delta \vdash \Gamma \not\vdash \Delta \} \).

We should remark that, for any \( \Gamma \vdash A \), Subproof\( \Gamma \vdash A \) can be generated lazily; in fact, constructing the set \( S = \text{Subproof} \Gamma \vdash A \) following the inductive definition, we generate a sequence of subproofs from the bottom of the proof tree \( \Gamma \vdash A \); it is obvious that such generation can be performed in stages, that means, one step per application of an inference rule. The advantage of this method to generate the set lies in space we need to represent Subproof\( \Gamma \vdash A \); in fact, calculating this set explicitly occupies an exponential space with respect to the size of \( \Gamma \vdash A \), while calculating it in a lazy way, occupies a size which is linear with respect to the size of \( \Gamma \vdash A \).

**Collecting Information**

The Collection Method is based on two operations:

- composing subproofs of a given set of proofs;
- instantiating eigenvariables of parametric proofs to generate new proofs.
The collection operator performs the first function. The idea is that, if

\[
\begin{align*}
\Gamma & \vdash_1 \Delta, A \\
A & \vdash_2 B
\end{align*}
\]

are subproofs of \( \mathcal{I} \), a fixed set of proofs, then the proof

\[
\begin{align*}
\Gamma, \Delta & \vdash_1 \Gamma \\
\vdash_1 A & \equiv \Delta \\
B & \vdash_2 B
\end{align*}
\]

is an element of the collection over \( \mathcal{I} \).

The key idea is that the direct truth content of the set of proofs \( \mathcal{I} \) is given by the conclusions of the collected proofs without undischarged assumptions.

There are many ways to mathematically formalize the previous intuitions; we will introduce an abstract description of the algorithm we employ in our verification system.

**Definition 6.1.3** Let \( \mathcal{I} \) be a set of proofs; \( \text{Coll}(\mathcal{I}) = \bigcup_{i \in \omega} \text{Coll}^i(\mathcal{I}) \), where

\[
\begin{align*}
\text{Coll}^0(\mathcal{I}) & = \emptyset \\
\text{Coll}^{i+1}(\mathcal{I}) & = \text{Coll}^i(\mathcal{I}) \cup \left\{ \begin{array}{l}
\Gamma \vdash_1 \Gamma \\
\vdash_1 A \\
A \equiv \Delta \\
\vdash_2 B
\end{array} \in \text{Subproof}(\mathcal{I}) \land \right.
\end{align*}
\]

\[
\land \Gamma \subseteq \{ \phi \mid \exists \Pi, \phi \text{ is the conclusion of } \Pi \land \Pi \in \text{Coll}^i(\mathcal{I}) \} .
\]

**Definition 6.1.4** Let \( \mathcal{I} \) be a set of proofs,

\[
\text{Inf}(\mathcal{I}) = \{ \phi \mid \phi \text{ is the conclusion of a proof in } \text{Coll}(\mathcal{I}) \} .
\]

Also,

\[
\text{Inf}^i(\mathcal{I}) = \{ \phi \mid \phi \text{ is the conclusion of a proof in } \text{Coll}^i(\mathcal{I}) \} .
\]

**Lemma 6.1.1** \( \bigcup_{i \in \omega} \text{Inf}^i(\mathcal{I}) = \text{Inf}(\mathcal{I}). \)

**Proof:** By unfolding definitions, it is obvious that

\[
\begin{align*}
\text{Inf}^0(\mathcal{I}) & = \emptyset \\
\text{Inf}^{i+1}(\mathcal{I}) & = \left\{ \phi \mid \exists \vdash_1 \Gamma, \Gamma, \vdash_1 \phi, \phi, \phi, \phi \in \text{Coll}^i(\mathcal{I}) \right\} .
\end{align*}
\]

Hence, \( \text{Inf}^0(\mathcal{I}) \subseteq \text{Inf}(\mathcal{I}) \), and \( \text{Inf}^{i+1}(\mathcal{I}) \subseteq \text{Inf}(\mathcal{I}) \), since \( \text{Coll}^{i+1}(\mathcal{I}) \subseteq \text{Coll}(\mathcal{I}) \), by definition.
6.1. THE COLLECTION METHOD

Thus, by induction on natural numbers, for every \( i \), \( \text{Inf}^i(\mathcal{I}) \subseteq \text{Inf}(\mathcal{I}) \), which implies \( \bigcup_{i \in \omega} \text{Inf}^i(\mathcal{I}) \subseteq \text{Inf}(\mathcal{I}) \).

Supposing \( A \in \text{Inf}(\mathcal{I}) \), by definition, there is a proof \( \vdash_A \in \text{Coll}(\mathcal{I}) \), but it means that there is an index \( i \), such that \( \vdash_A \in \text{Coll}^i(\mathcal{I}) \), that is, \( A \in \text{Inf}^i(\mathcal{I}) \). So \( \text{Inf}(\mathcal{I}) \subseteq \bigcup_{i \in \omega} \text{Inf}^i(\mathcal{I}) \). \( \square \)

We have to remark that the definition of Coll provides an abstract description of an algorithm to implement this operator; the same holds for the definition of Inf.

Managing Eigenvariables

As we stated in the beginning of the previous subsection, the second main operation underlying the Collection Method, is the instantiation of eigenvariables.

\( \vdash_{\Gamma, A(p)} \)

The idea is simple: if \( \vdash_{\vdash_{B}} \) is a proof depending on the eigenvariable \( p \), \( p \) is free in the proof, hence \( (\vdash_{\vdash_{B}}(p := t)) \), the proof obtained by substituting the term \( t \) for \( p \), is a valid proof. We can enrich the initial set of proofs \( \mathcal{I} \), by adding to it instances of its parametric subproofs.

We need a compromise, of course: we should decide what instances are useful to add, and what are superfluous. Our proposal is to add instances to \( \mathcal{I} \) that do not enlarge the set of conclusions of proofs in \( \text{Coll}(\mathcal{I}) \). Of course they will eventually enlarge the set of conclusions in \( \text{Coll}(\mathcal{I} \cup N) \), where \( N \) is the set of instances.

We will prove in Chapter 8 that this decision suffices for proving that \( \text{IL} \) is uniformly constructive, which was our goal.

**Definition 6.1.5** Let \( \mathcal{I} \) be a set of proofs, we define

\[
\forall I - \text{Sub}(\mathcal{I}) = \left\{ \left( \vdash_{\vdash_{B}} (p := t) \right) \mid \Gamma \cup \{ A(t) \} \in \text{Inf}(\mathcal{I}) \land \right. \\
\left. \forall x. A(x) \right\}. 
\]

**Definition 6.1.6** Let \( \mathcal{I} \) be a set of proofs, we define

\[
\exists E - \text{Sub}(\mathcal{I}) = \left\{ \left( \vdash_{\vdash_{B}} (p := t) \right) \mid \Gamma \cup \Delta \cup \{ A(t) \} \in \text{Inf}(\mathcal{I}) \land \\
\Gamma, [A(p)] \land \exists x. A(x) \right\}. 
\]
Since the $\forall I$ and the $\exists E$ inference rules are the only parametric rules in $IL$, the previous definitions covers the whole set of possible instantiations we allow with our policy.

**Definition 6.1.7** Given a set $\mathcal{I}$ of proofs, we define the expansion operator as

$$\text{Exp}(\mathcal{I}) = \mathcal{I} \cup \forall I-\text{Sub}(\mathcal{I}) \cup \exists E-\text{Sub}(\mathcal{I}) .$$

The expansion operation can be iterated, producing more and more proofs to analyze; since the Exp operator is monotone with respect to set inclusion, it has a least fixed point with a standard characterization. The limit of this construction provides a set of proofs and the information we extract from it via the Inf operator is the relevant constructive part of the truth content of the initial set of proofs.

**Definition 6.1.8** Let $\mathcal{I}$ be a set of proofs,

$$\text{Exp}^0(\mathcal{I}) = \mathcal{I}$$

$$\text{Exp}^{i+1}(\mathcal{I}) = \text{Exp}(\text{Exp}^i(\mathcal{I})) ;$$

The closure is

$$\text{Exp}^*(\mathcal{I}) = \bigcup_{i \in \omega} \text{Exp}^i(\mathcal{I}) .$$

**Definition 6.1.9** Let $\mathcal{I}$ be a set of proofs,

$$\text{Coll}^*(\mathcal{I}) = \bigcup_{i \in \omega} \text{Coll}(\text{Exp}^i(\mathcal{I}))$$

and

$$\text{Inf}^*(\mathcal{I}) = \bigcup_{i \in \omega} \text{Inf}(\text{Exp}^i(\mathcal{I}))$$

The goal of an algorithm implementing the Collection Method is to generate $\text{Inf}^*(\mathcal{I})$, where $\mathcal{I}$ is the input, a finite set of proofs. It should be evident that, when $\mathcal{I}$ is finite, $\text{Inf}^*(\mathcal{I})$ can be computed, and it should be obvious that a lazy generation of $\text{Inf}^*(\mathcal{I})$ is the best way to compute it, since it can be infinite. More than this, it is easy to prove (Miglioli and Ornaghi, 1978) that $\text{Inf}^*(\mathcal{I})$ is recursively enumerable, but, in general it is not recursive.

The role of the Coll operator is to keep track of the interesting subproofs, while the Inf operator computes the relevant part of the truth content of a set of proofs $\mathcal{I}$. The words interesting and relevant are strictly related to the computational meaning of $\mathcal{I}$. The interesting property of $\text{Inf}^*(\mathcal{I})$ is that it permits a constructive reading of the logic $IL$, as well as a computational reading of formulas. In Chapter 8 we will prove that $\text{Inf}^*(\mathcal{I})$ is a pseudo-truth set; here we will comment this definition with respect to the computational reading we were speaking before:
6.1. \textbf{THE COLLECTION METHOD}

- $A \land B \in \text{Inf}^*(\mathcal{I})$ implies $A \in \text{Inf}^*(\mathcal{I})$ and $B \in \text{Inf}^*(\mathcal{I})$; if $A \land B$ is a specification, it means that a program which computes $A \land B$, has to compute both $A$ and $B$.

- $A \lor B \in \text{Inf}^*(\mathcal{I})$ implies $A \in \text{Inf}^*(\mathcal{I})$ or $B \in \text{Inf}^*(\mathcal{I})$; if $A \lor B$ is a specification then a program which computes $A \lor B$ has to compute $A$ or to compute $B$.

The other way around, the computational interpretation of a disjunction as a decision procedure between two alternatives is sound with the constructive reading we adopted in the whole thesis.

- $\exists x. A(x) \in \text{Inf}^*(\mathcal{I})$ implies that there is a term $t$ such that $A(t) \in \text{Inf}^*(\mathcal{I})$; being $\exists x. A(x)$ a specification, we have to compute a term $t$ which satisfies $A(t)$. We can find this witness in $\text{Inf}^*(\mathcal{I})$, thus, again, the constructive reading and the computational reading coincide.

The Collection Method, as we will prove in Chapter 8, provides a general way to show that the constructive reading of a specification formula and its computational reading coincide.

6.1.2 \textbf{The E Logic}

The key idea underlying the Collection Method we tried to suggest in the previous exposition, is that the information extraction procedure is largely independent from the specific format of axioms and rules; the relevant notion is that the information we need is contained in the proofs.

In fact, the Collection Method for the E logic in its natural deduction presentation, is, essentially, the same as for \textbf{IL}. The definition of the Subproof operator is the obvious expansion of the definition we gave for \textbf{IL}; the Coll and Inf operators are exactly the same as for \textbf{IL}.

The definition of the expansion operator Exp has to take in account also the negated parametric rules $\neg \forall E$ and $\neg \exists I$.

**Definition 6.1.10** Let $\mathcal{I}$ be a set of proofs;

\[\neg \forall \text{E-Sub}(\mathcal{I}) = \left\{ \begin{array}{c} \left( \Gamma, \neg A(p) \right) \quad (p := t) \quad \Gamma \quad \Gamma, [\neg A(p)] \quad \Gamma, \neg x. A(x) \quad B \quad \in \text{Coll}(\mathcal{I}) \land \\
\Lambda \Gamma \cup \{ \neg A(t) \} \subseteq \text{Inf}(\mathcal{I}) \end{array} \right\} \]
\[
\neg \exists \text{I-Sub}(I) = \left\{ \begin{array}{c}
\left( \begin{array}{c}
\Gamma \\
\vdots \\
\neg A(p)
\end{array} \right) (p := t) \\
\end{array} \right| \\
\Gamma \vdash \neg A(p) \in \text{Coll}(I) \land \\
\exists x. A(x) \land \\
\Gamma \cup \{ \neg A(t) \} \subseteq \text{Inf}(I) \right\}.
\]

Consequently

\[\text{Exp}(I) = I \cup \forall \text{I-Sub}(I) \cup \exists \text{E-Sub}(I) \cup \forall \text{E-Sub}(I) \cup \neg \exists \text{I-Sub}(I) \] .

In the standard way one defines Exp*, Coll* and Inf*.

As we will prove in Chapter 8, the computational and the constructive reading of specifications in the E system coincide, in the same way as we already remarked for IL. We want to remark that the negation in E is substantially different from the intuitionistic negation. In IL, in fact, \(\neg A\) is equivalent to \(A \rightarrow \bot\), while in E the negation, being constructible, has to satisfy De Morgan’s Laws.

We want to observe here some important facts about the Collection Method:

- the exact definition of the Subproof operator depends on the calculus, but the notion of subproof is largely independent; as a result, we have that, for any calculus, the definition of the Subproof operator appears to be trivial.

- The definitions of the Coll and Inf operators are independent from the calculus; they just depend on the notion of subproof.

- The Exp operator depends on the substitution of eigenvariables in the proofs; we note that the way Exp is defined, and how the substitution operators \(\forall \text{I-Sub}, \exists \text{E-Sub}, \neg \exists \text{I-Sub} \) and \(\neg \forall \text{E-Sub}\) are defined, does not really depend on the system, but they are instances of a common pattern.

- The \(\Box\) rules, being non parametric, do not affect the construction of Coll* and Inf*.

### 6.1.3 □-Theories

In the logical system E, □-theories are the equivalent of Harrop formulas in intuitionistic logic. We have already shown their definition in Chapter 5.

The following facts can be proven in E:

- \(\Box(A \land B) \leftrightarrow \Box A \land \Box B\);
- \(\Box(A \rightarrow B) \leftrightarrow (A \rightarrow \Box B)\);
- \(\Box \forall x. A \leftrightarrow \forall x. \Box A\).
Using these facts, by induction on the definition of $\mathbf{E}$-Harrop formula, one proves

**Lemma 6.1.2** For any $\mathbf{E}$-Harrop formula $H$, $\vdash \square H \iff H$.

One may note that the fact $\square \forall x. A \iff \forall x. \square A$ is an alternative way to express the Kuroda principle in the $\mathbf{E}$ logic; in fact the $\square$ connective acts very similarly to double negation in $\mathbf{IL}$.

The importance of $\square$-theories comes from Chapter 5, where we imposed that the axioms of an open framework have to be $\mathbf{E}$-Harrop formulas; hence, we want an instance of the Collection Method which permits to extract information from a system $\mathbf{E} + T$, where $T$ is a $\square$-theory.

We consider the system $\mathbf{E} + T$ composed by the same inference rules as $\mathbf{E}$ plus

$$-\phi \in T.$$

The definitions of the basic operators, Subproof, Coll and Inf is the same as for the $\mathbf{E}$ logic. The definition of the $\text{Exp}$ operator is enlarged to take into account the peculiar nature of the $\square$-theory $T$; we define

**Definition 6.1.11**

\[
\begin{align*}
\text{Har}_\land\text{-Sub}(I) & = \left\{ \frac{A \land B}{A} \mid A, B \in T \right\} \\
\text{Har}_\rightarrow\text{-Sub}(I) & = \left\{ \frac{A \rightarrow B}{A} \mid A \rightarrow B \in T \land A \in \text{Inf}(I) \right\} \\
\text{Har}_\forall\text{-Sub}(I) & = \left\{ \frac{\forall x. A(x)}{A(t)} \mid \forall x. A(x) \in T \land A(t) \in \text{Inf}(I) \right\} \\
\text{Har}\text{-Sub}(I) & = \text{Har}_\land\text{-Sub}(I) \cup \text{Har}_\rightarrow\text{-Sub}(I) \cup \text{Har}_\forall\text{-Sub}(I) .
\end{align*}
\]

Thus, calling $\text{Exp}_\mathbf{E}$ the expansion operator for the $\mathbf{E}$ logic, the $\text{Exp}$ operator becomes

$$\text{Exp}(I) = \text{Exp}_\mathbf{E}(I) \cup \text{Har}\text{-Sub}(I).$$

Then, in the standard way, we define $\text{Coll}^*$ and $\text{Inf}^*$.

The idea behind the definition of the $\text{Har}\text{-Sub}$ operator is the same as we have seen for $\text{Exp}_\mathbf{IL}$: the expansion operator provides a way to *dismount* the subproofs of a set of proofs $I$ which cannot be managed by the Coll operator. One way is to instantiate eigenvariables, another way is to apply the appropriate elimination rules to the $\mathbf{E}$-Harrop axioms.

In Chapter 8, we will prove that the $\mathbf{E}$ system plus a $\square$-theory enjoys the nice properties permitting to interpret specifications both in a constructive and in a computational way, as we illustrated before.
6.1.4 Identity

The axioms about equality have a special role in logic, because the intended meaning of this relation is fixed.

The theory ID for equality is composed by an axiom and a rule:

\[
\begin{align*}
\text{Ref} & : \quad t = t \\
\text{Sub} & : \quad x = y \quad A(y) \quad \Rightarrow \quad A(x)
\end{align*}
\]

Usually, and this is the case for IL and E, we can impose that \(A(x)\) in the Sub rule is atomic; then we can prove the general case by induction on the structure of formulas.

To extract information from the system \(L + \text{ID}\), where \(L\) is any logical system, we construct, in the standard way, the Subproof, the Coll and the Inf operators.

As usual, we have to enlarge the \(\text{Exp}_L\) operator:

**Definition 6.1.12** Let \(\mathcal{I}\) be a set of proofs, the substitution operator for the ID theory is defined as

\[
\text{ID-Sub}(\mathcal{I}) = \left\{ \frac{t = s\quad A(s)}{A(t)} \right\} \quad \{t = s, A(s) \subseteq \text{Inf}(\mathcal{I})\}.
\]

So, \(\text{Exp}(\mathcal{I})\) becomes \(\text{Exp}_L(\mathcal{I}) \cup \text{ID-Sub}(\mathcal{I})\).

Then we construct Coll*(\(\mathcal{I}\)) and Inf*(\(\mathcal{I}\)) following the usual guidelines.

In Chapter 8 we will prove that, if \(L\) has the property of permitting both a constructive and a computational reading of specifications, then \(L + \text{ID}\) has the same property.

The most important remark we have to make, is that the Collection Method appears to be largely independent from the specific format of axioms and rules of the logical system we are examining.

We treated in a uniform way two different logics, IL and E, and a number of theories, covering everything we can express through the specification framework approach. The way to build up an instance of the Collection Method for a theory we can generate inside our Constructive Verification Environment can be automated, thus giving a real way to analyze proofs by extracting information.

In Chapter 8 we will introduce the notion of *uniformly constructive formal system*, which is the formal way to analyze the Collection Method itself; in a uniformly constructive formal system we have the coincidence of the constructive and the computational reading of specifications, thus making our choice of the instrument to analyze proofs perfectly homogeneous with the notion of specification framework.

6.1.5 Induction Principles

As we have seen in Chapter 5, most theories need one or more induction principles; in that chapter, we gave them a computational reading by showing how they become program schemas involving recursion or cycles.
6.1. THE COLLECTION METHOD

To treat them in our method to extract information from proofs we should keep in mind that induction rules are parametric, i.e., they have eigenvariables. We treat eigenvariables by enlarging the definition of the Exp operator, eventually adding proofs which make explicit some aspects of the induction.

In the following we will show how to treat induction in Peano arithmetic, and the Descending Chain Principle. These two examples will clarify the general technique, and, at the same time, they show that our infrastructure, as developed in the previous chapters, is homogeneous.

We introduced Peano arithmetic in Chapter 4; here we note that the logical system $\mathbf{E} + \mathbf{PA}$ can be reduced to $\mathbf{E} + T + \text{Ind}$, where $T$ is a $\Box$-theory and Ind is the induction principle on natural numbers, i.e.,

\[
\begin{align*}
[A(p)] \\
\vdots \\
A(0) \quad A(s(p)) \\
\hline
A(t) \quad \text{Ind}
\end{align*}
\]

The way to treat induction is to enlarge the Exp operator as defined for the system $\mathbf{E} + T$; let us call it $\text{Exp}_{\mathbf{E}+T}$.

**Definition 6.1.13** Let $\mathcal{I}$ be a set of proofs, we define $\text{Ind-Sub}(\mathcal{I})$ as the smallest set such that, if,

\[
\begin{align*}
\Gamma & \vdash [A(p)] \\
\vdots & \vdash (p) \\
A(0) & \quad A(s(p)) \\
\hline
A(t) \quad \text{Ind}
\end{align*}
\]

with $t$ a closed term, and $\Gamma \cup \{A(t)\} \subseteq \text{Inf}(\mathcal{I})$, then

\[
\begin{align*}
\{ & t = s^i(0), A(s^i(0)) \\
\vdots & \\
& t = s^i(0), A(t) \\
\} \cup \\
\cup \left\{ \left( \Gamma, A(p) \right) \left| (p := s^j(0)) \quad 0 \leq j < i \right\} \subseteq \text{Ind-Sub}(\mathcal{I}),
\end{align*}
\]

where $t = s^i(0)$ is any proof which converts $t$ to its canonical form, i.e., as a numeral.

We remark that there is a standard way to perform the proof $t = s^i(0)$, as one can deduce from the discussions in Chapter 4.

In this way, in the system $\mathbf{E} + T + \text{Ind}$, one defines

\[
\text{Exp}(\mathcal{I}) = \text{Exp}_{\mathbf{E}+T}(\mathcal{I}) \cup \text{Ind-Sub}(\mathcal{I}).
\]
Then, we can construct in the standard way \( \text{Coll}^*(I) \) and \( \text{Inf}^*(I) \), for any set of proofs \( I \).

In Chapter 8 we will prove that, restricting ourselves to closed proof, i.e., to proofs where the only occurrences of variables are eigenvariables, the system \( E + T + \text{Ind} \) has the usual coincidence of the constructive and computational readings of specifications, this confirming the ideas on program synthesis we illustrated in Chapter 5.

The Descending Chain Principle is treated in a singular way; let us suppose to work in the logical system \( E + T + \text{DCP} \), where \( T \) is any \( \Box \)-theory whose signature contains a binary relational symbol \(<\), and let us suppose that the intended model of \( T \) makes \(<\) to be interpreted as a well-founded ordering.

The Descending Chain Principle has the shape:

\[
\begin{align*}
[B(p)] \\
\vdots \\
\exists x, B(x) & (\exists z.B(z) \land z < p) \lor A & \text{DCP}
\end{align*}
\]

As before, being DCP a parametric rule, we enlarge \( \text{Exp}_{E+T} \):

**Definition 6.1.14** Let \( I \) be a set of closed proofs

\[
\text{DCP-Sub}(I) = \left\{ \left( \Gamma, B(p) \middle| p := t \right) \mid \Gamma \cup \{B(t)\} \subseteq \text{Inf}(I) \land \\
\left\{ \vdots \\
(\exists z.B(z) \land z < p) \lor A & \text{[B(p)]} \\
\vdots \\
\exists x, B(x) & (\exists z.B(z) \land z < p) \lor A \in \text{Coll}(I) \right\} \right. \\
\vdots \\
\right\}.
\]

The \text{Exp} operator becomes \( \text{Exp}(I) = \text{Exp}_{E+T}(I) \cup \text{DCP-Sub}(I) \), and, as usual, one applies the standard construction to build \( \text{Inf}^*(I) \).

As for the other cases we treated, in Chapter 8 we will show that the correspondence between the constructive and the computational views is preserved.

### 6.2 How to Distinguish Relevant Information

In the previous section, we have seen that it is possible to extract information from a formal proof; we have remarked that the set of facts, i.e., \( \text{Inf}^*(I) \), we extract, may be infinite.

Of course, most of the content of \( \text{Inf}^*(I) \) is useless from the point of view of the user; many facts are needed by the extraction algorithms because it has to have enough information to be able to collect new proofs via the \text{Exp} operator; many facts are simply collected because they are there, but we have no interest in them.

There are two important points to remark:
• the Collection Method is not intrinsically oriented. We mean that the procedure to extract information we described is not conceived to produce the minimal amount of facts in a proof which permits to redo the proof itself, as, for example, is the case of normalization based techniques (Benl et al., 1998; Berger and Schwichtenberg, 1995; Schwichtenberg, 1993); the Collection Method produces a set of facts which the proof makes true, either explicitly or implicitly.

• the notion of what is interesting depends on the user. We mean that, in our belief, is up to the user to choose what kind of information has to be considered as relevant. Moreover, we are not allowed to assume to know in advance, before doing a proof, what kind of information the user wants to extract.

Hence, the Collection Method is the right procedure to analyze correctness proofs, because it permits to query the proof for the kind of facts we consider as relevant, thus solving the second remark.

Of course, we have to pay a price: being non oriented, the Collection Method can be very inefficient, that is, it may take a great number of steps to produce the information we are interested in.

We see the need for two implementation techniques: orientation and filtering. The former takes care of directing the Collection Method on trying to extract the relevant information as soon as possible; the latter discriminates between interesting and non interesting facts.

The idea behind orientation techniques is to implement the algorithm which constitutes the Collection Method using ML in the lazy functional style, that is, it has to compute step by step, producing a series of approximations \( I_0, I_1, \ldots \) which tend to \( \text{Inf}^*(I) \).

These approximate results are already present in the definition of \( \text{Inf}^*(I) \) as the \( \text{Inf}^r(I) \) steps.

We are not bound to produce exactly \( \text{Inf}^*(I) \) at every step \( i \); as soon as we are able to ensure that \( \bigcup_{i \in \omega} I_i = \text{Inf}^*(I) \), any sequence of approximations \( I_0, I_1, \ldots \) will be an instance of the Collection Method. Then, we can show to the user a sequence of facts \( F_0, F_1, \ldots \) which is related to \( I_0, I_1, \ldots \) by the formula \( F_i = \tau(I_i) \), where \( \tau \) is a filtering function which takes care of retaining only the part of \( I_i \) which is relevant for the purposes of the user.

What we discussed till now is the general approach we follow in the Constructive Verification Environment; of course, we have to fix one or more algorithms to generate our approximating sequence \( I_0, I_1, \ldots \), and some filtering functions which permit to extract some information which is commonly considered as relevant in the context of program verification.

There is a wide space for research on the idea of approximation with respect to the Collection Method construction; we have got some results on this field, but still too preliminary to be discussed in this thesis. We plan to continue on this research line in the near future.
For the moment being, we make use of a marking superstructure for proofs; some nodes in a proof tree are marked, and we treat in a privileged way the subproofs whose last inference step is marked.

When we have to apply the basic operators Subproof, Coll, Inf and Exp to a set of proofs \( \mathcal{I} \) with a marking superstructure \( \mathcal{M} \), we choose to generate first the subproofs of the marked parts, and we collect the information giving maximum priority to the marked subproofs. We have also tried to use markers which are natural numbers, with different combination of rules to spread the markers when running the extraction algorithm.

The results are encouraging, and we summarize them here:

- The marking superstructure, imposing an ordering on the subproofs, is fair; that is, the approximating sets of facts it permits to extract from \( \mathcal{I} \) tend to the limit \( \text{Inf}^*(\mathcal{I}) \).

- The sets the extraction algorithm produces first are more significant, that is, they contain more interesting information.

- It seems harder for this specialization of the Collection Method to provide deeply hidden interesting information, that is, facts which are conclusions of subproofs generated by the Exp operator after a number of iterations. This result is not surprising since the Exp operator is effective when a large amount of facts is extracted in every phase, so to produce the possibility for unusual combinations of subproofs.

We studied also how to automatically mark points in a correctness proof in a sensible way. Our heuristic is to extract information from the user: we mark the subproofs which are proved by the user, while we do not mark the proofs which are automatically solved by our tactics.

The idea behind this choice is that the user does significant proofs, while leaving trivial proofs to the automatic provers, see also Chapters 3 and 4.

It seems that this heuristic works, thus extracting information the user recognizes as related to the proof, while, in the plain version of the Collection Method, the user sometimes ask questions like “Where does this fact come from?”, indicating that some information is considered as extraneous.

