Proof by reflection in semantics

Kuntal das Barman, Yves Bertot

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Proof by reflection in semantics

Kuntal Das Barman — Yves Bertot

N° 5134

Mars 2004

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--5134--FR+ENG

Proof by reflection in semantics

Kuntal Das Barman , Yves Bertot

Thème 2 — Génie logiciel
et calcul symbolique
Projets Lemme

Rapport de recherche n° 5134 — Mars 2004 — 17 pages

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

Key-words: formal methods, type theory, calculus of constructions, semantics, reflection

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

Preuve par réflexion dans la sémantique

Résumé : Les approches conventionnelles pour décrire la sémantique des langages de programmation reposent habituellement sur des relations, en particulier sur des relations inductives. On peut ensuite simuler l'exécution de programmes en faisant fonctionner des outils de recherche de preuve. Nous décrivons une approche fonctionnelle pour automatiser les démonstrations portant sur la sémantique des langages de programmation. La méthode de réflexion est utilisée pour prendre en compte les faits déjà connus dans le contexte de la démonstration. La contribution principale de ce travail est que nous avons développé une approche systématique pour décrire et manipuler des inconnues dans l'exécution symbolique de programme pour le développement de preuves formelles. L'outil que nous obtenons est plus rapide et plus puissant que les approches conventionnelles.

Mots-clés : méthodes formelles, théorie des types, calcul des constructions, sémantique, réflexion

Contents

1 Introduction

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

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

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

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

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

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

In type theory based proof assistant like **Coq** [8] functions are provided and reductions are used to compute them. Functions compute on data objects. In usual formal proofs the facts related to computations are provided as assertions, which are in fact relations between data objects. We use reflection to bridge this gap to use functions in proof search. We collect the data objects from the given facts and put them in several tables. Unlike unification and resolution based approach, we do not look for a match for a hypothesis in the context to do the computation, we consult these tables and use functions to do the computation. Functional approach is more focused and does not need to search in the entire search space, as we can directly consult the particular table related to the enquiry. Function evaluation is performed by term reduction and reasoning on unknown expression needs special care. We show a way to work around this problem. To implement the function which does not follow the structural recursion we recall *iteration technique* [2].

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

Here is the structure of the paper.

In Section 2 we define the simple imperative language IMP. We formalize its operational semantics by inductive relations and discuss about our main objective, a step towards automation, giving a look into the current situation. We show the difficulties with the current solution. In Section 3 we present the functional interpretations of IMP. We show how to collect and use data objects from the provided facts to execute instructions in a functional approach. In Section 4 we give the idea of reflection and how it will be used in our context. We provide a systematic and general technique to work with metavariable. To implement the function which does not follow structural recursion we use *iteration technique*, which can be found in detail in our previous work [2].

All the definitions have been implemented in **Coq** and all the results proved formally in it. We use here an informal mathematical notation, rather than giving **Coq** code. There is a direct correspondence between this notation and the **Coq** formalization. Using the **PCoq**

graphical interface (available on the web[1]), we also implemented some of this more intuitive notation. The **Coq** files of the development are on the web[2].

2 IMP and its semantics

Winskel [10] presents a small programming language IMP with *while* loops. IMP is a simple imperative language with integers, truth values true and false, memory locations to store the integers, arithmetic expressions, boolean expressions and instructions. The formation rules are

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

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

instructions: $i ::= \mathsf{skip} \mid X \leftarrow a \mid i_0; i_1 \mid \mathsf{if}\ b\ \mathsf{then}\ i_0\ \mathsf{else}\ i_1 \mid \mathsf{while}\ b\ \mathsf{do}\ i$

where n ranges over integers, X ranges over locations, a ranges over arithmetic expressions, b ranges over boolean expressions and i ranges over instructions. We formalize this language in **Coq** by three inductive types AExp, BExp, and Inst.

Locations are represented by integers. One should not confuse the integer denoting a location with the integer contained in the location. Therefore, in the definition of AExp, we denote the constant value n by Num(n) and the memory location with address v by Loc(v). The specification of arithmetic expressions is as follows.

$$\begin{array}{l}\mathsf{AExp}: \mathbf{Set}\\ \mathsf{Loc}(\cdot)\ : \mathbb{Z} \to \mathsf{AExp}\\ \mathsf{Num}(\cdot): \mathbb{Z} \to \mathsf{AExp}\\ (\cdot\ +\ \cdot): \mathsf{AExp} \to \mathsf{AExp} \to \mathsf{AExp}\\ \quad\vdots\end{array}$$

We see instructions as state transformers, where a state is a partial map from memory locations to integers. We can actually represent a state as a list of bindings between memory locations and values. Two states are different if there's at least one different binding.

