**Université de Nice - Sophia Antipolis**
**UFR SCIENCES**

École Doctorale STIC

**THÈSE**

Présentée pour obtenir le titre de :
*Docteur en SCIENCES de l'Université de Nice - Sophia Antipolis*

Spécialité : INFORMATIQUE

par

**Kuntal DAS BARMAN**

Équipe d'accueil : LEMME - INRIA Sophia Antipolis

# Type theoretic semantics for programming languages

Thèse dirigée par **Yves BERTOT**

Soutenue publiquement à l'INRIA le 30 Septembre 2004 devant à 14h30
devant le jury composé de :

## Abstract

Semantics of programming languages gives the meaning of program constructs. Operational and denotational semantics are two main approaches for programming languages semantics. Operational semantics is usually given by inductive relations. Denotational semantics is given by partial functions. Implementing the denotational semantics inside type theory is difficult as the type theory expects total functions.

In this dissertation we develop a functional semantics for a small imperative language inside type theory and show its equivalence with operational semantics. We then exploit this functional semantics to obtain a more direct proof search tool, while developing a way to describe and manipulate unknown expressions in the symbolic computation of programs for formal proof development. In a third part, we address the problem of encoding complex programs inside type theory and we show how to circumvent the limitations of guardedness conditions as the are used in the Calculus of Inductive Constructions.

**Key words:** Type Theory, CIC, Coq, Semantics, Reflection, Compiler

## Résumé

La sémantique des langages de programmation donne la signification des constructions de programme. Les sémantiques opérationnelle et dénotationelle sont les deux principales approches pour la sémantique de langages de programmation. La sémantique opérationnelle est habituellement donnée par des relations inductives. La sémantique dénotationelle est donnée par des fonctions partielles. Mettre en application la sémantique dénotationelle à l'intérieur de la théorie des types est difficile car cette théorie ne supporte que les fonctions totales.

Dans cette thèse nous développons une sémantique fonctionnelle pour un petit langue impératif à l'intérieur de la théorie des types et montrons son équivalence avec la sémantique opérationnelle. Nous exploitons ensuite cette sémantique fonctionnelle pour obtenir un outil plus direct de recherche de preuve, tout en développant une manière de décrire et manipuler des expressions inconnues dans le calcul symbolique des programmes pour le développement formel de preuve. Dans une troisième partie, nous adressons le problème de coder des programmes complexes à l'intérieur de la théorie des types et nous montrons comment éviter les limitations des conditions de garde telle qu'elles sont employés dans le calcul des constructions inductives.

**Mot clés:** Théorie des types, CIC, Coq, Sémantique, Réflexion, Compilateur

To my family,

# Acknowledgments

I would like to express my deep sense of gratitude and everlasting indebtness to my thesis supervisor Yves Bertot for his constant support, patience and guidance with which he put me on the right path towards the completion of my thesis. His helps were not limited in academic activities. I still remember the initial days when he guided me to have a comfortable life in France. Such generous helps never stopped during my three and half years stay. He is a nice supervisor in every sense.

I thank to Gilles Dowek, Jean-François Monin, Savitri Maharaj and Laurence Pierre to kindly agree to be jury members of my thesis. They were very helpful in reading the thesis quickly, giving proper suggestions and finding time very early for my PhD defense. Their suggestions made my thesis more correct and complete.

I wish to thank Loïc Pottier, Laurence Rideau, Gilles Barthe, Laurent Théry, Marieke Huisman, Janet Bertot, Frédérique Guilhot and Philippe Audebeau for their constant support and encouragement. The members of the Lemme and Everest group were really co-operative and showed care whenever I needed.

I thank to my family who encouraged me to pursue higher education. Without their constant support all this would not have been possible.

It will be too little if I say thanks to my wife Sampa. She showed constant support during my PhD studies. Different language and culture in France did not stop her to come here to stay and share her life with me.

I thank to my officemates Nicolas, Benjamin and Asia for helping me whenever I sought their assistance. I am really thankful to Venanzio, Laurence, Benjamin and Bernard for the nice discussions we had during my PhD studies.

My friends and colleagues have contributed immensely to make my stay in France enjoyable and memorable. I thank to Marieke, who helped a lot to have a comfortable stay in France. I will remember several discussions I had with her during my life in INRIA. I also thank to Tamara, Hanane, Aubin, Guillaume, Pierre, Nestor, Simao, Laurent, Christoph, Sabrina, Mariela for their constant support, coupled with the humor and several going outs that have a long way in keeping my spirits up and making my stay enjoyable.

This space is really too small to name all those good people in INRIA and recount the great times I have had with them. I went for several memorable hiking and rock-climbing with the *Agos montagne* members, which helped to refresh my mind.

I should specially thank my friends in Antibes. Ana, Luis, Sylvain, Adeline, Olivier, Vijay, Jan, Charith, Balazs and me had several unforgettable get together. Be it hiking, snorkeling or 2VB party, they made my life really enjoyable.

I am thankful to Nathalie for her helpful support in administrative works. I also thank to administrative people in INRIA who took immediate and necessary steps whenever I needed. People in Semir were extremely supportive for the logistic support to make my work environment comfortable.

I am very thankful to every person and incident that has come my way, whose contact has made what I am.

# Contents

**1 Introduction** **1**
1.1 Programming languages . . . 3
1.1.1 Syntax . . . 3
1.1.2 Semantics . . . 3
1.2 Calculus of Inductive Constructions . . . 10
1.2.1 Terms and types . . . 10
1.2.2 Program execution and term reductions . . . 11
1.2.3 Predicates . . . 12
1.2.4 Logic . . . 12
1.2.5 Dependent types and dependent products . . . 13
1.2.6 Inductive types . . . 14
1.2.7 Inductive predicates . . . 17
1.2.8 Specification of functions . . . 18
1.2.9 General recursive functions . . . 19
1.3 Conclusion . . . 22

**2 Type-theoretic functional semantics** **25**
2.1 Introduction . . . 25
2.2 IMP and its semantics . . . 27
2.3 Functional interpretation . . . 31
2.3.1 The iteration technique . . . 32
2.3.2 Extracting an interpreter . . . 33
2.3.3 Characterizing terminating programs . . . 35
2.4 The Accessibility predicate . . . 36
2.5 Conclusions . . . 40

**3 Proof by reflection in semantics** **43**
3.1 Introduction . . . 43
3.2 IMP and its semantics . . . 45
3.2.1 Difficulty in automation . . . 45
3.3 Functional interpretation . . . 46
3.4 Proof by reflection . . . 48
3.4.1 Giving names . . . 49

3.4.2 The iteration technique . . . . . . . . . . . . . . . . . 53
3.5 Collection of data objects and table building . . . . . . . . . 54
3.6 Usefulness of Ltac . . . . . . . . . . . . . . . . . . . . . . . . 55
3.7 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

**4 An experimental compiler for Cminor to RTL - I 59**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.2 Cminor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.2.1 Memory structure . . . . . . . . . . . . . . . . . . . . . 63
4.2.2 Cminor operations . . . . . . . . . . . . . . . . . . . . . 64
4.2.3 Cminor expressions . . . . . . . . . . . . . . . . . . . . 65
4.2.4 Cminor instructions . . . . . . . . . . . . . . . . . . . . 67
4.2.5 Cminor procedures . . . . . . . . . . . . . . . . . . . . 69
4.2.6 Cminor program . . . . . . . . . . . . . . . . . . . . . . 71
4.2.7 Comparison with C . . . . . . . . . . . . . . . . . . . . 71
4.3 Register Transfer Language . . . . . . . . . . . . . . . . . . . 72
4.3.1 Operations in RTL . . . . . . . . . . . . . . . . . . . . 73
4.3.2 Memory address calculation in RTL . . . . . . . . . . . 77
4.3.3 RTL Instructions . . . . . . . . . . . . . . . . . . . . . . 77
4.3.4 Functions and programs in RTL . . . . . . . . . . . . . 79
4.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

**5 An experimental compiler for Cminor to RTL - II 81**
5.1 Cminor to RTL translation . . . . . . . . . . . . . . . . . . . 81
5.2 The translation environment . . . . . . . . . . . . . . . . . . 82
5.3 Allocation of registers for intermediate results . . . . . . . . . 85
5.4 Instruction selection . . . . . . . . . . . . . . . . . . . . . . . 87
5.5 Code linearization . . . . . . . . . . . . . . . . . . . . . . . . 89
5.6 Heuristics for static branch prediction . . . . . . . . . . . . . 91
5.7 Guardedness problem . . . . . . . . . . . . . . . . . . . . . . 91
5.8 Construction of the RTL control flow graph . . . . . . . . . . 93
5.9 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

**6 General conclusion 99**

**A Sémantique en théorie des types pour les langages de programmation 101**
A.1 Sémantique fonctionnelle en théorie des types . . . . . . . . . 103
A.2 Preuve par réflexion en sémantique . . . . . . . . . . . . . . . 111
A.2.1 Preuves par réflexion . . . . . . . . . . . . . . . . . . . 114
A.3 Un compilateur . . . . . . . . . . . . . . . . . . . . . . . . . . 116

**Bibliography 123**

**Index 127**

Chapter 1

# Introduction

One of the major concern in both the academics and in industry is to reduce the amount of errors or bugs in the computer programs. Often the source of the errors in computer programs is due to the bad implementation of the computer programs. We spend a lot of time to debug the programs, which finally does not guarantee the non-existence of errors. Formal methods provide us a way to reason about the computer programs we write. It not only helps to find out the bugs, the major benefits of the formal methods is in describing the computer programs in precise and rigorous specifications. Such specifications pave the way for rigorous formal proofs about the computer programs. In general these proofs are tedious and not often trivial. Proofs can be mechanized with the help of computers, where the latter is known to perform well for tedious work.

Verifying the source code does not remove all the errors. The code which is effectively executed by the machine, is generated from this source code by a compiler. Compilers can be another source of generating errors. With complex optimizations in the compiler, bugs are not rare. An incorrect compiler can introduce errors in the target code whose source code is well verified. Thus a formally verified compiler is of utmost importance. It will be more useful if the formalized compiler is moderately optimized and closer to one of the languages used in academics and in industries.

In practice, the source code of a program is often verified on a set of test data and the assembler code generated by the compiler is verified manually to see that the assembler code correlates to the source code. Both these tasks are easily error prone. In addition, there is no mathematical proof that they will produce safe and error free code. Type theory provides a basic pillar to produce safe and error free code. In fact, static typing, based on a sound type system is a basic requirement for robust systems engineering. Machine checked proofs about the source language and the compiler provide the mathematical base.

In this dissertation we study the formal semantics of programming languages in type theory. We start with an introduction to the programming languages in the next section 1.1. We give the overall picture of syntax and semantics of the programming languages. In the section 1.2, we describe the Calculus of Inductive Constructions, which we will use for the formalization of programming language properties. We machine check all the proofs in the proof assistant Coq, which is based on the Calculus of Inductive Constructions.

In chapter 2, we describe the operational and denotational semantics of a small imperative language in type theory with inductive and recursive definitions. We show the problem that arises to prove the equivalence between operational and denotational semantics, when the language contains partial and nested recursion. We provide a technique to work around this problem inside type theory.

Conventional approaches to describe the semantics of programming language usually rely on relations, in particular inductive relations. Simulating program execution then relies on proof search tools. In chapter 3, we describe a functional approach to automate proofs about programming language semantics. Reflection is used to take facts from the context into account. The main contribution of this work is that we developed a systematic approach to describe and manipulate unknown expressions in the symbolic computation of programs for formal proof development. The tool we obtain is faster and more powerful than the conventional proof tools.

The language C suits the best with the criteria we have set earlier for a formalized compiler. The work of this thesis is a contribution to the collective project *Concert*, where we plan to produce a formally correct compiler for C-like languages. The initial objective is to develop this formalization and this certification in the framework of the calculus of inductive constructions and to take advantage of the characteristics of this system to produce certified software, for instance with the extraction mechanism. The idea is to start with Cminor, a subset of C, which is sufficiently rich to describe almost all the C programs. Instead of generating the object code directly, we move to an intermediate code, written in a register transfer language(RTL), where we decide the control flow, allocate registers, verify constant propagation, eliminate dead code and optimize the code. The Register Transfer Language is closer to the language of the processor and RTL programs represent the control flow graph of the execution of the program. We generate the object code by linearization of the RTL graph. In chapter 4, we describe the abstract syntax of Cminor and RTL.

In the chapter 5, we present the formalization of the Cminor to RTL compiler. We discuss the difficulties that arise for this formalization and how we work around these difficulties. In this dissertation we realize a compiler which constructs the RTL control flow graph from the Cminor programs. The formal description of the compiler follows a style that is close to the Caml programming style and keeps the modularity of programming and significantly helps in its correctness proofs, though we do not consider any proofs in this dissertation. Finally we conclude giving the future directions of the work we have done.

## 1.1 Programming languages

The description of a programming language has two parts, syntax and semantics. A formal semantics for a programming language is a mathematically precise description of the intended meaning of each construct in the language. In contrast to a formal syntax for a language, which tells us which sequences of symbols are correctly formed programs, a formal semantics tells us what programs will actually do when we run them. The ideas of semantics are of importance for language designers, compiler writers, and programmers; they also provide a basis for mathematical proofs of the correctness of programs.

### 1.1.1 Syntax

The Grammatical structure of a program is expressed by its syntax. Syntax can be divided in two broad categories. The *Abstract syntax* of a programming language describes how to build the expressions and statements in that language, specifying the connections between logical parts of the language. It specifies the tree forms of a language, known as syntax trees. *Concrete syntax* provides sufficient information to construct unique parse trees. In other words, concrete syntax decides in which order evaluation of expressions in the language should take place. In this dissertation, we will consider only abstract syntax for syntactic entities so that there is no ambiguity in their parse tree structure, as we will only be interested in the meaning of programming languages and for that reason syntactic categories will suffice.

### 1.1.2 Semantics

The meaning of a grammatically correct program is revealed by its semantics. Semantics can also be viewed as relation between inputs, programs and outputs. The semantics of programming languages can be formalized mainly in three different approaches. *Operational semantics* is not only concerned with the result of the execution of a program, it puts stress on how to execute programs. *Denotational semantics* is only concerned with the result of the execution of a program and not on how to execute programs. *Axiomatic semantics* is concerned with the fact whether a program satisfies some partial correctness properties given a precondition and a postcondition.

#### 1.1.2.1 Operational semantics

The meaning of a program construct is specified by the computation which the program construct induces while executing on a machine. Operational semantics provides an abstraction of how the program is executed on a machine, and is independent of machine architectures and implementation strategies. There is an abstract notion of states. In operational semantics we are interested in how the states are modified during the execution of a program construct. In fact, operational semantics provides a relation between the old state, a program construct and the new state. There exists two different approaches for operational semantics. *Structural operational semantics* describes how the individual steps of the execution of a program take place [31]. For this reason this kind of operational semantics is also known as *small step semantics*. *Natural semantics* gathers all the execution for immediate constituents of a program construct to describe the final result of the execution and thus differs from structural operational semantics by hiding some execution details [23]. For this reason this kind of operational semantics is also known as *big step semantics*.

#### 1.1.2.2 Structural operational semantics

Let us assume $i$ is a metavariable ranging over instructions and $\sigma$ is a metavariable ranging over states for any imperative programming language. The transition relation in operational semantics specifies the final state of the execution of an instruction $i$ in a state $\sigma$. In small step semantics the transition relation is expressed by $\langle i,\ \sigma\rangle \rightsquigarrow \gamma$, where $\gamma$ can be either $\langle i',\ \sigma'\rangle$ or $\sigma'$. The transition specifies the first step of the execution of $i$ in the state $\sigma$. If the execution does not complete in the first step, the result of the execution is obtained from the execution of the instruction $i'$ in the state $\sigma'$, where $i'$ and $\sigma'$ are the transformed instruction and state after one step of execution, respectively. If the execution completes in the first step then it returns the final state $\sigma'$. The execution is said to be stuck if there is no $\gamma$ such that $\langle i,\ \sigma\rangle \rightsquigarrow \gamma$. An execution in small step semantics can therefore be a sequence of $\gamma$s, and this derivation sequence could be finite or infinite, depending on whether we reach a $\gamma$ in the derivation sequence which is either a terminal configuration or a stuck configuration, or not.

This execution process can be viewed as a reduction system. In a reduction system, when a term is not further reducible we call such a term as a normal form. A term which reduces to its normal form in a finite sequence of reductions, is called strong normalizing. A term which has a normal form, but also infinite chains of reductions, is called weakly normalizing. In structural operational semantics we say an execution terminates if there is a finite derivation sequence for that execution resulting the last element in a normal form. An execution loops if there is an infinite derivation sequence. In general proofs on structural operational semantics is conducted by induction on the length of the derivation sequence.

#### 1.1.2.3 Natural semantics

Instead of describing how the individual steps are performed, big step semantics is concerned with the relationship between the initial and the final state of an execution. In big step semantics the transition relation is given in the form $\langle i,\ \sigma\rangle \rightsquigarrow \sigma'$, thus specifying the relationship between the initial state $\sigma$ and the final state $\sigma'$ in the execution of the instruction $i$. For immediate constituents of the instruction $i$, say $i_1, i_2, \ldots, i_n$, natural semantics rule is given by a number of premises of the form $\langle i_1,\ \sigma_1\rangle \rightsquigarrow \sigma_1', \langle i_2,\ \sigma_2\rangle \rightsquigarrow \sigma_2', \ldots, \langle i_n,\ \sigma_n\rangle \rightsquigarrow \sigma_n'$, where $\sigma_1, \sigma_2, \ldots, \sigma_n$ are intermediate states in the execution, with the conclusion $\langle i,\ \sigma\rangle \rightsquigarrow \sigma'$. Any rule with no premises is known as an axiom. A derivation tree can be built for an execution of an instruction $i$ in a state $\sigma$ leading to a final state $\sigma'$ using the rules provided by natural semantics, where the root node is $\langle i,\ \sigma\rangle \rightsquigarrow \sigma'$ and the leaves are instances of axioms, while the internal nodes are conclusions of instantiated rules and their immediate sons are their corresponding premises. Execution of an instruction $i$ in a state $\sigma$ terminates if there is a state $\sigma'$ such that $\langle i,\ \sigma\rangle \rightsquigarrow \sigma'$. If there is no such state $\sigma'$ the execution loops. Proofs on natural semantics is conducted by induction on the shape of the derivation tree.

As we have noticed that natural semantics does not provide a stuck configuration, abnormal termination in natural semantics can not be distinguished from looping, whereas in structural operational semantics an infinite derivation sequence is given as a proof for looping and a finite sequence ending in a stuck configuration is given as a proof for abnormal termination. Again if we allow non-determinism in the programming language, given a choice between looping and termination natural semantics will possibly suppress looping because natural semantics always tries to find out the final state at termination. In such a case, structural operational semantics does not suppress looping as it tries to provide information about computation for the first step of execution and can not foresee looping. Finally, if we allow parallelism in the programming language, the set of results obtained by natural semantics will be a subset of the set of results obtained by structural operational semantics. The reason behind this is the fact that in natural semantics we consider the execution of each instruction, including its immediate constituents, as atomic entity. So we cannot express the interleaving of computations, whereas in structural operational semantics we express the execution of each small steps and can express the interleaving of computations.

In later chapters when we discuss our work, we consider a small imperative language and do not consider either abnormal termination, non-determinism or parallelism. In such a case natural semantics and structural operational semantics are equivalent, and we continue our work with natural semantics.

When the programming language is deterministic, the semantic relation is a function, but it is only a partial function when the programming language allows looping program construct. In our work we will consider an imperative language with looping construct, thus our semantic relation will be a partial function on states. Structural operational semantics was introduced by Gordon Plotkin and natural semantics was later derived from it.

#### 1.1.2.4 Denotational Semantics

Denotation semantics is interested in the association between initial states and final states for program constructs. In denotational semantics the meaning of a program is at a more abstract level where denotation of an instruction is considered to be a partial function on states. For each syntactic category in the programming language a semantic function has to be defined, which will map each syntactic construct to a function describing the effect of executing that construct. Semantic functions in denotational semantics are defined compositionally. A semantic clause is defined for each of the base elements of the syntactic category and for each of the composite elements of the syntactic category a semantic clause is defined in terms of the semantic function applied to the immediate constituents of the composite element. In other words structural induction is used to define denotational semantics. Execution of an instruction $i$ in a state $\sigma$ leads to a final state $\sigma'$ is denoted by $[\![i]\!]_\sigma = \sigma'$. Though the semantic relations for structural operational semantics and natural semantics associate a partial function from state to state to each instruction, but they are not defined compositionally.

We cannot define denotational semantics compositionally if there exists any syntactic category of nested recursive nature. For example, loops in programming languages. In such a case, semantic function has the form $[\![i]\!]_\sigma = (\ldots [\![i]\!]_{[\![i]\!]_{\sigma'}} \ldots)$. Structural induction or compositionality does not work in such a case, as the immediate constituent of $i$ is $i$ itself. To work around such a problem the semantic function is viewed as $[\![i]\!]_\sigma = F([\![i]\!]_{\sigma'})$, where the functional $F$ can be applied to the immediate constituent of $[\![i]\!]_\sigma$ or more precisely $[\![i]\!]_{\sigma'}$. This indicates $[\![i]\!]_\sigma$ must be a fixed point of $F$. Let us look at the fixed point theory a little more.

#### 1.1.2.5 Fixed point theory

A fixed point of a function $f$, is any value $x$, for which $f\ x\ =\ x$. A function may have any number of fixed points from none (e.g. $f\ x\ =\ x+1$) to infinitely many (e.g. $f\ x\ =\ x$). If $f$ is recursive, we can represent it as

$f = \text{fix } F$ where $F$ is some higher-order function and $\text{fix } F = F\ (\text{fix } F)$. The standard denotational semantics of $f$ is then given by the least fixed point of $F$. This is the least upper bound of the infinite sequence (the ascending Kleene chain) obtained by repeatedly applying $F$ to the totally undefined value, $\perp$ or bottom. I.e. $\text{fix } F = \text{LUB}(\perp,\ F\ \perp)$. The least fixed point is guaranteed to exist for a continuous function over a complete partial order.

Let $f$ be any fixed point of $F$ in $[\![i]\!]_\sigma = F([\![i]\!]_{\sigma'})$, that is $F\ f = f$, where $f$ is a partial function from state to state. There exist two different approaches to solve this fixed point equation. A general way to solve this is to use the theory of complete partial order and continuous functions. When the instruction $i$ keeps on looping on itself, $[\![i]\!]_{\sigma_n} = \sigma_{n+1}$ for all $n$. Thus we have $f\ \sigma_n = \sigma_{n+1}$ for all $n$, which leads to the fact that $f\ \sigma_0 = \sigma_n$ for all $n$ and makes it difficult to determine the value of $f\ \sigma_0$. Various fixed points of $F$ may differ in such a situation, let $f$ be the desired fixed point fix $F$ then for any other fixed point of $F$, say $f'$, $f\ \sigma = \sigma'$ should imply $f'\ \sigma = \sigma'$ for all choices of $\sigma$ and $\sigma'$. This automatically leads to an ordering between $f$ and $f'$ which we discuss in the next subsection.

Another way to solve the fixed point equation $[\![i]\!]_\sigma = F([\![i]\!]_{\sigma'})$ is to consider the operator on sets determined by rule instances given by rule induction, of which induction principle, viz. structural induction for operational semantics is a special case. Let $\hat{R}$ be an operator on set $R$ such that it takes the premises of the rule instances in $R$ and returns the conclusions. Applying $\hat{R}$ on empty set will return the conclusions of instances of axioms, say $\hat{R}(\emptyset)$. Remember, axioms are the rule instances with no premises. Applying $\hat{R}$ on $\hat{R}(\emptyset)$ will return the conclusions of the rule instances whose premises are in the set $\hat{R}(\emptyset)$. Similarly it goes on. Clearly this forms a monotonic chain $\emptyset \subseteq \hat{R}(\emptyset) \subseteq \hat{R}^2(\emptyset)\ldots$ Let $A$ be the set of conclusions obtained as $\bigcup \hat{R}^n(\emptyset)$, where $n$ ranges from 0 to $\infty$. Intuitively $A$ contains all the possible derivations from the set of rule instances $R$. It can be easily proved that $\hat{R}(A) = (A)$ holds and $A$ is the least fixed point of it. In the fixed point equation $[\![i]\!]_\sigma = F([\![i]\!]_{\sigma'})$, $F$ matches with $\hat{R}$ for the properly defined set of rule instances for $i$.

#### 1.1.2.6 Complete partial order and continuous functions

In general, very few recursive functions can be easily expressed by least fixed points of operators on sets. Denotational semantics uses complete partial order and continuous functions to deal with them. A partial order is defined by a pair of a set, say $P$, and a binary relation $\sqsubseteq$ over $P$, such that $\sqsubseteq$ is reflexive, transitive and antisymmetric. For a partial order $(P, \sqsubseteq)$ and a subset $P' \subseteq P$, $p$ is an upper bound of $P'$ if and only if $\forall q \in P'.\ q \sqsubseteq p$ and in addition if for all upper bounds $q$ of $P'$ $p \sqsubseteq q$ holds, $p$ is called least upper bound of $P'$. A partial order $(P, \sqsubseteq)$ is called a complete partial order if it has least upper bounds of all infinite chains $p_1 \sqsubseteq p_2 \sqsubseteq \ldots$, where $p_1, p_2, \ldots \in P$.

A function $f : P \rightarrow Q$, where $P$ and $Q$ are complete partial orders, is continuous if it is monotonic and $f$ (LUB $P'$) = LUB ($f$ $p'$) $\mid p' \in P'$ for all directed sets $P'$ in $P$. By directed set $P'$ we mean, if $p'_1, p'_2, \ldots$ are elements of $P'$ then there exists a chain $p'_1 \sqsubseteq p'_2 \sqsubseteq \ldots$. In other words, the image of the least upper bound in continuous function is the least upper bound of any directed image. A continuous function has a least fixed point if its domain has a least element. When a least element, bottom or $\perp$, is added in a complete partial order the least fixed point is given by fix $f$ = (LUB ($f^n$ $\perp$)) where $n$ ranges from 0 to $\infty$.

Semantic functions for instructions in denotational semantics are given by partial functions $\Sigma \rightharpoonup \Sigma$, where $\Sigma$ is the set of states and $\Sigma$ forms a complete partial order with inclusion order of states. Partial functions on states can be considered as continuous total functions if we extend the set of output states $\Sigma$ to a complete partial order by adding a least element $\perp$ to form $\Sigma_\perp$, ordered by $\forall \sigma.\ \perp \sqsubseteq \sigma$. Intuitively this means adding a state corresponding to an undefined state. There exists an one to one correspondence between the partial functions $\Sigma \rightharpoonup \Sigma$ and the total functions $\Sigma \rightarrow \Sigma_\perp$. Inclusion order between partial functions corresponds to the pointwise order between total functions. Thus semantic functions in denotational semantics can be considered continuous functions over a complete partial order and recursive equations are solved by taking the least fixed point of this function.

In denotational semantics programs are translated into functions about which properties can be proved using the standard mathematical theory of functions, and especially domain theory.

#### 1.1.2.7 Domain theory

In the denotational semantics of programming languages, the meaning of a program construct is given by assigning it to an element in a domain of possible meanings. Different domains correspond to the different types of object with which a program deals. A domain is a mathematical structure consisting of a set of values (or ”points”) and an ordering relation, $\leq$ on those values. Domain theory studies such structures. We have already given a small description of complete partial order and continuous functions. The traditional approach to have a theoretical development of denotational semantics is to develop a meta language for expressing denotational definitions. The theoretical foundation of this language then ensures that the semantic functions exist as long as we use domains and operations from the meta language.

Denotational semantics is much more widely applicable than to simple imperative programming languages, it can handle virtually all programming languages, though the standard framework appears inadequate for parallelism and fairness. Denotational semantics can handle abnormal execution and non-determinism using power-domains. Power-domains are complete partial order analogues of powersets enabling denotations to represent sets of possible outcomes. Gordon Plotkin has introduced them in [32]. $\lambda$-calculus provides the mathematical base of denotational semantics. Christopher Strachey and Dana Scott pioneered the approach of denotational semantics by providing mathematical foundations.

#### 1.1.2.8 Axiomatic semantics

Axiomatics semantics is appropriate for reasoning about program correctness. The use of operational semantics or denotational semantics to reason about specific properties of programs is not always convenient and sometimes may not be possible.

In axiomatic semantics, properties of programs are specified as assertions. An assertion is a triple of the form $\{A\}\ P\ \{B\}$, where $P$ is a program construct and $A$ and $B$ are predicates. $A$ is called the precondition and $B$ is called the postcondition. This triple is also known as the Hoare triple. The meaning of the assertion $\{A\}\ P\ \{B\}$ is that if $A$ holds in the initial state $\sigma$ and if the execution of the program construct $P$ from the initial state $\sigma$ terminates in a state $\sigma'$ then $B$ holds in the final state $\sigma'$. Assertions of the form $\{A\}\ P\ \{B\}$ are called partial correctness assertions because they do not say anything about the program construct $P$ if it fails to terminate. Intuitively, if we view the program as a state transformer or a collection of state transformers, the axiomatic semantics is a set of invariants on the state which the state transformer satisfies.

There are two approaches on specifying the preconditions and postconditions of the assertions. More commonly used the intensional approach introduces an explicit language called an assertion language and then the conditions (pre- or post-) will be formulæ of that language. This assertion language is much more powerful than the boolean expressions as it needs to express all the possible preconditions and postconditions. The extensional approach reformulates the meaning of $\{A\}\ P\ \{B\}$ as if $A$ holds on a state $\sigma$ and if $P$ is executed from $\sigma$ results in the state $\sigma'$ then $B$ holds on $\sigma'$. In the extensional approach an inference system is built to specify the partial correctness assertions. It consists of a set of inference axioms and inference rules similar to the derivation rules for natural semantics. An inference rule has the form $\{A_i\}\ i\ \{B_i\}$ for each instruction $i$, where $A_i$ and $B_i$ are predicates. In general there exists a rule of consequence in the inference system which states that a precondition can be weakened at the cost of strengthening the postcondition. Similar to the derivation tree in natural semantics an inference tree can be built with the inference rules for an execution. An inference tree provides a proof of the property expressed by its root. Proofs of properties about program constructs are conducted on the shape of the inference tree.

In the extensional approach the inference system needs to be sound and complete. An inference rule is valid if its conclusion is valid under valid premises and when all the inference rules are valid the inference system is sound. An inference system is complete when all the valid partial correctness assertions can be obtained by its rules. *Gödel's incompleteness theorem* suggests that there is no effective proof system for partial correctness assertions such that its theorems are precisely the valid partial correctness assertions. In the intensional approach the problem comes with the expressiveness of the assertion language, which also needs to be finitely computable.

Axiomatic semantics can be extended to verify total correctness properties, where assertions have the form $\{A\}\ P\ \{\Downarrow B\}$, which says that if the precondition $A$ is satisfied in the initial state then the program construct $P$ is guaranteed to terminate and the final state satisfies the postcondition $B$. The proof system for total correctness can be further extended to prove the order of magnitude of the execution time of a program construct.

Axiomatic semantics can handle abnormal execution, non-determinism and parallelism naturally. Traditionally first order predicate logic or temporal logic provides the mathematical foundation of axiomatic semantics. Floyd invented rules for reasoning on flow charts and later Hoare modified and extended these rules to pioneer axiomatic semantics. In our work we will limit ourselves in natural semantics and functional semantics.

## 1.2 Calculus of Inductive Constructions

To machine check the proofs about the programming language properties we use the proof assistant **Coq**, which helps to build the proofs in an interactive way with the help of automatic search tools whenever possible. A remarkable characteristic of **Coq** is the possibility to generate certified programs from the proofs, and more recently, certified modules. The main objective of this section is to give a brief introduction to the underlying theory, the Calculus of Inductive Constructions, of the Coq proof assistant.

### 1.2.1 Terms and types

Terms are the basic ingredients of Calculus of Inductive Constructions. In most type theories, one usually makes a syntactic distinction between types and terms. This is not the case for Calculus of Inductive Constructions which defines both types and terms in the same syntactical structure. This is because the type theory itself forces terms and types to be defined in a mutual recursive way and also because similar constructions can be applied to both terms and types and consequently can share the same syntactic structure. In general there are two kind of terms, expressions and types.

Expressions are formed with constants and identifiers, following a few construction rules. Every expression has a type, the type for an identifier is usually given by a *declaration* and the rules that make it possible to form combined expressions come with *typing rules* that express the links between the type of the parts and the type of the whole expression.

Type checking is done with respect to an environment and a context. An environment contains all the global declarations and a context contains all the local declarations. For examples, axioms are in the environment and hypotheses are in the context. To start with, types are of two kinds. *Atomic types* are made of single identifiers. For example, $\mathbb{N}, \mathbb{Z}$ and $\mathbb{B}$. *Arrow types* are of the form $A \rightarrow B$, where $A$ and $B$ are themselves types. Arrow types represent types of functions, thus $A \rightarrow B$ is a function which takes an argument of type $A$ and returns a value of type $B$. An arrow type $A \rightarrow B \rightarrow C$ is either the type of a function taking two arguments of type $A$ and $B$ and returning a value of type $C$, or the type of a function taking an argument of type $A$ and returning a function of type $B \rightarrow C$. In Calculus of Inductive Constructions we consider all functions are total, making every function a terminating computation process.

In the **Coq** proof assistant we can write programs in a similar way we write them in functional programming languages. In this dissertation we are going to discuss about the type theory based semantics for programming languages. In the following sections we prepare ourselves for the programming language constructs in type theoretical context. In the Calculus of Inductive Constructions, a declaration attaches a type to an identifier, without giving the value. A definition either assigns a well formed term to an identifier as a value, or gives both the type and the value to an identifier. Notions of free and bound variable comes from the notion of the $\lambda$-abstraction. When all free occurrences of a variable $v$ in a term $t$ is replaced by another term $u$ we denote it by $t\{v/u\}$, where all free variables in $u$ remains free in $t\{v/u\}$. Renaming of a bound variable in a $\lambda$-abstraction is known as $\alpha$-conversion.

### 1.2.2 Program execution and term reductions

Programs can be executed in Calculus of Inductive Constructions. Though the main intention is to develop correct programs, computation is necessary even to check that some expressions are well typed. Computations are performed by term reductions. In general reductions are done along with $\alpha$-conversions. There exists four kinds of reductions. $\delta$-reduction replaces an identifier with its definition. $\beta$-reduction is the application of a lambda abstraction to an argument expression. $\zeta$-reduction removes local definitions occurring in terms by replacing the defined variable by its value. More precisely, it replaces any formula of the form *let* $v := e_1$ *in* $e_2$ into $e_2\{v/e_1\}$. $\iota$-reduction deals with recursive functions. The typing rules of Calculus of Inductive Constructions interact with reductions. The type of a term remains same after any series of reductions. Every sequence of reductions from a given term in the Calculus of Inductive Constructions terminates and this property is known as strong normalization. Reductions on a term can be applied in any order. In the Calculus of Inductive Constructions it is decidable whether two terms are convertible, where two terms are convertible if they can be reduced to the same term.

A call-by-value strategy or lazy strategy can be used to evaluate the arguments of a function call. In call by value strategy the arguments will be evaluated first and later their values will be passed. In the lazy strategy the evaluation of arguments will be delayed as long as possible.

### 1.2.3 Predicates

In the Calculus of Inductive Constructions, the expressions and types are considered as particular cases of terms. The type of a type is called a sort. Two basic sorts in the language of the Calculus of Inductive Constructions are **Set** and **Prop**. The sort **Prop** intends to be the type of the logical propositions. The logical propositions themselves are typing the proofs. An object of type **Prop** is called a proposition. The sort **Set** intends to be the type of specifications. The specifications themselves are typing the programs. An object of type Set is called a program. These two sorts constitute two different semantic subclass of the syntactic class *term*. The Calculus of Inductive Constructions considers an infinite hierarchy of sorts called universes. Universes are constituted with types *Type(i)* for every $i$ in $\mathbb{N}$, such that the type of **Set** and **Prop** is Type($i$), for every $i$ and type of Type($i$) is Type($j$), where $i < j$. Set of terms is organized in different levels and the type of every term at the level $i$ is a term at the level $i + 1$. The Calculus of Inductive constructs hides the hierarchy and instead the notation *Type* is used for any type Type($i$). This leads to the fact that the type of **Set** is Type and the type of Type is also Type.

