Proof of translation in natural semantics
Joelle Despeyroux

To cite this version:

HAL Id: inria-00076040
https://hal.inria.fr/inria-00076040
Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
PROOF OF TRANSLATION
IN NATURAL SEMANTICS

Joëlle DESPEYROUX

Avril 1986
PROOF OF TRANSLATION IN NATURAL SEMANTICS

Joëlle Despeyroux

INRIA, Sophia-Antipolis
Route des Lucioles, 06565 Valbonne Cedex, France

Abstract

We have retained the purely 'formal system' part of the structural operational semantics 'a la Plotkin', and have named that view of it 'natural semantics'. Numerous examples written in this semantics have been entered in the meta-compiler Mentor + Typol. We show here that natural semantics enables us in a single formalism, to define dynamic semantics and translation of languages and to prove the correctness of a translation. Our criterion of correctness is the validity of two inference rules in a theory. Proofs are made by induction on the length of the proof. We illustrate the method on an example, treated in full: translation from Mini-ML into CAM (Categorical Abstract Machine).

Résumé

Nous avons retenu la partie purement "système formel" de la sémantique operationelle structurelle 'a la Plotkin', et avons appelé cette vue de celle-ci "sémantique naturelle". De nombreux exemples écrits en sémantique naturelle ont été entrés dans le méta-compilateur Mentor + Typol. Nous montrons ici que la sémantique naturelle nous permet dans un seul formalisme, de définir des sémantiques dynamiques et des traductions et de prouver la correction d'une traduction. Notre critère de correction est la validité de deux règles d'inference dans une théorie. Les preuves sont faites par induction sur la longueur de la preuve. Nous illustrons la méthode sur un exemple, complètement traité: la traduction de Mini-ML dans la CAM (Categorical Abstract Machine).
PROOF OF TRANSLATION IN NATURAL SEMANTICS

Joëlle Despeyrouz

INRIA, Sophia-Antipolis
Route des Lucioles, 06565 Valbonne Cedex, France

Abstract

We have retained the purely ‘formal system’ part of the structural operational semantics ‘à la Plotkin’, and have named that view of it ‘natural semantics’. Numerous examples written in this semantics have been entered in our meta-compiler Mentor + Typol. We show here that natural semantics enables us in a single formalism, to define dynamic semantics and translation of languages and to prove the correctness of a translation. Our criterion of correctness is the validity of two inference rules in a theory. Proofs are made by induction on the length of the proof. We illustrate the method on an example, treated in full: translation from Mini-ML into CAM (Categorical Abstract Machine).

1. Introduction

1.1. Natural Semantics

The natural semantics of a language is given by a formal system (a set of axioms and inference rules) which defines a set of valid theorem (a theory). Theorems of interest are, for example:

\[ \vdash P : \alpha \quad \text{P executes to } \alpha \]
\[ \vdash P \in \pi \quad \text{type of P is } \pi \]
\[ \vdash P \rightarrow P' \quad \text{P translates to } P' \]

The name natural comes from the fact that this system is given in the Gentzen’s system style [5][1], in which we can make natural deduction 1 [17] [6]. Furthermore, semantics written in this style appears to be rather intuitive, so that natural may also be understood in the lay-man’s sense. Natural semantics has its origin in the structural operational semantics most fully developed in [15]. But we focus on the pure logical part of it. Note that natural semantics is not intrinsically operational (for us “→” simply denotes a predicate, and not a transition of an abstract machine), and can even be non structural (see later on the usual semantics of application in ML).

1 We use natural deduction in its first meaning in that we don’t always write an introduction rule and an elimination rule for each constructor, so we don’t have a notion of normal proof, in dynamic semantics at least.

Numerous examples, in static semantics, dynamic semantics and translation, have been written in natural semantics and are presented in [7]. All these examples have been compiled and so have mechanically generated running type-checkers, interpreters and translators (compilers). The programming language supporting natural semantics is called (for historical reasons) Typol. The development of Typol is an extension of the work on the syntactic meta-editor Mentor [14]. It is not our purpose here to describe this language [3][4]. It is sufficient to say that a Typol program is a first order logic whose terms are abstract syntax trees (which may be graphs as we shall see later on).

1.2. Proof of translation

Our purpose here is to show that natural semantics enables us (in a single formalism) to define dynamic semantics and translations, and to prove the correctness of these translations. Let’s recall the usual diagram L. Morris:

\[
\begin{align*}
L_1 & \xrightarrow{T} L_2 \\
SD(L_1) & \xrightarrow{t} SD(L_2)
\end{align*}
\]

where the dynamic semantics of \( L_i \) is given by a semantic domain \( SD(L_i) \) and a semantic mapping \( L_i.DS \) from objects (programs) of \( L_i \) into values of \( SD(L_i) \). The mapping \( T \) is the translation of programs and \( t \) is the translation of semantics values. In our context semantic domains are integers, booleans, closures..., and all (four) arrows in the diagram are predicates described by a formal system. \( L_1.DS, L_2.DS \) and \( T \) must be disjoint and \( t \) may use \( T \). Note that these theories have the same language (Typol). The key-idea is to consider the formal system:

\[
\mathcal{T} = T \cup L_1.DS \cup L_2.DS \cup t
\]

We have (three or) four distinguished sets of rules in \( \mathcal{T} \). Each sequent makes appeal to one particular set of rules, as it is in fact always the case in a Typol program: This is no more than a notion of modularity in our logic. Since these theories have no connection we could not have altered them in the join operation. We may only have added new facts... We shall say that the translation is correct iff [those
facts are wanted facts] two certain inference rules are valid in $T$. The proof will use induction on the length of the proof.

Outline.

To provide an illustration of the method, we work on a specific example. Part 2 of the paper gives the dynamic semantics of Mini-ML (the purely applicative part of ML). Part 3 gives the dynamic semantics of CAM: "categorical abstract machine", a very interesting machine language developed by G. Cousineau and P.L. Curien [2]. Part 4 gives the translation of Mini-ML into CAM. We develop in Part 5 a notion of "approximate normal form" of a program (inspired by [18][9]), which will enable us to say that the criterion of correctness of translation given above deals with infinite programs as well. Part 6 introduces the method for proving the correctness of the translation. Finally part 7 gives the complete proof for our example: Mini-ML to CAM.

A complete proof of Mini-ML to CAM can be found in [11]. But this proof is done in a completely different context and style: M. Mauny proved that the CAM machine correctly simulates the $\beta$-reduction of the $\lambda$-calculus, by induction on the length of the transitions of the machine. Also W. Li [8] deals with correctness of translation, but he is interested in concurrent languages, and, taking the operational semantics 'a la Plotkin', considers equivalences of behaviours of transitions systems.

