Proof of translation in Natural Semantics

Unpublished. November 1987

Joëlle Despeyroux
INRIA, Sophia-Antipolis
e-mail: joelle.despeyroux@sophia.inria.fr

We show here how we prove the correctness of a translation, when the dynamic semantics of the two languages together with the translation are given in Natural Semantics. 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), including translation of partially compiled programs. [1]

1. Introduction

Our purpose here is to show how to prove the correctness of a translation, when the dynamic semantics of the two languages and the translations are specified in Natural Semantics.

Natural semantics has its origin in the structural operational semantics most fully developed in [19]. But we focus on the pure logical part of it. The natural semantics of a language is given by one (or more) formal systems (sets of axioms and inference rules) which define sets of valid theorems (theories). Theorems of interest are, for example:

$$\begin{array}{cc} \vdash P : \alpha & \text{P executes to } \alpha \\ \vdash P \in \pi & \text{type of P is } \pi \\ \vdash P \to P' & \text{P translates to } P' \end{array}$$

The name *natural* comes from the fact that we write our systems in the *natural deduction* style [21][22], using Gentzen's sequents [10].

So a semantic definition is a logic, and reasoning with the language is proving theorems in this logic. Type-checking, interpreting, compiling,... are seen as a way to solve equations. For example, given a dynamic environment ρ and a program p, is there a semantic value α s.t. $\rho \vdash p : \alpha$ holds? For further comments on Natural Semantics see [14].

The programming language supporting natural semantics is called Typol. Typol is developed under the syntactic meta-editor Mentor [9] (and its new version Centaur). It is not our purpose here to describe this language [7][8]. 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). Numerous examples, in static semantics, dynamic semantics and translation, have been written in natural semantics and are presented in [2][3][4]. All these examples have been compiled and so have mechanically generated running type-checkers, interpreters and compilers. A first experience with incremental type-checking and errors is reported in [12].

Now, let's give an idea of our method of proof of translation. Let's recall the usual diagram [18]:

$$\begin{array}{ccc} L_1 & \xrightarrow{T} & L_2 \\ {\scriptstyle 1}\downarrow & & \downarrow{\scriptstyle 2} \\ SD(L_1) & \xrightarrow[t]{} & SD(L_2) \end{array}$$

[1] This paper is a polished version of [6], with the additional taking acompt of partially compiled programs. This work is partially supported under ESPRIT project 348

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 disjoints [2] 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 proofs (not theorems)... We shall say that the translation is correct iff two certain inference rules are valid in $\mathcal{T}$. The proof will use induction on the length of the proof.

Outline.

We develop in Part 2 a notion of 'approximate normal form' of a program (inspired by [23][15]), which will enable us to say that the criterion of correctness of translation given above deals with infinite programs as well. Part 3 presents our criteria of correctness of a translation. We prove in part 4 that the composition of two correct translations is correct. Part 5 shows how we can deal with partially compiled programs. Then the rest of the paper is devoted to an illustration of the method on an example. Part 6 gives the dynamic semantics of Mini-ML (the purely applicative part of ML). In part 7, we discuss in full the semantics of Mini-ML infinite terms. Part 8 gives the dynamic semantics of CAM: 'categorical abstract machine', a very interesting machine language developed by G. Cousineau and P.L. Curien [5]. Part 9 gives the translation from Mini-ML into CAM. Part 10 gives the complete proof for our example: Mini-ML to CAM. Finally, part 11 present the partial translation of the same example, and part 12 proves its correctness.

*

A complete proof of Mini-ML to CAM can be found in [17]. But this proof is done in a completely different context and style: M. Mauny proved that the CAM machine correctly simulates the β-reduction of the λ-calculus, by induction on the length of the transitions of the machine. Also W. Li [16] deals with correctness of translation, but he is interested in concurrent languages, and, chooses the operational semantics 'à la Plotkin', so he considers equivalences of behaviours of transitions systems.

2. Infinite execution

We show here how we can extend the natural dynamic semantics of a language to deal with infinite programs as well. We are very brief in this section, as this is just a general method. We will discuss the ML case in full later on.

We have seen that the natural semantics of a language is given by an inference system (which defines a set of valid theorem i.e. a theory). In the case of the dynamic semantics, theorems of interest will often have the form:

$$\vdash P : \alpha$$

meaning that 'P executes to α'. In our context semantic domains are integers, booleans, closures...

Now to deal with infinite programs, we follow the spirit of denotational semantics. We add an undefined value $\perp$ to our semantic domain, thus defining the set of approximate normal forms [23][15] for our language.