### 1.2.4 Logic

The Calculus of Inductive Constructions uses intuitionistic logic to reason about programs. Heyting introduced the intuitionistic logic in [22]. In the intuitionistic logic to prove a proposition $P$ true, we look for proofs of $P$. A proof of implication $P \Rightarrow Q$ is considered as a process to obtain a proof of $Q$ from a proof of $P$. From the functional programming point of view, a proof of $P \Rightarrow Q$ is a function that given an arbitrary proof of $P$ constructs a proof of $Q$. The Curry-Howard isomorphism provides the correspondence between $\lambda$-Calculus as a model of functional programming and proof calculi like natural deduction [35]. In intuitionistic logic we can extract correct programs from proofs. In functional language a proof can be considered as a expression and the proven statement as the type of proofs for the statement. For example, the implication $P \Rightarrow Q$ is the arrow type $P \rightarrow Q$. The implication can be stated as an abstraction of the form $\lambda H : P,\ t$ where $t$ is a proof of $Q$, well formed in the context which has a hypothesis $H$ stating $P$. Logical developments and programs are developed in the similar way, where the universe of logical propositions is the sort **Prop** and the universe of specifications is the sort **Set**. Two programs may be totally different for the same specification as they may differ in implementation or efficiency, whereas two proofs of a proposition, if they exist, are equally important to ensure the truth of the proposition. Thus proofs of a given proposition can be interchanged, and this possibility is known as proof irrelevance.

### 1.2.5 Dependent types and dependent products

In the Calculus of Inductive Constructions arrow types represent type of functions. But types can also be passed as arguments to functions in the Calculus of Inductive Constructions. If we consider a function $f$ which takes an argument of type $A$ and returns a type, we may consider another function that returns its value in type $(f\ a)$. Such a definition makes it possible to consider functions whose result type vary with the argument. These return types are called dependent types. When a function returns a proposition as a result we call such a function as a predicate. In the Calculus of Inductive Construction typing rule for dependent types says that in an environment and in a context if $A$ is of type **Set** and $B$ is of type Type then the arrow type $A \rightarrow B$ has type Type in the same environment and context. Types for the predicates and the parameterized data types can be obtained if $B$ is assigned to **Prop** and **Set** respectively in the typing rule for dependent types. And the case where both $A$ and $B$ has type Type leads to higher order types.

In the Calculus of Inductive Constructions, a unified formalism for universal quantification and product types is provided by the dependent product construct. A dependent product is a type of the form $\forall a : A,\ B$ where $A$ and $B$ are types and $a$ is a bound variable of type $A$ and scope of $a$ covers $B$. Thus the variable $a$ can have free occurrences in $B$. Dependent products are similar to the cartesian products in mathematics, the main difference between them is that dependent products use types whereas cartesian products in mathematics use sets. Dependent products also differ from abstractions as the abstractions describe type expressions in which functions may occur and dependent type describe functions and not the function types. If $t$ is term of type $B$ then the type of the abstraction $\lambda v : A \Rightarrow t$ is the dependent product $\forall v : A, B$. When $v$ does not occur in B the product $\forall v : A, B$ is the simple arrow type $A \rightarrow B$. They are also called non-dependent products as the variable for the input does not occur in the result type. The expressive power of dependent products in the Calculus of Inductive Constructions is based on Martin Lof's type theory. Dependent products are also subject to $\alpha$-conversion.

Dependent types make it possible that the type of a program can contain constraints expressed as propositions that must be satisfied by the data. A certified program can be expressed as a function which computes and also provides a certificate for the computation. There are two approaches to define functions and providing proofs that they satisfy a given specification. In the *weak specification*, functions are specified to do the computations and later companion lemmas are added for the proof. For example, given a relation $R : A \rightarrow B \rightarrow \textbf{Prop}$, we define a function $f$ of type $A \rightarrow B$ and prove a lemma with a statement of the form $\forall a : A,\ R\ a\ (f\ a)$. In the *strong specification*, the type of the function includes the type of the proof. For example, the type of the function $f$ states that it takes an argument $a$ of type $A$ and the result is the combination of a value $b$ of type $B$ and a proof that $(R\ a\ b)$ holds. Adding proof arguments to functions makes it possible to make the type of these functions more explicit about their behavior. Functions defined using the strong specification style are also known as well-specified functions and usually such functions rely on dependent types.

### 1.2.6 Inductive types

Inductive types provide ways to specify programs and to verify the consistency of the specification. Thus the Calculus of Inductive Constructions provides a lot of power to the inductive types with respect to the types given in conventional programming languages. Inductive types help to build certified programs, programs whose type specifies exactly the behavior.

Inductive type paves the way of building the data structures and type definitions in the Calculus of Inductive Constructions. Inductive types are constructed with the help of constructors, where the constructors represent different possible cases of the type. For example, to define the type of boolean values, there exists two possible values, true and false. The boolean type is defined as follows:

```
Inductive bool : Set := true : bool | false : bool.
```

For each inductive definition of a type $T$, the **Coq** system adds several theorems and functions that make it possible to reason and compute on data in this type. of theorems. These theorems always have the same form, they contain a universal quantification over a variable $P$ of type $T \rightarrow s$ where $s$ is a sort and their statement ends with formula of the form $\forall x : T\ (P\ x)$. When the sort $s$ is **Prop** the theorem is named *T_ind*, when the sort $s$ is **Set** the function is named *T_rec* and when the sort $s$ is **Type** the function is called *T_rect*. *T_ind* is the induction principle associated to the inductive definition. For example, the induction principle of the boolean type defined above is as follows:

```
bool_ind :
  ∀ P: bool→Prop,
  P true → P false →
∀ b:bool, P b
```

Each inductive type corresponds to a computation structure, based on pattern matching and recursion.

Inductive types can have types or values as parameters. Constructors of the inductive types can have dependent types or functions. To write recursive functions with these inductive types the dependent arguments must of the constructors should appear in the pattern for the pattern matching constructs. Inductive types can be formed where subterms are in another instance of the same inductive type than the whole term.

The strength of inductive types in the Calculus of Inductive Constructions is mostly a consequence of their interaction with the dependent product. With the extra expressive power a large variety of properties on data and programs can be formulated simply using type expressions. Dependent inductive types easily cover the usual aspects of logic: connectives, existential quantification, equality, natural numbers. All these concepts are uniformly described and handled as inductive types. The expressive power of inductive types also covers logic programming, in other words, the languages of the Prolog family.

Non-recursive functions are defined with the *Definition* construct. Pattern matching makes it possible to describe functions that perform a case analysis on the value of an expression whose type is an inductive type. The pattern matching construct is associated with the $\iota$-reduction. $\iota$-reduction adds some conversion rules for different cases of the pattern matching construct in a conversion table. These rules are used to compare two terms by the type-checking process.

Recursive types can be defined in the Calculus of Inductive Constructions. Recursive types are simple inductive types, where some data fragments that has the same nature as the whole. Recursive types help to reason about data structures of unknown size. Recursive types allow to build infinite sets where each element is constructed in finite number of steps. For example, natural number is represented by Peano's arithmetic in the Calculus of Inductive Constructions as follows:

```
Inductive nat : Set := O : nat | S : nat→nat.
```

where S stands for successor function. Thus any natural number can be constructed by repeatedly applying the successor function over zero.

The Calculus of Inductive Constructions allows to define an empty type. We call a type empty if no element of this type can be built. An usual way to define an empty type is to define an inductive type with no constructor. Later we will see such an example to build the inductive predicate *False*. A dependent type may be empty for some of its arguments, but the Curry-Howard isomorphism says that even in such a case the type can carry logical information.

The computation structures of the inductive types provide the basis of recursive programming. In the Calculus of Inductive Constructions the recursive functions are defined over recursive types using the *fixpoint* construct. To ensure that the recursive function is well-defined, definitions of recursive functions are organized around the structure of the inductive type. For each constructor of the inductive type a value is defined for the function, if the constructor is recursive then a recursive value is given. Such pattern of recursive definition is called structural recursion. The Calculus of Inductive Constructions allows to define functions whose terminations are guaranteed.

Recursive functions can be defined on any number of arguments, but the recursive structure of the function relies on a particular argument which is called the principal argument. The constructors in the principal argument help to carry out the function computation by $\iota$-reduction. Function computation is done by pattern matching on the principal argument and recursive calls are only available for the subterms of the principal argument of the same type.

The computation of a structural recursive function follows the structure of its principal argument. Reasoning about a structural recursive function relies on a proof by induction on the principal argument of this function. Thus the proofs about such functions follow the structure of the pattern matching constructs present in the function.

In the Calculus of Inductive Constructions abstraction is used to build a non-recursive function inside a term. To build a recursive function directly inside a term the *fix* command is used, which does not provide a name for the function. The *fix* construct only describes a function and unlike the *fixpoint* construct it does not define a constant having this function as its value.

Dependent types make it possible to define polymorphic types in the Calculus of Inductive Constructions. In general, polymorphic types are defined as dependent types with arguments. These arguments appear as parameters in the inductive definition of the polymorphic type. When parameters are provided, they must appear at every use of the type being defined. Similar to the inductive types, recursive functions and pattern matching can be performed on the polymorphic types. Since the parameters can not be bound in the pattern, parameter arguments do not appear in the constructor of pattern matching clauses. To describe a large class of partial functions an *option type* can be defined as a polymorphic type of the following form:

```
Inductive option (A:Set) : Set :=
    Some : A→option A
  | None : option A.
```

To define a partial function which takes an argument of type $A$ and returns a value $y$ of type $B$, it is often possible to define the partial function as a total function from $A$ to *option* $B$, such that the function returns *None* when the function is not defined and returns *Some y* when the partial function should have returned *y*.

### 1.2.7 Inductive predicates

A dependent type with one argument could be empty or not depending on the value of this argument. The Curry-Howard isomorphism can exploit this aspect. A dependent inductive type can represent a predicate that is provable or not depending on the value of its argument. In the Calculus of Inductive Constructions inductive types are systematically used to describe predicates. In general, these inductive types are defined in the sort **Prop** rather than in the sort **Set** because they are used to represent properties rather than the data types. Proof irrelevance plays a role here; the exact form of an inhabitant of an inductive property is irrelevant, only its existence matters. The choice between **Set** and **Prop** as the sort of an inductive type also influences the extraction tools that produce executable programs from **Coq** developments.

In general, in the Calculus of Inductive Constructions logical connectives are represented as inductive types. Among the exceptions are implication, universal quantification and negation. Implication and universal quantification are directly represented using products and negation is represented as a function on top of *False*. The constructors in the inductive definition of a logical connective correspond to the introduction rules for these connectives in natural deduction and the induction principles correspond to the elimination rule. The proposition which can be proved in any context is given by the inductive definition of *True*, which has a constructor I representing the proof without conditions.

```
Inductive True : Prop := I : True.
```

The contradictory proposition can never be proved and thus given by the inductive definition of *False* which has no constructor.

```
Inductive False : Prop := .
```

The logical connective *there exists* is obtained with the following inductive definition:

```
Inductive ex (A:Type)(P:A→Prop) : Prop :=
             ex_intro : ∀x:A, P x → ex A P.
```

Equality between two terms, of the same type or which can be converted to the same type, is expressed by a parameterized inductive type.

```
Inductive eq (A:Type)(x:A) : A→Prop := refl_equal : eq A x x.
```

Dependent types make it possible to have two terms in two types which are provably equal but not convertible.

The Calculus of Inductive Constructions puts restriction on function specification that the function should be total. This requirement for termination of reductions imposes strong limitations on what terms can be formed. Sometimes this makes it difficult to specify a function $f$ by a functional term in the Calculus of Inductive Constructions. In particular, partial functions are difficult to describe. Using an *option* type does not help enough. Inductive definitions make it possible to relax the constraints on the functions to describe them formally. For example, to describe a function $f$ from $A$ to $B$, the inductive definition is very useful if it is undecidable whether a given value belongs to the function's domain. The inductive definition of the function $f$ gives a logical characterization of the set of pairs $(x, f(x))$. The inductive predicate describing the function $f$ has the type $A \to B \to \mathbf{Prop}$.

Inductive definition makes it possible to describe a function whose termination is not guaranteed. This approach is very useful for the description of programming languages. To represent the semantics of a programming language directly by a function, which takes the initial state and the program as arguments and returns the final state after executing the program, is impossible to describe as soon as the language is Turing-complete. Otherwise this would mean solving the halting problem, which is undecidable. On the other hand a description of the semantics by the inductive relation is possible.

The description of a function $f$ that takes $k$ arguments as an inductive relation is given by an inductive predicate $P_f$ that relates $k$ input values with the result value. Thus the predicate $P_f$ has $k+1$ arguments. This predicate $P_f$ is described with a collection of constructors which describes all the cases that appear in the function. For each of these cases, the form of the input data is described in the constructor's final type, the constraints on the input are described as premises, recursive calls of the form $f\ x_1\ x_2\ \ldots\ x_k$ are also represented by premises of the form $P_f\ x_1\ x_2\ \ldots\ x_k\ y$ where $y$ is a fresh variable representing the final value of this recursive call. To describe the type of all variables appearing in the constructor universal quantifications are added. If a function takes $k$ argument and returns a tuple of $p$ arguments as a final value, an inductive predicate with $k+p$ arguments can be defined to describe the function as an inductive relation.

Induction principle of inductively defined predicate precisely uses the facts that are expressed by the constructors, thus makes the proof by induction on inductive predicates more efficient.

### 1.2.8 Specification of functions

Strong specification of functions rely on the inductive types in the **Set** sort that have constructors with the arguments in the**Prop** sort. Thus the specification combines a data type and a predicate over this type as a proof. This proof argument is not used for computation. In the **Coq** library such specification has the following type:

```
Inductive sig (A:Set) (P:A→Prop) : Set :=
     exist : ∀x:A, P x → (sig A P).
```

where in the constructor *exist* the computation component is described by $\forall x : A$ and the proof over this data type is $P\ x$. The inductive type *sig* corresponds to the Σ-type[6]. Since the result type of sig is in the **Set** sort, a function of type $(sig\ A\ P) \rightarrow A$ can be constructed such that given a value of type $(sig\ A\ P)$ a witness of type $A$ can be obtained which satisfies the property $P$. From a $(exist\ x\ p)$, where $p : (P\ x)$ we can extract its witness x:A (using an elimination construct) but not its justification $p$, which stays hidden, like in an abstract data type. In technical terms, sig is a *weak (dependent) sum.* A *strong (dependent) sum* is defined by defining the predicate $P$ as $A \rightarrow$ **Set**.

The constructive sum of two propositions $A$ and $B$ is given by the type sumbool, which serves as a well-specified version of the type bool. The sumbool type is defined inductively as follows:

```
Inductive sumbool (A B:Prop) : Set :=
    left : A → (sumbool A B)
  | right : B → (sumbool A B).
```

Traditionally in the **Coq** system functions with the result type sumbool are denoted with _dec suffix. Such functions are used to decide between two alternatives. Using the sumbool type is like attaching a comment to the function definition, which gives the meaning of this function and the type system verifies the validity of this comment.

### 1.2.9 General recursive functions

Sometimes it is difficult to express the termination of a recursive function as structural recursion with respect to one of the function arguments. Later in the chapter we discuss such an example coming from the recursive nature of *while loops* in programming language instructions. In the Calculus of Inductive Constructions there are several methods to work around this difficulty.

#### 1.2.9.1 Bounded recursion

In the first method the function is defined by structural recursion on an artificial argument. In general this artificial argument represents the complexity of the function to be defined, it is calculated before calling the function and then added to the function. In this way we express the recursive function by bounded recursion. Bounded recursion adds extra computation on top of the desired function. First, the bound must be determined before calling the function and second, the function needs to be modified adding extra computation on this argument so that the function becomes structurally recursive with respect to this argument, thus making the function different from the original function. This extra computation remains even after the extraction process.

#### 1.2.9.2 Well founded recursion

A second method uses a well founded relation to define the recursive function. A well founded relation guarantees a finite descending chain and all recursive calls of the function are done on an expression which is a predecessor of the initial argument for this relation, thus ensuring the fact that there exists no infinite loop. In general, the relation that relates any element of an inductive type to its strict subterms can be proved well-founded by structural induction. The **Coq** proof assistant provides a collection of theorems to build and work on well-founded relations in the module Well-founded. Recursive functions are built with the help of the recursor well_founded_induction. The type of well_founded_induction is as follows:

```
well_founded_induction
    :∀(A:Set)(R:A→A→Prop),
        well_founded A R→
        ∀P:A→Set, (∀x:A, (∀y:A, R y x → P y) → P x) →
             ∀a:A, P a
```

Thus the recursor takes five arguments and returns a function of the type $\forall a : A,\ P\ a$. The first argument $A$ describes the input type of the function to be defined. The second argument $R$ describes a binary relation. The third argument $well\_founded\ A\ R$ is a proof that the relation $R$ is well founded. The fourth argument $P$ is a dependent type and gives the output type of the function to be defined. The fifth argument is a dependently typed function which describes the computation process for an element of type $A$ such that the recursive calls are performed on expressions which are smaller than this element with respect to the relation $R$. This function takes two arguments. The first argument $x$ represents the initial argument of the function we want to define and the second argument is a dependently typed function representing the recursive calls of the function to be defined. The relation $(R\ y\ x)$ inside the fifth argument is the type of a proof that recursive calls are only allowed to a predecessor $y$ of $x$ for the relation $R$ provided for the function to be defined. Since the third argument in the function to be defined says that the relation $R$ is well-founded, to ensure that there is no infinite chain of recursive calls it is enough to show that the recursive calls are only allowed on smaller elements for the relation $R$. The induction principle corresponding to the recursor well_founded_induction is also provided in the **Coq** module Well-founded as well_founded_ind.

Though this method makes the extracted code more reliable, this method is in general more complex in nature and difficult to reason about for functions with weak specification. Well-specified recursive function when defined directly with the well-founded induction removes the complexity of extra proofs for the function specification. Well-founded recursive functions are difficult to reason about directly. An approach to reason directly about them is to first make sure that we are working in a good context where enough of the definitions are theorems are available for such a purpose. Then the proof is done using the maximal induction predicate provided in the **Coq** library as Acc_inv_dep. Another approach to reason directly about a well-founded function is to rely on a fix-point equation for the function. With the fixpoint equation the proofs are carried on by the well_founded_induction using the induction principle called well_founded_ind.

#### 1.2.9.3 Recursion by iteration

In a third method, proposed by Balaa and Bertot [1], recursion is achieved by iteration. This approach makes the style of programming closer to the usual style of conventional functional programming languages. In this approach, first the functional associated to the recursive function is determined. Iterating this functional a function is defined which performs the intended computation of the recursive function we want to define. With the help of the well-founded induction this function is proved to be total, in other words it is proved that this function terminates. The recursive function is then built from this function by ignoring the termination proof. Let $f$ be a recursive function which takes an argument $x$ of type $A$ and returns a type $B$. The definition of $f$ can be written as an equation of the following form

$$f\ x\ =\ \ldots f\ x\ \ldots$$

We can define a functional $F$ which takes $f$ and $x$ as arguments and maps them to the right side of the above equation. The above equation then takes the following form

$$f\ x\ =\ F\ f\ x$$

The function $F$ has the type $(A \to B) \to A \to B$. If $g$ is a function of type $A \to B$, $F\ g$ has type $A \to B$, $F\ (F\ g)$ has type $A \to B$ and so on. Thus we can iterate $F$ for a number of times to do the computation, when $F$ is iterated $k$ times it is represented as $F^k$. If the function $f$ takes a $p$ number of recursive calls for the computation $f\ x$, then by construction $F^k\ g\ x = f\ x$, for every $k > p$ and for every function $g$ of right type. The result value $F^k\ g\ x$ does not depend on $g$ for any $k$ larger than $p$, therefore this result value can be defined without defining the function $f$. The recursive function is constructed with the help of a proof by well-founded relation on $A$. The recursive calls are described by a function which corresponds to the induction hypothesis given by the induction step. The termination proof of the recursive function follows the structure of the described computation in the functional $F$.

In fact the function $f$ is a fixpoint of $F$ in the previous equation. This approach makes it easy to obtain a *fix-point equation*, which helps to prove properties of general recursive functions even for weakly specified functions. The companion theorems for the specification of the recursive function is proved with the help of the fix-point equation. This approach of achieving recursion by iteration makes it possible to work separately on the algorithm, the proof of termination and the specification of the recursive function.

This approach of recursion by iteration takes care of separating the computations on the number of iterations from the computations performed on the real arguments, thus making the programming style more closer to the usual functional programming. The first approach of bounded recursion is similar to this approach, but there is an extra computation for the bound. The second approach of well-founded recursion is complex in nature and needs expertise to use. The approach of recursion by iteration is easier to use and the fixpoint equation makes it easier to prove the companion lemmas for the specification of the function. But this approach is not suitable when the termination of the function is not known, in other words the functions are partial. In the chapter 2 we will provide another method to deal with the recursive function where the structural recursion is not adaptable for the specification and the function is partial.

#### 1.2.9.4 Nested recursion

A recursive function $f$ is nested if the function has the following form:

$$f(x) = \ldots f(g(f(y))) \ldots$$

Nested functions are difficult to prove to be total and thus they are difficult to define. In the next chapter we show a technique to describe the nested recursion in programming languages inside type theory.

## 1.3 Conclusion

In the context of programming languages the algebraic types are suitable for syntax and inductive propositions are suitable for natural semantics. Denotational semantics fit with the functional definitions. For denotational semantics we need unrestricted use of recursive functions. Recursion is also common in the context of programming languages. One of the main question studied in this thesis, how do we represent the programming constructs with mutual and nested recursion inside type theory, where the type theory puts the constraint of structural recursion, total functions etc and the programming languages do not inherit them? How do we benefit from the functional descriptions of the programming language? We discuss these questions in the following chapters.

Chapter 2

# Type-theoretic functional semantics

*The work of this chapter is a joint work with Yves Bertot and Venanzio Capretta [5]*

## 2.1 Introduction

In the previous chapter we described two main kinds of semantics for programming languages.

*Operational semantics* consists in describing the steps of the computation of a program by giving formal rules to derive judgments of the form $\langle p, a\rangle \rightsquigarrow r$, to be read as "the program $p$, when applied to the input $a$, terminates and produces the output $r$".

*Denotational semantics* consists in giving a mathematical meaning to data and programs, specifically interpreting data (input and output) as elements of certain domains and programs as functions on those domains; then the fact that the program $p$ applied to the input $a$ gives $r$ as result is expressed by the equality $[\![p]\!]([\![a]\!]) = [\![r]\!]$, where $[\![-]\!]$ is the interpretation.

Our main goal is to develop operational and denotational semantics inside type theory, to implement them in the proof-assistant **Coq** [33], and to prove their main properties formally. In this chapter we will prove a soundness and completeness theorem stating that operational and denotational semantics agree.

The implementation of operational semantics is straightforward: The derivation system is formalized as an inductive relation whose constructors are direct rewording of the derivation rules.

The implementation of denotational semantics is much more delicate. Traditionally, programs are interpreted as partial functions, since they may diverge on certain inputs. However, all function of type theory are total. The problem of representing partial functions in a total setting has been the topic of recent work by several authors [15, 13, 34, 10, 37]. A standard way of solving it is to restrict the domain to those elements that are interpretations of inputs on which the program terminates and then interpret the program as a total function on the restricted domain. There are different approaches to the characterization of the restricted domain. Another approach is to lift the co-domain by adding a bottom element, this approach is not sufficient here because the expressive power of the programming language imposes a limit to computable functions. For example, reduction always terminates in the Calculus of Inductive Constructions, therefore we cannot represent all computable functions through reduction.

When considering the nested recursive function, a direct formalization needs to define domain and function simultaneously. This is not possible in standard type theory, but can be achieved if we extend it with Dybjer's simultaneous induction-recursion [14]. This is the approach adopted in [10].

An alternative way, adopted by Balaa and Bertot in [1], sees the partial function as a fixed point of an operator $F$ that maps total functions to total functions. It can be approximated by a finite number of iterations of $F$ on an arbitrary base function. The domain can be defined as the set of those elements for which the iteration of $F$ stabilizes after a finite number of steps independently of the base function.

The drawback of the approach of [10] is that it is not viable in standard type theories (that is, without Dybjer's schema). The drawback of the approach of [1] is that the defined domain is the domain of a fixed point of $F$ that is not in general the least fixed point. This maybe correct for lazy functional languages (call by name), but is incorrect for strict functional languages (call by value), where we need the least fixed point. The interpretation of an imperative programming language is essentially strict and therefore the domain is too large: The function is defined for values on which the program does not terminate.

Here we combine the two approaches of [10] and [1] by defining the domain in a way similar to that of [10], but disentangling the mutual dependence of domain and function by using the iteration of the functional $F$ with a variable index in place of the yet undefined function.

We claim two main results. First, we develop denotational semantics in type theory. Second, we model the accessibility method in a weaker system, that is, without using simultaneous induction-recursion.

Here is the structure of this chapter. In Section 2.2 we define the simple imperative programming language IMP. We give an informal description of its operational and denotational semantics. We formalize the operational semantics by an inductive relation. We explain the difficulties related to the implementation of the denotational semantics.

In Section 2.3 we describe the iteration method. We point out the difficulty in characterizing the domain of the interpretation function by the convergence of the iterations.

In Section 2.4 we give the denotational semantics using the accessibility method. We combine it with the iteration technique to formalize nested recursion without the use of simultaneous induction-recursion.

All the definitions have been implemented in **Coq** and all the results proved formally in it. We use here an informal mathematical notation, rather than giving **Coq** code. There is a direct correspondence between this notation and the **Coq** formalization. Using the **PCoq** graphical interface [1], we also implemented some of this more intuitive notation. The **Coq** files of the development are on the web [2].

## 2.2 IMP and its semantics

Winskel [38] presents a small programming language IMP with *while* loops. IMP is a simple imperative language with integers, truth values true and false, memory locations to store the integers, arithmetic expressions, boolean expressions and commands. The formation rules are

arithmetic expressions: $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$;

boolean expressions: $b ::= \mathsf{true} \mid \mathsf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1$;

commands: $c ::= \mathsf{skip} \mid X \leftarrow a \mid c_0; c_1 \mid \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \mid \mathsf{while}\ b\ \mathsf{do}\ c$

where $n$ ranges over integers, $X$ ranges over locations, $a$ ranges over arithmetic expressions, $b$ ranges over boolean expressions and $c$ ranges over commands.

We formalize it by three inductive types AExp, BExp, and Command.

For simplicity, we work with natural numbers instead of integers. We do so, as it has no significant importance in the semantics of IMP. Locations are also represented by natural numbers. One should not confuse the natural number denoting a location with the natural number contained in the location. Therefore, in the definition of AExp, we denote the constant value $n$ by Num(n) and the memory location with address $v$ by Loc(v)

We see commands as state transformers, where a state is a map from memory locations to natural numbers. The map is in general partial, indeed it is defined only on a finite number of locations. Therefore, we can represent a state as a list of bindings between memory locations and values. If the same memory location is bound twice in the same state, the most recent binding, that is, the leftmost one, is the valid one.

$$\begin{aligned}&\mathsf{State}\colon \mathbf{Set}\\ &[\,]\colon \mathsf{State}\\ &[\cdot \mapsto \cdot, \cdot]\colon \mathbb{N} \to \mathbb{N} \to \mathsf{State} \to \mathsf{State}\end{aligned}$$

[1]http://www-sop.inria.fr/lemme/pcoq/index.html

[2]http://www-sop.inria.fr/lemme/Kuntal.Das_Barman/imp/

The state $[v \mapsto n, s]$ is the state $s$ with the content of the location $v$ replaced by $n$.

Operational semantics consists in three relations giving meaning to arithmetic expressions, boolean expressions, and commands. Each relation has three arguments: The expression or command, the state in which the expression is evaluated or the command executed, and the result of the evaluation or execution.

$$
\begin{array}{l}
(\langle\cdot,\cdot\rangle_{\mathsf{A}} \rightsquigarrow \cdot)\colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{N} \to \mathbf{Prop} \\
(\langle\cdot,\cdot\rangle_{\mathsf{B}} \rightsquigarrow \cdot)\colon \mathsf{BExp} \to \mathsf{State} \to \mathbb{B} \to \mathbf{Prop} \\
(\langle\cdot,\cdot\rangle_{\mathsf{C}} \rightsquigarrow \cdot)\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \to \mathbf{Prop}
\end{array}
$$

For arithmetic expressions we have that constants are interpreted in themselves, that is, we have axioms of the form

$$\langle \mathsf{Num(n)}, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n$$

for every $n\colon \mathbb{N}$ and $\sigma\colon \mathsf{State}$. Memory locations are interpreted by looking up their values in the state. Consistently with the spirit of operational semantics, we define the lookup operation by derivation rules rather than by a function.

$$\frac{(\mathsf{value\_ind}\ \sigma\ v\ n)}{\langle \mathsf{Loc(v)}, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n}$$

where

$$
\begin{array}{l}
\mathsf{value\_ind}\colon \mathsf{State} \to \mathbb{N} \to \mathbb{N} \to \mathbf{Prop} \\
\mathsf{no\_such\_location}\colon (v\colon \mathbb{N})(\mathsf{value\_ind}\ []\ v\ 0) \\
\mathsf{first\_location}\colon (v, n\colon \mathbb{N}; \sigma\colon \mathsf{State})(\mathsf{value\_ind}\ [v \mapsto n, \sigma]\ v\ n) \\
\mathsf{rest\_locations}\colon\ (v, v', n, n'\colon \mathbb{N}; \sigma\colon \mathsf{State}) \\
\qquad\qquad v \neq v' \to (\mathsf{value\_ind}\ \sigma\ v\ n) \to (\mathsf{value\_ind}\ [v' \mapsto n', \sigma]\ v\ n)
\end{array}
$$

Notice that we assign the value 0 to empty locations, rather that leaving them undefined. This corresponds to giving a default value to uninitialized variables rather than raising an exception.

The operations are interpreted in the obvious way, for example,

$$\frac{\langle a_0, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0 \quad \langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1}{\langle a_0 + a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0 + n_1}$$

where the symbol $+$ is overloaded: $a_0 + a_1$ denotes the arithmetic expression obtained by applying the symbol $+$ to the expressions $a_0$ and $a_1$, $n_0 + n_1$ denotes the sum of the natural numbers $n_0$ and $n_1$.

In short, the operational semantics of arithmetic expressions is defined by the inductive relation

$$
\begin{array}{l}
(\langle\cdot,\cdot\rangle_{\mathsf{A}} \rightsquigarrow \cdot)\colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{N} \to \mathbf{Prop}\\
\mathsf{eval\_Num}\colon (n\colon \mathbb{N}; \sigma\colon \mathsf{State})(\langle \mathsf{Num(n)}, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n)\\
\mathsf{eval\_Loc}\colon (v, n\colon \mathbb{N}; \sigma\colon \mathsf{State})(\mathsf{value\_ind}\ \sigma\ v\ n) \to (\langle \mathsf{Loc(v)}, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n)\\
\mathsf{eval\_Plus}\colon\ (a_0, a_1\colon \mathsf{AExp}; n_0, n_1\colon \mathbb{N}; \sigma\colon \mathsf{State})\\
\qquad\qquad (\langle a_0, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0) \to (\langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0) \to\\
\qquad\qquad (\langle a_0 + a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0 + n_1)\\
\mathsf{eval\_Minus}\colon \cdots\\
\mathsf{eval\_Mult}\colon \cdots
\end{array}
$$

For the subtraction case the cutoff difference is used, that is, $n - m = 0$ if $n \leq m$.

The definition of the operational semantics of boolean expressions is similar and we omit it.

The operational semantics of commands specifies how a command maps states to states. skip is the command that does nothing, therefore it leaves the state unchanged.

$$\frac{}{\langle \mathsf{skip}, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma}$$

The assignment $X \leftarrow a$ evaluates the expression $a$ and then updates the contents of the location $X$ to the value of $a$.

$$\frac{\langle a, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n \qquad \sigma_{[X \mapsto n]} \rightsquigarrow \sigma'}{\langle X \leftarrow a, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma'}$$

where $\sigma_{[X \mapsto n]} \rightsquigarrow \sigma'$ asserts that $\sigma'$ is the state obtained by changing the contents of the location $X$ to $n$ in $\sigma$. It could be realized by simply $\sigma' = [X \mapsto n, \sigma]$. This solution is not efficient, since it duplicates assignments of existing locations and it would produce huge states during computation. A better solution is to look for the value of $X$ in $\sigma$ and change it.

$$
\begin{array}{l}
(\cdot_{[\cdot \mapsto \cdot]} \rightsquigarrow \cdot)\colon \mathsf{State} \to \mathbb{N} \to \mathbb{N} \to \mathsf{State} \to \mathbf{Prop}\\
\mathsf{update\_no\_location}\colon (v, n\colon \mathbb{N})([]_{[v \mapsto n]} \rightsquigarrow [])\\
\mathsf{update\_first}\colon (v, n_1, n_2\colon \mathbb{N}; \sigma\colon \mathsf{State})([v \mapsto n_1, \sigma]_{[v \mapsto n_2]} \rightsquigarrow [v \mapsto n_2, \sigma])\\
\mathsf{update\_rest}\colon\ (v_1, v_2, n_1, n_2\colon \mathbb{N}; \sigma_1, \sigma_2\colon \mathbb{N}) v_1 \neq v_2 \to\\
\qquad\qquad (\sigma_{1[v_2 \mapsto n_2]} \rightsquigarrow \sigma_2) \to ([v_1 \mapsto n_1, \sigma_1]_{[v_2 \mapsto n_2]} \rightsquigarrow [v_1 \mapsto n_1, \sigma_2])
\end{array}
$$

Notice that we require a location to be already defined in the state to update it. If we try to update a location not present in the state, we leave the state unchanged. This corresponds to requiring that all variables are explicitly initialized before the execution of the program. If we use an uninitialized variable in the program, we do not get an error message, but an anomalous behavior: The value of the variable is always zero.

Evaluating a sequential composition $c_1; c_2$ on a state $\sigma$ consists in evaluating $c_1$ on $\sigma$, obtaining a new state $\sigma_1$, and then evaluating $c_2$ on $\sigma_1$ to obtain the final state $\sigma_2$.

$$\frac{\langle c_1, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1 \quad \langle c_2, \sigma_1\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}{\langle c_1; c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}$$

Evaluating conditionals uses two rules. In both rules, we evaluate the boolean expression $b$, but they differ on the value returned by this step and the sub-instruction that is executed.

$$\frac{\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true} \quad \langle c_1, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1} \qquad \frac{\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false} \quad \langle c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}$$

As for conditionals, we have two rules for *while* loops. If $b$ evaluates to $\mathsf{true}$, $c$ is evaluated on $\sigma$ to produce a new state $\sigma'$, on which the loop is evaluated recursively. If $b$ evaluates to $\mathsf{false}$, we exit the loop leaving the state unchanged.

$$\frac{\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true} \quad \langle c, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma' \quad \langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma'\rangle_{\mathsf{C}} \rightsquigarrow \sigma''}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma''} \qquad \frac{\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma}$$

The above rules can be formalized in **Coq** in a straightforward way by an inductive relation.

