INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *A certified compiler for an imperative language*

Yves Bertot

**N° 3488**

Septembre 1998

THÈME 2

Rapport de recherche

ISSN 0249-6399

# A certified compiler for an imperative language

Yves Bertot

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport de recherche n° 3488 — Septembre 1998 — 30 pages

**Abstract:** This paper describes the process of mechanically certifying a compiler with respect to the semantic specification of the source and target languages. The proofs are performed in type theory using the Coq system. These proofs introduce specific theoretical tools: fragmentation theorems and general induction principles.

**Key-words:** Automatic proof, Natural semantics, Compilation, Type theory, Coq, CtCoq, Assembly language¸ Formal Methods

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

# Un compilateur certifié pour un langage impératif

**Résumé :** Cet article décrit la vérification mécanique de la démonstration de certification d'un compilateur vis-à-vis des spécifications sémantiques du langage source et du langage cible. Ces vérifications sont effectuées dans le formalisme de la théorie des types, à l'aide du système Coq. Cette vérification permet d'introduire des outils théoriques adaptés: théorèmes de fragmentation et principe de récurrence général.

**Mots-clés :** Démonstration automatique, sémantique naturelle, compilation, théorie des types, Coq, CtCoq, Assembleur, Méthodes formelles

# 1 Introduction

This paper describes an experiment in mechanically checking the proof of correctness for a compiler. The source language is a simplistic imperative language corresponding to the basic kernel common to all imperative languages and the target language is an assembly language with a goto statement that also corresponds to a very basic subset of most machine languages for micro-processors. The proof has been verified using the Coq proof assistant [17] and its graphical interface CtCoq [3].

The main complications in this work come from the lack of structure in the assembly language used as the target language. The contributions of this work provide ways to recover the missing structure, first by making it possible to reason independently on various program fragments, even though the presence of *goto* statements hinders this kind of independence, and second by proposing techniques for proof by induction that are more powerful than those normally provided by inductive tools, as embodied in inductive packages [11, 24, 28] or proof systems [17, 20, 21].

In the rest of this section we give an overview of inductive type theory that will serve as the language for all the reasoning described in this paper, we describe more precisely the languages addressed by this experiment, we detail the main contributions of this paper, and we study how they compare to related work in this domain. In the next section, we concentrate on the first main contribution of the paper, a set of theorems to decompose the execution of assembly programs into fragments and to recover structure in this unstructured programming language. The third section describes precisely the structure of the proof and exhibits a difficulty in the treatement of looping constructs. The fourth section develops a solution for this issue. The fifth section groups some conclusive remarks.

## 1.1 Type theory, induction, and computation

The logical framework in which this mechanical proof has been performed is that of type theory [22] with inductive types, as provided by the inductive calculus of constructions [17, 13, 27]. The inductive aspects of type theory provide the user with the possibility to define sets by giving a collection of constructors for these sets. To each such collection of constructors, the theory associates basic recursors. This inductive capability also makes it possible to define inductive properties, corresponding to inductive families of dependent types. For instance, the order "less or equal" on natural numbers can be obtained with the following two defining axioms:

$$
\begin{aligned}
le\_n &: \forall n : nat . n \leq n, &(1)\\
le\_S &: \forall n, m : nat . n \leq m \Rightarrow n \leq (m+1). &(2)
\end{aligned}
$$

The two argument property "less or equal" is the smallest subset of $\mathcal{N} \times \mathcal{N}$ such that the two theorems $le\_n$ and $le\_S$ always hold. This is expressed by the following “induction theorem”:

$$
\begin{array}{l}
\forall P: \ nat \rightarrow nat \rightarrow Prop. \\
\quad (\forall n : nat.\ P(n,n)) \Rightarrow \\
\quad (\forall n, m : nat.\ P(n,m) \Rightarrow P(n,(m+1))) \Rightarrow \\
\qquad\qquad \forall n, m : nat.\ (n \leq m) \Rightarrow P(n,m).
\end{array}
$$

Another way to view this is that whenever $n \leq m$ holds, there must exist a finite derivation that proves this fact, constructed only with the two statements *le_n* and *le_S*. We can then reason by induction on the structure of this derivation. This technique derived from the work on natural deduction (proofs on the degree of derivations are used for the *cut-elimination* theorem in [33]) is called *rule induction* in [35]. In this paper, we shall use rather the term: *induction on the structure of a derivation*.

These inductive aspects will play an important role in our work. Inductive sets will be used to encode the syntax of programming languages: each syntactic category of the language will be represented as an inductive type and each basic construct of the programming language will be represented as a constructor for one of these types. The basic recursors associated to inductive types will be used to define total recursive functions over these types and, for instance, the compiler will be one of these recursive functions. Lastly, inductive properties will be used to describe properties of programs.

Inductive type theory provides a single framework to model both a pure fragment of functional programming languages using recursive functions and a pure fragment of logic programming languages using inductive properties. The many experiments done with natural semantics have shown that logic programming was a suitable framework to describe the semantics of programming languages. Thanks to the work done to relate natural semantics descriptions and inductively defined types [34], we shall use inductive sets to represent source and target programming languages, inductive properties to describe the semantics of these programming languages, and recursive functions to describe the compiler.

## 1.2 Source and target languages

The source language studied in this paper is a basic language where programs are composed of a block of variable declarations and an instruction. Each variable declaration provides both the initial value for the variable and a type declaration. Instructions are composed of assignments, conditional instructions, while loops and sequences. The expressions used in conditions and as right-hand sides of assignments are boolean or integer expression, with "and", "or", "not", "eq", "plus", or "minus" operations. This language is very close to the *imp* language of [35], but we have not given a syntactic separation between boolean expressions and arithmetic expressions, so that the programming language needs a type-checker to ensure consistent use of boolean and arithmetic variables. This source language has already been used for other experiments of mechanically checked proofs [7].

In our proof development, the syntactic description of the language is given by mutually inductive types *PROGRAM*, *INST*, *EXP*, *DECLS*, *DECL*, *TYPE*, *ID*, and *VAL*, corresponding to a variety of syntactic categories. We shall only detail a few of these types. *INST* is described by four constructors, with the following types:

- *assign: ID* $\rightarrow$ *EXP* $\rightarrow$ *INST*, represents an assignment.
- *if: EXP* $\rightarrow$ *INST* $\rightarrow$ *INST* $\rightarrow$ *INST*, represents a conditional construct.
- *sequence: list(INST)* $\rightarrow$ *INST*, represents a sequence of instructions.
- *while: EXP* $\rightarrow$ *INST* $\rightarrow$ *INST*, represents a while loop.

*DECLS* represents the category of declaration blocks it has one constructor:

- *decls: list(DECL)* $\rightarrow$ *DECLS*

*DECL* represents the category of single declaration, it has one constructor:

- *decl: ID* $\rightarrow$ *TYPE* $\rightarrow$ *VAL* $\rightarrow$ *DECL*

Note that a variable declaration contains both the intended type for the variable and the initial value.

The dynamic semantics of the language is described using a collection of inductively defined properties that describe the behavior of each syntactic category. For instance, *eval(D,E,V)* is well-formed when *D, E*, and *V* have respective types *DECLS*, *EXP*, and *VAL*. When this proposition holds, it expresses that evaluating *E* in the environnement *D* works correctly and returns the value *V*. The other important property is *exec*: *exec(D,I,D′)* is well-formed when *D, I*, and *D′* have respective types *DECLS*, *INST*, and *DECLS*. When this property holds, it means that executing *I* in the initial state *D* terminates normally and returns the new state *D′*. This property is defined by the constructors given in figure 1.

The target language used in this paper represents a simple assembly language. It contains fourteen instructions devised to pilot a machine with a register, a stack, and a memory. Instructions are *label*, *goto*, *branch_if_0* to manage the control flow, *push*, *pop*, to manage the stack, *add*, *sub*, *and*, *not*, *eq*, *gt*, to compute values, *load*, *store* to exchange data between memory locations and the register, and *immediate* to store data in the register. These instructions are grouped in an inductive type named *Assembly*. All computation instructions return their value in the register, those that take two arguments take one in the register and one on top of the stack, which is popped in the process. Branching instructions only refer to label instructions. All values are natural values, even though some instructions have a boolean connotation. For boolean instructions, *1* stands for the true value and *0* for the false value. Also, *plus* and *minus* implement unbounded operations. In our proof development, an abstract machine for this assembly language is implemented by encoding the stack and the memory as lists of natural numbers and the register as a natural number. This data is grouped in a three field record named *exec_state*. The computation of the new state when executing an instruction is described using three functions:

- *new_stack: nat* $\rightarrow$ *list(nat)* $\rightarrow$ *Assembly* $\rightarrow$ *optional list(nat)*.
  This is a partial function that computes the new stack depending on the old stack and the instruction.