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

When the content of a location v in a state σ is mapped to n, we represent the new state as $[v \mapsto n, \sigma]$.

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

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

$$(\langle\cdot,\cdot\rangle_{\mathsf{A}} \rightsquigarrow \cdot)\colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{Z} \to \mathbf{Prop}$$
$$(\langle\cdot,\cdot\rangle_{\mathsf{B}} \rightsquigarrow \cdot)\colon \mathsf{BExp} \to \mathsf{State} \to \mathbb{B} \to \mathbf{Prop}$$
$$(\langle\cdot,\cdot\rangle_{\mathsf{I}} \rightsquigarrow \cdot)\colon \mathsf{Inst} \to \mathsf{State} \to \mathsf{State} \to \mathbf{Prop}$$

For arithmetic expressions constants are evaluated to themselves. Memory locations are interpreted by looking up their values in the state. Consistently with the spirit of operational semantics, we define the lookup operation by derivation rules rather than by a function.

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

where

$$\begin{array}{ll}
\mathsf{lookup}\colon \mathsf{State} \to \mathbb{Z} \to \mathbb{Z} \to \mathbf{Prop} & \\
\mathsf{lookup_found}\colon & (v, n\colon \mathbb{Z}; \sigma\colon \mathsf{State})(\mathsf{lookup}\ [v \mapsto n, \sigma]\ v\ n) \\
\mathsf{lookup_recursively}\colon & (v, v', n, n'\colon \mathbb{Z}; \sigma\colon \mathsf{State}) \\
& \quad v \neq v' \to (\mathsf{lookup}\ \sigma\ v\ n) \to (\mathsf{lookup}\ [v' \mapsto n', \sigma]\ v\ n)
\end{array}$$

Notice that we do not assign any value to empty locations, rather leave them undefined. Therefore instead of giving a default value to uninitialized variables, the semantics decides to stop evaluation.

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

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

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

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

$$\begin{array}{ll}
(\langle\cdot,\cdot\rangle_{\mathsf{A}} \rightsquigarrow \cdot)\colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{Z} \to \mathbf{Prop} & \\
\mathsf{eval_Num}\colon & (n\colon \mathbb{Z}; \sigma\colon \mathsf{State})(\langle \mathsf{Num(n)}, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n) \\
\mathsf{eval_Loc}\colon & (v, n\colon \mathbb{Z}; \sigma\colon \mathsf{State})(\mathsf{lookup}\ \sigma\ v\ n) \to (\langle \mathsf{Loc(v)}, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n) \\
\mathsf{eval_Plus}\colon & (a_0, a_1\colon \mathsf{AExp}; n_0, n_1\colon \mathbb{Z}; \sigma\colon \mathsf{State}) \\
& (\langle a_0, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0) \to (\langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0) \to \\
& (\langle a_0 + a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0 + n_1) \\
\vdots &
\end{array}$$

For boolean expressions we have a similar form of evaluation. The operational semantics of boolean expressions is defined by the inductive relation

$$
\begin{array}{l}
(\langle\cdot,\cdot\rangle_{\mathsf{B}} \rightsquigarrow \cdot)\colon \mathsf{BExp} \to \mathsf{State} \to \mathbb{B} \to \mathbf{Prop} \\
\mathsf{eval\_}\mathbf{True}\colon (\sigma\colon \mathsf{State})(\langle \mathsf{true}, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \\
\mathsf{eval\_}\mathbf{False}\colon (\sigma\colon \mathsf{State})(\langle \mathsf{false}, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \\
\mathsf{eval\_}\mathbf{Less\_or\_equal}\colon \ (a_1, a_2\colon \mathsf{AExp}; n_1, n_2\colon \mathbb{Z}; \sigma\colon \mathsf{State}) \\
\qquad\qquad\qquad\qquad (\langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1) \to (\langle a_2, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_2) \to \\
\qquad\qquad\qquad\qquad (n_1 \leq n_2) \to (\langle a_1 \leq a_2, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \\
\mathsf{eval\_}\mathbf{not\_Less\_or\_equal}\colon \ (a_1, a_2\colon \mathsf{AExp}; n_1, n_2\colon \mathbb{Z}; \sigma\colon \mathsf{State}) \\
\qquad\qquad\qquad\qquad\qquad (\langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1) \to (\langle a_2, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_2) \to \\
\qquad\qquad\qquad\qquad\qquad (n_2 < n_1) \to (\langle a_1 \leq a_2, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \\
\quad \vdots
\end{array}
$$

The operational semantics of instructions specifies how an instruction maps states to states. For instance, the assignment $X \leftarrow a$ evaluates the expression a and then updates the contents of the location X to the value of a.