Then, we add one (or several) axiom in our dynamic semantics, saying that the evaluation of any expression of the language may return $\perp$.

$$\rho \vdash e : \perp$$

and we define the set of approximate normal forms of an expression:

[2] L_2 may also be identical to L_1, in the case of partially compiled program for example, as we will see later on.

Definition 2.1. *α is an approximate normal form (a.n.f.) of t if*

- *α is an approximate normal form*
- *$\vdash t : \alpha$ holds*

Now our language has became nondeterministic: for each expression we have the choice to evaluate it or not. In the case of a deterministic language, however, the adding nondeterminism is likely not truely nondeterminism. To show this we can prove a "Church-Rosser property" saying that

$$\vdash p : \alpha \;\&\; \vdash p : \beta \;\Rightarrow\; \exists \gamma,\; \gamma \succeq \alpha,\; \gamma \succeq \beta \; s.t. \; \vdash p : \gamma$$

with the existence of a limit of the sequence α_n such that $\vdash p : \alpha_n$. This guarantees the existence and unicity of the result, even if it is a limit. This Church-Rosser property will be proved later on for Mini-ML.

3. Proof of translation

We give here our criteria of correctness of a translation. The first subsection present those criteria, the second one discuss their adequacy, while the last subsection shows that they are equivalent in the deterministic case (under some condition).

3.1. *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 we want to define the correctness of a translation. Let us first try the following definition:

Definition 3.1. *The translation is correct iff the following inference rules are valid in $\mathcal{T}$:*

$$\frac{\overset{1}{\vdash} p : \alpha \qquad \overset{\mathrm{T}}{\vdash} p \rightarrow p' \qquad \overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'}{\overset{2}{\vdash} p' : \alpha'} \tag{1}$$

$$\frac{\overset{\mathrm{T}}{\vdash} p \rightarrow p' \qquad \overset{2}{\vdash} p' : \alpha' \qquad \overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'}{\overset{1}{\vdash} p : \alpha} \tag{2}$$

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

Note that our criteria of correctness are sufficient for all programs, no-matter whether they terminate or not, are erroneus or not: α and α' are approximate normal forms of p and p', or errors. Our inference rules are expressing the commutativity of a diagram. In pictures we ask for:

$$\begin{tikzcd} p \arrow[r, "T"] \arrow[d, "1"'] & p' \arrow[d, dashed, "2"] \\ \alpha \arrow[r, "t"'] & \alpha' \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} p \arrow[r, "T"] \arrow[d, dashed, "1"'] & p' \arrow[d, "2"] \\ \alpha \arrow[r, "t"'] & \alpha' \end{tikzcd}$$

(1) completeness (2) soundness

[3] All implicit quantifiers are universal quantifiers.

where the arrows denote the given facts and the dashed arrows denote the facts to prove.

Now, we have to refine our definition as it may be too strong in certain cases. Namely,

* Rule (1) may be too strong in the case where t is not a function and $\overset{2}{\vdash}$ is deterministic.
* Rule (2) may be too strong in the case where t is not injective and $\overset{1}{\vdash}$ is deterministic.

So we change our previous definition for the following one:

Definition 3.2. *The translation is correct iff the following inference rules are valid in $\mathcal{T}$:*

$$\frac{\overset{1}{\vdash} p:\alpha \qquad \overset{\mathrm{T}}{\vdash} p\rightarrow p' \qquad \overset{\mathrm{t}}{\vdash} \alpha\rightarrow\alpha'}{\overset{2}{\vdash} p':\alpha'} \quad (1) \quad or \quad \exists\alpha' \ \frac{\overset{1}{\vdash} p:\alpha \qquad \overset{\mathrm{T}}{\vdash} p\rightarrow p'}{\overset{2}{\vdash} p':\alpha' \qquad \overset{\mathrm{t}}{\vdash} \alpha\rightarrow\alpha'} \quad (1')$$

$$\frac{\overset{\mathrm{T}}{\vdash} p\rightarrow p' \qquad \overset{2}{\vdash} p':\alpha' \qquad \overset{\mathrm{t}}{\vdash} \alpha\rightarrow\alpha'}{\overset{1}{\vdash} p:\alpha} \quad (2) \quad or \quad \exists\alpha \ \frac{\overset{\mathrm{T}}{\vdash} p\rightarrow p' \qquad \overset{2}{\vdash} p':\alpha'}{\overset{1}{\vdash} p:\alpha \qquad \overset{\mathrm{t}}{\vdash} \alpha\rightarrow\alpha'} \quad (2')$$

The second inference rules are somewhat unusual. However, notice that an instance of, say, rule $2'$ is not too surprising as a sub-proof-tree. Anyway, a more usual form of this rule would be:

$$\exists\alpha \qquad \frac{\overset{\mathrm{T}}{\vdash} p\rightarrow p' \qquad \overset{2}{\vdash} p':\alpha'}{\overset{1\cup \mathrm{t}}{\vdash} \ p:\alpha \ \& \ \alpha\rightarrow\alpha'} \qquad (2'a)$$

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 $2'a$.