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle\cdot,\cdot\rangle_{\mathsf{C}} \rightsquigarrow \cdot \colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \to \mathbf{Prop}} \\
\multicolumn{2}{l}{\mathsf{eval\_skip}\colon (\sigma\colon \mathsf{State})(\langle \mathsf{skip}, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma)} \\
\mathsf{eval\_assign}\colon & (\sigma, \sigma'\colon \mathsf{State}; v, n\colon \mathbb{N}; a\colon \mathsf{AExp}) \\
& (\langle a, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n) \to (\sigma_{[v\mapsto n]} \rightsquigarrow \sigma') \to (\langle v \leftarrow a, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma') \\
\mathsf{eval\_scolon}\colon & (\sigma, \sigma_1, \sigma_2\colon \mathsf{State}; c_1, c_2\colon \mathsf{Command}) \\
& (\langle c_1, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \to (\langle c_2, \sigma_1\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \to (\langle c_1; c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \\
\mathsf{eval\_if\_true}\colon & (b\colon \mathsf{BExp}; \sigma, \sigma_1\colon \mathsf{State}; c_1, c_2\colon \mathsf{Command}) \\
& (\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \to (\langle c_1, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \to \\
& (\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \\
\mathsf{eval\_if\_false}\colon & (b\colon \mathsf{BExp}; \sigma, \sigma_2\colon \mathsf{State}; c_1, c_2\colon \mathsf{Command}) \\
& (\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \to (\langle c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \to \\
& (\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \\
\mathsf{eval\_while\_true}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma, \sigma', \sigma''\colon \mathsf{State}) \\
& (\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \to (\langle c, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma') \to \\
& (\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma'\rangle_{\mathsf{C}} \rightsquigarrow \sigma'') \to (\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma'') \\
\mathsf{eval\_while\_false}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}) \\
& (\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \to (\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma)
\end{array}$$

For the rest of the paper we leave out the subscripts $\mathsf{A}$, $\mathsf{B}$, and $\mathsf{C}$ in $\langle\cdot,\cdot\rangle \rightsquigarrow \cdot$.

## 2.3 Functional interpretation

Denotational semantics consists in interpreting program evaluation as a function rather than as a relation. We start by giving a functional interpretation to expression evaluation and state update. This is quite straightforward, since we can use structural recursion on expressions and states. For example, the interpretation function on arithmetic expressions is defined as

$$\begin{array}{l}
[\![\cdot]\!] \colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{N} \\
[\![\mathsf{Num(n)}]\!]_\sigma := n \\
[\![\mathsf{Loc(v)}]\!]_\sigma := \mathsf{value\_rec}(\sigma, v) \\
[\![a_0 + a_1]\!]_\sigma := [\![a_0]\!]_\sigma + [\![a_1]\!]_\sigma \\
[\![a_0 - a_1]\!]_\sigma := [\![a_0]\!]_\sigma - [\![a_1]\!]_\sigma \\
[\![a_0 * a_1]\!]_\sigma := [\![a_0]\!]_\sigma \cdot [\![a_1]\!]_\sigma
\end{array}$$

where $\mathsf{value\_rec}(\cdot,\cdot)$ is the function giving the contents of a location in a state, defined by recursion on the structure of the state. It differs from $\mathsf{value\_ind}$ because it is a function, not a relation; $\mathsf{value\_ind}$ is its graph. We can now prove that this interpretation function agrees with the operational semantics given by the inductive relation $\langle\cdot,\cdot\rangle \rightsquigarrow \cdot$ (all the lemmas and theorems given below have been checked in a computer-assisted proof).

**Lemma 2.1.** $\forall\sigma \colon \mathsf{State}.\forall a \colon \mathsf{AExp}.\forall n \colon \mathbb{N}.\langle\sigma, a\rangle \rightsquigarrow n \Leftrightarrow [\![a]\!]_\sigma = n.$

In the same way, we define the interpretation of boolean expressions

$$[\![\cdot]\!] \colon \mathsf{BExp} \to \mathsf{State} \to \mathbb{B}$$

and prove that it agrees with the operational semantics.

**Lemma 2.2.** $\forall\sigma \colon \mathsf{State}.\forall b \colon \mathsf{BExp}.\forall t \colon \mathbb{B}.\langle\sigma, b\rangle \rightsquigarrow t \Leftrightarrow [\![a]\!]_\sigma = t.$

We overload the Scott brackets $[\![\cdot]\!]$ to denote the interpretation function both on arithmetic and boolean expressions (and later on commands).

Similarly, we define the update function

$$\cdot[\cdot/\cdot] \colon \mathsf{State} \to \mathbb{N} \to \mathbb{N} \to \mathsf{State}$$

and prove that it agrees with the update relation

**Lemma 2.3.** $\forall\sigma, \sigma' \colon \mathsf{State}.\forall v, n \colon \mathbb{N}.\sigma_{[v \mapsto n]} \rightsquigarrow \sigma' \Leftrightarrow \sigma[n/v] = \sigma'.$

The next step is to define the interpretation function $[\![\cdot]\!]$ on commands. Unfortunately, this cannot be done by structural recursion, as for the cases of arithmetic and boolean expressions. Indeed we should have

$$\begin{aligned}
&[\![\cdot]\!] : \mathsf{Command} \to \mathsf{State} \to \mathsf{State}\\
&[\![\mathsf{skip}]\!]_\sigma := \sigma\\
&[\![X \leftarrow a]\!]_\sigma := \sigma[[\![a]\!]_\sigma / X]\\
&[\![c_1; c_2]\!]_\sigma := [\![c_1]\!]_{[\![c_2]\!]_\sigma}\\
&[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma := \begin{cases} [\![c_1]\!]_\sigma & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true}\\ [\![c_2]\!]_\sigma & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false}\end{cases}\\
&[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma := \begin{cases} [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_{[\![c]\!]_\sigma} & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true}\\ \sigma & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false}\end{cases}
\end{aligned}$$

but in the clause for *while* loops the interpretation function is called on the same argument if the boolean expression evaluates to true. Therefore, the argument of the recursive call is not structurally smaller than the original argument.

So, it is not possible to associate a structural recursive function to the instruction execution relation as we did for the lookup, update, and expression evaluation relations. The execution of *while* loops does not respect the pattern of structural recursion and termination cannot be ensured: for good reasons too, since the language is Turing complete. We describe a way to work around this problem.

### 2.3.1 The iteration technique

A function representation of the computation can be provided in a way that respects typing and termination if we don't try to describe the execution function itself but the *second order function of which the execution function is the least fixed point.* This function can be defined in type theory by cases on the structure of the command.

$$\begin{aligned}
&\mathsf{F} : (\mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \to \mathsf{Command} \to \mathsf{State} \to \mathsf{State}\\
&(\mathsf{F}\ f\ \mathsf{skip}\ \sigma) := \sigma\\
&(\mathsf{F}\ f\ (X \leftarrow a)\ \sigma) := \sigma[[\![a]\!]_\sigma / X]\\
&(\mathsf{F}\ f\ (c_1; c_2)\ \sigma) := (f\ c_2\ (f\ c_1\ \sigma))\\
&(\mathsf{F}\ f\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma) := \begin{cases} (f\ c_1\ \sigma) & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true}\\ (f\ c_2\ \sigma) & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false}\end{cases}\\
&(\mathsf{F}\ f\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma) := \begin{cases} (f\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ (f\ c\ \sigma)) & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true}\\ \sigma & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false}\end{cases}
\end{aligned}$$

Intuitively, writing the function F is exactly the same as writing the recursive execution function, except that the function being defined is simply replaced by a bound variable (here $f$). In other words, we replace recursive calls with calls to the function given in the bound variable $f$.

The function F describes the computations that are performed at each iteration of the execution function and the execution function performs the same computation as the function $\mathsf{F}$ when the latter is repeated *as many times as needed.* We can express this with the following theorem.

**Theorem 2.1** (eval_com_ind_to_rec)**.**

$$\begin{aligned}&\forall c\colon \mathsf{Command}.\forall \sigma_1, \sigma_2\colon \mathsf{State}.\\ &\langle c, \sigma_1\rangle \leadsto \sigma_2 \Rightarrow \exists k\colon \mathbb{N}.\forall g\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{F}^k\ g\ c\ \sigma_1) = \sigma_2\end{aligned}$$

*where we used the following notation*

$$\mathsf{F}^k = (\mathsf{iter}\ (\mathsf{Command} \to \mathsf{State} \to \mathsf{State})\ \mathsf{F}\ k) = \lambda g.\underbrace{(\mathsf{F}\ (\mathsf{F}\ \cdots\ (\mathsf{F}}_{k\ times}\ g)\ \cdots\ ))$$

*definable by recursion on k,*

$$\begin{aligned}&\mathsf{iter}\colon (A\colon \mathbf{Set})(A \to A) \to \mathbb{N} \to A \to A\\ &(\mathsf{iter}\ A\ f\ 0\ a) := a\\ &(\mathsf{iter}\ A\ f\ (\mathsf{S}\ k)\ a) := (f\ (\mathsf{iter}\ A\ f\ k\ a)).\end{aligned}$$

*Proof.* Easily proved using the theorems described in the previous section and an induction on the derivation of $\langle c, \sigma_1\rangle \leadsto \sigma_2$: This kind of induction is also called *rule induction* in [38]. $\square$

Note that our definition of iteration is different from the denotational semantics, as we do not check for definiteness. We call this semantics as functional semantics. In our functional semantics we separate the two kind of information that the denotational semantics gathers in the same object, namely the definiteness information and the resulting value. In the section 2.4 we show how we include this definiteness information in our functional semantics.

### 2.3.2 Extracting an interpreter

The **Coq** system provides an *extraction* facility [28], which makes it possible to produce a version of any function defined in type theory that is written in a functional programming language's syntax, usually the **OCaml** implementation of ML. In general, the extraction facility performs some complicated program manipulations, to ensure that arguments of functions that have only a logical content are not present anymore in the extracted code. For instance, a division function is a 3-argument function inside type theory: The first argument is the number to be divided, the second is the divisor, and the third is a proof that the second is non-zero. In the extracted code, the function takes only two arguments: The extra argument does not interfere with the computation and its presence cannot help ensuring typing, since the programming language's type system is too weak to express this kind of details.

The second order function $\mathsf{F}$ and the other recursive functions can also be extracted to ML programs using this facility. However, the extraction process is a simple translation process in this case, because none of the various function actually takes proof arguments.

To perform complete execution of programs, using the ML translation of $\mathsf{F}$, we have the possibility to compute using the extracted version of the $\mathsf{iter}$ function. However, we need to guess the right value for the $k$ argument. One way to cope with this is to create an artificial "infinite" natural number, that will always appear to be big enough, using the following recursive data definition:

$$\mathsf{letrec}\ \omega = (\mathsf{S}\ \omega).$$

This definition does not correspond to any natural number that can be manipulated inside type theory: It is an infinite tree composed only of $\mathsf{S}$ constructors. In memory, it corresponds to an $\mathsf{S}$ construct whose only field points to the whole construct: It is a loop.

Using the extracted $\mathsf{iter}$ with $\omega$ is not very productive. Since ML evaluates expressions with a call-by-value strategy, evaluating

$$(\mathsf{iter}\ \mathsf{F}\ g\ \omega\ c\ \sigma)$$

imposes that one evaluates

$$(\mathsf{F}\ (\mathsf{iter}\ \mathsf{F}\ g\ \omega)\ c\ \sigma)$$

which in turn imposes that one evaluates

$$(\mathsf{F}\ (\mathsf{F}\ (\mathsf{iter}\ \mathsf{F}\ g\ \omega))\ c\ \sigma)$$

and so on. Recursion unravels unchecked and this inevitably ends with a stack overflow error. However, it is possible to use a variant of the iteration function that avoids this infinite looping, even for a call-by-value evaluation strategy. The trick is to $\eta$-expand the expression that provokes the infinite loop, to force the evaluator to stop until an extra value is provided, before continuing to evaluate the iterator. The expression to define this variant is as follows:

$$\begin{array}{l}\mathsf{iter}'\colon (A, B\colon \mathbf{Set})((A \to B) \to A \to B) \to \mathbb{N} \to (A \to B) \to A \to B\\ (\mathsf{iter}'\ A\ B\ G\ 0\ f) := f\\ (\mathsf{iter}'\ A\ B\ G\ (\mathsf{S}\ k)\ f) := (G\ \lambda a\colon A.(\mathsf{iter}'\ A\ B\ G\ k\ f\ a))\end{array}$$

Obviously, the expression $\lambda a\colon A.(\mathsf{iter}'\ A\ B\ G\ k\ f\ a)$ is $\eta$-equivalent to the expression $(\mathsf{iter}'\ A\ B\ G\ k\ f)$. However, for call-by-value evaluation the two expression are not equivalent, since the $\lambda$-expression in the former stops the evaluation process that would lead to unchecked recursion in the latter.

With the combination of $\mathsf{iter}'$ and $\omega$ we can now execute any terminating program without needing to compute in advance the number of iterations of

$\mathsf{F}$ that will be needed. In fact, $\omega$ simply acts as a *natural number that is big enough.* We obtain a functional interpreter for the language we are studying, that is (almost) proved correct with respect to the inductive definition $\langle\cdot,\cdot\rangle \rightsquigarrow \cdot$.

Still, the use of $\omega$ as a natural number looks rather like a dirty trick: This piece of data cannot be represented in type theory, and we are taking advantage of important differences between type theory and ML's memory and computation models: How can we be sure that what we proved in type theory is valid for what we execute in ML? A first important difference is that, while executions of $\mathsf{iter}$ or $\mathsf{iter}'$ are sure to terminate in type theory, $(\mathsf{iter}'\ \mathsf{F}\ \omega\ g)$ will loop if the program passed as argument is a looping program.

The purpose of using $\omega$ and $\mathsf{iter}'$ is to make sure that $\mathsf{F}$ will be called as many times as needed when executing an arbitrary program, with the risk of non-termination when the studied program does not terminate. This can be done more easily by using a *fixpoint* function that simply returns the fixpoint of $\mathsf{F}$. This fixpoint function is defined in ML by

$$\mathsf{letrec}\ (\mathsf{fix}\ f) = f(\lambda x.\mathsf{fix}\ f\ x).$$

Obviously, we have again used the trick of $\eta$-expansion to avoid looping in the presence of a call-by-value strategy. With this $\mathsf{fix}$ function, the interpreter function is

$$\begin{array}{l}\mathsf{interp}\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}\\ \mathsf{interp} := \mathsf{fix}\ \mathsf{F}.\end{array}$$

To obtain a usable interpreter, it is then only required to provide a parser and printing functions to display the results of evaluation. This shows how we can build an interpreter for IMP in ML. But we realized it by using some tricks of functional programming that are not available in type theory. If we want to define an interpreter for IMP in type theory, we have to find a better solution to the problem of partiality.

### 2.3.3 Characterizing terminating programs

Theorem **2.1** gives one direction of the correspondence between operational semantics and functional interpretation through the iteration method. To complete the task of formalizing denotational semantics, we need to define a function in type theory that interprets each command. As we already remarked, this function cannot be total, therefore we must first restrict its domain to the terminating commands. This is done by defining a predicate $D$ over commands and states, and then defining the interpretation function $[\![\cdot]\!]$ on the domain restricted by this predicate. Theorem **2.1** suggests the following definition:

$$\begin{array}{ll}D\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Prop} & \\ (D\ c\ \sigma) := & \exists k\colon \mathbb{N}.\forall g_1, g_2\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.\\ & \quad (\mathsf{F}^k\ g_1\ c\ \sigma) = (\mathsf{F}^k\ g_2\ c\ \sigma).\end{array}$$

Unfortunately, this definition is too weak. In general, such an approach cannot be used to characterize terminating "nested" iteration. This is hard to see in the case of the IMP language, but it would appear plainly if one added an exception instruction with the following semantics:

$$\langle \mathsf{exception}, \sigma\rangle \rightsquigarrow [].$$

Intuitively, the programmer could use this instruction to express that an exceptional situation has been detected, but all information about the execution state would be destroyed when this instruction is executed.

With this new instruction, there are some commands and states for which the predicate $D$ is satisfied, but whose computation does not terminate.

$$c := \mathsf{while\ true\ do\ skip; exception}.$$

It is easy to see that for any state $\sigma$ the computation of $c$ on $\sigma$ does not terminate. In terms of operational semantics, for no state $\sigma'$ is the judgment $\langle c, \sigma\rangle \rightsquigarrow \sigma'$ derivable.

However, $(D\ c\ \sigma)$ is provable, because $(\mathsf{F}^k\ g\ c\ \sigma) = []$ for any $k > 1$.

In the next section we work out a stronger characterization of the domain of partial functions, that turn out to be the correct one in which to interpret the operational semantics.

## 2.4 The Accessibility predicate

A common way to represent partial functions in type theory is to restrict their domain to those arguments on which they terminate. A partial function $f\colon A \rightharpoonup B$ is then represented by first defining a predicate $D_f\colon A \to \mathbf{Prop}$ that characterizes the domain of $f$, that is, the elements of $A$ on which $f$ is defined; and then formalizing the function itself as $f\colon (\Sigma x\colon A.(D_f\ x)) \to B$, where $\Sigma x\colon A.(D_f\ x)$ is the type of pairs $\langle x, h\rangle$ with $x\colon A$ and $h\colon (D_f\ x)$.

The predicate $D_f$ cannot be defined simply by saying that it is the domain of definition of $f$, since, in type theory, we need to define it before we can define $f$. Therefore, $D_f$ must be given before and independently from $f$. One way to do it is to characterize $D_f$ as the predicate satisfied by those elements of $A$ for which the iteration technique converges to the same value for every initial function. This is a good definition when we try to model lazy functional programming languages, but, when interpreting strict programming languages or imperative languages, we find that this predicate would be too weak, being satisfied by elements for which the associated program diverges, as we have seen at the end of the previous section.

Sometimes the domain of definition of a function can be characterized independently of the function by an inductive predicate called *accessibility* [29, 15, 13, 9]. This simply states that an element of $a$ can be proved to be in the domain if the application of $f$ on $a$ calls $f$ recursively on elements that have already been proved to be in the domain. For example, if in the recursive definition of $f$ there is a clause of the form

$$f(e) := \cdots f(e_1) \cdots f(e_2) \cdots$$

and $a$ matches $e$, that is, there is a substitution of variables $\rho$ such that $a = \rho(e)$; then we add a clause to the inductive definition of Acc of type

$$\mathsf{Acc}(e_1) \to \mathsf{Acc}(e_2) \to \mathsf{Acc}(e).$$

This means that to prove that $a$ is in the domain of $f$, we must first prove that $\rho(e_1)$ and $\rho(e_2)$ are in the domain.

This definition does not always work. In the case of nested recursive calls of the function, we cannot eliminate the reference to $f$ in the clauses of the inductive definition Acc. If, for example, the recursive definition of $f$ contains a clause of the form

$$f(e) := \cdots f(f(e')) \cdots$$

then the corresponding clause in the definition of Acc should be

$$\mathsf{Acc}(e') \to \mathsf{Acc}(f(e')) \to \mathsf{Acc}(e)$$

because we must require that all arguments of the recursive calls of $f$ satisfy Acc to deduce that also $e$ does. But this definition is incorrect because we haven't defined the function $f$ yet and so we cannot use it in the definition of Acc. Besides, we need Acc to define $f$, therefore we are locked in a vicious circle.

In our case, we have two instances of nested recursive clauses, for the sequential composition and *while* loops. When trying to give a semantics of the commands, we come to the definition

$$[\![c_1; c_2]\!]_\sigma := [\![c_2]\!]_{[\![c_1]\!]_\sigma}$$

for sequential composition and

$$[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma := [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_{[\![c]\!]_\sigma}$$

for a *while* loop, if the interpretation of $b$ in state $\sigma$ is $\mathsf{true}$.

Both cases contain a nested occurrence of the interpretation function $[\![-]\!]$.

An alternative solution, presented in [10], exploits the extension of type theory with simultaneous induction-recursion [14]. In this extension, an inductive type or inductive family can be defined simultaneously with a function on it. For the example above we would have

$$
\begin{array}{l}
\mathsf{Acc}\colon A \to \mathbf{Prop} \\
f\colon (x\colon A)(\mathsf{Acc}\ x) \to B \\
\vdots \\
\mathsf{acc}_n\colon (h'\colon (\mathsf{Acc}\ e'))(\mathsf{Acc}\ (f\ e'\ h')) \to (\mathsf{Acc}\ e) \\
\vdots \\
(f\ e\ (\mathsf{acc}_n\ h'\ h)) := \cdots (f\ (f\ e'\ h)\ h) \cdots \\
\vdots
\end{array}
$$

This method leads to the following definition of the accessibility predicate and interpretation function for the imperative programming language IMP:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{comAcc}\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Prop}} \\
\multicolumn{2}{l}{[\![]\!]\colon (c\colon \mathsf{Command}; \sigma\colon \mathsf{State})(\mathsf{comAcc}\ c\ \sigma) \to \mathsf{State}} \\
\\
\mathsf{accSkip}\colon & (\sigma\colon \mathsf{State})(\mathsf{comAcc}\ \mathsf{skip}\ \sigma) \\
\mathsf{accAssign}\colon & (v\colon \mathbb{N}; a\colon \mathsf{AExp}; \sigma\colon \mathsf{State})(\mathsf{comAcc}\ (v \leftarrow a)\ \sigma) \\
\mathsf{accScolon}\colon & (c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; h_1\colon (\mathsf{comAcc}\ c_1\ \sigma)) \\
& (\mathsf{comAcc}\ c_2\ [\![c_1]\!]_\sigma^{h_1}) \to (\mathsf{comAcc}\ (c_1; c_2)\ \sigma) \\
\mathsf{accIf\_true}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}) \\
& [\![b]\!]_\sigma = \mathsf{true} \to (\mathsf{comAcc}\ c_1\ \sigma) \\
& \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma) \\
\mathsf{accIf\_false}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}) \\
& [\![b]\!]_\sigma = \mathsf{false} \to (\mathsf{comAcc}\ c_2\ \sigma) \\
& \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma) \\
\mathsf{accWhile\_true}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!] = \mathsf{true} \\
& \to (h\colon (\mathsf{comAcc}\ c\ \sigma))(\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ [\![c]\!]_\sigma^h) \\
& \to (\mathsf{comAcc}(\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma) \\
\mathsf{accWhile\_false}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!] = \mathsf{false} \\
& \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma)
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{skip}]\!]_\sigma^{(\mathsf{accSkip}\ \sigma)} := \sigma \\
[\![(v := a)]\!]_\sigma^{(\mathsf{accAssign}\ v\ a\ \sigma)} := \sigma[a/v] \\
[\![(c_1; c_2)]\!]_\sigma^{(\mathsf{accScolon}\ c_1\ c_2\ \sigma\ h_1\ h_2)} := [\![c_2]\!]^{h_2}_{[\![c_1]\!]_\sigma^{h_1}} \\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma^{(\mathsf{accIf\_true}\ b\ c_1\ c_2\ \sigma\ p\ h_1)} := [\![c_1]\!]_\sigma^{h_1} \\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma^{(\mathsf{accIf\_false}\ b\ c_1\ c_2\ \sigma\ q\ h_2)} := [\![c_2]\!]_\sigma^{h_2} \\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma^{(\mathsf{accWhile\_true}\ b\ c\ \sigma\ p\ h\ h')} := [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]^{h'}_{[\![c]\!]_\sigma^h} \\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma^{(\mathsf{accWhile\_false}\ b\ c\ \sigma\ q)} := \sigma
\end{array}
$$

This definition is admissible in systems that implement Dybjer's schema for simultaneous induction-recursion. But on systems that do not provide such schema, for example **Coq**, this definition is not valid.

We must disentangle the definition of the accessibility predicate from the definition of the evaluation function. As we have seen before, the evaluation function can be seen as the limit of the iteration of the functional $F$ on an arbitrary base function $f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}$. Whenever the evaluation of a command $c$ is defined on a state $\sigma$, we have that $[\![c]\!]_\sigma$ is equal to $(F_f^k\ c\ \sigma)$ for a sufficiently large number of iterations $k$. Therefore, we consider the functions $F_f^k$ as approximations to the interpretation function being defined. We can formulate the accessibility predicate by using such approximations in place of the explicit occurrences of the evaluation function. Since the iteration approximation has two extra parameters, the number of iterations $k$ and the base function $f$, we must also add them as new arguments of comAcc. The resulting inductive definition is

$$
\begin{array}{ll}
\mathsf{comAcc}\colon \mathsf{Command} \to \mathsf{State} \to \mathbb{N} \to (\mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \to \mathbf{Prop} & \\
\mathsf{accSkip}\colon (\sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) & \\
\qquad\qquad (\mathsf{comAcc}\ \mathsf{skip}\ \sigma\ k+1\ f) & \\
\mathsf{accAssign}\colon & (v\colon \mathbb{N}; a\colon \mathsf{AExp}; \sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \\
& (\mathsf{comAcc}\ (v \leftarrow a)\ \sigma\ k+1\ f) \\
\mathsf{accScolon}\colon & (c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon (\mathsf{Command} \to \mathsf{State} \to \mathsf{State})) \\
& (\mathsf{comAcc}\ c_1\ \sigma\ k\ f) \to (\mathsf{comAcc}\ c_2\ (F_f^k\ c_1\ \sigma)\ k\ f) \\
& \to (\mathsf{comAcc}\ (c_1; c_2)\ \sigma\ k+1\ f) \\
\mathsf{accIf\_true}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \\
& (\langle b, \sigma\rangle \leadsto \mathsf{true}) \to (\mathsf{comAcc}\ c_1\ \sigma\ k\ f) \\
& \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma\ k+1\ f) \\
\mathsf{accIf\_false}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \\
& (\langle b, \sigma\rangle \leadsto \mathsf{false}) \to (\mathsf{comAcc}\ c_2\ \sigma\ k\ f) \\
& \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma\ k+1\ f) \\
\mathsf{accWhile\_true}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \leadsto \mathsf{true}) \\
& \to (\mathsf{comAcc}\ c\ \sigma\ k\ f) \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ (F_f^k\ c\ \sigma)) \\
& \to (\mathsf{comAcc}(\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma\ k+1\ f) \\
\mathsf{accWhile\_false}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \leadsto \mathsf{false}) \\
& \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma\ k+1\ f).
\end{array}
$$

This accessibility predicate characterizes the points in the domain of the program parametrically on the arguments $k$ and $f$. To obtain an independent definition of the domain of the evaluation function we need to quantify on them. We quantify existentially on $k$, because if a command $c$ and a state $\sigma$ are accessible in $k$ steps, then they will still be accessible in a higher number of steps. We quantify universally on $f$ because we do not want the result of the computation to depend on the choice of the base function.

$$\begin{array}{l}\mathsf{comDom}\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Set}\\ (\mathsf{comDom}\ c\ \sigma) = \Sigma k\colon \mathbb{N}.\forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{comAcc}\ c\ \sigma\ k\ f)\end{array}$$

The reason why the sort of the predicate comDom is **Set** and not **Prop** is that we need to extract the natural number $k$ from the proof to be able to compute the following evaluation function:

$$\begin{array}{l}[\![]\!]\colon\ (c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})\\ \qquad (\mathsf{comDom}\ c\ \sigma) \to \mathsf{State}\\ [\![c]\!]_{\sigma,f}^{\langle k,h\rangle} = (F_f^k\ c\ \sigma)\end{array}$$

To illustrate the meaning of these definitions, let us see how the interpretation of a sequential composition of two commands is defined. The interpretation of the command $(c_1; c_2)$ on the state $\sigma$ is $[\![c_1; c_2]\!]_\sigma^H$, where $H$ is a proof of $(\mathsf{comDom}\ (c_1; c_2)\ \sigma)$. Therefore $H$ must be in the form $\langle k, h\rangle$, where $k\colon \mathbb{N}$ and $h\colon \forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{comAcc}\ (c_1; c_2)\ \sigma\ k\ f)$. To see how $h$ can be constructed, let us assume that $f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}$ and prove $(\mathsf{comAcc}\ (c_1; c_2)\ \sigma\ k\ f)$. This can be done only by using the constructor accScolon. We see that it must be $k = k' + 1$ for some $k'$ and we must have proofs $h_1\colon (\mathsf{comAcc}\ c_1\ \sigma\ k'\ f)$ and $h_2\colon (\mathsf{comAcc}\ c_2\ (F_f^{k'}\ c_1\ \sigma)\ k'\ f)$. Notice that in $h_2$ we don't need to refer to the evaluation function $[\![]\!]$ anymore, and therefore the definitions of comAcc does not depend on the evaluation function anymore. We have now that $(h\ f) := (\mathsf{accScolon}\ c_1\ c_2\ \sigma\ k'\ f\ h_1\ h_2)$. The definition of $[\![c_1; c_2]\!]_\sigma^H$ is also not recursive anymore, but consists just in iterating $F$ $k$ times, where $k$ is obtained from the proof $H$.

We can now prove an exact correspondence between operational semantics and denotational semantics given by the interpretation operator $[\![\cdot]\!]$.

**Theorem 2.2.**

$$\begin{array}{l}\forall c\colon \mathsf{Command}.\forall \sigma, \sigma'\colon \mathsf{State}.\\ \langle c, \sigma\rangle \leadsto \sigma' \Leftrightarrow \exists H\colon (\mathsf{comDom}\ c\ \sigma).\forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.[\![c]\!]_{\sigma,f}^H = \sigma'.\end{array}$$

*Proof.* From left to right, it is proved by rule induction on the derivation of $\langle c, \sigma\rangle \leadsto \sigma'$. The number of iterations $k$ is the depth of the proof and the proof of the comAcc predicate is a translation step by step of it. From right to left, it is proved by induction on the proof of comAcc. □

## 2.5 Conclusions

The combination of the iteration technique and the accessibility predicate has, in our opinion, a vast potential that goes beyond its application to denotational semantics. Not only does it provide a path to the implementation and reasoning about partial and nested recursive functions that does not require simultaneous induction-recursion; but it gives a finer analysis of convergence of recursive operators. As we pointed out in Section 2.3, it supplies not just any fixed point of an operator, but the least fixed point.

We were not the first to formalize parts of Winskel's book in a proof system. Nipkow [26] formalized the first 100 pages of it in ISABELLE/HOL. The main difference between our work and his, is that he does not represent the denotation as a function but as a subset of $\mathsf{State} \times \mathsf{State}$ that happens to be the graph of a function. Working on a well developed library on sets, he has no problem in using a least-fixpoint operator to define the subset associated to a *while* loop: But this approach stays further removed from functional programming than an approach based directly on the functions provided by the prover. In this respect, our work is the first to reconcile a theorem proving framework with total functions with denotational semantics. One of the gains is directly executable code (through extraction or $\iota$-reduction). The specifications provided by Nipkow are only executable in the sense that they all belong to the subset of inductive properties that can be translated to PROLOG programs. In fact, the reverse process has been used and those specifications had all been obtained by a translation from a variant of PROLOG to a theorem prover [7]. However, the prover's functions had not been used to represent the semantics.

Our method tries to maximize the potential for automation: Given a recursive definition, the functional operator $\mathsf{F}$, the iterator, the accessibility predicate, the domain, and the evaluation function can all be generated automatically. Moreover, it is possible to automate the proof of the accessibility predicate, since there is only one possible proof step for any given argument; and the obtained evaluation function is computable inside type theory.

We expect this method to be widely used in the future in several areas of formalization of mathematics in type theory. A direct application of our method will be discussed in chapter 4, during the formalization of Cminor, a subset of the language C, in the Calculus of Inductive Constructions.

Chapter 3

# Proof by reflection in semantics

## 3.1 Introduction

In the previous chapter we have seen that in the context of theorem proving, there are two ways to describe the semantics of a programming language. The most commonly used, operational semantics, relies on inductive definitions. The execution of the program statements is represented by a proposition relating a program, its input and its output. Universally quantified logical formulas, presented as inference rules, are provided to describe under which conditions a given program fragment executes correctly. Proving that a given program maps a given input to a given output in a given context corresponds to showing that the proposition relating this program, input, and output is a consequence of the inference rules and the context.

An alternative approach is to describe the programming language semantics as a function mapping programs and inputs to outputs. This function can be used to compute the result of executing a given program on a given input, but it is not suitable to reason on programs using information coming from the context. There is a way to make the functional approach more powerful, so that it also uses the context.

We talk about functional semantics rather than denotational semantics because our work does not come with the usual background on domain theory or complete partial orders.

To simulate the execution of a program using the operational semantics, we need to combine this semantics with a proof search procedure. To simulate the execution of a program using the functional semantics, we only need to apply a function to the program and inputs and reduce it to the output. In this sense, the functional semantics opens the door to proof by reflection [8, 36, 21] because it makes it possible to represent both the semantics and the proof procedure. But the proof tool that we obtain is still rather weak, because it can reduce to final value only for ground programs and it does not use the context. Our goal is to obtain a proof procedure that is more powerful than conventional proof search. The most important result is a technique which helps to reason on metavariables, in other words, symbolically represented expressions and instructions. This technique is systematic and general.

Common methods to automate the proof search are based on unification and resolution. A proof can be viewed as a goal to solve, given a context of hypotheses. A unification and resolution based procedure looks into the local context and tries to match the current goal against the conclusion of one of the hypotheses. If it succeeds, then it returns a subgoal for each of the premises of the matched hypothesis.

The drawbacks of this approach are, first, that the general proof strategy is not focused and loses time in exploring a large search space and second, that it does not have computation power. So we can arrive in a situation where, even if we have enough information to execute an instruction we won't be able to execute. We will show such an example in the next section.

In type theory based proof assistants like **Coq** [33] functions are provided and reductions are used to compute with them. Functions compute on data objects. In usual formal proofs the facts related to computations are provided as assertions, which are in fact relations between data objects. We use reflection to bridge this gap to use functions in proof search. We collect the data objects from the given facts and put them in several tables. Unlike unification and resolution based approach, we do not look for a match for a hypothesis in the context to do the computation, we consult these tables and use functions to do the computation. This functional approach is more focused and does not need to search in the entire search space, as we can directly consult the particular table related to the enquiry. Function evaluation is performed by term reduction and reasoning on unknown expression needs special care. In this chapter we show a way to work around this problem.

In this chapter we claim two main results. First, using reflection and a functional approach to automate proof search we produce an easier and better way than the currently available unification and resolution based technique in proof automation. Second, more important, we present another general and systematic way to reason on unknown expressions, which is different from resolution techniques and thereby facilitating symbolic computations (or computations involving metavariables) inside type theory.

Here is the structure of this chapter. In the section 3.2 we consider the simple imperative language IMP, which we have already discussed in the section 2.2 and the formalization of its operational semantics by inductive relations. We discuss about our main objective, a step towards automation, giving a look into the current situation. We show the difficulties with the current solution. In Section 3.3 we present the functional interpretations of IMP. We show how to collect and use data objects from the provided facts to execute instructions in a functional approach. In Section 3.4 we give the idea of reflection and how it will be used in our context. We provide a systematic and general technique to work with metavariables. To implement the function which does not follow structural recursion we use *iteration technique*, which we have described in section 2.3.

All the definitions have been implemented in **Coq** and all the results proved formally in it. We use here an informal mathematical notation, rather than giving **Coq** code. There is a direct correspondence between this notation and the **Coq** formalization. Using the **PCoq** graphical interface (available on the web[1]), we also implemented some of this more intuitive notation. The **Coq** files of the development are on the web[2].

## 3.2 IMP and its semantics

In the previous chapter we presented a small programming language IMP with *while* loops given by Winskel [38]. IMP is a simple imperative language with integers, truth values true and false, memory locations to store the integers, arithmetic expressions, boolean expressions and instructions.

The only difference we have made is that we have decided to work with integers instead of natural numbers. But it has no significant importance in the semantics of IMP. Locations are also represented by integers.

### 3.2.1 Difficulty in automation

In proof assistants, like **Coq**, Isabelle/HOL, executing instructions can be viewed as proving them as lemmas, where the facts needed to help the execution are provided as hypotheses. For instance, we would like to show an execution of a sequence of two instructions $i_1$ and $i_2$ starting from a state $\sigma$ will obtain a final state $\sigma''$, given the facts that execution of $i_1$ in the state $\sigma$ yields a state $\sigma'$ and the state $\sigma''$ will be obtained in an execution of $i_2$ in the state $\sigma'$.