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

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

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

Notice that we require a location to be already defined in the state to update it. We have not specified any operational semantics to update a location not present in the state. This corresponds to requiring that all variables are explicitly initialized before execution.

Evaluating a sequential composition $i_1; i_2$ on a state σ consists in evaluating i_1 on σ, obtaining a new state σ_1, and then evaluating i_2 on σ_1 to obtain the final state σ_2.

$$\frac{\langle i_1, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma_1 \quad \langle i_2, \sigma_1\rangle_{\mathsf{I}} \rightsquigarrow \sigma_2}{\langle i_1; i_2, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma_2}$$

We have two rules for *while* loops. If b evaluates to true, i is executed with σ to produce a new state σ', with which the loop is evaluated again. If b evaluates to false, execution terminates and leaves the state unchanged.

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

These rules and the rules for skip and conditional expressions can be formalized in **Coq** in a straightforward way by an inductive relation.

$$
\begin{array}{ll}
\langle\cdot,\cdot\rangle_{\mathsf{I}} \rightsquigarrow \cdot : \mathsf{Inst} \to \mathsf{State} \to \mathsf{State} \to \mathbf{Prop} & \\
\mathsf{eval\_skip} : (\sigma : \mathsf{State})(\langle \mathsf{skip}, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma) & \\
\mathsf{eval\_assign} : & (\sigma, \sigma' : \mathsf{State}; v, n : \mathbb{Z}; a : \mathsf{AExp}) \\
& (\langle a, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n) \to (\sigma_{[v \mapsto n]} \rightsquigarrow \sigma') \to (\langle v \leftarrow a, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma') \\
\mathsf{eval\_scolon} : & (\sigma, \sigma_1, \sigma_2 : \mathsf{State}; i_1, i_2 : \mathsf{Inst}) \\
& (\langle i_1, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma_1) \to (\langle i_2, \sigma_1\rangle_{\mathsf{I}} \rightsquigarrow \sigma_2) \to (\langle i_1; i_2, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma_2) \\
\quad \vdots & \\
\mathsf{eval\_while\_true} : & (b : \mathsf{BExp}; i : \mathsf{Inst}; \sigma, \sigma', \sigma'' : \mathsf{State}) \\
& (\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \to (\langle i, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma') \to \\
& (\langle \mathsf{while}\ b\ \mathsf{do}\ i, \sigma'\rangle_{\mathsf{I}} \rightsquigarrow \sigma'') \to (\langle \mathsf{while}\ b\ \mathsf{do}\ i, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma'') \\
\mathsf{eval\_while\_false} : & (b : \mathsf{BExp}; i : \mathsf{Inst}; \sigma : \mathsf{State}) \\
& (\langle b, \sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \to (\langle \mathsf{while}\ b\ \mathsf{do}\ i, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma)
\end{array}
$$

2.1 Difficulty in automation

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

Lemma 1. $\forall \sigma, \sigma', \sigma'' : \mathsf{State}. \forall i_1, i_2 : \mathsf{Inst}.$
$$\underbrace{(\langle i_1, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma') \to (\langle i_2, \sigma'\rangle_{\mathsf{I}} \rightsquigarrow \sigma'')}_{\text{facts}} \to \underbrace{(\langle i_1; i_2, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma'')}_{\text{goal}}.$$

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

Lemma 2. $\forall \sigma : \mathsf{State}. \forall v : \mathbb{Z}. \underbrace{(\mathsf{lookup}\ \sigma\ v\ 1)}_{\text{facts}} \to \underbrace{\langle \mathsf{while}\ 3\ \leq\ v\ \mathsf{do}\ skip, \sigma\rangle_{\mathsf{I}} \rightsquigarrow \sigma}_{\text{goal}}.$

Given the fact that the location v is bound to 1 in the state σ, we need to prove that execution of the while loop will not change state. We have necessary informations to prove

that the boolean expression $3 \leq 1$ will be evaluated to false, but such a proof does not exist in the context and needs to be computed. A unification and resolution based procedure does not have computation power for this. A way to solve this problem is to use functions instead of relations. Functions can also compute intermediate results, thus handling lemma 1.