We experimented also another way of orienting the extraction procedure; we may divide a complete correctness proof \( \Pi \) into a set \( \mathcal{L} = \{\Pi_1, \ldots, \Pi_n\} \) of lemmas whose composition reconstitute \( \Pi \). In this way we really discard a part of the information we extract from \( \Pi \); formally one may prove that \( \text{Inf}^*(\mathcal{L}) \subseteq \text{Inf}^*(\{\Pi\}) \), but, in general, equality does not hold.

Hence, this technique, we call it lemmification, does not respect the idea of approximating \( \text{Inf}^*(\{\Pi\}) \) by an appropriate sequence of sets of facts, but, rather, it really discards parts which, a priori, we consider as non relevant.

We adopted lemmification as the standard way to analyze proofs where some parts are proved by our automatic decision procedures.
6.2. HOW TO DISTINGUISH RELEVANT INFORMATION

For example, if the proof $\phi^1$ is divided into $L = \left\{ \phi^1, \phi^2 \right\}_{A}$, where $\phi^2$ has been proven by the SUP-INF algorithm, see Chapter 4, then we mark the proof $\phi^1$, thus we obtain as a result that

1. First, we produce results about $\phi^1$, because it is marked, then about $\phi^2$.

2. We will never collect information by examining instances of subproofs of $\phi^1$ which are neither subproofs of $\phi^2$ or subproofs of $\phi^2$.

At this point it should be clear to the reader the purpose of our techniques to orient the extraction procedure; on one side, we mark some parts to introduce a priority measure in the process; on the other side, we discard information which comes from the combination of unrelated proofs.

We know that these techniques are quite rough, and do not constitute a refined solution to the problem of orienting the information extraction process, but, as we said before, at the moment, no other technique is available.

The problem of filtering a sequence of facts $I_0, I_1, \ldots$, to produce a similar sequence $F_0, F_1, \ldots$, where every $F_i$ contains just the relevant elements of $I_i$, is of a different nature. In fact, the operation of filtering is very easy to describe from a technical point of view, but it requires a formal definition of the word relevant.

As we remarked in the beginning of this section, we do not want to fix once and forever what we consider as relevant, but, rather, to provide a flexible frame which permits to produce different kinds of information in a homogeneous way.

In our opinion, we must provide at least a tool which filters the facts which directly express what is valid in the program code. We developed this algorithm and we will show it in the following section.

On the other side, a great amount of interesting facts are collected in the extraction process, and, yet, we have not a syntactical way to characterize them. As an example, in the following section, we will show how to extract complexity measures for the simple examples we will show.

Again, as for the orientation techniques, we feel the need for a deeper series of theoretical results; without the ability to characterize classes of formulas by their role in a correctness proof, it appears to be very hard to describe in a formal, syntactical way relevant properties of programs we wish to filter out from the result Inf* produces.
1: MOVE # − 1, d1  
2: ADD #1, d1  
3: MOVE d1, d2  
4: ADD d2, d2  
5: CMP d2, d0  
6: BGE 2  
7: SUB #1, d1

Figure 6.1: An Example Program for Computing Division by Two.

6.2.1 Labelling Algorithm

As we anticipated in the previous discussion, we will show now a simple filtering algorithm which extracts information on what is true in specific points of an object code program.

We assume that the program P we are analyzing is coded into the logical representation RepP, as described in Chapter 7.

\[
\text{Rep}_P
\]

**Definition 6.2.1** Let \( \text{Spec}_P \) be a correctness proof for the program P, coded in a logical form as RepP, with respect to the specification SpecP; let \( C = \text{Inf}^* \left( \left\{ \text{Rep}_P \right\} \right) \).

We define

\[
\mathcal{L}_i = \{ \exists x. A | (\exists x. \text{pc}(x) = i + 1 \land A) \in C \} \cup \{ \forall x. A | (\forall x. \text{pc}(x) = i + 1 \rightarrow A) \in C \} ,
\]

for every instruction, referred to by its position i inside the program code.

We define \( \mathcal{G} = \{ \forall x. A | \forall x. A \in C \} \setminus \bigcup_i \mathcal{L}_i \).

Intuitively \( \mathcal{L}_i \) is a set of assertions which hold on the \( i^{th} \) position (line) of the source code, while \( \mathcal{G} \) contains a set of facts which are true everywhere in the program.

Consider the assembly code program in Figure 6.1: given a natural number in register \( d_0 \), it is divided by 2 and the result is returned in register \( d_1 \). Formally:

\[
\text{Spec} \equiv \exists t. \text{pc}(t) = 8 \land (d_0(0) = 2d_1(t) \lor d_0(0) = 2d_1(t) + 1)
\]

The logical representation for this program is shown in Figure 6.2.

The correctness proof for this program has been developed in the E logic with the aid of the Computer Arithmetic Toolkit. The essential schema for the correctness proof can be found in Figure 6.3.
6.2. HOW TO DISTINGUISH RELEVANT INFORMATION

\[
\begin{align*}
\text{Rep} & \equiv \text{pc}(0) = 1 \land 0 \leq d_0(0) \land (\forall t. I_1(t) \land \ldots \land I_5(t)) \land \\
& \quad (\forall t. \text{pc}(t) = 1 \lor \ldots \lor \text{pc}(t) = 8) \\
I_1(t) & \equiv \text{pc}(t) = 1 \rightarrow \text{pc}(t + 1) = 2 \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = -1 \\
I_2(t) & \equiv \text{pc}(t) = 2 \rightarrow \text{pc}(t + 1) = 3 \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = 1 + d_1(t) \\
I_3(t) & \equiv \text{pc}(t) = 3 \rightarrow \text{pc}(t + 1) = 4 \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = d_1(t) \land \\
& \quad \land d_2(t + 1) = d_1(t) \\
I_4(t) & \equiv \text{pc}(t) = 4 \rightarrow \text{pc}(t + 1) = 5 \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = d_1(t) \land \\
& \quad \land d_2(t + 1) = 2d_2(t) \\
I_5(t) & \equiv \text{pc}(t) = 5 \rightarrow \text{pc}(t + 1) = 6 \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = d_1(t) \land \\
& \quad \land (N(t + 1) \leftrightarrow d_1(t) \leq d_0(t)) \\
I_6(t) & \equiv \text{pc}(t) = 6 \rightarrow (N(t) \rightarrow \text{pc}(t + 1) = 2) \land (-N(t) \rightarrow \text{pc}(t + 1) = 7) \land \\
& \quad \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = d_1(t) \\
I_7(t) & \equiv \text{pc}(t) = 7 \rightarrow \text{pc}(t + 1) = 8 \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = d_1(t) - 1 \\
I_8(t) & \equiv \text{pc}(t) = 8 \rightarrow \text{pc}(t + 1) = 8 \land d_0(t + 1) = d_0(t) \land d_1(t + 1) = d_1(t)
\end{align*}
\]

Figure 6.2: The Logical Representation for the Example Program.

\[
A \equiv \text{pc}(t) = 2 \rightarrow (2(1 + d_1(t)) \leq d_0(t) \rightarrow \text{pc}(t + 5) = 2) \land \\
& \quad \land (d_0(t) < 2(1 + d_1(t)) \rightarrow \text{pc}(t + 5) = 7) \land \\
& \quad \land d_1(t + 5) = d_1(t)
\]

\[
B \equiv \forall t. d_0(t) = d_0(0)
\]

\[
C \equiv \exists t. \text{pc}(t) = 2 \land d_0(t) < 2(1 + d_1(t))
\]

\[
D \equiv \forall t. \text{pc}(t) = 2 \rightarrow 2d_1(t) \leq d_0(t)
\]

\[
\begin{array}{c|c|c|c|c}
\text{Rep} & \text{Rep} & \text{Rep} & \text{Rep} & \text{Rep} \\
\hline
A & B & C & D & \text{Spec} \\
\hline
\exists t. 2(d_1(t) - 1) \leq d_0(0) \leq 2d_1(t) - 1 \land \text{pc}(t) = 7 & \text{Rep} & \exists \text{E}
\end{array}
\]

Figure 6.3: The Schema of the Correctness Proof for Our Example.
: $\mathcal{L}_0 = \{0 \leq d_0(0)\}$

1: MOVE $\# - 1, d_1$

: $\mathcal{L}_1 = \{\exists t. d_0(t) < 2(d_1(t) + 1), \forall t. 2d_1(t) < d_0(t)\}$

2: ADD $\#1, d_1$
3: MOVE $d_1, d_2$
4: ADD $d_2, d_2$
5: CMP $d_2, d_0$
6: BGE 2

: $\mathcal{L}_6 = \{\exists t. 2(d_1(t) - 1) \leq d_0(0) \leq 2d_1(t) - 1\}$

7: SUB $\#1, d_1$

: $\mathcal{L}_7 = \{\exists t. d_0(0) = 2d_1(t) \lor d_0(0) = 2d_1(t) + 1\}$

Figure 6.4: The Result of the Labeling Algorithm on Our Example Program.

Applying the Collection Method algorithm to this proof where the schema is considered as marked, and filtering the result according to Definition 6.2.1, we get that the set of facts which are true in the whole program, and which are relevant for its correctness are

$\mathcal{G} = \{\forall t. d_0(t) = d_0(0)\}$

while the labels indexed by the instruction number are

\[
\begin{align*}
\mathcal{L}_0 &= \{0 \leq d_0(0)\} \\
\mathcal{L}_1 &= \{\exists t. d_0(t) < 2(1 + d_1(t)), \forall t. 2d_1(t) < d_0(t)\} \\
\mathcal{L}_2 = \mathcal{L}_3 = \mathcal{L}_4 = \mathcal{L}_5 &= \emptyset \\
\mathcal{L}_6 &= \{\exists t. 2(d_1(t) - 1) \leq d_0(0) \leq 2d_1(t) - 1\} \\
\mathcal{L}_7 &= \{\text{Spec}\}
\end{align*}
\]

Thus, we can automatically label the source code, obtaining Figure 6.4.

### 6.3 Program and Specification Analysis

To analyze a correctness proof is not different from analyzing any other proof. In fact, a correctness proof is not a special kind of proof, except for its meaning. We want to remark that the meaning we are speaking of, is not intrinsic to the
proof, and this is especially true in our constructive setting. Any proof, in a sense, is a correctness proof for the program we can extract from it, either using the synthesis techniques presented in Chapter 5, or using the Collection Method as a universal computing machine (Miglioli and Ornaghi, 1978; Miglioli and Ornaghi, 1981; Miglioli and Ornaghi, 1981b). The coincidence of the constructive and the computational interpretations of formulas is what permits the execution of a proof as a program.

So it make sense to speak of program analysis, intending with these words the analysis of correctness proofs, taking care of their meaning. The labelling algorithm is a first step in this direction: it is conceived to work on a correctness proof, and interprets some of its formulas in a special way; in particular, it is aware of the semantics of the pc function.

Other types of program analysis tools are possible; just as an example, we considered to develop in the near future a tool, similar in its structure to the labelling algorithm, which extracts information about the computational complexity of a program.

In the example of the previous section, it would provide a set of labels like the following:

\[ \mathcal{L}_0 = \{1\} \]
\[ \mathcal{L}_1 = \{2\} \]
\[ \mathcal{L}_2 = \{3, 6, 11, \ldots\} \]
\[ \mathcal{L}_3 = \{4, 7, 12, \ldots\} \]
\[ \mathcal{L}_4 = \{5, 8, 13, \ldots\} \]
\[ \mathcal{L}_5 = \{6, 9, 14, \ldots\} \]
\[ \mathcal{L}_6 = \{x\} \]
\[ \mathcal{L}_7 = \{x + 1\} \]

where \( x \) depends on the value of \( d_0(0) \) and \( \mathcal{L}_i \) represents the time after the execution of the instruction \( i \).

This kind of information can be generated automatically searching for the witnesses of the existential labels in Figure 6.4. Obviously, closed forms for the equations describing the value of time after each instruction will be collected, if they were used in the correctness proof.

There are many other possible ways to perform program analysis following similar schemas.

About the value of the information we collect using the labelling algorithm, or other program analysis tools, we may cite the help such information give to the reusability problem, the software maintenance, the documentation problem. Although we believe that all these aspects are important, we prefer not to continue the discussion since we think that a number of real case studies have to be produced to really examine these topics.

But we have to say that, although our actual results may appear incomplete (and they are, as we said also in the previous section), we have enough of them to tackle one of the main problems in formal verification.
As we already explained in Chapters 1 and 2, convincing a non-expert about the validity of a correctness proof is hard since a formal proof is huge and difficult to understand. Our labelling algorithm, because of the way it works, provides information a programmer can check, and this information is present in the correctness proof, giving empirical evidence of the validity of the proof.

On this line, we want to introduce the concept of specification analysis.

A very big problem when we really try to verify a program, is that its specifications are informal, hence our first step is to try to formalize them. In fact, there is no way to be sure that the formalization we provide for specifications is really what it should be; in other words, we have no formal way to show that the meaning of our formal specifications coincides with the intended meaning of informal specifications.

But the code developers use informal specifications to write down the program.

What we are suggesting is to use program analysis to provide empirical evidence that our formalization of specification is sound with the intended meaning the programmers perceived. Since a programmer can read annotated code, and he is able to recognize when the facts coming out from our correctness proof do not coincide with his intuition on what the code he wrote has to do, we can use this information from humans to check if our formal specifications are consistent with the intuition behind them.

When a programmer says “this fact does not agree with my mental picture of what the program should do here”, we have two possibilities:

- The programmer is wrong; his code works but simply he missed to understand every possible consequence of his own work.

- The programmer is right; our correctness proof is based on a goal which does not closely correspond to what the informal specification says. We should keep in mind that the programmer does not judge if our labels are right or wrong with respect to our specifications but with respect to the real specifications, i.e., the informal ones.

This kind of usage of our information extraction tools, to analyze formal versus informal specifications, is what we call specification analysis.

6.4 Summary

In this chapter we presented the construction underlying the Collection Method; in particular we discussed what kind of techniques can be adopted for an implementation which achieves the goals the Proof Analyzer Package, as introduced in Chapter 2, should meet.

Finally, a general discussion about the role automatic analysis of formal proofs, and, specifically, the labeling algorithm, plays in the field of formal verification, is shown.
Chapter 7

Representing Object Programs

In this chapter we show how to translate object code into a logical representation.

In Chapter 2, we posed this as a requirement to be met by the Constructive Verification Environment. We already used the representation in the example in Chapter 6, and here we will analyze how it is generated starting from the object code.

Most of the content of this chapter has been developed by the author in the Department of Computer Science, University of Warwick (UK), hence we have to acknowledge the contribution of S. Kalvala, D. Nowotka and C. Pulley.

In this respect, it is important to say that the tool we are going to describe has been implemented, tested and used in practice. In fact, in the project described in (Benini et al., 1998a), the translation from object code to higher-order logic has been completely implemented, and, then it has been used to prove the correctness of real code (Benini, 1998b; Benini et al., 1998b). An example of this effort will be presented in Appendix A.

7.1 Translating Object Code into Logic

In Chapter 2, we said that our design for a Constructive Verification Environment has to provide an automatic translation from object code to a logical representation. We also posed the problem of guaranteeing that the representation behaves as the code it represents. Finally, in that chapter, we proposed a representation which is a direct coding of the transformations on registers an instruction performs.

The same representation has been used in the example in Chapter 6, where we collected information via the labelling algorithm over the correctness proof of a simple program.

In this section we present the translation procedure and its main properties with respect to the requirements of Chapter 2.

As we said, the translation procedure takes an object code as input and has to produce a logical representation for it.
The notion of object code is slightly ambiguous; in fact, we may interpret this notion in four different ways:

1. The object code of a program is the content of (the relevant part of) the computer memory when we are going to execute the program.

2. The object code of a program is the content of the executable file we load when we ask the operating system to run the program.

3. The object code for a program is the output of a compiler, hence it is the content of an object (.o) file.

4. The object code is the symbolic representation of the output of a compiler, i.e., it is an assembly program.

We believe that all these possibilities are right, and we support all of them in our translation tool.

The tool is divided into two parts, the preprocessor and the translation procedure; the former transforms an executable file or an object file into an assembly program; the latter takes an assembly program and represents it into a logical format suitable for reasoning with ISABELLE.

The preprocessor is implemented by a simple shell script in Unix, which, using standard programs like objdump, reconstructs an assembly program where every address is resolved, that is, the assembly code is allocated in memory from a given address. Thus, the output of the preprocessor is equivalent to the symbolic (assembly) representation of the content of the memory when the program will be executed.

The translation procedure takes as input an assembly source code where no macros are present and where every address is resolved, and it translates this code in a logical representation.

Our target assembly language is the one of the microprocessor MC68000; there are two main reasons for the choice of this particular architecture:

- the author, as reported in Chapters 2 and 4, worked on a formal verification project in the Department of Computer Science, University of Warwick (UK), where the MC68000 was the target architecture. Hence, we have some expertise on the problems related to formal reasoning on this microprocessor.

- there is a number of case studies for formal verification problems developed over the MC68000 microprocessor; in particular, we can benefit from the good work in (Yu, 1993), where many functions from the standard libc library has been proven correct. Thus, we have a number of test cases for our system, too.

The translation algorithm is, essentially, the same for both versions of the Constructive Verification Environment; the algorithm operates on the language of IL plus modular arithmetic where the types byte, word and longword (integers modulo $2^8$, integers modulo $2^{16}$ and integers modulo $2^{32}$, respectively), see Chapter 4.
The output of the translation procedure is an ISABELLE theory file containing a series of axioms, one per instruction, encoding the program. This theory file inherits from the theory of the microprocessor the necessary type declarations as well as the constants representing registers and memory.

The theory of the microprocessor has three roles:

- it provides the minimal set of instruments to reason about object code programs.
- it declares the types which are needed to represent the code.
- it declares the constants which constitute the world the microprocessor operates on.

The set of instruments is given by the logic (HOL or E, depending on the version of the Constructive Verification Environment), the theory of identity, the Computer Arithmetic Toolkit and their provers, namely, the Simplifier, the Constructive Reasoner, the Rewriting Engine and the set of decision procedures of CAT.

The types are byte, word and longword, used to represent the quantities the microprocessor operates on, and time, which is used to model how the flow of control is passed from one instruction to another.

The types byte, word and longword are specializations of modular numbers, and specifically, byte = Int/( (mod 2)^8), word = Int/( (mod 2)^16), longword = Int/( (mod 2)^32); we use both the signed and unsigned bytes (words, longwords, respectively). Thus, following the syntax of Chapter 4,

\[
\text{byte} = \text{ModInt}_{2^8} \quad \text{sbyte} = \text{SModInt}_{2^8} \quad \text{ubyte} = \text{UModInt}_{2^8} \\
\text{word} = \text{ModInt}_{2^{16}} \quad \text{sword} = \text{SModInt}_{2^{16}} \quad \text{uword} = \text{UModInt}_{2^{16}} \\
\text{longword} = \text{ModInt}_{2^{32}} \quad \text{slongword} = \text{SModInt}_{2^{32}} \quad \text{ulongword} = \text{UModInt}_{2^{32}}
\]

The type time, following the fact that the microprocessor clock is discrete, is equivalent to Int, that is, the time is modeled by integer numbers.

In the microprocessor theory, the constants for memory and registers are declared.

Specifically, the MC68000 microprocessor provides sixteen registers, eight of them being data registers, the others being address registers. The details of the MC68000 architecture can be found in (Motorola, 1989).

We model registers as functions from time to values:

\[
d_i : \text{time} \rightarrow \text{slongword} \quad , 0 \leq i \leq 7 \\
a_i : \text{time} \rightarrow \text{slongword} \quad , 0 \leq i \leq 7 \\
\text{pc} : \text{time} \rightarrow \text{ulongword}
\]

A particular case is the status register which is modeled by a set of functions,
one for each flag in the register:

\[
\begin{align*}
Z\text{flag}: & \text{ time } \to \text{ bool } \quad (* \text{ zero } *) \\
N\text{flag}: & \text{ time } \to \text{ bool } \quad (* \text{ negative } *) \\
C\text{flag}: & \text{ time } \to \text{ bool } \quad (* \text{ carry } *) \\
V\text{flag}: & \text{ time } \to \text{ bool } \quad (* \text{ overflow } *) \\
X\text{flag}: & \text{ time } \to \text{ bool } \quad (* \text{ extension } *)
\end{align*}
\]

The memory is represented as a function from addresses and times to values:

\[
\text{memory: ulongword } \times \text{ time } \to \text{ byte}
\]

It is quite handy to define predicates for reading and writing bytes, words and longwords in memory. We leave to the reader their definitions.

With these information we have the instruments to write the representation of assembly instructions. The general format of the logical representation of an instruction \( I \) is

\[
\forall t: \text{ time}. \text{pc}(t) = A \to B \land C
\]

where \( A \) is the address of the instruction \( I \), \( B \) specifies the value of the program counter at time \( t + 1 \), and \( C \) specifies the value of every register, flag and memory cell at time \( t + 1 \), depending on the instruction operands, the status of memory at time \( t \), and the values of registers and flags at time \( t \).

The format of the \( B \) part can be either

\[
\text{pc}(t + 1) = H(\text{pc}(t))
\]

or

\[
(f(t) \to \text{pc}(t + 1) = H_1(\text{pc}(t))) \land (\neg f(t) \to \text{pc}(t + 1) = H_2(\text{pc}(t)))
\]

where \( H, H_1 \) and \( H_2 \) are expressions depending on the current value of the program counter and calculating the address of the next instruction to execute; the \( f(t) \) is a formula, depending on the time \( t \), and usually, it is a literal representing a flag, but, in general it may be a conjunction/disjunction of (negations of) flag predicates.

For example:

\[
64: \text{MOVE } \#1, d_0
\]

is translated into

\[
\forall t. \text{pc}(t) = 64 \to \text{pc}(t + 1) = 66 \land \\
\land d_0(t + 1) = 1 \land d_1(t + 1) = d_1(t) \land \ldots \land d_7(t + 1) = d_7(t) \land \\
\land a_0(t + 1) = a_0(t) \land \ldots \land a_7(t + 1) = a_7(t) \land \\
\land \neg V\text{flag}(t + 1) \land \neg C\text{flag}(t + 1) \land (Z\text{flag}(t + 1) \leftrightarrow 1 = 0) \land \\
\land (N\text{flag}(t + 1) \leftrightarrow 1 < 0) \land \neg X\text{flag}(t + 1) \land \\
\land \forall a. \text{memory}(a, t + 1) = \text{memory}(a, t)
\]
Also,

\[72: \text{BEQ 8}\]

is translated into

\[
\forall t. pc(t) = 72 \rightarrow (Z\text{flag}(t) \rightarrow pc(t + 1) = pc(t) + 8) \land \\
\land (-Z\text{flag}(t) \rightarrow pc(t + 1) = pc(t) + 2) \land \\
\land d_0(t + 1) = d_0(t) \land \ldots \land d_7(t + 1) = d_7(t) \land \\
\land a_0(t + 1) = a_0(t) \land \ldots \land a_7(t + 1) = a_7(t) \land \\
\land (V\text{flag}(t + 1) \leftrightarrow V\text{flag}(t)) \land \\
\land (Z\text{flag}(t + 1) \leftrightarrow Z\text{flag}(t)) \land \\
\land (N\text{flag}(t + 1) \leftrightarrow N\text{flag}(t)) \land \\
\land (C\text{flag}(t + 1) \leftrightarrow C\text{flag}(t)) \land \\
\land (X\text{flag}(t + 1) \leftrightarrow X\text{flag}(t)) \land \\
\land \forall a. \text{memory}(a, t + 1) = \text{memory}(a, t) .
\]

Although it may appear to the reader that this particular representation is very simple, we want to remark some points we think are worth noticing, also with respect to the other parts of the Constructive Verification Environment:

- The representation is dependent on the MC68000 architecture, but in an elementary way, since the information we need to compute the logical representation of an instruction is clearly and unambiguously documented in the data book of the microprocessor (Motorola, 1989).

- The preprocessor takes care of eliminating the dependency on the system architecture; in our case, the way Unix uses to code an executable file, is hidden in the preprocessor, which, when generating the assembly code, uses the same functions as the operating system when loading the program. Thus, we are really going to verify what will be executed.

- The theory of the microprocessor is an Harrop theory in \textbf{IL}; with the addition of the following axioms

\[
\forall x. \Box V\text{flag}(x) \rightarrow V\text{flag}(x) \\
\forall x. \Box Z\text{flag}(x) \rightarrow Z\text{flag}(x) \\
\forall x. \Box N\text{flag}(x) \rightarrow N\text{flag}(x) \\
\forall x. \Box C\text{flag}(x) \rightarrow C\text{flag}(x) \\
\forall x. \Box X\text{flag}(x) \rightarrow X\text{flag}(x)
\]

it becomes a \(\Box\)-theory in \textbf{E}, see Chapter 6.

- The output of the translation procedure is an Harrop theory in \textbf{IL} and a \(\Box\)-theory in \textbf{E}, thus, we are guaranteed that our tool to analyze proofs will work in the expected way on correctness proofs, as we remarked in Chapter 6.
7.2 Compressing the Representation

The representation of object code we described in the previous section is satisfactory for our purposes, as we noticed in the remarks, but it suffers from being quite redundant.

We mean that most of the information we put into the formula which encodes an instruction, is, in many cases, useless. For example, let us consider the following fragment

\[
\begin{align*}
40: & \text{MOVE} \ #7, \ d_0 \\
42: & \text{ADD} \ #1, \ d_0 \\
44: & \text{MOVE} \ d_0, \ (a_1)
\end{align*}
\]

its encoding in a logical format can be reduced to

\[
\forall t. pc(t) = 40 \rightarrow pc(t+3) = 46 \land d_0(t+3) = 8 \land \text{memory}(a_1(t), t+3) = 8
\]

In this section we want to introduce the techniques we can use to compress the logical representation in order to make it more manageable.

The techniques we will speak about can be classified in three groups:

- enveloping
- analysis of the flow of control
- reductions according to a specification

The enveloping technique is based on the fact that compilers produce object code with a peculiar structure; in particular, a procedure is compiled in a way which can be represented as

\[
\begin{array}{c}
E \\
C \\
E
\end{array}
\]

the $C$ part is the code which implements the procedure, while the $E$ part, the \textit{envelope} takes care of getting the parameters and returning the result.

For example, let us consider the following function in the C language:
int f(int x) {
    return (x + 1); }

It is compiled by the \texttt{gcc} compiler into the following object code

\begin{tabular}{|c|}
\hline
0: LINK \#0, \texttt{a}_6 \\
2: MOVE 0(\texttt{a}_7), \texttt{d}_0 \\
5: ADD \#1, \texttt{d}_0 \\
7: MOVE \texttt{d}_0, -8(\texttt{a}_7) \\
10: UNLK \texttt{a}_6 \\
12: RTS \\
\hline
\end{tabular}

we marked the parts which constitute the body of the function and its envelope.

Normally, when we verify that a procedure is correct, we can limit ourselves to
the body, since the correctness proof for the envelope is standard.

In fact, one can prove once and for all that envelopes are correct, that is, they
correctly take the parameters from the stack, referred to by the \texttt{a}_7 register, and they
correctly return the result. An example of application of the enveloping technique
can be found in Appendix A.

The analysis of the flow of control of a program is probably the most important
technique for compressing the logical representation. It is based on the grouping of
sequential blocks of instructions; normally, when we want to prove that a piece of
code is correct, we consider sequential blocks of instructions as units.

The algorithm which performs this kind of compression is as follows; being \( P = [I_1, \ldots, I_n] \) a program whose instructions are \( I_1, \ldots, I_n \), we can derive a compressed
representation whose axioms are \( J_1, \ldots, J_m \), applying the algorithm

\[
\begin{align*}
& k := 1 \\
& i := 1 \\
& J_k := I_i \\
& \text{while } i < n \text{ do} \\
& \quad J_k := \tau(J_k, I_i) \\
& \quad \text{if } I_i \in \text{BranchingInstructions then} \\
& \quad \quad k := k + 1 \\
& \quad \quad J_k := I_{i+1} \\
& \quad \fi \\
& \quad i := i + 1 \\
& \text{od}
\end{align*}
\]

where the \( \tau \) function substitutes the values of registers as calculated in \( J_k \) with the
ones computed by \( I_i \).

For example, the fragment of code we have shown at the beginning of this section,
after the compression, is represented by the formula

\[
\forall t. pc(t) = 40 \rightarrow pc(t + 3) = pc(t) + 46 \land \\
\land d_0(t + 3) = 8 \land d_1(t + 3) = d_1(t) \land \ldots \land d_7(t + 3) = d_7(t) \land \\
\land a_0(t + 3) = a_0(t) \land \ldots \land a_7(t + 3) = a_7(t) \land \\
\land \neg V\text{flag}(t + 3) \land \\
\land \neg Z\text{flag}(t + 3) \land \\
\land \neg N\text{flag}(t + 3) \land \\
\land \neg C\text{flag}(t + 3) \land \\
\land \neg X\text{flag}(t + 3) \land \\
\land \text{memory}(a_1(t), t + 3) = 8 \land \\
\land \forall x. x \neq a_1(t) \rightarrow \text{memory}(x, t + 3) = \text{memory}(x, t) .
\]

The analysis of the flow of control of a program gives rise to another compression technique; the idea behind what we are going to describe, is that, in correctness proof, most of the time, it is important what is changed, and not what maintains the same value.