2. Dynamic semantics of Mini-ML

ML is a functional language with polymorphism and higher-order functions, first used in the proof system LCF [10]. Mini-ML consist in the purely applicative part of ML, more precisely a simple typed $\lambda$-calculus with constants, products, conditionals, and recursive function definitions. The abstract syntax of Mini-ML is given in Fig. 1. Simple programs in Mini-ML are for example, in concrete syntax, a term with both simultaneous definitions and block structure, or a simultaneous recursive definition:

\[
\text{let } (x,y) = (2,3) \\
\text{in } \text{let } (x,y) = (y,x) \text{ in } x
\]

\[
\text{letrec } (\text{even}, \text{odd}) = (\lambda x. \text{if } x = 0 \text{ then true else odd}(x - 1), \\
(\lambda x. \text{if } x = 0 \text{ then false else even}(x - 1)) \\
\text{in } \text{even } 3
\]

2.1. The formal semantics of Mini-ML

Because of higher-order functions, the domain of semantic values of Mini-ML is slightly more complicated than for a less expressive language:

- integers $\mathbb{N}$
- truth values: $true$, $false$

- closures: $[\lambda \text{P.E.}, \rho]$, where $E$ is an expression and $\rho$ is an environment. A closure is just a pair of a $\lambda$-expression and an environment.

- identifiers for predefined operators: $\text{plus}$, ...

- pairs of semantic values (which may in turn be pairs, so lists of semantic values maybe constructed)

Naturally the value of an expression $e$ depends on the values of the identifiers that occur free in it. An environment $\rho$ is an ordered list of pairs $P \mapsto \alpha$ where $P$ is a pattern and $\alpha$ a value. Here is an example of environment:

\[
x \mapsto 1 \cdot (x,y) \mapsto (true,5).
\]

We say that expression $e$ evaluates to $\alpha$ in environment $\rho$ if the theorem

\[
\rho \vdash e : \alpha
\]

can be derived from the formal system in Fig. 2.

2.2. Comments on the formal definition

In Figure 2, rules 1 to 3 associate values to integer or boolean literals. Rule 4 says that the evaluation of an expression begins with an initial environment (mapping a few predefined operators to "themselves"). Rule 4 constructs a closure for a $\lambda$-expression, pairing it with the environment. The value associated to an identifier must be looked up in the environment (rule 5). Given that the environment maps patterns to values, rather than identifiers to values, we need auxiliary rules, the set $\text{VAL-OF}$. Rules 6 and 7 associate values to conditional expression. Rule 8 is equally transparent.

The next rules deal with functional values. Rule 9 is a special case of rule 10, where $E_1$ evaluates to a predefined operator. Rule 10 is the general case of the evaluation of an application. Because of type-checking, the operator of an application can only evaluate to a functional value, i.e. a closure. This closure is taken apart, and its body is evaluated in its environment, prefixed with the parameter association $P \mapsto \alpha$. Note that the rule is valid whether $P$
program ML_DS is
use ML
ρ, ρ₁ : PAT;
α, β : VALUE;

\[
\begin{align*}
\text{init_env} \vdash E : \alpha & \quad (i) \\
\rho \vdash E : \alpha & \quad (1) \\
\rho \vdash \text{number } N : N & \quad (2) \\
\rho \vdash \text{true : true} & \quad (2) \\
\rho \vdash \text{false : false} & \quad (3) \\
\rho \vdash \lambda P.E : (\lambda P.E, \rho) & \quad (4) \\
\rho \vdash \text{ident } t \mapsto \alpha & \quad (5) \\
\rho \vdash \text{ident } t \mapsto \alpha & \quad (5) \\
\rho \vdash E₁ : \text{true} & \rho \vdash E₂ : \alpha & \quad (6) \\
\rho \vdash E₁ \text{ if } E₂ \text{ then } E₃ \text{ else } E₄ : \alpha & \quad (7) \\
\rho \vdash E₁ : \text{false} & \rho \vdash E₂ : \alpha & \quad (7) \\
ρ \vdash E₁ : \alpha & ρ \vdash E₂ : β & \quad (8) \\
ρ \vdash (E₁, E₂) : (\alpha, β) & \quad (8) \\
ρ \vdash E₁ : \text{ident } OP & ρ \vdash E₂ : (\alpha, β) & \quad \text{eval} & \quad (9) \\
ρ \vdash E₁ : E₂ : γ & \quad (9) \\
ρ \vdash E₁ : (\lambda P.E, ρ₁) & ρ \vdash E₂ : \alpha & ρ \vdash E₁ \rightarrow \alpha \cdot ρ₁ \rightarrow E : β & \quad (10) \\
ρ \vdash E₁ : E₂ : β & \quad (10) \\
ρ \vdash E₁ : \alpha & \rho \vdash E₁ \rightarrow \alpha \cdot ρ \vdash E₂ : β & \quad (11) \\
ρ \vdash \text{let } P = E₂ \text{ in } E₁ : \beta & \quad (11) \\
ρ₁ \rightarrow P \rightarrow \text{letrec } P = E₂ \text{ in } E₁ : \alpha & \quad (12) \\
\end{align*}
\]

end ML_DS

because in rules 10 to 12 we have merely prefixed the environment with new associations.

\[
\begin{align*}
\text{set VAL_OF is} & \\
\text{ident } t \mapsto \alpha \cdot ρ \vdash \text{ident } t \mapsto \alpha & \quad (1) \\
ρ \vdash \text{ident } t \mapsto \alpha & \quad (x \neq 1) \quad (2) \\
\rho \vdash \text{ident } x \mapsto β \cdot ρ \vdash \text{ident } t \mapsto \alpha & \quad (3) \\
(P₁, P₂) \mapsto (\alpha, β) \cdot ρ \vdash \text{ident } t \mapsto γ & \quad (3) \\
(P₁, P₂) \mapsto \text{letrec } P₁ = E₁, P₂ = E₂ \text{ in } E₁ : \alpha & \quad (4) \\
\end{align*}
\]

end VAL_OF

Figure 2. The dynamic semantics of Mini-ML

is a pattern or a single variable. As, at evaluation time, \( \lambda P.E₂ \) \( E₃ \) \( \{ \text{let } P = E₂ \text{ in } E₁ \} \), rule 11 appears to be an other optimisation of rule 10.

The last rule, rule 12, defines in one and the same way the simple recursive functions and the mutually recursive ones. The environment in which \( E₁ \) is evaluated is prefixed with a self-referencing closure. Notice that since \( ρ \vdash E₂ : \alpha \) is a premise of rule 10, we have an ML with call by value.

The separate set VAL_OF (see Fig. 3) defines rules to associate values to identifiers, given some environment. Since the environment maps patterns to values, the patterns must be traversed to find the relevant identifier. Furthermore, block structure is present in the environment.

2 Optimisations of both dynamic semantics of Mini-ML and translation from Mini-ML to CAM are presented and prooved correct in [12].

Rules 1 and 2 scan the environment until the first occurrence of an identifier is found, in a left to right scan. Rule 3 relies on the fact that, except for the case taken care of in rule 4, a pair of identifiers is bound to a value which is a pair. Hence searching is propagated to two new pattern-value pairs. Rule 4 takes care of the mutually recursive definitions. When a pair of patterns is associated to a single closure, this closure must come from a pair of functions. The environment of the closure is distributed over these functions and searching is propagated to simpler components. Thanks to this simple idea, the letrec rule 12 remains transparent, while accessing the environment is made only slightly more complex.