**Lemma 3.1.** $\forall \sigma, \sigma', \sigma'' \colon \mathsf{State}. \forall i_1, i_2 \colon \mathsf{Inst}.$
$$\underbrace{(\langle i_1, \sigma\rangle_{\mathsf{I}} \leadsto \sigma') \to (\langle i_2, \sigma'\rangle_{\mathsf{I}} \leadsto \sigma'')}_{\mathsf{facts}} \to \underbrace{(\langle i_1; i_2, \sigma\rangle_{\mathsf{I}} \leadsto \sigma'')}_{\mathsf{goal}}.$$

To automate this proof, which is similar to the operational semantics description of eval_scolon, difficulties arise to derive the intermediate results, the state $\sigma'$ in our case, which does not appear in the goal. Usually this kind of proof is done by resolution and unification, as in Prolog interpreters, and missing values are replaced with existential variables to be instantiated later through unification. For our example in **Coq**, a unification and resolution based procedure **EAuto** finds a match with eval_scolon and replaces $\sigma'$ by $?1$. Then it needs to solve two more subgoals,viz., $(\langle i_1, \sigma\rangle_{\mathsf{I}} \leadsto ?1)$ and $(\langle i_2, ?1\rangle_{\mathsf{I}} \leadsto \sigma'')$. It then finds a match in the context for both the subgoals and therefore instantiates $?1$ to $\sigma'$, thus solving the goal. However a

[1]http://www-sop.inria.fr/lemme/pcoq/index.html
[2]http://www-sop.inria.fr/lemme/Kuntal.Das_Barman/reflsem/

unification and resolution based procedure fails when computation power is needed. For instance, consider the following lemma

**Lemma 3.2.** $\forall \sigma \colon \mathsf{State}.\forall v \colon \mathbb{Z}.$
$$\underbrace{(\mathsf{lookup}\ \sigma\ v\ 1)}_{\mathsf{facts}} \to \underbrace{\langle \mathsf{while}\ 3\ \le\ v\ \mathsf{do}\ \mathit{skip}, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma}_{\mathsf{goal}}.$$

Given the fact that the location $v$ is bound to 1 in the state $\sigma$, we need to prove that execution of the while loop will not change state. We have the necessary information to prove that the boolean expression $3 \le 1$ will be evaluated to false, but such a proof does not exist in the context and needs to be computed. A unification and resolution based procedure does not have the computation power for this. A way to solve this problem is to use functions instead of relations. Functions can also compute intermediate results, thus handling lemma3. 1.

As we have mentioned before, previously computed results are available as assertions, which actually are relations between data objects. But the functions compute on data objects, not on assertions. Unification and resolution based procedures use a context to do the proof search. To use functions, we need to construct the data objects from the context that truly represent the context. Now we show how to achieve this, along with the functional interpretations of the language.

## 3.3 Functional interpretation

Functions are well suited to describe how to evaluate arithmetic and boolean expressions, lookup for variable values or update the state, as they follow structural recursion. But we want these *evaluation functions to use data from the context.* We build a few tables to collect this information from the context and then design evaluation functions to consult regularly these tables. For instance,

**Lemma 3.3.** $\forall a_1, a_2 \colon \mathsf{AExp}.\forall \sigma \colon \mathsf{State}$
$$\underbrace{\langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1 \to \langle a_2, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_2}_{\mathsf{facts}} \to \underbrace{\langle a_1 + a_2, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1 + n_2}_{\mathsf{goal}}.$$

To keep these information on arithmetic expressions, $a_1$ and $a_2$, we create a table $\mathsf{T^r}_{\mathsf{AExp}}$ (to be read as *list of results given for arithmetic subexpressions*). $\mathsf{T^r}_{\mathsf{AExp}}$ a list of triplets, where each triplet consists of a state, an arithmetic expression and the integer value of this arithmetic expression in this state. For example for lemma 3.3, $\mathsf{T^r}_{\mathsf{AExp}} = [(\sigma, a_2, n_2), (\sigma, a_1, n_1)]$. We ensure that this table contains information that is only provided in the context with the *consistency function*:

$$[\![ \cdot ]\!]^{\mathsf{c}}_{\mathsf{AExp}} \colon (\mathsf{list\ State} * \mathsf{AExp} * \mathbb{Z}) \to \mathbf{Prop}$$

$$\begin{aligned}
&[]^{\mathsf{r}}_{\mathsf{AExp}} \colon \mathsf{True}\\
&[(\sigma,\ \mathsf{a},\ \mathsf{n}) :: \ \mathsf{l}^{\mathsf{r}}_{\mathsf{AExp}}]^{\mathsf{r}}_{\mathsf{AExp}} \colon\\
&\qquad \forall a \colon \mathsf{AExp}.\ \forall n \colon \mathbb{Z}.\ \forall \sigma \colon \mathsf{State}.\\
&\qquad\qquad \langle a, \sigma \rangle_{\mathsf{A}} \rightsquigarrow n \quad \wedge \quad [\![ \ \mathsf{l^r}_{\mathsf{AExp}} \ ]\!]^{\mathsf{c}}_{\mathsf{AExp}}.
\end{aligned}$$

Similarly we create different tables to collect different types of information from the context. We can have results to lookup in the memory states, to update a memory state, evaluate boolean expressions and execute instructions. We will represent them as $\mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}}$, $\mathsf{T}^{\mathsf{r}}_{\mathsf{update}}$, $\mathsf{T}^{\mathsf{r}}_{\mathsf{BExp}}$ and $\mathsf{T}^{\mathsf{r}}_{\mathsf{Inst}}$ respectively. Again, we ensure that these tables are consistent with the context with a set of consistency functions, viz., $[\![ \cdot ]\!]^{\mathsf{c}}_{\mathsf{lookup}}$, $[\![ \cdot ]\!]^{\mathsf{c}}_{\mathsf{update}}$, $[\![ \cdot ]\!]^{\mathsf{c}}_{\mathsf{BExp}}$ and $[\![ \cdot ]\!]^{\mathsf{c}}_{\mathsf{Inst}}$, respectively. The definitions of these functions are similar and we omit them.

To evaluate any expression, we first look in the corresponding table, whether we already know the result or not. If not, we follow the semantics. Therefore, the evaluation function for arithmetic expressions is defined as follows:

$$\begin{aligned}
&[\![\cdot]\!] \colon \mathsf{AExp} \to \mathsf{State} \to (\mathsf{list\ State} * \mathbb{Z} * \mathbb{Z})\\
&\qquad\qquad\qquad\qquad \to (\mathsf{list\ State} * \mathsf{AExp} * \mathbb{Z}) \to (\mathsf{option}\ \mathbb{Z})\\
&\lambda \mathsf{a} \colon \mathsf{AExp}, \lambda \sigma \colon \mathsf{State},\\
&\lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}} \colon (\mathsf{list\ State} * \mathbb{Z} * \mathbb{Z}),\\
&\lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}} \colon (\mathsf{list\ State} * \mathsf{AExp} * \mathbb{Z}),
\end{aligned}$$

$$\begin{aligned}
&\mathsf{x}\ \ \mathsf{when}\ (\sigma,\ \mathsf{a},\ \mathsf{x}) \in \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}\\
&\mathsf{Otherwise:}\\
&\ [\![\mathsf{Num(n)},\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma := n\\
&\ [\![\mathsf{Loc(v)},\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma := \mathsf{flookup}(\sigma,\ \mathsf{v},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}})\\
&\ [\![a_1 + a_2,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma := \begin{cases}
[\![a_1, \mathsf{T^r}_{\mathsf{lookup}}, \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma\\
\ + [\![a_2, \mathsf{T^r}_{\mathsf{lookup}}, \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma\\
\qquad \text{if } [\![a_1,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma \neq \mathsf{None}\\
\qquad \&\ [\![a_2,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma \neq \mathsf{None}\\
\mathsf{None}\\
\qquad \text{if } [\![a_1,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma = \mathsf{None}\\
\qquad \text{or } [\![a_2,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma = \mathsf{None}
\end{cases}\\
&\ [\![a_0 - a_1,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma := \cdots\\
&\ [\![a_0 * a_1,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma := \cdots
\end{aligned}$$

where $\mathsf{flookup}(\cdot, \cdot, \cdot)$ is the function giving the contents of a location in a state, defined by recursion on the structure of the state. It differs from lookup because it is a function, not a relation; lookup is its graph. We use the option type of **Coq** for type lifting.

In the same way, we define the evaluation function for boolean expressions

$$\begin{aligned}[\![\cdot]\!]\colon \mathsf{BExp} \to \mathsf{State} \to (\mathsf{list\ State} * \mathbb{Z} * \mathbb{Z}) \to (\mathsf{list\ State} * \mathsf{AExp} * \mathbb{Z})\\ \to (\mathsf{list\ State} * \mathsf{BExp} * \mathbb{B}) \to (\mathsf{option}\ \mathbb{B}).\end{aligned}$$

We overload the Scott brackets $[\![\cdot]\!]$ to denote the evaluation function both on arithmetic and boolean expressions.

Similarly, we define the update function

$$\begin{aligned}&\mathsf{fupdate}\colon \mathsf{State} \to \mathbb{Z} \to \mathbb{Z} \to (\mathsf{list\ State} * \mathbb{Z} * \mathbb{Z} * \mathsf{State}) \to (\mathsf{option\ State})\\ &\lambda\sigma\colon \mathsf{State}.\ \ \lambda \mathsf{n}_1, \mathsf{n}_2\colon \mathbb{Z}.\\ &\lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{update}}\colon (\mathsf{list\ State} * \mathbb{Z} * \mathbb{Z} * \mathsf{State}).\end{aligned}$$

$$\begin{aligned}&\mathsf{x}\ \ \mathsf{when}\ (\sigma,\ \mathsf{n}_1,\ \mathsf{n}_2,\ \mathsf{x}) \in \mathsf{T}^{\mathsf{r}}_{\mathsf{update}}\\ &\mathsf{Otherwise:}\\ &[\![[], \mathsf{T^r}_{\mathsf{update}}]\!]_\sigma := \mathsf{None}\\ &[\![[x \mapsto v, \sigma'], \mathsf{T^r}_{\mathsf{update}}]\!]_\sigma := \begin{cases}[n_1 \mapsto n_2, \sigma'] \quad \mathsf{if}\ x = n_1\\ \begin{cases}\mathsf{None}\\ \quad \mathsf{if}\ x \neq n_1\ \&\\ \quad\quad (\mathsf{fupdate}\ \sigma'\ \mathsf{n}_1\ \mathsf{n}_2\ \mathsf{T}^{\mathsf{r}}_{\mathsf{update}}) = \mathsf{None}\\ [x \mapsto v, \sigma'']\\ \quad \mathsf{if}\ x \neq n_1\ \&\\ \quad\quad (\mathsf{fupdate}\ \sigma'\ \mathsf{n}_1\ \mathsf{n}_2\ \mathsf{T}^{\mathsf{r}}_{\mathsf{update}}) = \sigma''\end{cases}\end{cases}\end{aligned}$$

## 3.4 Proof by reflection

In usual formal proofs, hypotheses about computations are represented as assertions that some relation hold for some piece of data. The proof search mechanisms usually search the context containing all these hypotheses to see if the goal can be solved directly because it is one of these assumptions. In a proof system based on type theory like **Coq**, functions are also provided and reduction can be used to compute with these functions [27]. The idea of reflection is to use these functions to perform the proof search. But functions compute on formalized data and the (context) hypotheses are not data at the level of these functions but only at the level of the proof system.

This discussion on levels can be found in [3]. Reflection was pioneered in Coq by Samuel Boutin [8], where reflection was used to decide efficiently whether two expressions, denoting values in a ring, are equal. Similarly, Kumar Neeraj Verma and Jean Goubault-Larrecq [36] have recently used reflection to build a certified BDD algorithm in **Coq**. Work on reflection in **Coq** can be also found in [21], where Dimitri Hendriks described a formalization of natural deduction for intuitionistic first-order logic in **Coq**.

In the literature we do not find any reference where reflection was used in semantics.

To prove a given property $P$ applied to some term $t$ using reflection, is as follows: Consider a proof-assistant where we can both describe and prove programs. Then write a program $Q$ that takes $t$ as input and returns true only when $P(t)$ holds. Prove that $Q(t) = \text{true} \Rightarrow P(t)$ and then use $Q$ to prove instances of $P$.

In our case, we would like to prove that $\langle\sigma, i\rangle_{\mathsf{I}} \rightsquigarrow \sigma'$ holds given a set of hypotheses $\Gamma$. We should, therefore, write a function $Q$ which takes *data objects* that represent $\Gamma, \sigma$ and $i$ as inputs and returns $\sigma'$ only when $\langle\sigma, i\rangle_{\mathsf{I}} \rightsquigarrow \sigma'$ holds along with $\Gamma$. To have this last criteria we need to prove $Q(\sigma,\ i,\ \textit{data objects from}\ \Gamma) = \sigma' \Rightarrow \Gamma \rightarrow \langle\sigma, i\rangle_{\mathsf{I}} \rightsquigarrow \sigma'$. For the needs of reflection, we have already built data to represent the hypotheses at the level where functions can compute.

In section 3.2 we described two problems that need to be solved by proof search engines to build proofs in semantics. The first problem is to find intermediate values in computation. This is solved in a natural way, the evaluation computations. The second problem is to interleave arithmetic computation with proof search and this, too, can easily be solved if evaluation functions call the relevant arithmetic functions. For these problems, proof tools based on function evaluation are better than proof search tools based on unification and resolution. Function evaluation is also more focused than proof search and therefore more efficient. All this process becomes very powerful if fast reduction mechanisms are implemented in proof assistant [19].

### 3.4.1 Giving names

A new difficulty arises when we reason on unknown expressions. Consider the following lemma :

**Lemma 3.4.** $\forall\sigma\colon \mathsf{State}.\forall v\colon \mathbb{Z}.(\mathsf{lookup}\ \ \sigma\ v\ 3) \rightarrow$
$$\langle\sigma, (\mathsf{while}\ v\ \ \leq\ \ 1\ \mathsf{do\ skip})\rangle_{\mathsf{I}} \rightsquigarrow \sigma.$$

In this case the memory state is given by a universally quantified variable $\sigma$. If we remember, in section 3.2, we defined state as an Inductive type Set with two constructors, one describes the case when the list is empty and the other describes the case when the list is non empty. But it does not say anything if we don't know anything about this list, in other words if such a list is described by a metavariable, for example by $\sigma$ as above.

In type theory based proof-assistant, function evaluation is performed by term reduction. Term reduction changes the term being studied, only if the current data matches one of the reduction rules. For instance,

$$[\![a_1 + a_2, \mathsf{T^r}_{\mathsf{lookup}}, \mathsf{T^r}_{\mathsf{AExp}}\ ]\!]_\sigma$$

reduces to $[\![a_1,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma\ +\ [\![a_2,\ \mathsf{T^r}_{\mathsf{lookup}},\ \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma$. But reduction does not occur if one of the expression is represented by a universally quantified variable, for instance $[\![a, \mathsf{T^r}_{\mathsf{lookup}}, \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma$ stays $[\![a, \mathsf{T^r}_{\mathsf{lookup}}, \mathsf{T^r}_{\mathsf{AExp}}]\!]_\sigma$.

The context can still contain enough information related to these metavariables to execute instructions, consider lemma 3.3. Therefore we need a way to reason about them. The solution is to associate a number to metavariables like $\sigma$. We do it in a systematic way. We define a new inductive type called n_State (to read as *named state*) which contains an extra constructor for these numbered terms.

$$
\begin{aligned}
&\mathsf{n\_State}\colon \mathbf{Set}\\
&[\,]_n\colon \mathsf{n\_State}\\
&[\cdot \mapsto \cdot, \cdot]_n\colon \mathbb{Z} \to \mathbb{Z} \to \mathsf{n\_State} \to \mathsf{n\_State}\\
&\mathsf{metavariable_s}\colon \mathbb{N} \to \mathsf{n\_State}
\end{aligned}
$$

Elements of n_State are names for expressions of type State in the theorem prover when an unknown expression of type State is available we assign a number (a name) $n$ and we represent it by $\mathsf{metavariable_s}(n)$. When an expression of type State is $[\cdot \mapsto \cdot, tl]$ we construct the name $n\_tl$ for $tl$ and we give the name $[\cdot \mapsto \cdot, n\_tl]$ for the whole expression. Similarly we associate numbers to all those arithmetic expressions, boolean expressions and instructions which are known by their symbolic names. For arithmetic expressions it is as follows:

$$
\begin{aligned}
&\mathsf{n\_AExp}\colon \mathbf{Set}\\
&\mathsf{Loc_n}(\cdot)\colon \mathbb{Z} \to \mathsf{n\_AExp}\\
&\mathsf{Num_n}(\cdot)\colon \mathbb{Z} \to \mathsf{n\_AExp}\\
&(\cdot\ +_{\mathsf{n}}\ \cdot)\colon \mathsf{n\_AExp} \to \mathsf{n\_AExp} \to \mathsf{n\_AExp}\\
&\quad\vdots\\
&\mathsf{metavariable_a}\colon \mathbb{N} \to \mathsf{n\_AExp}
\end{aligned}
$$

The specification for named boolean expressions and named instructions are similar and we omit them.

Once we assigned numbers to metavariables we need to keep track of them. We do so by creating a few more tables where we have entries only for metavariables as in other cases it's easy to get back the original. We have four such tables, namely, $\mathsf{T_{State}}, \mathsf{T_{AExp}}, \mathsf{T_{BExp}}$ and $\mathsf{T_{Inst}}$ for metavariables representing states, arithmetic expressions, boolean expressions and instructions, respectively. After we have assigned names to all arithmetic and boolean expressions, instructions and states, to work with the metavariables we define new result tables, $\mathsf{T^r}_{\mathsf{n\_lookup}}, \mathsf{T^r}_{\mathsf{n\_update}}, \mathsf{T^r}_{\mathsf{n\_AExp}}, \mathsf{T^r}_{\mathsf{n\_BExp}}$ and $\mathsf{T^r}_{\mathsf{n\_Inst}}$, that have a similar role and structure to the result tables, introduced in section 3.3, but contain named expressions.

Similarly, we need to change the consistency functions to work with named expressions. For instance, the consistency function for the arithmetic function is as follows.

$$\begin{aligned}
&\mathcal{C}_{\mathsf{n\_AExp}}(\cdot)\colon (\mathsf{list\ State} * \mathsf{n\_State}) \to (\mathsf{list\ AExp} * \mathsf{n\_AExp})\\
&\qquad\qquad\qquad \to (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}) \to \mathbf{Prop}\\
&\lambda \mathsf{T}_{\mathsf{State}}\colon (\mathsf{list\ State} * \mathsf{n\_State}),\\
&\lambda \mathsf{T}_{\mathsf{AExp}}\colon (\mathsf{list\ AExp} * \mathsf{n\_AExp}),
\end{aligned}$$

$$[]^{\mathsf{r}}_{\mathsf{n\_AExp}}\colon \mathsf{True}$$

$$[(\mathsf{n\_}\sigma,\ \mathsf{n\_a},\ \mathsf{n}) :: \ \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_AExp}}]^{\mathsf{r}}_{\mathsf{n\_AExp}}\colon \begin{cases} \langle a,\sigma\rangle_{\mathsf{A}} \rightsquigarrow n \ \ \wedge \\ \quad \mathcal{C}_{\mathsf{n\_AExp}}(\mathsf{T}_{\mathsf{State}},\ \mathsf{T}_{\mathsf{AExp}},\ \mathsf{T^r}_{\mathsf{n\_AExp}}) \\ \qquad \mathsf{if}\ [\![n\_\sigma, \mathsf{T}_{\mathsf{State}}]\!] \rightarrowtail \sigma \\ \quad \&\ \ [\![n\_a, \mathsf{T}_{\mathsf{AExp}}]\!] \rightarrowtail \mathsf{a} \\ \\ \mathsf{False} \\ \quad \mathsf{if}[\![n\_\sigma, \mathsf{T}_{\mathsf{State}}]\!] \rightarrowtail \mathsf{None} \\ \quad \mathsf{or}\ [\![n\_a, \mathsf{T}_{\mathsf{AExp}}]\!] \rightarrowtail \mathsf{None} \end{cases}$$

In the above, $[\![n\_\sigma,\ \mathsf{T}_{\mathsf{State}}]\!] \rightarrowtail \sigma$ is the *translation* from the named state to state. Given a named state the following function looks for a matching pair (state, named state) in the table $\mathsf{T}_{\mathsf{State}}$.

$$\begin{aligned}
&\mathsf{find\_nstate}\colon \mathsf{n\_State} \to (\mathsf{list\ State} * \mathsf{n\_State}) \to (\mathsf{option\ State})\\
&\lambda\ n\_\sigma\colon \mathsf{n\_State}
\end{aligned}$$

$$[]_{\mathsf{State}} \rightarrowtail \mathsf{None}$$

$$[(n\_\sigma', \sigma') :: \mathsf{T}_{\mathsf{State}}]_{\mathsf{State}} \rightarrowtail \begin{cases} \sigma' & \mathsf{if}\ n\_\sigma = n\_\sigma' \\ \\ (\mathsf{find\_nstate\ n\_}\sigma\ \mathsf{T}_{\mathsf{State}}) & \\ & \mathsf{if}\ n\_\sigma \neq n\_\sigma' \end{cases}$$

The translation function is defined as follows:

$$\begin{aligned}
&\mathsf{translate\_nstate}\colon \mathsf{n\_State} \to (\mathsf{list\ State} * \mathsf{n\_State}) \to (\mathsf{option\ State})\\
&\lambda\ \mathsf{T}_{\mathsf{State}}\colon (\mathsf{list\ State} * \mathsf{n\_State})
\end{aligned}$$

$$[]_n \rightarrowtail []$$

$$[x \mapsto v, n\_\sigma']_n \rightarrowtail \begin{cases} [x \mapsto v, \sigma'] \\ \qquad \mathsf{if}\ (\mathsf{translate\_nstate\ n\_}\sigma'\ \mathsf{T}_{\mathsf{State}}) = \sigma' \\ \\ \mathsf{None} \\ \qquad \mathsf{if}\ (\mathsf{translate\_nstate\ n\_}\sigma'\ \mathsf{T}_{\mathsf{State}}) = \mathsf{None} \end{cases}$$

$$(\mathsf{metavariable_s}\ s) \rightarrowtail\ (\mathsf{find\_nstate}\ (\mathsf{metavariable_s\ s})\ \mathsf{T}_{\mathsf{State}})$$

Similarly we need to translate arithmetic expressions, boolean expressions and instructions. We change the consistency functions for lookup in a state, arithmetic expressions, boolean expressions and instructions as well. They are similar and we omit their detail.

We also need to change the evaluation functions accordingly, so that we can work with the named expressions. For instance, now evaluation function for arithmetic expressions will be the following.

$$\begin{aligned}&[\![\cdot]\!] : \mathsf{n\_AExp} \rightarrow \mathsf{n\_State} \rightarrow (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z})\\ &\qquad\qquad\qquad\qquad \rightarrow (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}) \rightarrow (\mathsf{option}\ \mathbb{Z})\\ &\lambda \mathsf{n\_a} : \ldots \lambda \mathsf{n\_}\sigma : \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_lookup}} : \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_AExp}} : \ldots\end{aligned}$$

$$\begin{aligned}&\mathsf{x\ \ when\ \ } (\mathsf{n\_}\sigma,\ \mathsf{n\_a},\ \mathsf{x}) \in \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_AExp}}\\ &\mathsf{Otherwise:}\\ &[\![\mathsf{n\_Num(n)},\ \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} := n\\ &[\![\mathsf{n\_Loc(v)},\ \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} := \mathsf{n\_flookup}(\mathsf{n\_}\sigma, \mathsf{v}, \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_lookup}})\\ &[\![n\_a_1 +_n n\_a_2, \mathsf{T^r}_{\mathsf{n\_lookup}}, \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} := \begin{cases} [\![n\_a_1, \mathsf{T^r}_{\mathsf{n\_lookup}}, \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} \\ +[\![n\_a_2, \mathsf{T^r}_{\mathsf{n\_lookup}}, \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} \\ \qquad \text{if} \quad [\![n\_a_1, \cdots]\!]_{n\_\sigma} \neq \mathsf{None} \\ \qquad \& \quad [\![n\_a_2, \cdots]\!]_{n\_\sigma} \neq \mathsf{None} \\ \\ \mathsf{None} \\ \qquad \text{if} \quad [\![n\_a_1, \cdots]\!]_{n\_\sigma} = \mathsf{None} \\ \qquad \text{or} \quad [\![n\_a_2, \cdots]\!]_{n\_\sigma} = \mathsf{None} \end{cases}\\ &[\![n\_a_0 -_n n\_a_1,\ \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} := \cdots\\ &[\![n\_a_0 *_n n\_a_1,\ \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} := \cdots\\ &[\![\mathsf{metavariable_a(n)},\ \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]_{n\_\sigma} := \mathsf{None}\end{aligned}$$

Our approach is systematic. The structure of the inductive type and the functions over the type is kept unchanged with two main differences. First, we changed the input to their named counterparts and second, we added an extra rule to deal with the metavariable.

We prove that this evaluation function agrees with the operational semantics given by the inductive relation $\langle\cdot,\cdot\rangle \rightsquigarrow \cdot$ (all the theorems given below have been checked in a computer-assisted proof). Here is the statement for arithmetic expressions.

**Theorem 3.1.**
$\forall\ \mathsf{T_{State}} :\ (\mathsf{list\ State} * \mathsf{n\_State}). \forall\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_lookup}} :\ (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z}).$
$\forall\ \mathsf{T_{AExp}} :\ (\mathsf{list\ AExp} * \mathsf{n\_AExp}).\ \forall\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_AExp}} :\ (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}).$
$\forall\ \mathsf{n\_}\sigma : \mathsf{n\_State}. \forall\ \mathsf{n\_a} : \mathsf{n\_AExp}. \forall\ \sigma : \mathsf{State}. \forall\ \mathsf{a} : \mathsf{AExp}. \forall\ \mathsf{n} : \mathbb{Z}.$

$$\begin{aligned}&\mathcal{C}_{\mathsf{n\_lookup}}(\mathsf{T_{State}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n\_lookup}}) \rightarrow \mathcal{C}_{\mathsf{n\_AExp}}(\mathsf{T_{State}},\ \mathsf{T_{AExp}},\ \mathsf{T^r}_{\mathsf{n\_AExp}})\\ &\qquad\qquad \rightarrow [\![\mathsf{n\_}\sigma,\ \mathsf{n\_a},\ \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}}]\!]\ =\ \mathsf{n}\\ &\qquad\qquad\qquad\qquad \rightarrow [\![n\_\sigma,\ \mathsf{T_{State}}]\!] \rightarrowtail \sigma \rightarrow [\![n\_a,\ \mathsf{T_{AExp}}]\!] \rightarrowtail \mathsf{a}\end{aligned}$$

$$\rightarrow \langle a, \sigma \rangle_{\mathsf{A}} \rightsquigarrow n.$$

We also proved that evaluation functions for boolean expression, to update memory state and to lookup into a memory state agree. They are similar and therefore we omit them.

### 3.4.2 The iteration technique

We cannot define the evaluation function for instructions in the similar way as we did for arithmetic and boolean expressions, since the execution of instruction (in particular *while loop*) does not follow structural recursion. In the previous chapter, we have already presented *the iteration technique* to work around this problem.

The evaluation function for instructions can be provided in a way that respects typing and termination if we don't try to describe the evaluation function itself but the *second order function of which the evaluation function is the least fixed point.* This function can be defined in type theory by cases on the structure of the instruction.

$$
\begin{aligned}
&\mathsf{F}\colon (\mathsf{n\_Inst} \rightarrow \mathsf{n\_State}\\
&\qquad\qquad \rightarrow (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z}) \rightarrow (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z} * \mathsf{n\_State})\\
&\qquad\qquad \rightarrow (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}) \rightarrow (\mathsf{list\ n\_State} * \mathsf{n\_BExp} * \mathbb{B})\\
&\qquad\qquad \rightarrow (\mathsf{list\ n\_State} * \mathsf{n\_Inst} * \mathsf{n\_State}) \rightarrow (\mathsf{option\ n\_State}))\\
&\quad \rightarrow \mathsf{n\_Inst} \rightarrow \mathsf{n\_State}\\
&\quad \rightarrow (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z}) \rightarrow (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z} * \mathsf{n\_State})\\
&\quad \rightarrow (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}) \rightarrow (\mathsf{list\ n\_State} * \mathsf{n\_BExp} * \mathbb{B})\\
&\quad \rightarrow (\mathsf{list\ n\_State} * \mathsf{n\_Inst} * \mathsf{n\_State}) \rightarrow (\mathsf{option\ n\_State})\\
&\lambda f\colon \ldots \lambda n\_i\colon \ldots \lambda n\_\sigma\colon \ldots\\
&\lambda \mathsf{T^r}_{\mathsf{n\_lookup}}\colon \ldots \lambda \mathsf{T^r}_{\mathsf{n\_update}}\colon \ldots \lambda \mathsf{T^r}_{\mathsf{n\_AExp}}\colon \ldots \lambda \mathsf{T^r}_{\mathsf{n\_BExp}}\colon \ldots \lambda \mathsf{T^r}_{\mathsf{n\_Inst}}\colon \ldots
\end{aligned}
$$

$$
\begin{aligned}
&n\_\sigma' \ \ \mathsf{when}\ (n\_\sigma,\ n\_i,\ n\_\sigma') \in \mathsf{T^r}_{\mathsf{n\_Inst}}\\
&\mathsf{Otherwise}\colon\\
&(\mathsf{F}\ f\ \mathsf{skip}_n\ n\_\sigma\ \mathsf{T^r}_{\mathsf{n\_lookup}}\ \mathsf{T^r}_{\mathsf{n\_update}}\ \mathsf{T^r}_{\mathsf{n\_AExp}}\ \mathsf{T^r}_{\mathsf{n\_BExp}}\ \mathsf{T^r}_{\mathsf{n\_Inst}}) := n\_\sigma\\
&\qquad\vdots\\
&(\mathsf{F}\ f\ (\mathsf{while}\ n\_b\ \mathsf{do}\ n\_i)_n\ n\_\sigma\ \mathsf{T^r}_{\mathsf{n\_lookup}}\ \mathsf{T^r}_{\mathsf{n\_update}}\ \mathsf{T^r}_{\mathsf{n\_AExp}}\ \mathsf{T^r}_{\mathsf{n\_BExp}}\ \mathsf{T^r}_{\mathsf{n\_Inst}})\\
&\qquad := \begin{cases}
(f\ \ (\mathsf{while}\ n\_b\ \mathsf{do}\ n\_i)_n\ \ (f\ n\_i\ n\_\sigma\ \cdots)\ \cdots)\\
\qquad\qquad \text{if } [\![n\_b, \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}},\ \mathsf{T^r}_{\mathsf{n\_BExp}}]\!]_{n\_\sigma} = \mathsf{true}\\
\qquad\qquad\quad \&\ \ (f\ n\_i\ n\_\sigma\ \cdots) \neq \mathsf{None}\\
n\_\sigma \qquad \text{if } [\![n\_b, \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}},\ \mathsf{T^r}_{\mathsf{n\_BExp}}]\!]_{n\_\sigma} = \mathsf{false}\\
\mathsf{None} \qquad \text{if } [\![n\_b, \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_AExp}},\ \mathsf{T^r}_{\mathsf{n\_BExp}}]\!]_{n\_\sigma} = \mathsf{None}\\
\qquad\quad \text{or } (f\ n\_i\ n\_\sigma\ \cdots) = \mathsf{None}\\
\qquad\quad \text{or } (f\ \ (\mathsf{while}\ n\_b\ \mathsf{do}\ n\_i)_n\ \ (f\ n\_i\ n\_\sigma\ \cdots)\ \cdots) = \mathsf{None}
\end{cases}
\end{aligned}
$$

$$(\mathsf{F}\ f\ (\mathsf{metavariable_i(n)})_n\ n\_\sigma\ \cdots) := \mathsf{None}$$

We omit the full description of the evaluation function $\mathsf{F}$, as it makes the text unreadable and can be easily understood from the description of *while* clause.

Intuitively, writing the function $\mathsf{F}$ is exactly the same as writing the recursive function, except that recursive calls are simply replaced by calls to a bound variable (here $f$).

The function $\mathsf{F}$ describes the computations that are performed at each iteration of the execution function and the execution function performs the same computation as the function $\mathsf{F}$ when the latter is repeated *as many times as needed.* Later we will use the following notation

$$\mathsf{F}^k = \lambda g.\, \underbrace{(\mathsf{F}\ (\mathsf{F}\ \cdots\ (\mathsf{F}}_{k \text{ times}}\ g)\ \cdots\ ))$$

And finally to complete the task of reflection we prove that the evaluation function for instruction yields the same result as the operational semantics.

**Theorem 3.2.**
$\forall\ \mathsf{T_{State}}\colon\ (\mathsf{list\ State} * \mathsf{n\_State}).\forall\ \mathsf{T^r_{n\_lookup}}\colon\ (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z}).$
$\forall\ \mathsf{T^r_{n\_update}}\colon\ (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z} * \mathsf{n\_State}).$
$\forall\ \mathsf{T_{AExp}}\colon\ (\mathsf{list\ AExp} * \mathsf{n\_AExp}).\ \forall\ \mathsf{T^r_{n\_AExp}}\colon\ (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}).$
$\forall\ \mathsf{T_{BExp}}\colon\ (\mathsf{list\ BExp} * \mathsf{n\_BExp}).\ \forall\ \mathsf{T^r_{n\_BExp}}\colon\ (\mathsf{list\ n\_State} * \mathsf{n\_BExp} * \mathbb{B}).$
$\forall\ \mathsf{T_{Inst}}\colon\ (\mathsf{list\ Inst} * \mathsf{n\_Inst}).\ \forall\ \mathsf{T^r_{n\_Inst}}\colon\ (\mathsf{list\ n\_State} * \mathsf{n\_Inst} * \mathsf{n\_State}).$
$\forall\ \mathsf{k}\colon \mathbb{N}.\forall\ \mathsf{n\_}\sigma, \mathsf{n\_}\sigma'\colon \mathsf{n\_State}.\forall\ \mathsf{n\_i}\colon \mathsf{Inst}.\forall\ \sigma\colon \mathsf{State}.\forall\ \mathsf{i}\colon \mathsf{Inst}.$

$$\mathcal{C}(\mathsf{T_{State}},\ \mathsf{T_{AExp}},\ \mathsf{T_{BExp}},\ \mathsf{T_{Inst}},\ \mathsf{T^r}_{\mathsf{n\_lookup}},\ \mathsf{T^r}_{\mathsf{n\_update}},\ \mathsf{T^r}_{\mathsf{n\_AExp}},\ \mathsf{T^r}_{\mathsf{n\_BExp}},\ \mathsf{T^r}_{\mathsf{n\_Inst}})$$

$$\to (\mathsf{F^k}\ \bot\ \mathsf{n\_i}\ \mathsf{n\_}\sigma\ \mathsf{T^r_{n\_lookup}}\ \mathsf{T^r_{n\_update}}\ \mathsf{T^r_{n\_AExp}}\ \mathsf{T^r_{n\_BExp}}\ \mathsf{T^r_{n\_Inst}}) = \mathsf{n\_}\sigma'$$

$$\to [\![\mathsf{n\_}\sigma,\ \mathsf{T_{State}}]\!] \rightarrowtail \sigma \to [\![\mathsf{n\_i},\ \mathsf{T_{Inst}}]\!] \rightarrowtail \mathsf{i}$$

$$\to \exists\ \sigma'\colon \mathsf{State}.[\![\mathsf{n\_}\sigma',\ \mathsf{T_{State}}]\!] \rightarrowtail \sigma' \wedge \langle\sigma, \mathsf{i}\rangle_{\mathsf{I}} \rightsquigarrow \sigma'$$

$\mathcal{C}$ is a function which sums up the work by all consistency functions, namely $\mathcal{C}_{\mathsf{n\_lookup}}$, $\mathcal{C}_{\mathsf{n\_update}}$, $\mathcal{C}_{\mathsf{n\_AExp}}$, $\mathcal{C}_{\mathsf{n\_BExp}}$ and $\mathcal{C}_{\mathsf{n\_Inst}}$.

Note that we provide $\bot$, replacing the bound variable (denoted by $f$ earlier), to the functional $F$ to execute the program. Execution of a program fails in either of two cases. First, if the execution encounters any runtime error, or second, if the given number of iterations (here $k$) is not enough to finish the execution.

## 3.5 Collection of data objects and table building

The **Coq** system provides tools to make theorem proving easier. Among these tools, *tactics* play an important role. These tactics make it possible to obtain proofs in a semi-automatic way. Given a proposition that one wants to prove, the system proposes tools to construct a proof. These tools, called *tactics*, make it easier to construct the proof of a proposition, using elements taken from a context: declarations, definitions, axioms, hypotheses, lemmas and theorems that were already proven. In many cases, tactics make it possible to construct proofs automatically, but this cannot always be the case. This notion of tactics dates back to very old versions of proof assistants; before **Coq**, we can cite LCF [18], HOL [17], and Isabelle [30].