We can state some relations between our inference rules, depending on the properties of t (the translation of semantic values):

Theorem 3.1.

- if t is always defined then rule 1 implies rule $1'$,
- if t is a function then rule $1'$ implies rule 1,
- if t is surjective then rule 2 implies rule $2'$,
- if t is injective then rule $2'$ implies rule 2.

The proof, in each case, is by drawing a derived rule. In the case where the source (or the code) is deterministic, we have more relations between our rules. We will give (and prove) them later on.

3.2. *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 those chosen, are the following: [4]

[4] other rules, of minor interest, are derived rules.

$$
\begin{array}{ccc}
\begin{tikzcd}
p \arrow[r, dashed, "T"] \arrow[d, "1"'] & p' \arrow[d, dashed, "2"] \\
\alpha \arrow[r, "t"'] & \alpha'
\end{tikzcd}
& \text{and} &
\begin{tikzcd}
p \arrow[r, "T"] \arrow[d, "1"'] & p' \arrow[d, "2"] \\
\alpha \arrow[r, dashed, "t"'] & \alpha'
\end{tikzcd}
\\
(1'') & & (3) \\
\text{completeness} & & \text{correctness}
\end{array}
$$

It is easy to check that T is defined on all program and (1) implies $(1'')$. Now (3) is difficult to handle in the case where L_1 and L_2 are nondeterministic, because it requires to define an equality on semantic domains [20]. Our rules enables us to avoid this difficult problem. Thus, our criteria of correctness of a translation seems to be adequate one.

3.3. *The deterministic case*

As in the previous subsection, we take as hypothesis that t is always defined, and there exists an a.n.f. both of any source term, and of its corresponding code. We say that $\vdash$ is deterministic if $\vdash p:\alpha \& \vdash p:\beta \Rightarrow \alpha = \beta$. In the case where the source (or the code) is deterministic, provided certain properties of t (the translation on semantic values), it is sufficient to prove one of our two required rules, as we have the following theorem:

Theorem 3.2.

If $\overset{2}{\vdash}$ is deterministic then

- rule 1 or $1'$ implies rule $2'$,
- if t is injective then rule 1 or $1'$ implies rule 2.

If $\overset{1}{\vdash}$ is deterministic then

- rule $2'$ implies rule $1'$,
- if t is a function then rule $2'$ implies rule 1,
- if t is surjective then rule 2 implies rule $1'$,
- if t is a surjective function then rule 2 implies rule 1.

Proof. The proofs of each case are very similar to each other. Let us prove one case, (1) $\Rightarrow$ (2'): We can exhibit a proof tree for (2'), using the fact that their exists an a.n.f. of any program p, t is always defined and $\overset{2}{\vdash}$ is deterministic.

$$\dfrac{\dfrac{\overset{T}{\vdash} p \to p' \qquad \overset{2}{\vdash} p' : \alpha' \qquad \overset{1}{\vdash} p : \alpha\ [\exists\alpha \text{ a.n.f. of p}] \qquad \overset{t}{\vdash} \alpha \to \alpha''\ [\text{t always.def.}]}{\overset{2}{\vdash} p' : \alpha''\ [\text{Rule (1)}]}}{\overset{1}{\vdash} p : \alpha \qquad \overset{t}{\vdash} \alpha \to \alpha'\ [\overset{2}{\vdash} \text{ det.}]}$$

□

3.4. *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 premisses, we can draw a proof of the conclusion. For that we shall use *induction on the length of the proofs* of the premisses. 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.

4. Composition of translations

We shall see in this section that the composition of two correct translations is also correct. Las, in some cases, there is an extra condition to that theorem: one of the translations on semantic values must be surjective (on the set of approximate normal forms of the translation of some source). Anyway, this requirement arises in two cases among four, and it is not a too strong requirement.