3. Dynamic semantics of CAM

The Categorical Abstract Machine [2] has its roots both in categories and in De Bruijn's notation for lambda-calculus. It is a very simple machine where, according to its inventors, "categorical terms can be considered as code acting on a graph of values". Instructions are few in number and quite close to real machine instructions. Instructions car and cdr serve in accessing data in the stack and the special instruction rplac is used to implement recur- 10

3.1. The formal semantics of CAM

The state of the CAM machine is a stack, whose top element may be viewed as a register. The values stored in this stack are:

- integers \( \mathbb{N} \)
- truth values: \( \text{true, false} \)
- closures of the form \([c, ρ]\), where \( c \) is a fragment of CAM code and \( ρ \) is a value, meant to denote an environment
sorts VALUE, COM, PROGRAM, COMS
subsorts COMS
constructors
  program : COMS → PROGRAM
  coms : COM* → COMS
  quote : VALUE → COM
  op, car, cdr, cons : → COM
  push, swap, app, rplac : → COM
  cur : COMS → COM
  branch : COMS × COMS → COM
  int, bool, null, value : → VALUE

Figure 4. Abstract syntax of CAM code

- pairs of semantic values (which may in turn be pairs, so that trees may be constructed)

Except in the first rule, all sequents have the form

\[ s ⊢ c : s' \]

where \( c \) is CAM-code and \( s \) and \( s' \) are states of the CAM machine. The sequent \( s ⊢ c : s' \) may be read as executing code \( c \) when the machine is in state \( s \) takes it to state \( s' \).

The rules describing the transitions of the CAM appear in Fig. 5.

3.2. Comments on the formal definition

In Figure 5, rule 1 says that evaluating a program begins with an initial stack and ends with a value on top of the stack that is the result of the program. The initial stack contains closures corresponding to the predefined operators. Rules 2 and 3 deal with sequences of commands; rules 4 to 11 are self-explanatory axioms. Rule 12 switches to an external evaluator EVAL for predefined operators.

Rule 13 and 14 define the branch instruction. It takes its (evaluated) condition from the top of the stack, and continues with either the true or the false part. The cur instruction is described in rule 15: cur(c) builds a closure with the code c and the current environment (top of the stack) placing it on top of the stack. Rule 16 says that the app instruction must find on top of the stack a pair consisting of a closure and a parameter environment. Then the code of the closure is evaluated in a new environment: that of the closure prefixed by the parameter environment.

The last rule is the less intuitive one. An rplac instruction takes a pair consisting of an environment \( ρ \) and a variable \( ν \), followed by an environment \( ρ_1 \) on the stack. It identifies \( ν \) and \( ρ_1 \) and places the pair \( (ρ, ρ_1) \) on the stack. Notice that each occurrence of \( ν \) in \( ρ_1 \) has been replaced by \( ρ_1 \). The use of this instruction will be explained by the translation of the letrec instruction (see rule 9 on Fig. 6).

Figure 5. The definition of the CAM

4. Translation from Mini-ML to CAM

We are now ready to generate CAM code for mini-ML.

4.1. The formal system

The translation rules from mini-ML to CAM are given in Fig. 6. In these rules, except for rule 1, all sequents have the form:

\[ ρ ⊢ e → c \]

where \( ρ \) is an environment, \( e \) is an ML expression, and \( c \) is its translation into CAM-code. In words, the sequent may
be read as in environment \( \rho \), expression in is compiled into code \( c \). The notion of environment used in this translation is exactly the notion of an ML-pattern, i.e. a binary tree with identifiers at the leaves.

**program ML.CAM is**

**use ML**

**use CAM**

c, c1, c2, c3 : CAM;

\[
\begin{align*}
\rho, \rho_1 : ENV; \\
\text{init:\_pat} \vdash E \rightarrow c \\
\downarrow \quad E \rightarrow \text{program}(c) \\
\rho \vdash \text{number} n \rightarrow \text{quote}(\text{int} n) \\
\rho \vdash \text{true} \rightarrow \text{quote}(\text{bool} "\text{true}" ) \\
\rho \vdash \text{false} \rightarrow \text{quote}(\text{bool} "\text{false}" ) \\
\rho \vdash \text{access} \vdash \text{ident} i \rightarrow c \\
\rho \vdash \text{ident} i \rightarrow c \\
\rho \vdash \text{if} E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow \text{push} ; c_1 ; \text{branch}(c_2 , c_3) \\
\rho \vdash \text{let} \ p = E_1 \text{ in } E_2 \rightarrow \text{push} ; c_1 ; \text{cons} ; c_2 \\
\rho \vdash \text{letrec} \ p = E_1 \text{ in } E_2 \rightarrow \{ \text{push} ; \text{quote}(\rho_1) ; \\
\text{cons} ; \text{push} ; c_1 ; \text{swap} ; c_2 ; \text{rplac} ; c_3 \} \\
\rho \vdash \text{app} \vdash \text{AP} . F \rightarrow \text{cur}(F) \\
\rho \vdash E_2 \rightarrow c_2 \\
\text{trans\_const} \vdash E_1 \rightarrow c_1 \\
\rho \vdash E_1 \rightarrow c_1 \\
\rho \vdash E_2 \rightarrow c_2 \\
\rho \vdash E_2 \rightarrow \text{push} ; c_1 ; \text{swap} ; c_2 ; \text{cons} ; \text{app}
\end{align*}
\]

**end ACCESS**

**Figure 7. Generating access paths for identifiers**

Rules 2, 3, and 4 generate code for literal values. Rule 5 generates an access path for an identifier. Rules 6 and 7 are straightforward once the following inductive assertion is understood: the code for an expression expects its evaluation environment on top of the stack, and it will overwrite this environment with its result. Thus the environment must be saved, by a push instruction, when necessary.

Rule 8 shows how a run time environment is built up in the stack in parallel with the static environment. Rule 9 is a little surprising because it leaves a free variable \( \rho_1 \) in the code. This is a technique for leaving a reference to be resolved at run time. The instruction \( \text{quote}(\rho_1) \) will leave (at execution time) a free variable on top of the stack. A closure will be built using the environment on top of the stack. Hence this closure will refer to variable \( \rho_1 \). Instruction \( \text{rplac} \) will tie a knot, freezing the value of \( \rho_1 \) as the appropriate closure.

In this way, we build a self-referencing environment.

The remaining rules deal with closures. Rule 10 merely generates the instruction \( \text{cur} \) that constructs closures. Rule 11 concerns predefined operators. Finally, rule 12 is the general case for an application.

**Figure 6. Translation from mini-ML to CAM**

**4.2. Comments on the formal system**

Translation of an ML program is invoked, in rule 1, with an initial environment \( \text{init\_pat} \) that is merely a list of predefined functions. The environment builds up whenever one introduces new names (rules 9 and 10). It is consulted when one wants to generate code for an identifier (rule 5). Then an access path is computed in the ACCESS rule set (see Fig 7). The access path is a sequence of car and cdr instructions (a coding of the De Bruijn number associated to that occurrence of the identifier) that will access the corresponding value in the stack of the CAM.
Definition 5.1. An approximate normal form (a.n.f.) of Mini-ML is either
- ⊥,
- a constant (integer, boolean, predefined identifier or closure), or
- a pair of a.n.f.
and can only be obtained by this recursive definition.