In practice, we want to compress the logical representation of an instruction (sequential block) by deleting the equalities stating that a register (memory cell, flag) which is never used by the program, retains the same value.

In the previous example this idea leads us to the representation

\[
\forall t. pc(t) = 40 \rightarrow pc(t + 3) = 46 \land d_0(t + 3) = 8 \land \text{memory}(a_1(t), t + 3) = 8
\]

for the whole sequential block.

The algorithm which performs this kind of compression is complex because of the amount of details, hence we show just its main structure.

Given an assembly program \( P \), we can draw a graph whose nodes are instructions and whose directed edges go to an instruction which may be executed just after the current one.

The compression of sequential blocks can be formulated as a transformation on this graph, which collapses two nodes into one if there is exactly one edge connecting them.

Similarly, the compression of equalities is modeled by a transformation on a labelled version of that graph.

Let us suppose that \( G \) is the graph associated to the program \( P \); we can label every node \( N \) with the set \( C_N \) of registers, flags and memory cells that are changed by the execution of the instruction the node represents; moreover, we label every node \( N \) of \( G \) with the set \( R_N \) of registers, flags and memory cells which are read by the instruction in \( N \), that is, the registers, flags and cells which are on the right-hand side of equalities whose left-hand side is in \( C_N \).

The transformation that performs the compression operates as follows: let \( N \) and \( M \) be two nodes connected by an arc from \( N \) to \( M \), we modify the labels of \( N \) as \( C_N' \) becomes \( C_N \cup R_M \) and \( R_N' \) is \( R_N \cup (R_M \setminus C_N) \).
7.2. **COMPRESSION THE REPRESENTATION**

The least fixed point of this transformation produces a graph $G^*$ such that, for every node $N$ in $G^*$, the label $C_N$ contains exactly the left-hand side of the equalities we have to retain in the compressed representation of the instruction associated with $N$.

The two compression techniques derived from the analysis of the flow of control may be combined, as in the example.

The last compression technique we discuss, uses a specification $S$ with the format $\exists t. \text{pc}(t) = x \land A$, where $A$ is a formula, depending on the time $t$, and $x$ is an address in the program. The idea is to retain just the parts of the program representation that contribute to assign values to the registers, flags and memory cells appearing in $S$.

The algorithm which does this kind of compression is a variant of the algorithm which performs the compression of equalities.

Let us suppose that $G$ is the labelled graph for the program, constructed as above, and let us suppose that the specification $S$ has the form $\exists t. \text{pc}(t) = x \land A$; let the node corresponding to the instruction at the address $x$ be $N$, we substitute the label $C_N$ with the set of all registers, flags and memory cells occurring in $S$, then we run the algorithm to perform the compression of equalities.

The result is a representation which is the minimal one which computes just the values mentioned in the specification.

We want to end this section with a series of remarks about the compression techniques:

- The compressed logical representation of a program is again an Harrop theory in $\mathbf{IL}$, and a $\Box$-theory in the $\mathbf{E}$ logical system. The consequence of this fact is that the whole set of remarks we did at the end of the previous section about the homogeneous nature of the Constructive Verification Environment still apply.

- The compression techniques are independent, hence we can combine them, according to our needs.

- The compression techniques really discard information from the logical representation. Thus, it is convenient to let the user to choose if they should be applied.

As we already said, compressing a program representation discards information. In particular, the enveloping technique discards information about the way different subroutines are linked, and, for this reason, the compression algorithm provides lemmas which act like specifications, in the sense of Chapter 5, for the envelope.

The techniques based on the analysis of the flow of control of a program discard information on what happens in the middle of a sequential block of instructions, and about registers, flags and memory cells which are not used in the computation the
program performs. Sometimes this kind of information is useful, for example when we want to prove a specification which says that something does not hold (usually, this happens for safety properties, opposed to liveness properties, see, e.g, (Lamport, 1994)).

The technique which compresses a logical representation for a program according to a specification, is even more fragile: it does not ensure that we are able to perform the correctness proof using the information we retain in the logical representation. Despite this fact, when it works, it is absolutely powerful since we have to manage a very compact representation.

7.3 Summary

In this section, we discussed the logical representation of programs we adopt for the Constructive Verification Environment.

In particular, we have shown that the chosen representation gives raise to a specification framework, which models the program. Thus, we have an homogeneous design, which shares the same concepts in every tool, starting from the program representation up to the most sophisticate proof analysis techniques.

The chosen representation can be compressed with various techniques, as we have seen in the last section, providing manageable theories.
Chapter 8

Theoretical Aspects

In this chapter we will show some theoretical aspects which are particularly relevant for the purposes of our thesis.

In the first part, we will introduce the notion of uniformly constructive system and, then, we will prove that all the logical systems we used in the development of the Constructive Verification Environment are uniformly constructive.

We will show that the notion of uniformly constructive system is the one which permits to make the constructive and computational readings of formulas to be coincident, thus proving that the specification framework approach and the Collection Method are perfectly compatible. In other words, we will show that the modeling tool, the synthesis tool and the analysis tool have a common root, which allows to smoothly combine their results.

In the last part of this chapter we will prove that the natural deduction presentation of the E logic is equivalent to its tableau presentation; it was a topic we left open in Chapter 3, which is important to give a measure of the power of our reasoning tool for the pure version of the Constructive Verification Environment.

8.1 Uniformly Constructive Formal Systems

In this section we want to introduce the formal device which enables us to interpret in both a constructive and a computational way specification formulas.

Since we are going to introduce a formal instrument from mathematical logic, we must be very precise; when we speak about a constructive reading of formulas we intend that a formula is evaluated in the logical system IL (E) in the following sense:

• an atomic formula is evaluated if it can be proven;
• (only for the E system) a boxed formula is evaluated if it can be proven.
• \( A \land B \) is evaluated if \( A \) is evaluated and \( B \) is evaluated in \( IL (E) \).
• \( A \lor B \) is evaluated either if \( A \) is evaluated or if \( B \) is evaluated in \( IL (E) \).
• $A \rightarrow B$ is evaluated if, when $A$ is evaluated, then $B$ is evaluated in $\textbf{IL}$ ($\textbf{E}$).

• $\neg A$ is evaluated in $\textbf{IL}$ if $\neg A$ is proven in $\textbf{IL}$; $\neg A$ is evaluated in $\textbf{E}$ if we can constructively evaluate $A$ as false in $\textbf{E}$, that is, we are able to construct a counterexample to $A$.

• $\forall x. A(x)$ is evaluated if, for every term $t$, $A(t)$ is evaluated in $\textbf{IL}$ ($\textbf{E}$).

• $\exists x. A(x)$ is evaluated if there is a term $t$ such that $A(t)$ is evaluated in $\textbf{IL}$ ($\textbf{E}$).

Also, when we speak about a computational reading of formulas, we think to them as specifications, and precisely

• an atomic formula represents an elementary computation.

• (only for the $\textbf{E}$ system) a boxed formula represents no computation but just a property to satisfy.

• $A \wedge B$ represents the computation requiring to compute $A$ and to compute $B$.

• $A \lor B$ represents the computational task requiring either to compute $A$ or to compute $B$.

• $A \rightarrow B$ represents the computational task that requires to transform any computation of $A$ into a computation of $B$; in other words, if we have a computation of $A$, then we have a computation of $B$.

• $\neg A$ represents no computation in $\textbf{IL}$ but the fact that $A$ is a property we require not to have; $\neg A$ represents the computational task in $\textbf{E}$ to find a counterexample to the computability of $A$.

• $\forall x. A(x)$ represents the computational task that, for every term $t$, calculates $A(t)$.

• $\exists x. A(x)$ represents the computation requiring to find a term $t$ such that $A(t)$ is computed.

There is an evident correspondence between the two notions.

At this point, clarified what we intend for constructive reading of formulas, the reader may ask what we intend for a constructive system.

We left this notion unexplained, and the reader may assume that it has the usual meaning that is used in literature.

In fact, this is true; our notion of constructive system is not essentially different from what one may find around. But, since this thesis is about constructivity, it is time to define things precisely.

There is not a well consolidated formalization of what a constructive system is; there are a number of approaches to mathematics which are accepted as constructive, and they are, more or less, similar, but with small, subtle, and important, differences.
Since this is a thesis in Computer Science, and not in Philosophy, we will not try to analyze the details of this fascinating topic, but the reader may find interesting to read (Beeson, 1985), to get a general overview of these aspects.

To be precise, there is no definition (well, not a commonly accepted one) of constructive logic: in fact, although no one has doubts on the fact that Intuitionistic Logic is constructive, and Classical Logic is not, it is not a trivial task to prove that a non standard logic is, or is not, constructive. The first problem is to disambiguate the word constructive, indeed. The key point, as very well remarked in (Troelstra, 1977a; Troelstra and van Dalen, 1988), is that the notion of constructive system is not intended to be captured in a definition, but in the process of building theorems.

It is a commonly accepted fact that a logic is naively constructive iff

- if $\vdash A \lor B$ then $\vdash A$ or $\vdash B$;
- if $\vdash \exists x. A(x)$ then there is a term $t$ such that $\vdash A(t)$.

It has been remarked (Troelstra, 1973a) that this notion has not a direct relation with constructivity; in fact, there are system which are “constructive” and not naively constructive, e.g., non standard intuitionistic analysis (Bishop, 1967), and, conversely, systems which are naively constructive and not “constructive”, e.g., the pathological system developed in (Ferrari, 1997b).

Hence we have the strong need to fix a definition for what we intend as a constructive system.

In the previous chapters, we fixed some important points:

- we require to have a $\Box$ operator which represents classical truth.
- we require to have a constructive interpretation for negation.
- we need to extract information from proofs.

Our solution to these problems has been the choice of the $E$ logic, which, by construction, satisfies the first two requirements.

In Chapter 6 we have seen that there is a way, namely the Collection Method, which permits to satisfy the third requirement as well.

Our aim when we adopted the Collection Method as our analysis tool can be synthesized in the two following points:

- we thought the information is contained in a proof; an information extraction mechanism should explicit this information. Hence, it has not to be oriented on the proven properties, but, rather, it should be orientable to extract information of different nature, depending on what we may be interested in.
- we are designing a software system that have to make possible to represent different theories and to effectively use them, hence, our analysis tool must be largely independent from the specific details of a theory and from the particular shape of inference rules.
We gave evidence that both points are feasible in the case of the Collection Method in Chapter 6.

But we left open one question, the one we used to start this section, how to prove that the constructive reading of a specification formula is compatible with its computational reading.

We have already shown that the computational reading and the constructive reading are, essentially, identical.

But, how to assure that the E system, or IL, or any other theory we used in this thesis really admits such a computational/constructive reading?

As we remarked before, we need to fix a particular view of what we intend for constructivism.

The right notion for our purposes is the one of uniformly constructive system (Avellone et al., 1996; Bertoni et al., 1978; Bertoni et al., 1988a; Bertoni et al., 1988b; Ferrari et al., 1999; Miglioli et al.; Miglioli and Ornaghi, 1978; Miglioli and Ornaghi, 1981a; Miglioli and Ornaghi, 1981b; Miglioli and Ornaghi, 1982). It is not in the scope of this thesis to discuss this notion from a mathematical point of view, since we use it as an instrument to show that our methodology (the specification framework approach, the Collection Method, ... ) is founded. Hence we will limit ourselves to introduce the notion.

We start by criticizing the notion of naively constructive system:

- no link between the proof of \( A \lor B \) and a proof of \( A \) or a proof of \( B \) is required in the definition;

- no way on how to choose \( t \), nor on how to prove \( A(t) \) comes from the proof of \( \exists x. A(x) \) in the definition.

The consequence is that, given a formal system \( \mathcal{L} \), we can prove that \( \mathcal{L} \) is naively constructive iff there is a Gödelization of the proofs in \( \mathcal{L} \) such that one can exhibit a function \( f \) which, for every proof of a disjunctive (existential) theorem, \( f \) maps its Gödel number into the Gödel number of a proof for its witness. This theorem is really easy to prove for most systems, but its consequences are interesting. Let us just think what happens if \( f \) is not recursive: of course, it will be very hard to say that the \( \mathcal{L} \) system is “constructive”, since we have no effective way to establish this fact!

The solution which goes in the direction we need, has to create a link between the proof of a disjunctive (existential) statement with the proof of its witness.

We say that a system is uniformly constructive if

- if \( \vdash A \lor B \) with the proof \( \Pi \), then the information content of \( \Pi \) contains either a proof which conclusion is \( A \), or a proof which conclusion is \( B \);

- if \( \vdash \exists x. A(x) \) with the proof \( \Pi \), then the information content of \( \Pi \) contains a subproof which conclusion is \( A(t) \), for an appropriate term \( t \).
For information content, we intend what can be extracted from a proof Π, that is, Inf*(<Π>).

If we are able to prove that the logical systems employed in the Constructive Verification Environment are, indeed, uniformly constructive, then we reached our goal, to show that every adopted tool share the same reading (or semantics, if you prefer), and so they work smoothly together.

Finally, we note two important things with respect to the topic of our thesis:

- the proofs that the systems presented so far are uniformly constructive, share the same pattern, confirming the generality of the Collection Method as an analysis tool. We will show this pattern in details when we will prove that IL is uniformly constructive.

- the property of a system to be uniformly constructive gives us a measure of the quality of information we are able to collect from its proofs; the idea of evaluation we gave before is this measure. We will prove, and this is the case for every uniformly constructive formal system, that for every set of proofs I, Inf*(I) is closed under the notion of evaluation. In other words, Inf*(I) explains why each of its formulas are true in a constructive sense.

### 8.2 IL is Uniformly Constructive

In this section, we will prove that first order intuitionistic logic is uniformly constructive. The proof technique we are going to show has been introduced in (Miglioli and Ornaghi, 1978).

The main goal of this section, apart from the proof of the result itself, is to make clear the logical passages which bring us to the result.

For this very reason, we will not insist on the value of the result, although it provides a good theoretical measure of the quality of our information extraction technique, but we prefer to focus on the way we follow to produce that result.

The starting point is to define the notion of pseudo-truth set. The intuitive meaning we want to induce into this definition, is that of a syntactically consistent set. We want that every formula in this set is a theorem, and the whole set, in a sense, provides an explanation for itself, following the semantics of the logic we are working on.

**Definition 8.2.1** A set $\mathcal{F}$ of formulas is a pseudo-truth set if and only if

- $A \in \mathcal{F}$ implies $\vdash A$.

- $\neg A \in \mathcal{F}$ implies that $A \notin \mathcal{F}$.

- $A \lor B \in \mathcal{F}$ implies $A \in \mathcal{F}$ or $B \in \mathcal{F}$.

- $A \land B \in \mathcal{F}$ implies $A \in \mathcal{F}$ and $B \in \mathcal{F}$.  

• \( A \rightarrow B \in \mathcal{F} \) and \( A \in \mathcal{F} \) implies \( B \in \mathcal{F} \).

• \( \exists x. A(x) \in \mathcal{F} \) implies that there is a term \( t \), such that \( A(t) \in \mathcal{F} \).

The notion of pseudo-truth set is global, that is, it is given looking at the entire set of formulas; the local notion is that of evaluation. A formula is evaluated in a set of formulas if it is explained by that set.

**Definition 8.2.2** Let \( A \) be a formula, it is evaluated in \( \mathcal{F} \), a set of formulas, if and only if

• \( A \in \mathcal{F} \).

• \( A \) is an atomic or a negated formula.

• \( A \equiv B \land C \) and \( B \) and \( C \) are evaluated in \( \mathcal{F} \).

• \( A \equiv B \lor C \) and \( B \) is evaluated in \( \mathcal{F} \), or \( C \) is evaluated in \( \mathcal{F} \).

• \( A \equiv B \rightarrow C \) and, if \( B \) is evaluated in \( \mathcal{F} \), then also \( C \) is evaluated in \( \mathcal{F} \).

• \( A \equiv \exists x. B(x) \) and there is a term \( t \) such that \( B(t) \) is evaluated in \( \mathcal{F} \).

• \( A \equiv \forall x. B(x) \) and, for all terms \( t \) such that \( B(t) \in \mathcal{F} \), the formula \( B(t) \) is evaluated in \( \mathcal{F} \).

The main goal of our proving effort is to show that the global notion and the local notion coincide in the case of \( \text{Inf}^*(\mathcal{I}) \), for any set \( \mathcal{I} \) of proofs.

In order to gain this result, we need two closure lemmas; the first one proves that our construction is closed for membership; the second one proves that the construction is closed under evaluation.

**Lemma 8.2.1** Let \( \mathcal{I} \) be a set of proofs; if \( \frac{}{A} \) is a subproof of a proof in \( \text{Exp}^*(\mathcal{I}) \), and \( \Gamma \subseteq \text{Inf}^*(\mathcal{I}) \) then \( A \in \text{Inf}^*(\mathcal{I}) \).

**Proof:** Let \( \Gamma = \{ B_1, \ldots, B_n \} \).

From \( \frac{}{A} \) \( \text{Exp}^*(\mathcal{I}) \) it follows that there is an index \( j \) such that \( \frac{}{A} \) \( \text{Exp}^*(\mathcal{I}) \)

and, for all \( i \geq j \), \( \frac{}{A} \) \( \text{Exp}^*(\mathcal{I}) \).

From \( \{ B_1, \ldots, B_n \} \subseteq \text{Coll}^*(\mathcal{I}) \) it follows that there are indexes \( i_1, \ldots, i_n \) such that \( B_1 \in \text{Inf}(\text{Exp}^{i_1}(\mathcal{I})), \ldots, B_n \in \text{Inf}(\text{Exp}^{i_n}(\mathcal{I})) \).

Let \( k \) be the maximum in \( j, i_1, \ldots, i_n \), then

\( \frac{}{A} \) \( \text{Exp}^k(\mathcal{I}) \) and \( \Gamma \subseteq \text{Inf}(\text{Exp}^k(\mathcal{I})) \),

so, by definition, \( A \in \text{Inf}(\text{Exp}^k(\mathcal{I})) \), implying \( A \in \text{Inf}^*(\mathcal{I}) \). \( \square \)
Lemma 8.2.2 Let $\mathcal{I}$ be a set of proofs, and let $\Gamma \vdash_{A} \neg \text{Exp}^{*}(\mathcal{I})$, and let $\Gamma$ be evaluated in $\text{Inf}^{*}(\mathcal{I})$, then $A$ is evaluated in $\text{Inf}^{*}(\mathcal{I})$.

Proof: By Lemma 8.2.1, we know that $A \in \text{Inf}^{*}(\mathcal{I})$.
We prove that $A$ satisfies the other condition to be evaluated by induction on the structure of proofs.

- Assumption:

\[
\Gamma \vdash_{A} A
\]

$A \in \Gamma$, thus, by hypothesis, it is evaluated in $\text{Inf}^{*}(\mathcal{I})$.

- $\neg$ Introduction:

\[
\Gamma, [A] \quad \Gamma, [A] \\
\vdash_{A} \quad \vdash_{A} \\
\neg C \quad C \\
\vdash_{A} \neg B
\]

Since $\neg B$ is negated, by definition, it is evaluated in $\text{Inf}^{*}(\mathcal{I})$.

- $\neg$ Elimination:

\[
\Gamma \quad \Gamma \\
\vdash_{A} \quad \vdash_{A} \\
\neg B \quad B \\
\vdash_{A} A
\]

$A$ is an atomic formula, so, by definition, it is evaluated.

- $\land$ Introduction:

\[
\Gamma \quad \Gamma \\
\vdash_{A} C \quad D \\
\vdash_{A} C \land D
\]

by induction hypothesis, $C$ and $D$ are evaluated in $\text{Inf}^{*}(\mathcal{I})$, so, $B \land C$ gets evaluated.

- $\land$ Elimination:

\[
\Gamma \quad \Gamma \\
\vdash_{A} C \land D \\
\vdash_{A} C
\]
or
\[ \Gamma ; A \Rightarrow C \land D \]

by induction hypothesis, \( C \land D \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), so, \( C \) and \( D \) are evaluated, as well.

- \( \lor \) Introduction:

\[
\Gamma \vdash A \quad \Gamma \vdash C
\]

or
\[
\Gamma \vdash A \quad \Gamma \vdash D
\]

by induction hypothesis, \( C \lor D \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), so, \( C \lor D \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \).

- \( \lor \) Elimination:

\[
\begin{array}{c}
\Gamma \vdash C \\
\Gamma \vdash D
\end{array}
\]

by induction hypothesis, \( C \lor D \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), so \( C \lor D \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), too.

If \( C \) is evaluated, then from the induction hypothesis on \( \Gamma ; C \), we can deduce that \( A \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \); if \( D \) is evaluated, then from the induction hypothesis on \( \Gamma ; D \), we can deduce that \( A \) is evaluated. Hence, \( A \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \).

- \( \rightarrow \) Introduction:

\[
\Gamma \vdash A \quad \Gamma \vdash C
\]

by induction hypothesis, \( C \rightarrow D \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), so \( C \rightarrow D \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \).
8.2. IL IS UNIFORMLY CONSTRUCTIVE

by definition, if $C$ is not evaluated in $\text{Inf}^*(\mathcal{I})$, then $C \rightarrow \mathcal{D}$ is evaluated; if $C$ is evaluated, then, by induction hypothesis, $\mathcal{D}$ is evaluated in $\text{Inf}^*(\mathcal{I})$. So, in general, $C \rightarrow \mathcal{D}$ is evaluated in $\text{Inf}^*(\mathcal{I})$.

- **→ Elimination:**

\[
\begin{array}{c}
\Gamma \vdash C \\
\Gamma \vdash C \rightarrow A \\
\hline
\Gamma \vdash A
\end{array}
\]

by induction hypothesis, $C$ and $C \rightarrow A$ are evaluated in $\text{Inf}^*(\mathcal{I})$, hence, by definition, $A$ is evaluated in $\text{Inf}^*(\mathcal{I})$.

- **∀ Introduction:**

\[
\begin{array}{c}
\Gamma \vdash C(p) \\
\hline
\Gamma \vdash \forall x. C(x)
\end{array}
\]

If $C(t) \notin \text{Inf}^*(\mathcal{I})$, for any term $t$, then $\forall x. C(x)$ is evaluated in $\text{Inf}^*(\mathcal{I})$.

Let us suppose the there is a term $t$ such that $C(t) \in \text{Inf}^*(\mathcal{I})$; $\forall x. C(x)$ is evaluated in $\text{Inf}^*(\mathcal{I})$ if and only if, $C(t)$ is evaluated in $\text{Inf}^*(\mathcal{I})$.

Now, since $C(t) \in \text{Inf}^*(\mathcal{I})$, there is an index $k$ such that $C(t) \in \text{Inf}(\text{Exp}^k(\mathcal{I}))$, $\Gamma \subseteq \text{Inf}(\text{Exp}^k(\mathcal{I}))$ and $\vdash \text{Exp}^k(\mathcal{I})$, so, by definition, $\vdash (p=t) \iff \text{Exp}^{k+1}(\mathcal{I})$, $\vdash C(p)$ that is, $\vdash (p=t) \iff \text{Exp}^*(\mathcal{I})$, and, by induction hypothesis, $C(t)$ is evaluated in $\text{Inf}^*(\mathcal{I})$; from this fact, it follows that $\forall x. C(x)$ is evaluated in $\text{Inf}^*(\mathcal{I})$.

- **∀ Elimination:**

\[
\begin{array}{c}
\Gamma \vdash \forall x. C(x) \\
\hline
\Gamma \vdash C(t)
\end{array}
\]

by induction hypothesis, $\forall x. C(x)$ is evaluated in $\text{Inf}^*(\mathcal{I})$, so, since $C(t) \in \text{Inf}^*(\mathcal{I})$, by definition, it is evaluated in $\text{Inf}^*(\mathcal{I})$.

- **∃ Introduction:**

\[
\begin{array}{c}
\Gamma \vdash C(t) \\
\hline
\Gamma \vdash \exists x. C(x)
\end{array}
\]
by induction hypothesis, $C(t)$ is evaluated in $\text{Inf}^*(\mathcal{I})$, so, since $\exists x. C(x) \in \text{Inf}^*(\mathcal{I})$, by definition, it is evaluated.

- $\exists$ Elimination:

\[
\begin{array}{c}
\Gamma \\
\vdash \exists x. C(x) \\
A
\end{array} \quad \begin{array}{c}
\Gamma \\
\vdash C(p) \\
\Gamma, C(p)
\end{array} \quad \begin{array}{c}
\Gamma \\
\vdash \ (p) \\
\Gamma
\end{array} \quad \begin{array}{c}
\vdash A \\
\vdash A
\end{array} \quad \begin{array}{c}
\vdash A \\
\vdash A
\end{array}
\]

by induction hypothesis, $\exists x. C(x)$ is evaluated in $\text{Inf}^*(\mathcal{I})$, and, by definition, there is a closed term $t$ such that $C(t)$ is evaluated in the same set, hence $C(t) \in \text{Inf}^*(\mathcal{I})$.

Then, there must be an index $k$ such that $\Gamma \cup \{C(t)\} \in \text{Inf}(\text{Exp}^k(\mathcal{I}))$ and $\Gamma \vdash C(p)$, so, by definition, $\Gamma, C(p) \vdash \text{Exp}^k(\mathcal{I})$, that is, $\Gamma, (p=t) \vdash \text{Exp}^{k+1}(\mathcal{I})$, and, by induction hypothesis, $A$ is evaluated in $\text{Inf}^*(\mathcal{I})$. \hfill \Box

The conclusion we get from the previous lemmas is that every formula in $\text{Inf}^*(\mathcal{I})$ gets evaluated. When this happens, remembering our intuitive reading of evaluation, it means that every collected formula is explained by the information content of the set of proofs $\mathcal{I}$. Hence, it should be clear that we need nothing else to verify that the logic is constructive, and, as we remarked before, this is the intended meaning of uniformly constructive formal system.

**Theorem 8.2.1** Let $\mathcal{I}$ be a set of proofs, and let $A \in \text{Inf}^*(\mathcal{I})$, then $A$ is evaluated in $\text{Inf}^*(\mathcal{I})$.

**Proof:** From $A \in \text{Inf}^*(\mathcal{I})$, we deduce that there is an index $j$ such that $A \in \text{Inf}(\text{Exp}^j(\mathcal{I}))$.

Let’s define a function:

- $\deg_{\text{Exp}^j(\mathcal{I})}(B) = 0$

if $B \vdash \text{Exp}^j(\mathcal{I})$ without undischarged assumptions;

- $\deg_{\text{Exp}^j(\mathcal{I})}(B) = \max\{\deg_{\text{Exp}^j(\mathcal{I})}(C_1), \ldots, \deg_{\text{Exp}^j(\mathcal{I})}(C_n)\} + 1$

if $B \vdash \text{Exp}^j(\mathcal{I})$ and $\{C_1, \ldots, C_n\} \subseteq \text{Inf}(\text{Exp}^j(\mathcal{I}))$. 

By induction on degree_ex^I(B), we prove that, for all B ∈ \text{Inf}(\text{Exp}^j(I)), B is evaluated in Inf^*(I):

- if degree_ex^I(B) = 0, then \( \vdash_B \exp^j(I) \), so \( \vdash_B \exp^*(I) \), and, by Lemma 8.2.2, B is evaluated in Inf^*(I).

- if degree_ex^I(B) > 0, then \( \vdash_B \exp^j(I) \), so \( \vdash_B \exp^*(I) \),

but, by induction hypothesis, \( C_1, \ldots, C_n \) are evaluated in Inf^*(I), and, by Lemma 8.2.2, B is evaluated in Inf^*(I), too.

So, because \( A ∈ \text{Inf}(\exp^j(I)) \), then it is evaluated in Inf^*(I).

The previous theorem is the key to link the global view, i.e., pseudo-truth sets, with the local view, i.e., the notion of evaluation.

**Theorem 8.2.2** Let I be a set of proofs, then Inf^*(I) is a pseudo-truth set.

**Proof:** From the construction of Inf^*(I),

- \( A ∈ \text{Inf}^*(I) \) implies \( \vdash A \), since it is the conclusion of a proof without undischarged assumption.

  In fact, by an immediate induction on the structure of the Coll operator, as defined in Chapter 6, we get that there is proof with no undischarged assumptions which is composed by combining proofs in Coll^*(I).

- \( \lnot B ∈ \text{Inf}^*(I) \) implies that \( \vdash \lnot B \), hence \( \not\vdash B \), that implies \( B \not\in \text{Inf}^*(I) \), by construction.