$\forall$ *e: EXP.* $\forall$ *d, d′: DECLS.* $\forall$ *I: ID.* $\forall$ *v: VAL.*
*eval(d,e,v)* $\Rightarrow$ *update(d, I, v, d′)* $\Rightarrow$ *exec(d,assign(I,e), d′)*

$\forall$ $i_1$, $i_2$*: INST.* $\forall$ *e: EXP.* $\forall$ *d, d′: DECLS.*
*eval(d,e, true)* $\Rightarrow$ *exec(d,*$i_1$*, d′)* $\Rightarrow$ *exec(d, if(e,* $i_1$*,* $i_2$*), d′)*

$\forall$ $i_1$, $i_2$*: INST.* $\forall$ *e: EXP.* $\forall$ *d, d′: DECLS.*
*eval(d,e, false)* $\Rightarrow$ *exec(d,* $i_2$*, d′)* $\Rightarrow$ *exec(d, if(e,* $i_1$*,* $i_2$*), d′)*

$\forall$ *i: INST.* $\forall$ *e: EXP.* $\forall$ *d,: DECLS.*
*eval(d,e, false)* $\Rightarrow$ *exec(d, while(e, i), d)*

$\forall$ *i: INST.* $\forall$ *e: EXP.* $\forall$ *d, d′, d″: DECLS.*
*eval(d,e, true)* $\Rightarrow$ *exec(d, i, d′)* $\Rightarrow$ *exec(d′, while(e, i), d″)* $\Rightarrow$
*exec(d, while(e, i), d″)*

$\forall$ $i_1$*: INST.* $\forall$ $i_2$*: list(INST).* $\forall$ *d, d′, d″: DECLS.*
*exec(d,* $i_1$*, d′)* $\Rightarrow$ *exec(d′, sequence(*$i_2$*), d″)* $\Rightarrow$ *exec(d, sequence(*$i_1$*.* $i_2$*), d″)*

$\forall$ *d: DECLS. exec(d, sequence([]), d)*

Figure 1: The dynamic semantics of the source language.

- *new_register: nat* $\rightarrow$ *list(nat)* $\rightarrow$ *list(nat)* $\rightarrow$ *Assembly* $\rightarrow$ *optional nat.* This is a partial function that computes the new value of the register depending on the old register, the old memory, the old stack, and the instruction.
- *new_mem: nat* $\rightarrow$ *list(nat)* $\rightarrow$ *Assembly* $\rightarrow$ *list(nat).*
  This is a total function that computes the new memory depending on the value of the register, the old memory, and the instruction.

An important part of the machine behavior is also given by the way the next executed instruction is chosen, this is described using one function:

- *new_pc: list(Assembly)* $\rightarrow$ *list(Assembly)* $\rightarrow$ *nat* $\rightarrow$ *Assembly* $\rightarrow$
  *optional list(Assembly).*
  This is a partial function that computes a new list of instructions, depending on the whole executed program, the program fragment that follows the currently executed instruction, the value of the register, and the current instruction.

Using these functions we describe the behavior of the abstract machine using an inductively defined property *execute* with the following type:

*execute: list(Assembly)* $\Rightarrow$ *exec_state* $\Rightarrow$ *list(Assembly)* $\Rightarrow$ *exec_state* $\Rightarrow$ *Prop.*

Intuitively, the statement *execute(pg,s,pg′,s′)* means: *executing the program pg′ in context pg with initial state s terminates and returns the final state s′.* There are two constructors for this definition, the first one, *execute_end*, expresses that executing an empty program terminates without changing the state:

$\forall$ *pg: list(Assembly).* $\forall$ *s: exec_state. execute(pg, s, [], s).*

The second constructor, *execute_rec* expresses that to execute a non-empty program, you first have to check that computing the new stack, register, and program works correctly, and then you can continue your execution with the new stack, memory, register, and program:

$\forall pg,\ pg',\ pg''$*: list(Assembly).* $\forall$ *i: Assembly.*
$\forall$ *m, s, s'*: *list(nat).* $\forall$ *r, r'*: *nat.* $\forall$ *result: exec_state.*
*new_pc(pg, pg', r, i)* $=$ *pg''* $\Rightarrow$ *new_stack(r, s, i)*$=$ *s'* $\Rightarrow$
*new_register(r, m, s, i)* $=$ *r'* $\Rightarrow$
*execute(pg, (new_mem(r, m, i), s', r'), pg'', result)* $\Rightarrow$
*execute(pg, s, i.pg', result).*

## 1.3 Contributions

Thanks to the extraction mechanism provided in Coq, it is possible to obtain a program from the proved algorithm *with no manual re-writing*, thus avoiding spurious errors (although the front end of the compiler, that takes care of parsing the input string, and the back end, that takes care of printing the output in a file have to be written by hand). Today, we believe there is only one other comprehensive proof of a compiler that has been *completely* verified mechanically (see section 1.4).This work may also contribute ideas to researchers interested in compiler verification for two aspects of reasoning on assembly language execution.

Assembly programs have no structure and their behavior is very context dependent. This is due to the semantics of branching statements. To study these statements, we have devised a more abstract form of assembly language which isolates the control flow part, giving a special status to label and branching statements. With this abstract assembly language, we have described two properties that are interesting for assembly language programs. First, it is important that assembly programs be *unique*, in the sense that all labels are used only once. Second, it is important that program fragments be *self-contained*, in the sense that all branching statements in the program fragment refer to labels that are also in the program fragment. We have then proved three basic *fragmentation theorems* that show that the execution of self-contained fragments can be isolated from the context as if they were single plain (i.e., non branching) instructions. This is our first contribution.

Several theorem provers provide packages for defining inductive sets or properties and reasoning by induction on these objects. When considering the execution of the compiled code for loop constructs, the basic induction principles provided by these packages are usually sufficient but unwieldy. We use a comparison with the notion of course-of-value induction on natural numbers to describe a technique to derive and prove a more general induction theorem. This new induction theorem will be more adapted for the formal study of looping assembly programs. This our second contribution.

### 1.4 Related work

McCarthy and Painter [23] addressed the question of proving the correctness of a compiler for simple arithmetic expressions. The target language describes a single address computer with an accumulator and four instructions, of which none breaks the control flow. This is an interesting seminal work, but it does not address mechanical verification[1] or control flow issues, as the target language does not contain conditional or branching instructions. Many researchers have built upon this experiment to study various approaches to proving compiling algorithms, but seldom with mechanized verification in mind. Polak [30] gives a summary of this work.

Polak [30] presents the verification of a Pascal-like programming language using the Stanford Verifier [31]. That work is more comprehensive than ours, as Polak also proved the correctness of the front end part: the lexical analyser is proved correct with respect to a regular expression description and the parser is proved correct with respect to an LR context-free grammar. Moreover, the language addressed is much stronger that our basic example: procedures and pointers are allowed in the language. However, the logic used in the Stanford verifier is first order and some of the proofs in Polak's work were carried out by hand, in fact almost all the proofs that involved induction. By comparison, we have also mechanically verified the generalized induction principles that we used. Another significant difference between the two pieces of work is that Polak relies on a denotational semantics description of the programming languages while our description is based on natural semantics, which we believe requires less mathematical background.

Moore and his team performed the mechanical verification of a high-level assembly programming language called Piton and an Algol-like language called Gypsy [25, 36] using the Nqthm theorem prover [10]. In some sense, our experiment is close to that work, as in both cases the mechanical verification tool embeds a programming language in which the compiler is written. In their case, the programming language is Lisp. However, the logic provided by Nqthm is weaker than the one provided by Coq and the semantic descriptions suffer from this lack of power. For instance, programming language dynamic semantics can only be represented using a total computable function, so that it was necessary to add a "step count" argument to this function to ensure the termination of this function. It is then not possible to express that a program executes and terminates in general: instead, one can only state that a program executes and terminates in less than $n$ steps. The $n$ argument then gets into the way when proving equivalence between steps. By comparison, the work described in this paper attempts to show that type-theory is a more suitable framework to perform these proofs. Also, the proof by induction performed in [36] is said to use an induction hint that has 12 parameters and is 250 lines long. We believe our method for structuring the proof, using fragmentation theorems and a generalized induction principle, can help make the human intervention less like impressive witchcraft.