Now, we add an axiom in our theory ML,DS, saying that the evaluation of an expression may return ⊥:
\[ \rho \vdash e : \bot \]
and we define the set of approximate normal forms an expression:

Definition 5.2. \( \alpha \) is an approximate normal form (a.n.f.) of \( t \) if
- \( \alpha \) is an a.n.f.
- \( \vdash t : \alpha \) holds

Now, an infinite ML term as no semantic value - ML differs here from the \( \lambda \)-calculus - but an infinite set of a.n.f. For example, let \( \text{rec} F = \lambda x. [x.F x] \) in \( F \) as for a.n.f.:
\[ \bot \quad \{ \bot, \bot \} \quad \{ \bot, \bot, \bot \} \quad \{ \bot, \bot, \bot, \bot \} \quad \cdots \]

5.2. Discussion on ML

Using the method developed in the previous subsection, our language ML become nondeterministic: for each expression we have the choice to evaluate it or not. But this is not truly nondeterminism as we shall now see.

Definition 5.3. We define a partial order \( \preceq \) on a.n.f. as follows: For all \( e_i \) constant, \( \varphi, \varphi_i, \varphi_j \) a.n.f.:
- \( e_i \preceq e_j \iff i = j \)
- \( \bot \preceq \varphi \)
- \( (\varphi_i, \varphi_j) \preceq (\varphi_i', \varphi_j') \iff \varphi_i \preceq \varphi_i' \quad \& \quad \varphi_j \preceq \varphi_j' \)

Fact. \( \Phi = \{ \{ \text{a.n.f.}, \preceq \} \} \) can be embedded in a cpo, by considering the set of directed sets of \( \Phi \), with the induced partial order, then identifying each a.n.f. \( \alpha \) with the set of a.n.f. dominated by \( \alpha \).

Definition 5.4. From the partial order on a.n.f. we induce a partial order on environments, defined by extension:
- \( \emptyset \preceq \emptyset \)
- \( P \preceq \rho \preceq P \preceq \alpha \cdot \rho' \iff \alpha \preceq \alpha' \quad \& \quad \rho \preceq \rho' \)

Theorem 5.1. For each ML term \( t \), the set of a.n.f. of \( t \) is a directed set.

Proof. Prove that for any \( t, \rho_1, \rho_2, \rho_\alpha, \beta \), such that \( \rho_1 \vdash t : \alpha, \rho_2 \vdash t : \beta \), \( \rho_1 \preceq \rho_1, \rho_2 \preceq \rho \) there exists \( \gamma \) such that \( \rho \vdash t : \gamma \), \( \alpha \preceq \gamma \), \( \beta \preceq \gamma \). The theorem follows, with \( \rho_1 = \rho_2 = \rho = \emptyset \). The proof use induction on the length of the proof. We do not give it here, as it is very similar to the proof of the correctness of the translation, given in full later on.

Fact. In a cpo, a directed set admits a least upper bound.

So we can define:

Definition 5.5. For each term \( t \), the limit of the set of a.n.f. of \( t \), \( \Phi(t) \), is its least upper bound.

Now it is clear that, in adding the rule \( \rho \vdash e : \bot \), we have not really added non-determinism in our language, as we have a "Church-Rosser property":

\[ \vdash p : \alpha \quad \& \quad \vdash p : \beta \Rightarrow \exists \gamma, \gamma \geq \alpha, \gamma \geq \beta \text{ s.t. } \vdash \gamma \]

Furthermore, we have a limit of the sequence \( \alpha_n \) such that \( \vdash \alpha_n \). This guarantees the existence and unicity of the result, even if it is a limit.

6. Proof of translation

We are now ready to give our criteria of correctness of a translation. The first subsection present those (two) criteria, the second one shows that they are adequate criteria of correctness of a translation, while the last subsection shows that they are equivalent in simple cases.

6.1. General case

We follow here the idea developed in the introduction. We work in a theory

\[ \mathcal{T} = T \cup L_1,DS \cup L_2,DS \cup t \]

and our criteria of correctness of translation are as follows:

Definition 6.1. The translation is correct iff the following inference rules are valid in \( \mathcal{T} \):

\[
\frac{1}{\vdash p : \alpha} \quad \frac{\vdash p \rightarrow p'}{\vdash p : \alpha'} \quad \frac{1}{\vdash \alpha \rightarrow \alpha'} \quad \frac{2}{\vdash p' : \alpha'}
\]

where \( p \) and \( p' \) are source and object programs, \( \alpha \) and \( \alpha' \) are semantic values (or approximate normal forms), and the superscripts of the turnstiles denote the set of rules under consideration.\(^3\)

These inference rules are expressing commutativity of a diagram. In pictures we ask for:

\[
\begin{array}{c}
\overset{p}{\longrightarrow} & \overset{p'}{\longrightarrow} \\
\overset{1}{\downarrow} & \overset{2}{\downarrow} \\
\alpha & \alpha'
\end{array}
\]

completeness soundness

\(^3\) All implicit quantifiers are universal quantifiers.
where $\rightarrow$ denote the given facts and $\rightarrow$ denote the facts to prove.

The second inference rule is somewhat unusual. However, notice that an instance of this rule is not too surprising as a sub-proof-tree. Anyway, a more usual form of this rule would be:

$$
\exists \alpha \quad \frac{\frac{T}{p \rightarrow p'} \quad \frac{\alpha}{p \rightarrow \alpha' \alpha \rightarrow \alpha'}}{\frac{T}{p \rightarrow \alpha' \land \alpha \rightarrow \alpha'}} \quad (2a)
$$

This form is equivalent to the previous one for the theory $1$ and $t$ have no connection. But experience shows that Rule (2) is easier to manipulate, in our context, than Rule (2a).

Note that our criteria of correctness are sufficient for all programs, no matter whether they terminate or not, are erroneous or not: $\alpha$ and $\alpha'$ are approximate normal forms of $p$ and $p'$, or errors.

6.2. Discussion

For the sake of completeness, we shall examine all other possible diagrams which would be chosen as criteria of correctness. We take as hypothesis that $T$ and $t$ are always defined, and there exists an a.n.f. of any source term (which means that all typed-checked programs are executable). We do not consider diagrams which are obviously too strong requirements for the correctness of a translation. The remaining diagrams, apart from (1) and (2), are mainly:

$$
\frac{p \rightarrow p'}{1} \quad \frac{p \rightarrow p'}{1} \quad \frac{p \rightarrow p'}{1} \quad \frac{p \rightarrow p'}{1} \quad \frac{p \rightarrow p'}{1} \quad \frac{p \rightarrow p'}{1} \\
\frac{\alpha \rightarrow \alpha'}{t} \quad \frac{\alpha \rightarrow \alpha'}{t} \quad \frac{\alpha \rightarrow \alpha'}{t} \\
1' \quad 2' \quad 3
$$

completeness \quad soundness \quad correctness

It is easy to check that $T$ is defined on all program and (1) implies (1'). Also, (2') is too strong in general, and is equivalent to (2) in the case where $t$ is a one-one mapping. Now (3) is too strong in the case where $L_1$ and $L_2$ are nondeterministic. It appears that (1) and (2) are sufficient criteria of correctness for nondeterminism and enables one to avoid the difficult problem of equality of semantic domains [16]. Thus, (1) and (2) seems to be adequate criteria of correctness of a translation.