- \( B \lor C ∈ \text{Inf}^*(I) \) implies, by Theorem 8.2.1 that \( B \lor C \) is evaluated in Inf^*(I), and, by definition, \( B \lor C \) is evaluated in Inf^*(I), so \( B ∈ \text{Inf}^*(I) \lor C ∈ \text{Inf}^*(I) \).

- \( B \land C ∈ \text{Inf}^*(I) \) implies, from Theorem 8.2.1 that \( B \land C \) is evaluated in Inf^*(I), and, by definition, \( B \land C \) is evaluated in Inf^*(I), so \( B ∈ \text{Inf}^*(I) \land C ∈ \text{Inf}^*(I) \).

- \( B \rightarrow C ∈ \text{Inf}^*(I) \) and \( B ∈ \text{Inf}^*(I) \) implies that \( B \rightarrow C \) and \( B \) are evaluated in Inf^*(I), hence, by definition, \( C \) is evaluated in Inf^*(I), so \( C ∈ \text{Inf}^*(I) \).

- \( \exists x. B(x) ∈ \text{Inf}^*(I) \) implies that \( \exists x. B(x) \) is evaluated in Inf^*(I), and, by definition, there is a term t such that \( B(t) \) is evaluated in Inf^*(I), so \( B(t) ∈ \text{Inf}^*(I) \).

So, by definition, \( \text{Inf}^*(I) \) is a pseudo-truth set.
If we consider two special cases of the previous theorem, namely the singleton set \{A \lor B\} and the singleton set \{\exists x. C(x)\}, we can immediately deduce that, for every disjunction, one of the disjuncts is present in the information content of any proof for \(A \lor B\), and, similarly, for any existential formula \(\exists x. C(x)\), a witness for its truth can be found in the information content of its proof. Since we can exhibit a proof for both kind of witnesses, just looking at the result of Coll* applied to the corresponding singleton set, it follows that:

**Corollary 8.2.1** The logic IL is uniformly constructive.

### 8.3 The E Logic is Uniformly Constructive

The proof which states that E is uniformly constructive follows the same pattern we have seen in Section 8.2 for IL. Hence we will just illustrate the additional notions, the differences and the reasons for them in the following.

The notion of evaluation has to be enlarged because we have a new connective, \(\Box\), and negation has a local semantics.

**Definition 8.3.1** A formula \(A\) is evaluated in a set of formulas \(\mathcal{F}\), iff \(A \in \mathcal{F}\) and

- \(A\) is a literal, i.e., atomic or negated atomic;
- \(A \equiv \Box B\);
- \(A \equiv \neg \Box B\);
- \(A \equiv B \land C\) and \(B\) and \(C\) are both evaluated in \(\mathcal{F}\);
- \(A \equiv \neg (B \land C)\) and \(\neg B\) is evaluated in \(\mathcal{F}\) or \(\neg C\) is evaluated in \(\mathcal{F}\);
- \(A \equiv B \lor C\) and \(B\) is evaluated in \(\mathcal{F}\) or \(C\) is evaluated in \(\mathcal{F}\);
- \(A \equiv \neg (B \lor C)\) and \(\neg B\) and \(\neg C\) are both evaluated in \(\mathcal{F}\);
- \(A \equiv B \rightarrow C\) and, if \(B\) is evaluated in \(\mathcal{F}\), then \(C\) is evaluated in \(\mathcal{F}\);
- \(A \equiv \neg (B \rightarrow C)\) and \(B\) and \(\neg C\) are both evaluated in \(\mathcal{F}\);
- \(A \equiv \neg \neg B\) and \(B\) is evaluated in \(\mathcal{F}\);
- \(A \equiv \forall x. B(x)\) and, for every term \(t\) such that \(B(t) \in \mathcal{F}\), \(B(t)\) is evaluated in \(\mathcal{F}\);
- \(A \equiv \neg \forall x. B(x)\) and there is a term \(t\) such that \(\neg B(t)\) is evaluated in \(\mathcal{F}\);
- \(A \equiv \exists x. B(x)\) and there is a term \(t\) such that \(B(t)\) is evaluated in \(\mathcal{F}\);
• $A \equiv \neg \exists x. B(x)$ and, for every term $t$ such that $\neg B(t) \in \mathcal{F}$, $\neg B(t)$ is evaluated in $\mathcal{F}$.

From this definition it is easy to prove

**Lemma 8.3.1** Let $\mathcal{I}$ be a set of proofs, if $\begin{array}{c} \Gamma \\ \vdots \\ A \end{array} \vdash A \in \text{Exp}^t(\mathcal{I}) \text{ and } \Gamma \subseteq \text{Inf}^t(\mathcal{I})$, then $A \in \text{Inf}^t(\mathcal{I})$.

**Proof:** See Lemma 8.2.1. \hfill \Box

**Lemma 8.3.2** Let $\mathcal{I}$ be a set of proofs, let $\begin{array}{c} \Gamma \\ \vdots \\ A \end{array} \vdash A \in \text{Exp}^t(\mathcal{I})$ and let, for every $\gamma \in \Gamma$, $\gamma$ be evaluated in $\text{Inf}^t(\mathcal{I})$, then $A$ is evaluated in $\text{Inf}^t(\mathcal{I})$.

**Proof:** See Lemma 8.2.2. For the remaining cases in the induction,

• $\neg \wedge$ Introduction:

$$
\begin{array}{c}
\Gamma \\
\vdots
\end{array} \equiv \\
\begin{array}{c}
\Gamma \\
\vdots
\end{array} \vdash \\
A \\
\begin{array}{c}
\vdash \neg B \\
\vdash (B \wedge C)
\end{array}
$$

or

$$
\begin{array}{c}
\Gamma \\
\vdots
\end{array} \equiv \\
\begin{array}{c}
\Gamma \\
\vdots
\end{array} \vdash \\
A \\
\begin{array}{c}
\vdash \neg B \\
\vdash (C \wedge B)
\end{array}
$$

By induction hypothesis, $\neg B$ is evaluated, then, by definition $\neg (B \wedge C)$ and $\neg (C \wedge B)$ are evaluated in $\text{Inf}^t(\mathcal{I})$.

• $\neg \wedge$ Elimination:

$$
\begin{array}{c}
\Gamma \\
\vdots
\end{array} \equiv \\
\begin{array}{c}
\Gamma \\
\vdots
\end{array} \vdash \\
\begin{array}{c}
\Gamma, \neg B \\
\Gamma, \neg C
\end{array} \\
\begin{array}{c}
\vdash (B \wedge C) \\
A
\end{array} \\
\begin{array}{c}
\vdash A \\
\vdash A
\end{array} \\
A
$$

By induction hypothesis, $\neg (B \wedge C)$ is evaluated, then $\neg B$ is evaluated or $\neg C$ is evaluated. In the former case, applying the induction hypothesis to $\begin{array}{c}
\Gamma, \neg B \\
\vdots
\end{array} \vdash A$, we get that $A$ is evaluated; in the latter case, applying the induction hypothesis to $\begin{array}{c}
\Gamma, \neg C \\
\vdots
\end{array} \vdash A$, we get that $A$ is evaluated.
• \( \neg \lor \) Introduction:

\[
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\equiv
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\begin{array}{c}
\vdots
\hline
\vdots
\end{array}
\begin{array}{c}

\neg B \\
\hline
\neg C \\
\hline
\neg (B \lor C)
\end{array}
\]

By induction hypothesis, \( \neg B \) and \( \neg C \) are both evaluated, thus, by definition, \( \neg (B \lor C) \) is evaluated in \( \text{Inf}^* (I) \).

• \( \neg \lor \) Elimination:

\[
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\equiv
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\begin{array}{c}

\neg (B \lor C) \\
\hline
\neg B
\end{array}
\]

or

\[
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\equiv
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\begin{array}{c}

\neg (B \lor C) \\
\hline
\neg C
\end{array}
\]

By induction hypothesis, \( \neg (B \lor C) \) is evaluated, then, by definition, both \( \neg B \) and \( \neg C \) are evaluated in \( \text{Inf}^* (I) \).

• \( \rightarrow \rightarrow \) Introduction:

\[
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\equiv
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\begin{array}{c}

B \\
\hline
\neg C \\
\hline
\neg (B \rightarrow C)
\end{array}
\]

By induction hypothesis, both \( B \) and \( \neg C \) are evaluated in \( \text{Inf}^* (I) \), hence \( \neg (B \rightarrow C) \) is evaluated, too.

• \( \rightarrow \rightarrow \) Elimination:

\[
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\equiv
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\begin{array}{c}

\neg (C \rightarrow B) \\
\hline
C
\end{array}
\]

or

\[
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\equiv
\begin{array}{c}
\Gamma \\
\hline
\vdots \\
\vdots \\
\vdots
\end{array}
\begin{array}{c}

\neg (C \rightarrow B) \\
\hline
\neg B
\end{array}
\]

By induction hypothesis, \( \neg (C \rightarrow B) \) is evaluated, then it follows that both \( C \) and \( \neg B \) are evaluated in \( \text{Inf}^* (I) \).
8.3. THE E LOGIC IS UNIFORMLY CONSTRUCTIVE 151

- ¬¬ Introduction:

\[
\begin{array}{c}
\Gamma \\
\vdots \\
A \quad \Gamma \\
\overline{\overline{B}} \\
\overline{-\neg B}
\end{array}
\]

By induction hypothesis, \( B \) is evaluated, then, by definition, \( \neg\neg B \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \).

- ¬¬ Elimination:

\[
\begin{array}{c}
\Gamma \\
\vdots \\
A \quad \Gamma \\
\overline{-A} \\
A
\end{array}
\]

By induction hypothesis, \( \neg\neg A \) is evaluated, then, by definition \( A \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), as well.

- ¬∀ Introduction:

\[
\begin{array}{c}
\Gamma \\
\vdots \\
A \quad \Gamma, \neg B(t) \quad \neg B(t) \\
\overline{\forall x. B(x)} \\
\neg\forall x. B(x)
\end{array}
\]

By induction hypothesis, \( \neg B(t) \) is evaluated, hence, by definition \( \neg\forall x. B(x) \) gets evaluated in \( \text{Inf}^*(\mathcal{I}) \).

- ¬∀ Elimination:

\[
\begin{array}{c}
\Gamma \\
\vdots \\
A \quad \Gamma, \neg B(p) \\
\overline{\forall x. B(x)} \\
\overline{A}
\end{array}
\]

By induction hypothesis \( \neg\forall x. B(x) \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), so there is an index \( k \) and a term \( t \) such that \( A \vdash \top \) \( \forall \) \( \exp^k(\mathcal{I}) \) and \( \Gamma \cup \{B(t)\} \subseteq \text{Inf}(\exp^k(I)) \).

Then \( \left( \begin{array}{c} \Gamma, \neg B(p) \\
\vdots \\
A \end{array} \right)(p := t) \in \exp^k(\mathcal{I}) \), that is \( \left( \begin{array}{c} \Gamma, \neg B(p) \\
\vdots \\
A \end{array} \right)(p := t) \in \exp^k(\mathcal{I}) \).

Applying the induction hypothesis, we get that \( A \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \).
• $\neg \exists$ Introduction:

$$
\frac{
\Gamma \\
A
}{
\frac{
\Gamma \\
B(p)
}{\neg \exists \, x. \, B(x)}
}
$$

For every term $t$ such that $\neg B(t) \in \text{Inf}^*(\mathcal{I})$, there is an index $k$ such that $\Gamma : \neg B(t) \in \text{Exp}^k(\mathcal{I})$ and $\Gamma \cup \{ \neg B(t) \} \subseteq \text{Inf}(\text{Exp}^k(\mathcal{I}))$, hence $(\Gamma : \neg B(p)) (p := t) \in \text{Exp}^{k+1}(\mathcal{I})$, that is, $(\Gamma : \neg B(p)) (p := t) \in \text{Exp}^*(\mathcal{I})$.

By induction hypothesis it follows that $\neg B(t)$ is evaluated. So, by definition, $\neg \exists \, x. \, B(x)$ is evaluated in $\text{Inf}^*(\mathcal{I})$.

• $\neg \exists$ Elimination:

$$
\frac{
\Gamma \\
A
}{
\frac{
\Gamma \\
\neg B(t)
}{\neg \exists \, x. \, B(x)}
}
$$

By induction hypothesis $\neg \exists \, x. \, B(x)$ is evaluated, thus $\neg B(t)$ is evaluated in $\text{Inf}^*(\mathcal{I})$, by definition.

• $\neg$ Contradiction:

$$
\frac{
\Gamma \\
A
}{
\frac{
\Gamma \\
B \quad \neg B
}{\neg P}
}
$$

Being $P$ atomic, $\neg P$ is evaluated by definition.

• $\Box$ Introduction:

$$
\frac{
\Gamma, \neg B \\
\Gamma, \neg B \\
C \\
\neg C
}{\Box B}
$$

Since the conclusion is a boxed formula, it is evaluated in $\text{Inf}^*(\mathcal{I})$.

• $\neg \Box$ Elimination:

$$
\frac{
\Gamma, B \\
\Gamma, B \\
C \\
\neg C
}{\neg \Box B}
$$

Since the conclusion is a negated boxed formula, it is evaluated in $\text{Inf}^*(\mathcal{I})$. $\Box$
Theorem 8.3.1 Let \( \mathcal{I} \) be a set of proofs, and let \( A \in \text{Inf}^\flat(\mathcal{I}) \), then \( A \) is evaluated in \( \text{Inf}^\flat(\mathcal{I}) \).

Proof: See Theorem 8.2.1. \( \square \)

At this point, we can prove, exactly in the same way as for \( \text{IL} \), Theorem 8.2.2, and so \( \text{E} \) is uniformly constructive. But we prefer to modify the definition of pseudo-truth sets to give a proof which provides a stronger theorem, taking into account the validity of De Morgan’s laws.

Definition 8.3.2 A set \( \mathcal{F} \) of formulas is a with negation iff

- \( A \in \mathcal{F} \) implies \( \vdash A \);
- \( A \lor B \in \mathcal{F} \) implies \( A \in \mathcal{F} \) or \( B \in \mathcal{F} \);
- \( \neg(A \lor B) \in \mathcal{F} \) implies \( \neg A \in \mathcal{F} \) and \( \neg B \in \mathcal{F} \);
- \( A \land B \in \mathcal{F} \) implies \( A \in \mathcal{F} \) and \( B \in \mathcal{F} \);
- \( \neg(A \land B) \in \mathcal{F} \) implies \( \neg A \in \mathcal{F} \) or \( \neg B \in \mathcal{F} \);
- \( A \rightarrow B \in \mathcal{F} \) implies if \( A \in \mathcal{F} \), then \( B \in \mathcal{F} \);
- \( \neg(A \rightarrow B) \in \mathcal{F} \) implies \( A \in \mathcal{F} \) and \( \neg B \in \mathcal{F} \);
- \( \exists x. A(x) \in \mathcal{F} \) implies that there is a term \( t \) such that \( A(t) \in \mathcal{F} \);
- \( \forall x. A(x) \in \mathcal{F} \) implies that there is a term \( t \) such that \( \neg A(t) \in \mathcal{F} \).

Theorem 8.3.2 Let \( \mathcal{I} \) be a set of proofs, then \( \text{Inf}^\flat(\mathcal{I}) \) is a pseudo-truth set with negation.

Proof: See Theorem 8.2.2. The extra cases are immediate consequences of the notion of evaluation. \( \square \)

Corollary 8.3.1 The logical system \( \text{E} \) is uniformly constructive.

8.4 Analysis of Constructive Principles

In this part, we will prove that every logical system used in the Constructive Verification Environment is, indeed, a uniformly constructive formal system.

We have already discussed our interest in the proofs we are going to show; here, we will not be so detailed as in the previous sections, since the formal definition of the Collection Method for the systems we will discuss, has been presented in Chapter 6, and, moreover, the schema of the proofs is always the same.
8.4.1 □-Theories

In this section we want to prove that extending \( E \) with a set \( \mathcal{H} \) of \( E \)-Harrop axioms, we get a uniformly constructive formal system.

As for other theories and logics we have presented in this chapter, the only part of the proof which leads to state that \( E + \mathcal{H} \) is uniformly constructive, which has to be changed is the induction in Lemma 8.2.2.

We just report the new case:

- \( E \)-Harrop Axiom:

\[
\Gamma \vdash A, \ A \in \mathcal{H}
\]

By induction on the structure of the formula \( A \), we prove that \( A \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \):

- if \( A \equiv \Box B \) then, by definition, it is evaluated in \( \text{Inf}^*(\mathcal{I}) \).
- if \( A \equiv B \land C \) then, by definition, \( B \) and \( C \) are in \( \text{Inf}^*(\mathcal{I}) \), thus, by induction hypothesis, being \( E \)-Harrop formulas, both are evaluated, and so also \( B \land C \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \).
- if \( A \equiv B \rightarrow C \), if \( B \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), then \( B \in \text{Inf}^*(\mathcal{I}) \), hence, by definition of Har-Sub, \( C \in \text{Inf}^*(\mathcal{I}) \).

But, by induction hypothesis, being an \( E \)-Harrop, formula \( C \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), then \( B \rightarrow C \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), too.

- if \( A \equiv \forall x \cdot B(x) \), then, for every term \( t \) such that \( B(t) \in \text{Inf}^*(\mathcal{I}) \), since \( B(t) \) is an \( E \)-Harrop formula, by induction hypothesis, \( B(t) \) is evaluated. Hence, by definition, \( \forall x \cdot B(x) \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \). □

In Chapter 5 we imposed that a specification framework has to contain axioms of the form \( \Box P \leftrightarrow P \), for any atomic formula \( P \) of the signature.

In this part about \( E \)-Harrop theories, we have to remark that the addition of a family of rules of the kind

\[
\frac{\Box P \in \mathcal{A}}{P \in \mathcal{A}}
\]

where \( \mathcal{A} \) is a set of atomic formulas, does not modify the uniformly constructive character of the theory. In fact, if \( P \in \text{Inf}^*(\mathcal{I}) \), then, by definition, \( P \) is evaluated in \( \text{Inf}^*(\mathcal{I}) \), hence, applying the usual proof pattern, one proves that the whole system is still uniformly constructive, as required.

As a final remark, we note that a theory \( \mathcal{T} = \mathcal{A} \cup \mathcal{B} \), where \( \mathcal{A} \) is a set of atomic formulas, and \( \mathcal{B} \) is a \( \Box \)-theory is still uniformly constructive.

This fact, in the setting of specification frameworks, as we introduced them, allows to relax the syntax with no impacts on the expressive power, since \( \Box P \leftrightarrow P \) for any atomic formula \( P \), and on tools, since the Collection Method still works.
8.4.2 Identity

We introduced the Collection Method construction for the theory ID of equality in Chapter 6. In this section, we will prove that the logical system \( L + \text{ID} \) is uniformly constructive, if \( L \) is a uniformly constructive logical system.

We assume that the proof that \( L \) is uniformly constructive follows the schema we employed in all other cases.

The proof that \( L + \text{ID} \) is uniformly constructive, is identical to the one for \( L \), except, as usual, two new cases in the induction of Lemma 8.2.2:

- Reflexivity:

\[
\frac{\Gamma \vdash t = t}{A}
\]

Since \( t = t \) is an atomic formula, it is evaluated in \( \text{Inf}^*(\mathcal{I}) \).

- Substitution:

\[
\frac{\Gamma, \Gamma' \vdash t = s \quad a(s)}{a(t)}
\]

Again, by definition of the substitution rule, \( a(t) \) is an atomic formula, and, thus, it gets evaluated in \( \text{Inf}^*(\mathcal{I}) \).

\[\square\]

In Chapter 6, we introduced the ID-Sub operator; it is not necessary when we adopt the restricted version of the substitution rule, but it is the device which permits to perform the proof if we have only the unrestricted substitution rule. We leave the proof of that case to the reader, since it is just an induction on the structure of formulas.

8.4.3 Induction on Natural Numbers

As we noted in Chapter 6, Peano arithmetic is a \( \Box \)-theory plus the induction rule. We already know that adding to \( \mathbf{E} \) a \( \Box \)-theory leads us to a uniformly constructive system; now we will prove that adding also the induction principle does not modify this character of the theory.

We said in Chapter 6 that on closed proofs, it is possible to prove that \( \mathbf{E} + \mathbf{PA} \) is uniformly constructive; the reason for this fact lies in the definition of the Ind-Sub operator, where we need to fix a numeral which is the value of \( t \), the term up to which we use the induction.

Technically, the proof that \( \mathbf{E} + \mathbf{PA} \) is uniformly constructive is the same as for \( \mathbf{E} \) plus a \( \Box \)-theory, except for Lemma 8.2.2 where, having another inference rule, we need a new case:
• Induction:

\[
\begin{align*}
\Gamma & \vdash \Gamma, [B(p)] \\
\vdash & \quad (p) \\
A & \equiv B(0) \quad B(s(p)) \\
B(t) & \equiv \frac{\exists x. B(x) \quad (\exists z. B(z) \land z < p) \lor A}{A}
\end{align*}
\]

Let \( i \) be a number such that \( s^i(0) = t \) is provable; by induction on \( k, 0 \leq k \leq i \), we prove that \( B(s^k(0)) \) gets evaluated in \( \text{Inf}^*(I) \):

- \( k = 0 \): \( B(0) \) is evaluated by the primary induction hypothesis.
- \( k = s(k') \): by the secondary induction hypothesis \( B(k') \) is evaluated in \( \text{Inf}^*(I) \); applying the primary induction hypothesis to the induction step, and remembering the definition of Ind-Sub, one gets that \( B(s(k')) \) is evaluated in \( \text{Inf}^*(I) \).

So \( B(s^i(0)) \) is evaluated in \( \text{Inf}^*(I) \), but, by definition of Ind-Sub, \( s^i(0) = t \in \text{Inf}^*(I) \), and, being atomic, it is also evaluated. From the definition of Ind-Sub again,

\[
\frac{s^i(0) = t \quad B(s^i(0))}{B(t) \in \text{Coll}^*(I)},
\]

thus, by induction hypothesis, \( B(t) \) gets evaluated in \( \text{Inf}^*(I) \).  

Hence, \( \textbf{E} + \textbf{PA} \) is uniformly constructive; of course, one can prove on the same guidelines the \( \textbf{IL} + \textbf{PA} \) is uniformly constructive as well.

From this result, and remembering that adding a \( \Box \)-theory (Harrop theory) on a uniformly constructive system based on \( \textbf{E} \) (\( \textbf{IL} \)) does not change its character, one may prove that the logical theories developed for the Computer Arithmetic Toolkit, see Chapter 4, are all uniformly constructive, as well.

### 8.4.4 Descending Chain Principle

We discussed the role of the Descending Chain Principle in Chapter 5, and we introduced the Collection Method instance for it in Chapter 6; here we want to prove that the system \( L + \text{DCP} \), where \( L \) is a uniformly constructive system where the \( < \) relation is a well-ordering, is uniformly constructive.

As the reader may suppose, the proof that \( L \) is uniformly constructive follows the usual schema; adding the DCP rule does not modify the proof, except for the Lemma 8.2.2, where a new case in the main induction has to be considered:

• Descending Chain Principle:

\[
\begin{align*}
\Gamma & \vdash \Gamma, [B(p)] \\
\vdash & \quad (p) \\
A & \equiv \exists x. B(x) \quad (\exists z. B(z) \land z < p) \lor A \\
\end{align*}
\]
By induction hypothesis, \( \exists x. B(x) \) is evaluated in \( \text{Inf}^* (\mathcal{I}) \), then, there is a term \( t \) such that \( B(t) \) is evaluated.

Hence, we can find an index \( k \) such that
\[
\Gamma, B(p) : A \quad \vdash \quad \text{Exp}^k(\mathcal{I}) \quad \text{and} \quad \Gamma \cup \{ B(t) \} \subseteq \text{Inf}(\text{Exp}^k(\mathcal{I})),
\]

thus
\[
\left\{ \begin{array}{l}
\Gamma, B(p) : A \\
(\exists z. B(z) \land z < p) \lor A
\end{array} \right\} (p := t) \in \text{Exp}^{k+1}(\mathcal{I}) \subseteq \text{Exp}^* (\mathcal{I}).
\]

Applying the induction hypothesis again, we get that \( (\exists z. B(z) \land z < p) \lor A \) is evaluated, that is, \( \exists z. B(z) \land z < p \) is evaluated or \( A \) is evaluated. In the latter case, we are done; in the former, we know that there is a term \( t' \) such that \( B(t') \land t' < p \) is evaluated, i.e., \( B(t') \) and \( t' < t \) are both evaluated.

Iterating the reasoning on \( t' \), we have a sequence of terms, \( t, t', \ldots \), for which \( B(t), B(t'), \ldots \) are evaluated and \( t' < t, t'' < t', \ldots \) are evaluated, hence they are true since they are proved.

But \( < \) is a well ordering relation on the domain, so this sequence of terms cannot be infinite, and, for this reason, eventually \( A \) gets evaluated in \( \text{Inf}^* (\mathcal{I}) \).

Then, in the same way as for \( L \), it follows that \( L + \text{DCP} \) is uniformly constructive.

### 8.4.5 Further Remarks

In the previous sections, we have proven that many logical systems are uniformly constructive. One may easily verify that any logical system which can be defined from the tools we provide in the Constructive Verification Environment in its pure version, is, indeed, one of the systems we treated.

It is important to note that the traditional version of the Constructive Verification Environment has not this character: in fact, higher order logic is not uniformly constructive, since the law of excluded middle holds.

The formal machinery we presented, the Collection Method and the general schema for proving a system to be uniformly constructive, is far more powerful: it is possible to prove that many other logics and theories are uniformly constructive, essentially in the same way as we did till now.

We invite the reader to try, e.g., with \( \mathbf{IL} \) + Kur, that is, Kuroda logic (Gabbay, 1981; Avellone et al., 1996; Miglioli et al., 1994b; Miglioli et al., 1997), where the Kur rule is
\[
\forall x. \neg \neg A(x) \\
\neg \neg \forall x. A(x)
\]
or with \( \mathbf{IL} + \mathcal{H} \), where \( \mathcal{H} \) is a set of Harrop formulas.

The proving technique we adopted is not the only way to prove that a system is uniformly constructive; in (Ferrari, 1997b) another technique is presented which
leads to proofs for the Kreisel-Putnam logic, the Scott logic, the Grzegorzyck logic and Markov arithmetic. All these systems are problematic with our approach.

As a final remark, it is important to say that the notion of uniformly constructive formal system has a proper logical meaning, we borrowed to understand the relation between a constructive and a computational view of specification formulas.

8.5 Equivalence of the E and E-T Calculi

As we anticipated in Chapter 3, now will prove that the E and E-T calculi are equivalent, that is, they prove the same set of theorems. We will obtain this result by proving that E-T is sound and complete with respect to an appropriate Kripke semantics.

This results have been already proved in (Miglioli et al., 1989b), for the natural deduction presentation of E. Here we will work on the tableau calculus, which is new, and so both proofs are new, as well. The proving technique follows the main guidelines as in (Avelone et al., 1997b; Avelone et al., 1998b; Avelone et al., 1997c; Ferrarì, 1995; Ferrarì, 1997a; Miglioli et al., 1994b; Miglioli et al., 1995; Miglioli et al., 1997).

Before starting the formal part of the proofs, we want to make some notes on the semantics. In the whole thesis, we didn’t provide a semantical characterization of the E system, but we introduced, especially in Chapter 5, an intended semantics, the one which permits a computational reading of specifications. This kind of semantics has been studied in (Miglioli et al., 1989b) under the name of valuation forms semantics; the reading we gave for specifications in Chapter 5 is a direct simplification of this kind of semantics, restricted to the format we allow for specification formulas.

Hence, it may appear odd to give here a soundness proof and a completeness proof based on a Kripke semantics. It is not. In fact, the role of these proofs in the thesis is to show the equivalence of the E and the E-T calculi, as shown in Chapter 3. In this way, we are using the Kripke semantics of the E logic as an instrument to prove the result we are interested in, namely the equivalence of the E and the E-T calculi.

The interest of the Kripke semantics is wider: in fact, it permits to understand how the tableau calculus for the E logic has been synthesized, and how the translation algorithm which forms the main theme of Chapter 3, has been conceived.

Thus, we do not give the status of meaning to the semantics we are about to show, but it should be regarded as a precious instrument to construct algorithms and calculi enjoying interesting properties as the ones we presented in the previous chapters.