Let us see first which are these cases. A translation can be proved correct by proving rule 1 or rule $1'$, and rule 2 or rule $2'$. That makes four cases. Le us call *i-j correct* a translation which correctness has been proved by rules i and j.

Theorem 4.1.

- If T_1 is i-2 correct and T_2 is j-2 correct then $T_2 \circ T_1$ is k-2 correct.
- If T_1 is i-$2'$ correct and T_2 is j-$2'$ correct then $T_2 \circ T_1$ is k-$2'$ correct.
- If T_1 is i-2 correct and T_2 is j-$2'$ correct then $T_2 \circ T_1$ is k-$2'$ correct if t_{T1} is surjective.
- If T_1 is i-$2'$ correct and T_2 is j-2 correct then $T_2 \circ T_1$ is k-$2'$ correct if t_{T2} is surjective.

where k is $1'$ if i or j is $1'$, otherwise k is 1 and t surjective means t surjective on the set of approximate normal forms of the translation of some source.

In picture, it means that we can *compose* diagrams:

$$\begin{array}{ccccc} L_1 & \xrightarrow{T_1} & L_2 & \xrightarrow{T_2} & L_3 \\ {\scriptstyle 1}\downarrow & & \downarrow{\scriptstyle 2} & & \downarrow{\scriptstyle 3} \\ SD(L_1) & \xrightarrow[t_1]{} & SD(L_2) & \xrightarrow[t_2]{} & SD(L_3) \end{array}$$

Proof.

The proof of each case are very similar to each other. The inference rules to prove are derived rules of those that characterise the correctness of the two translations involved. Let us prove for example that Rule 1 is valid for T_1 and T_2 implies that Rule 1 is also valid for $T_2 \circ T_1$.

$$\cfrac{\cfrac{\overset{1}{\vdash p_1 : \alpha_1} \quad \overset{T_1}{\vdash p_1 \to p_2} \quad \overset{t_1}{\vdash \alpha_1 \to \alpha_2} \quad \overset{T_2}{\vdash p_2 \to p_3} \quad \overset{t_2}{\vdash \alpha_2 \to \alpha_3}}{\overset{2}{\vdash p_2 : \alpha_2}}}{\overset{3}{\vdash p_3 : \alpha_3}}$$

□

5. Translation of partially compiled programs

Let us state the problem. We have an interpretor of both a source and a code language, together with a translation between the too. We would like to point (with the mouse) to possibly several sub-expressions in the source program, and compile them, then interpreting the *mixed* program constructed. We can imagine we want to debug this mixed program, then once again compile some parts of it, and go on. At each stage we want to be sure that the semantics of the program built did not changed. We have tried hard to have this *for free*, from the specification of a correct translation... but, las this does not seem possible. At least we can treat the translation of partially compiled programs as any translation, that is specify it and prove it correct. Let us present the two methods.

5.1. *A too ambitious attempt.*

To prove the validity of our inference rules we can use different inductions, the most natural one being induction on the length of the proof. We will have to prove two stronger rules, whose precise forms depend on the example. Usually, the specifications will use environments, and the rules to prove will have the form:

$$\frac{\rho \overset{1}{\vdash} p : \alpha \qquad \bar{\rho} \overset{\mathrm{T}}{\vdash} p \rightarrow p' \qquad \overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'}{\vec{\rho} \overset{2}{\vdash} p' : \alpha'} \tag{1_ρ}$$

$$\frac{\bar{\rho} \overset{\mathrm{T}}{\vdash} p \rightarrow p' \qquad \vec{\rho} \overset{2}{\vdash} p' : \alpha' \qquad \overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'}{\rho \overset{1}{\vdash} p : \alpha} \tag{2_ρ}$$

where ρ, $\bar{\rho}$, and $\vec{\rho}$ are environments corresponding to each other. Once the correctness of a translation as been proved, we can add the just proved inference rules in our theory $\mathcal{T}$. These inference rules may be seen as some kind of 'swiching rules' between the execution of some source and the execution of its corresponding code. Suppose we want to execute a program p. We can draw a proof tree in L_1_DS. This means that we have interpreted p. But we can also use rule 2_ρ on any sub-expression e of our program, drawing a proof tree in full $\mathcal{T}$:

$$\cfrac{\cfrac{\cfrac{\bigvee \qquad \text{or} \qquad \{\bar{\rho} \overset{\mathrm{T}}{\vdash} e \rightarrow e' \qquad \vec{\rho} \overset{\bigvee}{\vdash} e' : \alpha' \qquad \overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'\}}{\rho \overset{1}{\vdash} e : \alpha}}{\cdots}}{\vdash p : \alpha}$$

This shows that we have a confluence on the different proof trees corresponding to the different possible executions of the same program. In terms of compilation this means that we safely can translate one (or several) part of our program, then executing the code obtained. But the compilation is *dynamic* i.e. the system choose, in a non-deterministic manner, to compile a part of the program. The execution of a partially compiled program seems to correspond to a choice among all the possible proof trees in $\mathcal{T}$. The choice is fixed by the parts of the program we choose to compile.

