Theorem proving support in programming language semantics

Yves Bertot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Theorem proving support in programming language semantics

Yves Bertot

N° 6242

July 2007

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6242--FR+ENG

Theorem proving support in programming language semantics

Yves Bertot

Thème SYM — Systèmes symboliques
Projet Marelle

Rapport de recherche n° 6242 — July 2007 — 23 pages

Abstract: We describe several views of the semantics of a simple programming language as formal documents in the calculus of inductive constructions that can be verified by the Coq proof system. Covered aspects are natural semantics, denotational semantics, axiomatic semantics, and abstract interpretation. Descriptions as recursive functions are also provided whenever suitable, thus yielding a a verification condition generator and a static analyser that can be run inside the theorem prover for use in reflective proofs. Extraction of an interpreter from the denotational semantics is also described. All different aspects are formally proved sound with respect to the natural semantics specification.

Key-words: Coq, natural semantics, structural operational semantics, denotational semantics, axiomatic semantics, abstract interpretation, formal verification, calculus of inductive constructions, proof by reflection, program extraction

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Sémantique des langages de programmation avec le support d'un outil de preuve

Résumé : Nous décrivons plusieurs points de vue sur la sémantique d'un petit langage de programmation, vus comme des documents dans le calcul des constructions inductives qui peuvent être vérifiés par le système Coq. Les aspects couverts sont la sémantique naturelle, la sémantique dénotationnelle, la sémantique axiomatique, et l'interprétation abstraite. Des descriptions sous formes de fonctions récursives sont fournies quand c'est adapté, et on obtient ainsi un générateur de conditions de vérification et un analyseur statique qui peuvent être utilisés dans des preuves par réflexion. L'extraction d'un interprète à partir de la sémantique dénotationnelle est également décrite. Des preuves formelles assurant la correction des différents aspects vis-à-vis de la sémantique naturelle sont également abordes.

Mots-clés : Coq, sémantique naturelle, sémantique dénotationnelle, sémantique axiomatique, interprétation abstraite, vérification formelle, calcul des constructions inductives, preuve par réflexion, extraction de programmes

This paper is dedicated to the memory of Gilles Kahn, my thesis advisor, my mentor, my friend.

1 introduction

Nipkow demonstrated in [24] that theorem provers could be used to formalize many aspects of programming language semantics. In this paper, we want to push the experiment further to show that this formalization effort also has a practical outcome, in that it makes it possible to integrate programming tools inside theorem provers in an uniform way. We re-visit the study of operational, denotational semantics, axiomatic semantics, and weakest pre-condiction calculus as already studied by Nipkow and we add a small example of a static analysis tool based on abstract interpretation.

To integrate the programming tools inside the theorem prover we rely on the possibility to execute the algorithms after they have been formally described inside the theorem prover and to use theorems about these algorithms to assert properties of the algorithm's input, a technique known as *reflection* [2, 9]. Actually, we also implemented a parser, so that the theorem prover can be used as a playground to experiment on sample programs. We performed this experiment using the Coq system [16, 6]. The tools that are formally described can also be "extracted" outside the proof environment, so that they become stand alone programs, thanks to the extracting capabilities provided in [22].

The desire to use computers to verify proofs about programming language semantics was probably one of the main incentives for the design of modern interactive theorem provers. The LCF system was a pioneer in this direction. The theory of programming languages was so grounded in basic mathematics that a tool like LCF was quickly recognized as a tool in which mathematical reasoning can also be simulated and proofs can be verified by decomposing them in sound basic logical steps. LCF started a large family of theorem proving tools, among which HOL [18] and Isabelle [25] have achieved an outstanding international recognition. Nipkow's experiments were conducted using Isabelle.

In the family of theorem proving tools, there are two large sub-families: there are the direct descendants of the LCF system [19], which rely on simply-typed λ-calculus and the axioms of higher-order logic to provide foundations for a large portion of mathematics; on the other hand, there are systems descending from de Bruijn's Automath system and Martin-Löf's theory of types, where propositions are directly represented as types, "non-simple" types, namely dependent types, can be used to represent quantified statements, and typed functions are directly used to represent proofs (the statement they prove being their type). In systems of the LCF family, typed λ-terms are used in the representation of logical statements and proofs are objects of another nature. In systems of the latter family, usually called type theory-based theorem proving tools, typed λ-terms are used both in the representation of logical statements and in the representation of proofs. Well-known members of the type theory based family of theorem proving tools are Nuprl [11], Agda [12], and Coq.

The fact that typed λ-terms are used both to represent logical statements and proofs in type theory-based theorem proving tool has the consequence that computation in the typed λ-calculus plays a central role in type-theory based theorem proving tools, because verifying that a theorem is applied to an argument of the right form may require an arbitrary large computation in these systems. By contrast, computation plays only a secondary role in LCF-style theorem proving tools and facilities to execute programs efficiently inside the theorem prover to support proofs was only added recently [3].

With structural operational semantics and natural semantics, Gordon Plotkin and Gilles Kahn provided systematic approaches to describing programming languages relying mostly on the basic concepts of inductive types and inductive propositions. Execution states are represented as environments, in other words lists of pairs binding a variable name and a value. Programs themselves can also be represented as an inductive data-type, following the tradition of *abstract syntax* trees, a streamlined form of parsing trees. Execution of instructions can then be described as inductive propositions, where executing an instruction is described as a ternary relation between an input environment, an instruction, and an output value. The execution of each program construct is described by composing "smaller" executions of this construct or its sub-components. We will show that descriptions of execution can also be represented using functions inside the theorem prover and we will prove that these functions are consistent with the initial semantics, in effect producing certified interpreters for the studied language.