A Kripke model for the E logic is a quadruple \( \mathcal{R} = \langle P, \leq, \iota, D \rangle \), where \( P = \langle P, \leq \rangle \) is a partial ordered set, the frame, with the constraint that, for every \( \alpha \in P \), there is a \( \beta \in P \), with \( \alpha \leq \beta \) such that \( \beta \) is final, that is, for every \( \gamma \in P \), with \( \beta \leq \gamma \), \( \beta = \gamma \); \( D \) is the domain function, associating, to any \( \alpha \in P \), a domain \( D(\alpha) \) such that, for any \( \alpha, \beta \in P \), if \( \alpha \leq \beta \) then \( D(\alpha) \subseteq D(\beta) \); the \( \iota \) function, the valuation
function, associates with every $\alpha \in P$ a function from the set of atomic formulas to the set \{T, $\bot$, $\top$\} of truth values, where $\top$ stands for undefined. The $\iota$ function must satisfy the following conditions:

- for every $\alpha, \beta \in P$ and for every atomic formula $\phi$, if $\alpha \leq \beta$ and $\phi$ belongs to the domain of $\iota_\alpha$ then $\phi$ belongs to the domain of $\iota_\beta$, and $\iota_\alpha(\phi) = \iota_\beta(\phi)$.

- for every $\alpha \in P$ and for every atomic formula $\phi$, there is a $\beta \in P$ such that $\alpha \leq \beta$ and $\iota_\beta(\phi) \neq \top$.

The $\iota$ function can be extended to arbitrary formulas:

- $\iota_\alpha(\neg A) = \top$ iff $\iota_\alpha(A) = \bot$; $\iota_\alpha(\neg A) = \bot$ iff $\iota_\alpha(A) = \top$.

- $\iota_\alpha(A \land B) = \top$ iff $\iota_\alpha(A) = \top$ and $\iota_\alpha(B) = \top$; $\iota_\alpha(A \land B) = \bot$ iff $\iota_\alpha(A) = \bot$ or $\iota_\alpha(B) = \bot$.

- $\iota_\alpha(A \lor B) = \top$ iff $\iota_\alpha(A) = \top$ or $\iota_\alpha(B) = \top$; $\iota_\alpha(A \lor B) = \bot$ iff $\iota_\alpha(A) = \bot$ and $\iota_\alpha(B) = \bot$.

- $\iota_\alpha(A \rightarrow B) = \top$ iff, for every $\beta \in P$, $\alpha \leq \beta$, either $\iota_\beta(A) = \top$ or $\iota_\beta(A) = \bot$, or $\iota_\beta(B) = \bot$; $\iota_\alpha(A \rightarrow B) = \bot$ iff $\iota_\alpha(A) = \top$ and $\iota_\alpha(B) = \bot$.

- $\iota_\alpha(\Box A) = \top$ iff, for every $\beta \in P$, $\alpha \leq \beta$, final, $\iota_\beta(A) = \top$; $\iota_\alpha(\Box A) = \bot$ iff, for every $\beta \in P$, $\alpha \leq \beta$, either $\iota_\alpha(A) = \top$ or $\iota_\alpha(A) = \bot$.

- $\iota_\alpha(\forall x. A(x)) = \top$ iff, for every $\beta \in P$, $\alpha \leq \beta$, for every $c \in D(\beta)$, $\iota_\beta(A(c)) = \top$; $\iota_\alpha(\forall x. A(x)) = \bot$ iff there is $c \in D(\alpha)$ such that $\iota_\alpha(A(c)) = \bot$.

- $\iota_\alpha(\exists x. A(x)) = \top$ iff there is $c \in D(\alpha)$ such that $\iota_\alpha(A(c)) = \top$; $\iota_\alpha(\exists x. A(x)) = \bot$ iff, for every $\beta \in P$, $\alpha \leq \beta$, for every $c \in D(\beta)$, $\iota_\beta(A(c)) = \bot$.

One can easily verify that, for every formula $A$,

- if $\alpha \leq \beta$, and $\iota_\alpha(A) \neq \bot$, then $\iota_\alpha(A) = \iota_\beta(A)$.

- for every $\beta \in P$, final, $\iota_\beta(A) \neq \top$.

The meaning of the signs $\textbf{T}$, $\textbf{F}$, $\textbf{F}_c$, $\textbf{A}_\Box$ and $\textbf{D}_\Box$ is explained in terms of realizability as follows: given a Kripke model $\mathfrak{R} = \langle P, \leq, \iota, D \rangle$, an element $\alpha \in P$ and a suff $H$, we say that $\alpha$ realizes $H$ (in $\mathfrak{R}$) if

1. if $H \equiv \textbf{T}A$, then $\iota_\alpha(A) = \top$;
2. if $H \equiv \textbf{F}A$, then $\iota_\alpha(A) = \top$;
3. if $H \equiv \textbf{F}_cA$, then $\iota_\alpha(A) = \bot$;
4. if $H \equiv \textbf{A}_\Box A$, then for every $\beta$ final, $\iota_\beta(A) = \top$;
5. if $H \equiv \textbf{D}_\Box A$, then for every $\beta$ final, $\iota_\beta(A) = \bot$. 

We say that $\alpha$ realizes a set of swiffs $S$ (and we write $\alpha \triangleright S$) iff $\alpha$ realizes every swiff in $S$. A set of swiffs $S$ is realizable iff there is some element $\alpha$ of a Kripke model $\mathfrak{A}$ such that $\alpha \triangleright S$. A configuration $S_1 \mid \ldots \mid S_n$ is realizable iff at least an $S_j$ is realizable.

8.5.1 Soundness

The calculus we will work on is shown in Tables 8.1, 8.2 and 8.3. If one compares the calculus we have illustrated in Chapter 3 with this one, there are few differences. In fact, the soundness theorem we are about to prove holds for both calculi. Of course, this is not the case for the completeness theorem. From a practical point of view, this is not a great problem, since, from our tests of the Constructive Reasoner, it appears that the valid formulas of $\mathbf{E}$ which are not theorems of $\mathbf{E}$-T are rare, and, apparently, not important for the use we want to make of the prover in a verification system.

As one could guess, the soundness theorem is proven starting from the notion of realizability, we already introduced in Chapter 3.

The outline of the proof of the soundness theorem is as follows:

- We prove that whenever a state in a Kripke model realizes a set of formulas, then there is another state in the same Kripke model which realizes the tableau configuration as generated from the application of an expansion rule.

- We prove that a closed node cannot be realized.

- We show that the initial configuration of a closed tableau cannot be realized.

- It follows that the $\mathbf{E}$-T calculus is sound.

In the following, we will assume that $\alpha$ is a state in a generic Kripke model $\mathfrak{A}$. Also, we will use lowercase greek letters to denote states in Kripke models.

Remembering the semantics for the $\mathbf{E}$ calculus, it will be useful to have a notation for the set of final states over a given state $\alpha$:

$$\text{Fin}(\alpha) = \{\beta \mid \beta \geq \alpha \land \beta \text{ is final}\}.$$ 

For every inference rule of $\mathbf{E}$-T, the antecedent has the form $S, A$, where $A$ is the active signed formula, and $S$ will be referred to as the context of that node. The consequent will be a configuration where the context appears unchanged, as $S$, or it may modified, either as $S_1$ or as $S_2$.

The following lemmas will show that if $S$ is realized by $\alpha$, then $S_1$ is realized by every successor or $\alpha$, and $S_2$ is realized by every final state over $\alpha$.

**Lemma 8.5.1** If $\alpha \triangleright S$, then $\forall \beta \geq \alpha. \beta \triangleright S_1$.

**Proof:** By cases on the elements of $S_1$:
<table>
<thead>
<tr>
<th>$S, T(A \land B)$</th>
<th>$S, F(A \land B)$</th>
<th>$S, F_e(A \land B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, F_eA</td>
<td>S, F_eB$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S, T(A \lor B)$</th>
<th>$S, F(A \lor B)$</th>
<th>$S, F_e(A \lor B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, T \lor A</td>
<td>S, T, A$</td>
<td>$S, F \lor A</td>
</tr>
<tr>
<td>$S, F_eA</td>
<td>S, T, A$</td>
<td></td>
</tr>
</tbody>
</table>

For $T$-rules see Table 8.2

<table>
<thead>
<tr>
<th>$S, F(A \rightarrow B)$</th>
<th>$S, T, A, F_eB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, F(A \rightarrow B)$</td>
<td>$S, T, A, F_eB$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S, T \square A$</th>
<th>$S, F \square A$</th>
<th>$S, F_e \square A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, T \square A$</td>
<td>$S, F \square A$</td>
<td>$S, F_e \square A$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S, T \forall x. A(x)$</th>
<th>$S, F_e \forall x. A(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, T A(c), T \forall x. A(x)$</td>
<td>$S, F_e A(a)$</td>
</tr>
<tr>
<td>$S, F \forall x. A(x)$</td>
<td>$S, F \forall x. A(x)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S, T \exists x. A(x)$</th>
<th>$S, F_e \exists x. A(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, T A(a)$</td>
<td>$S, T A(a)$</td>
</tr>
<tr>
<td>$S, F \exists x. A(x)$</td>
<td>$S, F \exists x. A(x)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S, T A(c), T \exists x. A(x)$</th>
<th>$S, F_e A(a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, T A(c), T \exists x. A(x)$</td>
<td>$S, T A(a)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S, F A(c), F \exists x. A(x)$</th>
<th>$S, T A(a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, F A(c), F \exists x. A(x)$</td>
<td>$S, T A(a)$</td>
</tr>
</tbody>
</table>

### Table 8.1: The Calculus \(E-T\)
\[
\begin{align*}
  & \frac{S, T(A \rightarrow B)}{S, FA | S, F_e A | S, TB} \quad \text{T→At with A atomic}, \\
  & \quad \frac{S, T((A \land B) \rightarrow C)}{S, T((A \lor B) \rightarrow C)} \quad \text{T→∧} \\
  & \quad \frac{S, T(A \rightarrow (B \rightarrow C))}{S, T(A \rightarrow C), T(B \rightarrow C)} \quad \text{T→∨} \\
  & \quad \frac{S, T((A \rightarrow B) \rightarrow C)}{S, T(A \rightarrow B), T(B \rightarrow C) | S, TC | S, T A, F_e B} \quad \text{T→→} \\
  & \quad \frac{S, F(A \rightarrow B), T(B \rightarrow C) \mid S, TC \mid S, T A, F_e B}{S, T(\forall x. A(x) \rightarrow C)} \quad \text{T→∀} \\
  & \quad \frac{S, T(\exists x. A(x) \rightarrow B)}{S, T(\exists x. A(x) \rightarrow B)} \quad \text{T→∃} \\
  & \frac{S, F\neg A | S, F_e \neg A | S, TB}{S, T(\neg A \rightarrow B)} \quad \text{T→¬At with A atomic}, \\
  & \quad \frac{S, T((\neg A \lor B) \rightarrow C)}{S, T((\neg A \land B) \rightarrow C)} \quad \text{T→¬∧} \\
  & \quad \frac{S, T((\neg A \rightarrow B) \rightarrow C)}{S, T((\neg A \rightarrow B) \rightarrow C)} \quad \text{T→¬∨} \\
  & \quad \frac{S, T(A \rightarrow B)}{S, T(\neg (A \rightarrow B))} \quad \text{T→¬¬} \\
  & \quad \frac{S, T(\neg \forall x. A(x) \rightarrow C)}{S, T(\neg \exists x. A(x) \rightarrow C)} \quad \text{T→¬∀} \\
  & \quad \frac{S, T(\exists x. \neg A(x) \rightarrow C)}{S, T(\forall x. \neg A(x) \rightarrow C)} \quad \text{T→¬∃}
\end{align*}
\]

Table 8.2: Rules for T→.
Table 8.3: Rules for $T_\Box$ and $F_\Box$.

- $TA \in S_c \implies TA \in S \implies \alpha \triangleright TA \implies t_\alpha(A) = \top \implies \forall \beta \geq \alpha. t_\beta(A) = \top \implies \forall \beta \geq \alpha. \beta \triangleright TA$.

- $F_\Box A \in S_c$: this case is proven as the previous one.

- $T_\Box A \in S_c \implies T_\Box A \in S \implies \alpha \triangleright T_\Box A \implies \forall \gamma \in \text{Fin}(\alpha). t_\gamma(A) = \top \implies$ being $\text{Fin}(\beta) \subseteq \text{Fin}(\alpha)$, when $\beta \geq \alpha$, $\forall \gamma \in \text{Fin}(\beta). \beta \triangleright T_\Box A$.

- $F_\Box A \in S_c$: this case is proven as the previous one.

Lemma 8.5.2 If $\alpha \triangleright S$, then $\forall \beta \in \text{Fin}(\alpha). \beta \triangleright S_\Box$.

Proof: By cases on the elements of $S_\Box$:

- $T_\Box A \in S_\Box \implies T_\Box A \in S \lor TA \in S$
  - $T_\Box A \in S \implies \alpha \triangleright T_\Box A \implies \forall \beta \in \text{Fin}(\alpha). t_\beta(A) = \top \implies$ the set of final states over a final state id the singleton containing the state, $\forall \beta \in \text{Fin}(\alpha). \forall \gamma \in \text{Fin}(\beta). t_\gamma(A) = \top \implies \forall \beta \in \text{Fin}(\alpha). \beta \triangleright T_\Box A$.
  - $TA \in S \implies \alpha \triangleright TA \implies t_\alpha(A) = \top \implies \forall \beta \geq \alpha. t_\beta(A) = \top \implies \forall \beta \in \text{Fin}(\alpha). t_\beta(A) = \top \implies \alpha \triangleright T_\Box A$, hence, $\forall \beta \in \text{Fin}(\alpha). \beta \triangleright T_\Box A$.

- $F_\Box A \in S_\Box$: this case is specular to the previous one.
Now, it will follow a long list of properties that takes care of establishing which states realizes the configuration derived from the expansion of an active formula by means of an inference rule in $E\Gamma$. Most of the proofs are trivial, so we omit them. We recall that $\alpha \nvdash \{A_1, A_2, \ldots\}$ is a shorthand for $\alpha \nvdash A_1$ and $\alpha \nvdash A_2, \ldots$

**Lemma 8.5.3**  
1. If $\alpha \vdash T(A \land B)$ then $\alpha \vdash \{TA, TB\}$.
2. If $\alpha \vdash T(A \lor B)$ then $\alpha \vdash TA$ or $\alpha \vdash TB$.
3. If $\alpha \vdash T\neg A$ then $\alpha \vdash F_c A$.
4. If $\alpha \vdash T\Box A$ then $\alpha \vdash T\Box A$.
5. If $\alpha \vdash T(A \rightarrow B)$ then $\forall \beta \geq \alpha. \beta \vdash FA \lor \beta \vdash F_c A \lor \beta \vdash TB$.
6. If $\alpha \vdash T(A \rightarrow B)$ then $\alpha \vdash FA \lor \alpha \vdash F_c A \lor \alpha \vdash TB$.

**Proof:** All the proofs are immediate by expanding the definitions of $\vdash$ relation and of the $\iota$ function. \hfill \square

**Lemma 8.5.4**  
1. If $\alpha \vdash F_c (A \land B)$ then $\alpha \vdash F_c A$ or $\alpha \vdash F_c B$.
2. If $\alpha \vdash F_c (A \lor B)$ then $\alpha \vdash F_c A$ and $\alpha \vdash F_c B$.
3. If $\alpha \vdash F_c \neg A$ then $\alpha \vdash TA$.
4. If $\alpha \vdash F_c \Box A$ then $\alpha \vdash F_c A$.
5. If $\alpha \vdash F_c (A \rightarrow B)$ then $\alpha \vdash TA$ and $\alpha \vdash F_c B$.

**Proof:** All the proofs are immediate by expanding the definitions of $\vdash$ relation and of the $\iota$ function. \hfill \square

**Lemma 8.5.5**  
1. If $\alpha \vdash F(A \land B)$ then $\alpha \vdash \{FA, FB\}$ or $\alpha \vdash \{FA, TB\}$ or $\alpha \vdash \{TA, FB\}$.
2. If $\alpha \vdash F(A \lor B)$ then $\alpha \vdash \{FA, FB\}$ or $\alpha \vdash \{FA, F_c B\}$ or $\alpha \vdash \{F_c A, FB\}$.
3. If $\alpha \vdash F\neg A$ then $\alpha \vdash FA$.
4. If $\alpha \vdash F\Box A$ then $\exists \beta, \gamma \in \text{Fin}(\alpha). \beta \vdash T\Box A \land \gamma \vdash F\Box A$.
5. If $\alpha \vdash F(A \rightarrow B)$ then $\exists \beta \geq \alpha. \beta \vdash \{TA, FB\} \lor \beta \vdash \{TA, F_c B\}$.
6. If $\alpha \vdash F(A \rightarrow B)$ then $\alpha \vdash \{TA, FB\} \lor \alpha \vdash \{FA, FB\}$.

**Proof:** Most of the cases are solved by expanding definitions. The non trivial ones are
4. \( \alpha \uparrow \square F A \implies \iota_A(\square A) = \uparrow \implies \exists \beta, \gamma \in \text{Fin}(\alpha). \iota_{\beta}(A) = \top \land \iota_{\gamma}(A) = \bot \implies \exists \beta, \gamma \in \text{Fin}(\alpha). \beta \uparrow TA \land \gamma \uparrow FC A \implies \) being \( \beta \) and \( \gamma \) final, and following Lemma 8.5.2, \( \exists \beta, \gamma \in \text{Fin}(\alpha). \beta \uparrow TA \land \gamma \uparrow FC A. \)

5. \( \alpha \uparrow F (A \rightarrow B) \implies \iota_A(A \rightarrow B) = \uparrow \implies \exists \beta \geq \alpha. \iota_{\beta}(A) = \top \land \iota_{\beta}(B) \neq \top \implies \exists \beta \geq \alpha. (\iota_{\beta}(A) = \top \land \iota_{\beta}(B) = \bot) \implies \exists \beta \geq \alpha. (\iota_{\beta}(A) = \top \land \iota_{\beta}(B) = \bot) \implies \exists \beta \geq \alpha. \beta \uparrow \{TA, FB\} \lor \top \beta \uparrow \{TA, FA, FB\}. \)

6. \( \alpha \uparrow F(A \rightarrow B) \implies \iota_A(A \rightarrow B) = \uparrow \implies \iota_A(A) \neq \top \lor \iota_A(B) \neq \bot. \) Now, if \( \iota_A(B) = \top \), then, by definition, it follows that \( \iota_A(A \rightarrow B) = \top \), which is impossible, thus \( \alpha \uparrow FB \). Considering \( A \), it cannot be that \( \iota_A(A) = \bot \), otherwise \( \iota_A(A \rightarrow B) = \top \), then we get that \( \alpha \uparrow \{TA, FB\} \) or \( \alpha \uparrow \{FA, FB\} \).

Lemma 8.5.6  
1. If \( \alpha \uparrow TA(A \land B) \) then \( \alpha \uparrow \{TA, TA\} \).

2. If \( \alpha \uparrow TA(A \lor B) \) then \( \forall \beta \in \text{Fin}(\alpha). \beta \uparrow TA \lor \beta \uparrow TB \).

3. If \( \alpha \uparrow TA \neg A \) then \( \alpha \uparrow FA \).

4. If \( \alpha \uparrow TA \square A \) then \( \alpha \uparrow TA \).

5. If \( \alpha \uparrow TA(A \rightarrow B) \) then \( \forall \beta \in \text{Fin}(\alpha). \beta \uparrow FA \lor \beta \uparrow TB \).

Proof: All the proofs are immediate by expanding the definitions of \( \uparrow \) relation and of the \( \iota \) function.

\[ \square \]

Lemma 8.5.7  
1. If \( \alpha \uparrow FA(A \land B) \) then \( \forall \beta \in \text{Fin}(\alpha). \beta \uparrow FA \lor \beta \uparrow FB \).

2. If \( \alpha \uparrow FA(A \lor B) \) then \( \alpha \uparrow FA \) and \( \alpha \uparrow FA \).

3. If \( \alpha \uparrow FA \neg A \) then \( \alpha \uparrow TA \).

4. If \( \alpha \uparrow FA \square A \) then \( \alpha \uparrow FA \).

5. If \( \alpha \uparrow FA(A \rightarrow B) \) then \( \alpha \uparrow TA \land \alpha \uparrow FA \).

Proof: All the proofs are immediate by expanding the definitions of \( \uparrow \) relation and of the \( \iota \) function.

\[ \square \]

Lemma 8.5.8  
1. If \( \alpha \uparrow TA(A \rightarrow B), A \) atomic, then \( \alpha \uparrow FA \) or \( \alpha \uparrow FA \).

2. If \( \alpha \uparrow TA(\neg A \rightarrow B), A \) atomic, then \( \alpha \uparrow FA \) or \( \alpha \uparrow FA \).

3. If \( \alpha \uparrow TA(A \land B \rightarrow C) \) then \( \alpha \uparrow TA(A \rightarrow (B \rightarrow C)). \)

4. If \( \alpha \uparrow TA(\neg(A \land B) \rightarrow C) \) then \( \alpha \uparrow TA(\neg A \lor \neg B \rightarrow C). \)

Proof: All the proofs are immediate by expanding the definitions of \( \uparrow \) relation and of the \( \iota \) function.

\[ \square \]
5. If $\alpha \models T(A \lor B \rightarrow C)$ then $\alpha \models T(A \rightarrow C)$ and $\alpha \models T(B \rightarrow C)$.

6. If $\alpha \models T(\neg (A \lor B) \rightarrow C)$ then $\alpha \models T(\neg A \land \neg B \rightarrow C)$.

7. If $\alpha \models T(\neg A \rightarrow B)$ then $\alpha \models T(A \rightarrow B)$.

8. If $\alpha \models T(\square A \rightarrow B)$ then $\alpha \models F \square A$ or $\alpha \models F_c \square A$ or $\alpha \models TB$.

9. If $\alpha \models T(\neg \square A \rightarrow B)$ then $\alpha \models F \neg \square A$ or $\alpha \models F_c \neg \square A$ or $\alpha \models TB$.

10. If $\alpha \models T((A \rightarrow B) \rightarrow C)$ then $\alpha \models TC$, or $\alpha \models \{F(A \rightarrow B), T(B \rightarrow C)\}$ or $\alpha \models \{T_A, F_c B\}$.

11. If $\alpha \models T(\neg (A \rightarrow B) \rightarrow C)$ then $\alpha \models T(A \land \neg B \rightarrow C)$.

Proof: Cases 1, 2, 8, 9 are instances of point 6 in Lemma 8.5.3. Cases 4, 6, 7 and 11 are immediate consequence of the definition of realizability and of the semantics. The remaining cases are slightly more complex, and we will show the proof of case 5.

From $\alpha \models T(A \lor B \rightarrow C)$ it follows that $\forall \beta \geq \alpha. \iota_\beta(A \lor B) \models \top \land \iota_\beta(C) = \top$, that is, $\forall \beta \geq \alpha. (\iota_\beta(A) \not= \top \land \iota_\beta(B) \not= \top) \lor \iota_\beta(C) = \top$, thus $(\forall \beta \geq \alpha. \iota_\beta(A) \not= \top \land \iota_\beta(B) \not= \top \land \iota_\beta(C) = \top)$, that can be rewritten as $\iota_\alpha(A \rightarrow C) = \top \land \iota_\beta(B \rightarrow C) = \top$ which is equivalent to $\alpha \models \{T(A \rightarrow C), T(B \rightarrow C)\}$.

Lemma 8.5.9

1. If $\alpha \models T \forall x. A(x)$ then $\forall \beta \geq \alpha. \forall c \in D(\beta). \beta \models A(c)$.

2. If $\alpha \models T \forall x. A(x)$ then $\forall c \in D(\alpha). \alpha \models A(c)$.

3. If $\alpha \models F \forall x. A(x)$ then $\exists \beta \geq \alpha. \exists a \in D(\beta). \beta \models F_c A(a)$ or else $\exists \beta \geq \alpha. \exists a \in D(\beta). \beta \models F A(a)$.

4. If $\alpha \models F \forall x. A(x)$ then $\alpha \models F A(c)$ or $\alpha \models T A(c)$, for any $c \in D(\alpha)$.

5. If $\alpha \models F_c \forall x. A(x)$ then $\exists a \in D(\alpha). \alpha \models F_c A(a)$.

6. If $\alpha \models T \exists x. A(x)$ then $\exists a \in D(\alpha). \alpha \models T A(a)$.

7. If $\alpha \models F \exists x. A(x)$ then $\exists \beta \geq \alpha. \exists a \in D(\beta). \beta \models T A(a)$ or $\exists \beta \geq \alpha. \exists a \in D(\beta). \beta \models F A(a)$.

8. If $\alpha \models F \exists x. A(x)$ then $\forall c \in D(\alpha). \alpha \models F A(c) \lor \alpha \models F_c A(c)$.

9. If $\alpha \models F_c \exists x. A(x)$ then $\forall \beta \geq \alpha. \forall c \in D(\beta). \beta \models F_c A(c)$.

10. If $\alpha \models F_c \exists x. A(x)$ then $\forall c \in D(\alpha). \alpha \models F_c A(c)$.

Proof: These facts are consequences of the definition of the $\models$ relation and of the semantics of quantifiers in the $\mathbf{E}$ logic.
Lemma 8.5.10  1. If \( \alpha \vdash T((\forall x. A(x)) \rightarrow B) \) then \( \alpha \vdash F\forall x. A(x) \) or \( \alpha \vdash TB \) or \( \alpha \vdash Fc\forall x. A(x) \).

2. If \( \alpha \vdash T((\exists x. A(x)) \rightarrow B) \) then \( \alpha \vdash T(\forall x. (A(x) \rightarrow B)) \).

3. If \( \alpha \vdash T(\neg(\forall x. A(x)) \rightarrow B) \) then \( \alpha \vdash T((\exists x. \neg A(x)) \rightarrow B) \).

4. If \( \alpha \vdash T(\neg(\exists x. A(x)) \rightarrow B) \) then \( \alpha \vdash T((\forall x. \neg A(x)) \rightarrow B) \).

5. If \( \alpha \vdash T\Box\forall x. A(x) \) then \( \forall c \in D(\alpha). \alpha \vdash T\Box A(c) \).

6. If \( \alpha \vdash F\Box\forall x. A(x) \) then \( \forall \beta \in Fin(\alpha), \exists a \in D(\beta). \beta \vdash F\Box A(a) \).

7. If \( \alpha \vdash T\Box\exists x. A(x) \) then \( \forall \beta \in Fin(\alpha), \exists a \in D(\beta). \beta \vdash T\Box A(a) \).

8. If \( \alpha \vdash F\Box\exists x. A(x) \) then \( \forall c \in D(\alpha). \alpha \vdash F\Box A(c) \).

Proof: By unfolding the definitions of \( \vdash \) and of the semantics of quantifiers, all these facts follow. \( \square \)

The result of putting together the preceding lemmas is

Lemma 8.5.11 Let \( \alpha \) be a node in Kripke model \( \mathcal{K} \), and let \( N \) be a set of suffs such that \( \alpha \vdash N \); if \( \Gamma \) is a configuration obtained by expanding \( N \) according to the \( \mathbf{E} \) tableau rules, then there is \( N' \in \Gamma \) and \( \beta \succeq \alpha \) such that \( \beta \vdash N' \).

Proof: By cases on the expansion rules; it trivially follows from the facts we have proven in Lemmas 8.5.1, 8.5.2, 8.5.3, 8.5.4, 8.5.5, 8.5.6, 8.5.7, 8.5.8, 8.5.9 and 8.5.10.

The meaning of Lemma 8.5.11 is that realizability is preserved by the inference rule of \( \mathbf{E} \)-\( \mathbf{T} \). Having proved this fact, we finish the first part of the soundness proof. The rest of the proof is much more compact, since it does not involve a deep case analysis as the one we have just shown.

Lemma 8.5.12 A closed node in a tableau cannot be realized.

Proof: Let \( N \) be a closed node, and let \( \alpha \) be any node in any Kripke model:

- \( TA, FA \in N \): Let us suppose that \( \alpha \vdash N \Rightarrow \alpha \vdash TA \) and \( \alpha \vdash FA \Rightarrow \nu_\alpha(A) = \top \) and \( \nu_\alpha(A) = \bot \), which is clearly impossible.

- \( TA, FcA \in N \): Let us suppose that \( \alpha \vdash N \Rightarrow \alpha \vdash TA \) and \( \alpha \vdash FcA \Rightarrow \nu_\alpha(A) = \bot \) and \( \nu_\alpha(A) = \bot \), which is impossible.

- \( FcA, FA \in N \): Let us suppose that \( \alpha \vdash N \Rightarrow \alpha \vdash FcA \) and \( \alpha \vdash FA \Rightarrow \nu_\alpha(A) = \bot \) and \( \nu_\alpha(A) = \top \), impossible.

- \( T\Box A, F\Box A \in N \): Let us suppose that \( \alpha \vdash N \Rightarrow \alpha \vdash T\Box A \) and \( \alpha \vdash F\Box A \Rightarrow \nu_\alpha(A) = \bot \) and \( \nu_\beta(A) = \bot \), again impossible. \( \square \)
Lemma 8.5.13 The initial configuration of a closed tableau cannot be realized.