More recently, Wand, Guttman, Oliva, Ramsdell, and Swarup [18] described the complete proof of a compiler for the scheme language, starting from a denotational semantics

[1] According to Polak, some mechanical verification has been done by W. Diffie.

description. This proof is very interesting as it comprehensively studies the various aspects of the language, for instance including garbage collection and optimization. The proof has been done completely by hand and the compiler has been implemented, so that its efficiency and the efficiency of the code it produced could be compared with traditional compilers. The distance between that work and ours lies in the fact that the compiler verification relies on denotational semantics to describe the source language and many of the intermediate languages, although operational semantics is used for the last target language. Another significant difference is their use of infinite regular trees to represent programs, so that the question of understanding how to reason around a linear representation with goto statements is discussed only very late in their development. When they do, they go directly to a machine that uses an integer program counter instead of a machine with lists of instructions and symbolic addresses as we do.

More closely related to this effort is all the work around *natural semantics* [19]. Natural semantics is a style of programming language description derived from natural deduction and closely related to *structural operational semantics* [29]. Early work has shown that natural semantics could be used to describe thoroughly a variety of programming languages [12] and that programming environments could be based on these descriptions [16, 4, 2, 1]. More precisely on the proof of compilers, J. Despeyroux presented in [15] a proof of translation between a kernel of the ML language and an abstract machine. In that work, the target language remains a structured language, thus making it possible to avoid the issue of branching statements. Also this proof was done, by hand, without mechanical verification.

# 2 Abstract study of `goto` statements

## 2.1 A very abstract machine

To study `goto` statements, it is not necessary to give a completely concrete description of a microprocessor. It is only necessary to consider a processor with some abstract state that executes instructions that may modify this state, among which some instructions are *labels* and some are *branching constructs*. For the labels and branching statements, it is not necessary to know exactly whether they are indexed by numbers or character strings. It is only necessary to know that they use an abstract type for which one has a function that computes whether two labels are equal.

We shall thus reason using three abstract data types: *Inst*, *Lab*, and *state* that represent respectively instructions, label values, and abstract machine states. Using these abstract sets, one can consider an assembly language named *abs_assembly* that only contains three kinds of instructions:

- *plain(i)* where $i \in Inst$. This kind corresponds to instructions that modify the state and transfer the control to the instruction after them.
- *label(l)* where $l \in Lab$. This kind corresponds to instructions that do not change the state but mark some location in the sequence of instructions.

- *branch(i,l)* where $i \in Inst$ and $l \in Lab$. This kind corresponds to instructions that modify the state and may branch to the first instruction following *label(l)*.

In our development *abs_assembly* is an inductive type and *plain*, *label*, and *branch* are its constructors.

Branching while executing a *branch* command is conditioned by the state of the abstract machine. When branching does not occur, control is simply passed to the next instruction.

The fact that instructions may modify the state of the processor is represented by a partial function *new_state* that takes two arguments: an instruction from the set *Inst* and a state from the set *state*. In our development *new_state* is a variable with the following type:

$$new\_state : Inst \rightarrow state \rightarrow optional\ state.$$

The fact that branching is conditioned by the state of the processor is modeled by a function *test_branch* that also takes the instruction and the state as argument. In our development *test_branch* is a variable with the following type:

$$test\_branch : Inst \rightarrow state \rightarrow bool.$$

The fact that the processor is able to resume execution at the right point in the program after a branching instruction is modeled by a function *test_eq_lab* that compares two labels. In our development *test_eq_lab* is a variable of type:

$$test\_eq\_lab : \prod x,\ y\!: Lab : \{x{=}y\}{+}\{\neg x{=}y\}.$$ [2]

As far as states are concerned, there are only two cases: either the current instruction contains an object from *Inst*, and in this case *new_state* is applied to this object and the current state to obtain the new state, either the instruction is a label, and in this case the new state is the same as the previous one. This behavior is modeled by a partial function *next_state* that takes an instruction from the set *abs_assembly* and a *state* as argument and optionally returns a *state*. Cases where the *next_state* function does not return a *state* value correspond to run-time errors: we choose to express that execution fails altogether when an error occurs.

As far as control flow is concerned, there are also only two cases: if the instruction is a branching statement, one calls *test_branch* with the instruction and the initial state to compute whether the branch will occur. If the branch does not occur or if the instruction is a label or a plain instruction the control is simply transferred to the next instruction. This behavior is modeled by a function *next_step* that takes four arguments: the instruction's body (that belongs to the set *Inst*), the initial state, the whole program, and the program fragment that immediately follows the current instruction. This function returns an optional program fragment, that is, a list of *abs_assembly* instructions. When branching occurs,

[2]This dependent type expresses that the value returned also contains a proof that the compared values are equal or different. This will be handy when reasoning on programs.

the next program fragment to execute is found by traversing the whole program until one finds the right label, using the function *test_eq_lab* to compare labels. Note that the function *next_step* may fail when no label instructions are found with the right value. In our development, the function *next_step* uses an auxiliary function *find_lab* that performs the search in the context program when branching occurs. This function is defined using the structural recursion capabilities of the Coq system.

Executing an abstract assembly program is modeled using an inductive property called *machine* that repeatedly applies *next_state* and *next_step* to determine the successive abstract states and instructions to execute. The predicate *machine* is a four-place predicate, taking as arguments two programs (the whole context of execution, and the current program)and two states (the initial state and the final state). The proposition $machine(pg, pg', s, s')$ must be interpreted as *in context pg, the execution of the program pg′ from the initial state s terminates and leads to the final state s′*. The inductive definition of this proposition has two constructors. One is used to represent termination (when $pg'$ is empty), it is called *mach_end*:

$\forall$ *s: state.* $\forall$ *pg : list(abs_assembly). machine(pg, [], s, s).*

The other constructor is used to represent the execution of one step, applying *next_state* and *next_step*, checking that they return actual values, and proceeding with their results as new state and new program to execute; it is called *mach_step*:

$\forall\ s, s', s''\!: state.\ \forall\ i\!: abs\_assembly.\ \forall\ pg, pg', tail\!: list(abs\_assembly).$
$next\_step(i, s, pg, tail) = pg' \Rightarrow next\_state(i, s) = s'$
$machine(pg, pg', s', s'') \Rightarrow machine(pg, i.tail, s, s'').$

Although this abstract model of a processor is very simple, it is complete enough to represent all the instructions used in our assembly language. However, it would have to be extended to admit also instructions that provide indirect jumps, where the target of a branch instructions is computed from the state (thus making it possible to have branching constructs that branch to several different locations). Such indirect jumps make it much simpler to compile procedure calls or switch statements as found in the `C` programming language.

## 2.2 The unique property

The objective of our proof development at this abstract level is to provide tools that make it possible to reason on program fragments in isolation from the whole context. For this, we want to be able to establish a correspondance between executing a program fragment when the context is the program fragment itself and executing the program fragment when the context is a larger program. In more concrete terms, we want to relate instances of the following proposition:

$$machine(pg', pg', s, s')$$

and corresponding instances of the following proposition:

$$machine(pg,\ pg',\ s,\ s').$$

We first need that $pg'$ be a part of the whole program, which can be expressed with the following equality:

$$pg = pg_1.pg'.pg_2.$$

A second important consistency property is that branching behaves "the same", that is, that the program fragment executed after branching must be similar in both settings.

To ensure this, it is important that all the labels in $pg$ be unique. In our development, *unique(pg)* means that all labels occuring in $pg$ occur only once. The property *unique* has been defined using two inductive definitions.

A first interesting theorem is that if labels are unique, the next step of $pg'$ with $pg'$ as context is a prefix of the next step of $pg'$ with $pg_1.pg'.pg_2$ as context (this theorem is called *unique_next_step*):

$\forall\ pg_1,\ pg',\ pg_2,\ pg'',\ tail\text{: } list(abs\_assembly).$
$\forall\ i\text{: } abs\_assembly.\ \forall\ s\text{: } state.$
$next\_step(i,\ s,\ pg',\ tail) = pg'' \Rightarrow unique(pg_1.pg'.pg_2) \Rightarrow$
$\exists pg'''\text{: } list(abs\_assembly).\ next\_step(i,\ s,\ pg_1.pg'.pg_2,\ tail) = pg''.pg'''.$

The property *unique* exhibits some stability with respect to program concatenation. First, if a program is the concatenation of several fragments and has unique labels, then all the fragments have unique labels (the converse is false). Second, the uniqueness property is preserved through any shuffle of the fragments. This stability with respect to concatenation will be instrumental in the compiler proof, as the compiler simply compiles a construct by concatenating the fragments obtained by compiling the sub-components and adding a few instructions.

## 2.3 The self-contained property

Intuitively, program fragments that are placed one after the other in a program are executed one after the other. But this is only partly true, as branching statements will often interfere and disrupt the control flow. However, it is possible to describe large classes of program fragments that will respect this intuitive property. A first class of programs contains programs that consist only of *plain* and *label* statements. These programs obviously do not suffer from any control flow breaks.