Las, this lovely idea does not work. Let us see why. Suppose we have a program in L_1 and we replace some parts of it by their translations, which are some peaces of code in L_2. Now we want to execute the mixed program we just have constructed. This execution is specified by L_1_DS as usual, except for the compiled part of the program. For them we will use the following rule:

$$\frac{\vec{\rho} \overset{2}{\vdash} p' : \alpha' \qquad \overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'}{\rho \overset{1}{\vdash} p' : \alpha} \tag{2_m}$$

Rule 2_m is rule 2_ρ except that the sequent $\bar{\rho} \vdash p \rightarrow p'$ as been removed - its proof tree as been 'executed' during the compilation - and p as been replaced by p' in the conclusion - remember that we have replaced some source codes by their translations. This rule just explain how to communicate environments between interpreted and compiled code. The execution of a partially compiled program is specified by $\mathcal{T}$, with rule 2_m, instead of rule 2_ρ. This (informal) modification seems to define a 'court-circuit' in our proof trees. May be this can be formalized so far. But, now suppose that our source language has functions in it. A definition of a function build a source closure, which may be called in the code. To be able to execute a program in L_2 calling a program in L_1, we are tempted to modify rule 1_ρ as we have modified rule 2_ρ. Now, all crashes. It does not seem possible to compare $\mathcal{T}$ and $\mathcal{T} \cup 1_m \cup 2_m$.

Furthermore, in the case where t is not injective and $\overset{1}{\vdash}$ is deterministic, the proof of correctness of the translation could require to prove the following rule, instead of rule (2_ρ):

$$\exists\alpha \qquad \frac{\bar{\rho} \overset{\mathrm{T}}{\vdash} p \rightarrow p' \qquad \vec{\rho} \overset{2}{\vdash} p' : \alpha'}{\rho \overset{1}{\vdash} p : \alpha \quad \& \quad \overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'} \tag{$2'_\rho$}$$

This rule is not easy to use to realise the mixed execution, as it must be used backward, so find an α' and invent an α such that $\overset{\mathrm{t}}{\vdash} \alpha \rightarrow \alpha'$. The set of such α may be infinite, in case of closures, for example. So the above proposal, if ever works, seem to require the use of rules 1 and 2 (neither rule $1'$ nor $2'$), and no function (!). Let us present a more reasonable alternative.

5.2. *A reasonable view.*

The translation of partially compiled programs, as any translation, must be specified and proved correct. The specification and the proof are just a little bit particular.

As the source can call the code and vis-versa, there is just one dynamic semantics of a mixed language. This semantics must *extend* the dynamic semantics of both the source and the code. This is likely to be trivial. The extensions are mainly rules that deal with a sub-expression in another language, or an application of a closure of another language. To enable the swich of environment from code to source, the translation may have to be significantly changed. We have to generate some new code instructions that maintain the source environment during the execution of the code, or at least enable to recover this source environment at any time. We must also say how to translate an already translated expression.

Once the specification has been written, we have to prove it correct. We can use our technic here, provided that we define, by means of inference rules, what we do so simply with the mouse, i.e. pointing to several sub-expressions in the source program, and translate them. This new translation, say T_m, is likely to be an easy extension of the (basic) translation T, saying that for each compose expression we have the choice to compile it, using T, or recursively to 'translate' each of it's sub-expression, using T_m. We can ever imagine to generate T_m from T. Now, the translation will be proved correct by the (usual) proof of commutativity of the following diagram:

$$\begin{array}{ccc} L_m & \xrightarrow{T_m} & L_m \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m} \\ SD(L_m) & \xrightarrow[t_m]{} & SD(L_m) \end{array}$$

where L_m is the language mixing the source and the code, T_m is the mixed translation described above, and t_m the desired translation of semantic values of L_m. Note that t_m must at least contain the translation of semantic values that has been chosed in the rules that deal with a sub-expression in another language. If that requirement is not clear yet, it will become obvious in the example described.