Another approach to describing the behavior of programs is to express that a program links properties of inputs with properties of outputs. In other words, one provide a logical system to describe under which condition on a program's input a given condition on the program's output can be guaranteed (as long as the program terminates). This style of description is known as *axiomatic semantics* and was proposed by Hoare [20]. Here again, we can use an inductive type to represent a basic language of properties of input and output of programs. We will show that axiomatic semantics can easily be described using inductive properties and recursive functions and again we will show that the new reasoning rules are consistent with the initial operational semantics. Axiomatic semantics also support an algorithmic presentation, known as a *verification condition generator* for the *weakest precondition calculus* as advocated by Dijkstra [15]. Again, we provide an implementation of this generator and a proof that it is correct. Thanks to the reflection approach, this generator can be used inside the theorem prover to establish properties of sample programs.

The next style of semantic description for programming language that we will study will be the style known as *denotational semantics* or *domain theory*, actually the style that motivated the first implementation of the LCF system. Here, the semantics of the instructions is described as a collection of partial functions from a type of inputs to a type of outputs. The kind of functions that are commonly used in type-theory based theorem proving tools are not directly suited for this approach, for fundamental reasons. We will show what axioms of classical logical can be used to provide a simple encoding of the partial functions we need. However, using these axioms precludes computing inside the theorem prover, so that the function we obtain are executable only after extraction outside the

theorem prover. This approach can still be used to derive an interpreter, a tool to execute programs, with a guarantee that the interpreter respects the reference operational semantics.

The last category of semantic approaches to programming languages that we want to address in this paper is an approach to the static analysis of programs known as *abstract interpretation*. While other approaches aim at giving a completely precise understanding of what happens in programs, abstract interpretation focusses on providing abstract views of execution. The goal is to hide enough details so that the information that is obtained from the analysis is easier to manage and more importantly the computations to perform the analysis can be performed by a program that is guaranteed to terminate.

1.1 Related work

The main reference we used on programming language semantics is Winskel's text book [29].

Many publications have been provided to show that these various aspects of programming language could be handled in theorem provers. Our first example is [7] where we described the correctness of a program transformation tool with respect to the language's operational semantics. This work was performed in the context of the Centaur system [8] where semantic descriptions could be executed with the help of a prolog interpreter or reasoned about using a translation to the Coq theorem prover [26]. The most impressive experiment is described in [24], who approximately formalizes the first 100 pages of Winskel's book, thus including a few more proofs around the relations between operational semantics, axiomatic semantics, and denotational semantics than we describe here. The difference between our work and Nipkow's is that we rely more on reflection and make a few different choices, like the choice to provide a minimal syntax for assertions, while Nipkow directly uses meta-level logical formulas and thus avoid the need to describe substitution. On the other hand, our choice of an abstract syntax for assertions makes it possible to integrate our verification generator with a parser, thus providing a more user-friendly approach to annotated programs.

The work on denotational semantics is a transposition and a reduction of the work on domain theory that could already be described formally in the framework of *logic of computable functions*, in Isabelle [23].

The study of interactions between abstract interpretation and theorem provers is the object of more recent work. Intermediate approaches use abstract interpreters to generated proofs of correctness of programs in axiomatics semantics as in [10]. Pichardie [14] actually goes all the way to formally describing a general framework for abstract interpretation and then instantiating it for specific domains to obtain static analysis tools. Our work is similar except that Pichardie's work is based on transition semantics, this imposes that recursion is based on well-founded recursion, a feature that makes it ill-suited for use in reflection.

Application domains for theorem prover-aware formal semantics of programming languages abound. Nipkow and his team [28], Jacobs and his team, [27], and Barthe and his team [1, 5] showed the benefits there could be in describing the Java programming language and the Java virtual machine, to verify soundness properties of the byte-code verifier and apply this the guarantees of the security that the Java language and its Smartcard-aware

offspring, JavaCard. More recent work by Leroy and his team show that this work can be extended to the formalization of efficient compilers.

2 Concrete and abstract syntax

We consider a *while loop* programing language with simple arithmetic expressions: it is the Imp language of [24] without the conditional instruction. The language has been trimmed to a bare skeleton, but still retains the property of being Turing complete. We will use ρ as meta-variables for variable declarations (we will also often use the word *environment*), e for expressions, b for boolean expressions, and i for instructions. We use an infinite denumerable set of variable names whose elements are written $x, y, x_1, \ldots$ and we use n, n_1, n' to represent integers. The syntactic categories are defined as follows:

$$\rho ::= (x, n) \cdot \rho | \emptyset \qquad e ::= n \mid x \mid e{+}e \qquad b ::= e <e$$

$$i ::= \mathsf{skip} \mid x{:=}e \mid i;i \mid \mathsf{while}\ b\ \mathsf{do}\ i\ \mathsf{done}$$

The intended meaning of most of these constructs should be obvious. The only suprising element may be the skip instruction: this is an *empty* program, which does nothing.

In the theorem prover, we use inductive types to describe these syntactic categories. The convention that numbers are expressions needs to be modified: there is a constructor anum in the type of arithmetic expression aexpr that maps a number to the corresponding expression. Similarly, variable names are transformed into arithmetic expressions and assignments just use variable names as first components.

```
Inductive aexpr : Type := avar (s : string) | anum (n : Z)  | aplus (a1 a2 :aexpr).

Inductive bexpr : Type := blt (a1 a2 : aexpr).

Inductive instr : Type :=
  assign (s: string)(e:aexpr) | sequence (i1 i2:instr) | while (b:bexpr)(i:instr) | skip.
```

3 Operational semantics

3.1 Evaluation and environment update

3.1.1 Inference rules

We will describe the evaluation of expressions using judgments of the form $\rho \vdash e \rightarrow v$ or $\rho \vdash b \rightarrow v$ (with a straight arrow). These judgments should be read as *in environment ρ, the arithmetic expression e (resp. the expression b) has the value v*. The value v is an integer or a boolean value depending on the kind of expression being evaluated. The rules describing

evaluation are as follows:

$$\frac{}{\rho \vdash n \to n} \qquad \frac{}{(x,n)\cdot\rho \vdash x \to n}$$

$$\frac{\rho \vdash x \to n \quad x \neq y}{(y,n')\cdot\rho \vdash x \to n} \qquad \frac{\rho \vdash e_1 \to n_1 \quad \rho \vdash e_2 \to n_2}{\rho \vdash e_1{+}e_2 \to n_1 + n_2}$$

$$\frac{\rho \vdash e_1 \to n_1 \quad \rho \vdash e_2 \to n_2 \quad n_1 < n_2}{\rho \vdash e_1{<}e_2 \to \mathsf{true}} \qquad \frac{\rho \vdash e_1 \to n_1 \quad \rho \vdash e_2 \to n_2 \quad n_2 \leq n_1}{\rho \vdash e_1{<}e_2 \to \mathsf{false}}$$

During the execution of instructions, we will regularly need describing the modification of an environment, so that the value associated to a variable is modified. We use judgments of the form $\rho \vdash x, n \mapsto \rho'$, which should be read as *x has a value in ρ and ρ' and the value for x in ρ' is n; every other variable that has a value in ρ has the same value in ρ'.* This is simply described using two inference rules, in the same spirit as rules to evaluate variables.