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

3 Functional interpretation

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

Lemma 3. $\forall a_1, a_2 : \mathsf{AExp}. \forall \sigma : \mathsf{State}$
$$\underbrace{\langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1 \rightarrow \langle a_2, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_2}_{\text{facts}} \rightarrow \underbrace{\langle a_1 + a_2, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1 + n_2}_{\text{goal}}.$$

To keep these information on arithmetic expressions, a_1 and a_2, we create a table $\mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}$ (to be read as *list of results given for arithmetic subexpressions*). $\mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}$ a list of triplets, where each triplet consists of a state, an arithmetic expression and the integer value of this arithmetic expression in this state. For example, for lemma 3 $\mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}} = [(\sigma, a_2, n_2), (\sigma, a_1, n_1)]$. We ensure that this table contains information that is only provided in the context with the *consistency function*:

$\mathcal{C}_{\mathsf{AExp}}(\cdot) : (\mathsf{list\ State} * \mathsf{AExp} * \mathbb{Z}) \rightarrow \mathbf{Prop}$
$[]^{\mathsf{r}}_{\mathsf{AExp}} : \mathsf{True}$
$[(\sigma,\ \mathsf{a},\ \mathsf{n}),\ \mathsf{l}^{\mathsf{r}}_{\mathsf{AExp}}]^{\mathsf{r}}_{\mathsf{AExp}} : \forall\ a : \mathsf{AExp}. \forall\ n : \mathbb{Z}. \forall\ \sigma : \mathsf{State}.\ \langle a, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n \wedge \mathcal{C}_{\mathsf{AExp}}(\mathsf{l}^{\mathsf{r}}{}_{\mathsf{AExp}})$.

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

To evaluate any expression, we first look in the corresponding table, whether we already know the result or not. If not, we follow the semantics. Therefore, the evaluation function

for arithmetic expressions is defined as follows:

$$
\begin{array}{l}
[\![\cdot]\!]\colon \mathsf{AExp} \to \mathsf{State} \to (\mathsf{list}\ \mathsf{State} * \mathbb{Z} * \mathbb{Z}) \\
\qquad\qquad \to (\mathsf{list}\ \mathsf{State} * \mathsf{AExp} * \mathbb{Z}) \to (\mathsf{option}\ \mathbb{Z}) \\
\lambda \mathsf{a}\colon \ldots \lambda\sigma\colon \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}}\colon \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}\colon \ldots \\
\mathsf{x}\ \text{ where } (\sigma,\ \mathsf{a},\ \mathsf{x}) \in \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}} \\
\text{Otherwise:} \\
[\![\mathsf{Num(n)},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma := n \\
[\![\mathsf{Loc(v)},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma := \mathsf{flookup}(\sigma,\ \mathsf{v},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}}) \\
[\![a_1 + a_2,\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma := \left\{ \begin{array}{l} [\![a_1, \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma \\ \quad + [\![a_2, \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma \\ \qquad \text{if } [\![a_1,\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma \neq \mathsf{error} \\ \qquad \&\ [\![a_2,\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma \neq \mathsf{error} \\ \mathsf{error} \\ \qquad \text{if } [\![a_1,\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma = \mathsf{error} \\ \qquad \text{or } [\![a_2,\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma = \mathsf{error} \end{array} \right. \\
[\![a_0 - a_1,\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma := \cdots \\
[\![a_0 * a_1,\ \mathsf{T}^{\mathsf{r}}_{\mathsf{lookup}},\ \mathsf{T}^{\mathsf{r}}_{\mathsf{AExp}}]\!]_\sigma := \cdots
\end{array}
$$

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

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

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

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

Similarly, we define the update function

$$
\cdot[\cdot/\cdot]\colon \mathsf{State} \to \mathbb{Z} \to \mathbb{Z} \to (\mathsf{list}\ \mathsf{State} * \mathbb{Z} * \mathbb{Z} * \mathsf{State}) \to (\mathsf{option}\ \mathsf{State}).
$$

4 Proof by reflection

In usual formal proofs, hypotheses about computations are represented as assertions that some relation hold for some piece of data. The proof search mechanisms usually search the context containing all these hypotheses to see if the goal can be solved directly if it is one of these assumptions. In a proof system based on type theory like **Coq**, functions are also provided and reduction can be used to compute with these functions [7]. The idea of reflection is to use these functions to perform the proof search. But functions compute on formalized data and the (context) hypotheses are not formalized data at the level of these functions but only at the level of the proof system. For the needs of reflection, we have already built data to represent the hypotheses at the level where functions can compute.

This discussion on levels can be found in [1]. Reflection was pioneered in Coq by Samuel Boutin [3], where reflection was used to decide efficiently whether two expressions, denoting values in a ring, are equal. Similarly, Kumar Neeraj Verma and Jean Goubault-Larrecq [9] has recently used reflection to build a certified BDD algorithm in **Coq**. Work on reflection in **Coq** can be also found in [6]. In the literature we do not find any reference where reflection was used in semantics.