Proof: By induction on the depth of the tableau one proves that, if the initial configuration can be realized then a terminal node can be realized; the induction step is Lemma 8.5.11.

From Lemma 8.5.12, we know that no terminal node in a closed tableau can be realized, hence the initial configuration cannot, as well. □

The preceding lemma gives us the instrument to prove the soundness statement.

Theorem 8.5.1 (Soundness) If \{FA\} and \{F □ A\} generate closed tableaus, then \(E \models A\).

Proof: From Lemma 8.5.13 applied to a closed tableau for \{FA\}, one gets that, for every node \(\alpha\) in any Kripke model \(\mathfrak{A}\), \(\alpha \Vdash FA\), that is \(\iota_\alpha(A) = \top\) or \(\iota_\alpha(A) = \bot\). Thus, \(\alpha \Vdash T(A \lor \neg A)\).

Now, from Lemma 8.5.13 applied to a closed tableau for \{F □ \}, one gets that, for every node \(\alpha\) in any Kripke model \(\mathfrak{A}\), \(\alpha \Vdash F □ A\), that means, for every final node \(\beta\), \(\iota_\beta(A) = \top\).

But every model has a final state, and truth is preserved between nodes, so no state \(\alpha\) in any Kripke model may exists for which \(\iota_\alpha(A) = \bot\).

Then, for every node in any Kripke model, \(\iota(A) = \top\), i.e., \(E \models A\). □

We want to conclude this part with some minor remarks:

- The proof we gave holds for both versions of E-T we presented, here and in Chapter 3; we took care of covering all the small differences in the calculi, when we did the long case analysis which leads to Lemma 8.5.11.

- The rules for T □ and F □ are, in fact, classical, hence one can simply use a theorem prover for classical logic when there is the need to decide whether T □ A or F □ A.

- In the translation to natural deduction, we presented in Chapter 3, we mapped FA into \(A \lor \neg A\); the reason now, after showing the soundness proof, should be clear. This fact should give to the reader an idea on how a translation algorithm, like the ones we gave, can be conceived for a generic logic.

- A small improvement in the compactness of the soundness argument will be to consider, instead of two tableaus, just one, starting from the initial configuration FA | F □ A; it reduces immediately to the form we considered above.

- Considering other initial configurations we get other soundness arguments: if a tableau for FA | FcA is closed then E \(\models A\); if a tableau for FA | T □ A is closed then E \(\models \neg A\); if a tableau for FA | TA is closed then E \(\models \neg A\). All these formulations are equivalent, as one may easily prove.

- Defining intuitionistic negation as \(A \rightarrow P \land \neg P\), where \(P\) is any atomic formula, it is immediate to prove that IL can be embedded into E.
8.5.2 Completeness

A proof of a wff $B$ is a closed proof table starting from $FB \mid F_B B$. A possibly infinite set $S$ is consistent iff no proof table starting from any finite subset of $S$ is closed.

We call regular wff any wff to which a regular rule is applicable, where the regular rules of $\text{E-T}$ are:

- $T \land, F \land, F_c \land, T \Box \land$;
- $T \lor, F \lor, F_c \lor, F \Box \lor$;
- $T \neg, F \neg, F_c \neg, T \Box \neg, F \Box \neg$;
- Each of the $T \rightarrow$ rules, $F \rightarrow_1, F_c \rightarrow, F \Box \rightarrow$;
- $T \Box, F \neg \Box$;
- $T \lor, F \lor, F_c \lor, T \Box \lor$;
- $T \exists, F \exists, F_c \exists, F \Box \exists$.

We call $c$-wff any wff to which a $c$-rule is applicable where the $c$-rules are:

- $F \rightarrow_2$;
- $F \lor_2$;
- $F \exists_2$.

Finally, we call $cl$-wff any wff to which a $cl$-rule is applicable, where the $cl$-rules are:

- $F \Box_1, F \Box_2$;
- special;
- $F \Box \land$;
- $T \Box \lor$;
- $T \Box \rightarrow$;
- $F \Box \lor$;
- $T \Box \exists$.

Given a wff $H$ in the language of $\text{E}$, we call extension(s) of $H$ the set(s) $R^1_H, \ldots, R^n_H$ $(n \geq 1)$ coinciding with the sets in the configuration obtained by applying the rule related to $H$ in $\text{E}$ to the configuration $\{H\}$.

Moreover, let us define the following measures on wffs and swffs. The degree of a wff $A$, denoted by $\operatorname{deg}(A)$, is defined as:
• $\text{deg}(A) = 1$ if $A$ is atomic;
• $\text{deg}(\neg A) = \text{deg}(\Box A) = \text{deg}(A) + 1$,
• $\text{deg}(\forall x. A(x)) = \text{deg}(\exists x. A(x)) = \text{deg}(A) + 1$,
• $\text{deg}(A) = \max\{\text{deg}(B), \text{deg}(C)\} + 1$ if $A$ is $B \land C$, $B \lor C$ or $B \rightarrow C$.

The degree of a swff $SA$ (where $S$ is either $T$, $F$, $F_c$, $T_{\Box}$ or $T_{\land}$, denoted by $\text{deg}(SA)$, coincides with the degree of $A$.

The implicative complexity of $A$, denoted by $\text{IC}(A)$, is defined as follows:

• if $A$ is implication free, then $\text{IC}(A) = 0$,
• if $A \equiv \neg B \rightarrow C$, then $\text{IC}(A) = \text{deg}(\neg B) + 1$,
• if $A \equiv B \rightarrow C$ and $B$ is not a negated wff, then $\text{IC}(A) = \text{deg}(B)$,
• if $A \equiv \neg B$ or $A \equiv \Box B$, then $\text{IC}(A) = \text{IC}(B)$,
• if $A \equiv B \land C$ or $A \equiv B \lor C$, then $\text{IC}(A) = \max\{\text{IC}(B), \text{IC}(C)\}$.
• if $A \equiv \forall x. A(x)$ or $A \equiv \exists x. A(x)$ then $\text{IC}(A) = \text{IC}(B(x))$.

The well founded relation $\prec$ on pairs of swffs of $E$ is defined as follows: $SA \prec S'A'$ iff

• either $\text{deg}(SA) < \text{deg}(S'A')$,
• or $\text{deg}(SA) = \text{deg}(S'A')$ and $\text{IC}(A) < \text{IC}(A')$.

Now, it is easy to check that every rule of the calculus $E$ except for the rules $F \rightarrow 1$, $T \forall$, $F \lor 1$, $F \exists 1$, $F \exists 3$, $T \rightarrow \forall$ is duplication free in the sense explained in (Avellone et al., 1997c; Dyckhoff, 1992; Miglioli et al., 1994c; Miglioli et al., 1994b; Miglioli et al., 1997). The formal counterpart of "duplication-free" rule is given in terms of the $\prec$ relation as follows: a rule is duplication-free if, denoted with $H$ its main swff, for every extension $R^i_H$ of $H$, $K \prec H$ for every $K \in R^i_H$. For instance, the rule $T \rightarrow \neg \land$ replaces the swff $T((\neg A \land \neg B) \rightarrow C)$ with the swffs $T((\neg A \lor \neg B) \rightarrow C)$ and $T((\neg A \land \neg B) \rightarrow C) \prec T((\neg A \lor \neg B) \rightarrow C)$. This gives rise to an obvious definition of a well founded measure which is lowered passing from a configuration to which a rule different from $F \rightarrow 1$, $T \forall$, $F \lor 1$, $F \exists 1$, $F \exists 3$, $T \rightarrow \forall$ is applied to the subsequent one.

Now we begin to discuss the problem of the completeness of the calculus $E$. The completeness theorem has the following form: If a formula $A$ is valid in every $E$-model, then there is a closed proof-table in $E-T$ starting from $FA \mid F \Box A$. According to the semantical interpretation of the swffs, it suffices to prove the following fact: If a set $S$ of swffs is consistent, then there is an $E$-model $\mathcal{R}$ together with an element $\alpha$ of $\mathcal{R}$ such that $S$ is realized in $\alpha$. Thus, following (Fitting, 1969; Miglioli et al., 1994b; Miglioli et al., 1994c; Miglioli et al., 1997), our proof is based on a general
method allowing to build up a rooted Kripke model $K_E(S)$ on which a consistent set of swffs $S$ is realized. In this way, given a configuration $F_A \mid F \oplus A$ for which no closed proof-table in $E$ exists, we can build a $E$-model either realizing $F_A$ or $F \oplus A$.

First of all, given a language $L_C$ over a set of constant symbols $C$, let $\mathcal{K}$ be a denumerable set of parameters such that $\mathcal{K} \cap C = \emptyset$. Given a consistent set of swffs of the language $L_{C \cup \mathcal{K}}$, and a set $\Pi \subseteq C \cup \mathcal{K}$, we call $\Pi$ a reference set for $S$ if $\Pi$ is non empty and contains all the parameters occurring in $S$.

Now, given a consistent set $S$ of swffs of $L_{C \cup \mathcal{K}}$, and an (at most denumerable) reference set $\Pi$ for $S$ such that $\mathcal{K} \not\subseteq \Pi$, let us consider the enumerations:

- $\Xi_1 : A_1, \ldots, A_n \ldots$ of all the swffs of $S$ (without duplications of swffs);
- $\Xi_2 : p_1, \ldots, p_n, \ldots$ of the parameters of $\Pi$.

Starting from $\Xi_1$ and $\Xi_2$, we construct the sequences:

- $\{S_i\}_{i \in \omega}$ of sets of swffs of $L_{C \cup \mathcal{K}}$;
- $\{\Pi_i\}_{i \in \omega}$ of sets of parameters such that, for any $i \in \omega$, $\Pi_i \subseteq C \cup \mathcal{K}$, $\mathcal{K} \not\subseteq \Pi_i$, and $\Pi_i$ is a reference set for $S_i$.

defined as follows

- $S_0 = \emptyset$, $\Pi_0 = \emptyset$;
- Let $S_i = \{H_1, \ldots, H_k\}$ and $\Pi_i = \{p_1, \ldots, p_r\}$; then:

$$
S_{i+1} = \bigcup_{H_j \in S_i} U(H_j, i) \cup \{A_{i+1}\}$$

$$
\Pi_{i+1} = \Pi_i \cup \{p_{i+1}\} \cup \bigcup_{H_j \in S_i} \text{NewPar}(H_j, i)
$$

where, setting

$$S' = U(H_1, i) \cup \cdots \cup U(H_{j-1}, i) \cup \{H_j, \ldots, H_k, A_{i+1}, A_{i+2}, \ldots\}$$

we have that $U(H_j, i)$ is a set of swffs and $\text{NewPar}(H_j, i)$ is a set of new parameters, defined as follows:

1. If $H_j$ is a regular swff different from
   - $T(A \rightarrow B)$ with $A$ atomic,
   - $T(\Box A \rightarrow B)$,
   - $T(\neg A \rightarrow B)$ with $A$ atomic,
   - $T(\neg \Box A \rightarrow B)$,
   - $T \forall x. A(x)$,
   - $F_c \forall x. A(x)$,
• \( \mathcal{T} \models x. A(x) \),
• \( \mathcal{F} \models x. A(x) \),
• \( \mathcal{T} \vdash \forall x. A(x) \),
• \( \mathcal{F} \vdash \forall x. A(x) \),

then \( \mathcal{U}(H_j, i) \) is any extension \( \mathcal{R}_{H_j} \) of \( H_j \) such that \( (S' \setminus \{H_j\}) \cup \mathcal{R}_{H_j} \) is consistent. In this case \( \text{NewPar}(H_j, i) = \emptyset \).

2. If \( H_j \equiv \mathcal{T}(A \rightarrow B) \) with \( A \) atomic, then:
   
   (a) if \( (S' \setminus \{H_j\}) \cup \{TB\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{TB\} \);

   (b) else, if \( (S' \setminus \{H_j\}) \cup \{F_cA\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{F_cA\} \);

   (c) else, \( \mathcal{U}(H_j, i) = \{H_j\} \).

Moreover, \( \text{NewPar}(H_j, i) = \emptyset \).

3. If \( H_j \equiv \mathcal{T}(\Box A \rightarrow B) \), then:
   
   (a) if \( (S' \setminus \{H_j\}) \cup \{TB\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{TB\} \);

   (b) else, if \( (S' \setminus \{H_j\}) \cup \{F_c\Box A\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{F_c\Box A\} \);

   (c) else, \( \mathcal{U}(H_j, i) = \{H_j\} \).

Moreover, \( \text{NewPar}(H_j, i) = \emptyset \).

4. If \( H_j \equiv \mathcal{T}(\neg A \rightarrow B) \) with \( A \) atomic, then:
   
   (a) if \( (S' \setminus \{H_j\}) \cup \{TB\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{TC\} \);

   (b) else, if \( (S' \setminus \{H_j\}) \cup \{F_c\neg A\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{F_c\neg A\} \);

   (c) else, \( \mathcal{U}(H_j, i) = \{H_j\} \).

Moreover, \( \text{NewPar}(H_j, i) = \emptyset \).

5. If \( H_j \equiv \mathcal{T}(\neg \Box A \rightarrow B) \), then:
   
   (a) if \( (S' \setminus \{H_j\}) \cup \{TB\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{TC\} \);

   (b) else, if \( (S' \setminus \{H_j\}) \cup \{F_c\neg \Box A\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{F_c\neg \Box A\} \);

   (c) else, \( \mathcal{U}(H_j, i) = \{H_j\} \).

Moreover, \( \text{NewPar}(H_j, i) = \emptyset \).

6. If \( H_j \equiv \mathcal{F}(A \rightarrow B) \), then:
   
   (a) if \( (S' \setminus \{H_j\}) \cup \{TA, FB\} \) is consistent, then \( \mathcal{U}(H_j, i) = \{TA, FB\} \);

   (b) otherwise, \( \mathcal{U}(H_j, i) = \{FA, FB, F(A \rightarrow B)\} \);

Moreover, \( \text{NewPar}(H_j, i) = \emptyset \).
7. If $H_j \equiv \forall x. A(x)$, then
\[
\mathcal{U}(H_j, i) = \{ \top A(p_1), \ldots, \top A(p_r), \forall x. A(x) \}
\]
(where $\{p_1, \ldots, p_r\}$ are the parameters in $\Pi_i$), and $\text{NewPar}(H_j, i) = \emptyset$.

8. If $H_j \equiv \exists x. A(x)$, then
\[
\mathcal{U}(H_j, i) = \{ S_1 A(p_1), \ldots, S_r A(p_r), \exists x. A(x) \}
\]
be a set of swffs such that: $\{p_1, \ldots, p_r\}$ are all the parameters in $\Pi_i$, for any $i = 1, \ldots, r$ $S_i$ is either $\top$ or $\top$ and $S' \cup \mathcal{U}(H_j, i)$ is consistent. We remark that such a set $\mathcal{U}(H_j, i)$ must exist by the presence in $\mathcal{E}$-$\mathcal{T}$ of the rule $\top \exists$.

Moreover, $\text{NewPar}(H_j, i) = \emptyset$.

9. Let $H_j \equiv F_c \forall x. A(x)$, then
\[
\mathcal{U}(H_j, i) = \{ F_c A(q) \}
\]
where $q$ is a new parameter (i.e., an element of $\mathcal{K} \setminus \Pi_i$), and $\text{NewPar}(H_j, i) = \{q\}$.

10. Let $H_j \equiv \top \exists x. A(x)$, then
\[
\mathcal{U}(H_j, i) = \{ \top A(q) \}
\]
where $q$ is a new parameter, i.e., an element of $\mathcal{K} \setminus \Pi_i$, and $\text{NewPar}(H_j, i) = \{q\}$.

11. If $H_j \equiv \top \exists x. A(x)$, then let
\[
\mathcal{U}(H_j, i) = \{ S_1 A(p_1), \ldots, S_r A(p_r), \top \exists x. A(x) \}
\]
be a set of swffs such that: $\{p_1, \ldots, p_r\}$ are all the parameters in $\Pi_i$, for any $i = 1, \ldots, r$ $S_i$ is either $\top$ or $\top$ and $S' \cup \mathcal{U}(H_j, i)$ is consistent. We remark that such a set $\mathcal{U}(H_j, i)$ must exist by the presence in $\mathcal{E}$-$\mathcal{T}$ of the rule $\top \forall$.

Moreover, $\text{NewPar}(H_j, i) = \emptyset$.

12. If $H_j \equiv F_{c} \exists x. A(x)$, then
\[
\mathcal{U}(H_j, i) = \{ F_{c} A(p_1), \ldots, F_{c} A(p_r), F_{c} \exists x. A(x) \}
\]
(where $\{p_1, \ldots, p_r\}$ are the parameters in $\Pi_i$), and $\text{NewPar}(H_j, i) = \emptyset$.

13. If $H_j \equiv \top \forall x. A(x)$, then
\[
\mathcal{U}(H_j, i) = \{ \top A(p_1), \ldots, \top A(p_r), \top \forall x. A(x) \}
\]
(where $\{p_1, \ldots, p_r\}$ are the parameters in $\Pi_i$), and $\text{NewPar}(H_j, i) = \emptyset$. 

14. If $H_j \equiv \mathbf{F} \exists x. A(x)$, then
   \[
   \mathcal{U}(H_j, i) = \{\mathbf{F} \exists A(p_1), \ldots, \mathbf{F} \exists A(p_r), \mathbf{F} \exists x. A(x)\}
   \]
   (where \(\{p_1, \ldots, p_r\}\) are the parameters in \(\Pi_i\)), and \(\text{NewPar}(H_j, i) = \emptyset\).

15. If \(S = S_{\Box}\) and \(H_j\) is either \(\mathbf{F} \Box (A \land B)\) or \(\mathbf{T} \Box (A \lor B)\) or \(\mathbf{T} \Box (A \rightarrow B)\) then \(\mathcal{U}(H_j, i)\) is any extension \(\mathcal{R}_{H_j}\) of \(H_j\) such that \((S' \setminus \{H_j\}) \cup \mathcal{R}_{H_j}\) is consistent. In this case \(\text{NewPar}(H_j, i) = \emptyset\).

16. Let \(S = S_{\Box}\) and \(H_j \equiv \mathbf{F} \exists x. A(x)\), then
   \[
   \mathcal{U}(H_j, i) = \{\mathbf{F} \exists A(q)\}
   \]
   where \(q\) is a new parameter (i.e., an element of \(K \setminus \Pi \cup \Pi_i\)), and \(\text{NewPar}(H_j, i) = \{q\}\).

17. Let \(S = S_{\Box}\) and \(H_j \equiv \mathbf{T} \exists x. A(x)\), then
   \[
   \mathcal{U}(H_j, i) = \{\mathbf{T} \exists A(q)\}
   \]
   where \(q\) is a new parameter, i.e., an element of \(K \setminus \Pi \cup \Pi_i\), and \(\text{NewPar}(H_j, i) = \{q\}\).

18. In all the other cases, \(\mathcal{U}(H_j, i) = \{H_j\}\) and \(\text{NewPar}(H_j, i) = \emptyset\).

   It is easy to check, by induction on \(i\) (and by an accurate inspection of the rules of the calculus), that:
   1. for any \(i \geq 0\), the set \(S_i\) is consistent;
   2. for any \(i \geq 0\), the set \(\Pi_i\) is a reference set for \(S_i\) and \(\Pi \subseteq C \cup K\) and \(K \not\subseteq \Pi\).

Now, let us define
\[
S^* = \bigcup_{i \in \omega} S_i \quad \Pi_{S^*} = \bigcup_{i \in \omega} \Pi_i
\]

Under the hypothesis that \(S\) is a denumerable set of swffs, \(\Pi_{S^*}\) is clearly a denumerable set and it is a reference set for \(S^*\). We call \(S^*\) the saturated set of \(S\).

Now, we say that a swff \(H\) is final in a set \(S^*\) if \(H \in S^*\) and one of the following conditions holds:

1. No regular rule can be applied to \(H\);

2. \(H \equiv \mathbf{T} \forall x. A(x)\) or \(H \equiv \mathbf{F}_c \exists x. A(x)\) or \(\mathbf{T} \forall x. A(x) \rightarrow B\) (i.e., we consider the swffs, which give rise to duplications in the related rules of the calculus).

3. \(H \equiv \mathbf{T}(A \rightarrow B)\) with \(A\) atomic, and \(\mathbf{T}B \not\in S^*\) and \(\mathbf{F}_c A \not\in S^*\).

4. \(H \equiv \mathbf{T}(\Box A \rightarrow B)\), and \(\mathbf{T}B \not\in S^*\) and \(\mathbf{F}_c \Box A \not\in S^*\).
5. \( H \equiv T((A \rightarrow B) \rightarrow C) \), and \( T C \not\in S^* \) and \( \{TA, C B\} \not\in S^* \).
6. \( H \equiv T(\neg A \rightarrow B) \) with \( A \) atomic, and \( T B \not\in S^* \) and \( C B \neg A \not\in S^* \).
7. \( H \equiv T(\neg \square A \rightarrow B) \), and \( T B \not\in S^* \) and \( C B \neg \square A \not\in S^* \).
8. \( H \equiv F(A \rightarrow B) \), and \( \{TA, FB\} \not\in S^* \).
9. \( H \equiv F \forall x. A(x) \) and, for every \( a \in \Pi S^* \), \( F A(a) \not\in S^* \).
10. \( H \equiv F \exists x. A(x) \) and, for every \( a \in \Pi S^* \), \( F A(a) \not\in S^* \).

We call node set of \( S \) the set:
\[
\mathfrak{S} = \{ H \mid H \text{ is final in } S^* \}
\]
and we call \( S^* \) the saturated set related to \( \mathfrak{S} \) and we denote with \( \Pi \mathfrak{S} \) the set \( \Pi S^* \).

Since \( \mathfrak{S} \subseteq S^* \), we immediately have that \( \mathfrak{S} \) is consistent if \( S^* \) is consistent. Moreover, \( \Pi \mathfrak{S} \) is denumerable reference set for \( \mathfrak{S} \).

Given a non-empty node set \( \mathfrak{S} \), the c-successor sets \( \mathfrak{S} \) and their reference sets are defined as follows:

\( (c.1) \) If \( F(A \rightarrow B) \in \mathfrak{S} \), then:
(a) if \( U = \mathfrak{S} \cup \{TA, C B\} \) is consistent, then \( U \) is a c-successor of \( \mathfrak{S} \);
(b) else \( U = \mathfrak{S} \cup \{TA, FB\} \) is a c-successor of \( \mathfrak{S} \).

Moreover, \( \Pi U = \Pi \mathfrak{S} \).

\( (c.2) \) If \( F \forall x. A(x) \in \mathfrak{S} \), then:
(a) if \( U = \mathfrak{S} \cup \{F C A(q)\} \), where \( q \in K \setminus \Pi \mathfrak{S} \), is consistent, then \( U \) is a c-successor of \( \mathfrak{S} \);
(b) else \( U = \mathfrak{S} \cup \{FA(q)\} \), where \( q \in K \setminus \Pi \mathfrak{S} \), is a c-successor of \( \mathfrak{S} \).

Moreover, \( \Pi U = \Pi \mathfrak{S} \cup \{q\} \).

\( (c.3) \) If \( F \exists x. A(x) \in \mathfrak{S} \), then:
(a) if \( U = \mathfrak{S} \cup \{TA(q)\} \), where \( q \in K \setminus \Pi \mathfrak{S} \) is consistent, then \( U \) is a c-successor of \( \mathfrak{S} \);
(b) else \( U = \mathfrak{S} \cup \{FA(q)\} \), where \( q \in K \setminus \Pi \mathfrak{S} \), is a c-successor of \( \mathfrak{S} \).

Moreover, \( \Pi U = \Pi \mathfrak{S} \cup \{q\} \).

\( (c.4) \) If \( T((A \rightarrow B) \rightarrow C) \in \mathfrak{S} \), then:
(a) if \( U = \mathfrak{S} \cup \{TA, C B, T(B \rightarrow C)\} \) is consistent, then \( U \) is a c-successor of \( \mathfrak{S} \);
(b) else \( U = \overline{S}_c \cup \{TA, FB, T(B \rightarrow C)\} \) is a c-successor of \( \overline{S} \).

Moreover, \( \Pi_U = \Pi_{\overline{S}} \).

We remark that, if \( \overline{S} \) is consistent then at least one of the sets in points (a) and (b) of above points (c.1)-(c.4) is consistent by the rule of the calculus.

If \( U \) is a c-successor set of \( \overline{S} \) and \( H \) is the swff used to build \( U \) according to the previous definition, we say that \( U \) is the \emph{c-successor set of \( \overline{S} \) related to \( H \).}

Given a non-empty node set \( \overline{S} \), the cl-successor sets of \( \overline{S} \) and their reference sets are defined as follows:

(cl.1) If \( F \square A \in \overline{S} \), then \( U = \overline{S}_c \cup \{T \square A\} \) and \( U' = \overline{S}_c \cup \{F \square A\} \) are cl-successors of \( \overline{S} \). Moreover, \( \Pi_U = \Pi_U' = \Pi_{\overline{S}} \).

(cl.2) If no swff of the kind \( F \square A \) is in \( \overline{S} \) and \( \overline{S}_c \neq \overline{S} \) then \( U = \overline{S}_c \) is the only cl-successor of \( \overline{S} \) and \( \Pi_{\overline{S}_c} = \Pi_{\overline{S}} \).

We remark that, if \( \overline{S} \) is consistent then any cl-successor of \( \overline{S} \) is consistent; for point (cl.1) this comes by the presence in \( E \) of the rules \( F \square_1 \) and \( F \square_2 \), for point (cl.2) this comes from the special rule.

If \( U \) is a cl-successor set built up according to point (cl.1) and \( H \) is the swff \( F \square A \) used to build \( U \), we say that \( U \) is the \emph{cl-successor set of \( \overline{S} \) related to \( H \).}

Now, given a consistent set of swffs \( S \), we define the structure \( \mathfrak{S}(S) = \langle P, \leq, D, \ell \rangle \) as follows:

1. The node set \( \overline{S} \) of \( S \) belongs to \( P \), moreover \( D(\overline{S}) = \Pi_{\overline{S}} \).

2. For any \( \overline{U} \in P \), let \( U \) be any c-successor set or any cl-successor set of \( \overline{S} \) and let \( \Pi_U \) be its corresponding reference set. Let \( \overline{U} \) be a node set of \( U \) and let \( \Pi_{\overline{U}} \) be the corresponding reference set; then \( \overline{U} \in P \), \( \overline{U} \) is an immediate successor of \( \overline{S} \) and \( D(\overline{U}) = \Pi_{\overline{U}} \).

3. \( \leq \) is the transitive and reflexive closure of the immediate successor relation over the set \( P \).

4. For any \( \overline{U} \in P \) and any atomic closed wff \( A(\{c_1, \ldots, c_n\}) \) in the language \( L_{\overline{U}} \), we set:

   (a) if \( \overline{U} \neq \overline{U} \) then:

   \[
   \nu_{\overline{U}}(A(\{c_1, \ldots, c_n\})) = \begin{cases} 
   \top & \text{if } TA(\{c_1, \ldots, c_n\}) \in \overline{U} \\
   \bot & \text{if } FcA(\{c_1, \ldots, c_n\}) \in \overline{U} \\
   \uparrow & \text{otherwise}
   \end{cases}
   \]

   (b) if \( \overline{U} = \overline{U} \) then:

   \[
   \nu_{\overline{U}}(A(\{c_1, \ldots, c_n\})) = \begin{cases} 
   \top & \text{if } T \square A(\{c_1, \ldots, c_n\}) \in \overline{U} \\
   \bot & \text{if } otherwise
   \end{cases}
   \]
It is easy to check that $\mathcal{A}(S) = \langle P, \leq, D, \iota \rangle$ is an $\mathbf{E}$-model. In particular:

- The structural conditions are trivially satisfied by construction of $\mathcal{A}(S)$; in particular, any element of $P$ is followed, in the ordering $\leq$, by a final element. First of all we remark that the final elements of $\mathcal{A}(S)$ are the sets $\overline{\Psi}$ such that $\overline{\Psi} = \overline{\Psi}_d$. Now, by point (2) of the definition of $\mathcal{A}(S)$, every element $\overline{\Gamma}$ in $P$ such that $\overline{\Gamma} \neq \overline{\Gamma}_d$, is followed by a final element $\overline{\Psi}$ (corresponding to the node set of a cl-successor set of $\overline{\Gamma}$).

- The condition on preservation of falsity and truth is satisfied by the fact that if $\mathbf{F}_c A$ (respectively $\mathbf{T} A$) with $A$ atomic belongs to a node set $\overline{\Gamma}$ then it belongs to any c-successor set of $\overline{\Gamma}$ (these swffs are final in any saturated set) while $\mathbf{F}_d A$ (respectively $\mathbf{T}_d A$) belongs to every cl-successor set of $\overline{\Gamma}$.