3.1.2 Theorem prover encoding

Judgments of the form $\cdot \vdash \cdot \to \cdot$ are represented by three-argument inductive predicates named aeval and beval. We need to have two predicates to account for the fact that the same judgment is actually used to describe the evaluations of expressions of two different types. The encoding of premises is quite straight forward using nested implications, and we add universal quantifications for every variable that occurs in the inference rules. All inference rules for a given judgment are grouped in a single inductive definition. This makes it possible to express that the meaning of the judgment $\cdot \vdash \cdot \to \cdot$ is expressed by these inferences *and only these inferences rules.*

Environments are encoded as lists of pairs of a string and an integer, so that the environment $\emptyset$ is encoded as nil and the environment $(x,n)\cdot\rho$ is (x,n)::r.

```
Definition env := list(string*Z).

Inductive aeval : env → aexpr → Z → Prop :=
  ae_int : ∀ r n, aeval r (anum n) n
| ae_var1 : ∀ r x n, aeval ((x,n)::r) (avar x) n
| ae_var2 : ∀ r x y v v' , x ≠ y → aeval r (avar x) v → aeval ((y,v')::r) (avar x) v
| ae_plus : ∀ r e1 e2 v1 v2, aeval r e1 v1 → aeval r e2 v2 →
            aeval r (aplus e1 e2) (v1 + v2).

Inductive beval : env → bexpr → bool → Prop :=
| be_lt1 : ∀ r e1 e2 v1 v2, aeval r e1 v1 → aeval r e2 v2 → v1 < v2 →
          beval r (blt e1 e2) true
| be_lt2 : ∀ r e1 e2 v1 v2, aeval r e1 v1 → aeval r e2 v2 → v2 ≤ v1 →
          beval r (blt e1 e2) false.
```

the four place judgment $\cdot \vdash \cdot, \cdot \mapsto \cdot$ is also encoded as an inductive definition for a predicate named update.

Induction principles are automatically generated for these declarations of inductive predicates. These induction principles are instrumental for the proofs presented later in the paper.

3.2 Functional encoding

The judgment $\rho \vdash e \to n$ actually describes a partial function: for given ρ and e, there is at most one value n such that $\rho \vdash e \to n$ holds. We describe this function in two steps with lookup and af, which return values in option Z. When computing additions, we need to compose total functions with partial functions. For this, we define a bind function that takes care of undefined values in intermediate results. The pre-defined function string_dec is used to compare two strings.

```
Fixpoint lookup (r:env)(s:string){struct r} : option Z :=
match r with
 nil ⇒ None  |  (a,b)::tl ⇒ if (string_dec a s) then Some b else lookup tl s
end.

Definition bind (A B:Type)(v:option A)(f:A→option B) : option B :=
  match v with Some x ⇒ f x | None ⇒ None end.

Fixpoint af (r:env)(a:aexpr) : option Z :=
match a with
  avar index ⇒ lookup r index          | anum n ⇒ Some n
| aplus e1 e2 ⇒ bind (af r e1) (fun v1 ⇒ bind (af r e2) (fun v2 ⇒ Some (v1+v2)))
end.
```

We can define functions bf to evaluate boolean expressions and uf to compute updated environments in a similar way.

We use two functions to describe the evaluation of arithmetic expressions, because evaluating variables requires a recursion where the environment decreases at each recursive call (the expression staying fixed), while the evaluation of additions requires a recursion where the expression decreases at each recursive call (the environment staying fixed). The Fixpoint construct imposes that the two kinds of recursion should be separated.

With aeval and af, we have two encodings of the same concept. We need to show that these encoding are equivalent, this is done with the following lemmas.

```
Lemma lookup_aeval : ∀ r s v, lookup r s = Some v → aeval r (avar s) v.

Lemma af_eval : ∀ r e v, af r e = Some v → aeval r e v.

Lemma aeval_f : ∀ r e n, aeval r e n → af r e = Some n.
```

The proof of the first lemma is done by induction on the structure of r, the proof of the second lemma is done by induction on e, while the proof of the third lemma is done by induction on the structure of the proof for aeval (using the induction principle, which is generated when the inductive predicate is declared). Using simple proof commands, each of these proofs is less than ten lines long. We also have similar correctness proofs for bf and uf.

3.3 Natural semantics

With *natural semantics* [21], Gilles Kahn proposed that one should rely on judgments expressing the execution of program fragments until they terminate. The same style was also called *big-step* semantics. The main advantage of this description style is that it supports very concise descriptions for sequential languages. For our little language with four instructions, we only need five inference rules.