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

In our case, we would like to prove that $\langle \sigma, i \rangle_{\mathsf{I}} \rightsquigarrow \sigma'$ holds given a set of hypotheses Γ. We should, therefore, write a function Q which takes *data objects from* Γ that truly represents Γ, σ and i as inputs and returns σ' only if when $\langle \sigma, i \rangle_{\mathsf{I}} \rightsquigarrow \sigma'$ holds along with Γ. To have this last criteria we need to prove $Q(\sigma,\ i,\ \textit{data objects from}\ \Gamma) = \sigma' \Rightarrow \Gamma \rightarrow \langle \sigma, i \rangle_{\mathsf{I}} \rightsquigarrow \sigma'$.

In section 2 we described two problems that need to be solved by proof search engines to build proof in semantics. The first problem is to find intermediate values in computation. This is solved in a natural way, the evaluation computations. The second problem is to interleave arithmetic computation with proof search and this, too, can be easily solved if evaluation functions call the relevant arithmetic functions. For these problems, proof tools based on function evaluation are better than proof search tools based on unification and resolution. Function evaluation is also more focused than proof search and therefore more efficient. All this process becomes very powerful if fast reduction mechanisms are implemented in proof assistant [5].

4.1 Giving names

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

Lemma 4. $\forall \sigma\colon \mathsf{State}.\forall v\colon \mathbb{Z}.(\mathsf{lookup}\ \ \sigma\ v\ 3) \rightarrow \langle \sigma, (\mathsf{while}\ v\ \leq\ 1\ \mathsf{do\ skip}) \rangle_{\mathsf{I}} \rightsquigarrow \sigma.$

In this case the memory state is given by a universally quantified variable σ. If we remember, in section 2, we defined state as an Inductive type Set with two constructors, one describes the case when the list is empty and the other describes the case when the list is non empty. But it does not say anything if we don't know anything about this list, in other words if such a list is described by a metavariable, for example by σ as above.

In type theory based proof-assistant, function evaluation is performed by term reduction. Term reduction changes the term being studied, only if the current data matches one of the reduction rules. For instance, $[\![a_1 + a_2, \mathsf{T}'_{\mathsf{lookup}}, \mathsf{T}'_{\mathsf{AExp}}]\!]_\sigma$ reduces to $[\![a_1,\ \mathsf{T}'_{\mathsf{lookup}},\ \mathsf{T}'_{\mathsf{AExp}}]\!]_\sigma\ +$ $[\![a_2,\ \mathsf{T}'_{\mathsf{lookup}},\ \mathsf{T}'_{\mathsf{AExp}}]\!]_\sigma$. But reduction does not occur if one of the expression is represented by a universally quantified variable, for instance $[\![a, \mathsf{T}'_{\mathsf{lookup}}, \mathsf{T}'_{\mathsf{AExp}}]\!]_\sigma$ stays $[\![a, \mathsf{T}'_{\mathsf{lookup}}, \mathsf{T}'_{\mathsf{AExp}}]\!]_\sigma$.

The context can still contain enough information related to these metavariables to execute instructions, consider lemma 4. Therefore we need a way to reason about them. The solution is to associate a number to metavariables like σ. We do it in a systematic way. We

define a new inductive type called n_State (to read as *named state*) which contains an extra constructor for these numbered terms.

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

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

$$
\begin{array}{l}
\mathsf{n\_AExp}: \mathbf{Set} \\
\mathsf{n\_Loc}(\cdot): \mathbb{Z} \to \mathsf{n\_AExp} \\
\mathsf{n\_Num}(\cdot): \mathbb{Z} \to \mathsf{n\_AExp} \\
(\cdot \ +_{\mathsf{n}} \ \cdot): \mathsf{n\_AExp} \to \mathsf{n\_AExp} \to \mathsf{n\_AExp} \\
\quad \vdots \\
\mathsf{metavariable_a} : \mathbb{N} \to \mathsf{n\_AExp}
\end{array}
$$

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

Once we assigned numbers to metavariables we need to keep track of them. We do so by creating few more tables where we have entries only for metavariables as in other cases it's easy to get back the original. We have four such tables, namely, $\mathsf{T_{State}}, \mathsf{T_{AExp}}, \mathsf{T_{BExp}}$ and $\mathsf{T_{Inst}}$ for metavariables representing states, arithmetic expressions, boolean expressions and instructions, respectively. After we have assigned names to all arithmetic and boolean expressions, instructions and states, to work with the metavariables we define new result tables, $\mathsf{T'_{n\_lookup}}, \mathsf{T'_{n\_update}}, \mathsf{T'_{n\_AExp}}, \mathsf{T'_{n\_BExp}}$ and $\mathsf{T'_{n\_Inst}}$, that have a similar role and structure to the result tables, introduced in section 3, but contain named expressions.