Note: Finally, the alternative presented so far is really the only reasonable way to deal with mixed execution in our framework. The only thing that could be added is the mathematical material necessary to define 'projections' from the extended semantics into the 'pure' one. But this would be a mathematical exercise, of no real use in practice. Furthermore, it appears that the extended semantics are useful for debugging purpose. The instructions added in the code in order to maintain the source environment make it possible to see -and eventually to modify- this source environment, during the execution of a compiled code. This will makes the debugging task of a compiled code much easier.

The rest of the paper is devoted to an illustration of the method of proof of translation on the Mini-Ml and Cam example. We will first present the 'usual' case then the mixed case.

6. Dynamic semantics of Mini-ML

ML is a functional language with polymorphism and higher-order functions, first used in the proof system LCF [11]. Mini-ML consist in the purely applicative part of ML, more precisely a simple typed λ-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:

$$\begin{array}{l}\textbf{let } (x,y) = (2,3)\\ \textbf{in let } (x,y) = (y,x) \textbf{ in } x\end{array}$$

$$\begin{array}{l}\textbf{letrec}(\mathit{even}, \mathit{odd}) = (\lambda x.\, \textbf{if } x = 0 \textbf{ then } \mathit{true} \textbf{ else } \mathit{odd}(x-1),\\ \qquad\qquad\qquad\quad \lambda x.\, \textbf{if } x = 0 \textbf{ then } \mathit{false} \textbf{ else } \mathit{even}(x-1))\\ \textbf{in } \mathit{even}\ 3\end{array}$$

sorts EXP, IDENT, PAT, NULLPAT

subsorts EXP$\supset$ NULLPAT, IDENT
PAT$\supset$ NULLPAT, IDENT

constructors

'Patterns'

pairpat	:	PAT×PAT	$\rightarrow$	PAT
nullpat	:		$\rightarrow$	NULLPAT

'Expressions'

ident	:		$\rightarrow$	IDENT
number,false, true	:		$\rightarrow$	EXP
apply, mlpair	:	EXP×EXP	$\rightarrow$	EXP
lambda	:	PAT×EXP	$\rightarrow$	EXP
let, letrec	:	PAT×EXP×EXP	$\rightarrow$	EXP
if	:	EXP×EXP×EXP	$\rightarrow$	EXP

Figure 1. Abstract Syntax of mini-ML

6.1. *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 ρ is an environment. A closure is just a pair of a λ-expression and an environment.
- identifiers for predefined operators: *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* ρ is a list of pairs $P \mapsto \alpha$ where P is a pattern and α a value. Here is an example of environment: $(((x,y) \mapsto (true, 5)), x \mapsto 1)$.

We say that expression e evaluates to α in environment ρ if the theorem

$$\rho \vdash e : \alpha$$

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

program ML_DS is

use ML

$\rho, \rho_1 : PAT;$

$\alpha, \beta : VALUE;$

$$\frac{init\_env \vdash \mathrm{E} : \alpha}{\vdash \mathrm{E} : \alpha} \qquad \text{(i)}$$

$$\rho \vdash \mathrm{number\,N} : \mathrm{N} \qquad (1)$$

$$\rho \vdash \mathrm{true} : true \qquad (2)$$

$$\rho \vdash \mathrm{false} : false \qquad (3)$$

$$\rho \vdash \lambda \mathrm{P.E} : [\![\lambda \mathrm{P.E}, \rho]\!] \qquad (4)$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \mathrm{ident\,I} \mapsto \alpha}{\rho \vdash \mathrm{ident\,I} : \alpha} \qquad (5)$$

$$\frac{\rho \vdash \mathrm{E}_1 : true \qquad \rho \vdash \mathrm{E}_2 : \alpha}{\rho \vdash \mathrm{if\,E}_1\,\mathrm{then\,E}_2\,\mathrm{else\,E}_3 : \alpha} \qquad (6)$$

$$\frac{\rho \vdash \mathrm{E}_1 : false \qquad \rho \vdash \mathrm{E}_3 : \alpha}{\rho \vdash \mathrm{if\,E}_1\,\mathrm{then\,E}_2\,\mathrm{else\,E}_3 : \alpha} \qquad (7)$$

$$\frac{\rho \vdash \mathrm{E}_1 : \alpha \qquad \rho \vdash \mathrm{E}_2 : \beta}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) : (\alpha, \beta)} \qquad (8)$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathrm{ident\,OP} \qquad \rho \vdash \mathrm{E}_2 : \alpha \qquad \overset{\text{eval}}{\vdash} \mathrm{OP}, \alpha : \beta}{\rho \vdash \mathrm{E}_1\;\mathrm{E}_2 : \beta} \qquad (9)$$