A more subtle class, for which control flow can be sorted out, contains program fragments that are *self contained*. These are fragments where every branching instruction refers to a label that is also present in the fragment. A graphical image for this class of fragments is given in figure 2. In our proof development, *self_contained(pg)* means that $pg$ is self contained, this property has been defined using two inductive definitions.

When the whole program has the unique property, self contained program fragments are a very close approximation of plain instructions or programs containing only plain instructions in two ways:

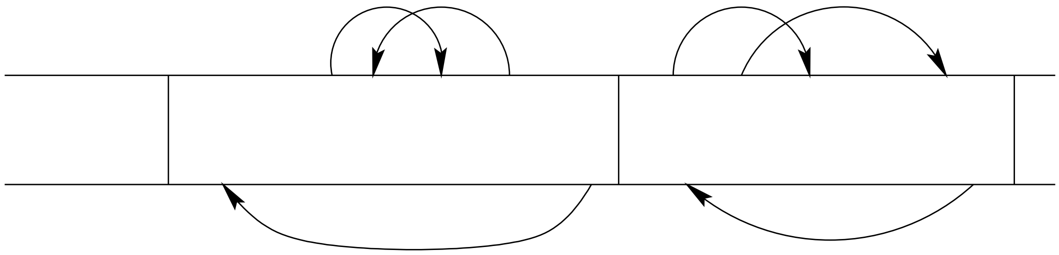

Figure 2: A graphical intuition of self contained programs.

- The behavior of these program fragments does not depend on the context.
- When the execution leaves a self contained program fragment, the next instruction is always the first instruction that follows. In other terms, execution does not "jump out" of a self contained program fragment, it simply leaves through the back door. This does not mean that self-contained fragments always execute correctly and terminate, as their execution may encounter run-time errors (like plain instructions) and they may also loop.

Here again, the self-contained property exhibits some stability with respect to concatenation: the concatenation of self-contained program fragments remains self-contained (but the fragmentation of a self contained program fragment does not give self-contained fragments). The self-contained property is also preserved through any shuffle of the various fragments.

## 2.4 Fragmentation theorems

There are two situations in which we need fragmentation theorems. In the first situation, we know that the whole program executes and we want to show that this execution can be studied by looking separately at the execution of fragments. In the second situation, we know that fragments can be executed separately and we want to prove properties of the whole program.

To these situation correspond two main theorems. The first main theorem applies when one already knows that executing the concatenation of a self-contained program and an arbitrary program fragment terminates. It states that one can first execute the self contained program, using itself as context, and then execute the other fragment, this time with the whole program as context. This theorem is called *machine_decompose* in our development.

$\forall$ *pg*, $pg_1$, $pg_2$: *list(abs_assembly)*. $\forall$ *s*, $s''$: *state.*
*machine*($pg.pg_1.pg_2$, $pg_1.pg_2$, $s$, $s''$) $\Rightarrow$ *unique*($pg.pg_1.pg_2$) $\Rightarrow$
*self_contained*($pg_1$)$\Rightarrow$
 $\exists$ $s'$:*state.* *machine*($pg_1$, $pg_1$, $s$, $s'$) $\wedge$ *machine*($pg.pg_1.pg_2$, $pg_2$, $s'$, $s''$)
.

The proof of this statement is done by induction on the structure of the derivation of the assumption *machine(pg.pg*$_1$*.pg*$_2$*, pg*$_1$*.pg*$_2$*, s, s'').* Actually, this statement is not directly the one that is proved by induction, but a stronger one, that expresses the same statement, but for any program fragment that is a suffix of $pg_1$:

$\forall$ *pg, pg*$_1$*, pg*$_2$*, pg'*$_1$*: list(abs_ assembly).* $\forall$ *s, s'': state.*
*machine(pg.pg*$_1$*.pg*$_2$*, pg'*$_1$*.pg*$_2$*, s, s'')* $\Rightarrow$ *unique(pg.pg*$_1$*.pg*$_2$*)* $\Rightarrow$
*self_ contained(pg*$_1$*)*$\Rightarrow$ $\forall$ *pg'': list(abs_ assembly). pg*$_1$ = *pg''.pg'*$_1$
$\exists$ *s':state. machine(pg*$_1$*, pg'*$_1$*, s, s')* $\wedge$ *machine(pg.pg*$_1$*.pg*$_2$*, pg*$_2$*, s', s'')*
.

This theorem is an existential statement, which makes it harder to use in automatic procedures, as it is not adapted for backward reasoning. For this reason, it is interesting to derive the following corollary, easier to use when the tail program fragment is empty. This theorem is called *machine_ decompose_ end* in our development:

$\forall$ *pg, pg*$_1$*: list(abs_ assembly).* $\forall$ *s, s': state.*
*machine(pg.pg*$_1$*, pg*$_1$*, s, s')* $\Rightarrow$ *unique(pg.pg*$_1$*)* $\Rightarrow$ *self_ contained(pg*$_1$*)*$\Rightarrow$
*machine(pg*$_1$*, pg*$_1$*, s, s').*

The second main theorem applies when one already knows how to execute two program fragments where the first one is self contained. This theorem is called *machine_ compose* in our development:

$\forall$ *pg, pg*$_1$*, pg*$_2$*:list(abs_ assembly). unique(pg.pg*$_1$*.pg*$_2$*)* $\Rightarrow$ *self_ contained(pg*$_1$*)* $\Rightarrow$
$\forall$ *s, s', s'': state. machine(pg*$_1$*, pg*$_1$*, s, s')* $\Rightarrow$ *machine(pg.pg*$_1$*.pg*$_2$*, pg*$_2$*, s', s'')* $\Rightarrow$
*machine(pg.pg*$_1$*.pg*$_2$*, pg*$_1$*.pg*$_2$*, s, s'').*

Here again the proof is done by induction on the structure of a derivation of the first *machine* assumption, with a stronger statement that considers all possible suffixes of $pg_1$.

These theorems are the only theorems we provide at this abstract level. They will make it possible for us to change the granularity with which we reason about and observe the execution of assembly program. Self-contained program fragments play an important role because the compiler always constructs self-contained assembly sequences.

## 2.5 Connecting to the "concrete machine"

A given processor is described by instantiating the sets *Inst*, *Lab*, and *state* with concrete datatypes, providing the functions *new_ state*, *test_ branch*, and *test_ eq_ lab*, and providing translation functions from the concrete assembly language to the abstract assembly language.

In our case, we have instantiated *Inst* with *Assembly*, *Lab* with *nat*, and *state* with *exec_ state*. The function *new_ state* has been easily described using the functions *new_ stack*, *new_ register*, and *new_ mem*. On the other hand, the function *test_ branch* has been defined from scratch. We have also defined translation functions *abstract_ to_ assembly* and *assembly_ to_ abstract* that map concrete programs to abstract ones and vice-versa. The former maps abstract *label* instructions carrying a natural value to a concrete *label* instruction with the same natural value, *branch* and *plain* instruction are simply mapped to the *Inst* value they contain. The function *assembly_to_abstract* is slightly more complicated as it must select for each non-label instruction whether it will be mapped towards a *branch* or a *plain* construct. After defining these functions, we have proved several properties:

**structural properties.** There is a partial inversion property between the two translation functions:

$$\forall\ pg : Assembly.\ abstract\_to\_assembly(assembly\_to\_abstract(pg))=pg.$$

Note that the converse is false. The reason is that *assembly_to_abstract* is not surjective. For instance, it cannot produce an abstract instruction *plain(goto(n))*. In fact the following equality holds:

$$assembly\_to\_abstract(abstract\_to\_assembly(plain(goto(n)))) = branch(goto(n),n).$$

**semantic properties.** Executing an assembly program and its abstract representation are equivalent (theorems *abstract_sound* and *abstract_complete* in our proof development).

With these tools, we will be able to use the theorems established at the abstract level by lifting them to the concrete level, while still maintaining a complete mechanical check of our proof development. We have concrete level equivalents of *machine_decompose*, *machine_decompose_end*, and *machine_compose*.

# 3 Compiler and proof structure

A compiler is traditionally composed of at least two components. The first one performs a static analysis and checks that the basic disciplines are respected, mostly the type discipline. This component is referred to as a *type-checker*. The second component actually performs code production, assuming that all type constraints are respected. Sometimes, the collaboration between components is stronger, as the first component may construct tables to give information on the various symbols used in programs or modify the source program to disambiguate some constructs (for instance, when some operators are overloaded). This is not the case in our experiment, because we have abstracted variable names away and consider them as indexed variables: the index is directly used as a memory address.