We rely on judgments of the form $\rho \vdash i \rightsquigarrow \rho'$ (with a twisted arrow). These judgments should be read as *executing i from the initial environment ρ terminates and yields the new environment ρ'.*

$$\frac{}{\rho \vdash \mathsf{skip} \rightsquigarrow \rho} \qquad \frac{\rho \vdash e \to n \qquad \rho \vdash x, n \mapsto \rho'}{\rho \vdash x{:=}e \rightsquigarrow \rho'}$$

$$\frac{\rho \vdash i_1 \rightsquigarrow \rho' \qquad \rho' \vdash i_2 \rightsquigarrow \rho''}{\rho \vdash i_1;i_2 \rightsquigarrow \rho''} \qquad \frac{\rho \vdash b \to \mathsf{false}}{\rho \vdash \mathsf{while}\ b\ \mathsf{do}\ i\ \mathsf{done} \rightsquigarrow \rho}$$

$$\frac{\rho \vdash b \to \mathsf{true} \quad \rho \vdash i \rightsquigarrow \rho' \quad \rho' \vdash \mathsf{while}\ b\ \mathsf{do}\ i\ \mathsf{done} \rightsquigarrow \rho''}{\rho \vdash \mathsf{while}\ b\ \mathsf{do}\ i\ \mathsf{done} \rightsquigarrow \rho''}$$

Because it is described using collections of rules, the judgment $\cdot \vdash \cdot \rightsquigarrow \cdot$ can be described with an inductive predicate exactly like the judgments for evaluation and update. We use the name exec for this judgment.

Like the judgment $\rho \vdash e \to v$, the judgment $\rho \vdash i \rightsquigarrow \rho'$ actually describes a partial function. However, this partial function cannot be described as a structural recursive function as we did when defining the functions lookup and af. For while loop, Such a function would present a recursive call where neither the environment nor the instruction argument would be a sub-structure of the corresponding initial argument. This failure also relates to the fact that the termination of programs is undecidable for this kind of language, while structural recursion would provide a terminating tool to compute whether programs terminate. In the later section on denotational semantics, we will discuss ways to encode a form of recursion that is powerful enough to describe the semantics as a recursive function.

4 Axiomatic semantics

We study now the encoding of axiomatic semantics as proposed by Hoare [20] and the weakest pre-condition calculus as proposed by Dijkstra [15]. The principle of this semantic approach is to consider properties that are satisfied by the variables of the program before and after the execution.

4.1 The semantic rules

To describe this approach, we use judgments of the following form: $\{P\}i\{Q\}$. This should be read as *if P is satisfied before executing i and executing i terminates, then Q is guaranteed to be satisfied after executing i.*

There are two key aspects in axiomatic semantics: first the behavior of assignment is explained by substituting variables with arithmetic expressions; second the behavior of control operators is explained by isolating properties that are independent from the choice made in the control operator and properties that can be deduced from the choice made in the control operator.

$$\frac{}{\{P\}\mathsf{skip}\{P\}} \qquad \frac{\{P\}i_1\{Q\} \quad \{Q\}i_2\{R\}}{\{P\}i_1;i_2\{R\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x{:=}e\{P\}} \qquad \frac{\{b \wedge P\}i\{P\}}{\{P\}\mathsf{while}\ b\ \mathsf{do}\ i\ \mathsf{done}\{\neg b \wedge P\}}$$

$$\frac{P \Rightarrow P_1 \quad \{P_1\}i\{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\}i\{Q\}}$$

In the rule for while loops, the property P corresponds to something that should be verified whether the loop body is executed 0, 1, or many times: it is independent from the choice made in the control operator. However, when the loop terminates, one knows that the test must have failed, this is why the output property for the loop contains $\neg b$. Also, if P should be preserved independently of the number of executions of the loop, then it should be preserved through execution of the loop body, but only when the test is satisfied.

We call the first four rules *structural rules*: each of them handles a construct of the programming language. The last rule, known as the *consequence* rule, makes it possible to mix logical reasoning about the properties with the symbolic reasoning about the program constructs. To prove the two premises that are implications, it is necessary to master the actual meaning of the properties, conjunction, and negation.

4.2 Theorem prover encoding

The first step is to define a data-type for assertions. Again, we keep things minimal. Obviously, the inference rules require that the language of assertions contain at least conjunctions, negations, and tests from the language's boolean expressions. We also include the possibility to have abitrary predicates on arithmetic expressions, represented by a name given as a string.

```
Inductive assert : Type :=
  a_b (b: bexpr) | a_not (a: assert) | a_conj (a a': assert) | pred (s: string)(l: int aexpr).

Inductive condition : Type := c_imp (a a':assert).
```

For variables that occur inside arithmetic expressions, we use valuation functions of type string$\rightarrow$ Z instead of environments and we define a new function af' (respectively bf', lf')

to compute the value of an arithmetic expression (respectively boolean expressions, lists of arithmetic expressions) for a given valuation. The function af' is more practical to use and define than af because it is total, while af was partial.

```
Fixpoint af' (g:string→Z)(a:aexpr) : Z :=
match a with avar s ⇒ g s | anum n ⇒ n | aplus e1 e2 ⇒ af' g e1 + af' g e2 end.
```

To give a meaning to predicates, we use lists of pairs associating names and predicates on lists of integers as *predicate* environments and we have a function f_p to map an environment and a string to a predicate on integers.

With all these functions, we can interpret assertions as propositional values using a function i_a and conditions using a function i_c.

```
Definition p_env := list(string*(list Z→Prop)).

Fixpoint i_a (m: p_env)(g:string→Z)(a:assert) : Prop :=
  match a with
    a_b e ⇒ bf' g e                | a_not a ⇒ ~ i_a m g a
  | pred p l ⇒ f_p m p (lf' g l)   | a_conj a1 a2 ⇒ i_a m g a1 ∧ i_a m g a2
  end.

Definition i_c (m:p_env)(g:string→Z)(c:condition) :=
 match c with c_imp a1 a2 ⇒ i_a m g a1 → i_a m g a2 end.
```

The validity of conditions can be expressed for a given predicate environment by saying that their interpretation should hold for any valuation.

```
Definition valid (m:p_env)(c:condition) := ∀ g, i_c m g c.
```