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

$$
\begin{array}{l}
\mathcal{C}_{\mathsf{n\_AExp}}(\cdot): (\mathsf{list\ State} * \mathsf{n\_State}) \to (\mathsf{list\ AExp} * \mathsf{n\_AExp}) \\
\qquad\qquad \to (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}) \to \mathbf{Prop} \\
\lambda \mathsf{T_{State}}: \cdots, \lambda \mathsf{T_{AExp}}: \cdots \\
[]^{\mathsf{r}}_{\mathsf{n\_AExp}}: \mathsf{True}
\end{array}
$$

$$
[(\mathsf{n\_}\sigma,\ \mathsf{n\_a},\ \mathsf{n}),\ \mathsf{T'_{n\_AExp}}]^{\mathsf{r}}_{\mathsf{n\_AExp}}: \begin{cases}
\langle a, \sigma \rangle_{\mathsf{A}} \rightsquigarrow n \wedge \mathcal{C}_{\mathsf{n\_AExp}}(\mathsf{T_{State}},\ \mathsf{T_{AExp}},\ \mathsf{T'_{n\_AExp}}) \\
\qquad \text{if } [\![n\_\sigma, \mathsf{T_{State}}]\!] \rightarrowtail \sigma \ \&\ [\![n\_a, \mathsf{T_{AExp}}]\!] \rightarrowtail \mathsf{a} \\
\mathsf{False} \\
\text{if } [\![n\_\sigma, \mathsf{T_{State}}]\!] \rightarrowtail \mathsf{error} \text{ or } [\![n\_a, \mathsf{T_{AExp}}]\!] \rightarrowtail \mathsf{error}
\end{cases}
$$

In the above, $[\![n\_\sigma,\ \mathsf{T_{State}}]\!] \rightarrowtail \sigma$ is the *translation* from the named state to state. Similarly we need to translate arithmetic expressions, boolean expressions and instructions. We change the consistency functions for lookup in a state, arithmetic expressions, boolean expressions and instructions as well. They are similar and we omit their detail.

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

$$
\begin{array}{l}
[\![\cdot]\!] \colon \mathsf{n\_AExp} \to \mathsf{n\_State} \to (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z}) \\
\qquad\qquad \to (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}) \to (\mathsf{option}\ \mathbb{Z}) \\
\lambda \mathsf{n\_a} \colon \ldots \lambda \mathsf{n\_}\sigma \colon \ldots \lambda \mathsf{T^r_{n\_lookup}} \colon \ldots \lambda \mathsf{T^r_{n\_AExp}} \colon \ldots \\
\\
\mathsf{x}\ \text{ where } (\mathsf{n\_}\sigma,\ \mathsf{n\_a},\ \mathsf{x}) \in \mathsf{T^r_{n\_AExp}} \\
\text{Otherwise:} \\
[\![\mathsf{n\_Num(n)},\ \mathsf{T^r_{n\_lookup}},\ \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} := n \\
[\![\mathsf{n\_Loc(v)},\ \mathsf{T^r_{n\_lookup}},\ \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} := \mathsf{n\_flookup}(\mathsf{n\_}\sigma, \mathsf{v}, \mathsf{T^r_{n\_lookup}}) \\
[\![n\_a_1 +_n n\_a_2, \mathsf{T^r_{n\_lookup}}, \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} := \left\{ \begin{array}{ll}
[\![n\_a_1, \mathsf{T^r_{n\_lookup}}, \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} & \\
\quad + [\![n\_a_2, \mathsf{T^r_{n\_lookup}}, \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} & \\
\qquad\qquad \text{if} \quad [\![n\_a_1, \cdots]\!]_{n\_\sigma} \neq \mathsf{error} & \\
\qquad\qquad \& \quad [\![n\_a_2, \cdots]\!]_{n\_\sigma} \neq \mathsf{error} & \\
\mathsf{error} & \\
\qquad\qquad \text{if} \quad [\![n\_a_1, \cdots]\!]_{n\_\sigma} = \mathsf{error} & \\
\qquad\qquad \text{or} \quad [\![n\_a_2, \cdots]\!]_{n\_\sigma} = \mathsf{error} &
\end{array} \right. \\
[\![n\_a_0 -_n n\_a_1,\ \mathsf{T^r_{n\_lookup}},\ \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} := \cdots \\
[\![n\_a_0 *_n n\_a_1,\ \mathsf{T^r_{n\_lookup}},\ \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} := \cdots \\
[\![\mathsf{metavariable_a(n)},\ \mathsf{T^r_{n\_lookup}},\ \mathsf{T^r_{n\_AExp}}]\!]_{n\_\sigma} := \mathsf{error}
\end{array}
$$