6.3. Simple (but not infrequent) cases

In simple cases, rule (1) and rule (2) are equivalent, since it is easy to prove the following

$$
\begin{align*}
\text{Theorem 6.1. If } & 1 \text{ and } 2 \text{ are "deterministic"} \\
& (\text{i.e. } \vdash p: \alpha \land \vdash p: \beta \Rightarrow \alpha = \beta) \text{ and if } t \text{ is a one-one mapping (i.e. } \forall \alpha \text{ (resp. } \beta) \exists! \beta \text{ (resp. } \alpha) \text{ s.t. } \vdash \alpha \rightarrow \beta) \\
& \text{then the inference rules (1) and (2) are equivalent.}
\end{align*}
$$

6.4. Induction on the length of the proof

To prove the validity of an inference rule, we have to prove that for each proof of the premises, we exhibit a proof of the conclusion. For that we shall use induction on the length of the proofs of the premises. We are allowed to use induction on the length of the proof for we are working in the "deductive system", and/or we only consider "syntactic models". Let's say we want to do semantics and still be "purely syntactic"... Proofs are for us a -very- formal-game of "dominos". We could also attempt to use structural induction on the source program, but, as we shall see in our example, this induction is not sufficient to carry out the proof.

We are ready now to prove the correctness of the translation from Mini-ML to CAM. Unfortunately, in this case $t$ is not one-one (because of the closures), so we have to prove that both Rule (1) and Rule (2) hold.

7. Proof of the translation from Mini-ML to CAM

We have already described three of the four inference systems required: ml ds, ml cam and cam ds. They specify respectively the dynamic semantics of Mini-ML, the translation from Mini-ML to CAM and the dynamic semantics of CAM. Now we must give the formal system, $t$, describing the translation of semantic values.

7.1. Translation of semantic values

We need here two auxiliary definitions. Given $\rho$, an environment used by ML, consisting of a list of mapping of the form $[p \mapsto \alpha, q \mapsto \beta, r \mapsto \gamma]$, we define:

Definition 7.1. $\bar{\rho}$ is an environment used by the translation. It is the "pattern" corresponding to $\rho$ and defined by the rules:

- $\bar{\rho} = \epsilon$
- $\bar{p} \mapsto \bar{p} \mapsto \bar{p}
- \bar{\rho}; \rho_1 = (\bar{\rho}_1, \bar{\rho})$

Definition 7.2. $\bar{\rho}$ is a value put on the stack used by CAM. It is the "term" corresponding to $\rho$ and defined by:

- $\bar{\rho} = \emptyset$
- $\bar{p} \mapsto \bar{t} \mapsto \bar{t}(\alpha)$
- $\bar{\rho}; \rho_1 = (\bar{\rho}_1, \bar{\rho})$

Example. For $\rho = [p \mapsto 1, q \mapsto 2, r \mapsto 3]$ we have $\bar{\rho} = ((1(1), q), p)$ and $\bar{\rho} = ((1(0, 3), 2), 1)$.

The deinition of $t$ on semantic values is given in Fig. 8. It is quite natural, with the possible exception of closures.
\[ \vdash t(n) = N \]
\[ \vdash t(\tau) = \tau \]
\[ \vdash t((\alpha, \beta)) = (t(\alpha), t(\beta)) \]
\[ \vdash t([\lambda \beta. e, \rho]) = [\lambda \beta. e, t(\rho)] \]
\[ \vdash t([\beta, \rho]) = ([\beta, t(\rho)]) \]
\[ \vdash t(\bot) = \bot \]

\[ \begin{array}{c}
\text{init.env} \vdash e : \alpha \\
\text{init.pat} \vdash e : \rightarrow \text{program}(c) \\
\text{ml.dls} \vdash e : \alpha \\
\text{ml.cam} \vdash e : \rightarrow \\
\text{init.stack} \vdash \text{program}(c) : t(\alpha) \\
\text{cam.dls} \vdash e \vdash e : t(\alpha) \\
\text{cam.dls} \vdash e : \rightarrow \\
\end{array} \]

This suggests what inference rule we should attempt to prove. It is stronger than the above inference rule, as it is often the case in proofs by induction. By definition, \text{init.env} = \text{init.pat} and \text{init.env} = \text{init.stack}. So we shall prove that, for all expressions \( e \) of Mini-ML, for all environments \( \rho \), and for all stack \( s \) of CAM:

\[ \begin{array}{c}
\text{ml.dls} \vdash e : \alpha \\
\text{ml.cam} \vdash e : \rightarrow \\
\text{rho} \vdash e : \alpha \\
\text{rho} : s \vdash e : t(\alpha) : s \\
\end{array} \]

is valid, by induction on the length of the proofs of the premises.

For each step of the induction, we shall draw a proof tree containing the three proof trees under consideration. The symbol \( \vdash \) will be overloaded as the set of rules involved will become evident from the context. Uses of lemma or hypothesis of induction will be indicated by (lemma \( L \)) or (induction).

We have to consider all possible proof trees of \( \rho \vdash_{\text{ml.dls}} e : \alpha \) and \( \rho \vdash_{\text{ml.cam}} e \rightarrow c \). For simple cases, when there is one inference rule in \( \text{ml.dls} \) and one in \( \text{ml.cam} \) (with the exception of the \( \bot \)-rule) dealing with a given constructor of ML, this looks like structural induction. The most interesting case, for which structural induction on the source term is not powerfull enough, is the general case of an application (Rules \( \text{ml.dls.1} \) & \( \text{ml.cam.2} \))

\[ \text{ml.dls.1} & \text{ml.cam.2}: \ e = \text{number}(n) \]
\[ \rho \vdash \text{number}(n) : n \]
\[ \rho \vdash \text{quote}(\text{int}(n)) : n \cdot s \]

For this we have only used rules \( \text{ml.dls.2} \) & \( \text{ml.cam.2} \) and rules \( \text{cam.dls}.5 \). No induction was needed as the proof trees are of length 1.

\[ \bot-\text{Rules in ml.dls & ml.cam}: \ e = \text{number}(n) \]
\[ \rho \vdash \text{number}(n) : \bot \]
\[ \rho \vdash \text{quote}(\text{int}(n)) : \bot \cdot s \]

For each proof tree of \( \rho \vdash e : \alpha \) we can use the axiom \( \rho \vdash e : \bot \). Each time we have made that choice in \( \text{ml.dls} \) we can make the similar choice in \( \text{cam.dls} \) and use the axiom \( \alpha \cdot s \vdash e : \bot \cdot s \). So each one of these cases will be as trivial as the proof above, and we will not consider them in the following.

Figure 8. The definition of \( t \) on semantic values.