## 3.1 Type-checking as an inductive property

Even though the compiler *per se* does not use the type-checker to produce the assembly code, this type-checker will play a role in our correction proof, since the correspondance between executing source and target code will be established only for well-typed programs.

We have used a type-checker that had already been produced for other studies of the same language [7]. This type-checker is obtained automatically from a natural semantics specification [34]. For this reason it takes the form of a collection of inductive properties: *check_decls* to check the declarations, that is, to check that a variable is not declared twice and that its initial value belongs to the declared type, *lookup_type* to find the type of a variable in the declarations, *check_exp* to check that an expression is well-typed and determine its type, and *check_inst* to check that an instruction is well-typed.

The specification expresses that to check types for any construct in the program, it is necessary to check types for its sub-components and that variables have to be checked against the list of declarations. For this reason, the derivation to prove that a given program is well-typed is almost isomorphic to the program itself. Proofs by induction on the structure of this derivation will somehow be equivalent to proofs by induction on the structure of the program.

If we had not been in a position to re-use an existing specification, we could have considered defining the type-checker as a total function over programs returning boolean values. However, the proof would have been made more complicated by the necessity to consider and discard the cases corresponding to erroneous programs, while these cases are systematically avoided by a proof by induction on the structure of derivations.

## 3.2 The compiler as a set of structural recursive functions

The compiler consists of five main functions. The first function is called *compile_val* and is used to cope with the fact that the two types from the source language *integer* and *boolean* are mapped to one type in the target language: *nat*. Boolean values are simply mapped to pre-determined, distinct, `nat` values (0 for *false* and 1 for *true*). The second function is called *compile_bin* and is used to handle binary operations in a systematic manner. This function takes three assembly program fragments and returns a program fragment that executes the first one, pushes the register's value on the stack, and then executes the other two program fragments.

The third function, *compile_exp*, is used to handle boolean and arithmetic expressions. Immediate values are compiled using *compile_val*, binary operators are compiled using *compile_bin*, and variables are compiled into a *load* instruction that fetches the value stored in memory at the address given by the variable index. In the following we shall note $[[E]]$ for *compile_exp(E)*. This function has been devised in a way to ensure that executing the compiled code for a given expression returns the same stack as before, even though the stack may have evolved during execution. Also, none of the functions described so far produce code containing branching statements or memory modifications.

The last two functions *compile_inst* and *compile_seq* are mutual recursive functions that respectively handle instructions and lists of instructions. These two functions take as arguments an instruction (or a list of instructions) and a natural number and return pairs of an assembly program and a natural number. The natural number taken as argument is used to index the labels that will be produced during compilation. The natural number returned in the pair represents the highest label index used in the assembly program. This is a simple scheme to ensure that all label indices will be used only once in the compiled program (the *unique* property). In the following we shall note $[[pg]]_n$ for *compile_ inst(pg,n)*, we shall also use this notation to denote the program that is the first component of the pair. The notation will be disambiguated by the context.

For instance, suppose the instruction $if(E,\ I_1,\ I_2)$ is compiled using $n$ as label index. The compiler first compiles $I_1$ using $n+2$ as label index and this returns a pair $(pg_1,\ n_1)$. The compiler then compiles $I_2$ using $n_1$ as label index and this returns a pair $(pg_2,\ n_2)$. The result of the whole compilation is the pair $(pg, n_2)$, where $pg$ is given by the following equation:

$$pg = [[E]].branch\_if\_0(n).pg_1.goto(n{+}1).label(n).pg_2.label(n{+}1).$$

The *unique* property is easily ensured: all labels used in $pg_1$ are between $n+2$ and $n_1$, all labels used in $pg_2$ are between $n_1$ and $n_2$ and all labels constructed directly by this compilation phase are between $n$ and $n+1$.

## 3.3 Dynamic semantics as an inductive property

An important specificity of this semantic description is exhibited by the rule *exec_ while_ true* that describes how a *while* construct is executed when the condition evaluates to *true*.

$\forall$ *i: INST.* $\forall$ *e: EXP.* $\forall$ *d, d′, d″: DECLS.*
*eval(d,e, true)* $\Rightarrow$ *exec(d, i, d′)* $\Rightarrow$ *exec(d′, while(e, i), d″)* $\Rightarrow$
*exec(d, while(e, i), d″).*

In most of the definition rules for the *exec* predicate, one states that it is possible to prove that an instance of a construct executes normally if some strict sub-component also executes normally. If this property was present in all the rules, then one would be sure that the “height” of the derivation to prove the execution would be smaller than the height of the executed instruction itself, and one would be able to reason on this derivation using only an induction on the size of the instruction. This rule is different because one needs to be able to execute the whole construct again, but in a different context. For this reason, the derivation to prove that an instruction executes normally may have a size that is not necessarily related to the size of the instruction. *A proof by induction on the structure of a derivation for exec cannot be replaced by a proof by structural induction on the instruction.*

This property is not specific to the language in our experiment. In fact, it is present for any language that is Turing-complete. If the derivation to prove that a program executes correctly had a size related to the size of the program, one would be able to solve the halting problem.

We have seen that the type-checker could be described as a structural recursive function as well as with the inductive definition of a predicate. In the case of dynamic semantics, this is not possible: a structural recursive function would be ensured to terminate and tell whether the program executes normally or not. That would be a solution of the halting problem. Other researchers have been confronted to this problem, especially the authors of [25, 36] who worked in a framework that only implemented predicates that could be represented using computable functions. To work around this problem, they formalized execution with a four-place predicate with the intuitive meaning: *exec(D, I, D′, n)* holds when $I$ executes normally and terminates in less than $n$ steps, transforming state $D$ in state $D'$.

## 3.4 Subject reduction properties

The type checker plays an important role in compiler proofs. In general, badly typed programs may still execute normally, both with respect to the source level dynamic semantics, or in compiled form, but they will often exhibit different behaviors. However, it is quite obvious that badly-typed programs may execute normally in compiled form, even though they cannot be executed using the source-level dynamic semantics (for example take the program `x = false * 1` that executes normally in compiled form and assigns the value 0 or `false` to `x` depending on `x`'s type). To make our development uniform, we have only considered the evaluation, compilation, and execution of well-typed programs.

The next issue is whether a program fragment that is a well-typed program with respect to some declaration block is still well-typed in another block obtained from the first one by executing some other program fragment. Intuitively, this should hold: when checking the types in an instruction, one only looks at the types in variable declarations; when evalutating an instruction, one only modifies the values in variable declarations and the "well-typed" property should still hold.

We have introduced an extra relation between declaration blocks, called *types_ compatible*, that holds for any pair of blocks that define the same variables, in the same order, and with the same types, that is, if the two blocks are equal as long as you don't look at variable values. Then we have proved a collection of theorems.

The first theorem is a subject reduction theorem: in a well-typed declaration block, if an expression has a type $t$ then its value has the same type $t$.

The second theorem expresses that the initial and final declaration blocks when executing an instruction are *types_ compatible*.

The third and fourth theorem state that if an expression or an instruction is well-typed with respect to a given declaration block, then it is also well-typed with respect to any declaration block that is type compatible.

The fifth theorem states that the resulting block of executing a well-typed instruction in an initial state given by a well-typed block is still well-typed.

All these theorems are proved by induction on the structure of the *eval* or *exec* proposition that occurs among the premises.

## 3.5 Proof organization

The proof has two distinct parts. The first part is a *completeness* proof, where one states that any behavior exhibited by the source program will also be exhibited by the compiled program. Thus, any computation expressible in the source language will be correctly implemented by the compiler program: the compiled program can do what you want. That it can do it is not enough, though. You also want to make sure that it only does what you want. This is the basis for the second part of the compiler verification, usually called a *soundness* proof. In a soundness proof, one states that any behavior exhibited by the compiled program will also be exhibited by the source program. In other words, the compiled program only does what you asked for.

### 3.5.1 A completeness theorem

The completeness theorem has the following statement:

$\forall$*i: INST.* $\forall$ *d, d′: DECLS.* $\forall$ *s: exec_state.*
*exec(d, i, d′)* $\Rightarrow$ *check_inst(d,i)* $\Rightarrow$ *check_decls(d)* $\Rightarrow$ *coherent(d, s)* $\Rightarrow$
$\forall$ *n: nat.* $\exists$ *s′: exec_result. execute(*$[[i]]_n$*, s,* $[[i]]_n$*, s′)* $\wedge$ *coherent(d′, s′)*

We have shown in section 3.3 that proofs involving the execution of programs should be organized around an induction on the structure of the execution derivation. This principle applies very well to the proof of completeness, where the derivation of the statement describing the execution of the source program has a structure that is very close to the source program's structure. For each case raised by the proof by induction, one then needs to show that the execution of the sub-components takes place in the right conditions to make it possible to use induction hypotheses: it is necessary to prove that the sub-components are well-typed in the new environment. This is done in two steps. First, one can use a technique called *inversion* and studied in [14] to derive the property that sub-components are well-typed from the assumption *check_inst(d,i)*. Then one can use subject reduction properties to show that the sub-components are also well-typed after the execution of the components that precede them in the program.