We also define substitution for arithmetic expressions, boolean expressions, and so on, each time traversing structures. The function at the level of assertions is called a_subst. We can then define the axiomatic semantics.

```
Inductive ax_sem (m :p_env): assert → instr → assert → Prop:=
  ax1 : ∀ P, ax_sem m P skip P
| ax2 : ∀ P x e, ax_sem m (a_subst P x e) (assign x e) P
| ax3 : ∀ P Q R i1 i2, ax_sem m P i1 Q → ax_sem m Q i2 R →
          ax_sem m P (sequence i1 i2) R
| ax4 : ∀ P b i, ax_sem m (a_conj (a_b b) P) i P →
          ax_sem m P (while b i) (a_conj (a_not (a_b b)) P)
| ax5 : ∀ P P' Q' Q i,
        valid m (c_imp P P') → ax_sem m P' i Q' → valid m (c_imp Q' Q) →
        ax_sem m P i Q.
```

4.3 Proving the correctness

We want to certify that the properties of programs that we can prove using axiomatic semantics hold for actual executions of programs, as described by the operational semantics. We first define a mapping from the environments used in operational semantics to the valuations used in the axiomatic semantics. This mapping is called e_to_f, the expression e_to_f e g x is the value of x in the environment e, when it is defined, and g x otherwise. The formula e_to_f e g is also written e@g. We express the correctness of axiomatic semantics by stating that if "exec r i r'" and "ax_sem P i Q" hold, if P holds in the initial environment, Q should hold in the final environment Q.

```
Theorem ax_sem_sound : ∀ m r i r' g P Q, exec r i r' → ax_sem m P i Q →
  i_a m (r@g) P → i_a m (r'@g) Q.
```

When we attempt to prove this statement by induction on exec and case analyis on ax_sem, we encounter problem because uses of consequence rules may make axiomatic semantics derivations arbitrary large. To reduce this problem we introduce a notion of *normalized* derivations where exactly one consequence step is associated to every structural step. We introduce an extra inductive predicate call nax to model these normalized derivation, with only four constructors. For instance, here is the constructor for loops:

```
nax4 : ∀ P P' Q b i,
  valid m (c_imp P P') → valid m (c_imp (a_conj (a_not (a_b b)) P') Q) →
  nax m (a_conj (a_b b) P') i P' → nax m P (while b i) Q.
```

We prove that ax_sem and nax are equivalent. This "organisational" step is crucial. We can now prove the correctness statement by induction on exec and by cases on nax, while a proof by double induction would be required with ax_sem.

Another key lemma shows that updating an environment for a variable and a value, as performed in operational semantics, and substituting an arithmetic expression for a variable, as performed in axiomatic semantics, are consistent.

```
Lemma a_subst_correct : forall a r1 e v m g r2 x,
  aeval r1 e v → s_update r1 x v r2 →
  (i_a m (r1@g) (a_subst a x e) ↔ i_a m (r2@g) a).
```

4.4 The weakest pre-condition calculus

Most of the structure of an axiomatic semantics proof can be deduced from the structure of the instruction. However, the assertions in loop invariants and in consequence rules cannot be guessed. Dijkstra proposed to annotate programs with the unguessable formulas and to automatically gather the implications used in consequence steps as a collection of conditions to be proved on the side. The result is a *verification condition generator* which takes annotated program as input and returns a list of conditions. We will now show how to encode such a verification condition generator (vcg).

We need to define a new data-type for these annotated programs.

```
Inductive a_instr : Type :=
  prec (a:assert)(i:a_instr) | a_skip | a_assign (s:string)(e:aexpr)
| a_sequence (i1 i2:a_instr) | a_while (b:bexpr)(a:assert)(i:a_instr).
```

The prec constructor is used to assert properties of a program's variables at any point in the program.

The computation of all the implications works in two steps. The first step is to understand what is the pre-condition for an annotated instruction and a given post-condition. For the a_while and prec constructs, the pre-condition is simply the one declared in the corresponding annotation, for the other constructs, the pre-condition has to be computed using substitution and composition.

```
Fixpoint pc (i:a_instr)(a:assert) {struct i} : assert :=
  match i with
    prec a' i ⇒ a'    | a_while b a' i ⇒ a'   | a_skip ⇒ a
  | a_assign x e ⇒ a_subst a x e              | a_sequence i1 i2 ⇒ pc i1 (pc i2 a)
  end.
```

The second step is to gather all the conditions that would appear in a minimal axiomatic semantics proof for the given post-condition, starting from the corresponding pre-condition.

```
Fixpoint vcg (i:a_instr)(post : assert) {struct i} : list condition :=
match i with
  a_skip ⇒ nil     | a_assign _ _ ⇒ nil   | prec a i ⇒ c_imp a (pc i post)::vcg i post
| a_sequence i1 i2 ⇒ vcg i2 post ++ vcg i1 (pc i2 post)
| a_while e a i ⇒
   c_imp (a_conj (a_not (a_b e)) a) post :: c_imp (a_conj (a_b e) a) (pc i a) :: vcg i a
end.
```

The correctness of this verification condition generator is expressed by showing that it suffices to prove the validity of all the generated conditions to ensure that the Hoare triple holds. This proof is done by induction on the instruction. We can then obtain a proof that relates the condition generator and the operational semantics. In this statement, un_annot maps an annotated instruction to the corresponding bare instruction.

```
Theorem vcg_sound :
  ∀ m i A, (valid_l m (vcg i A)) → ∀ g r1 r2, exec r1 (un_annot i) r2 →
  i_a m (e_to_f g r1) (pc i A) → i_a m (e_to_f g r2) A.
```

4.5 An example of use in proof by reflection

We consider the program that adds the n first positive integers. We use a predicate environment ex_m that maps two names le and pp to two predicates on lists of two integers. For the two integers x and y, the first predicate holds when $x \leq y$ and the second holds when $2 \times y = x \times (x + 1)$. With the help of a parser function, we can state the properties of interest for our program in a concise manner:

```
Example ex1 : ∀ g r2, 0 < n →
 exec (("x", 0)::("y", 0)::("n",n)::nil)
   (un_annot (parse_instr'
    "while x < n do [le(x,n) ∧ pp(y,x)] x:=x+1;y:=x+y done")) r2 →
   2*(r2@g)"y" = (r2@g)"x"*((r2@g)"x"+1).
```