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

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

Theorem 1.
$\forall\ \mathsf{T_{State}} \colon\ (\mathsf{list\ State} * \mathsf{n\_State}).\forall\ \mathsf{T^r_{n\_lookup}} \colon\ (\mathsf{list\ n\_State} * \mathbb{Z} * \mathbb{Z}).$
$\forall\ \mathsf{T_{AExp}} \colon\ (\mathsf{list\ AExp} * \mathsf{n\_AExp}).\ \forall\ \mathsf{T^r_{n\_AExp}} \colon\ (\mathsf{list\ n\_State} * \mathsf{n\_AExp} * \mathbb{Z}).$
$\forall\ \mathsf{n\_}\sigma \colon \mathsf{n\_State}.\forall\ \mathsf{n\_a} \colon \mathsf{n\_AExp}.\forall\ \sigma \colon \mathsf{State}.\forall\ \mathsf{a} \colon \mathsf{AExp}.\forall\ \mathsf{n} \colon \mathbb{Z}.$

$$
\begin{array}{l}
\mathcal{C}_{\mathsf{n\_lookup}}(\mathsf{T_{State}},\ \mathsf{T^r_{n\_lookup}}) \to \mathcal{C}_{\mathsf{n\_AExp}}(\mathsf{T_{State}},\ \mathsf{T_{AExp}},\ \mathsf{T^r_{n\_AExp}}) \\
\qquad \to [\![\mathsf{n\_}\sigma,\ \mathsf{n\_a}, \mathsf{T^r_{n\_lookup}},\ \mathsf{T^r_{n\_AExp}}]\!]\ =\ \mathsf{n} \\
\qquad\qquad \to [\![\mathsf{n\_}\sigma,\ \mathsf{T_{State}}]\!] \rightarrowtail \sigma \to [\![\mathsf{n\_a},\ \mathsf{T_{AExp}}]\!] \rightarrowtail \mathsf{a}
\end{array}
$$

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

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

4.2 The iteration technique

We cannot define the evaluation function for instructions in the similar way as we did for arithmetic and boolean expressions, since execution of instruction (in particular *while loop*) does not follow structural recursion. In our previous work [2], we presented *the iteration technique* to work around this problem. We give a short account here, for more details we suggest to read the original work.

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

$$\begin{array}{l}
\mathsf{F}\colon (\mathsf{n_Inst} \rightarrow \mathsf{n_State} \rightarrow (\mathsf{list\ n_State} * \mathbb{Z} * \mathbb{Z}) \rightarrow (\mathsf{list\ n_State} * \mathbb{Z} * \mathbb{Z} * \mathsf{n_State}) \\
\qquad\qquad \rightarrow (\mathsf{list\ n_State} * \mathsf{n_AExp} * \mathbb{Z}) \rightarrow (\mathsf{list\ n_State} * \mathsf{n_BExp} * \mathbb{B}) \\
\qquad\qquad \rightarrow (\mathsf{list\ n_State} * \mathsf{n_Inst} * \mathsf{n_State}) \rightarrow (\mathsf{option\ n_State})) \\
\quad \rightarrow \mathsf{n_Inst} \rightarrow \mathsf{n_State} \rightarrow (\mathsf{list\ n_State} * \mathbb{Z} * \mathbb{Z}) \rightarrow (\mathsf{list\ n_State} * \mathbb{Z} * \mathbb{Z} * \mathsf{n_State}) \\
\quad \rightarrow (\mathsf{list\ n_State} * \mathsf{n_AExp} * \mathbb{Z}) \rightarrow (\mathsf{list\ n_State} * \mathsf{n_BExp} * \mathbb{B}) \\
\quad \rightarrow (\mathsf{list\ n_State} * \mathsf{n_Inst} * \mathsf{n_State}) \rightarrow (\mathsf{option\ n_State}) \\
\lambda f\colon \ldots \lambda n_i\colon \ldots \lambda n_\sigma\colon \ldots \\
\lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{n_lookup}}\colon \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{n_update}}\colon \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{n_AExp}}\colon \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{n_BExp}}\colon \ldots \lambda \mathsf{T}^{\mathsf{r}}_{\mathsf{n_Inst}}\colon \ldots
\end{array}$$