### 3.5.2 A soundness theorem

The soundness theorem has the following statement:

$\forall$ *d: DECLS.* $\forall$ *i: INST.* $\forall$ *s, s′: exec_state.* $\forall$ *n: nat.*
*check_inst(d,i)* $\Rightarrow$ *execute(*$[[i]]_n$*, s,* $[[i]]_n$*, s′)* $\Rightarrow$ *check_decls(d)* $\Rightarrow$
*coherent(d,s)* $\Rightarrow$
$\exists$ *d′: DECLS. exec(d, i, d′)* $\wedge$ *coherent(d′,s′).*

The proof of soundness is trickier, because the derivation to prove that the compiled program terminates and returns some result is not structured in a way that makes it easy to recover the structure of the source program. Instead, this derivation is a long monotonous succession of elementary steps, each corresponding to the execution of one assembly instruction. A major decision in the organization of this proof has been to avoid working by induction on the derivation of *execute(*$[[i]]_n$*, s,* $[[i]]_n$*, s′)* and to come back to an induction on the structure of the type-checking derivation. To break down the property of execution we will have the option of using the inversion technique or using the fragmentation theorems *execute_decompose* and *execute_decompose_end* derived from the abstract study described in section 2. This works for most of the cases that arise in the proof because executing the whole program involves only executing strict sub-components of the program, for which induction hypotheses are provided by the induction on the type-checking derivation.

Let us make this more precise. First, the proof uses an auxialiary theorem, with a stronger statement:

$\forall d$*: DECLS.* $\forall i$*: INST. check_inst(d,i)* $\Rightarrow \forall$ *s, s′: exec_state*
$\forall$ *n: nat. execute(*$[[i]]_n$*, s,* $[[i]]_n$*, s')* $\Rightarrow$
$\forall$ *d′:DECLS. types_compatible(d,d′)* $\Rightarrow$ *check_decls(d′)* $\Rightarrow$ *coherent(d′, s)* $\Rightarrow$
$\exists$ *d″:DECLS. exec(d′, i, d″)* $\wedge$ *coherent (d″, s′)* $\wedge$ *types_compatible(d, d″).*

This statement has been carefully designed to be easy to use in a proof by induction. First, we have been careful to distinguish between the declaration block with respect to which type checking is done ($d$) and the declaration block that is used as the initial state for the execution ($d'$). Second, we have carefully stated the properties that $d'$ must achieve. These are the properties that any declaration block would achieve after execution using $d$ as the initial block and any well-typed instruction. Third, we have been careful to state that the properties needed for $d'$ will also be achieved by $d''$. This is important since the value corresponding to $d''$ in the use of an induction hypothesis will sometimes take the role of $d'$ in the use of a second induction hypothesis.

For instance, let us suppose the compiled instruction is a sequence $I_1; I_2$. In this case, the compiler returns a program that is simply the concatenation of the compiled form of the two instructions: $[[I_1]].[[I_2]]$. The proof by induction will require that we prove the existence of a final declaration block for the execution of $[[I_1]].[[I_2]]$, but we will have as hypotheses the properties stated for $d'$ and two induction hypotheses that are full copies of the theorem statement for $I_1$ and $I_2$. Using fragmentation theorems, we will also get that *execute(*$[[I_1]]$*, s,* $[[I_1]]$*,* $s_1$*)* and *execute(*$[[I_2]]$*,* $s_1$*,* $[[I_2]]$*, s′)* hold for some state $s_1$. Using the first induction hypothesis, instantiating the bound variable named $d'$ of that hypothesis with $d'$, the bound $s$ with $s$, and the bound $s'$ with $s_1$, we will get the existence of a new declaration block, let us call it $d_1$ such that *exec(d′,* $I_1$*,* $d_1$*)* and such that $d_1$ is well-typed (thanks to subject reduction properties), coherent with $s_1$ and type-compatible with $d$. Thus, it will be possible to use the second induction hypothesis, instantiating the bound variable $d'$ this time with $d_1$, $s$ with $s_1$, $s'$ with $s'$. This will yield the existence of a new declaration block, which will be shown to have the right properties.

This structure repeats similarly for most other cases, with variations when the compiler inserts assembly instructions that do not appear in the compilation of sub-components. In this case, one needs to use inversion to show that there exists a proof of execution of the tail of the assembly program, behind the instruction.

We already know that the case of the *while* loop, when the conditional expression evaluates to *true*, will not fit in this scheme. In this case, the result of compiling *while(E,I)* takes the following form:

*label(n).*$[[E]]$*.branch_if_0(n+1).*$[[I]]$*.goto(n).label(n+1).*

In this case, we have theorems and induction hypotheses to relate the execution of $[[E]]$ to *eval(d', E, true)* and the execution of $[[I]]$ to *exec(d', I, d'')* for some $d''$, but after executing the instruction *goto(n)*, we get in a situation where the program to execute is:

$$[[E]].branch\_if\_0(n{+}1).[[I]].goto(n).label(n{+}1).$$

and we need to relate this execution to the source-level execution of *while(E, I)*. There is no induction hypothesis that is adapted to this case. The next section provides an analysis of this problem and proposes a systematic solution.

# 4 A generalized induction principle for looping programs

In this section, we describe a new method to derive an induction theorem from some inductive definitions. The new induction theorem, although equivalent to the basic one provided by the technique described in [27] will make some proofs much easier. In particular, it is better adapted to our problem.

## 4.1 Basic induction principles and couse-of-value induction

When performing proofs on natural numbers, mathematicians use (often without making the difference) two methods of proof by induction. The first method, the one that is presented in foundational books uses the simpler form of induction expressed in the following principle:

$$\begin{array}{l}\forall P : nat \rightarrow Prop. \\ \quad P(0) \Rightarrow \\ \quad (\forall n : nat.\ P(n) \Rightarrow P(n{+}1)) \Rightarrow \\ \qquad \forall m : nat.P(m).\end{array}$$

In everyday practice, this means that to prove $\forall$ *n:nat. P(n)*, one first needs to prove *P(0)* and then prove *P(n)* for any $n$ greater than *0*, but with the possibility to use $P(n-1)$.

The second method is more often used in mathematical practice and uses the order $<$ on naturals. It uses the form of induction expressed in the following theorem:

$$\begin{array}{l}\forall P : nat \rightarrow Prop. \\ \quad (\forall n : nat.\ (\forall m : nat.\ m < n \Rightarrow P(m)) \Rightarrow P(n)) \Rightarrow \forall n : nat.\ P(n).\end{array}$$

In daily practice, this means that to prove $\forall$ *n:nat. P(n)*, one does not need to prove *P(0)* anymore, and it is enough to prove *P(n)* for any $n$ with the possibility to use *P(m)* for any $m$ that is smaller than $n$. Obviously, the proof that one needs to perform using the second induction principle is simpler, as we can use more hypotheses.

Fortunately, the second induction principle can be proved using the first one. Thus, using this induction principle does not change the validity of our developments. Let us look at the proof of this result.

Let us suppose we have a property $P$ on natural numbers and a property $Q$ that is defined by the following equivalence:

$Q(n) \Leftrightarrow (\forall m : nat.\ m < n \Rightarrow P(m))$.

Let us suppose that $P$ verifies the following property, which corresponds to the induction hypothesis of the second method of induction:

$$\forall n : nat.\ Q(n) \Rightarrow P(n). \qquad (1)$$

Now let us prove that $P(n)$ holds for any $n$. To do this, we prove by an induction on $n$ that $Q(n) \wedge P(n)$ always holds. The first step is to prove that $Q(0) \wedge P(0)$ holds. That $Q(0)$ holds is trivially true, as there is no natural number that is strictly smaller than 0. Now, since $Q(0)$ holds, we also have $P(0)$ by using *(1)*.

The second step is to prove $Q(n+1) \wedge P(n+1)$ knowing $Q(n) \wedge P(n)$. By definition of $Q$, having $Q(n)$ and $P(n)$ we get directly $Q(n+1)$. Using *(1)* we get $P(n+1)$ and the proof is complete.