- The condition on totality of $\iota$ on final states is trivially satisfied by the fact that any element of $P$ has a final element as immediate successor and by the definition of the interpretation function for the final elements.

The following lemma is the main step towards the proof of the completeness theorem:

**Lemma 8.5.14** Let $S$ be a consistent set of swffs and let $\mathcal{A}(S) = \langle P, \leq, D, \iota \rangle$ be one of the models defined above. Then, for any $\overline{\Gamma} \in P$ and any swff $H \in \Gamma^*$ (where $\Gamma^*$ is the saturated set related to $\overline{\Gamma}$) $\overline{\Gamma} \gg H$ in $\mathcal{A}(S)$.

**Proof:** The proof goes by induction on the well founded relation $\prec$.

**Basis:** For $\deg(H) = 0$ we have that $H \equiv S_p(c_1, \ldots, c_n)$ with $p(c_1, \ldots, c_n)$ a closed atomic swff of the language $L_{\Pi^p}$. Since in this case $H$ is final in $S^*$ we have that $H \in \overline{\Gamma}$. Now:

1. If $S \equiv T$ ($S = \mathbf{F}_c$ respectively), $H \in \overline{\Gamma}$ implies, by definition of $\iota$, that $\nu_{\overline{\Gamma}}(p(c_1, \ldots, c_n)) = \top$ ($\nu_{\overline{\Gamma}}(p(c_1, \ldots, c_n)) = \bot$ respectively), and hence $\overline{\Gamma} \gg H$.

2. If $S \equiv F, F_p(c_1, \ldots, c_n) \in \overline{\Gamma}$ then, by consistency of $\overline{\Gamma}$, we have that neither $\mathbf{T}_p(c_1, \ldots, c_n) \in \overline{\Gamma}$ nor $\mathbf{F}_c p(c_1, \ldots, c_n) \in \overline{\Gamma}$. Hence, by definition of $\iota$, $\nu_{\overline{\Gamma}}(p(c_1, \ldots, c_n)) = \bot$. Since $S \equiv T_d, \mathbf{F}_c p(c_1, \ldots, c_n) \in \overline{\Gamma}$, it is easy to check that $\mathbf{F}_d p(c_1, \ldots, c_n)$ belongs to every element $\overline{\Delta}$ such that $\overline{\Gamma} \leq \overline{\Delta}$. In particular, for every $\overline{\Psi} \in \text{Fin}(\overline{\Gamma}), \mathbf{F}_d p(c_1, \ldots, c_n) \in \overline{\Psi}$; henceforth, by definition of $\iota$, $\nu_{\overline{\Psi}}(p(c_1, \ldots, c_n)) = \bot$ for every $\overline{\Psi} \in \text{Fin}(\overline{\Gamma})$ and this means $\overline{\Gamma} \gg H$.

4. If $S \equiv \mathbf{F}_d$, it is easy to check that $\mathbf{F}_c p(c_1, \ldots, c_n)$ belongs to every element $\overline{\Delta}$ such that $\overline{\Gamma} \leq \overline{\Delta}$. In particular, for every $\overline{\Psi} \in \text{Fin}(\overline{\Gamma}), \mathbf{F}_d p(c_1, \ldots, c_n) \in \overline{\Psi}$. Since, any element of $\mathcal{A}(S)$ is consistent we have that no node set $\overline{\Psi} \in \text{Fin}(\overline{\Gamma})$ contains $\mathbf{T}_d p(c_1, \ldots, c_n)$ and thus $\nu_{\overline{\Psi}}(p(c_1, \ldots, c_n)) = \top$ for every $\overline{\Psi} \in \text{Fin}(\overline{\Gamma})$; thus $\overline{\Gamma} \gg H$. 


Step: Now, let us assume that the assertion holds for any swff $H' \in S^*$ such that $H' \prec H$. The proof goes by cases according to the form of the swff $H$. Here we give only some illustrative examples.

- $H \equiv F(A \rightarrow B)$: We have two cases:
  
  Case 1: $F(A \rightarrow B) \not\in \bar{T}$, then $F(A \rightarrow B)$ is not final in $\Gamma^*$, this means that $\{TA, FB\} \in \Gamma^*$, hence, by induction hypothesis $\bar{T} \triangleright TA$ and $\bar{T} \triangleright FB$ and this immediately implies $\bar{T} \triangleright F(A \rightarrow B)$.

  Case 2: $F(A \rightarrow B) \in \bar{T}$. In this case $F(A \rightarrow B)$ is final in $\Gamma^*$, then, by construction of $\mathcal{R}(S)$ there exists an immediate successor $\bar{\Delta}$ of $\bar{T}$ which is the node set of the c-successor set $\Delta$ of $\bar{T}$ related to $F(A \rightarrow B)$. This means that either $\{TA, FC\} \in \Delta^*$ or $\{TA, FB\} \in \Delta^*$. By induction hypothesis, this implies $\bar{\Delta} \triangleright \{TA, FC\}$ or $\bar{\Delta} \triangleright \{TA, FB\}$; in both cases we easily deduce $\bar{T} \triangleright F(A \rightarrow B)$.

- $H \equiv F \forall x. A(x)$: We have two cases:

  Case 1: $F \forall x. A(x) \not\in \bar{T}$, then $F \forall x. A(x)$ is not final in $\Gamma^*$, this means that there exists $a \in \Pi_{\bar{T}}$ such that $F(A(a)) \in \Gamma^*$, moreover, for every $c \in \Pi_{\bar{T}}$ either $F(A(a))$ or $TA(a)$ belongs to $\Gamma^*$ (by point (12) in the construction of saturated sets). Hence, we easily deduce that $\bar{T} \triangleright F \forall x. A(x)$. Case 2: $F \forall x. A(x) \in \bar{T}$. Since $F \forall x. A(x)$ is final in $\Gamma^*$, we have by construction of $\Gamma^*$ that $TA(a) \in \Gamma^*$ for every $a \in \Pi_{\bar{T}}$, hence by induction hypothesis we deduce that does not exists any $a \in \Pi_{\bar{T}}$ such that $\bar{T} \triangleright FcA(a)$. Moreover, since $F \forall x. A(x) \in \bar{T}$, by construction of $\mathcal{R}(S)$, there exists $\bar{\Delta} \in P$ such that $\bar{\Delta}$ is an immediate successor of $\bar{T}$ and $\bar{T}$ is the node set of a c-successor set of $\bar{T}$ related to $F \forall x. A(x)$. Thus $\bar{\Delta} \triangleright FcA(q)$ or $\bar{\Delta} \triangleright FA(q)$ for some $q \in \Pi_{\bar{T}}$. The above facts implies that $\bar{T} \triangleright F \forall x. A(x)$.

- $H \equiv T(A \rightarrow B)$ with $A$ atomic: Let $\bar{\Delta}$ be any element of $P$ such that $\bar{T} \triangleright \bar{\Delta}$. We have to cases:

  Case 1: If $T(A \rightarrow B) \not\in \bar{\Delta}$ then, there exists an element $\bar{\Phi} \in P$ either such that $\bar{T} \triangleright \bar{\Phi} \prec \bar{\Delta}$ and $T(A \rightarrow B) \in \Phi^*$ but $T(A \rightarrow B)$ is not final in $\Phi^*$ ($T(A \rightarrow B) \not\in \bar{\Phi}$). This implies that $FcA$ or $TB$ belongs to $\Phi^*$. In both cases, by induction hypothesis, we immediately get $\bar{\Delta} \triangleright FcA$ or $\bar{\Delta} \triangleright TB$.

  Case 2: If $T(A \rightarrow B) \in \bar{\Delta}$ then it is final in $\Delta^*$. This implies, by consistency of $\bar{\Delta}$ that neither $FcA$ nor $TA$ belongs to $\bar{\Delta}$, and this implies that $\bar{\Delta} \ntriangleright FA$.

  By the above analysis, we deduce that $\bar{T} \triangleright F(A \rightarrow B)$.

- $H \equiv F A$. In this case, by definition of the model, we have that there exists two immediate successors $\bar{\Delta}_1$ and $\bar{\Delta}_2$ of $\bar{T}$ (node sets of the two $c$-successor sets built up according to point (cl.1)) which are final elements of the model and such that $\bar{\Delta}_1 \triangleright T A$ and $\bar{\Delta}_2 \triangleright F A$.

\[\square\]

From the previous lemma we finally obtain, along the lines described in (Miglioli et al., 1994b; Miglioli et al., 1994c; Miglioli et al., 1997), the completeness theorem:

**Theorem 8.5.2 (Completeness of E-T)** If a wff $A$ is valid, then there exist closed proof-table starting from $FA$ and $F A$. 
Proof: Suppose the contrary; then either $S = \{ F A \}$ is consistent or $S = \{ F_n A \}$ is consistent. Let us consider an $\mathbf{E}$-model $\mathfrak{M}(S)$. By lemma 8.5.14 we get, in the first case, $\nu_{\mathfrak{M}}(A) = \uparrow$ and in the latter case $\nu_{\mathfrak{M}}(A) = \bot$. Both cases contradicts the fact that $A$ is valid.
Chapter 9

Conclusions

The main goal of this thesis was to design a Constructive Verification System. This software program should permit to develop correctness proofs for object code programs, using constructive techniques.

We described our idea of what should be the architecture of the whole system, and we developed the design of the new components.

The focus of our application of constructive techniques to formal verification has been in the development of an homogeneous environment: hence, we developed a system which permits to model the problem domain, to carry on correctness proofs in that domain, to analyze the obtained proofs, and, if needed, to synthesize programs from specifications.

The distinctive character of our approach is that all these tasks are performed using a small set of common concepts, of a constructive nature.

Most of the material in this thesis is not new: the theory behind the Constructive Reasoner is well-established, the Computer Arithmetic Toolkit is now available to the ISABELLE community, the Collection Method and the idea of Specification Frameworks were developed in the last twenty years by P. Miglioli and M. Ornaghi.

Despite this lack of new results in the foundations, our approach which combines all of the previous aspects in one single view, is a complete novelty. To reach this unified view, we had to develop many new results; for example, the whole treatment of the tableau calculus for the E system, developed in Chapters 3 and 8, is original. Moreover, some improvements have been done on the theory of specification frameworks, namely, an analysis of non standard induction principles and their relation with program schemas. The list of new results is longer, and it comprises the “validation by translation” technique (Chapter 3), the labeling algorithm (Chapter 6), the techniques to compress the logical representation of object code (Chapter 7).

Many applications, like the labelling algorithm or the translation of tableau proofs into natural deduction, were specifically developed to meet the purposes of the thesis; although their contribution does not increase the knowledge in the area of formal verification or theorem proving, they fit nicely in the framework we constructed.
There are many open points, we would like to work on in the future. First of all, we have a design, but not yet a system; we think that it is time to develop the Constructive Verification Environment, and to test it “on the field”.

There is space for many theoretical improvements, that will eventually generate new, more powerful tools for the design we proposed here.

In particular, the approach to datatype modeling, based on specification frameworks, and, consequently, the synthesis of programs from logical specifications, can be greatly improved by allowing implicit definitions. We mean that there is no need for a datatype specification to denote every element with a term; we have to develop new formal instruments to treat the relation between concrete datatypes, i.e., closed specification frameworks, and abstract ones, i.e., open specification frameworks.

Another point which needs improvements is the Collection Method. The techniques to prove that a logical system is uniformly constructive that we have shown in this thesis are nothing more than the most elementary ones; we already know, see (Ferrari, 1997b), other ways to instantiate the Collection Method that make it possible to prove that more complex logics are, indeed, uniformly constructive. But we still lack a second order formulation for the same method. We have got, in the period where we developed the thesis, some intermediate results, like (Benini, 1997; Benini, 1998a; Benini, 1999), but we still lack the final result. Such a result would open the door, in the Constructive Verification Environment, to the theory which supports analysis of higher-order proofs, in the same way we did for first-order ones.

About analysis, it is clear that the labelling algorithm is nice, being small and simple, but there is space for many others kind of automatic tools which can automatically analyze/improve the code of a program, basing their action on the information content of the correctness proof. In particular the ideas on orienting the extraction algorithm are good for the purposes of the thesis, but they require a deeper study.
Appendix A

A Full Example

A.1 Description

The goal of this appendix is to show, by means of a real correctness proof, a sketch of our proposal for an approach to formal verification of object code.

Our choice is `strcmp`, a standard routine in the `libc` library. We compiled it using `gcc` obtaining as an intermediate result an assembly version for the MC68000 microprocessor.

This code is optimized by the compiler and it shows almost every feature which makes difficult to verify object code in a formal way, but still its size is manageable.

The relevant features are:

- bitwise operations,
- non structured jumps,
- no abstract data types.

Our choice was driven by the size of the code (big examples are too long to be explained) and by the fact that every problem comes like in a nutshell.

A.2 Formalization

In this section we describe the representation and the specification for `strcmp`. As already stated in the introduction, we work on the object code as shown in Figure A.1.

The informal specification for `strcmp` says `given two non-null strings, it returns 0 if and only if they are equal`. In order to formalize this statement we need to disambiguate all the obscure points. First of all, every string is a pointer, so a non-null string means that the pointer is not equal to 0. The second point regards the word `given`: it means that we know where the pointers (strings) are stored.

To simplify our task, and to distinguish between a correctness proof for the subroutine, and a correctness proof of the calling procedure, we discard part of the
1 link a6,#0
2 movel d3,sp0-
3 movel d2,sp0-
4 movel a60(8),a1
5 movel a60(12),a0
6 moveq #0,d3
7 moveq #0,d2
L8: 8 movel a10+,d1
     9 movel a00+,d0
10 tstb d1
11 jeq L10
12 cmpb d1,d0
13 jeq L8
14 movel d1,d3
15 movel d0,d2
16 movel d3,d0
17 subl d2,d0
18 jra L13
L10: 19 andl #0xFF,d0
20 negl d0
L13: 21 movel a60(-8),d2
22 movel a60(-4),d3
23 unlk a6
24 rts

Figure A.1: Assembly Source for strcmp.
code, and precisely the \textit{envelope} which surrounds the real code, and takes care of retrieving the parameters and returning the exit value. In this way, the code we are going to analyze is shown in Figure A.2

\begin{verbatim}
  6 moveq #0, d3
  7 moveq #0, d2
  L8:  8 moveb a10+, d1
       9 moveb a00+, d0
       10 tstd d1
       11 jeq L10
       12 cmpb d1, d0
       13 jeq L8
       14 moveb d1, d3
       15 moveb d0, d2
       16 movel d3, d0
       17 sub1 d2, d0
       18 jra L13
  L10: 19 and1 #0xFF, d0
       20 neg1 d0
  L13: 21
\end{verbatim}

Figure A.2: The Source Code to Verify.

At this point it is possible to give a complete specification for this piece of code: given two strings, \texttt{argA} and \texttt{argB}, stored into registers \texttt{a0} and \texttt{a1}, respectively, both well-formed (i.e., non-null and zero-terminated), and the program counter at location 6 in the program, the program will terminate at a time \( t \) and, at that time, there is a value \( x \) such that the register \( d_0 \) contains zero if and only if the byte pointed by \texttt{argA} \( x \) is zero and the byte pointed by \texttt{argB} \( x \) is zero, and, moreover, up to \( x \), the strings are equal and every byte pointed by any value between \texttt{argB} and \texttt{argB} \( x \) is non-zero.

Introducing the following abbreviations (where \( \zeta \) is the function which calculates the length of a string)

\[
\alpha(x) = \text{ReadByte}_0 (\text{argA} + x) = \text{ReadByte}_0 (\text{argB} + x)
\]

\[
\beta(x) = (\forall y. 0 \leq y < x \rightarrow \alpha(y) \wedge \text{ReadByte}_0 (\text{argB} + y) \neq 0)
\]

\[
\phi(x) = (\text{ReadByte}_0 (x + \zeta(x)) = 0 \wedge
\forall y. 0 \leq y < \zeta(x) \rightarrow \text{ReadByte}_0 (x + y) \neq 0))
\]

the precondition becomes

\[
a_0(0) = \text{argA} \land a_1(0) = \text{argB} \land \phi(\text{argA}) \land \phi(\text{argB}) \land pc(0) = 6
\]

and the postcondition is

\[
\exists t, x. \beta(x) \land
(d_0(t) = 0) \leftrightarrow (\text{ReadByte}_0 (\text{argA} + x) = 0 \land \text{ReadByte}_0 (\text{argB} + x) = 0)
\]
In order to represent the program and to divide the correctness proof into a suitable number of lemmas, we partition the code into functional units. Every block is a sequential set of instructions which performs an elementary step in the computation. The result of this approach is shown in Figure A.3.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>moveq</td>
<td>#0, d3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>moveq</td>
<td>#0, d2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L8:</td>
<td>8</td>
<td>moveb</td>
<td>a10+, d1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>moveb</td>
<td>a0@+, d0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>tstb</td>
<td>d1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>jeq</td>
<td>L10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>cmpb</td>
<td>d1, d0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>jeq</td>
<td>L8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>moveb</td>
<td>d1, d3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>moveb</td>
<td>d0, d2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>move</td>
<td>d3, d0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>subl</td>
<td>d2, d0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>jra</td>
<td>L13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L10:</td>
<td>19</td>
<td>andl</td>
<td>#0xFF, d0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>negl</td>
<td>d0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L13:</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.3: Dividing the Source Code into Blocks.

To represent the program means to provide a logical formula which encodes the computation a single block performs. The general shape of a representation is

\[ \forall t. \text{pc}(t) = B \rightarrow \text{pc}(t + \Delta) = E \land \ldots \]

where \( B \) is the location of the first instruction, \( E \) is the location where the program will be after executing the block, \( \Delta \) is the amount of time (number of instructions) the code takes to execute, and the ellipsis stand for the final values of registers. In order to keep our representations short, that means, manageable, we represent just the relevant modifications to registers. The result of this representational effort is shown in Figure A.4.

Since every block is just a piece of code, we want to provide pre- and postconditions for every block, so to divide the complexity of the correctness proof among many lemmas. Essentially, this amounts to move the specification through blocks. Since we have a precondition and a postcondition for the whole program, we have two starting points for this process. By inspecting and analyzing the code we are able to reconstruct a reasonable amount of specifications for every block. Except for the loop between instruction 8 to 13, that we will discuss later, the complete picture is shown in Figure A.5. It contains every information we need in order to start the correctness proof.
<table>
<thead>
<tr>
<th>L8:</th>
<th>8 moveb a10+,d1</th>
<th>12 cmpb d1,d0</th>
<th>14 moveb d1,d3</th>
<th>19 andl #0xFF,d0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9 moveb a00+,d0</td>
<td>13 jeq L8</td>
<td>15 moveb d0,d2</td>
<td>20 negl d0</td>
</tr>
<tr>
<td></td>
<td>10 tstb d1</td>
<td>16 movel d3,d0</td>
<td>17 subl d2,d0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11 jeq L10</td>
<td>18 jra L13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| L10: | 19 andl #0xFF,d0 | 20 negl d0 |
|      |                  |            |
|      |                  |            |
|      |                  |            |

| L13: | 21 |
|      |    |

| 6 moveq #0,d3 | \forall t. pc(t) = 6 \rightarrow pc(t + 2) = 8 \land d_2(t + 2) = 0 \land d_3(t + 2) = 0 \land a_0(t + 2) = a_0(t) \land a_1(t + 2) = a_1(t) |
| 7 moveq #0,d2 | |

Figure A.4: Logical Representation of Blocks.
\[ a_0(0) = \text{arg A} \land a_1(0) = \text{arg B} \land \phi(\text{arg A}) \land \phi(\text{arg B}) \land \text{pc}(0) = 6 \]
\[ a_0(0) = \text{arg A} \land a_1(0) = \text{arg B} \land \phi(\text{arg A}) \land \phi(\text{arg B}) \land \text{pc}(0) = 6 \]

### L8:

<table>
<thead>
<tr>
<th>Move</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>moveq #0,d3</td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>moveq #0,d2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \forall t. ; \text{pc}(t)=6 \rightarrow \text{pc}(t+2)=8 \land d_2(t+2)=0 \land a_0(t+2)=a_0(t) \land a_1(t+2)=a_1(t) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>moveb a10+,d1</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td>moveb a00+,d0</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>tstb d1</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>jeq L10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \forall t. ; \text{pc}(t)=8 \rightarrow d_0(t+4) = \text{Read Byte}_0(a_0(t)) \land d_1(t+4) = \text{Read Byte}_0(a_1(t)) \land d_2(t+4) = d_2(t) \land d_3(t+4) = d_3(t) \land a_0(t+2) = a_0(t) \land a_1(t+4) = a_1(t) + 1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>cmpb d1,d0</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>jeq L8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \forall t. ; \text{pc}(t)=12 \rightarrow d_1(t) = d_0(t) \rightarrow \text{pc}(t+2) = 8 \land d_0(t+2) = d_0(t) \land d_4(t) = d_4(t+2) = d_3(t) \land a_0(t+2) = a_0(t) \land a_1(t+2) = a_1(t) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>moveb d1,d3</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>moveb d0,d2</td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>movel d3,d0</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>subl d2,d0</td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>jra L13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \forall t. ; \text{pc}(t)=14 \rightarrow d_0(t+5) = d_0(t) \land d_1(t+5) = d_1(t) \land a_0(t+5) = a_0(t) \land a_1(t+5) = a_1(t) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>andl #0xFF,d0</td>
<td></td>
</tr>
<tr>
<td>1.10</td>
<td>negl d0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \forall t. ; \text{pc}(t)=19 \rightarrow \text{pc}(t+2)=21 \land d_0(t+2) = d_0(t) \land d_1(t+2) = d_1(t) \land d_2(t+2) = d_2(t) \land d_3(t+2) = d_3(t) \land d_4(t+2) = d_4(t) \land d_5(t+2) = d_5(t) \land a_0(t+2) = a_0(t) \land a_1(t+2) = a_1(t) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.11</td>
<td>negl d0</td>
<td></td>
</tr>
</tbody>
</table>

### L10:

<table>
<thead>
<tr>
<th>Move</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.19</td>
<td>andl #0xFF,d0</td>
<td></td>
</tr>
<tr>
<td>0.20</td>
<td>negl d0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \forall t. ; \text{pc}(t)=19 \rightarrow \text{pc}(t+2)=21 \land d_0(t+2) = d_0(t) \land d_1(t+2) = d_1(t) \land d_2(t+2) = d_2(t) \land d_3(t+2) = d_3(t) \land d_4(t+2) = d_4(t) \land d_5(t+2) = d_5(t) \land a_0(t+2) = a_0(t) \land a_1(t+2) = a_1(t) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.21</td>
<td>jra L13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \forall t. ; \text{pc}(t)=21 )</td>
<td></td>
</tr>
</tbody>
</table>

---

Figure A.5: The Global Picture
This global picture makes easier to understand how the proof has been conceived, and what assertions hold in the code. But we have to say that Figure A.5 was written after the correctness proof was performed; in fact, it is the result we get after running for a few steps the labelling algorithm, see Chapter 6, on the correctness proof. Of course, it does not contain the whole set of things which happen to hold in the code because, we did not run the algorithm till the end, but we stopped after we got enough information. Then, obviously, we pruned the information we collected, retaining just what we thought to be useful to explain the arguments which lead to the correctness proof development.

A.3 Proving Correctness

As introduced in the previous section, the reason why we divide the program into blocks is to simplify the correctness proof, by introducing lemmas. Every such a lemma has the shape

\[
\text{Rep}_n, \text{Pre}_n \vdash \text{Post}_n
\]

where \(\text{Rep}_n\) is the representation of block number \(n\) and \(\text{Pre}_n, \text{Post}_n\) are its pre and postconditions, respectively.

We can roughly divide these lemmas into three categories:

- sequential arguments,
- logical arguments,
- loop arguments.

Every category involves a proper set of proving techniques, so the division is meaningful.

A sequential argument is a statement where the precondition is transformed into the postcondition applying the transformation encoded into the block representation. The proving technique in this case is to remove quantifiers by instantiating the body of the goal, and to simplify the result sometimes performing arithmetical calculations. In Figure A.6, A.7 and A.8 are shown the ISABELLE proofs for blocks 6-7, 14-18 and 19-20, respectively, which are sequential blocks.

A logical argument is a statement where the precondition implies the postcondition, not involving any program transformation. There is no general rule on how to prove such a statement, but a good heuristic is to reduce the goal to a propositional formulation and then to use decision procedures. In the proof of `strcmp`, block 21, see Figure A.9, is of this kind.

A block which ends with a conditional branch could be regarded, for the proving technique, as the sum of two sequential blocks, one assuming the branching condition, the other assuming its negation, and as a logical block, in the way it combines the postconditions of its sequential counterparts.
val prems = goal strctmp.thy "[| Rep6_7; Pre6 |] ==> Post6";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac
  (!simpset addsimps [Rep6_7_def, Pre6_def, Post6_def]) 1);
by (eres_inst_tac ["x","#0""] allE 1);
by (extbin_tac 1);
val Proof1 = result();

Figure A.6: Proof Script for Block 6-7.

val prems = goal strctmp.thy "[| Rep14_18; Pre14 |] ==> Post14";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac
  (!simpset addsimps [Rep14_18_def, Pre14_def, Post14_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
val Proof3 = result();

Figure A.7: Proof Script for Block 14-18.

val prems = goal strctmp.thy
  "[| Rep19_20; Pre19 |] ==> Post19";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac
  (!simpset addsimps [Rep19_20_def, Pre19_def,
                      Post19_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by ((asmnorm_tac THEN'
     (asm_full_simp_tac (!simpset addsimps [ArithLemma1]))) 1);
val Proof4 = result();

Figure A.8: Proof Script for Block 19-20.
val prems = goal_strcmp.thy "Pre21 ==> Post21";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Pre21_def, Post21_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac [("x","t")]) exI 1);
by (res_inst_tac [("x","x")]) exI 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
by (etac disjE 1);
by (rtac iffI 1);
by (rtac FalseE 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
by (etac notE 1);
by (subgoal_tac "?y - ?x = #0 --> ?y = ?x" 1);
by (supinf_tac 2);
by (REPEAT (eresolve_tac [mp,sym] 1));
by (((rotate_tac ~1) THEN' Asm_full_simp_tac) 1);
by (REPEAT (eresolve_tac [mp,sym] 1));
by (Asm_full_simp_tac 1);
by (rtac iffI 1);
by (((rotate_tac ~1) THEN' Asm_full_simp_tac) 1);
by (subgoal_tac "$\neg$ $\neg$ ?x = #0 --> ?x = #0" 1);
by (supinf_tac 2);
by (etac mp 1);
by ((asmnorm_tac THEN'
    (eres_inst_tac [("P","Q. $\neg$ Q = #0")]) subst)) 1);
by (extbin_tac 1);
by (((rotate_tac ~1) THEN' Asm_full_simp_tac) 1);
val Proof5 = result ();

Figure A.9: Proof Script for Block 21.
In the code of `strcmp` we have a loop (instructions 8 to 13); as usual in object code, the loop is not structured, i.e., it has not one entry point and one exit point, but some conditional branches control the way to exit or to continue the computation.

To reason about a loop involves two different steps:
- proving the correctness of one cycle;
- proving its termination.

The technique to cope with the first goal is not different from a normal branching block. In our case, with the aid of some additional abbreviations, it reduces to Figure A.10.

In order to prove that every unfolding of the cycle is correct, and in order to ensure that no infinite unfolding is possible, we must use induction. The exact form of the induction principle is suggested by the loop structure.

In the case of `strcmp`, we use the Bounded Chain Principle:

\[
[p \leq b], [P(p)] \\
\vdots \\
\exists x. x \leq b \land P(x) \land (\exists y. p < y \leq b \land P(y)) \\
B
\]

The premises say that, given a bound \( b \) and a set \( P \) of integers, there is a point less or equal than \( b \) in \( P \), and whenever this situation holds, either it is possible to find another point \( y \) which replicate the situation, or \( B \) holds.

Since integer numbers provide a discrete ordering, sooner or later, we will exhaust the interval \([x, b]\), so, eventually, \( B \) must hold.

Reading it as a loop, the principle says that, if we can prove that the loop is entered and, for every step, it either exits with postcondition \( B \) or it loops again, but decreasing a complexity measure given by the distance of \( y \) from \( b \), then, eventually, it will exit with the postcondition \( B \) true.

In our case, \( B \) is simply the postcondition of the cycle, as shown in Figure A.5; \( P(x) \) is \( \exists t. 0 \leq x \land \delta(t, x) \), i.e., there is a time such that the loop will reach position \( x \) in the string \( \text{argB} \); and the bound is simply \( \zeta(\text{argB}) \), the length of string \( \text{argB} \).

The reason for this bound is simple: if we will scan position \( \zeta(\text{argB}) \), it means that \( \text{argB} \) is shorter than \( \text{argA} \) and the loop will exit, see instructions 10–11 in the source code.