After a few logistic steps, we can show that the conclusion is an instance of the pp predicate, and then apply the correctness theorem, which leads to two logical requirements. The first is that the verification conditions hold:

```
valid_I ex_m
  (vcg (parse_instr' "while x < n do [le(x,n) ∧ pp(y,x)] x:=x+1;y:=x+y done")
   (parse_assert' "pp(y,n)"))
```

After forcing the computation of the parser and the condition generator and a few more logistic steps, this reduces to the following logical statement

$$\begin{array}{l}\forall x\ y\ n.\\ (x \not< n \wedge x \leq n \wedge 2y = x(x+1) \Rightarrow 2 * y = n(n+1)) \wedge\\ (x < n \wedge x \leq n \wedge 2y = x(x+1) \Rightarrow x+1 \leq n \wedge 2(x+1+y) = (x+1)(x+2)).\end{array}$$

This is easily proved using the regular Coq tactics. The second requirement is that the pre-condition should be satisfied and reduces to the statement

$$0 \leq n \wedge 0 = 0.$$

We have actually automated proofs about programs inside the Coq system, thus providing a simple model of tools like Why [17].

5 Denotational semantics

In denotational semantics, the aim is to describe the meaning of instructions as functions. The functions need to be partial, because some instructions never terminate on some inputs. We already used partial functions for the functional encoding of expression evaluation. However, the partial recursive function that we defined were structural, and therefore guaranteed to terminate. The execution function for instructions does not fit in this framework and we will first define a new tool to define recursive function. Most notably, we will need to use non-constructive logic for this purpose.

Again the partial functions will be implemented with the option inductive type, but the None constructor will be used to represent either that an error occurs or that computation does not terminate.

5.1 The fixpoint theorem

The approach described in [29] relies on Tarski's fixpoint theorem, which states that every continuous function in a complete partial order with a minimal element has a least fixpoint and that this fixpoint is obtained by iterating the function from the minimal element.

Our definition of complete partial order relies on the notion of chains, which are monotonic sequences. A partial order is a type with a relation $\subseteq$ that is reflexive, antisymmetric, and transitive; this partial order is complete if every chain has a least upper bound. A function f is continuous if for every chain c with a least upper bound l, the value $f(l)$ is the least upper bound of the sequence $f(c_n)$. Notice that when defining continuous function in this way, we do not require f to be monotonic; actually, we prove that every continuous function is monotonic.

The proof of Tarski's theorem is quite easy to formalize and it can be formalized using intuitionistic logic, so the plain calculus of constructions is a satisfactory framework for this.

5.2 Partial functions form a complete partial order

The main work in applying Tarski's theorem revolves around proving that types of partial functions are complete partial orders. A type of the form $\mathsf{option}\ A$ has the structure of a complete partial order when choosing as order the relation such that $x \subseteq y$ exactly when $x = y$ or $x = \mathsf{None}$. The element None is the minimal element. Chains have a finite co-domain, with at most two elements, the least upper bound can be proved to exist using the non-consructive excluded-middle axiom; this is our first step outside constructive mathematics.

Given an arbitrary complete partial order $(B, \subseteq)$, the type of functions of type $A \to B$ is a complete partial order for the order defined as follows:

$$f \subseteq g \Leftrightarrow \forall x, f(x) \subseteq g(x).$$

The proof that this is a complete partial order requires other non-constructive axioms: extensionality is required to show that the order is antisymetric and a description operator is required to construct the least upper bound of a chain of functions. We actually rely on the non-constructive ϵ operator proposed by Hilbert and already used in HOL or Isabelle/HOL. This ϵ operator is a function that takes a type T, a proof that T is inhabited, a predicate on T, and returns a value in T that is guaranteed to satisfy the predicate when possible.

For a sequence of functions f_n (not necessarily a chain), we can define a new function f, which maps every x to the value given by the ϵ operator for the predicate "to be the least upper bound of the sequence sequence $f_n(x)$". Now, if it happens that f_n is a chain, then each of the sequences $f_n(x)$ is a chain, $f(x)$ is guaranteed to be the least upper bound, and f is the least upper bound of f_n.

In practice, Tarski's least fixpoint theorem is a programming tool. If one wishes to define a recursive function with a definition of the form

$$f\ x = e$$

such that f appears in e, it suffices that the function $\lambda f.\lambda x.e$ is continuous and the theorem returns a function that satisfies this equation, a natural candidate for the function that one wants to define. We encode this fixpoint operator as a function called Tarski_fix.

5.3 Defining the semantics

For a while loop of the form while b do i done, such that the semantic function for i is f_i, we want the value of semantic function to be the function $\phi_{b,i}$ such that :

$$\phi_{b,i}(\rho) = \begin{cases} \rho & \text{if bf } b\text{=false} \\ \phi_{b,i}(\rho') & \text{if bf } b\text{=true and } f_i(\rho) = \text{Some } \rho' \\ \text{None} & \text{otherwise} \end{cases}$$

This function $\phi_{b,i}$ is the least fixpoint of the function F_phi obtained by combining a conditional construct, a sequential composition function (already described using the bind function, and a few constant functions. We encode F_phi and phi as follows:

```
Definition ifthenelse (A:Type)(t:option bool)(v w: option A) :=
  match t with Some true ⇒ v | Some false ⇒ w | None ⇒ None end.

Notation "'IF x 'THEN a 'ELSE b" := (ifthenelse _ x a b) (at level 200).

Definition F_phi (A:Set)(t:A→option bool)(f g :A→option A) : A → option A :=
   fun r ⇒ 'IF (t r) 'THEN (bind (f r) g) 'ELSE (Some r).
```

We proved that each of the constructs and F_phi are continuous. The semantics for instructions can then be described by the following functions:

```
Definition phi := fun A t f ⇒ Tarski_fix (F_phi A t f).

Fixpoint ds(i:instr) : (list(string*Z)) → option (list(string*Z)) :=
match i with
  assign x e ⇒ fun l ⇒ bind (af l e)(fun v ⇒ update l x v)
| sequence i1 i2 ⇒ fun r ⇒ (ds i1 r)(ds i2)
| while e i ⇒ fun l ⇒ phi env (fun l' ⇒ bf l' e)(ds i) l
| skip ⇒ fun l ⇒ Some l
end.
```

We also proved the equivalence of this semantic definition and the natural semantics specification:

```
Theorem ds_eq_sn : ∀ i l l', ds i l = Some l' ↔ exec l i l'.
```

We actually rely on the second part of the fixpoint theorem, which states that the least fixpoint of a continuous function is the least upper bound of the chain obtained by iterating the function on the least element. In our case, this gives the following corollary:

$$\forall x\ v, \phi\ x = \text{Some } v \Rightarrow \exists n, \text{F\_phi}^n\ (\text{fun } y \rightarrow \text{None})\ x = \text{Some } x$$

We can then proceed with a proof by induction on the number n.