We have presented \( t \) as a function in the interest of compactness in the layout of proof trees. Rules (1) and (2) say that the translation on simple semantic values is the identity. Rule (3) says that the translation of a pair is the pair of the translations. The translation of a predefined operator (plus, etc..) is -in short - a closure of the code corresponding to this operator (Rule (4)). The translation of a closure of a \( \lambda \)-expression is the corresponding closure (Rule (5)). The translation of a closure of a pair of expressions is the pair of the translation of the closure of each expression (Rule (6)). Finally \( t(\bot) \) is the identity on \( \bot \) (Rule (7)). This rule is for the case of partial evaluation of the program.

The only little difficulty is for closures: in this case, \( t(\alpha) \) is defined in term of \( \tilde{\rho} \) and \( \tilde{\rho} \) is defined in term of \( t(\alpha) \). In fact, there is no mistery: the translation of a graph is a graph. For example, consider \( \rho = \rho \mapsto [e, \rho] \). We have \( \rho = \rho \mapsto e \mapsto [e, \rho] \)

\[ \begin{array}{c}
\text{ml.dls} \vdash e : \alpha \\
\text{ml.cam} \vdash e : \rightarrow \\
\text{cam.dls} \vdash e : t(\alpha) \\
\end{array} \]

We are now ready to proceed with the complete proof.

7.2. Proof of rule (1)

We have to prove that the following inference rule:

\[ \begin{array}{c}
\text{ml.dls} \vdash e : \alpha \\
\text{ml.cam} \vdash e : \rightarrow \\
\text{cam.dls} \vdash e : t(\alpha) \\
\end{array} \]

is valid in the theory \( T = \text{ml.dls} \cup \text{ml.cam} \cup \text{cam.dls} \cup t \).

For each proof tree for the premises we must exhibit a proof tree for the conclusion. Let's make one step in this direction. We must construct the following proof tree:
Rules ml_ds.2, .3 & ml_cam.3, .4: The proofs are very similar to the previous one.

Rules ml_ds.4 & ml_cam.10: $e = \lambda p.e.$

$\rho \vdash \lambda p.e : [\lambda p,e,\rho]$

$\frac{\rho \vdash \lambda p.e \rightarrow cur(c)}{\rho \vdash \lambda p.e \rightarrow cur(c)}$

$\overline{\rho \cdot s \vdash c : t(a) \cdot s}$  (Lemma "environment simulation")

Note: This proof tree is unusual in that hypothesis used for derive the conclusion are not written just above it: we have not rewritten these two hypothesis, as one usually do. This will be general in the paper: in order to make our proof trees manageable, we shall not rewrite the hypothesis at each stage.

Now, we have used a lemma which says that the ML environment is correctly simulated in the Cam:

**Lemma "environment simulation".**

\[
\begin{align*}
\text{val.of} & \quad \rho \vdash \text{ident}i : \alpha \quad \rho \vdash \text{ident}i \rightarrow c \\
\text{access} & \quad \rho \vdash \text{ident}i \rightarrow c \\
\overline{\rho \cdot s \vdash c : t(a) \cdot s}
\end{align*}
\]

**Proof.** It is easy to prove, by induction on the length of the proof, the following stronger rule:

\[
\begin{align*}
\text{val.of} & \quad \rho \vdash \text{ident}i : \alpha \quad \rho \vdash \text{ident}i \rightarrow c \quad \rho \vdash \varphi_c(\beta) \vdash \text{ident}i \rightarrow c' : c \\
\text{access} & \quad \rho \vdash \varphi_c(\beta) \vdash \text{ident}i \rightarrow c' : c \\
\overline{\rho \cdot s \vdash c : t(a) \cdot s}
\end{align*}
\]

where $\varphi_c(\rho)$ is the environment mapping the identifiers of $\rho$ to their access paths in $\rho$, prefixed by $c$. The definition of $\varphi_c$ is as follows:

- $\varphi_c(s) = \varphi$
- $\varphi_c(\rho) = \rho \rightarrow c$
- $\varphi_c(\rho, p) = \varphi_{c;\text{cdr}(p_1)} \cdot \varphi_c(\rho)$

For example, $\varphi_c([(((\cdots ,r),q),p)\cdots] = p \rightarrow c; \text{cdr} \cdot q \rightarrow c;\text{car;} \cdots \text{cdr} \cdot r \rightarrow c;\text{car;}\cdots$

\[\square\]

**Lemma "environment simulation"**

From now on we shall not give the proof tree in cam_ds in full details. Executions of sequences of commands, or push, car... will be skipped.

Rules ml_ds.7 & ml_cam.6: $e = \text{if } E_1 \text{ then } E_2 \text{ else } E_3$.

The "false case" is similar to the previous (true) case.

Rules ml_ds.8 & ml_cam.7: $e = (E_1, E_3)$.

\[
\begin{align*}
\rho \vdash E_1 : \alpha \quad \rho \vdash E_2 : \beta & \quad \rho \vdash E_1 \rightarrow c_1 \\
\rho \vdash E_2 \rightarrow c_2 & \\
\rho \vdash \{E_1, E_2\} : \alpha(\beta) & \\
\rho \vdash \{E_1, E_2\} \rightarrow \text{push}; c_1; \text{swap}; c_2; \text{cons} \\
\rho \cdot \rho \cdot s \vdash c_1 : t(\alpha) \cdot s (\text{induction}) & \\
\overline{\rho \cdot s \vdash \text{push}; c_1; \text{swap}; c_2; \text{cons} : (t(\alpha), t(\beta)) \cdot s}
\end{align*}
\]

For Rule ml_ds.9, we use a Lemma on eval, which is taken as hypothesis:

**Lemma "eval".**

\[
\begin{align*}
\text{eval} & \quad \text{OP}, \alpha, \beta : \gamma \\
\text{trans.const} & \quad \text{idt OP} \rightarrow c \\
\text{eval} & \quad c_1, t(\alpha), t(\beta) : t(\gamma)
\end{align*}
\]

**Figure 9.** Rules ml_ds.6 & ml_cam.6: $e = \text{if } E_1 \text{ then } E_2 \text{ else } E_3$.

\[
\begin{align*}
\rho \vdash E_1 : \text{ident OP} & \quad \rho \vdash E_2 : (\alpha, \beta) \\
\text{eval} & \quad \text{OP}, \alpha, \beta : \gamma \\
\rho \vdash E_1, E_2 : \gamma & \\
\rho \vdash E_1, E_2 \rightarrow c_1 \\
\rho \vdash E_1 \rightarrow c_1 \\
\rho \vdash E_2 \rightarrow c_2 & \\
\text{trans.const} & \quad \rho \vdash E_1, E_2 \rightarrow c_2; c_1 \\
\text{eval} & \quad c_1, t(\alpha), t(\beta) : t(\gamma) \text{ (Lemma "eval" of)} \\
\overline{\rho \cdot s \vdash \text{push}; c_1; \text{branch}(c_2, c_3) : t(\alpha) \cdot s}
\end{align*}
\]