$$\begin{array}{l}
n_\sigma' \ \text{ where } (n_\sigma,\ n_i,\ n_\sigma') \in \mathsf{T}^{\mathsf{r}}_{\mathsf{n_Inst}} \\
\mathsf{Otherwise}\colon \\
\quad (\mathsf{F}\ f\ \mathsf{skip}_n\ n_\sigma\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_lookup}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_update}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_AExp}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_BExp}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_Inst}}) := n_\sigma \\
\qquad \vdots \\
\quad (\mathsf{F}\ f\ (\mathsf{while}\ n_b\ \mathsf{do}\ n_i)_n\ n_\sigma\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_lookup}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_update}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_AExp}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_BExp}}\ \mathsf{T}^{\mathsf{r}}_{\mathsf{n_Inst}}) \\
\qquad := \begin{cases}
(f\ \ (\mathsf{while}\ n_b\ \mathsf{do}\ n_i)_n\ \ (f\ n_i\ n_\sigma\ \cdots)\ \cdots) & \\
\qquad\qquad \text{if } [\![n_b, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_lookup}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_AExp}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_BExp}}]\!]_{n_\sigma} = \mathsf{true} & \\
\qquad\qquad\ \ \& \ \ (f\ n_i\ n_\sigma\ \cdots) \neq \mathsf{error} & \\
n_\sigma \qquad \text{if } [\![n_b, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_lookup}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_AExp}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_BExp}}]\!]_{n_\sigma} = \mathsf{false} & \\
\mathsf{error} \qquad \text{if } [\![n_b, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_lookup}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_AExp}}, \mathsf{T}^{\mathsf{r}}_{\mathsf{n_BExp}}]\!]_{n_\sigma} = \mathsf{error} & \\
\qquad\qquad \text{or } (f\ n_i\ n_\sigma\ \cdots) = \mathsf{error} & \\
\qquad\qquad \text{or } (f\ \ (\mathsf{while}\ n_b\ \mathsf{do}\ n_i)_n\ \ (f\ n_i\ n_\sigma\ \cdots)\ \cdots) = \mathsf{error}
\end{cases} \\
\quad (\mathsf{F}\ f\ (\mathsf{metavariable_i(n)})_n\ n_\sigma\ \cdots) := \mathsf{error}
\end{array}$$

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

Intuitively, writing the function F is exactly the same as writing the recursive function, except that recursive calls are simply replaced by a bound variable (here f) in recursive calls.

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

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

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

Theorem 2.
$\forall\ \mathsf{T_{State}}$: (list State $*$ n_State).$\forall\ \mathsf{T^r_{n\_lookup}}$: (list n_State $*\ \mathbb{Z} * \mathbb{Z}$).
$\forall\ \mathsf{T^r_{n\_update}}$: (list n_State $*\ \mathbb{Z} * \mathbb{Z} *$ n_State).
$\forall\ \mathsf{T_{AExp}}$: (list AExp $*$ n_AExp). $\forall\ \mathsf{T^r_{n\_AExp}}$: (list n_State $*$ n_AExp $*\ \mathbb{Z}$).
$\forall\ \mathsf{T_{BExp}}$: (list BExp $*$ n_BExp). $\forall\ \mathsf{T^r_{n\_BExp}}$: (list n_State $*$ n_BExp $*\ \mathbb{B}$).
$\forall\ \mathsf{T_{Inst}}$: (list Inst $*$ n_Inst). $\forall\ \mathsf{T^r_{n\_Inst}}$: (list n_State $*$ n_Inst $*$ n_State).
$\forall\ \mathsf{k}: \mathbb{N}.\forall\ \mathsf{n\_}\sigma, \mathsf{n\_}\sigma'$: n_State.$\forall\ \mathsf{n\_i}$: Inst.$\forall\ \sigma$: State.$\forall\ \mathsf{i}$: Inst.

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

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

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

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

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

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

5 Conclusions

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

could see that the functional approach carries over nicely to larger programming language, even though semantics requires mutual inductive propositions.

In the literature we found work by Nancy A. Day and Jeffrey J. Joyce where they discuss about *Symbolic Functional Evaluation*[4]. But this work is different from ours, as in their work symbolic functional evaluation is an algorithm for executing functional programs to evaluate expressions in higher order logic. It carries out the logical transformations of expanding definitions, beta-reduction and simplification of built-in constants in the presence of uninterpreted constants and quantifiers. They suggest different levels of evaluation for such an algorithm to terminate while evaluating the arguments of uninterpreted functions. However this kind of capability already exists in the **Coq** proof-assistant, where the tactic *Simpl* does a similar work. In our work we showed a way to reason on metavariables. One can have some information related to uninterpreted symbols and use it in an intelligent way.

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

References

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

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

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

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

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

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

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

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

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

[10] 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