Tactics are commands that can be applied to a goal. Their effect is to produce a new, possibly empty, list of goals. If $g$ is the input goal and $g_1$, $\ldots$, $g_k$ are the output goals, the tactic has an associated function that makes it possible to construct a proof of $g$ from the proofs of goals $g_i$.

In our case we write few tactics to collect data objects from the given facts (provided as propositions) in the context. We assign names to states, arithmetic expressions, boolean expressions and instructions. For the metavariables, we put them in tables $\mathsf{T}_{\mathsf{State}}$,$\mathsf{T}_{\mathsf{AExp}}$, $\mathsf{T}_{\mathsf{BExp}}$ and $\mathsf{T}_{\mathsf{Inst}}$. Similarly, we write tactics to build the tables $\mathsf{T^r}_{\mathsf{n\_update}}$, $\mathsf{T^r}_{\mathsf{n\_lookup}}$, $\mathsf{T^r}_{\mathsf{n\_AExp}}$, $\mathsf{T^r}_{\mathsf{n\_BExp}}$ and $\mathsf{T^r}_{\mathsf{n\_Inst}}$, where we keep the results of evaluations of program constructs.

Finally we write a tactic *Ltac* to automate the proof on program construct evaluation. *Ltac* first calls the tactics to build up different tables we need for the computations and assign names. Then the *Ltac* tactics apply the induction principle attached to the theorem 2, which generates few subgoals and all these subgoals can be solved using already available automatic tactics in the **Coq** proof system. We use these automatic tactics inside *Ltac* to make it a complete tool. In the following section we give an example to show the usefulness of the *Ltac* tactic.

## 3.6 Usefulness of Ltac

Let us consider the following program construct in C programming language [24].

```
{
int a=0;
while (a <= 2) {a = a+1};
}
```

We would like to check that when the above program construct stops the variable $a$ contains the value 3. The above program construct is represented in the Coq proof assistant as

```
Theorem example: (s: state)
 (exec (one_location ‘1‘ ‘1‘ s)
       (Scolon
           (Assign ‘1‘ (Num ‘0‘))
```

```
            (WhileDo (LessEq (Loc ‘1‘) (Num ‘2‘))
                 (Assign ‘1‘ (Plus (Loc ‘1‘) (Num ‘1‘)))))
        (one_location ‘1‘ ‘3‘ s)).
```

The variable $a$ is represented by the integer ‘1‘ in the above. We assume that before the program construct the variable $a$ (or ‘1‘) contains 1 and the rest of the memory state is represented by $s$. This fact is given by the predicate (one_location ‘1‘ ‘1‘ s), which represents the initial state. Similarly the final state (one_location ‘1‘ ‘3‘ s) says that the variable $a$ (or ‘1‘) contains the value 3 when the program construct stops. The predicate

```
        (Scolon
            (Assign ‘1‘ (Num ‘0‘))
            (WhileDo (LessEq (Loc ‘1‘) (Num ‘2‘))
                 (Assign ‘1‘ (Plus (Loc ‘1‘) (Num ‘1‘)))))
```

described in the section 3.5, we collect the data objects from the given facts and put them in different tables. Our proof search tool solves the above theorem in one step. Following is the proof of the above example.

```
Intros s.
Ltac ’(5).
Qed.
```

where 5 is the depth of the computation.

## 3.7 Conclusions

This technique to assign names to metavariables has a vast potential. It is systematic and does not depend on the language. Our method tries to maximize the potential for automation: we have implemented a tactic that successfully handles complex goals. We hope to apply the same technique for larger languages and use them in proofs about compilers. There are good reasons to believe that this will be possible because our approach is very systematic. In recent work on a more complete programming language with procedure, we could see that the functional approach carries over nicely to larger programming language, even though semantics requires mutual inductive propositions.

In the literature we found work by Nancy A. Day and Jeffrey J. Joyce where they discuss about *Symbolic Functional Evaluation*[12]. But this work is different from ours, as in their work symbolic functional evaluation is an algorithm for executing functional programs to evaluate expressions in higher order logic. It carries out the logical transformations of expanding definitions, beta-reduction and simplification of built-in constants in the presence of uninterpreted constants and quantifiers. They suggest different levels of evaluation for such an algorithm to terminate while evaluating the arguments of uninterpreted functions. However this kind of capability already exists in the **Coq** proof-assistant, where the tactic *Simpl* does a similar work. In our work we showed a way to reason on metavariables. One can have some information related to uninterpreted symbols and use it in an intelligent way.

The main lesson we learned in this work is the technique for naming subexpressions that makes it possible to reason about non-closed programs.

The second important lesson that we learn in this experiment is that we can reason by induction about the symbolic execution of a program through an induction on the natural number that counts the iterations that are required to obtain a result.

With the traditional presentation, based on Inductive predicates, reasoning by induction relies on the induction principle that is provided for the induction predicate. This induction principle corresponds to what Winskel calls *rule induction* in [38]. Rule induction is an approximation of induction on the size of derivations for proofs of execution, but it is only an weak approximation, because induction hypotheses are provided only for direct subderivations.

Actually, defining the generation of subderivations for proof execution is a tricky problem, much trickier than defining subterm for a non-dependent inductive type. As our technique uses induction on the number of iterations required to obtain a result we have another approximation, but this approximation is much more powerful because we can use induction hypotheses for derivations that are not even subderivations, as long as they are smaller in size. The experiments in the *Concert* effort around an optimizing compiler showed that induction on the number of iterations was the most practical solution, if not the only practical one, to cope with optimizations where one has to consider executions of subterm that are not direct subterms.

Chapter 4

# An experimental compiler for Cminor to RTL - I

## 4.1 Introduction

In the industry where formal methods, machine checked proof of programs, model checking, etc, are used to certify software, in general the source code is certified. But the code which is effectively executed by the machine, generated from the source code by a compiler, is not formally verified. An incorrect compiler can perfectly introduce errors in a code whose source is well verified by machine checked proofs. Bugs in compilers are not rare, especially in the area of complex optimizations. In addition, specialized compilers for embedded systems are more error prone than the general compilers, as they are less tested. In other words, when we want to raise the level of certification, compilers become a weak link between the formally verified source code and the formally certified microprocessor.

As a solution to this problem, the current practice in the industry is to manually verify the assembler code generated by the compiler in the correlation of the source code. But such a solution takes a lot of time and is easily error prone. In general, source code of a program are changed often due to modifications or corrections. Change of compiler is rare, but not impossible. One may as well like to verify the program running in different kind of processors. In such cases, each time we change the source program or the compiler we need to redo the verification, which takes a lot of effort and time. Verifying the assembly code generated for different kinds of processors makes it not a very practical solution. In addition, such a verification prevents high-level optimization.

Compilers are debugged using a large numbers of test cases, some of which are distilled at considerable expense from large application programs. Besides being expensive and time consuming, the effectiveness of testing is reduced by the practical impossibility of adequately covering the execution paths of the compiled programs. George C Necula and Peter Lee [25] proposed certifying compilers to check the result of each compilation instead of verifying the compiler's source code. A certifying compiler not only produces the machine code for each given program to it, but also produces a proof of correctness for the given machine code. The compiler itself does not need to be verified.

An alternative solution would be to use a certified compiler. The notion of certified compiler is close to but distinct from the notion of certifying compiler used in the context of proof carrying code. A certified compiler possesses a proof of correctness which is valid for all the programs that it accepts. In particular, the certified compiler should come with a formal proof to certify the compilation process that the generated machine code is semantically equivalent with the source code. At least there should be a formal proof that all the properties which are true in the source code remain true in the compiled machine code.

In general most of the source code for the software in the industry are written in C [24]. To our knowledge C compilers were never formally verified. A realistic solution would be a formally verified compiler which compiles a source language usable in practice (typically, the subset of C used to program embedded systems) which produces target code for a RISC microprocessor of the real world; and which carries out some optimizations among the most profitable optimizations. A group of researchers has joined together to share their experiences to look into the above. This project is known as ARC Concert. The objective of the ARC Concert is to realize a certified compiler which is complete and moderately optimized.

In this project we have to define formal semantics for each of the source, intermediate and target languages. Particularly, a usable semantics of a subset of C on a realistic machine is necessary. Use of these semantics on machine should comprise of the use in the proof systems and symbolic execution in order to test the specifications on example programs. Finally we need to write a compiler, written directly in the form of Coq functions. In other words, a purely functional programming style will be followed where all the recursions are either primitive or well founded. These constraints of recursion in writing the compiler ensure that the compiler terminates.

The general architecture of a compiler is well known and has not changed much in the last 30 years. After a syntax analysis and eventually a type verification phase, a series of program transformations progressively takes the source language to the machine language, via one or several intermediate languages.

Among the intermediate languages the most commonly used are the register transfer languages or *RTL*, also known as 3 *code addresses*: these are imperative languages with the basic constructors directly corresponding to the processor instructions, but they operate on arbitrary number of pseudo registers. More constrained forms of RTL, like the SSA (*single static assignment*) form or control flow graphs are also used.

Finally, we need to decide what would be the target language of our compiler. We chose RISC type PowerPC processors over the CISC type Pentium processors for two reasons. First, PowerPC is more widely used in the embedded world, and second, the semantics of the RISC processor is simpler to describe. Within the framework of the ARC Concert, among the vast choices over the source, intermediate and target languages, we decided to concentrate in priority on the following combination: a subset of C as the source language; RTL as intermediate language; the PowerPC machine code as target language. This combination is enough to address most of the practical problems we encounter and in this combination the semantics of the source and the target languages are not too different. The main goal of ARC Concert is to realize the formalized compiler for a C like language which can be used for practical purposes. Once we have achieved that goal, extensions of these techniques will help us to develop formalized compilers for other high-level languages.

Certain transformations pass from one language to another language which is more closer to the processor instructions: explicitation of v-tables (for the languages to objects), instruction selection (RTL production), allocation of registers (RTL with pseudo registers to RTL with real registers), machine code generation. Other transformations (the optimizations) stay in the same intermediate languages, but the transformations make the code more efficient: constant propagation, loop invariants, delooping, elimination of induction variables, linearization of instructions. In our case, register transfer language is described as graph and code linearization goes from one language (RTL graph) to another (linear).

Most of the transformations are applied after some static analysis of the code, which indicates which transformations are needed and which of these transformations will preserve the semantics of the code. For instance, explicitation of v-tables is applied after control flow analysis, the constant propagation and loop optimizations are applied after data-flow analysis, register allocation is applied on a data-flow analysis during the lifespan of a variable, completed by graph coloring of interference results.

Essentially, the program transformations need to be certified. Syntax analysis, generated by a formal grammar, is essentially correct by construction. The static type verification does not need to be verified, except if the rest of the compiler exploits the typing results in a crucial way. Otherwise every state of code transformation needs to be accompanied with a proof of semantic equivalence between its entry and exit. This implies that every language we are considering (source, intermediate and executable) must possess a formal semantics. Since the transformations exploit the results of static analyses, the latter also needs to be proved correct: the results of the static analyses should constitute an approximation of the dynamics of the program.

The transformation proofs on the machine and the static analyses can naturally be considered as instance of the general problem of proof of programs. It is necessary to represent these transformations and analyses in a form of objects that the proof system can manipulate. We think that in the context of compilers if we write these transformations and analyses directly in the form of Coq functions we will get the results in a more direct way. On these functions we can directly reason and finally we can extract the compiler program in Caml from the Coq functions.

In the *Concert* project, whose main objective is to realize a certified compiler which is complete and moderately optimized, my objective is to realize a compiler which constructs the RTL control flow graph from the Cminor programs. Another objective is to realize this compiler as close as the Caml programming style and to keep the modularity of programming, overcoming the constraints put by the type theory. The way we have written the compiler to construct the RTL control flow graph from the Cminor programs significantly helps in its correctness proofs, but we do not do any proofs.

In the next section we present the source language we have chosen for our compiler. It is a subset of C, that excludes some of the most complicated constructs of C. We call this subset of C as Cminor. Cminor is sufficiently rich to encode all ANSI C except for **setjmp** and **longjmp**. We tried to simplify Cminor as much as possible, keeping in mind that it should be able to express almost all the ANSI C programs. For instance, the different repetitive structures of ANSI C, *for, while, do-while*, can be described with the same constructs of Cminor after a little modifications. Our choice of Cminor also helps us in transformation to RTL. We describe this in a little more detail at the end of the next section.

## 4.2 Cminor

Cminor is described as an imperative language based on expressions and instructions. Similar to C, it is also weakly typed and it has both integers and floating point numbers. Pointers are considered as integers representing the memory addresses. In Cminor, the questions raised by the overloaded operators are solved and type conversions are made explicit. The addresses for the tables, records, etc., are also computed explicitly. Compared to C, control structures are simplified in Cminor. The specification of Cminor makes it possible to accommodate later addition of a garbage collector and exceptions.

We describe the language Cminor by its abstract syntax. The language Cminor is organized in four parts, expressions, instructions (or statements), procedures and programs. We start with the comparison relations, namely $=, \neq, <, \leq, >$ and $\geq$. Comparison relations are necessary for Cminor expressions. We use the same comparison relations for RTL. The comparison relations are given by the following type.

```
Inductive comparison : Set :=
 | Cequal        : comparison
 | Cnotequal     : comparison
 | Cless         : comparison
 | Clessequal    : comparison
 | Cgreater      : comparison
 | Cgreaterequal : comparison.
```

### 4.2.1 Memory structure

In Cminor the memory is designed similar to the memory of the processor. A pointer is an integer value which refers to a byte of the memory. Though this memory model is simpler, it can appear awkward in certain cases. For example, nothing prevents the stacks from allocating the memory which is already allocated to the static variables and this does not match totally with the semantics of C, where a local variable is never the alias of a global variable. Therefore it is necessary to plan dynamic tests of stack overflow and this should be tested until the assembler code generated at the end. In the interpreter the allowance of the global variables and stack in the memory capacity is deterministic. The specification needs to allow a nondeterministic placement, with conditions of non-overlap between global variables and the stack space. Finally, distributing the stack space in local blocks is also entirely deterministic, and does not allow to use, in later phases of the compiler, the same stack to store return addresses and intermediate results.

To access the memory the type *memory_chunk* indicates the size (8, 16, 32 or 64 bits) and the type (integer or float) of the given value. The 8 bits and 16 bits integer values are stored as 32 bits integer, by adding zeros if the integer values are not signed or by replicating the signed bit if the integer values are signed.

```
Inductive memory_chunk : Set :=
 | Mint8signed: memory_chunk
 | Mint8unsigned: memory_chunk
 | Mint16signed: memory_chunk
 | Mint16unsigned: memory_chunk
 | Mint32: memory_chunk
 | Mfloat32: memory_chunk
 | Mfloat64: memory_chunk.
```

Memory is accessed with the load and store operations, which we will discuss in the section 4.2.3. For the load and store operations in memory we need to specify the block of memory that should be accessed. The above definition of different memory blocks suitably serve our purpose.

### 4.2.2 Cminor operations

The types *unary_operation* and *binary_operation* gather all the strict operators of the language, of arity one and two respectively. In Cminor operations, overloading of operators are solved and we made them explicit. For example, negation operator for integer and float values are expressed in Cminor as *OPnegint* and *OPnegfloat*, respectively. Among the operators in C are the negation operators, like " ! ", " $\sim$ " , " $-$ ", and type casting operators are expressed as unary operators in Cminor. Here is the abstract syntax of unary operators in Cminor.

```
Inductive unary_operation : Set :=
 | OPnegint          : unary_operation
 | OPnegfloat        : unary_operation
 | OPabsfloat        : unary_operation
 | OPintoffloat      : unary_operation
 | OPfloatofint      : unary_operation
 | OPfloatofintu     : unary_operation
 | OPnotbool         : unary_operation
 | OPnotint          : unary_operation
 | OPcast8signed     : unary_operation
 | OPcast8unsigned   : unary_operation
 | OPcast16signed    : unary_operation
 | OPcast16unsigned  : unary_operation
 | OPsingleoffloat   : unary_operation.
```

In general, the semantics of the unary operations are straightforward. We negate an integer by subtracting it from zero. The *not* operations are of two types. For the boolean not operation we check the integer against zero, which returns a boolean value and finally we translate the boolean value to its corresponding integer. The bitwise not operation on an integer is performed by the bitwise nor operation over the two copies of the integer.

Overloading of operators are also explicit for binary operations. Here we have overload of some operators for three different data types, viz. integers, floats and pointers. None of the operations are allowed between two different data types. Thus for a C operation including two different data types, we need to type-cast one of the data types explicitly for the corresponding Cminor operation. For instance,

```
{
        int a=12, c;
        float b=2.5;
        c=a/b;}
```

will result $c = 4$ in C. If $c$ is declared as *float* then $c$ will contain 4.8. In Cminor, we need to specify the operation accordingly. For instance, if $c$

is a float we should write *(OPdivfloat (OPfloatofint a) b)*, if $c$ is an integer we should write *(OPintoffloat (OPdivfloat (OPfloatofint a) b))*. Note that we converted $a$ from integer to float in the previous expressions. This is the convention normally followed in C, automatic conversions convert a narrower operand into a wider operand without losing information.

```
Inductive binary_operation : Set :=
 | OPaddptr         : binary_operation
 | OPsubptr         : binary_operation
 | OPcmpptr         : comparison -> binary_operation
 | OPaddint         : binary_operation
 | OPsubint         : binary_operation
 | OPmulint         : binary_operation
 | OPdivint         : binary_operation
 | OPmodint         : binary_operation
 | OPdivintu        : binary_operation
 | OPmodintu        : binary_operation
 | OPandint         : binary_operation
 | OPorint          : binary_operation
 | OPxorint         : binary_operation
 | OPshiftleftint   : binary_operation
 | OPshiftrightint  : binary_operation
 | OPshiftrightintu : binary_operation
 | OPaddfloat       : binary_operation
 | OPsubfloat       : binary_operation
 | OPmulfloat       : binary_operation
 | OPdivfloat       : binary_operation
 | OPcmpint         : comparison -> binary_operation
 | OPcmpintu        : comparison -> binary_operation
 | OPcmpfloat       : comparison -> binary_operation.
```

For the comparison operator in Cminor we need to provide the name of the kind of the comparison we would like to carry out. In general the semantics of the binary operations in Cminor is straightforward. For example, *OPaddint* is evaluated by integer addition on two integers provided for this operation. But for *OPsubptr* we need to check that we do not refer to a wrong address, this is done by comparing the address and the amount that would be deducted from this address.

### 4.2.3 Cminor expressions

The abstract syntax for the Cminor expressions are described as follows.

```
Inductive expr : Set :=
 | Evar       : lident -> expr
| Eaddrsymbol : gident -> expr
| Eaddrstack  : intval -> expr
| Eassign     : lident -> expr -> expr
| Econstint   : intval -> expr
| Econstfloat : floatval -> expr
| Eunop       : unary_operation -> expr -> expr
| Ebinop      : binary_operation -> expr -> expr -> expr
| Eload       : cmemory_chunk -> expr -> expr
| Estore      : cmemory_chunk -> expr -> expr -> expr
| Ecall       : expr -> (list expr) -> type_res -> expr
| Eandbool    : expr -> expr -> expr
| Eorbool     : expr -> expr -> expr
| Econdition  : expr -> expr -> expr -> expr
| Esequence1  : expr -> expr -> expr
| Esequence2  : expr -> expr -> expr.
```

Constructors *Eaddrsymbol gv* and *Evar lv* return the value associated with the global variable *gv* and the local variable *lv*, respectively. *Eaddrstack n* returns the address stored in the stack at an offset *n*. *Eaddrstack* is used to access the array elements and we discuss it again in the section 4.2.5, where we describe how do we represent arrays in Cminor. We distinguish between an integer and a float type of constant by the expressions *Econstint n* and *Econstfloat n*.

Unary and binary operations are described by the constructors *Eunop* and *Ebinop*. Each of these constructors expect the name of the operation and proper number of expressions required for the operation. The *Ebinop* operator is a strict operator. In other words both of its expressions will be evaluated irrespective of the operation. For this reason we need to separate the non strict binary operations. In case of *Eandbool* and *Eorbool* the second expression will not be evaluated if the the first expression is evaluated to false and true, respectively. Similarly, the operator *Econdition* is non strict, thus depending on the value of the first expression either of the second or third expression will be evaluated. *Eassign* is used for the assignments. For load and store operations in the memory *Eload* and *Estore* is used. The expressions of assignment *Eassign* and *Estore* return the value of the right-hand member of the assignment as the result.

We provide two constructors to express the sequence of expressions. The first constructor *Esequence1* $e_1$ $e_2$ evaluates $e_2$ after $e_1$ and the returns the value of $e_1$ as the result. It is needed for expressions like $a = b++$ in C, where $a$ will be assigned the value of $b$, followed by the increment of $b$ by 1 and the result of this expression is the value returned by $a$. For example, the C program fragment $a = b++$ will be described in Cminor as *(Esequence1 (Eassign a (Evar b)) (Eassign b (Ebinop OPaddint (Evar b) (Econstint 1))))*. The result of this operation will return the value of $a$. The second constructor *Esequence2* $e_1$ $e_2$ evaluates $e_2$ after $e_1$ and the returns the value of $e_2$ as the result. The second constructor is more natural and will be used more often than the first operator for sequence. For instance, expressions like $a = ++b$ in C is evaluated as follows: first $b$ is incremented by 1 and then its value is assigned to $a$ and the final result of this expression is the value returned by $a$. For example, the C program fragment $a = ++b$ will be described in Cminor as *(Esequence2 (Eassign a (Evar b)) (Eassign b (Ebinop OPaddint (Evar b) (Econstint 1))))*. The result of this operation will return the value of $b$.

C functions are called by their name and providing appropriate number and types of arguments. In Cminor, function calls are expressed with the constructor *Ecall*, which expects the function name and list of arguments - each of which are provided as Cminor expression. Note the use of recursive types to describe the construct *Ecall* in the definition of *expr*,

```
 Ecall       : expr -> (list expr) -> type_res -> expr
```

This makes it possible to implement a systematic reasoning pattern, but writing a recursive function on this recursive type *expr* comes across the constraints of guardedness conditions, which we describe in the section 5.7. The subexpression (list expr) no longer remains a subterm of the initial term expr. We discuss this in the next chapter. We use this kind of recursive types also in the section 4.2.4.

In C, each function may or may not return a value. To verify that a function in Cminor returns a value of correct type or does not return any value we also provide the return type in *Ecall* as its third argument. This third argument for the result type *type_res* is given as follows.

```
Inductive typ : Set :=
 | TPint   : typ
 | TPfloat : typ
 | TPaddr  : typ.

Definition type_res := (option typ).
```

Thus the type indicates the expected type of the result of the function, for example if the function returns an integer value it is described as *Some TPint*. This type can be *None* if the function does not return any value, which corresponds to the function of the type *void* in C. In general, type verification is done each time we call a function and each time we leave a function, so that the caller and the callee function agree on the type of the arguments and the result.

### 4.2.4 Cminor instructions

The abstract syntax for the instructions(or statements) are given as follows.

```
Inductive statement : Set :=
 | Sexpr       : expr -> statement
 | Sifthenelse : expr -> (list statement)
                            -> (list statement) -> statement
 | Sswitch     : expr -> (list (list intval)*(list statement))
                            -> statement
 | Sloop       : (list statement) -> statement
 | Sblock      : label -> (list statement) -> statement
 | Sexit       : label -> statement
 | Sreturn     : (option expr) -> statement.
```

The constructor *Sexpr* describes an expression. We have already discussed different types of instructions. C conditional structure *if (condition) statement1 else statement2* is described with the constructor *Sifthenelse*. Note that, the constructor *Econdition* does not describe the conditional structure if-else of C, rather it describes the conditional operator "(_?_:_)". For instance, C statements like *(a==b)?a:b* are described with the constructor *Econdition*.

The selective structure *switch* of C is described with the constructor *Sswitch*. An expression and several branches, similar to the switch of C, are provided to this constructor. Each branch is composed of a list of integer values, the values of the expression for which this branch must be carried out and a list of instructions, the instructions to be executed if this branch is selected. An empty list of integer values indicates the default case, the corresponding branch will be executed if no other integer value corresponds to the value of the expression. Like in C and Java, when a branch is selected, its instructions are carried out and then the execution falls through to the following branches. Thus the following instructions of all the following branches are executed. This behavior can be changed by using a labeled block and the construction *Sexit*.

The instruction *Sblock* groups together a list of statements under a label. A similar example in the language C could be the use of { and } pair. In general, such a pair does not contain any label. The instruction *Sexit* moves the control flow to a specified location given as a label, where the label is an argument to the *Sexit* instruction.

The instruction *Sloop* is an infinite loop, without a termination condition. C repetitive structures or loops (for instance *while, for, do-while*) are described with this constructor. To terminate a loop it is necessary to place *Sloop* in a named block and to use construction *Sexit* for the terminating condition. With the expense of a few more *break* and *continue* instructions of C, constructors *Sblock* and *Sexit* make it possible to express all kinds of loops. For instance, the following C while loop

```
while (a!=b) c;
```

can be restructured as follows

```
if (a!=b) {
           loop {c; if (a!=b) continue; else break;}
          }
```

where *loop* is an infinite loop constructor, more precisely *loop* $x$ = *while true* $x$.

C expressions for bifurcation of control and jumps are described by the constructors *Sblock* and *Sexit.* For instance, C expressions *break* and *continue* can be described by the the constructor *Sexit* by providing the proper label. Similarly *goto* can be described by *Sblock.* When the execution of the instruction *Sexit(lbl)* is carried out, it prevents the execution of the following instructions and moves the execution control at the end of the block *Sblock* to the instruction whose label is equal to *lbl.* The *return* statement of the C language is described by the constructor *Sreturn.* The function that does not return a value, *void* in C, is described by *Sreturn None.* Otherwise if it returns the value of the expression *expr*, in Cminor we write *Sreturn (Some expr).*

### 4.2.5 Cminor procedures

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming in C can offer us. In general, a function is a block of instructions that is executed when it is called from some other point of the program. The following is its format in C:

*type name* (*argument1, argument2, . . .*) *<list of statements>*

where:

- *type* is the type of data returned by the function.
- *name* is the name by which it will be possible to call the function.
- *arguments* can be specified as many as we want. Each argument consists of a type of data followed by its identifier, like in a variable declaration (for example, int x) and which acts within the function like any other variable. They allow passing parameters to the function when it is called.
- *list of statements* is the function's body. It can be a single instruction or a block of instructions.

We can declare temporary variables for computation inside the function. The scope of these variables is restricted to the function. C functions are described by the procedures in Cminor. Here is the abstract syntax of the Cminor procedures.

```
Record procedure : Set := mkproc{
   proc_return     : type_res;
   proc_params     : (list lident * typ);
   proc_vars       : (list lident * typ);
   proc_stackspace : size;
   proc_body       : (list statement)
 }.
```

The field *proc_return* represents the type of the data returned by the procedure. It should be *None* if the procedure does not return any value, like the function of the type *void* in C. The field *proc_params* describes the parameters of the procedure as an associated list of the parameter identifiers and their corresponding types. The C functions with a variable number of arguments are not treated. In the similar way the field *proc_vars* describes the local variables of the procedure. Cminor does not allow the declaration of local variables at the level of a block. Thus the scope of all the local variables is the very whole procedure. The type of the variables and the parameters is either integer, floating point numbers, or pointers.

The field *proc_stackspace* indicates the size of the block in the number of octet to be allocated at the start of the function. This block can later be used during the lifetime of the function by the expression *Eaddrstack*. This block is automatically freed when the control leaves the function.

The main use of *proc_stackspace* is for arrays. It is better to take an example to understand how do we represent an array and access array elements in Cminor. Let's say in C, we consider the declarations of two following arrays.

```
int      intlist[8];
float    floatlist[10];
```

These two arrays should be declared in one of the functions, either *main* or some other, in C. In Cminor, we allocate spaces for these two arrays in the corresponding Cminor procedure. Spaces are allocated consecutively. We allocate $(8 * 32)/8$ or 32 octets for the *intlist* at the beginning of the *proc_space* and $(10 * 64)/8$ or 80 octets for the *floatlist* next to the *intlist*. The $n$th element of the *intlist* is accessed by *(Eaddrstack* $4*n$*)* and the $n$th element of the *floatlist* is accessed by *(Eaddrstack* $32+(8*n)$*)*.

Lastly, the field *proc_body* contains the instructions which constitute the body of the procedure.

The evaluation of a procedure applied to arguments results in an *option value* and represents the returned value of the procedure. The evaluation is not defined in the following cases. First, the effective arguments and the required parameters are different in numbers or in types. Second, we access a local variable which is not initialized. Third, the procedure execution terminates prematurely by the construction *Sexit* without a corresponding *Sblock*.

### 4.2.6 Cminor program

A complete program is composed of a set of procedures and a set of global variables. A global variable is either a pointer to a memory address allocated in the heap, or a pointer to a function.

```
Record program : Set := {
   prog_procs : (list  gident * procedure);
   prog_vars  : (list gident * size)
 }.
```

The evaluation of a complete program starts by allocating the memory space and by constructing the global environment by assigning the addresses to the global variables. Then we call the procedure named *main* in the program. The result of the program is composed of the value returned by the procedure *main* and the final contents of the global variables, in the form of an associated list (identifier, size of the variable).

The evaluation of expressions and instructions is carried out with the evaluator, we do not give their specification here. We give a few remarks on them. The evaluator consists of 4 mutually recursive functions, for expressions, instructions, sequence of instructions, and procedure calls. These functions are parameterized by store, global environment, local environment and stack pointer. The store represents memory, the global environment corresponds to the pointers and the procedure names, the local environment consists of local variables and the stack pointer is an address in the memory where the stack starts.

### 4.2.7 Comparison with C

The functions with a variable number of arguments could be coded as procedures taking a last parameter of the type addresses, pointing towards a block containing the additional arguments. However, such a coding would be extremely complex, for a feature which does not have any interest for embedded software. To translate a C code (annotated by the types) into Cminor requires few transformations, first overloaded arithmetic operators should be resolved, second Cminor needs explicit address computations at the time of the access to the tables, structures and unions, third the coding of the loops in terms of *Sblock, Sloop* and *Sexit* and finally pre-allocation of the local variables of table type or scalar in the stack, whose address is used. These are not very difficult but not trivial either.

Cminor is simplified as much as possible, under the constraint that it must remain able to express almost any ANSI C program. This will of simplification is apparent in the encoding of the loops in terms of *Sblock, Sloop* and *Sexit*, and in the elimination of the operator &. This last point not only makes it possible to simplify the semantics as the environment associates the local variables values and not of the addresses memory containing these values, but also the translation in lower level languages, for example a local variable Cminor is translated directly into a virtual register of language CFG-RTL. In spite of that, the result language of the experiment is quite heavy and complex. A good share of this heaviness is due to the great number of different operators in the expressions.

We do not provide the Cminor semantics in this dissertation, as our main intention is to write a compiler to convert the Cminor programs to an intermediate language, Register Transfer Language. For us, a description of Cminor and RTL suffices. We should mention that the techniques we have shown in the previous chapters are intensively used in the description of Cminor semantics, specially the technique we described in the chapter 2 to describe nested recursion for partial functions. In Cminor semantics, mutual recursions are expressed in a similar way. Another notable point is that the technique we described in the section 3.4.2 is the most practical solutions till now for the proofs where one has to consider executions of subterm that are not direct subterms.

## 4.3 Register Transfer Language

The Register Transfer Language or RTL is described as a control flow graph. The nodes of this graph are the elementary operations. RTL operations are close to the machine instructions. The arguments and results of the operations are stored in the registers. We consider two kinds of registers, viz. real and virtual. The real registers correspond to those of the processors and the virtual registers are those variables which are later replaced by placing their values in the real register or in the stack. The RTL we describe here can have infinite number of registers in theory. But in practice, to make the proofs easier we need to keep track of the maximum number of registers used. We try to optimize the number of registers as much as possible in the transformation from Cminor to RTL. Later RTL registers are allocated to the real registers by graph coloring.

Arcs of the RTL graph represent the flows of the control in the graph and give the successor operations. If the operation is arithmetic then there should be one successor that is for the next operation to be followed. For the conditional branches there are two successors, one for the *true* case and the other for the *false* case of the condition. For the selective structure or *switch* there are $n$ successors, one for each possible case of the switch expression and for *return* statement of a function there is no successor operation, as the *return* statement sends back the control to the caller of the function.

### 4.3.1 Operations in RTL

The RTL operations are close to the machine or processor operations. We have chosen RISC kind processor close to the PowerPC for the target language. We try to keep the RTL operations independent of the processor operations. But, we will later show how we exploit the special operations of PowerPC and how the design changes for that exploitation. The operations in the RTL language can be broadly classified in five categories. They are arithmetic operations involving integers and float values, conversions between integers and float values, pointer operations, operations on the memory addresses and the move operation. Arithmetic operations are further classified in two categories, one involves integer values and the other involves float values. We separate the arithmetic operations with immediate argument from the arithmetic operations with both arguments in registers. Integer operations can have signed or unsigned integers as arguments. Here is the abstract syntax for the operations in the RTL intermediate language.

```
Inductive operation : Set :=
 | Omove           : operation
 | Oconst_int      : intval -> operation
 | Oconst_float    : floatval -> operation

                  (* operations on the memory addresses *)

 | Oaddrglobal     : gident -> intval -> operation
 | Oaddrstack      : intval -> operation

                  (* Integer arithmetic *)

 | Osignextend8    : operation
 | Osignextend16   : operation
 | Oadd            : operation
 | Oadd_imm        : intval -> operation
 | Osub            : operation
 | Osub_imm        : intval -> operation
 | Omul            : operation
 | Omul_imm        : intval -> operation
 | Odiv            : operation
 | Odiv_imm        : intval -> operation
 | Odivu           : operation
```

```
  | Oand            : operation
  | Oand_imm        : intval -> operation
  | Oor             : operation
  | Oor_imm         : intval -> operation
  | Oxor            : operation
  | Oxor_imm        : intval -> operation
  | Onand           : operation
  | Onor            : operation
  | Onxor           : operation
  | Oandnot         : operation
  | Oornot          : operation
  | Oshiftleft      : operation
  | Oshiftright     : operation
  | Oshiftright_imm : intval -> operation
  | Oshiftrightu    : operation
  | Orlwinm         : intval -> intval -> operation

                   (* Floating-point arithmetic *)

  | Onegf           : operation
  | Oabsf           : operation
  | Osingleoffloat  : operation
  | Oaddf           : operation
  | Osubf           : operation
  | Omulf           : operation
  | Odivf           : operation
  | Omuladdf        : operation
  | Omulsubf        : operation

                    (* Conversion between int and float *)

  | Ofloatofint     : operation
  | Ofloatofintu    : operation
  | Ointoffloat     : operation

                    (* Pointers operations *)

  | Oaddptr         : operation
  | Oaddptr_imm     : intval -> operation
  | Osubptr         : operation

                    (* Boolean tests *)

  | Ocomp           : condition -> operation.
```

#### 4.3.1.1 Arithmetic operations

*Osignextend8* and *Osignextend16* returns 8 bit and 16 bit sign extension of the argument, respectively. The rightmost bit of the integer is considered as the sign bit. Note that these two operations correspond directly to the *OPcast8signed* and *OPcast16signed* operations in Cminor. To evaluate the *Osignextend8* or *Osignextend16* operations in RTL we split the integer in four bytes of 8 characters and return the rightmost or the two rightmost bytes, respectively.

We take advantage of PowerPC processor instructions, where operations with immediate arguments are treated separately from the operations with all the arguments in registers. For example, to add two integers we *Oadd* and *Oadd_imm* operations. The *Oadd* operation expects both its arguments in registers, whereas *Oadd_imm* expects one of its two arguments in register and the other as immediate value. Similarly for *subtraction, multiplication, division*, boolean *and*, boolean *or* and boolean *xor* operations. The division operation between two unsigned integers is special because of the sign bit, we keep it separate as *Odivu*.