Unlike the functions af, af', or vcg, the function phi is not usable for computation inside the theorem prover, but $\mathsf{F\_phi}^n$ can be used to compute using approximations. We can still extract this code and execute it in Ocaml, as long as we extract the Tarski fixpoint theorem to a simple fixpoint function:

```
let rec fix f = f (fun y → fix f y)
```

This interpreter loops when executing a looping program; this is predicted in the Coq formalization by a value of None.

6 Abstract interpretation

The goal of abstract interpretation [13] is to infer automatically properties about programs based on approximations described as an abstract domain of values. Approximations make it possible to consider several executions at a time, for example all the executions inside a loop. This way the execution of arbitrary programs can be approximated using an algorithm that has polynomial complexity.

Abstract values are supposed to represent subsets of the set of concrete values. Each abstract interpreter works with a fixed set of abstract values, which much have a certain structure. An operation on abstract values must be provided for each operation in the language (in our case we only have to provide an addition on abstract values). The subset represented by the result of an abstract operation must contain all the values of the corresponding operation when applied to values in the input subsets. The set of abstract values should also be ordered, in a way that is compatible with the inclusion order for the subsets they represent. Also, the type of abstract values should also contain an element corresponding to the whole set of integers. We will call this element the bottom abstract value. The theoretical foundations provided by Cousot and Cousot [13] actually enumerate all the properties that are required from the abstract values.

Given an abstract valuation where variable names are mapped to abstract values, we program an abstract evaluation function ab_eval for arithmetic expressions that returns a new abstract value. This function is programmed exactly like the function af' we used for axiomatic semantics, simply replacing integer addition with an abstract notion of addition on abstract values.

When we need to evaluate with respect to an abstract environment l, i.e., a finite list of pairs of variable names and abstract values, we use the function (ab_lookup l) that associates the bottom value to all variables that do not occur in the abstract environment.

Abstract execution of instructions takes as input an abstract environment and a bare instruction and returns the pair of an annotated instruction and an optional final abstract environment. When the optional environment is None, this means that the analysis detected that concrete execution never terminate. The annotations in the result instruction describe information that is guaranteed to be satisfied when execution reaches the corresponding point.

Abstract execution for assignments, sequences, and skip instructions is natural to express: we just compute abtract values for expressions and pass abstract environments around as we did in the concrete semantics. For while loops, we handle them in a static way: our abstract interpreter is designed as a tool that always terminate (even if the analyzed program loops for ever).

The approach is to make the abstract environment coarser and coarser, until we reach an approximation that is stable through abstract interpretation of the loop body. Thus, we want to find an invariant abstract environment for loops, as we did in axiomatic semantics. Finding the best possible approximation is undecidable and over-approximation is required. We chose to implement a simple strategy:

1. We first check whether the input abstract environment for the while loop is *stable*, that is, if abstract values in the output environment are included in the corresponding abstract value for the input environment,

2. If this fails, we use a widen function to compute an over-approximation of both the input and the output, and we then check whether the new environment is stable, widen), and we check whether this new environment is stable,

3. If the first two steps failed, we overapproximate every value with the bottom abstract value; this is necessarily stable but gives no valuable information about any variable.

We also incorporate information from the loop test. When the test has the form $v < e$, where v is a variable, we can use this to refine the abstract value for v. At this point, we may detect that the new abstract value represents the empty set, this only happens when the test can never succeed or never fail, and in this case some code behind this test is dead-code. This is performed by a function intersect_env. This function takes a first boolean argument that is used to express that we check whether the test is satisfied or falsified. This function returns None when the test can never be satisfied or can never be falsified. When dead-code is detected, we mark the instruction with false assertions, to express that the location is never reached, (this is done in mark).

To check for stability of environments, we first need to combine the input and the output environment to find a new environment where the value associated to each variable contains the two values obtained from the two other environments. This is done with a function noted l @@ l' (we named it join_env). For a given while loop body, we call intersect_env and mark or @@ three times, one for every stage of our simple strategy. These operations are gathered in a function fp_1.

```
Definition fp1(l0 l:ab_env)(b:bexpr)(i:instr)(f:ab_env→ a_instr*option ab_env) :=
match intersect_env true l b with
  None ⇒ (prec false_assert (mark i), Some l)
| Some l' ⇒ let (i', l'') := f l' in
  match l'' with None ⇒ (i', None) | Some l2 ⇒ (i', Some (l0 @@ l' @@ l2)) end
end.
```

This function takes as argument the function f that performs abstract interpretation on the loop body i. We use this function fp_1 several times and combine it with widening functions to obtain a function fp that performs our three stage strategy. When the result of fp is (i, Some l), l satisfies the equation snd(f i l)=l.