**Figure 10.** Rules ml_ds.9 & ml_cam.11: $e = E_1 E_2$ with $E_1 = \text{ident OP}$. 
\[\begin{align*}
\rho \vdash E_1 : \text{ident\,OP} & \quad \rho \vdash E_2 : (\alpha, \beta) & \quad \text{eval}\quad \frac{\rho \vdash E_1 : \alpha, \beta}{\rho \vdash E_2 : \gamma} & \vdash \text{ident\,OP} \to \text{t(op)}.
\end{align*}\]

\[\begin{align*}
\rho \vdash E_1, E_2 : \gamma & \vdash \text{OP:}\alpha, \beta : \gamma \quad \beta \vdash E_1 \to c_1 \quad \beta \vdash E_2 \to c_2 \\frac{\rho \vdash E_1, E_2 \to \text{push};c_1;\text{swap};c_2;\text{cons};\text{app}}{\quad \rho \vdash E_1, E_2 \to \text{push};c_1;\text{swap};c_2;\text{cons};\text{app}}
\end{align*}\]

Figure 11. Rules ml.ds.9 & ml.cam.13: \(e = E_1, E_2\) with \(E_1\) executes to ident OP.

\[\begin{align*}
\rho \vdash E_2 : \alpha & \quad \rho \vdash E_1 : \text{[[}\lambda \alpha. E, \rho_1\text{]} E \to \alpha \cdot \rho_1 \vdash E : \beta & \quad (\rho_1, \rho) \vdash E \to e
\end{align*}\]

\[\begin{align*}
\rho \vdash E_1, E_2 : \gamma & \quad \rho \vdash E_1 \to c_1 & \quad \rho \vdash E_2 \to c_2 \\frac{\rho \vdash E_1, E_2 \to \text{push};c_1;\text{swap};c_2;\text{cons};\text{app}}{\quad \rho \vdash E_1, E_2 \to \text{push};c_1;\text{swap};c_2;\text{cons};\text{app}}
\end{align*}\]

Figure 12. Rules ml.ds.10 & ml.cam.13: \(e = E_1, E_2\), general case.

\[\begin{align*}
\rho \vdash E_1 : \alpha & \quad \rho \vdash E_2 : \beta & \quad \rho \vdash E_1 \to c_1 \quad (\rho, \rho) \vdash E_2 \to c_2
\end{align*}\]

\[\begin{align*}
\rho \vdash \text{let}\,P = E_1\,\text{in}\,E_2 : \beta & \quad \rho \vdash \text{let}\,P = E_1\,\text{in}\,E_2 \to \text{push};c_1;\text{cons};c_2
\end{align*}\]

\[\begin{align*}
\rho \cdot \rho \cdot s & \vdash c_1 : t(\alpha) \cdot \rho \cdot s & \vdash t(\beta) \cdot s & \vdash \text{induction}
\end{align*}\]

\[\begin{align*}
\rho \cdot \rho \cdot s & \vdash c_1 : t(\alpha) \cdot \rho \cdot s & \vdash \text{induction}
\end{align*}\]

\[\begin{align*}
\rho \cdot \rho \cdot s & \vdash \text{push};c_1;\text{swap};c_2;\text{cons};\text{app} \to t(\beta) \cdot s & \vdash \text{induction}
\end{align*}\]

Figure 13. Rules ml.ds.11 & ml.cam.8: \(e = \text{let}\,P = E_1\,\text{in}\,E_2\).

\[\begin{align*}
\rho_1 \vdash E_1 : v_1 & \quad (\rho, \rho) \vdash E_1 \to c_1 & \quad (\rho, \rho) \vdash E_2 \to c_2
\end{align*}\]

\[\begin{align*}
\rho \vdash \text{letrec}\,P = E_1\,\text{in}\,E_2 : \beta & \quad \rho \vdash \text{letrec}\,P = E_1\,\text{in}\,E_2 \to \text{push};\text{quote}(x);\text{cons};\text{push};c;\text{swap};\text{rplac};c
\end{align*}\]

\[\begin{align*}
\rho \cdot \rho \cdot x \cdot (\rho, x) \cdot s & \vdash c : t(v_1) \cdot (\rho, x) \cdot s & \vdash \text{induction}
\end{align*}\]

\[\begin{align*}
\rho \cdot \rho \cdot x \cdot \text{push};\text{quote}(x);\text{cons};\text{push};c;\text{swap};\text{rplac};c & \to t(\beta) \cdot s
\end{align*}\]

Figure 14. Rules ml.ds.12 & ml.cam.9: \(e = \text{letrec}\,P = E_1\,\text{in}\,E_2\).
Rules ml.ds.12 & ml.cam.9: $e$ on letrec $p = e_1$ in $e_3$.

The proof tree is given in Fig. 14. To draw it, we have used Theorem 5.1, which states that $\Phi(E_s)$ is not empty, so $\exists E_1, \exists v_1$ s.t. $\rho_1 \vdash e_1 : v_1$. Uses of hypothesis of induction were valid with some extra hypothesis: The first one (1) needs $x = t([\rho_1, \rho_1])$, while the second one (2) needs $t(v_1) = t([\rho_1, \rho_1])$. These two extra hypotheses are valid by the following lemma:

Lemma $\lambda_1$. $\rho \vdash e : v$, $e$ is a list of $\lambda$-exp. $\Rightarrow t(v) = t([e, \rho])$

Proof. The proof uses induction on the length of the proof of $\rho \vdash e : v$.

1st case: $e = \lambda p. e$. We have $\rho \vdash e : [\lambda p. e, \rho]$. So $t(v) = t([\lambda p. e, \rho]) = t([e, \rho])$.

2nd case: $e = (\alpha, \beta)$, $\alpha, \beta \in \lambda$-exp.

\[
\frac{\rho \vdash \alpha : \alpha'}{\rho \vdash (\alpha, \beta) : (\alpha', \beta')}
\]

$t((\alpha', \beta')) = t((\alpha'), t(\beta'))$ by definition of $t$

$= (t([\alpha, \rho]), t([\beta, \rho]))$ by hyp. of induction

$= (t([\alpha, \rho]), t([\beta, \rho]))$ by definition of $t$

\[\square \text{Lemma } \lambda_1\]

\[\square \text{Rule } (1)\]

7.3. Proof of rule (2)

The proof of the second inference rule will complete the proof of the correctness of our translation.

\[
\begin{align*}
\exists \alpha & \vdash e : e \quad \mathtt{ml.cam} \quad \mathtt{cam.ds} \\
\exists \alpha & \vdash e : e' \quad \mathtt{ml.cam} \quad \mathtt{cam.ds}
\end{align*}
\]

As before, we shall prove a stronger rule:

\[
\exists \alpha, \rho \quad \frac{\rho \vdash e \rightarrow e' \quad \rho \vdash e : \alpha \quad \alpha \rightarrow \alpha'}{\rho \vdash e : e' \rightarrow e' : \alpha \rightarrow \alpha'}
\]

Here again, we can give an equivalent form of this rule, for the theory ml.ds and t have no connection and for the theorem of deduction holds:

\[
\exists \alpha, \rho \quad \frac{\rho \vdash e \rightarrow e' \quad \rho \vdash e : \alpha \quad \alpha \rightarrow \alpha'}{\rho \vdash e : e' : \alpha \rightarrow \alpha'}
\]