The order $<$ plays the role of an "iterator" in this description. The expression $m < n$ means that $n$ can be obtained from $m$ by iterating a certain number of times the constructor $S$ of the inductive definition.

## 4.2 Auxiliary "large step" semantics for the target code

All this carries over well to the inductive definition of assembly programs execution, except that we need to exhibit a relation that will play the role of the order $<$. Intuitively, this relation will relate two pairs of state and program $(s, p)$ and $(s', p')$ if the execution of $p$ in initial state $s$ can be obtained by iterating an certain number of times the constructor *run_one_step* from $(s', p')$ (in other words, there is a large step of execution between $(s, p)$ and $(s', p')$). We shall call this relation *execute'* and it will be defined by an inductive definition using the following two constructors (called *execute'_one* and *execute'_step* in our development).

$$\begin{array}{l}
\forall pg, pg', pg'' : list(Assembly).\ \forall i : Assembly.\ \forall m, s, s' : list(nat).\ \forall r, r' : nat.\\
new_pc(pg, pg', r, i) = pg'' \Rightarrow new_stack(r, s, i) = s' \Rightarrow\\
new_register(r, m, s, i) = r' \Rightarrow\\
\quad execute'(pg, (m, s, r), i.pg', (new_mem(r, m, i), s', r'), pg'').
\end{array}$$

and

$$\begin{array}{l}
\forall pg, pg', pg'', pg''' : list(Assembly).\ \forall i : Assembly.\ \forall m, s, s' : list(nat).\\
\forall r, r' : nat.\ \forall result : exec_state\\
new_pc(pg, pg', r, i) = pg'' \Rightarrow new_stack(r, s, i) = s' \Rightarrow\\
new_register(r, m, s, i) = r' \Rightarrow\\
execute'(pg, (new_mem(r, m, i), s', r'), pg'', result, pg''') \Rightarrow\\
\quad execute'(pg, (m, s, r), i.pg', result, pg''')
\end{array}$$

In fact these two constructor statements are obtained by systematically transforming the statement of *execute_step*.

With this relation *execute*$'$, we can now state the induction theorem that will play the same role as course-of-value induction for assembly induction (this theorem is named *new_execute_ind* in our proof development).

$\forall$ *pg: list(Assembly).* $\forall$ *P.* $\forall$ *s, s*$''$.
($\forall$*pg*$''$*: list(Assembly).* $\forall$*s*$'$. *execute*$'$*(pg, s, pg*$'$*, s*$'$*, pg*$''$*)* $\Rightarrow$
*execute(pg, s*$'$*, pg*$''$*, s*$''$*)* $\Rightarrow$ *P(s*$'$*, pg*$''$*, s*$''$*))* $\Rightarrow$
$\forall s_0, s_1.\ \forall pg_0 : list(Assembly).\ P(s_0, pg_0, s_1)$.

The proof of this theorem is easily done using the method exhibited in our study of the theorem for course-of-value induction on natural numbers.

In our development, we also have two theorems that relate *execute* and *execute*$'$, called *execute*$'$*_execute* and *execute_execute*$'$, with respective statements:

$\forall$ *pg, pg*$'$*: list(Assembly).* $\forall$ *s, s*$'$*: exec_state. execute*$'$*(pg, s, pg*$'$*, s', [])* $\Rightarrow$
*execute(pg, s, pg*$'$*, s*$'$*),*

*and*

$\forall$ *pg: list(Assembly).* $\forall$ *s, s*$'$*: exec_state. execute(pg, s, pg, s*$'$*)* $\Rightarrow$
$\forall$ *pg*$'$*, pg*$''$*, pg*$'''$*: list(Assembly). self_contained(pg)* $\Rightarrow$ *unique(pg)* $\Rightarrow$ *pg* $\neq$ [] $\Rightarrow$
*execute*$'$*(pg*$'$*.pg.pg*$''$*, s*$'$*, pg*$''$*, s*$''$*, pg*$'''$*)* $\Rightarrow$ *execute*$'$*(pg*$'$*.pg.pg*$''$*, s*$'$*, pg.pg*$''$*, s*$''$*, pg*$'''$*).*

Note that the second theorem, *execute_execute*$'$ is very similar to a fragmentation theorem. We did not need another fragmentation theorem, because *execute*$'$ is used only in the proof of soundness. However, we also established a trivial transitivity theorem called *execute*$'$*_trans*.

### 4.3 Application to the "while" loop

We use the "course-of-value" induction theorem during the proof of soundness for the case where the assembly program we execute is the result of compiling a *while* loop: *while(E,I)*. We recall that the compiled program fragment has the following form:

*label(n).*$[[E]]$*.branch_if_0(n+1).*$[[I]]_{n+2}$*.goto(n).label(n+1),*

that we work under the following assumptions (among others),

- *execute(*$[[while(E,I)]]_n$*, s,* $[[while(E,I)]]_n$*, s*$'$*),*
- *execute(*$[[E]]$*, s,* $[[E]]$*, (m, stk, 1)),*
- *check_inst(d, i),*
- *types_compatible(d, d*$'$*),*
- *coherent(d*$'$*, s),*

and that we want to find a declaration block $d''$ such that:

1. $exec(d', while(E, I), d'')$ and
2. $coherent(d'', s')$.

The proof of proposition 1 can be obtained using the constructor named *exec_while_true* from *exec*'s inductive definition. This requires that we show:

(i) $eval(d', E, true)$,

(ii) $exec(d', I, d_1)$ for some $d_1$,

(iii) $exec(d_1, while(E, I), d'')$.

Facts (i) and (ii) are easily obtained from the soundness property for the expression compiler and an induction on the derivation for *check_inst(d,while(E, I))*. We shall now see how we can use our general induction theorem to get fact (iii).

Using inversion techniques, we know that the evaluation proceeds by evaluating the whole program fragment without the label, with the same context, initial state and final state. Let us call this fragment $pg'$:

$$pg' = [[E]].\text{branch\_if\_0(n+1)}.[[I]]_{n+2}.\text{goto(n).label(n+1)}.$$

At this point, we use the course-of-value induction theorem to express that $pg''$ executes like *while(E, I)* in the corresponding initial state. More formally, we use *new_execute_ind* instantiating $P$ to have the following induction hypothesis:

$\forall pg''$: *list(Assembly)*. $\forall s'$: *exec_state*.
$execute'([[while(E,I)]]_n, s, pg', s', pg'') \Rightarrow$
$execute([[while(E,I)]]_n, s', pg'', s'') \Rightarrow$
$pg'' = pg' \Rightarrow s''=(m',k',v) \Rightarrow$
$\forall\ d_i$: *DECLS*. $types\_compatible(d, d_i) \Rightarrow check\_decls(d_i) \Rightarrow coherent(d_i,s') \Rightarrow$
$\exists\ d'$: *DECLS*. $exec(d_i, while(E,I), d') \wedge coherent(d',m') \wedge$
$\quad check\_decls(d') \wedge types\_compatible(d, d')$.

Continuing the proof, we show that the execution proceeds by first executing $[[E]]$. A soundness theorem for the expression compiler gives us (i). The conditional branching statement executes and does not branch. Then the execution proceeds by executing $[[I]]_{n+2}$. Here, we have an induction hypothesis (that comes from the proof by induction on the structure of the derivation to type-check the source program), that states that this execution correctly models the execution of $I$ in the source-level semantics, and gives us a result declaration block that has the right properties with respect to the machine state after executing $[[I]]_{n+2}$: we have $exec(d, I, d_1)$ and $execute(s, pg', s_1, goto(n)\ldots)$ for some block $\mathrm{d}_1$ and some state $\mathrm{s}_1$ such that $coherent(d_1, s_1)$ holds. Then the execution proceeds by executing the instruction *goto(n)* and we come to a situation where the program to execute is the whole program without the first label: this program has the right form to apply the induction hypothesis that has been provided by the course-of-value induction, instantiating $s'$ with $\mathrm{s}_1$ and $d_i$ with $d_1$. This gives the value for $d''$ that we need, and the proof that $exec(d_1, while(E,I), d'')$ and $coherent(d'', s')$ hold.

# 5 Conclusions

## 5.1 Time to develop, time to re-check

It took approximately three months for the author to develop the proofs presented in this paper. Some of the data was already present, since approximately the same language specification had been used for the experiment presented in [7]. The result is a collection of proof script files totalling 8,000 lines. The CtCoq user-interface, a tool to help develop and maintain large proofs in Coq was instrumental in this endeavour, especially with its *drag-and-drop* [5] and *script-management* [8] facilities. Re-running these proof scripts through the Coq system, version 6.2.1, takes approximately a quarter of an hour on a reaonnably powerful machine (Pentium, 333 MHz, 128 MBytes). The whole proof development can be found from the author's web page at INRIA Sophia Antipolis.