We keep the similarity of our RTL operations with the PowerPC instructions set. For that reason we do not have a binary not operations, rather we have *Onand, Onor* and *Onxor*. We also do not have boolean not operation and instead we have *Oandnot* and *Oornot*. The last two operations are in general more often used and they reduce the code length. For binary shift and rotation operations we have *Oshiftleft, Oshiftright, Oshiftright_imm, Oshiftrightu* and *Orlwinm*. Note that we do not have any *Oshiftleft_imm* and *Oshiftleftu* operation. These choices are based on PowerPC instruction set. In fact, PowerPC instruction set is well optimized. The *Oshiftleft_imm* operation can be achieved with the *Orlwinm* operation, which is more often used. The reason behind not having the *Oshiftleftu* is rather simple. When we shift left an integer we push zeros from the right, which has no effect on the sign bit. So a *Oshiftleft* operation does equally well with both signed and unsigned integers. The *Oshiftrightu* does not care the sign bit and always pushes zero on on the left, whereas the *Oshiftright* operation pushes the sign bit from the left.

The operation *Orlwinm* takes two arguments other than the principal argument on which the operation is to be applied. This is a combination of left rotation and mask operation. The first argument provides the amount by which the principal argument should be left rotated before masking by the integer provided in the second argument. This operation will be very much exploited in some operations, like *and, or* and *shift*, with immediate arguments.

Floating point arithmetic operations consist of negation, abstraction, addition, subtraction, multiplication, division etc. The operation *Osigleoffloat* is type casting operation, it converts a double precision floating point number to a single precision floating point number. We exploit some of the special PowerPC operations. For example, there is a built-in multiply and addition operation. Expressions like $e_1 * e_2 + e_3$ can be, therefore, directly translated in RTL instructions *Omuladdfloat* $e_1$ $e_2$ $e_3$. Similarly *Omulsubf* is a built-in multiply and subtraction operation.

#### 4.3.1.2 Type cast, pointer arithmetic and other operations

We have three type casting operations between integers and floating point numbers. *Ofloatofint* converts a signed integer to a floating point number, *Ofloatofintu* converts an unsigned integer to a floating point number and *Ointoffloat* converts a floating point number to a signed integer.

Pointer operations include addition, subtraction and comparison. We grouped together all the comparison operations in the operation *Ocomp*, where the *condition* describes the type of the comparison operation. *Condition* is described as follows.

```
Inductive condition : Set :=
 | Ccomp        : comparison -> condition
 | Ccompu       : comparison -> condition
 | Ccomp_imm    : comparison -> intval -> condition
 | Ccompu_imm   : comparison -> intval -> condition
 | Ccompf       : comparison -> condition
 | Cnotcompf    : comparison -> condition
 | Cmaskzero    : intval -> condition
 | Cmasknotzero : intval -> condition
 | Ccompptr     : comparison -> condition.
```

Remember that we have already defined *comparison* in the section 4.2. The operations *Ccomp, Ccomp_imm* or *Ccompu, Ccompu_imm* compare two integers signed or unsigned, without or with immediate argument. Similarly, *Ccompf* compares two floating point numbers and *Cnotcompf* compares two floating point numbers for non equality. PowerPC built-in mask operations can be exploited to test the results of *and* operations with constants. The outcome of such results can be either zero or nonzero. RTL operations *Cmaskzero* and *Cmasknotzero* describes this operation. For the mask operation we need to provide the amount as an argument by which it should be masked. Pointer comparisons are done using the *Ccompptr* operation.

The *Oaddrglobal* and *Oaddrstack* operations involve the memory locations. In fact each global symbol is associated to a pointer and an offset. The *Oaddrglobal* operation takes two arguments, a global symbol and an offset. It finds the pointer associated to the global symbol and adds the offset provided with the *Oaddrglobal* to the offset already associated to the global symbol. This is very useful for array data structure. The *Oaddrstack* operation takes an offset and adds it to the address of the stack pointer. This operation is very useful in stack operation.

### 4.3.2 Memory address calculation in RTL

The addressing mode type describes how the address of a memory load or store is computed from the argument registers.

```
Inductive addressing_mode : Set :=
 | Aindexed     : addressing_mode
 | Aindexed_imm : intval -> addressing_mode
 | Abased       : gident -> intval -> addressing_mode
 | Abased_imm   : gident -> intval -> addressing_mode
 | Ainstack     : intval -> addressing_mode.
```

We consider five types of relative addressing mode. To access an array using a static address we have indexed addressing mode, given by the constructor *Aindexed_imm*. In this addressing mode the starting address of the array is given in the register and an offset is provided along with the constructor. For example, to access a variable $a[3]$, where $a$ is an array of more than 4 places, we should write *Aindexed_imm* 3 and should provide address of the beginning of the array $a$ in the register. To access an array using dynamic address we have base-indexed addressing mode, given by the constructor *Aindexed*. In this addressing mode both the starting address of the array and the distance of the accessed element from this staring address are provided in registers. For example, to access a variable $a[n]$, where $a$ is an array of more than $n$ places, we should write *Aindexed* and should provide address of the beginning of the array $a$ and the value of $n$ in the registers.

We also have base-displacement addressing, given by the constructor *Abased*. The base address and the distance are provided along with the constructor. In base-indexed-displacement, given by the constructor *Abased_imm*, the base address and the distance are provided along with the constructor and the indexed address are provided in the register. For example, to access a global variable $a$ we should write *Abased_imm* $a$ 0. Finally, we calculate the addresses in the stack with the constructor *Ainstack*, where we provide the displacement in the stack along with the constructor. This displacement is added to the stack pointer to get the address of an element in the stack.

### 4.3.3 RTL Instructions

An instruction in the Register Transfer Language is described by three fields. The first constructor *instr_desc* describes the operation to be performed.

List of arguments needed for this operation is provided in the second constructor *instr_args* as a list of registers and to store the final value returned by the instruction the third constructor *instr_res* provides a register. The following record describes the instruction in the Register Transfer Language.

```
Record instruction : Set := mkinstr {
   instr_desc  : instruction_desc;
   instr_args  : (list Reg.T);
   instr_res   : (option Reg.T)
 }.
```

Note that we described the register *instr_res* by an option type as some instructions may not return any value.

In Cminor, implicitly all instructions have a list of successors. Most of the instructions fall through, in other words they continue with the next instruction. Only the branching instructions in Cminor do not fall through.

Each Cminor instruction describes a graph in RTL, which is in fact a subgraph of the graph describing the whole program. The list of successors of an instruction are made explicit in RTL. In other words, each node in RTL graph is connected to another node, which contains the successor instruction. When the Cminor branching instructions are transformed to RTL, they are connected to more than one node, one of which should be followed.

The *Inop instruction* does nothing and just branches to its successor. The instruction *(Iop op)* performs the operation *op* with the arguments given in the register list *instr_args* and returns the final value of the operation in the register *instr_res*.

The instruction *(Iload chunk adrmode)* loads the memory fragment chunk at the address determined by the addressing mode *adrmode* and stores the value just read in the register *instr_res*. The arguments which are needed to determine the memory location for this addressing mode are provided in the list of registers *instr_args*. Similarly, the instruction *(Istore chunk adrmode)* stores the value of the register *instr_res* in the memory fragment chunk and the address of the memory location is determined by the addressing mode adrmode. All these instructions are non-branching instructions and continue with their successors.

Among the branching instructions we have instructions for function call and conditions. The instruction *Icall* and *Icall_imm* performs the function call taking the arguments of the function from the list of registers *instr_args* and returns the result of the function call in the register *instr_res*. The function is called either through a pointer given in the register list, in case of *Icall* or by a direct call, in case of *Icall_imm* where the address is directly provided. One can consider that a function call makes a bigger node in the control graph which contains all the operations of the function and then follows the next instruction in the program.

The instruction *Icond cond* tests the condition *cond* with the arguments given in the list of registers *instr_args*. This instruction has two successors, for true and false values of the condition. The *Iswitch li* instruction the number of successor depends on *li*. *Iswitch* discriminates on the value provided in the register list and matches the value in the list *li* and goes to the corresponding node. The instruction *Ireturn* returns to the caller function. It returns the value stored in the register *instr_res*. For C functions which return *void*, the instruction *Ireturn* returns *None*. *Ireturn* has no successor.

```
Inductive instruction_desc : Set :=
 | Inop          : instruction_desc
 | Iop           : operation -> instruction_desc
 | Iload         : cmemory_chunk -> addressing_mode
                                   -> instruction_desc
 | Istore        : cmemory_chunk -> addressing_mode
                                   -> instruction_desc

(* branching operation *)
 | Icall         : instruction_desc
 | Icall_imm     : gident -> instruction_desc
 | Icond         : test -> instruction_desc
 | Iswitch       : (list intval) -> instruction_desc
 | Ireturn       : instruction_desc.
```

### 4.3.4 Functions and programs in RTL

A function is represented by the following record.

```
Record fundecl : Set := mkfun {
   fun_args        : (list Reg.T);
   fun_max_reg     : Map.range;
   fun_code        : (Graph.T instruction);
   fun_entry_point : Graph.key;
   fun_stackspace  : size
 }.
```

The list of arguments to the function is provided by the list of registers *fun_args*, which contains the values of the arguments in the registers. *fun_code* is the graph of instructions for this function and entry point of this function is also provided by *fun_entry_point*. Type checking ensures that in a function definition, number of the registers should be less than *fun_max_reg*. Type checking also ensures that the graph for this function is not disjoint from the graph presenting the program and the entry point for this function is a vertex of this graph.

A program is a collection of functions. The name of the distinguished *main* function should be provided. Global variables are also provided as a list tuples, name and its size in bytes. Type checking ensures that all the global identifiers are different. Abstract syntax of a program is given by the following record.

```
Record rprogram : Set := mkrprog {
   rprog_fun    : (list gident * fundecl);
   rprog_main   : gident;
   rprog_vars   : (list gident * size)
 }.
```

## 4.4 Conclusion

We tried to show the links between the Cminor instructions and their corresponding RTL instructions. These will help us to understand the transformation from Cminor programs to RTL programs. In the next chapter we show how to transform a Cminor program to a RTL program. It is to be noted that the RTL language is very close to the target language of RISC kind PowerPC processors. We need to go through a lot of transformations before producing the target code. These transformations will consist of optimization, register allocation, constant propagation, linearization etc. In that sense RTL plays a significant role in the formalized C compiler.

In the collective project *Concert*, Laurence Rideau and Sandrine Blazy implements the Cminor semantics and Benjamin Grégoire realizes the memory model and the RTL semantics. The compiler described in the next chapter connects their works and the implementation of the compiler depends very much on the formulation of Cminor and RTL they propose. In the next chapter, we will show how much we can push the expressivity of the restrictive language of the **Coq** proof system to keep the modularity of the compiler. Keeping the modularity of the compiler is helpful to add future changes in the Cminor and the RTL.

Chapter 5

# An experimental compiler for Cminor to RTL - II

## 5.1 Cminor to RTL translation

In this chapter we present how do we transform the Cminor programs to RTL programs and construct the control flow graph in RTL. In Cminor, the environment contains all the local and global variables, keeps track of the stack pointer and maintains the memory. This environment is necessary for the semantics of the Cminor language. We describe the environment of Cminor as follows.

```
Record env : Set := mkenv {
   genv : (assoc gident value);
   lenv : (assoc lident value);
   stack : ptr;
   store : cmemory
```

where *genv* is a list associating a value to each global variable, the *lenv* is a list associating a value to each local variable, *stack* is a pointer which contains topmost address of the stack and *store* is the memory where we keep the instructions, arrays etc. In other words, the environment in Cminor represents the state of the program and therefore the execution of the instructions is very much dependent on the environment.

Similarly the RTL semantics depends on the state of the program. To transform the Cminor programs into RTL programs, it is therefore necessary to transform the Cminor environment to its RTL counterpart, which will be later needed to show the equivalence between the Cminor program and its RTL translation. To facilitate this proof of equivalence we keep the memory structure in RTL mostly unchanged. In the previous chapter we told that in RTL all the local variables are stored in the registers and theoretically we can use as many as registers we need. Thus in RTL we keep the local variables in registers and the rest of the memory structures of Cminor are kept unchanged in RTL. Note that the translation of memory structure is needed for the semantics, not for the syntax

## 5.2 The translation environment

The translation environment contains all the necessary information about the RTL control flow graph. We start with the mapping of the Cminor local variables to RTL registers. The intermediate results of any computation are also kept in the registers. Cminor instructions are translated to RTL instructions and then mapped to the nodes of the control flow graph. It is necessary to know which of the variables are often accessed in a loop and for that reason we keep track of the loop nesting depths. This information helps in later optimizations in register allocations. Note that for this compiler we consider theoretically an infinite number of registers. When we finally produce machine code they run on a fixed number of registers, thus we have to go through the register allocation. We can significantly reduce the compilation time if the variables inside a loop are kept in registers, as the memory access will cost more time. Nodes of the control flow graph are mapped to loop nesting depths, as each node corresponds to an instruction in RTL and we keep the loop nesting depth of the instruction along with the instruction.

Later in this chapter we show the functions to translate Cminor expressions, Cminor conditions, Cminor instructions to their RTL counterparts. The translated environment is threaded through all the translated functions. Following is the translation environment in RTL.

```
Record env: Set := mkenv {
   vars      : (Stringmap.T Reg.T);
   nextreg   :  Map.key;
   instrs    : (Graph.T instruction);
   loops     : (Map.T positive);
   nesting   :  positive
}.
```

where *vars* is the list of registers containing all the local variables and intermediate results. To assign a new register we keep track of the key of the last used register in *nextreg*. The *instr* is the control flow graph in RTL we keep on building along the translation from Cminor to RTL. Similarly, *loops* keeps track of the depth of nesting of the instruction in the graph and *nesting* helps in giving the current depth of the nesting.

We initialize the above translated environment by providing zero or empty values to each of them.

```
Definition init_env :=
  (mkenv (Stringmap.init Reg.T (Reg.make xH TPint)) xH
    (Graph.empty (mkinstr Inop (nil Reg.T) (None Reg.T)))
      (Map.init xH) xH).
```

For each Cminor variable we create a pseudo register in RTL with the help of the following function.

```
Definition add_var
 [envir:env; key:string; ty:typ]:(env*Reg.T) :=

  let r=(Reg.make (nextreg envir) ty) in
  let envir'=(mkenv
          (Stringmap.set Reg.T key r (vars envir))
          (add_un (nextreg envir))
          (instrs envir) (loops envir) (nesting envir)) in
    (envir', r).
```

Thus, it creates a new register of the proper type, stores the value of the local variable and modifies the environment.

We also create a pseudo register for each intermediate result in an execution. Given a type the following function will create a pseudo register with that type and return the enriched environment.

```
Definition new_reg [envir:env; ty:typ]:(env*Reg.T) :=
 ((mkenv (vars envir) (add_un (nextreg envir)) (instrs envir)
               (loops envir) (nesting envir)),
               (Reg.make (nextreg envir) ty)).
```

Each Cminor instruction is translated to RTL instructions and added to the control flow graph. Every node in the control flow graph corresponds to a RTL instruction. To add an instruction in the control flow graph we need to provide a list of successors of that node. We then create a node for that RTL instruction, determine the loop nesting depth of that instruction and then return the modified environment and the key of that node. Remember that the environment keeps all the information about the control flow graph, list of registers used and the depth of the nested loop instruction. The following function describes how we add an instruction in the control flow graph.

```
Definition add_instr
    [envir:env; instr:instruction; succs:(list Graph.key)]:
                             (env*Graph.key) :=
  let (n, code) =
         (Graph.add_vertice instr succs (instrs envir)) in
  let envir' =
    (mkenv (vars envir) (nextreg envir) code
        (Map.set n (nesting envir) (loops envir))
                 (nesting envir)) in
     (envir', n).
```

The compilation process which compiles Cminor programs to RTL programs builds the control flow graph in a backward direction. In other words, for an expression we create a node for the top level operation, but this operation depends on the subexpressions of the expression. This node is fully constructed after its successor nodes, which correspond to its subexpressions, in the control flow graph is constructed. We discuss the construction of the control flow graph in the section .

For a loop we cannot add a node in the control flow graph straightaway. We can consider that each loop consists of a head part and a body part. We enter the loop through the head part and then the body of the loop follows. If it loops again, the head follows the body and so on. We leave the loop from the body part. Thus we cannot provide the successor node for the head part of the loop before creating the nodes for the body part of the loop. To construct loops, we break the *add_instr* in two steps. The first step *reserve_instr* just allocates a node in the control flow graph and returns the key of that node, without associating it with an instruction. The second step *update_instr* sets the instruction at the given node. We provide the list of successors in the second step, thus in the first step the created node is not connected to the graph. Below we present these two functions.

```
Definition reserve_instr [envir:env]:(env*Graph.key) :=
     (add_instr envir
                (mkinstr Inop (nil Reg.T) (None Reg.T))
                  (nil Graph.key)).

Definition update_instr
           [envir:env; key: Graph.key; instr:instruction;
                     succs:(list Graph.key)]:env :=
 let code = (Graph.update key instr succs (instrs envir)) in
 (mkenv (vars envir) (nextreg envir) code
   (Map.set key (nesting envir) (loops envir))
                                    (nesting envir)).
```

We record the current loop nesting in the environment in *nesting* of the translated environment. Each time we enter or leave a loop we modify the current loop nesting and this information on depth is later used in *update_instr*. The following functions *enter_loop* and *leave_loop* respectively increment and decrement the current loop nesting.

```
Definition enter_loop [envir:env]:env :=
 (mkenv (vars envir) (nextreg envir) (instrs envir)
               (loops envir) (add_un (nesting envir))).

Definition leave_loop [envir:env]:env :=
 (mkenv (vars envir) (nextreg envir) (instrs envir)
               (loops envir) (sub_un (nesting envir))).
```

## 5.3 Allocation of registers for intermediate results

To keep the intermediate results for an expression we need to allocate a register. Given an expression, the following function *alloc_reg* allocates a fresh pseudo register having the same type as the expression and therefore can hold the value of the expression. Since the environment records the list of registers, the function returns the modified environment and the new allocated register.

```
Definition alloc_reg [envir:env; e:expr]:(env*Reg.T) :=
                     (new_reg envir (type_of_expr envir e)).
```

where *type_of_expr* determines the type of a Cminor expression. To determine the type of an expression we also need to determine the type of unary and binary operations and memory chunks used to store the variable in the memory, as they are interleaved with the expression. We determine the type of a Cminor expression with the following functions.

```
Definition type_of_unop [unop: unary_operation]: typ :=
Cases unop of
   OPnegint         => TPint
 | OPnegfloat       => TPfloat
 | OPabsfloat       => TPfloat
 | OPintoffloat     => TPint
 | OPfloatofint     => TPfloat
 | OPfloatofintu    => TPfloat
 | OPnotbool        => TPint
 | OPnotint         => TPint
 | OPcast8signed    => TPint
 | OPcast8unsigned  => TPint
 | OPcast16signed   => TPint
 | OPcast16unsigned => TPint
 | OPsingleoffloat  => TPfloat
end.

Definition type_of_binop [binop: binary_operation]: typ :=
Cases binop of
```

```
   OPaddptr         => TPaddr
 | OPsubptr         => TPint
 | (OPcmpptr _)     => TPint
 | OPaddint         => TPint
 | OPsubint         => TPint
 | OPmulint         => TPint
 | OPdivint         => TPint
 | OPmodint         => TPint
 | OPdivintu        => TPint
 | OPmodintu        => TPint
 | OPandint         => TPint
 | OPorint          => TPint
 | OPxorint         => TPint
 | OPshiftleftint   => TPint
 | OPshiftrightint  => TPint
 | OPshiftrightintu => TPint
 | OPaddfloat       => TPfloat
 | OPsubfloat       => TPfloat
 | OPmulfloat       => TPfloat
 | OPdivfloat       => TPfloat
 | (OPcmpint _)     => TPint
 | (OPcmpintu _)    => TPint
 | (OPcmpfloat _)   => TPint
end.

Definition type_of_chunk
              [chunk: memory_chunk]: typ :=
Cases chunk of
   Mint8signed      => TPint
 | Mint8unsigned    => TPint
 | Mint16signed     => TPint
 | Mint16unsigned   => TPint
 | Mint32           => TPint
 | Mfloat32         => TPfloat
 | Mfloat64         => TPfloat
end.

Definition type_of_cmchunk
              [cmchunk: cmemory_chunk]: typ :=
Cases cmchunk of
   (Mchunk chunk)      => (type_of_chunk chunk)
 | Maddr               =>  TPaddr
end.
```

```
Fixpoint type_of_expr
              [env: env; ex: expr]: typ :=
Cases ex of
   (Evar id)                => (Reg.reg_type (find_var env id))
 | (Eaddrsymbol _)          => TPint
 | (Eaddrstack  _)          => TPint
 | (Eassign id e)           => (Reg.reg_type (find_var env id))
 | (Econstint _)            => TPint
 | (Econstfloat _)                 => TPfloat
 | (Eunop unop e)                  => (type_of_unop unop)
 | (Ebinop binop e1 e2)            => (type_of_binop binop)
 | (Eload chunk adr)               => (type_of_cmchunk chunk)
 | (Estore chunk addr data)        => (type_of_cmchunk chunk)
 | (Ecall fn args None)            => TPint
 | (Ecall fn args (Some tyres))    => tyres
 | (Eandbool e1 e2)                => TPint
 | (Eorbool e1 e2)                 => TPint
 | (Econdition e1 e2 e3)           => (type_of_expr env e2)
 | (Esequence1 e1 e2)              => (type_of_expr env e1)
 | (Esequence2 e1 e2)              => (type_of_expr env e2)
end.
```

## 5.4 Instruction selection

We decompose each Cminor expression into an RTL operation, condition or addressing mode and a list of sub-expressions in such a way that the operation applied to the values of the subexpressions computes the value of the original expression. Instead of doing directly this decomposition of Cminor expression into RTL expressions we divide them into five different functions. The first function *transl_cond* decomposes a Cminor condition *e* in an RTL condition *cond* and a list of subexpressions of *e*. Applying *cond* to the values of the list of these subexpressions produces the same truth value as evaluating *e*. The second function *transl_unop* takes a Cminor expression for unary operation *e = Eunop(unop e1)* and decomposes it to an RTL operation and a list of subexpressions of *e*.

The third function *transl_binop* takes a Cminor binary operation *e = Ebinop(binop e1 e2)* and decomposes it to an RTL operation and a list of subexpressions of *e*. The fourth function *transl_addressing* decides the addressing mode in RTL for a corresponding expression in Cminor. Finally, the fifth function *transl_expr* takes a Cminor expression, translates it to a RTL instruction, creates a node in the control flow graph, assigns the RTL instruction to that node and finally connects it to the control flow graph. Thus the first four functions can be viewed as subparts of the fifth function.

In fact we need to define each of the first four functions as mutually recursive with the fifth function, we will discuss it later in section. The function *transl_expr* is also mutually recursive with another function *transl_condition*. The function *transl_condition* checks the priority of branches in a branching instruction and sets the structure of the control flow graph accordingly. We discuss the priority of branches in section.

In general, the translation functions for unary and binary operations are simpler as most of the time the RTL operation is equivalent to the top-level operator and sub-expressions are just the argument of the top level operator. For instance, consider the Cminor expression (*Ebinop OPaddint* $e_1$ $e_2$) for the addition of two expressions $e_1$ and $e_2$, where each of these expressions return integer values. This Cminor expression when transformed to RTL instruction, it is decomposed into the operation Oadd and the subexpressions $e_1$ and $e_2$. Each of these subexpressions are then recursively transformed to RTL instructions.

However we decided to take advantage of complex instructions of PowerPC, which are already included as special operations of RTL, to decompose Cminor expressions. In section, we gave the example of built-in multiply add operation for C expression $e_1 * e_2 + e_3$, where each of $e_1$, $e_2$ and $e_3$ returns a float value. This expression is translated in Cminor as *(Ebinop OPaddfloat (Ebinop OPmulfloat* $e_1$ $e_2$*)* $e_3$*)*. In the transformation from Cminor to RTL, we try to decompose Cminor expressions more aggressively and exploit the complex PowerPC operations, thus we transform the above using RTL operation *Omuladdfloat* and recursively transforming the expressions $e_1$ and $e_2$.

Using such instructions needs special care as we need to reorient the list of arguments (or subexpressions in *transl_binop*). This is because we would like to follow the order or computation over the results of subexpressions and these instructions have fixed computation order. For instance, Cminor expressions *(Ebinop OPaddfloat* $e_1$ *(OPmulfloat* $e_2$ $e_3$*))* can be transformed to RTL instruction *Omuladdfloat* $e_2$ $e_3$ $e_1$. Thus, a simple change in the order of the subexpressions is enough to transform such expressions isomorphically.

In another case, to transform an unary Cminor *OPnotint* operation we lack a *binary not* operation in PowerPC. PowerPC provides a more general *not or* operation and this can be used to achieve *binary not* operation, since $\tilde{x} = \tilde{(}x \| x)$. Therefore we need to duplicate the expression, and then apply the RTL operation *Onor* to these two expressions.

Other than these few special operations RTL operation is to be applied to a list of argument registers that is isomorphic to the list of subexpressions. These suggest that we need three register list transformers. First, the *reg_id* is the identity function, in other words it returns the same list of registers. This function will be used in most of the operations. Second, the *reg_dup* duplicates the register. Its only use is the operation is *OPnegint*. Third, the *reg_123_231* rotates a 3-register list by one left shift and its only use is

*Omuladdfloat*. Here are the three register list transformers.

```
Inductive reg_list_trans: Set:=
   reg_id :  reg_list_trans
 | reg_dup :  reg_list_trans
 | reg_123_231 : reg_list_trans.

Definition eval_reg_list_trans
    [rt: reg_list_trans; l:(list Reg.T)]:(list Reg.T) :=
 Cases rt of
     reg_id => l
   | reg_dup =>
        Cases l of
            nil => l
          | (cons r rl) =>
                Cases rl of
                    nil => (cons r (cons r (nil Reg.T)))
                  | _ => l
                end
        end
   | reg_123_231 =>
      Cases l of
        nil => l
     | (cons r1 rl1) =>
         Cases rl1 of
           nil => l
        | (cons r2 rl2) =>
            Cases rl2 of
              nil => l
           | (cons r3 rl3) =>
               Cases rl3 of
                nil =>
                (cons r2 (cons r3 (cons r1 (nil Reg.T))))
              | _ => l
               end
            end
         end
      end
 end.
```

## 5.5 Code linearization

To execute the machine code that we generate from the RTL control flow graph in the RISC kind PowerPC processor, which is essentially sequential, we need to transform the RTL control flow graph into a linearized sequence of instructions. It is important to linearize the code in a careful manner as it will save the run time costs, though any linearization works for the processor.

Consider the following block of instructions in C.

```
i1;
while b
         i2;
```

where $i1$ and $i2$ are list of instructions and $b$ is a boolean expression.

When we try to produce the machine code for the above piece of source code in C, we have certain restrictions in the machine language. For example, the machine language does not contain a construct *while* and we need to express the above with the help of condition, gotos and labels. We gave some examples in this direction for transforming the code from Cminor to RTL in the section... Thus, the above piece of code can be rewritten as

```
     i1;
l1:  if !b then l2;
     i2;
     goto l1;
l2:  ...
```

where the jump instruction is built-in inside the *condition* (if-then). If the *while* construct repeats itself 10 times, we generate 10 gotos during runtime. We can optimize the code further by significantly reducing the number of gotos in the target code, if we exploit the target language in a more intelligent way. For instance, the above target code can be rewritten as

```
     i1;
     goto l1;
l1:  i2;
l2:  if b then l1;
...
```

If the *while* construct repeats itself 10 times, we execute only 1 goto during the runtime, therefore we reduce the number of gotos by 9.

Linearization of the code transforms the control flow graph of RTL instructions into a linearized sequence of instructions, inserting labels and gotos to express the flow of control. We remove the *Inop* instructions from the RTL program. In the beginning, each instruction in the linearized code has a label, which corresponds to the key of the node in the RTL graph. Other than the branching instructions, described in section 4.3.3, all instructions continues with their next nodes in the control flow graph. For the branching instructions we need to decide which of its successors should be chosen for the fall through case. This choice depends on the heuristics we present in the following section. We insert gotos for the other branches. Finally we remove the labels that are never used as the target of a branch instruction.

## 5.6 Heuristics for static branch prediction

Static branch prediction guesses which of the arms in a conditional executes most often. This prediction helps during linearization of code. We follow few heuristics to predict the branch to be more often executed in a conditional. In the first heuristic we adopt the convention that in RTL condition *Icond* execution of the *else* branch is more likely than the *then* branch. If it is not so, we can swap the two branches and negate the condition to meet the above convention. We implement the static branch prediction strategy proposed by Ball and Larus in the article *Branch prediction for free.* The function *likeliness_stmts* returns a probability that a list of statements is executed based on the following heuristics.

- An arm that contains a *return* statement is less likely to be executed than an arm that does not contain a *return.* Because early returns often correspond to error cases or base cases of a recursion.
- An arm that exits a loop, for instance *exit* in RTL, is unlikely to be executed because normally loops iterate several times.
- An arm that contains a function call is less likely to be executed than an arm that does not contain any function call. In general, function call often corresponds to the error reporting.

Another heuristic says that while comparing two floats, two pointers or two unsigned integers for equality in general the two numbers are not equal. In case of a conditional we compare the probability returned by the function *likeliness_stmts* for each of its arms, this gives us a first estimate. Finally the function *likeliness_cond* decides which arm of the conditional will be chosen based on this probability estimation and the last heuristic. It is to be noted that this transformation preserves semantics thus the correctness of the compiler is not compromised.

## 5.7 Guardedness problem

We have seen earlier that the functions *transl_cond, transl_unop, transl_binop* and *transl_addressing* are called in the function *transl_expr*. For instance to translate a Cminor instruction *Eunop OPnegfloat* $e_1$ we call the function *transl_unop.* The *transl_unop* decomposes this expression into a RTL operation and a list of subexpressions, to be precise *Onegfloat* and a list containing

$e_1$. The *transl_expr* should be called again recursively on the list containing $e_1$. But guardedness condition of Coq system does not allow it.

In the Calculus of Inductive Constructions we need to explicitly state what is the decreasing argument in a recursive function. The guardedness condition of Coq is satisfied when the actual value of the decreasing parameter for every recursive call is smaller than the one of the previous iteration. In the Calculus of Inductive Constructions the principal argument of a structural recursive function is guarded. In a case expression over an inductive type the guardedness continues through the constructor of the inductive type. For instance, if we have an inductive type $t$ with two constructor,

```
Inductive t:Set :=
    c1 x : ...
  | c2 y : ... .
```

then in a case expression of the following pattern

```
...
 Case t of
    c1 x => f x
  | c2 y => g y
 end.
```

both the $x$ and $y$ are guarded in $f$ and $g$ respectively. Among the other guardedness conditions,

In our case it is necessary to write the functions in a mutually recursive way. For instance, to translate the Cminor expression *Ebinop OPaddint* $e_1$ $e_2$ we should select the corresponding binary operation in RTL, thus we need to call the function *transl_binop.* The function *transl_binop* decomposes the Cminor expression in two subexpressions, $e_1$ and $e_2$ and a RTL binary integer operation over the result of the two subexpressions. Finally we call the function *transl_subexpressions* to translate the list of the subexpressions returned by the function *transl_binop.* The function *transl_subexpressions* calls the function *transl_expr* for each of the subexpressions in the list. As the function returns a list of subexpressions to the caller function *transl_instr* we no more have a sub-term of the principal argument, instead we have a super-term. Even though we are structurally decreasing the principal argument, our translating function is not well defined.

A possible to way to solve this problem is to call the function directly to each of the subexpressions produced by the function *transl_binop.* Earlier, *transl_binop* used to return the RTL operation and a list of subexpressions to the caller function *transl_expr*, so that *transl_expr* can be called to each of these subexpressions. We have seen that such a solution creates a super term instead of a sub-term, and the Calculus of Inductive Constructions reject it. Again, it is difficult to contain each of the functions *transl_unop,* *transl_binop, transl_addressing, transl_condition* inside the *transl_expr*, as we loose the modularity of programming and such a solution makes the proof extremely complicated. Another possible way to keep the modularity of programming we can build the graph in each of these mutually recursive functions. Thus, when the control passes from one function to another, the callee function will modify the graph and finally will return the modified graph to the caller function. We exploit the higher order function of the Calculus of Inductive Constructions to build each of these functions in a non mutual way. We have exploited this fact in the chapter 2, we defined the functional to describe the denotational semantics of Imp.

For instance, the *transl_binop* functional now expects two more functions, in addition to its previous arguments. The first function is the *transl_expr* itself, which is called to build the control flow graph for each of the subexpressions. The second function builds the control flow graph for the RTL operation over these subexpressions. Remember that earlier the *transl_binop* function used to return a RTL operation and a list of subexpressions.

We need to modify each of the functions *transl_unop*, *transl_binop*, *transl_addressing*, *transl_cond* and *transl_condition* to exploit the higher order functions in the Calculus of Inductive constructions. Even though we could keep the modularity of the function and expressed the functions in non mutual way, each of these functions are quite big and complex. This is due to the vastness of the Cminor.

## 5.8 Construction of the RTL control flow graph

We construct the RTL control flow graph in a bottom-up left to right direction. We start with a Cminor instruction, decompose the instruction to generate the top level RTL operation and a list of subexpressions. We create a node for this RTL operation in the control graph, assign the RTL instruction to this node, modify the environment and connect it to its successor. For each of the subexpressions we create a node and connect them to the node which contains the top level RTL operation over these subexpressions. We repeatedly call our procedures to build the graph for each of these subexpressions. We take them from left to right in the list. For conditional branches, we consider the heuristics we have described in the section. The function *transl_condition* decides which branch is most likely and makes it as the leftmost child of the node. We stop when all the Cminor expressions are translated.

Let us take an example to understand better the way we construct the RTL control graph, the problems we encountered to model the compiler in type theory and the procedures we followed to overcome those problems. To construct the RTL graph from the Cminor expression *Ebinop OPaddint e1 e2* we call the function *transl_expr*. Note that, one of our major objective is to keep the modularity in the compiler code. The function *transl_expr* is a generalized function to construct the RTL control flow graph from any Cminor expression. Following is the type definition of the *transl_expr* function in the Coq proof assistant.

```
Fixpoint transl_expr [envir:env; expr1:expr]:
                        Reg.T-> Graph.key -> (env*Graph.key)
```

The *transl_expr* function expects the RTL environment in which the Cminor expression needs to be transformed, the Cminor expression, the list of registers where the Cminor variables and the intermediate results are stored and the node of the graph which should be executed after the current instruction (to be precise, this node should be executed after the execution of the RTL graph corresponding to the Cminor instruction we are considering), in other words the exit point for the Cminor instruction in the RTL control flow graph. The *transl_expr* function returns the new environment and a pointer to the node which should be executed next, in other words the entry point for the next Cminor instruction in the RTL control flow graph. For example, in our case this pointer should be the node for the expression *e2*. It will be clearer at the end of this discussion.

Ideally, we should have a piece of code to transform the binary Cminor operation into corresponding RTL control flow graph. This piece of code should first decide the corresponding RTL operation for the binary Cminor operation, in our case it is *Oadd*. Remember that earlier in the section 5.4 we discussed how we exploit the PowerPC instruction set. Thus, the function which decides the RTL operation, should also return a list of subexpressions by decomposing the Cminor expression, in our case *e1* and *e2*. Then, the piece of code should allocate registers to store the intermediate results for these subexpressions. Finally it should add a node for the RTL operation, in our case for the operation *Oadd* and recursively call the *trasl_expr* function to construct the RTL graph for each of those subexpressions. The piece of code should have the following definition in the Coq proof assistant.

```
(Ebinop binop e1 e2) =>
   let (reg_fn, opel) = (select_binop binop e1 e2) in
   let (op', el) = opel in
   let (envir', rl) = (alloc_regs envir el) in
   let (envir'', no) =
    (add_instr envir'
      (mkinstr (Iop op') (eval_reg_list_trans reg_fn rl)
              (Some Reg.T rd)) (cons nd (nil Graph.key))) in
       (transl_subexprs envir'' rl el no)
```

The function *select_binop* decides the RTL operation and decomposes the Cminor expression into a list of subexpressions and in the function *trasl_subexprs* we call the function *transl_expr* on each element of the list of expressions *el*. Though in reality we are calling the recursive function on arguments which are smaller than the initial argument, the list of expressions returned by the function *select_binop* does not appear as a subterm with respect to the guardedness conditions of the Coq proof assistant. In other words, the above definition of the constructor is not well defined.