The proof will again use induction on the length of the proof. Proof of rule (2) is mostly similar to proof of rule (1), we shall not give it in full detail. The differences concern the apply and letrec rules, and the constant use of the following lemma:

Lemma Cam. Every Cam code translating an $\lambda$-expression preserves the stack. More formally:

\[
\exists \beta \quad \frac{\rho \vdash e \rightarrow e \quad \alpha \rightarrow \alpha'}{\rho \vdash e : e' \rightarrow e : \alpha \rightarrow \alpha'}
\]

Proof. The proof is obvious, by induction on the length of the proof, except maybe for rule ml.cam.12 (see Fig. 15).

\[\square \text{Lemma Cam}\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig15}
\caption{Proof of lemma cam. Rule ml.cam.12.}
\end{figure}
Figure 16. Rule ml_cam.12: $e = E_1 E_2$, 1st case.

\[
\begin{align*}
\beta \vdash E_1 \rightarrow c_1 & \quad \beta \vdash E_2 \rightarrow c_2 \\
\beta \vdash E_1 \rightarrow push; c_1; \text{swap}; c_2; \text{cons}; \text{app} & \\
\beta \vdash E_2 ; \beta \rightarrow \gamma \text{ (ind.)} & \\
\rho \vdash E_1 ; \alpha = \lambda \beta. e_1, \rho_1 & \\
\beta \vdash E_2 ; \beta \rightarrow \gamma' \text{ (ind.)} & \\
\rho \vdash E_1 ; E_2 ; \gamma & \\
\beta \vdash E_1 ; \gamma & \\
\beta \vdash E_2 ; \gamma & \\
\end{align*}
\]

Figure 17. Rule ml_cam.12: $e = E_1 E_2$, 2nd case.

\[
\begin{align*}
\beta \vdash E_1 \rightarrow c_1 & \quad \beta \vdash E_2 \rightarrow c_2 \\
\beta \vdash E_1 \rightarrow push; c_1; \text{swap}; c_2; \text{cons}; \text{app} & \\
\beta \vdash E_2 ; \beta \rightarrow \beta' \text{ (ind.)} & \\
\rho \vdash E_1 ; \alpha = \text{ident} op & \\
\beta \vdash E_2 ; \gamma & \\
\gamma \rightarrow \gamma' & \\
\end{align*}
\]

Figure 18. Rule ml_cam.9: $e = \text{letrec } P = E_1$ in $E_2$.

\[
\begin{align*}
\beta \vdash E_1 \rightarrow c_1 & \quad \beta \vdash E_2 \rightarrow c_2 \\
\beta \vdash \text{letrec } P = E_1 \text{ in } E_2 \rightarrow push; \text{quote}(x'); \text{cons}; \text{push}; c_1; \text{swap}; \text{rplac}; c_2 & \\
\rho \vdash x \cdot \beta \vdash E_1 ; v_1 & \\
\rho \vdash x \rightarrow x' & \\
\rho \vdash v_1 \rightarrow v_1' \text{ (ind.)} & \\
\rho \vdash v_1 \cdot \beta \rightarrow \beta' \text{ (ind.)} & \\
\rho \vdash \text{letrec } P = E_1 \rightarrow \beta & \\
\end{align*}
\]

Rule ml_cam.12: $e = E_1 E_2$, 1st case. The proof tree is given in Fig. 16. To draw it, we made one hypothesis on the form that can have $\alpha'$ for making the apply rule in cam.ds applicable. Hyp.1: $\alpha' = [c, v(\rho_1)] \Rightarrow \alpha = \lambda \beta. E_1, \rho_1$.

Rule ml_cam.12: $e = E_1 E_2$, 2nd case. The alternative hypothesis says that $\alpha'$ is a closure again, but translation of an identifier. Hyp.2: $\alpha' = [c, v(\rho_1)] \Rightarrow \alpha = \text{ident } \text{op} \Rightarrow \alpha' = (\gamma'_1, \gamma'_2)$. We use here a second lemma on eval, similar to the previous one, and taken as hypothesis as well:

Lemma eval2.

\[
\begin{align*}
\exists \alpha, \beta, \gamma & \\
\text{eval} & \\
\text{trans_const} & \\
\text{idt_op } & \\
\text{alpha} & \\
\beta & \\
\end{align*}
\]

Then we can draw our proof tree (see Fig. 17).
Rule ml.cam.9: $e = \text{letrec } P = e_1 \text{ in } e_2$. We need here a lemma on $\lambda$-exp. and list of closures. The intuitive meaning of this very technical lemma will be clear, we hope so, by its proof, and by the proof tree it allows to draw.

Lemma $\lambda_2$. For $E_1$ being a list of $\lambda$-expressions, we have:

$$
\begin{align*}
E_2 \vdash P & \rightarrow x \cdot \rho \vdash E_1 : v_1 \\
(\tilde{\rho}, t(x)) \cdot \sigma & \vdash c_1 : (v_1) \cdot \sigma \\
t(v_1) & = t(x)
\end{align*}
$$

$$
\frac{P \rightarrow v_1 \cdot \rho \vdash E_2 : \beta}{P' = P \rightarrow [(E_1, \rho')] \cdot \rho \vdash E_2 : \beta}
$$

Proof. The proof is easy, using the three technical facts we just state below:

- If $\rho \vdash (e_i)_i : v$ with $(e_i)_i \in \lambda$-exp. then $v = ([e_i, \rho]_i);
- \text{if } t(v) = t((e_i, \rho)_i) \text{ then } v = ([e_i, \rho]_i)$, or $v = [(e_i, \rho)_i];
- \text{if } (\tilde{\rho}, P) \vdash (e_i)_i \rightarrow c \text{ and } (\tilde{\rho}, t(x)) \cdot s \vdash c : t(v) \cdot s$ with $(e_i)_i \in \lambda$-exp. then $t(v) = t(([e_i, P \Rightarrow x \cdot \rho]_i));

\square \text{ Lemma } \lambda_2

Now we can draw our proof tree (see Fig. 18).

\square \text{ Rule (2)}

8. Conclusion

Semantic definitions appear to be very compact, thanks to a very general style, and thanks to some extra possibilities, such as the use of graphs, for example, which enables us to give a very clear and compact semantics for the letrec construct. Translation specification is written in the same style as static semantics. Semantic definitions deal with validity of theorems $(\rho \vdash e : \alpha, \rho \vdash e \rightarrow e ...)$ in formal systems, and proofs of translation deal with validity of inference rules in the union of these formal systems.

The device of using the union of several formal systems is not only interesting to formalize a proof, it appear to provide a good framework for formalizing mixed execution (i.e. execution of partially compiled programs). More precisely, once the correctness of a translation has been proved, we can add the just proved inference rules in $T$. Now, suppose we have compiled some parts of a program (using the semantic definition of the translation). The execution of the mixed program obtained is specified by the semantic semantics of the source language, the semantics of the machine code, together with the new rules which appear to be 'switching rules' from a language to another.

In particular, the new rules explain how to communicate environments between interpreted and compiled code. For the moment, this actually works when the translation on semantic values is a one-one mapping. Experiments have been made with a small Pascal-like language.

References