$$\frac{\rho \vdash \mathrm{E}_1 : [\![\lambda \mathrm{P.E}, \rho_1]\!] \qquad \rho \vdash \mathrm{E}_2 : \alpha \qquad (\rho_1, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E} : \beta}{\rho \vdash \mathrm{E}_1\;\mathrm{E}_2 : \beta} \qquad (10)$$

$$\frac{\rho \vdash \mathrm{E}_1 : \alpha \qquad (\rho, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E}_2 : \beta}{\rho \vdash \mathrm{let\,P} = \mathrm{E}_1\,\mathrm{in\,E}_2 : \beta} \qquad (11)$$

$$\frac{(\rho, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E}_1 : \alpha \qquad (\rho, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E}_2 : \beta}{\rho \vdash \mathrm{letrec\,P} = \mathrm{E}_1\,\mathrm{in\,E}_2 : \beta} \qquad (12)$$

end ML_DS

Figure 2. Dynamic semantics of Mini-ML

6.2. *Comments on the formal definition*

In Figure 2, rules 1 to 3 associate values to integer or boolean literals. Rule i 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 λ-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 VAL_OF. Rules 6 and 7 associate values to conditional expression. Rule 8 is equally transparent.

The next rules deal with functional values. Because of type-checking, the operator of an application can only evaluate to a functional value, i.e. a closure, or a predefined operator. 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. The semantic value of the operator is a closure. This closure is taken apart, and its body is evaluated in its environment, postfixed with the parameter association $\mathrm{P} \mapsto \alpha$. Note that the rule is valid whether P is a pattern or a single variable. As, at evaluation time, $\lambda \mathrm{P}.\mathrm{E}_1\ \mathrm{E}_2 = (\mathrm{let}\, \mathrm{P} = \mathrm{E}_2 \,\mathrm{in}\, \mathrm{E}_1)$, rule 11 appears to be an other optimisation of rule 10.[5]

The last rule, rule 12, defines in one and the same way the simple recursive functions and the mutually recursive ones. E_1 is evaluated in an environment which contains a free variable in place of the value of the function (or list of functions) P. The result of the evaluation of E_1 is a closure (or a list of closures) which environment is $(\rho, \mathrm{P} \mapsto \alpha)$. Asking this result to be α makes the environment of the closure(s) a self-referencing environment. E_2 is then evaluated in this environment. Note the similarity between the let-rule and the letrec-rule. Notice that since $\rho \vdash \mathrm{E}_2 : \alpha$ is a premisse 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 because in rules 10 to 12 we have merely postfixed the environment with new associations.

set VAL_OF is

$$(\rho, \mathrm{ident}\, \mathrm{I} \mapsto \alpha) \vdash \mathrm{ident}\, \mathrm{I} \mapsto \alpha \tag{1}$$

$$\frac{\rho \vdash \mathrm{ident}\, \mathrm{I} \mapsto \alpha}{(\rho, \mathrm{ident}\, \mathrm{X} \mapsto \beta) \vdash \mathrm{ident}\, \mathrm{I} \mapsto \alpha} \quad (\mathrm{X} \neq \mathrm{I}) \tag{2}$$

$$\frac{(\rho, (\mathrm{P}_1 \mapsto \alpha, \mathrm{P}_2 \mapsto \beta)) \vdash \mathrm{ident}\, \mathrm{I} \mapsto \gamma}{(\rho, (\mathrm{P}_1, \mathrm{P}_2) \mapsto (\alpha, \beta)) \vdash \mathrm{ident}\, \mathrm{I} \mapsto \gamma} \tag{3}$$

end VAL_OF

Figure 3. The ML environment rules

Rules 1 and 2 scan the environment until the first occurrence of an identifier is found, in a right to left scan. Rule 3 relies on the fact that a pair of identifiers is bound to a value which is a pair. Hence searching is propagated to two new pattern-value pairs.

7. Infinite execution in Mini-ML

The dynamic semantics of the previous section only specify the execution of programs that terminate. We will follow here the method presented in the first section for dealing with infinite programs as well, and discuss the ML case in full.