To keep the modularity and the well formedness, we use the higher order function which we have already exploited in the chapter 2. Earlier the *transl_expr* was the sole function to construct the RTL control flow graph. In the new construct, we call the function *transl_binop* to construct the RTL control flow graph for a binary Cminor operation. Note that the functions *transl_expr* and *transl_binop* are inter dependent, as the *transl_expr* function constructs the RTL graph for each of the subexpressions. Exploiting the higher order function we assume that there exists a function *transl_expr* to construct the control flow graph for each of the subexpressions and there also exists another function which selects the exact RTL operation and decomposes the Cminor expressions into subexpressions. The later function also adds a node in the RTL control flow graph for the top level RTL operation. Here is definition of the constructor to construct the RTL control flow graph for the binary Cminor operation which is well formed and keeps the modularity of the compiler code.

```
(Ebinop binop e1 e2) =>
   (transl_binop transl_expr
       [op',rl,envf]
       (add_instr envf (mkinstr (Iop op') rl (Some Reg.T rd))
                   (cons nd (nil Graph.key)))
             envir binop e1 e2)
```

For any binary Cminor operation we need to add a node, which contains the corresponding RTL operation, in the control flow graph. This is common to every binary Cminor operations. We significantly reduce the amount of code by providing the common part when we call the function *transl_binop*. This can be observed in the above definition of the constructor. The Coq proof assistant facilitates the modular programming, which we exploit in the definition of the function *transl_binop*. Below we show the constructor we need to construct the control flow graph for the Cminor operation *OPaddint*.

```
Section SELECT_BINOP.

Variable transl_expr: env -> expr -> Reg.T
                          -> Graph.key -> (env*Graph.key).

Variable f: operation -> (list Reg.T)
```

```
                    -> env -> (env*Graph.key).

Definition  transl_binop
 [envbinop:env; binop:binary_operation;
              expbinop1,expbinop2:expr]:
                              (env*Graph.key):=
⋮

| OPaddint e1 e2 =>
   let (envbinop',r1) = (alloc_reg envbinop e1) in
   let (envbinop'',r2) = (alloc_reg envbinop' e2) in
   let (envbinop''',n2) =
    (f Oadd
      (eval_reg_list_trans reg_id
                   (cons r1 (cons r2 (nil Reg.T))))
         envbinop'') in
    let (envbinop'''',n1) =
              (transl_expr envbinop''' e2 r2 n2) in
      (transl_expr envbinop'''' e1 r1 n1)
⋮

End SELECT_BINOP.
```

In short, to construct the control flow graph for the Cminor expression containing binary operation we call the function *transl_expr*. This function calls *transl_binop*. The function *transl_binop* decomposes the Cminor expression into a few subexpressions and allocates registers for each of the subexpressions. Then the function *transl_binop* calls the function *f*, which decides the top level RTL operation and adds a node for this operation in the control flow graph. Finally the function *transl_binop* calls *transl_expr* to construct the control flow graph for each of the subexpressions it created.

In our example, to construct the RTL control flow graph for the Cminor expression *Ebinop OPaddint e1 e2* we call *transl_expr*. Let's say *nd* as the node to follow in the RTL graph. *transl_expr* calls the function *transl_binop*. The *transl_binop* function first allocates registers for the subexpressions *e1* and *e2*, say *r1* and *r2* consecutively, then calls the function *f*. The function *f* decides the top level RTL operation as *Oadd* and adds a node in the control flow graph for this operation. This operation will be performed on the results stored in the registers *r1* and *r2*. Let's say that this node is *nd0*. Thus the node *nd* will be followed after the node *nd0*. The function *transl_binop* then calls *transl_expr* to construct the RTL control flow graph for the Cminor subexpressions *e2* and *e1* respectively.

## 5.9 Conclusion

In a recent work[4] by Yves Bertot on co-inductive types, syntactic restrictions on the termination of computation on inductive structures or the guardedness conditions are discussed. Claudio Sacerdoti Coen also discusses the guardedness conditions in his PhD thesis [11] on knowledge management of formal mathematics in the **Coq** system.

The section system of the **Coq** proof assistant, along with the higher order functions can be exploited in other cases where the guardedness conditions pose constraints. Our technique is easy to implement and we expect this method to be widely used in the future in several areas of formalization.

Chapter 6

# General conclusion

In this dissertation we studied how to use functional semantics to represent the programming constructs with mutual and nested recursion inside type theory. We also look into the benefit from the functional descriptions of the programming language. Below we recapitulate our works and give some perspectives of these works.

In the first work, we describe the operational and denotational semantics of a small imperative language in type theory with inductive and recursive definitions. The operational semantics is given by natural inference rules, implemented as an inductive relation. The realization of the denotational semantics is more delicate as the nature of the language imposes a few difficulties on us. First, the language is Turing-complete, and therefore the interpretation function we consider is necessarily partial. Second, the language contains strict sequential operators and while loops, and therefore the function necessarily exhibits nested recursion. Our solution combines and extends recent work by the authors [1, 2, 10] and others[9, 14, 16] on the treatment of general recursive functions and partial and nested recursive functions. The first new result is a technique to encode the approach of Bove and Capretta[10] for partial and nested recursive functions in type theories that do not provide simultaneous induction-recursion. A second result is a clear understanding of the characterization of the definition domain for general recursive functions, a key aspect in the approach by iteration of Balaa and Bertot[2]. The applicability of this technique extends to other circumstances where complex recursive functions need to be described formally. In the collective project *Concert*, Laurence Rideau has applied a similar technique to implement the functional representation of the Cminor semantics, where mutual recursions posed a similar problem. In short, let's say $f$ and $g$ be two mutually recursive function, we create a higher order function $F$ that expects a tuple of functions $(f', g')$. The function $f'$ has the same type as $f$ and the function $g'$ has the same type as $g$. We can then iterate $F$ to obtain the results that we were expecting from $f$ and $g$.

Conventional approach to describe the semantics of programming language usually rely on relation, in particular inductive relations. Simulating program execution then relies on proof search tools. In a second work, we describe a functional approach to automate proofs about programming language semantics. Reflection is used to take facts from the context into account. The main contribution of this work is that we developed a systematic approach to describe and manipulate unknown expressions in the symbolic computation of programs for formal proof development. The tool we obtain is faster and more powerful than the conventional proof tools. The use of bottom ($\perp$) as the function argument in the higher order function $F$ and the induction on the number of iterations give us a powerful approximation. In the collective project *Concert*, Laurence Rideau has already used this technique to implement the functional semantics of Cminor. The experiments in the *Concert* effort around an optimizing compiler showed that induction on the number of iterations was the most practical solution, if not the only practical one. The applicability of naming techniques and memory functions should extend to other circumstances where computations need to be done on unknown expressions.

In the collective project *Concert* we contributed a compiler that constructs the RTL control flow graph from the Cminor programs. We keep the modularity of programming while overcoming the constraints of guardedness conditions in the **Coq** system. The implementation of this compiler follows closely the Caml programming style. In the implementation we have seen that guardedness conditions can pose serious problems for recursions even if the principal argument is strictly deceasing. We have shown, with the help of higher order functions, how *the section system* to structure programs in the **Coq** system can be exploited to overcome this constraint. Our technique is practical and easy to follow. The Cminor to RTL compiler is implemented in a way that should help in its proofs. This compiler is still complex and its proofs are going to be bigger and complex. We hope our previous techniques should help in this respect.

Appendix A

# Sémantique en théorie des types pour les langages de programmation

L'une des préoccupations majeures dans le monde académique et dans l'industrie est de réduire la quantité d'erreurs (les bogues) dans les programmes. Les méthodes formelles fournissent un moyen de raisonner à propose des programmes que nous écrivons. Elle ne permette pas seulement de trouve les bogues, les bénéfices majeurs des méthodes formelles sont de décrire les programmes dans des spécifications précises et rigoureuses. De telles spécifications pavent la route vers des preuves formelles rigoureuses à propos des programmes. En général, de telles preuves sont pénibles et peu souvent triviales. Les preuves peuvent être mécanisées avec l'aide d'ordinateurs, car ces derniers sont connus pour leur efficacité dans les tâches fastidieuses.

Vérifier le code source d'un programme n'enlève pas toutes les erreurs. Le code qui est effectivement executé sur la machine est engendré à partir du code source par un compilateur. Les compilateurs peuvent être une source supplémentaire d'erreurs. Avec des optimisations complexes dans les compilateurs, les erreurs ne sont pas rares. Un compilateur incorrect peut introduire des erreurs dans un code cible dont le code source est bien vérifié. Ainsi un compilateur vérifié formellement est d'une importance capitale. Bien sûr, ce sera plus utile si the compilateur formalisé est modérément optimisé et proche d'un langage largement utilisé dans le milieu académique et dans l'industrie.

En pratique, le code source d'un programme est souvent vérifié sur un jeu de données de test et le code assembleur engendré par le compilateur est vérifié manuallement pour voir si le code assembleur est bien relié avec le code source. Ces deux tâches sont susceptible de laisser passer des erreurs. De plus, il n'existe aucune preuve mathématique qu'elle permettent de produire un code sur et sans erreurs. En fait, le typage statique, basé sur un système de type correct, est un pré-requis de base pour l'ingéniérie de systèmes robustes. Les preuves vérifiées par machine portant sur le langage source et le compilateur fournissent la base mathématique.

Dans cette dissertation, nous étudions la sémantique formelle des langages de programmation dans la théorie des types. Nous commençons avec une introduction sur les langages de programmation. Nous donnons une image globale de la syntaxe et de la sémantique des langages de programmation. Ensuite, nous fournissons une brève description du calcul des constructions inductives que nous utiliserons pour formaliser les propriétés des langages de programmations. Nous verifions toutes les preuves dans l'assistant de preuves Coq, qui est basé sur le calcul des constructions inductives.

Dans la section A.1, nous décrivons la sémantique opérationnelle et la sémantique denotationnelle d'un petit langage impératif en en théorie des types avec des définitions inductives et récursives. Nous montrons le problème qui apparait dans la démonstration d'équivalence entre ces sémantiques, lorsque le langage contient de la récursion partielle et imbriquée. Nous fournisson une technique pour contourner ce problème à l'intérieur de la théorie des types.

Les approches conventionnelles pour décrire la sémantique des langages de programmation reposent habituellement su des relations, en particulier des relations inductives. On peut alors simuler l'exécution de programmes à l'aide d'outils de recherche de preuves. Dans la section A.2, nous décrivons une approache fonctionnelle pour automatiser les preuves sur la sémantique des langages de programmation. La technique de "réflexion" est utilisée pour prendre en compte les faits venant du contexte d'exécution. La contribution principale de ce travail est que nous avons développé une approche systématique pour décrire et manipuler des expression inconnues dans l'exécution symbolique de programmes pour le développement de preuves formelles. L'outil que nous obtenons est plus rapide et plus puissant que les outils conventionnels.

Le langage C est l'un des meilleurs langages vis-à-vis des critères que nous avons cités précédement pour un compilateur formalisé. Le travail de cette thèse est une contribution au projet collectif Concert, où nous voulons produire un compilateur formellement correct pour des langages à la C. Dans cette thèse nous décrivons la syntaxe abstraite et la sémantique formelle de Cminor, un sous-ensemble de C.

Plutot que d'engendrer le code objet directement, nous passons par un code intermédiaire, écrit dans un langage de transfert de registres (RTL, register transfer language en anglais) où le flux de contrôle, l'allocation de registre, la propagation des constrantes, l'élimination de code mort, et d'autres optimisations sont calculées. Le langage de transfert de registres est plus du langage du micro-processeur que les programmes RTL représentent le graphe de flot de contrôle de l'exécution du programm. Le code objet est ensuite engendré par linéarisation du graphe RTL.

Dans cette dissertation, nous présentons la formalisation du traducteur de Cminor vers RTL. Dans la section A.3 nous exposons les difficultés qui apparaissent pour cette formalisation et les moyens de contourner ces difficultés. La description formelle du compilateur suit un style qui est proche du style de programmation en Caml, conserve la modularité des programmes ce qui facilite les preuves de correction, bien que nous ne considérions pas les preuves de correction dans cette thèse.

## A.1 Sémantique fonctionnelle en théorie des types

Dans le context des langages de programmation, les types algébriques sont adaptés pour la syntaxe et les propositions inductives sont adaptées pour la sémantique naturalle. La sémantique dénotationnelle fonctionne plutôt avec des définitions de fonctions. Pour la sémantique dénotationnelle, nous avons besoin d'utiliser des fonctions récursives sans contraintes. Pour nous la question devient la suivante: comment pouvons représenter les constructions de programmes qui font appel à de la récursion mutuelle et de la récursion imbriquée en théorie des types, alors que cette théorie impose les contraintes de la récursion structurelle et de fonctions totales alors que le langage de programmation ne possède pas toutes ces propriétés? Comment pouvons-nous bénéficier des descriptions fonctionnelles du langage de programmation? Nous abordons ces question dans les sections suivantes.

La sémantique opérationnelle consiste dans la description des pas de calcul d'un probram en fournissant des règles formelles pour dériver des jugements de la forme $\langle p, a\rangle \rightsquigarrow r$, qui se lit ■l'application du programme $p$ à l'entrée $a$ termine et retourne le résultat $r$■.

La sémantique dénotationnelle consiste dans la description d'une signification mathématique aux données et aux programmes, plus précisément en interprétant les données (entrées et sorties) comme des éléments de certains domaines et les programmes comme des fonctions sur ces domaines; ainsi le fait que le programme appliqué à l'entrée a retourne le résultat r sera exprimé par l'égalité $[\![p]\!](a) = r$, où $[\![-]\!]$ est la fonction d'interprétation.

Dans ce travail, nous démontrons un théorème de correction et de complétude qui exprime que la sémantique opérationnelle et la sémantique dénotationnelle sont en accord.

La mise en place de la sémantique opérationnelle est immédiate: le système de dérivation est décrit comme une relation inductive dont les constructeurs sont une paraphrase directe des règles de dérivation.

La mise en place de la sémantique dénotationnelle est bien plus délicate. Traditionnellement, les programmes sont interprétés comme des fonctions partielles, puisqu'ils peuvent diverger sur certaines entrées. Mais toutes les fonctions de la théorie des types sont totatles. Le problème de représenter les fonctions partielles dans un contexte total a été le sujet de travaux récents par de plusieurs auteurs [15, 13, 34, 10, 37]. Une technique standard pour résoudre ce problème est de réduire le domaine à exactement les entrées dont pour lesquelles le programme termine et d'interpreter le programme comme une fonction totale sur le domaine restraint.

Quand nous considérons une fonction récursive imbriquée, une formalisation directe requiert de définir le domaine et la fonction simultanément. Ceci n'est pas possible dans toutes les théories de types, mais seulement dans celles qui sont étendues avec de l'induction-récursion simultanée à la Dybjer [14]. C'est l'aproche adoptée dans [10].

Une approche alternative, adoptée par Balaa et Bertot dans [1], est de voir la fonction partielle comme le point fixe d'un opérateur $F$ qui envoie les fonctions totales vers les fonctions totales. Ce pointfixe peut être approché par un nombre fini d'itérations de $F$ à partir d'une fonction de base arbitraire. Le domaine peut être décrit comme l'ensemble des éléments pour lesquels la fonction F stabilise après un nombre fini d'itération, indépendamment de la fonction de base.

L'inconvénient de l'approche de [10] est qu'elle n'est pas viable dans les théories de types usuelles (c'est-à-dire privées du schéma de Dybjer). L'inconvénient de l'approche de [1] est que le domaine de définition obtenu est celui d'un point de fixe qui peut ne pas être le point fixe minimal. Cette approche peut être correcte si l'on modèle un langage paresseux, mais c'est certainement incorrect si l'on étudie un langage strict (en appel-par-valeur), pour lequel le point fixe minimal est requis. L'interprétation des langages impératifs est essentiellement stricte and le domaine obtenu est trop large: le program est défini sur des valeurs pour lesquelles le programme ne termine pas.

Nous combinons les deux approches de [10] et [1] pour définir un domaine de définition de façon similaire à [10] mais en démélant la dépendance mutuelle entre le domaine et la fonction, à l'aide d'intérations de la fonctionnelle $F$ avec un nombre d'itération variable à la place de la fonction qui n'est pas encore définie.

Nous affirmons deux résultats principaux. Premièrement, nous développons de la sémantique dénotationnelle en théorie des types. Deuxièmement, nous modélisons la méthode d'accessibilité dans un système plus faible, sans induction-récursion simultanée. Voici un aperçu rapide de ce travail.

Winskel [38] présente un petit langage de programmation IMP avec des boucles *while*. IMP est un langage impératif simple avec des entiers, des valeurs de vérité $\mathsf{true}$ et $\mathsf{false}$, des cellules de mémoire pour stocker des entiers, des expressions arithmétiques, des expressions booléennes, et des instructions. Les règles de formation sont les suivantes:

expressions arithmétiques: $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1;$

expressions booléennes: $b ::= \mathsf{true} \mid \mathsf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1;$

instructions: $c ::= \mathsf{skip} \mid X \leftarrow a \mid c_0; c_1 \mid \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \mid \mathsf{while}\ b\ \mathsf{do}\ c$

Ici, $n$ dénote un entier, $X$ dénote une cellule de mémoire, $a$ dénote une expression arithmétique $b$ dénote une expression booléenne, et $c$ dénote une instruction.

Nous formalisons ceci avec trois types inductifs $\mathsf{AExp}$, $\mathsf{BExp}$, et $\mathsf{Command}$.

Nous voyons les instructions comme des transformeurs d'état, où un état est une correspondance des cellules de mémoire avec des entiers. Cette correspondance est partielle en générale, en fait elle n'est définie que pour un nombre fini de cellules mémoire. Nous pouvons représenter cette correspondance à l'aide d'une liste de liaison entre cellules mémoire et valeurs. Si la même cellule mémoire est liée deux foix dans le même état, la liaison la plus récente, c'est-à-dire la plus à gauche, est la liaison valide.

$$\begin{array}{l}\mathsf{State}\colon \mathbf{Set}\\ []\colon \mathsf{State}\\ [\cdot \mapsto \cdot, \cdot]\colon \mathbb{N} \to \mathbb{N} \to \mathsf{State} \to \mathsf{State}\end{array}$$

L'état $[v \mapsto n, s]$ est l'état $s$ où le contenu de la cellule $v$ est remplacé par $n$.

La sémantique opérationnelle des instruction est donnée de la façon suivante:

$$\frac{}{\langle \mathsf{skip}, \sigma\rangle_{\mathsf{C}} \leadsto \sigma}$$

$$\frac{\langle a, \sigma\rangle_{\mathsf{A}} \leadsto n \qquad \sigma_{[X\mapsto n]} \leadsto \sigma'}{\langle X \leftarrow a, \sigma\rangle_{\mathsf{C}} \leadsto \sigma'}$$

$$\frac{\langle c_1, \sigma\rangle_{\mathsf{C}} \leadsto \sigma_1 \quad \langle c_2, \sigma_1\rangle_{\mathsf{C}} \leadsto \sigma_2}{\langle c_1; c_2, \sigma\rangle_{\mathsf{C}} \leadsto \sigma_2}$$

$$\frac{\langle b, \sigma\rangle_{\mathsf{B}} \leadsto \mathsf{true} \quad \langle c_1, \sigma\rangle_{\mathsf{C}} \leadsto \sigma_1}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle_{\mathsf{C}} \leadsto \sigma_1} \qquad \frac{\langle b, \sigma\rangle_{\mathsf{B}} \leadsto \mathsf{false} \quad \langle c_2, \sigma\rangle_{\mathsf{C}} \leadsto \sigma_2}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle_{\mathsf{C}} \leadsto \sigma_2}$$

$$\frac{\langle b, \sigma\rangle_{\mathsf{B}} \leadsto \mathsf{true} \quad \langle c, \sigma\rangle_{\mathsf{C}} \leadsto \sigma' \quad \langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma'\rangle_{\mathsf{C}} \leadsto \sigma''}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle_{\mathsf{C}} \leadsto \sigma''} \qquad \frac{\langle b, \sigma\rangle_{\mathsf{B}} \leadsto \mathsf{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle_{\mathsf{C}} \leadsto \sigma}$$

Les règles ci-dessus peuvent être formalisées en **Coq** de façon directe par une relation inductive.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\langle\cdot,\cdot\rangle_{\mathsf{C}} \rightsquigarrow \cdot \colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \to \mathbf{Prop}} \\
\multicolumn{2}{l}{\mathsf{eval\_skip}\colon (\sigma\colon \mathsf{State})(\langle \mathsf{skip}, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma)} \\
\mathsf{eval\_assign}\colon & (\sigma,\sigma'\colon \mathsf{State}; v, n\colon \mathbb{N}; a\colon \mathsf{AExp}) \\
& (\langle a,\sigma\rangle_{\mathsf{A}} \rightsquigarrow n) \to (\sigma_{[v\mapsto n]} \rightsquigarrow \sigma') \to (\langle v \leftarrow a, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma') \\
\mathsf{eval\_scolon}\colon & (\sigma,\sigma_1,\sigma_2\colon \mathsf{State}; c_1, c_2\colon \mathsf{Command}) \\
& (\langle c_1,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \to (\langle c_2,\sigma_1\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \to (\langle c_1; c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \\
\mathsf{eval\_if\_true}\colon & (b\colon \mathsf{BExp}; \sigma,\sigma_1\colon \mathsf{State}; c_1, c_2\colon \mathsf{Command}) \\
& (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \to (\langle c_1,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \to \\
& (\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \\
\mathsf{eval\_if\_false}\colon & (b\colon \mathsf{BExp}; \sigma,\sigma_2\colon \mathsf{State}; c_1, c_2\colon \mathsf{Command}) \\
& (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \to (\langle c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \to \\
& (\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \\
\mathsf{eval\_while\_true}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma,\sigma',\sigma''\colon \mathsf{State}) \\
& (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \to (\langle c,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma') \to \\
& (\langle \mathsf{while}\ b\ \mathsf{do}\ c,\sigma'\rangle_{\mathsf{C}} \rightsquigarrow \sigma'') \to (\langle \mathsf{while}\ b\ \mathsf{do}\ c,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma'') \\
\mathsf{eval\_while\_false}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}) \\
& (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \to (\langle \mathsf{while}\ b\ \mathsf{do}\ c,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma)
\end{array}
$$

La sémantique dénotationnelle consiste dans l'interprétation du program comme une fonction plutôt qu'une relation. Malheureusement, la fonction d'interprétation $[\![\cdot]\!]$ ne peut pas être donnée par récursion structurelle. Nous aurions alors:

$$
\begin{array}{l}
[\![\cdot]\!]\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \\
[\![\mathsf{skip}]\!]_\sigma := \sigma \\
[\![X \leftarrow a]\!]_\sigma := \sigma[[\![a]\!]_\sigma / X] \\
[\![c_1; c_2]\!]_\sigma := [\![c_1]\!]_{[\![c_2]\!]_\sigma} \\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma := \left\{ \begin{array}{ll} [\![c_1]\!]_\sigma & \text{if } [\![b]\!]_\sigma = \mathsf{true} \\ [\![c_2]\!]_\sigma & \text{if } [\![b]\!]_\sigma = \mathsf{false} \end{array} \right. \\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma := \left\{ \begin{array}{ll} [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_{[\![c]\!]_\sigma} & \text{if } [\![b]\!]_\sigma = \mathsf{true} \\ \sigma & \text{if } [\![b]\!]_\sigma = \mathsf{false} \end{array} \right.
\end{array}
$$

mais dans la clause pour les boucles *while*, la fonction d'interpretation est appelée sur le même argument si l'expression booléenne s'évalue à true. Pour cette raison l'argument de l'appel récursif n'est pas structurellement plus petit que l'argument initial. L'exécution des boucles *while* ne respecte pas le schéma de la récursion structurelle et la terminaison ne peut pas être assurée, pour une bonne raison, puisque le langage est complet au sens de Turing. Nous décrivons un moyen de contourner ce problème.

Une représentation fonctionelle des calculs peut être fournie d'une manière qui respecte le typage et la terminaison si nous n'essayons pas de décrire l'exécution de la fonction mais la fonction de *second-ordre dont la fonction d'exécution est le point fixe.* Cette fonction peut être définie en théorie des types par une analyse de la structure de la commande.

$$\begin{array}{l}
\mathsf{F}\colon (\mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \to \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \\
(\mathsf{F}\ f\ \mathsf{skip}\ \sigma) := \sigma \\
(\mathsf{F}\ f\ (X \leftarrow a)\ \sigma) := \sigma[[\![a]\!]_\sigma/X] \\
(\mathsf{F}\ f\ (c_1; c_2)\ \sigma) := (f\ c_2\ (f\ c_1\ \sigma)) \\
(\mathsf{F}\ f\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma) := \begin{cases} (f\ c_1\ \sigma) & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true} \\ (f\ c_2\ \sigma) & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false} \end{cases} \\
(\mathsf{F}\ f\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma) := \begin{cases} (f\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ (f\ c\ \sigma)) & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true} \\ \sigma & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false} \end{cases}
\end{array}$$

Intuitivement, écrire la fonction F est la même chose que d'écrire la fonction récursive d'exécution, sauf que la fonction en cours de définition est simplement replacé par une variable liée (is *f*). Autrement dit, nous remplaçons les appels récursifs par des appels à la fonction donnée par la variable liée *f*.

La fonction F décrit les calculs qui sont effectués à chaque itération de la fonction d'exécution et cette dernière effectue les même calculs que F lorsque celle-ci est répétée *autant de fois que nécessaire.* Nous pouvons exprimer ceci à l'aide du théorème suivant:

**Theorem A.1** (eval_com_ind_to_rec).

$$\begin{array}{l}
\forall c\colon \mathsf{Command}.\forall \sigma_1, \sigma_2\colon \mathsf{State}. \\
\langle c, \sigma_1 \rangle \rightsquigarrow \sigma_2 \Rightarrow \exists k\colon \mathbb{N}.\forall g\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{F}^k\ g\ c\ \sigma_1) = \sigma_2
\end{array}$$

*où nous avons utilisé la notation suivante*

$$\mathsf{F}^k = (\mathsf{iter}\ (\mathsf{Command} \to \mathsf{State} \to \mathsf{State})\ \mathsf{F}\ k) = \lambda g.\underbrace{(\mathsf{F}\ (\mathsf{F}\ \cdots\ (\mathsf{F}}_{k\ times}\ g)\ \cdots\ ))$$

*définissable par récursion sur k,*

$$\begin{array}{l}
\mathsf{iter}\colon (A\colon \mathbf{Set})(A \to A) \to \mathbb{N} \to A \to A \\
(\mathsf{iter}\ A\ f\ 0\ a) := a \\
(\mathsf{iter}\ A\ f\ (\mathsf{S}\ k)\ a) := (f\ (\mathsf{iter}\ A\ f\ k\ a)).
\end{array}$$

Le théoreme **A.1** donne l'une des directions de la correspondance entre la sémantique opérationnelle et la sémantique dénotationnelle à l'aide de la méthode des itérations. Pour compléter la tâche de formaliser la sémantique dénotationnelle, il nous faut définir une fonction en théorie des types qui interprète chaque commande. Comme nous l'avons déjà remarqué cette fonction ne peut pas être totale; nous devons donc réduire le domaine aux instructions qui terminent. Ceci est fait à l'aide d'un prédicat $D$ sur les instructions et les états et nous définissons ensuite la fonction d'interprétation $[\![\cdot]\!]$ sur le domaine restraint par ce prédicat.

Le théorème **A.1** suggère la définition suivante:

$$\begin{aligned}
&D\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Prop}\\
&(D\ c\ \sigma) := \ \exists k\colon \mathbb{N}.\forall g_1, g_2\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.\\
&\qquad\qquad\quad (\mathsf{F}^k\ g_1\ c\ \sigma) = (\mathsf{F}^k\ g_2\ c\ \sigma).
\end{aligned}$$

Malheureusement, cette définition est trop faible. En général une telle approche ne peut pas être utilisée pour caractériser la terminaison de récursion imbriquée. Il est difficile de s'en convaincre dans le cas du langage IMP, mais cela apparaitrait de façon plus directe si on ajoutait au langage une instruction *exception* avec la sémantique suivante:

$$\langle \mathsf{exception}, \sigma\rangle \rightsquigarrow [].$$

Intuitivement, le programmeur pourrait utiliser cette instruction pour exprimer qu'une situation exceptionnelle a été détectée, mais toute information sur le calcul en cours serait détruite.

Avec cette nouvelle instruction, on pourrait avoir des instructions et des entrées pour lesquelles le prédicat $D$ est satisfait mais pour lesquels l'exécution ne terminerait pas.

$$c := \mathsf{while\ true\ do\ skip}; \mathsf{exception}.$$

Il est facile de voire que pour n'importe quel état $\sigma$ le calcul de $c$ sur $\sigma$ ne termine pas. Du point de vue de la sémantique opérationnelle le jugement $\langle c, \sigma\rangle \rightsquigarrow \sigma'$ n'est dérivable pour aucun $\sigma'$. Néanmoins, $(D\ c\ \sigma)$ est prouvable, parce que $(\mathsf{F}^k\ g\ c\ \sigma) = []$ pour tout $k > 1$.

Nous allons maintenant concevoir une caractérisation plus forte du domaine des fonctions partielles qui est la correct pour interpréter la sémantique opérationnelle.

Le domaine de définition d'une fonction peut parfois être caractérisé indépendamment de la fonction par un prédicat inductif appelé *accessibilité* [29, 15, 13, 9]. Ce prédicat exprime simplement qu'un élément $a$ appartient provablement au domaine si l'application de $f$ sur $a$ provoque des appels récursifs sur des éléments qui appartiennent eux même prouvablement au domaine. Cette définition ne fonctionne pas toujours. Dans le cas de fonctions récursives imbriquées nous ne pouvons éliminer les références à la fonction $f$ dans les clauses de la définition inductive de l'accessibilité. Dans notre cas, nous avons deux instances de clauses récursives imbriquées, pour la composition séquentielle et pour les boucles *while*.

Une solution alternative, présentée dans [10], exploite une extension de la théorie des types avec l'induction-récursion simultanée [14]. Cette méthode mène à la définition suivante de l'attribut d'accessibilité et de la fonction d'interprétation pour le langage de programmation impératif IMP :

$$
\begin{array}{l}
\mathsf{comAcc}\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Prop}\\
[\![]\!]\colon (c\colon \mathsf{Command}; \sigma\colon \mathsf{State})(\mathsf{comAcc}\ c\ \sigma) \to \mathsf{State}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{accSkip}\colon & (\sigma\colon \mathsf{State})(\mathsf{comAcc}\ \mathsf{skip}\ \sigma)\\
\mathsf{accAssign}\colon & (v\colon \mathbb{N}; a\colon \mathsf{AExp}; \sigma\colon \mathsf{State})(\mathsf{comAcc}\ (v \leftarrow a)\ \sigma)\\
\mathsf{accScolon}\colon & (c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; h_1\colon (\mathsf{comAcc}\ c_1\ \sigma))(\mathsf{comAcc}\ c_2\ [\![c_1]\!]_\sigma^{h_1})\\
& \to (\mathsf{comAcc}\ (c_1; c_2)\ \sigma)\\
\mathsf{accIf\_true}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!]_\sigma = \mathsf{true} \to (\mathsf{comAcc}\ c_1\ \sigma)\\
& \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma)\\
\mathsf{accIf\_false}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!]_\sigma = \mathsf{false} \to (\mathsf{comAcc}\ c_2\ \sigma)\\
& \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma)\\
\mathsf{accWhile\_true}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!] = \mathsf{true}\\
& \to (h\colon (\mathsf{comAcc}\ c\ \sigma))(\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ [\![c]\!]_\sigma^h)\\
& \to (\mathsf{comAcc}(\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma)\\
\mathsf{accWhile\_false}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!] = \mathsf{false}\\
& \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma)
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{skip}]\!]_\sigma^{(\mathsf{accSkip}\ \sigma)} := \sigma\\
[\![(v := a)]\!]_\sigma^{(\mathsf{accAssign}\ v\ a\ \sigma)} := \sigma[a/v]\\
[\![(c_1; c_2)]\!]_\sigma^{(\mathsf{accScolon}\ c_1\ c_2\ \sigma\ h_1\ h_2)} := [\![c_2]\!]_{[\![c_1]\!]_\sigma^{h_1}}^{h_2}\\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma^{(\mathsf{accIf\_true}\ b\ c_1\ c_2\ \sigma\ p\ h_1)} := [\![c_1]\!]_\sigma^{h_1}\\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma^{(\mathsf{accIf\_false}\ b\ c_1\ c_2\ \sigma\ q\ h_2)} := [\![c_2]\!]_\sigma^{h_2}\\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma^{(\mathsf{accWhile\_true}\ b\ c\ \sigma\ p\ h\ h')} := [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_{[\![c]\!]_\sigma^h}^{h'}\\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma^{(\mathsf{accWhile\_false}\ b\ c\ \sigma\ q)} := \sigma
\end{array}
$$

Cette définition est admissible dans les systèmes qui mettent en application le schéma de Dybjer's pour l'induction-récursion simultanée. Mais sur les systèmes qui ne fournissent pas un tel schéma, par exemple **Coq**, cette définition est inadmissible.

Nous devons déméler la définition du prédicat d'accessibilité de la définition de la fonction d'évaluation. Comme nous l'avons déjà vu, la fonction d'évaluation peut être vue comme la limite de l'itération de la fonctionnelle F sur une fonction de base arbitraire $f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}$. A chaque fois que l'exécution d'une instruction $c$ est définie sur un état $\sigma$, nous obtenons que $[\![c]\!]_\sigma$ est égal à $(F_f^k\ c\ \sigma)$ pour un nombre suffisamment large d'itération $k$. Pour cette raison, nous considérons les fonction telles $F_f^k$ comme des approximations de la fonction d'interprétation en cours de définition. Nous pouvons formuler le prédicat d'accessibilité en utilisant de telles approximations à la place des occurrences explicites de la fonction d'évaluation. Puisque l'approximation par itération a deux paramètres supplémentaires, le nombre d'itérations $k$ et la fonction de base $f$ nous devons ajouter ces paramètres au prédicat que nous appelons comAcc. Nous obtenons une définition inductive de la forme suivante:

$$
\begin{array}{ll}
\mathsf{comAcc}\colon \mathsf{Command} \to \mathsf{State} \to \mathbb{N} \to (\mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \to \mathbf{Prop} & \\
\mathsf{accSkip}\colon (\sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\mathsf{comAcc}\ \mathsf{skip}\ \sigma\ k+1\ f) & \\
\mathsf{accAssign}\colon & (v\colon \mathbb{N}; a\colon \mathsf{AExp}; \sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \\
& (\mathsf{comAcc}\ (v \leftarrow a)\ \sigma\ k+1\ f) \\
\mathsf{accScolon}\colon & (c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon (\mathsf{Command} \to \mathsf{State} \to \mathsf{State})) \\
& (\mathsf{comAcc}\ c_1\ \sigma\ k\ f) \to (\mathsf{comAcc}\ c_2\ (F_f^k\ c_1\ \sigma)\ k\ f) \\
& \to (\mathsf{comAcc}\ (c_1; c_2)\ \sigma\ k+1\ f) \\
\mathsf{accIf\_true}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{true}) \\
& \to (\mathsf{comAcc}\ c_1\ \sigma\ k\ f) \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma\ k+1\ f) \\
\mathsf{accIf\_false}\colon & (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{false}) \\
& \to (\mathsf{comAcc}\ c_2\ \sigma\ k\ f) \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma\ k+1\ f) \\
\mathsf{accWhile\_true}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{true}) \\
& \to (\mathsf{comAcc}\ c\ \sigma\ k\ f) \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ (F_f^k\ c\ \sigma)) \\
& \to (\mathsf{comAcc}(\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma\ k+1\ f) \\
\mathsf{accWhile\_false}\colon & (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
& k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{false}) \\
& \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma\ k+1\ f).
\end{array}
$$

Ce prédicat d'accessibilité caractérise les points du domaine du programme de façon paramétrique par rapport aux arguments $k$ et $f$. Pour obtenir une définition du domaine indépendante nous devons quantifier par rapport à ces paramètres. La quantification est existentielle par rapport à $k$, parce que si une instruction et un état sont accessibles en $k$ étapes, alors ils sont accessibles en un nombre supérieur d'étape. La quantification est universelle par rapport à $f$ parce que nous ne voulons pas que le résultat dépende du choix de la fonction de base.

$$
\begin{array}{l}
\mathsf{comDom}\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Set} \\
(\mathsf{comDom}\ c\ \sigma) = \Sigma k\colon \mathbb{N}.\forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{comAcc}\ c\ \sigma\ k\ f)
\end{array}
$$

La fonction d'évaluation des instructions est donnée de la façon suivante:

$$
\begin{array}{ll}
[\![]\!]\colon & (c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \\
& (\mathsf{comDom}\ c\ \sigma) \to \mathsf{State} \\
[\![c]\!]_{\sigma,f}^{\langle k,h\rangle} = (F_f^k\ c\ \sigma) &
\end{array}
$$

Nous pouvons maintenat démontrer la correspondance exacte entre la sémantique opérationnelle et la sémantique dénotationnelle donnée par l'opérateur d'interprétation $[\![\cdot]\!]$.