Following standard proving techniques for sequential and logical arguments, the correctness of the loop is established, the proof script is shown in Figure A.11.

With these lemmas is easy to prove the correctness for the whole `strcmp` algorithm, as done in Figure A.12; the proof combines lemmas following the flow of control of the program.

### A.4 Concluding Remarks

In this technical appendix we have shown that the standard library function `strcmp` is correct. We gave a formal proof of this fact using Isabelle/HOL and the Computer
\[
\delta(x,t) \equiv (a_0(t) = \text{arg}A + x + 1 \land a_1(t) = \text{arg}B + x + 1 \land \\
d_0(t) = \text{ReadByte}_0(\text{arg}A + x) \land d_1(t) = \text{ReadByte}_0(\text{arg}B + x) \land \\
d_2(t) = 0 \land d_3(t) = 0 \land \beta(x) \land \\
(pc(t) = 8 \lor pc(t) = 14 \lor pc(t) = 19) \land \\
(pc(t) = 8 \rightarrow \text{ReadByte}_0(\text{arg}B + x) \neq 0) \land \\
\text{ReadByte}_0(\text{arg}A + x) = \text{ReadByte}_0(\text{arg}B + x) \land \\
(pc(t) = 14 \rightarrow \text{ReadByte}_0(\text{arg}B + x) \neq 0) \land \\
\text{ReadByte}_0(\text{arg}A + x) \neq \text{ReadByte}_0(\text{arg}B + x)) \land \\
(pc(t) = 19 \Rightarrow \text{ReadByte}_0(\text{arg}B + x) = 0))
\]

\[
\gamma(x) \equiv (\exists t. \delta(x,t))
\]

val prems = goal strcmp.tthy
    "[| Rep8_11; Rep12_13; beta x; pc t = #8; #0 <= x; \\
\a0 t = \text{arg}A + x; a1 t = \text{arg}B + x; d2 t = #0; d3 t = #0 |] ==> \\
\gamma \text{ gamma } x";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Rep8_11_def, Rep12_13_def, 
                                      gamma_def, delta_def]) 1);
by (((eres_inst_tac ["x","t"] allE) THEN'
    (etac (make_elim mp)) THEN' atac THEN' asmnorm_tac) 1);
by (case_tac "\text{ReadByte} (a1 t) #0 = #0" 1);
by (res_inst_tac ["x","t + #4"] exI 1);
by (Asm_full_simp_tac 1);
by (((eres_inst_tac ["x","t + #4"] allE) THEN'
    extbin_tac THEN' Asm_full_simp_tac) 1);
by (((etac (make_elim mp)) THEN' atac THEN' asmnorm_tac) 1);
by (res_inst_tac ["x","t + #6"] exI 1);
by (((Asm_full_simp_tac THEN' extbin_tac THEN' Asm_full_simp_tac) 1);
by (case_tac "\text{ReadByte} (x + \text{arg}B) #0 = \text{ReadByte} (x + \text{arg}A) #0" 1);
by (ALLGOALS ((rotate_tac ~1) THEN'
    Asm_full_simp_tac THEN'
    (TRY o (etac not_sym))));
val Proof2_1 = result ();

Figure A.10: Proof Script which Shows that the Body of the Loop is Correct.
val prems = goal_strm thy "[Rep8_11; Rep12_13; Pre81] ==> Post8";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Rep8_11_def, Rep12_13_def, 
    Pre8_def, Post8_def, Post6_def]) 1) THEN (asmnorm_tac 1));
by (res_inst_tac ["P","\% x. (? t. #0 <= x & delta x t)"],
    ("b","zeta argB") BCP 1);
by (res_inst_tac ["x","#0"] exI 1);
by ((REPEAT o (resolve_tac [conjI,zeta_pos,Proof2_2]))
    THEN' supinf_tac 1);
by ((fold_goals_tac [gamma_def]) THEN (rtac Proof2_1 1));
by (TRYALL (asm_full_simp_tac
    (!simpset addsimps [Rep8_11_def, Rep12_13_def, beta_def]))); 
by (TRYALL supinf_tac);
by (asm_full_simp_tac (!simpset addsimps [delta_def]) 1);
by ((asmnorm_tac THEN' (etac exE) THEN' asmnorm_tac) 1);
by (etac disjE 1);
by (((rotate_tac ~1) THEN' Asm_full_simp_tac) 1);
by (rtac disjI2 1); by (res_inst_tac ["x","p + #1"] exI 1);
by (((rtac conjI) THEN' supinf_tac THEN' (rtac conjI)) 1);
by (subgoal_tac
    "(p <= zeta argB) = (p < zeta argB | p = zeta argB)" 1);
by (Asm_full_simp_tac 1);
by (etac disjE 1); by (supinf_oneasm_tac ~1 1);
by (((rotate_tac ~1) THEN' 
    (asm_full_simp_tac (!simpset addsimps [phi_def])))) 1);
by (supinf_tac 1);
by (((rtac conjI) THEN' (supinf_oneasm_tac ~11)) 1);
by (fold_goals_tac [gamma_def,delta_def]);
by (((rotate_tac ~11) THEN' (rtac Proof2_1)) 1);
by (TRYALL (asm_full_simp_tac 
    (!simpset addsimps [Rep8_11_def, Rep12_13_def, beta_def])));
by (TRYALL supinf_tac); by ((strip_tac THEN' asmnorm_tac) 1);
by (subgoal_tac "(y < p + #1) = (y < p | y = p)" 1);
by (Asm_full_simp_tac 1); by (((etac disjE) THEN' Fast_tac) 1);
by (((rotate_tac ~1) THEN'
    (asm_full_simp_tac (!simpset addsimps [alpha_def])))) 1);
by (supinf_tac 1); by (supinf_oneasm_tac 0 1);
by (rtac disjI1 1);
by ((res_inst_tac ["x","t"] exI 1) THEN 
    (res_inst_tac ["x","p"] exI 1) THEN (Asm_simp_tac 1)); 
val Proof2 = result();

Figure A.11: Proof Script for Induction in the Loop.
val prems = goal strcmp.thy
  "[ ] Rep6_7; Rep8_11; Rep12_13;
  \    Rep14_18; Rep19_20; Pre [] == Spec"
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Pre_def, Spec_def]) 1);
by (rtac Proof5 1);
by (asm_full_simp_tac (!simpset addsimps [Pre21_def]) 1);
by (subgoal_tac "Pre14 | Pre19" 1);
by (etac disjE 1);
by (((etac (make_elim Proof3)) THEN' atac) 1);
by (asm_full_simp_tac (!simpset addsimps [Post14_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x","t + #5"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by (Fast_tac 1);
by (((etac (make_elim Proof4)) THEN' atac) 1);
by (asm_full_simp_tac (!simpset addsimps [Post19_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x","t + #2"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by (Fast_tac 1);
by (((etac (make_elim Proof2)) THEN' atac) 1);
by (asm_full_simp_tac (!simpset addsimps [Pre8_def]) 1);
by (((etac Proof1) THEN' atac) 1);
by (asm_full_simp_tac
     (!simpset addsimps [Post8_def, Pre14_def, Pre19_def]) 1);
by ((REPEAT (etac exE 1)) THEN (asmnorm_tac 1));
by (etac disjE 1);
by (rtac disjI1 1);
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by (Asm_full_simp_tac 1);
by (rtac disjI2 1);
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by (Asm_full_simp_tac 1);
val strcmp_correctness_proof = result();

Figure A.12: Correctness Proof for strcmp.
Arithmetic Toolkit as a workbench. We used most of the ideas included in the thesis, although on a very preliminary version of the traditional designed Constructive Verification Environment.

The key ideas behind the proof itself are general and it is worth spending few words to summarize them:

- dividing the proof into lemmas suggested by sequential pieces of code makes it possible to reduce complexity of the proof and isolate trivial parts from difficult ones (i.e., sequential and logical arguments from loops).

- to cope with loops we need an induction principle which resembles the structure of the loop itself. Most of the times this reduces to choose a variation of what is known as Noetherian Induction.

- the formal verification of object code requires a good treatment of computer arithmetic, since this is the only way a CPU has in order to perform calculations.
A.5 Source Files

The goal of this section is just to show the theory files we wrote to prove the `strcmp` example. They were, in part, automatically generated, and, then edited by hand. In the designed Constructive Verification System, this test case should not need any editing in the theory files, and the proofs should reduce themselves by a factor three.

A.5.1 `strcmp.thy`

```
strcmp = ExtBin + strcmpdef + ArithSupp +

consts (* Representation of blocks *)
  Rep19_20 :: bool
  Rep14_18 :: bool
  Rep12_13 :: bool
  Rep8_11 :: bool
  Rep6_7 :: bool

defs
  Rep19_20_def "Rep19_20 == ! t. pc t = #19 --> \\
  \ pc (t + #2) = #21 & \\
  \ a0 (t + #2) = a0 t & \\
  \ a1 (t + #2) = a1 t & \\
  \ d1 (t + #2) = d1 t & \\
  \ d0 (t + #2) = $" (d0 t zand #255)"
  Rep14_18_def "Rep14_18 == ! t. pc t = #14 --> \\
  \ pc (t + #5) = #21 & \\
  \ a0 (t + #5) = a0 t & \\
  \ a1 (t + #5) = a1 t & \\
  \ d1 (t + #5) = d1 t & \\
  \ d0 (t + #5) = d0 t - d1 t"
  Rep12_13_def "Rep12_13 == ! t. pc t = #12 --> \\
  \ ((d1 t = d0 t) --> (pc (t + #2) = #8)) & \\
  \ ((d1 t = d0 t) --> (pc (t + #2) = #14)) & \\
  \ a0 (t + #2) = a0 t & \\
  \ a1 (t + #2) = a1 t & \\
  \ d3 (t + #2) = d3 t & \\
  \ d2 (t + #2) = d2 t & \\
  \ d0 (t + #2) = d0 t & \\
  \ d1 (t + #2) = d1 t"
  Rep8_11_def "Rep8_11 == ! t. pc t = #8 --> \\
  \ ((ReadByte (a1 t) #0 = #0) --> \\
  \ (pc (t + #4) = #19)) & \\
  \ ((ReadByte (a1 t) #0 = #0) --> \\
```
\( \text{(pc (t + #4) = #12)) &} \\
\( \text{d0 (t + #4) = } \text{ReadByte} \ (a0 \ t) \ #0 \ &} \\
\( \text{d1 (t + #4) = } \text{ReadByte} \ (a1 \ t) \ #0 \ &} \\
\( \text{d3 (t + #4) = d3 t} \ &} \\
\( \text{d2 (t + #4) = d2 t} \ &} \\
\( \text{a0 (t + #4) = a0 t + #1} \ &} \\
\( \text{a1 (t + #4) = a1 t + #1} \ &} \\
\text{Rep6_7_def "Rep6_7 == ! t. pc t = #6 -->} \\
\text{pc (t + #2) = #8 &} \\
\text{d3 (t + #2) = #0 &} \\
\text{d2 (t + #2) = #0 &} \\
\text{a0 (t + #2) = a0 t &} \\
\text{a1 (t + #2) = a1 t"} \\
\text{consts (* Specifications for every block *)} \\
\text{Pre6 :: bool} \\
\text{Post6 :: bool} \\
\text{Pre8 :: bool} \\
\text{Post8 :: bool} \\
\text{Pre14 :: bool} \\
\text{Post14 :: bool} \\
\text{Pre19 :: bool} \\
\text{Post19 :: bool} \\
\text{Pre21 :: bool} \\
\text{Post21 :: bool} \\
\text{defs Post21_def "Post21 == (? t x. beta x &} \\
\text{\( (d0 t = #0)) = \} \\
\text{\( (ReadByte (argA + x) #0 = #0 &} \\
\text{\( \text{ReadByte (argB + x) #0 = #0})\"} \\
\text{Pre21_def "Pre21 == (? t x. beta x &} \\
\text{((d0 t =} \\
\text{\( \text{ReadByte (argA + x) #0} - \} \\
\text{\( \text{ReadByte (argB + x) #0} &} \\
\text{\( \text{ReadByte (argA + x) #0} = \} \\
\text{\( \text{ReadByte (argB + x) #0) |} \} \\
\text{\( \text{d0 t =} \\
\text{\$ ~ \text{ReadByte (argA + x) #0} &} \\
\text{\( \text{ReadByte (argB + x) #0 = #0}) &} \\
\text{pc t = #21)"} \\
\text{Post19_def "Post19 == (? t x. d0 (t + #2) =} \\
\text{\$ ~ \text{ReadByte (argA + x) #0} &} \\
\text{\( \text{ReadByte (argB + x) #0 = #0} \) &} \\
\text{pc t = #21)"} \\
\text{Post19_def "Post19 == (? t x. d0 (t + #2) =} \\
\text{\$ ~ \text{ReadByte (argA + x) #0} &} \\
\text{\( \text{ReadByte (argB + x) #0 = #0} \) &} \)
\betax & \pc(t + \#2) = \#21"
\Pre19_def "Pre19 == (? t x. a0 t = \argA + x + \#1 & \
a1 t = \argB + x + \#1 & \\
d0 t = \text{ReadByte} (\argA + x) \#0 & \\
d1 t = \text{ReadByte} (\argB + x) \#0 & \\
\betax & \\
d1 t = \#0 & \\
\pc t = \#19)"
\Post14_def "Post14 == (? t x. d0 t = (t + \#5) = \\
\text{ReadByte} (\argA + x) \#0 - \\
\text{ReadByte} (\argB + x) \#0 & \\
\text{ReadByte} (\argA + x) \#0 = \\
\text{ReadByte} (\argB + x) \#0 & \\
\betax & \\
d2 t = \#0 & \\
d3 t = \#0 & \\
d1 t = \#0 & \\
d0 t = d1 t & \\
\pc t = \#14)"
\Post8_def "Post8 == (? t x. a0 t = \argA + x + \#1 & \\
a1 t = \argB + x + \#1 & \\
d0 t = \text{ReadByte} (\argA + x) \#0 & \\
d1 t = \text{ReadByte} (\argB + x) \#0 & \\
(pc t = \#14 | \pc t = \#19) & \\
\betax & \\
d2 t = \#0 & \\
d3 t = \#0 & \\
(pc t = \#14 --> d1 t = \#0 & \\
d0 t = d1 t) & \\
\pc t = \#19 --> d1 t = \#0))"
\Pre8_def "Pre8 == Post6"
\Post6_def "Post6 == (\phi \argA & \\
\phi \argB & \\
a0 \#2 = \argA & \\
a1 \#2 = \argB & \\
d2 \#2 = \#0 &\"
\ 
\ \ Pre6_def  "Pre6 == (a0 #0 = argA & \ 
\ a1 #0 = argB & \ 
\ phi argA & \ 
\ phi argB & \ 
\ pc #0 = #6)"

consts  (* Specification of the whole algorithm *)
  Spec :: bool
  Pre  :: bool

defs
  Spec_def  "Spec == Post21"
  Pre_def   "Pre == Pre6"

end
A.5.2 strcmpdef.thy

```haskell
strcmpdef = ExtBin +

consts
  mm :: [int, int] => int

  sp :: int => int
  pc :: int => int
  d0 :: int => int
  d1 :: int => int
  d2 :: int => int
  d3 :: int => int
  d4 :: int => int
  d5 :: int => int
  d6 :: int => int
  d7 :: int => int
  a0 :: int => int
  a1 :: int => int
  a2 :: int => int
  a3 :: int => int
  a4 :: int => int
  a5 :: int => int
  a6 :: int => int
  a7 :: int => int

consts
  ReadByte :: [int, int] => int
  ReadWord :: [int, int] => int
  ReadLong :: [int, int] => int
  WriteByte :: [int, int, int] => bool
  WriteWord :: [int, int, int] => bool
  WriteLong :: [int, int, int] => bool

defs
  ReadByte_def  "ReadByte p t == mm p t"
  ReadWord_def  "ReadWord p t == \n    ReadByte p t + #256 * ReadByte (p + #1) t"
  ReadLong_def  "ReadLong p t == \n    ReadWord p t + #65536 * ReadWord (p + #2) t"
  WriteByte_def "WriteByte p t v == \n    v < #256 & ReadByte p t = v"```
WriteWord_def "WriteWord p t v == \
  v < #65536 & ReadWord p t = v"
WriteLong_def "WriteLong p t v == ReadLong p t = v"

c consts
  argA :: int
  argB :: int
  alpha :: int => bool
  beta :: int => bool
  gamma :: int => bool
  delta :: int => int => bool
  phi :: int => bool
  zeta :: int => int

c defs
  alpha_def "alpha x == (ReadByte (argA + x) #0 = \
    ReadByte (argB + x) #0)"
  beta_def "beta x == (! y. #0 <= y & y < x --> alpha y & 
    ReadByte (argB + y) #0 ~= #0)"
  gamma_def "gamma x == (? t. delta x t)"
  delta_def "delta x t == (a0 t = argA + x + #1 & \
    a1 t = argB + x + #1 & \
    d0 t = ReadByte (argA + x) #0 & \
    d1 t = ReadByte (argB + x) #0 & \
    d2 t = #0 & \
    d3 t = #0 & \
    beta x & \
    (pc t = #8 | \
     pc t = #14 | \
     pc t = #19) & \
    (pc t = #8 -->) \
    ReadByte (argB + x) #0 ~= #0 & \
    ReadByte (argA + x) #0 = \
    ReadByte (argB + x) #0) & \
    (pc t = #14 -->) \
    ReadByte (argB + x) #0 ~= #0 & \
    ReadByte (argA + x) #0 = \
    ReadByte (argB + x) #0) & \
    (pc t = #19 -->) \
    ReadByte (argB + x) #0 = #0)"
  phi_def "phi z == (ReadByte (z + zeta z) #0 = #0 & \
    (! y. #0 <= y & y < zeta z -->) \
    ReadByte (z + y) #0 ~= #0)"
rules
    zeta_pos "#0 <= zeta x"

    BCP "[| ? x. (x <= b & P x); \ \
    \ !! p. [| p <= b; P p |] ==> \ \
    \ B | (? y. p < y & y <= b & P y)] ==> B"

end

A.5.3 ArithSupp.thy

ArithSupp = strcmpdef +

rules
    ArithLemma1 "ReadByte x y zand #255 = ReadByte x y"

end
Bibliography


205
tableau systems for intermediate propositional logics. In Galmiche, D., edi-
tor, Proceedings of the 6th International Conference on Automated Reasoning
with Analytic Tableaux and Related Methods, volume 1227 of Lecture Notes in
Artificial Intelligence, pages 43–61. Springer Verlag.

103 of Studies in Logic and the Foundations of Mathematics. North Holland,
2nd edition.


CSLI Publications.

Analytic Tableaux and Related Methods: 4th International Workshop, Schloss
Rheinfeis, St. Goar, Germany, volume 918 of Lecture Notes in Artificial Intel-
ligence. Springer Verlag.

Studies. Springer Verlag.

Proceedings of the First Isabelle Users Workshop. Computer Laboratory, Cam-
bridge University.

report, SGS Thomson.

Paper.

given in the Conference Logic Colloquium '98.


Benini, M. (1999). The collection method in second-order intuitionistic logic. Tech-
nical Report 237-99, Dipartimento di Scienze dell'Informazione, Università degli
Studi di Milano.

the description and implementation of automated theorem provers. Technical
report, Computational Architectures Laboratory, Dipartimento di Scienze
dell’Informazione, Università degli Studi di Milano.


# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Formal Verification and Development of Computer Programs</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Standard Architecture of a Verification System</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>An Architecture Derived From the Standard</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>A Purely Constructive Architecture</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>A Constructive Architecture with the Admissibility Check</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Derived Inference Rules for Translating Tableaus into IL: Part I</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Derived Inference Rules for Translating Tableaus into IL: Part II</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Derived Inference Rules for Translating Tableaus into IL: Part III</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Derived Inference Rules for Translating Tableaus into IL: Part IV</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Derived Inference Rules for Translating Tableaus into IL: Part V</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Derived Inference Rules for Translating Tableaus into IL: Part VI</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>Derived Inference Rules for Translating Tableaus into E: Part I</td>
<td>45</td>
</tr>
<tr>
<td>3.8</td>
<td>Derived Inference Rules for Translating Tableaus into E: Part II</td>
<td>46</td>
</tr>
<tr>
<td>3.9</td>
<td>Derived Inference Rules for Translating Tableaus into E: Part III</td>
<td>46</td>
</tr>
<tr>
<td>3.10</td>
<td>Derived Inference Rules for Translating Tableaus into E: Part IV</td>
<td>47</td>
</tr>
<tr>
<td>3.11</td>
<td>Derived Inference Rules for Translating Tableaus into E: Part V</td>
<td>48</td>
</tr>
<tr>
<td>3.12</td>
<td>Derived Inference Rules for Translating Tableaus into E: Part VI</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Modular Numbers, an Example of Concrete Representations.</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Operations and Ordering Relations on Natural Numbers</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>Operations and Ordering Relations on Integer Numbers</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>Definitions for Bitwise Operations on Integers</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>The Syntax of the Calculational Engine Language.</td>
<td>78</td>
</tr>
<tr>
<td>4.6</td>
<td>An Example of Rewrite Function</td>
<td>79</td>
</tr>
<tr>
<td>4.7</td>
<td>A Reduction Mapping</td>
<td>81</td>
</tr>
<tr>
<td>4.8</td>
<td>An Associative Reduction Mapping</td>
<td>81</td>
</tr>
<tr>
<td>4.9</td>
<td>A Commutative Reduction Mapping</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>The Closed Framework PA, for Peano Arithmetic.</td>
<td>94</td>
</tr>
<tr>
<td>5.2</td>
<td>The Open Framework for Lists</td>
<td>97</td>
</tr>
<tr>
<td>6.1</td>
<td>An Example Program for Computing Division by Two</td>
<td>122</td>
</tr>
<tr>
<td>6.2</td>
<td>The Logical Representation for the Example Program</td>
<td>123</td>
</tr>
</tbody>
</table>

217
6.3 The Schema of the Correctness Proof for Our Example. . . . . . 123
6.4 The Result of the Labeling Algorithm on Our Example Program. . 124

A.1 Assembly Source for `strcmp`. ............................... 184
A.2 The Source Code to Verify. ............................... 185
A.3 Dividing the Source Code into Blocks. ......................... 186
A.4 Logical Representation of Blocks. ............................ 187
A.5 The Global Picture ............................... 188
A.6 Proof Script for Block 6-7. ............................... 190
A.7 Proof Script for Block 14-18. ............................... 190
A.8 Proof Script for Block 19-20. ............................... 190
A.9 Proof Script for Block 21. ............................... 191
A.10 Proof Script which Shows that the Body of the Loop is Correct. . 193
A.11 Proof Script for Induction in the Loop. ...................... 194
A.12 Correctness Proof for `strcmp`. ............................. 195
List of Tables

2.1 Architectural Requirements. ........................................ 16
3.1 Inference Rules for the First-Order Intuitionistic Logic IL. .... 33
3.2 Inference Rules for the E Logic. ................................ 34
3.3 Expansion Rules for the IL-T Tableau Calculus. ............... 35
3.4 Expansion Rules for the E-T Tableau Calculus: Propositional Part. 36
3.5 Expansion Rules for the E-T Tableau Calculus: Predicative Part. . 37
3.6 Expansion Rules for the E-T Tableau Calculus: Classical Part. . . 38
3.7 Expansion Rules for the E-T Tableau Calculus: Implicational Part. . 38

8.1 The Calculus E-T. ................................................. 161
8.2 Rules for T→. ................................................... 162
8.3 Rules for T□ and F□. ............................................. 163
Index

Symbols
\[\square\text{-theory} \quad 93, 114, 154\]

A
Admissibility Checker \quad 26
architectural requirement \quad 16
architecture \quad 17

C
c-successor set \quad 175
c-successor set,
\textit{see} completeness theorem
c-swff,  
\textit{see} completeness theorem
Calculational Engine \quad 78


D
decision procedure \quad 82
degree of a swff \quad 170
degree of a swff,  
\textit{see} completeness theorem
degree of a wff \quad 169
degree of a wff,  
\textit{see} completeness theorem
Descending Chain Principle \quad 101, 118, 156
dischargeable set \quad 100
Divide et Impera \quad 102

E
E-proof-table,  
\textit{see} tableau
evaluation \quad 137, 142, 148
evaluation form \quad 98
evaluation variable \quad 98
evaluation variable,  
\textit{see} specification

F
F, \quad \textit{see} tableau
F_\text{D}, \quad \textit{see} tableau
F_\text{C}, \quad \textit{see} tableau
formal specification \quad 13
framework
\textit{closed} \quad 89

extension \quad 92
intended model \quad 89, 95
open \quad 95
G
generic theorem prover,
   see interactive theorem prover

H
HOL ................................. 6, 11

I
IL-proof-table,
   see tableau
implicative complexity .......... 170
implicative complexity,
   see completeness theorem
individual variable ............... 98
individual variable,
   see specification
induction ...................... 155
induction principle ............... 116
inference rule
   dischargeability ............... 99
   excluded middle ............... 157
interactive theorem prover ...... 10
ISABELLE ....................... 6, 11, 18, 19
   Classical Reasoner .......... 20
   Induction Package .......... 20
   meta logic .................. 19
   Pure ........................ 19
   Simplifier .................. 20
tactic .......................... 20

L
labelling algorithm .............. 122
LCF .................................. 6
LEGO ................................ 7, 10
lemmification .................... 15, 120
logic
   E  ................. 24, 32, 113, 148
   E-T  .................... 161
   Grzegorzyck ............. 158
   IL  ....................... 32, 107, 141
   Kreisel-Putnam .......... 158
   Kuroda .................. 23, 157
   naively constructive .... 3, 139
   Scott .................... 158
uniformly constructive 5, 24, 137, 140
logical framework,
   see interactive theorem prover

M
Markov arithmetic ............... 158
model
   domain function .............. 158
   final ........................ 158
   frame ........................ 158
   isoinitial .................. 24, 88
   isomorphic embedding ..... 88
   Kripke ...................... 158
   reachable .................. 88
   valuation function ........ 159

N
node set ....................... 175
node set,
   see completeness theorem
number
   binary representation ....... 70
   bitwise operation .......... 67, 76
   integer .................... 67, 70
   modular .................... 76
   natural .................... 69
   signed modular ............ 77
   unsigned modular .......... 77
NUPRL .......................... 11

O
object code .................... 127
operator
   ∀I-Sub ..................... 111
   Coll ....................... 110
   Coll* ...................... 112
   DCP-Sub ................... 118
   ∃E-Sub .................... 111
   Exp  ...................... 112, 114, 115, 117, 118
   Exp* ...................... 112
   Fin ....................... 160
   Har-Sub .................. 115
   Har→-Sub ................ 115
   Har−→-Sub ................ 115
INDEX

Har-Sub .................. 115
ID-Sub .................. 116
Ind-Sub .................. 117
Inf .................. 110
Inf* .................. 112
\neg E-Sub .................. 113
\neg \exists I-Sub .................. 113
Subproof .................. 109
oracle .................. 22

P
parametric signature .......... 95
Peano arithmetic ........ 69, 117, 155
Presburger arithmetic .......... 82
program representation .......... 12
Program Synthesizer .......... 25
proof assistant,
see interactive theorem prover
Proof Analyzer ........ 21, 23
pseudo-truth set ........ 112, 141, 153
PVS .................. 6

R
real specification,
see informal specification
realizability .................. 159
reference set .................. 171
reference set,
see completeness theorem
regular rule,
see completeness theorem
regular swff,
see completeness theorem

S
saturated set .................. 174
saturated set,
see completeness theorem
Signature Morphisms Manager ... 25
signed well formed formula ........ 32
signed well formed formula,
see tableau
specification .................. 96
analysis .................. 126
specification framework ........ 15, 87
Specification Framework package .. 22
subproof relation ........ 108
SUP-INF algorithm .......... 82

T
T,
see tableau
tableau .................. 33, 36
active formula ........ 33
c-swff .................. 169
certain part ........ 33, 36
cl-swff .................. 169
classical part ........ 36
closed .................. 33
configuration ........ 32
consistent ........ 36
consistent set ........ 169
context ........ 33
contradictory set ........ 33, 39
dummy variable ........ 40
duplicating formula ........ 39
duplication extension ........ 169
F .................. 32, 159
F\Box .................. 36, 159
F_c .................. 32, 159
node .................. 32
non sound .................. 58
regular rule .................. 169
regular swff .................. 169
safe formula .................. 39
T .................. 32, 159
T\Box .................. 36, 159
terminal configuration ........ 33
terminal node ........ 33
translation algorithm .......... 40

T_\Box,
see tableau
term rewriting ........ 78
theorem prover ........ 10
theory
atomically complete ........ 89
E-Harrop ........ 93
Harrop ........ 93
identity ........ 116, 155
parametric .................. 95
trivial goal .................. 13  

**V**
validation by translation ........ 29

**W**
well formed formula ............ 32