The work on an abstract assembly compiler to obtain the fragmentation theorems can be detached from the rest of the proof development. It represents 1240 lines of proof script.

A simple extraction command can be called in the proof system to construct a Caml file containing only the compiler. The Caml file that one obtains is 150 lines long and corresponds faithfully to the compiler that one would normally write, except that natural numbers are represented in Peano form: natural numbers are a recusively defined type, with `O : nat` and `S: nat -> nat` as constructors. This compiler only corresponds to a function that takes abstract syntax trees representing source programs and a natural number as input and produces an abstract syntax tree that represents a list of assembly instructions as output.

It is also possible to extract the functions that perform the action of each assembly instruction. However, it is not possible to extract an interpreter from the dynamics semantics specification or the abstract machine description.

## 5.2 Motivation for this work

This experiment started as a complement to a course on natural semantics and its use to develop programming environments and certified programming tools [6]. Since the very beginning of this work, the objective was to show that a specification driven programming environment like Centaur [9] and a programmation oriented proof system like Coq could be united to form a powerful system to produce effective, reliable, and user-friendly programming tools. Early work by Despeyroux [15] had shown that certifying the tools running in Centaur was possible, and work by Terrasse [34] had bridged the gap between the executable semantics of Centaur and the logical framework of Coq, thus permitting experiments like the one described in [7]. However, the approach until then was to certify program manipulation tools that were described only in natural semantics, which was tantamount to proving programs that can only be executed in Prolog. This work is complementary, since the compiler is not described as a set of logic programming clauses (the executable *alter ego* of natural semantics inference rules) but as a purely functional program.

Looking around, it appeared that truly specified and machine-checked compilers were very rare, with the notable exception of the Piton-Gypsy effort done by Boyer, Moore, and their team [25, 36] using the Nqthm theorem prover [10]. Even so, the logic of Nqthm is not adapted to representing the natural semantic specifications that serve as a foundation for any programming language study within the Centaur framework.

Also, this paper really advocates the use of natural semantics to describe the semantics of programming languages, as opposed to denotational semantics. We have found that proofs based on natural semantics require less mathematical background than those based on denotational semantics. The amount of mathematical background required to perform the proof actually matters a lot in our case, since all the mathematical facts corresponding to this mathematical background would have to be entered in the proof engine. It is characteristic that no totally mechanically verified proof of a compiler has been published, where the semantics of languages were given using denotational semantics. This remark has to be tempered: when compared to the compiler verification performed in [18], the language we have addressed is ludicrously small. Still we believe that the amount of mathematics needed to verify a compiler for a stronger language continuously augments with the complexity of the language. Another weakness of denotational semantics, pointed out in [18] is that full abstractness is important to achieve in a compiler proof: to avoid having to prove properties of mathematical objects that do not represent any valid program. Fully abstract models are hard to obtain.

## 5.3 Remarks on the Coq proof assistant

So far, the Coq tool has proven well-adapted to proving compilers of this form. We have only suffered from a weakness of the system that prevents defining mutually recursive functions, when these function are applied to terms that do not belong to types that have been defined as mutually recursive. In our case, such types arise, because the *sequence* constructor takes one argument that is a list of instructions, and we would have liked to use a polymorphic type of list there, instead of defining a special one mutually recursively with the type of instructions. The extreme simplicity of the language we addressed may be misleading. In particular, a more elaborate compiler would contain an optimizer, which may not necessarily be defined as a structural recursive function, but as some other form of total function. Coq does accept more general functions, but they are less well integrated in the simplification routines and the proof task may become more tedious when such functions are used. However, other work in Coq has proved that sophisticated algorithms using general functions could be handled very well.

## 5.4 Possible evolution

A tempting direction for this work is to increase gradually the power of the source language, trying at each step to re-use the proof development done for the previous phase. With this approach in mind, we would like to progress on two seemingly independent fronts. The first deals with datatype management, adding progressively arrays, structured and recursive type definition, and memory management. The second front deals with control structures: procedure and function calls, exceptions, input and output. Some aspects of programming language may lie at the intersection of the two domains, like pattern matching and object orientation. For the target language, it will be important to relate to formal description of assembly language, like the descriptions of the Java Virtual Machine currently under development [26, 32].

If this progressive approach is to be taken, then a lot of automation has to be added into the proof process. A first area where automation seems easy to achieve is around the proof of *unique* and *self-contained* properties for compiler results. It should be easy to prove automatically that the assembly program fragment produced for any new construct has these properties, based on the hypothesis that each sub-instruction of the construct already has these properties. Thus, adding a new construct in the source language and a new pattern-matching rule in the compiler, these two properties should be maintained automatically. Another repetitive task during proof is the succession of application of inversion techniques to reconstruct the main structure of derivations from execution hypotheses. We believe that this task could be automated. A second area for possible automation is the derivation of the course-of-value induction theorem from the initial inductive definition. At first sight, it seems that the method we have described in this paper will apply only in restricted cases, for instance when the inductive definition has a strong "tail-recursive" flavor. This needs to be investigated.

Thus, the evolution of this work should not only lead to a more realistic verified compiler but also to a more automatic tool to help producing this verified compiler.

# References


[1] Isabelle Attali, Denis Caromel, and Sidi Ould Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(5), November 1996.

[2] Isabelle Attali, Denis Caromel, Sidi Ould Ehmety, and Sylvain Lippi. Semantics-based visualization of r parallel object-oriented programming. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA'96)*, number 31(10) in Sigplan Notices. ACM Press, October 1996.

[3] Janet Bertot and Yves Bertot. CtCoq: A system presentation. In *Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 231–234. Springer-Verlag, July 1996.

[4] Yves Bertot. Implementation of an Interpreter for a Parallel Language in Centaur. In *European Symposium On Programming*. Springer Verlag, 1990. LNCS 432.

[5] Yves Bertot. Direct manipulation of algebraic formulae in interactive proof systems. In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, September 1997.

[6] Yves Bertot. Programming language specifications and environments. Rapport technique RT0212, INRIA, 1997.

[7] Yves Bertot and Ranan Fraer. Reasoning with Executable Specifications. In *TAPSOFT'95*, volume 915, pages 531–545, 1995. Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95).

[8] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. To appear in Journal of Symbolic Computation.

[9] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).

[10] Robert S. Boyer and J Strother Moore. A theorem prover for a computational logic. In *10th Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science. Springer-Verlag, July 1990.

[11] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical report, University of Cambridge, 1992.

[12] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986.

[13] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[14] Christina Cornes and Delphine Terrasse. Automatizing inversion of inductive predicates in coq. In *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[15] Joëlle Despeyroux. Proof of translation in natural semantics. In *Symposium on Logic in Computer Science*, pages 193–205. IEEE Computer Society, June 1986.

[16] Thierry Despeyroux. Executable Specifications of Static Semantics. In *International Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[17] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.

[18] Joshua D. Guttman and Mitchell Wand (eds.). Special issue on VLISP. *Lisp and Symbolic Computation*, 8(1/2), 1995.

[19] Gilles Kahn. Natural Semantics. In K. Fuchi and Maurice Nivat, editors, *Programming of Future Generation Computers*. North-Holland, 1988. (also appears as INRIA Report no. 601).

[20] The LEGO World Wide Web page. url `http://www.dcs.ed.ac.uk/home/lego`.

[21] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer-Verlag, 1994.

[22] Per Martin-Löf. An intuitinistic theory of types. Technical report, University of stockholm, 1972.

[23] John McCarthy and James Painter. correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science 1*, volume 19 of *Proceedings of Symoposia in Applied Mathematics*, 1967.

[24] Thomas F. Melham. A package for inductive relation definitions in hol. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving system and its Applications*, pages 350–357. IEEE Computer Society Press, 1992.

[25] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[26] David von Oheimb and Tobias Nipkow. Machine checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998. To appear.

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

[28] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7:175–204, March 1997.

[29] Gordon Plotkin. Structural operational semantics. lecture notes DAIMI FN-19, Aarhus University, 1981. (reprinted 1991).

[30] Wolfgang Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[31] Stanford Verifier Group. Stanford pascal verifier user manual. Technical Report 11, Stanford Verification Group, 1979.

[32] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 149–160. ACM Press, January 1998.

[33] M. E. Szabo. *The Collected papers of Gerhard Gentzen.* North-Holland, 1969.

[34] D. Terrasse. Encoding natural semantics in Coq. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95*, Springer-Verlag LNCS, July 1995.

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

[36] William D. Young. A mechanical verified code generator. *Journal of Automated Reasoning*, 5:493–518, 1989.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399