Our abstract interpreter is then described as a recursive function abstract_i (here we use to_a to transform an environment into an assertion, and to_a' for optional environments, mapping None to false_assert).

```
Fixpoint abstract_i (i : instr)(l : ab_env) : a_instr*option ab_env :=
match i with
  skip ⇒ (prec (to_a l) a_skip, Some l)
| sequence i1 i2 ⇒
  let (i'1, l') := abstract_i i1 l in
  match l' with
    None ⇒ (a_sequence i'1 (prec false_assert (mark i2)), None)
  | Some l' ⇒ let (i'2, l'') := abstract_i i2 l' in (a_sequence i'1 i'2, l'')
  end
| assign x e ⇒
  (prec (to_a l) (a_assign x e), Some (ab_update l x (ab_eval (ab_lookup l) e)))
| while b i ⇒
  match intersect_env true l b with
    None ⇒
   (prec (to_a e)(a_while b (a_conj (a_not (a_b b)) (to_a l)) (mark i)), Some l)
  | Some l' ⇒
    let (i',l'') := fp l b i (abstract_i i) in
      match l'' with
        None ⇒ (prec (to_a l) (a_while b (to_a l) i'), intersect_env false l)
      | Some l'' ⇒ (prec (to_a l) (a_while b (to_a l'') i'), intersect_env false l'' b)
      end
  end
end.
```

This abstract interpreter is a programming tool: it can be run with an instruction and a set of initial approximations for variables. It returns the same instruction, where each location is annotated with properties about the variables at this location, together with properties for the variables at the end. This abstract interpreter is structurally recursive and can be run inside the Coq proof system.

We proved a correctness statement for this abstract interpreter. This statement relies on the verification condition generator that we described earlier.

```
Theorem abstract_i_sound:
  ∀ i e i' e' g, abstract_i i e = (i', e') → i_lc m g (vcg i' (to_a' e'))).
```

This theorem is proved by induction on i. We need to establish a few facts:

1. the order of variables does not change in successive abstract environments,
2. abstract execution is actually monotonic: given wider approximations, execution yields wider results (given reasonable assumptions for intersect_env)
3. the fp function (which handles loop bodies) either yields an abstract environment that is an over approximation of its input or detects non-termination of the loop body,
4. the verification condition generator is monotonic with respect to implication: if the conditions generated for i and a post-condition p hold and $p \rightarrow q$ is valid, then the conditions generated for i and q also hold and pc i $p \rightarrow$ pc i q is also valid. This property is needed because abstract interpreters and condition generators work in reverse directions.

This abstract interpreter was developed in a modular fashion, where the domain of abstract values is described using a module interface. We implemented an instance of this domain for intervals.

7 Conclusion

This overview of formalized programming language semantics is elementary in its choice of a very limited programming language. Because of this, some important aspects of programming languages are overlooked: *binding*, which appears as soon as local variables or procedures and functions are allowed, *typing*, which is a useful programming concept for the early detection of programming errors, *concurrency*, which is useful to exploit modern computing architectures, etc. Even for this simplistic programming language, we could also have covered two more aspects: program transformations [7] and compilation [4].

Three aspects of this work are original: we obtain tools that can be executed inside the Coq prover for proof by reflection; our work on denotational semantics shows that the conventional extraction facility of the Coq system can also be used for potentially non terminating functions, thanks to well chosen extraction for Tarski's fixpoint theorem; last, our description of an abstract interpreter is the first to rely on axiomatic semantics to prove the correctness of an abstract interpreter.

Concerning reflection, we find it exciting that the theorem prover can be used to execute programs in the object language (in work not reported here we show how to construct an incomplete interpreter from a structural operational semantics), to generate condition verifications about programs (thanks to the verification condition generator), and to prove the conditions using the normal mode of operation of theorem prover. More interestingly, the abstract interpreter can be run on programs to generate simultaneously annotated programs and the proof that these annotated programs are consistent.

Formal verification techniques based on verification condition generators suffer from the burden of explicitely writing the loop invariants. Chaieb already suggested that the loop invariants could be obtained through abstract interpretation [10], generating proof traces

that can be verified in theorem provers. Our partial correctness theorem for the abstract interpreter suggests a similar approach here, except that we also proved the abstract interpreter correct. An interesting improvement would be to make manually written assertions collaborate with automatically generated ones. First there should be a way to assume that all assertions computed by an abstract interpreter are implicitly present in assertions; second abstract interpreters could take manual annotations as clues to improve the quality of the abstract environments they compute.

References

[1] G. Barthe, G. Dufay, L. Jakubiec, S. Melo de Sousa, and B. Serpette. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 302–319. Springer-Verlag, 2001.

[2] Gilles Barthe, Mark Ruys, and Henk Barendregt. A two-level approach towards lean proof-checking. In *TYPES '95: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 16–35, London, UK, 1996. Springer-Verlag.

[3] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2000.

[4] Yves Bertot. A certified compiler for an imperative language. Research Report RR-3488, INRIA, 1998.

[5] Yves Bertot. Formalizing a jvml verifier for initialization in a theorem prover. In *Computer Aided Verification (CAV'2001)*, volume 2102 of *LNCS*, pages 14–24. Springer-Verlag, 2001.

[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 *TAPSOFT'95*, volume 915 of *LNCS*, pages 531–545, 1995.

[8] 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.

[9] S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.

[10] Amine Chaieb. Proof-producing program analysis. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 2006.

[11] Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[12] Catarina Coquand. Agda. www.cs.chalmers.se/~catarina/agda.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[14] Pichardie David. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Universit Rennes 1, 2005. In french.

[15] Edsger W. Dijkstra. *A discipline of Programming*. Prentice Hall, 1976.

[16] 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.

[17] Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *International Workshop TYPES'98*, volume 1657 of *Lecture Notes in Computer Science*. Pringer-Verlag, March 1998.

[18] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[19] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[20] Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, October 1969.

[21] 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).

[22] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES 2002*, volume 2646 of *LNCS*. Springer-Verlag, 2003.

[23] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[24] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Asp. Comput.*, 10(2):171–186, 1998.

[25] Lawrence C. Paulson and Tobias Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[26] D. Terrasse. Encoding natural semantics in Coq. In *AMAST'95*, Springer-Verlag LNCS, July 1995.

[27] Joachim van den Berg and Bart Jacobs. The loop compiler for java and jml. In *TACAS 2001*, pages 299–312. Springer-Verlag, 2001.

[28] David von Oheimb. *Analyzing Java in Isabelle/HOL, Formalization, Type Safety, and Hoare Logic*. PhD thesis, Technische Universität München, 2000.

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

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

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 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 - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

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