**Theorem A.2.**

$$\forall c\colon \mathsf{Command}.\forall \sigma, \sigma'\colon \mathsf{State}.$$
$$\langle c, \sigma\rangle \rightsquigarrow \sigma' \Leftrightarrow \exists H\colon (\mathsf{comDom}\ c\ \sigma).\forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.[\![c]\!]^{H}_{\sigma,f} = \sigma'.$$

La combinaison de la technique d'itération et de prédicats d'accessibilité a un potentiel important qui dépasse le cadre cette application à la sémantique dénotationnelle. Non seulement elle fournit un chemin vers l'implémentation et le raisonnement sur les fonction partielle et les définitions récursive sans utiliser d'induction-récursion simultanée ; elle donne une analyse de grain plus fin sur la convergence des opérateurs récursifs. Comme nous l'avons souligné, elle ne fournit pas seulement un point fixe, mais le point fixe minimal.

## A.2 Preuve par réflexion en sémantique

Dans la section précédente nous avons vu une approche pour décrire la sémantique d'un langage de programmation comme une fonction qui envoie les programmes et leurs entrées vers leurs sorties. Cette fonction peut être utilisée pour calculer le résultat de l'exécution d'un programme donné sur une entrée donnée, mais elle n'est pas adaptée pour raisonner sur des programmes en utilisant des informations venant du context. Il y a un moyen de rendre cette approche fonctionnelle plus puissante, de manière à ce qu'elle utilise aussi les informations du contexte.

Nous utilisons l'expression "sémantique fonctionnelle" plutôt que "sémantique dénotationnelle" parce que notre travail ne vien pas avec le bagage usuel de théorie des domaines ou des ordres partiels complets.

Pour simuler l'execution d'un program à l'aide de la sémantique fonctionnelle, nous avons seulement besoin d'appliquer la fonction à un programme et à ses entrées et à provoquer la réduction du terme jusqu'au résultat. En ce sens, la sémantique fonctionnelle ouvre la possibilité de preuve par réflexion [8, 36, 21], car elle permet de représenter à la fois la sémantique et une procédure de preuve. Mais l'outil de preuve que nous obtenons reste très faible parce qu'il ne permet de réduction jusqu'à une valeur adéquate que pour les programmes clos pour lesquels aucune information contextuelle n'est nécessaire. Notre prochain objectif est d'obtenir une procédure de preuve plus puissante que les méthodes de preuve conventionnelles.

Le résultat le plus important est une technique qui aide à raisonner sur des metavariables, en d'autres termes, des expressions et des instructions symboliquement représentées. Cette technique est systématique et générale.

Les méthodes les plus communes pour automatiser la recherche de preuve sont basées sur l'unification et la résolution. Une preuve peut être vue comme un but à résoudre étant donné un contexte d'hypothèses. Une procédure basée sur l'unification et la résolution recherche dans le contexte local et cherche à faire correspondre le but avec la concusion de l'une des hypothèses. En cas de succès, l'opération retourne un sous-but pour chacune des prémisses de l'hypothèse choisie.

Les inconvénients de cette méthode sont d'abord que la stratégie générale de recherche n'est pas concentrée sur le problème et perd beaucoup de temps à explorer un espace de recherche étendu et ensuite que cette méthode ne dispose pas de moyens de calcul. On peut se retrouver dans une situation où, même si le contexte contient assez d'information pour avancer dans le calcul, le calcul n'a pas lieu. Nous montrons un tel exemple.

Les assistants de preuve basés sur la théorie des types comme **Coq** [33] fournissent des fonctions et plusieurs notions de réduction sont fournies pour calculer avec ces fonctions. Les fonctions calculent sur des données. Dans les preuves formelles, les connaissances reliées aux calculs sont fournies comme des assertions, qui sont en fait des relations entre des données, mais pas elles-mêmes des données. Pour permettre le calcul de preuve par des fonctions, nous utilisons la réflection. Nous récoltons les données fournies dans les différentes connaissances et les plaçons dans plusieurs tables. Plutôt que de chercher des hypothèses dans un contexte local, comme nous le ferions dans une approche basée sur l'unification et la résolution, nous consultons les tables à l'aide de fonctions. Cette approache fonctionelle est plus concentrée sur l'objectif car les fonctions effectuent un meilleur choix des tables reliées aux questions posées. L'évaluation des fonctions est effectuée par la réduction de termes et le raisonnement sur es prexpressions contenant des inconnues demande un soin particulier. Nous décrivons une solution pour contourner ce problème.

Nous Defendons deux résultats principaux. Premièrement, l'utilisation de la réflexion et d'une approche fonctionnelle pour automatiser la recherche de preuve fournit un moyen plus facile et meilleur que la technique courante d'unification et de résolution. Deuxièmement, et de façon plus importante, nous présentons une nouvelle méthode systématique pour gérer les expressions inconnues, différentes des techniques de résolution et permettant d'intégrer les calculs (et les calculs concernant des méta-variables) dans la théorie des types. Ci dessous nous donnons une description rapide de ce travail.

Dans les assistants de preuve comme **Coq** ou Isabelle/HOL, l'exécution d'instruction peut être vue comme la preuve de lemmes, où les faits nécessaires pour permettre l'exécution sont fournis comme des hypothèses. Par exemple, nous voudrions montrer que que l'exécution d'une séquence de deux instructions $i_1$ et $i_2$ à partir d'un état $\sigma$ produira un état final $\sigma''$, en sachant que l'exécution de $i_1$ à partir de l'état $\sigma$ retourne un état $\sigma'$ et que l'exécution de $i_2$ à partir de l'état $\sigma'$ retourne $\sigma''$.

**Lemma A.1.** $\forall \sigma, \sigma', \sigma''\colon \mathsf{State}. \forall i_1, i_2\colon \mathsf{Inst}.$

$$\underbrace{(\langle i_1, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma') \rightarrow (\langle i_2, \sigma'\rangle_{\mathsf{I}} \rightsquigarrow \sigma'')}_{\text{facts}} \rightarrow \underbrace{(\langle i_1; i_2, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma'')}_{\text{goal}}.$$

Pour automatiser la preuve de cet énoncé, qui est similaire à la sémantique opérationnelle de la séquence, des difficultés apparaissent pour construire les résultats intermédiaires, l'état $\sigma'$ dans notre cas, qui n'apparaissent pas déjà dans la conclusion du but. Habituellement, ce type de preuve est effectué par résolution et unification, comme dans les interprètes Prolog, et les valeurs manquantes sont remplacées par des variables existentielles pour instantiation ultérieure à l'aide de l'unification. Pour notre exemple dans Coq, une procedure basée sur l'unification et la résolution, appelée EAuto, trouve la correspondance avec eval_scolon et remplace $\sigma'$ par une variable existentielle ?1. Elle trouve ensuite une autre correspondance dans le contexte des deux sous-buts et est amenée à instantier ?1 avec $\sigma'$, ce qui résout le but. En revanche, une procedure basée sur l'unification et la résolution échoue quand un calcul est nécessaire. Par exemple, on peut consider le lemme suivant:

**Lemma A.2.** $\forall \sigma \colon \mathsf{State}. \forall v \colon \mathbb{Z}.$

$$\underbrace{(\mathsf{lookup}\ \sigma\ v\ 1)}_{\text{facts}} \rightarrow \underbrace{\langle \mathsf{while}\ 3\ \leq\ v\ \mathsf{do}\ \mathit{skip}, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma}_{\text{goal}}.$$

Etand donné que la variable $v$ est associée à la valeur 1 dans l'état $\sigma$, nous devons prouver que l'exécution de la boucle "while" ne changera pas l'état. Nous disposons d'assez d'information pour établir que l'expression booléenne $3 \leq 1$ sera évaluée en $\mathsf{false}$, mais une telle preuve n'existe pas dans le contexte et il est nécessaire de la calculer. Une procédure basée sur l'unification et la résolution n'a pas les moyens de calculs pour ceci. Un moyen de résoudre ce problème est d'utiliser des fonctions au lieu de relations.| Les fonctions peuvent aussi obtenir des résultats intermédiaires, ce qui permet de résoudre aussi le lemmeA. 1.

Comme nous l'avons déjà mentionné, les résultats déjà calculés sont disponibles sous forme d'assertions, mais les fonctions calculent sur des données pas sur des assertions dans le contexte. Les procédures basées sur l'unification et la résolution utilisent le contexte dans leur recherche de preuves. Pour utiliser des fonctions, nous avons besoin de construire des données qui représentent fidèlement le contexte. Nous montrons comment satisfaire ce besoin.

Pour gardre les informations sur les expressions arithmétiques, nous créons une table $\mathsf{T^r}_{\mathsf{AExp}}$ (lire : liste des résultats donnés pour les expressions arithmétiques). Cette table est une liste de triplets où chaque triplet regroupe un état une expression arithmétique et la valeur de cette expression dans cet état.

De façon similaire nous créons des tables différentes pour les différents types d'information provenant du contexte. Nous avons des résultats pour les recherches dans l'état mémoire, les mise à jour de l'état mémoire, les évaluations d'expressions booléennes et les exécutions d'instructions. Nous les écrivons $\mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}}$, $\mathsf{T}^{\mathsf{r}}_{\mathsf{update}}$, $\mathsf{T}^{\mathsf{r}}_{\mathsf{BExp}}$ et $\mathsf{T}^{\mathsf{r}}_{\mathsf{Inst}}$, respectivement. Nous nous assurons que ces tables sont cohérentes avec le contexte à l'aide d'une collection de fonctions de vérifications, $[\![\cdot]\!]^{\mathsf{c}}_{\mathsf{lookup}}$, $[\![\cdot]\!]^{\mathsf{c}}_{\mathsf{update}}$, $[\![\cdot]\!]^{\mathsf{c}}_{\mathsf{BExp}}$ and $[\![\cdot]\!]^{\mathsf{c}}_{\mathsf{Inst}}$, respectivement.

Pour évaluer une expression, nous vérifions d'abord dans la table correspondante si le résultat est déja connu. Dans le cas négatif, nous suivons simplement la sémantique usuelle.

### A.2.1 Preuves par réflexion

Dans les preuves formelles habituelles, les hypothèses sur les calculs sont représentées par des assertions indiquant qu'une certaine relation est satisfaite entre certaines données. Les mécanismes de recherche de preuve fouillent le contexte pour retrouver ces assertions et vérifier si le but peut être résolu directement comme conséquence directe.

Dans les systèmes de preuve basés sur la théorie des types comme **Coq**, les fonctions sont aussi fournies et la réduction peut être utilisée pour calculer avec ces fonctions [27]. L'idée de la réflection est d'utiliser ces fonctions pour effectuer la recherche de preuve. Mais les fonctions calcuent normalement sur des données et les hypothèses du contexte ne sont pas des données au niveau de ces fonctions, mais seulement des données au niveau du système de preuve.

Pour prouver qu'une propriété $P$ donnée est satisfaite pour un argument $t$ en utilisant la réflexion.

Pour prouver qu'une propriété $P$ donnée est satisfaite par un terme $t$ à l'aide de la réflection on procède comme suit: on utilise un assistant de preuve où l'on peut à la fois décrire et prouver des programmes, on écrit un programme qui prend $t$ comme entrée et qui retourne la valeur $\mathsf{true}$ seulement lorsque $P(t)$ est satisfait. On prouve donc $Q(t) = \text{true} \Rightarrow P(t)$ et ensuite on utilise le programme $Q$ pour prouver des instances de $P$.

Dans notre cas, nous voulons montrer que $\langle\sigma, i\rangle_{\mathsf{I}} \rightsquigarrow \sigma'$ est satisfait étant données une collection d'hypothèses $\Gamma$. Il nous faut donc écrire une fonction $Q$ qui prend en arguments des données représentant $\Gamma$, $\sigma$ et $i$ et qui retourne $\sigma'$ seulement lorsque $\langle\sigma, i\rangle_{\mathsf{I}} \rightsquigarrow \sigma'$ est satisfait dans l'environnement $\Gamma$. Nous devons donc prouver $Q(\sigma,\ i,\ \textit{arguments des données représentant}\ \Gamma) = \sigma' \Rightarrow \Gamma \rightarrow \langle\sigma, i\rangle_{\mathsf{I}} \rightsquigarrow \sigma'$. Pour les besoins de la reflexion, nous avons déjà construit des données pour représenter les hypothèses au niveau où les fonctions peuvent calculer.

Nous avons déjà décrit deux problèmes qui doivent être résolus par les moteurs de recherche de preuve pour construire des preuves en sémantique. Le premier problème est de trouver des valeurs intermédiaires au cours des calculs. Celui-ci est résolu de façon naturelle par le calcul de fonctions. Le second problème est d'intercaler des calculs arithmétiques avec des opérations de recherche de preuve. Celui-ci peut aussi être résolution si les fonction d'évaluation de programmes peuvent appeler les fonctions arithmétiques adaptées. Pour ces problèmes les outils de preuve basés sur l'évaluation de fonctions sont meilleurs que les outils de preuves basés sur l'unification et la résolution. L'évaluation de fonctions est également mieux guidée que la recherche de preuve et donc plus efficace. De plus, ce procédé peut devenir extrêmement puissant lorsque des mécanismes de réduction rapide sont implantés dans les systèmes de preuve [19].

Une nouvelle difficulté apparait lorsque nous raisonnons sur des expressions inconnues. Considérons l'énoncé suivant:

**Lemma A.3.** $\forall \sigma\colon \mathsf{State}.\forall v\colon \mathbb{Z}.(\mathsf{lookup}\ \ \sigma\ v\ 3) \rightarrow$
$$\langle \sigma, (\mathsf{while}\ v\ \ \leq\ \ 1\ \mathsf{do\ skip})\rangle_{\mathsf{I}} \rightsquigarrow \sigma.$$

Dans ce cas l'état mémoire est représenté par une variable $\sigma$ quantifiée universellement. Dans la section A.1, nous avons décrit l'état comme un type inductif avec deux constructeurs, l'un décrit le cas lorsque la liste est vide et l'autre décrit le cas lorsque la liste nést pas vide. Mais aucun de ces constructeurs n'indique quoi que ce soit lorsque l'on ne sait rien sur cette liste, en d'autres termes, lorsque cette liste est décrite par une métavariable, comme $\sigma$ dans notre exemple précédent.

Dans les assistants de preuve basés sur la théorie des types, l'évaluation des fonctions est effectuée par la réduction de termes, mais cette réduction ne progresse que si la donnée observée correspond à des règles de réduction.

Le contexte peut malgré tout contenir assez d'informations reliées aux métavariables pour rendre l'exécution des instructions prévisible comme dans le lemme A.3. Pour cette raison, nous avons besoin d'un moyen pour raisonner sur elles. La solution est d'attacher un nombre aux métavariables comme $\sigma$. Nous le faisons de manière systématique. Nous définissons un nouveau type inductif appelé $\mathsf{n\_state}$ (en anglais *named state*) qui contient un constructeur supplémentaire pour ces termes numérotés.

$$\begin{array}{l}
\mathsf{n\_State}\colon \mathbf{Set} \\
[]_n\colon \mathsf{n\_State} \\
[\cdot \mapsto \cdot, \cdot]_n\colon \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathsf{n\_State} \rightarrow \mathsf{n\_State} \\
\mathsf{metavariable_s}\colon \mathbb{N} \rightarrow \mathsf{n\_State}
\end{array}$$

Une fois que nous avons affecté un nombre aux métavariables, nous devons garder leur origine. Nous le faisons en créant des tables supplémentaires où nous avons des entrées mettant en correspondance les métavariables et leur numéro. Nous avons autant de tables qu'il y a de types, donc quatre tables pour les métavariables représentant les états, les expressions arithmétiques, les expressions booléennes et les instructions. Une fois que nous avons affecté un nom à chaque expression booléenne nous définitions de nouvelles tables de résultats qui ont un rôle et une structure similaire aux tables résultats que nous utilisions dans les exemples précédentas, mais qui contiennement maintenant des expressions nommées.

De manière similaire, nous avons besoin de changer les fonctions de vérification de cohérence pour travailler sur les expressions nommées. Cette approche est sytématique. La structure du type inductif et les fonctions sur ce type sont pratiquement inchangées. Il n'y a que deux différences importantes. Premièrement nous remplaçons les entrées par leur correspondant nommé, deuxièmement nous ajoutons une règle de réduction pour prendre en compte les métavariables.

Nous démontrons que cette fonction d'évaluation est en accord avec la sémantique opérationnelle décrite par la relation $\langle \cdot, \cdot \rangle \leadsto \cdot$ (tous les théorèmes ont été vérifié formellement dans une preuve assistée par ordinateur).

Cette technique d'affectation de noms aux métavariables a un grand potentiel. Elle est systématique et ne dépend pas vraiment du langage. Notre méthode essaie d'optimiser le potentiel d'automatisation: nous avons développé une tactique résout avec succès des buts complexes. Nous espérons appliquer cette technique pour des langages plus larges et les utiliser dans des preuves sur des compilateurs. Les raison pour croire à cette application proviennent du caractère systématique des operations. Dans un travail récent sur un langage de programmation plus complet avec des procédures, nous avons pu observer que l'approche fonctionnelle se reporte agréablement à un langage de programmation plus large, bien que la sémantique requièrent des propositions inductives définies mutuellement.

Avec la présentation traditionnelle basée sur des prédicats inductifs, le raisonnement par récurrence repose sur le principe de récurrence qui est fourni pour prédicat inductif. Ce principe de récurrence correspond à ce que Winskel appelle *rule induction* dans [38]. Cette technique de preuve par récurrence est une approximation de la récurrence sur la taille des dérivations pour les preuves d'exécution, parce que les hypothèses de récurrence ne sont fournies que pour les sous-dérivations directes.

## A.3 Un compilateur

Dans l'industrie où les méthodes formelles, les preuves de programmes vérifiés par ordinateur, la vérification de modèles (en anglais *model-checking*), etc, sont utilisées pour certifier la qualité du logiciel, c'est en général le code source qui est certifié. Mais le code qui est effectivement par la machine, engendré par un compilateur, n'est pas vérifié formellement. Si nous voulons élever le niveau de certification, le compilateur devient un maillon faible entre le code source vérifié formellement et le micro-processeur vérifié formellement. George C. Necula et Peter Lee [25] on proposé de vérifier des compilateurs pour vérifier le résultat de chaque compilation plutôt que de vérifier le compilateur à partir de son propre code source. Un compilateur certifiant ne produit pas seulement un code machine, mais également une preuve de correction pour le code machine donné. Le compilateur lui-même n'a pas besoin d'être certifié.

Une solution alternative est d'utiliser un compilateur certifié. La notion de compilateur certifié est proche mais distincte de la notion de compilateur certifiant utilisée dans le contexte de code-avec-preuve (en anglais *proof-carrying-code*). Un compilateur certifié possède une preuve de correction qui est valide pour tous les programmes qu'il accepte.

En général, la majeure partie du code source pour le logiciel produit dans l'industrie est écrit en C [24]. A notre connaissance, les compilateurs C ne sont jamiais vérifiés formellement. Une solution réaliste serait de vérfier formellement un compilateur qui accepte un langage utilisable en pratique (par exemple le sous-ensemble de C utilisé pour programmer les systèmes embarqués) et qui produit du code pour un micro-processeur RISC réellement utilisé dans le monde industriel, avec des optimisations courantes. Un groupe de chercheurs s'est réuni pour aborder ce problème, dans le cadre d'une action de recherche appelée l'ARC Concert. L'objectif de Concert est de réaliser un compilateur qui est complet et modérément optimisé.

Dans ce projet nous devons définir la sémantique formelle des langages sources, du langage intermédiaire, et du langage cible. En particulier, une sémantique praticable pour un sous-ensemble de C sur une machine réaliste est nécessaire. Les utilisations possibles de cette description sémantique devrait comprendre l'utilisation dans les systèmes de preuve et l'exécution symbolique de façon à permettre le test des spécifications sur des programmes exemples. Enfin, nous avons besoin d'écrire un compilateur, directement sous la forme de fonctions Coq. En d'autres termes, un style purement fonctionnel doit être suivi, avec toutes les récursions répondant soit au schéma de la récursion primitive, soit au schéma de la récursion bien fondée. Ces contraintes dans l'écriture du compilateur permettent d'assurer qu'il termine.

L'architecture générale d'un compilateur est bien connue et n'a pas changé beaucoup au cours des 30 dernières années. Après l'analyse syntaxique et éventuellement une phase de vérification des types, une série de transformations de programmes mène le programme du langage source vers le langage cible, en passant par un ou plusieurs langages intermédiaires.

Parmi ces langages intermédiaires, le langage le plus utilisé est de la forme RTL (en anglais *Register Transfer Language*) aussi connu sous le nom anglais de *3 code addresses*. C'est un langage impératif dont les constructions de base correspondent à des instructions de processeur, sauf qu'elles opèrent sur un ensemble de pseudo-registre de taille arbitraire. Des formes plus contraintes de RTL, comme par exemple la forme SSA (en anglais *single static assignment*) ou les graphes de flôt de contrôle, sont aussi utilisées.

Certaines transformations passent d'un langage à un autre langage plus près des instructions d'un processeur: explicitation de v-tables (pour un langage à objets), sélection d'instructions (production de RTL), allocation de registres (RTL avec des pseudo-registres vers RTL avec des registres réels), génération de code machine. D'autres tranformations (les optimisations) restent dans le même langage intermédiaire mais rendent le code plus efficace: propagation de constantes, invariants de boucles, déroulement de boucles, élimination de variables d'induction. Le langage de transfert de registres est décrit comme un graphe et une dernière phase de linéarisation permet de revenir à une forme de séquence d'instructions.

Dans le projet *Concert*, dont l'objectif principal est de réaliser un compilateur certifié qui est complet et modérément optimisé, mon objectif est de fabriquer un traducteur qui construit le graph de flot de contrôle RTL à partir des programmes écrit en Cminor. Un autre objectif est de réaliser ce traducteur dans un style le plus proche possible du style de programmation en Caml, pour garder la modularité des programmes tout en surmontant les contraintes imposées par la théorie des types. La façon dont nous avons écrit le compilateur pour construire le graphe de flôt de contrôle à partir des programmes Cminor facilite probablement les preuves de correction, bien que ces preuves de correction ne fassent pas partie de notre travail.

Dans ce mémoire, nous présentons le langage source **Cminor** qui a été choisi pour notre compilatur. Cminor est un sous-ensemble de C, qui exclut certaines des constructions les plus complexes de C, par exemple **setjmp** et **longjmp**. Cminor est assez riche pour encoder la majorité des programmes C. Nous présentons ensuite le langage de transfert de registres. Enfin, nous décrivons la façon de construire les graphes de flôt de contrôle RTL à partir des des programmes Cminor.

Plutôt que de décrire le compilateur en détail, nous prenons un exemple pour comprendre comment le graphe RTL est construit, les problèmes que nous rencontrons pour modéliser le compilateur en théorie des types et les procédures que nous suivons pour surmonter ces problèmes. Pour construire le graphe RTL à partir de l'expression Cminor *Ebinop OPaddint e1 e2*, qui est une opération d'addition de deux entiers qui sont les résultats de deux expressions *e1* et *e2* nous appelons une fonction *transl_expr*. L'un de nos objectifs principaux est de conserver la modularité dans le code compilé. La fonction *transl_expr* est une fonction générale pour construire le graphe pour n'importe quelle expression Cminor. Ci-dessous, nous donnons la définition de type pour cette fonction dans l'assistant de preuve Coq.

```
Fixpoint transl_expr [envir:env; expr1:expr]:
                         Reg.T-> Graph.key -> (env*Graph.key)
```

La fonction *transl_expr* prend en argument l'environnement de compilation pour l'expression Cminor, le registre où le résultat de l'évaluation de cette expression devrait être stocké et le nœud du graphe correspondant à l'instruction par laquelle l'exécution va continuer (précisément, ce nœud devrait être exécuté après le graphe RTL correspondant à l'expression que nous considérons). La fonction *transl_expr* retourne le nouvel environnement de compilation and un pointeur vers l'instruction qui doit être exécuté pour commencer l'exécution de l'expression considérée.

Idéalement on devrait disposer d'une procédure pour transformer les opérations binaires de Cminor dans le graphe RTL de façon générale. Cette procédure devrait d'abord décider l'opération RTL choisie, elle devrait également retourner la liste des sous-expressions qui devraient être traitées également par *transl_expr*. Cette procédure devrait également allouer des registres pour le stockage des résultats de ces sous-expressions, enfin elle devrait ajouter un nœud dans le graphe de flôt de contrôle appeler la fonction *transl_expr* pour chacune des sous-expressions. Cette procédure devrait avoir la forme suivante.

```
(Ebinop binop e1 e2) =>
   let (reg_fn, opel) = (select_binop binop e1 e2) in
   let (op', el) = opel in
   let (envir', rl) = (alloc_regs envir el) in
   let (envir'', no) =
    (add_instr envir'
      (mkinstr (Iop op') (eval_reg_list_trans reg_fn rl)
              (Some Reg.T rd)) (cons nd (nil Graph.key))) in
       (transl_subexprs envir'' rl el no)
```

La fonction *select_binop* décide l'opération RTL et décompose l'expression Cminor en une liste de sous-expressions. Dans la fonctions *transl_subexprs* la fonction *transl_expr* est appelée sur chacun des éléments de la liste d'expression *el*. Bien qu'en réalité nous appelons effectivement la fonction récursive sur des arguments qui sont des sous-termes de l'argument initial de *transl_expr*, la liste d'expressions retournée par *select_binop* n'apparait pas comme un sous-terme de l'argument initial vis-à-vis des conditions de gardes imposées pour la programmation récursive structurelle. En d'autre termes, la procédure ci-dessus n'est pas bien formée.

Pour conserver la modularité et la bonne formation, nous utilisons une fonction d'ordre supérieur, une technique que nous avons déjà exploitée dans le section A.1. Dans l'approche précédente la fonction *transl_expr* était la seule fonction à construire des graphes RTL. Dans le nouveau schéma, nous appelons la fonction *transl_binop* pour construire le graphe pour les opérations binaires. Les fonctions *transl_expr* et *transl_binop* sont interdépendantes puisque la fonction *transl_expr* construit le graphe pour chacune des sous-expressions d'une expression binaire. Dans notre exploitation d'une fonction d'ordre supérieur, nous supposons l'existence d'une fonction *transl_expr* au moment de définir la fonction *transl_binop*: cette fonction *transl_expr* est passée en argument. La fonction *transl_expr* est ensuite définie en appelant la fonction *transl_binop* en lui donnant la fonction *transl_expr* en argument. Voici la définition de la procédure de construction de graphe pour le cas des opérations binaires: il est bien formé et conserve la modularité du code:

```
(Ebinop binop e1 e2) =>
   (transl_binop transl_expr
       [op',rl,envf]
       (add_instr envf (mkinstr (Iop op') rl (Some Reg.T rd))
                   (cons nd (nil Graph.key)))
              envir binop e1 e2)
```

Pour toute opération binaire nous avons besoin d'ajouter un nœud dans le graphe qui contient l'opération effectuée. Ce besoin est commun à toutes les opérations binaires et nous réduisons la quantité de code du compilateur si nous définissons une fonction *transl_binop* pour prendre en charge toute la partie commune. Le système Coq facilite ce type de programmation modulaire. Ci-dessous nous montrons comment le graphe de flôt de contrôle est construit pour l'opération d'addition.

```
Section SELECT_BINOP.

Variable transl_expr: env -> expr -> Reg.T
                          -> Graph.key -> (env*Graph.key).

Variable f: operation -> (list Reg.T)
                      -> env -> (env*Graph.key).

Definition  transl_binop
 [envbinop:env; binop:binary_operation;
                expbinop1,expbinop2:expr]:
                                  (env*Graph.key):=
⋮

| OPaddint e1 e2 =>
   let (envbinop',r1) = (alloc_reg envbinop e1) in
   let (envbinop'',r2) = (alloc_reg envbinop' e2) in
   let (envbinop''',n2) =
    (f Oadd
      (eval_reg_list_trans reg_id
                   (cons r1 (cons r2 (nil Reg.T))))
         envbinop'') in
    let (envbinop'''',n1) =
              (transl_expr envbinop''' e2 r2 n2) in
      (transl_expr envbinop'''' e1 r1 n1)
⋮
```

```
End SELECT_BINOP.
```

En résumé, pour construire le graphe de flôt de contrôle pour l'expression Cminor qui contient une expression binaire, nous appelons la fonction *transl_expr*, cettte fonction appelle *transl_binop*, la fonction *transl_binop* décompose l'expression en quelques sous-expressions et alloue des registres pour chacune de ces sous-expression. Esuite la fonction *transl_binop* appelle la fonction *f* qui décide l'opération RTL qui est employée et ajoute un nœud pour cette opération dans le graphe. Finallement, la fonction *transl_binop* appelle *transl_expr* pour construire le graphe pour chacune des sous-expressions.

Dans cette thèse, nous avons donc étudiés différentes approches pour utiliser des fonctions dans la description de langages de programmation et d'outils pour les langages de programmation. Nous montrons plusieurs techniques pour représenter les constructions de programmation avec de la récursion mutuelle et de la récursion imbriquée en théorie des types. Nos techniques sont déjà utiles pour la démonstration de compilateurs dans le cadre du projet *Concert*.

# Bibliography

[1] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [20], pages 1–16.

[2] Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. *JFLA*, 13:27–42, 2002.

[3] G. Barthe, M. Ruys, and H. P. Barendregt. A two-level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs (TYPES'95)*, volume 1158 of *LNCS*, pages 16–35. Springer, 1995.

[4] Yves Bertot. Filters on coinductive streams an application to eratosthenes' sieve. Unpublished Mansucript, 2004.

[5] Yves Bertot, Venanzio Capretta, and Kuntal Das Barman. Type-theoretic functional semantics. In Victor Carreño, César Muñoz, and Sofiène Tashar, editors, *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, volume 2410 of *Lecture Notes in Computer Science*, pages 83–98. Springer-Verlag, 2002.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions.* Springer Verlag, 2004.

[7] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In *International Joint Conference of Theory and Practice of Software Development (TAPSOFT/FASE'95)*, volume 915 of *LNCS*. Springer-Verlag, 1995.

[8] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software. Third International Symposium, TACS'97*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.

[9] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.

[10] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.

[11] Claudio Sacerdoti Coen. *Knowledge Management of Formal Mathematics and Interactive Theorem Proving.* PhD thesis, University of Bologna, 2003.

[12] Nancy A. Day and Jeffrey J. Joyce. Symbolic functional evaluation. In Yves Bertot, Gilles Dowek, André Hirschowits, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 341–358. Springer-Verlag, 1999.

[13] Catherine Dubois and Véronique Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. Presented at CADE-15, Workshop on Mechanization of Partial Functions, 1998.

[14] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[15] Simon Finn, Michael Fourman, and John Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, February 1997.

[16] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[17] Michael Gordon and Tony Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[18] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF : A Mechanized Logic of Computation.* Springer Verlag (LNCS 78), 1979.

[19] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. *Proceedings of ICFP - ACM Sigplan*, 37(9):235–246, 2002.

[20] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[21] Dimitri Hendriks. Proof reflection in coq. *Journal of Automated Reasoning*, 29(3–4):277–307, 2002.

[22] A. Heyting. *Intuitionism, an Introduction.* North-Holland, 1956.

[23] Glles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[24] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

[25] George Necula and Peter Lee. The design and implementation of a certifying compiler. *Proceedings of PLDI - ACM Sigplan*, 33(5):233–244, 1998.

[26] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 180–192. Springer, 1996.

[27] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, 1993. LIP research report 92-49.

[28] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[29] Lawrence C. Paulson. Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, 2:63–74, 1986.

[30] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[31] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[32] Gordon Plotkin. A powerdomain construction. In *SIAM Journal on Computing*, volume 5(3), pages 452–487, 1976.

[33] The Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual. Version 7.2.* INRIA, 2001.

[34] K. Slind. Another look at nested recursion. In Harrison and Aagaard [20], pages 498–518.

[35] Morten Heine B. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[36] Kumar Neeraj Verma and Jean Goubault-Larrecq. Reflecting bdds in coq. Technical report, INRIA RR-3859, 2000.

[37] Freek Wiedijk and Jan Zwanenburg. First order logic with domain conditions. Available at `http://www.cs.kun.nl/~freek/notes/partial.ps.gz`, 2002.

[38] Glynn Winskel. *The Formal Semantics of Programming Languages, an introduction.* Foundations of Computing. The MIT Press, 1993.

# Index

$\beta$-reduction, 11
$\delta$-reduction, 11
$\iota$-reduction, 11, 15
$\zeta$-reduction, 11

accessibility, 36
arrow types, 10
atomic types, 10
axiomatic semantics, 9

bounded recursion, 19

call by value, 12
Coq, 10

denotational semantics, 6, 28
dependent types, 13
domain, 8

EAuto, 45

fix, 16
fix-point, 6
fixpoint, 16
functional interpretation, 31

guardedness condition, 91, 97

IMP, 27
inductive predicates, 17
inductive type, 14
iter, 31, 53
iteration, 21

lazy, 12
Ltac, 55

metavariable, 49

natural semantics, 5

operational semantics, 3, 28

partial order, 7
Prop, 12

recursion, 19
recursive type, 15
reflection, 48

Set, 12
simul. induction-recursion, 37
structural operational semantics, 4

Type, 12

well founded recursion, 20

## Abstract

Semantics of programming languages gives the meaning of program constructs. Operational and denotational semantics are two main approaches for programming languages semantics. Operational semantics is usually given by inductive relations. Denotational semantics is given by partial functions. Implementing the denotational semantics inside type theory is difficult as the type theory expects total functions.

In this dissertation we develop a functional semantics for a small imperative language inside type theory and show its equivalence with operational semantics. We then exploit this functional semantics to obtain a more direct proof search tool, while developing a way to describe and manipulate unknown expressions in the symbolic computation of programs for formal proof development. In a third part, we address the problem of encoding complex programs inside type theory and we show how to circumvent the limitations of guardedness conditions as the are used in the Calculus of Inductive Constructions.

**Key words:** Type Theory, CIC, Coq, Semantics, Reflection, Compiler

## Résumé

La sémantique des langages de programmation donne la signification des constructions de programme. Les sémantiques opérationnelle et dénotationelle sont les deux principales approches pour la sémantique de langages de programmation. La sémantique opérationnelle est habituellement donnée par des relations inductives. La sémantique dénotationelle est donnée par des fonctions partielles. Mettre en application la sémantique dénotationelle à l'intérieur de la théorie des types est difficile car cette théorie ne supporte que les fonctions totales.

Dans cette thèse nous développons une sémantique fonctionnelle pour un petit langue impératif à l'intérieur de la théorie des types et montrons son équivalence avec la sémantique opérationnelle. Nous exploitons ensuite cette sémantique fonctionnelle pour obtenir un outil plus direct de recherche de preuve, tout en développant une manière de décrire et manipuler des expressions inconnues dans le calcul symbolique des programmes pour le développement formel de preuve. Dans une troisième partie, nous adressons le problème de coder des programmes complexes à l'intérieur de la théorie des types et nous montrons comment éviter les limitations des conditions de garde telle qu'elles sont employés dans le calcul des constructions inductives.

**Mot clés:** Théorie des types, CIC, Coq, Sémantique, Réflexion, Compilateur