We add an undefined value $\perp$ to the semantic domain in Mini-ML, thus defining the set of approximate normal forms for this language:

Definition 6.1. *An approximate normal form (a.n.f.) of Mini-ML is either*

- $\perp$,
- *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 $\perp$:

$$\rho \vdash e : \perp$$

and we define the set of approximate normal forms an expression:

[5] Optimisations of both dynamic semantics of Mini-ML and translation from Mini-ml to CAM are presented and proved correct in [3].

Definition 6.2. *α is an approximate normal form (a.n.f.) of t if*

- *α is an a.n.f.*
- *$\vdash t : \alpha$ holds*

Now, an infinite ML term as no semantic value - ML differs here from the λ-calculus - but an infinite set of a.n.f. For example, $\text{letrec}\,\text{F} = \lambda \text{X}.[\text{X}.\text{F X}]\,\text{in}\,\text{F}\,2$ as for a.n.f.: $\perp$ $[\perp.\perp]$ $[2.\perp]$ $[2.\perp.\perp]$ $[2.2.\perp]$ $\cdots$

Our language ML has became 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 6.3. *We define a partial order $\preceq$ on a.n.f. as follows: For all c_i constant, φ, φ_i, φ'_j a.n.f.:*

- $c_i \preceq c_j \;\Leftrightarrow\; i = j$
- $\perp \preceq \varphi$
- $(\varphi_i, \varphi_j) \preceq (\varphi'_i, \varphi'_j) \;\Leftrightarrow\; \varphi_i \preceq \varphi'_i \;\&\; \varphi_j \preceq \varphi'_j$

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

Definition 6.4. *From the partial order on a.n.f. we induce a partial order on environments, defined by extension:*

- $\emptyset \preceq \emptyset$
- $P \mapsto \alpha \cdot \rho \preceq P \mapsto \alpha' \cdot \rho' \;\Leftrightarrow\; \alpha \preceq \alpha' \;\&\; \rho \preceq \rho'$

Theorem 6.1. *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 $\rho_1 \vdash t : \alpha$, $\rho_2 \vdash t : \beta$, $\rho_1 \preceq \rho$, $\rho_2 \preceq \rho$ there exists γ 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.

□ Theorem 6.1

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

So we can define:

Definition 6.5. *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 $\rho \vdash \text{E} : \perp$, we have not really added non-determinism in our language, as we have a "Church-Rosser property":

$$\vdash p : \alpha \;\&\; \vdash p : \beta \;\Rightarrow\; \exists \gamma,\; \gamma \succeq \alpha,\; \gamma \succeq \beta \;s.t.\; \vdash p : \gamma$$

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

8. Dynamic semantics of CAM

The Categorical Abstract Machine [5] 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 (*cons*, *car*, *cdr*, *push*, *swap*...). Instructions *car* and *cdr* serve in accessing data in the stack. Predefined operations (such as addition, subtraction, division, etc.) may be added with the *op* instruction. The abstract syntax of CAM code is given in Fig. 4.

sorts VALUE, COM, PROGRAM, COMS

subsorts COM$\supset$COMS

constructors

program	: COMS	$\rightarrow$ PROGRAM
coms	: COM*	$\rightarrow$ COMS
quote	: VALUE	$\rightarrow$ COM
op,car,cdr,cons	:	$\rightarrow$ COM
push,swap,app, rplac	:	$\rightarrow$ COM
cur	: COMS	$\rightarrow$ COM
branch	: COMS$\times$COMS	$\rightarrow$ COM
int,bool, null_value	:	$\rightarrow$ VALUE

Figure 4. Abstract syntax of CAM code

8.1. *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: *true*, *false*
- closures of the form $[\![\mathrm{C}, \rho]\!]$, where C is a fragment of CAM code and ρ is a value, meant to denote an environment
- 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

$$s \vdash c : s'$$

where c is CAM-code and s and s' are states of the CAM machine. The sequent $s \vdash c : s'$ 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.

8.2. *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. Rule 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 (Rule 17) is to be compared with the *letrec*-rule of the dynamic semantics of Mini-ML. The code C of the *rec* instruction is the translation of a λ-expression (or of a list of λ-expressions). That means that C is a *cur* instruction (or contains several *cur*). So the execution of C builts a closure (or a list of closures), using the environment (ρ, α). Saying that this closure(s) is α makes its environment the desired self-referencing environment.

program CAM_DS is

use CAM

$s, s_1, s_2 : STACK;$

$\alpha, \beta : VALUE;$

$\rho, \rho_1 : ENV;$

$$\frac{init\_stack \vdash \text{COMS} : \alpha}{\vdash \text{COMS} : \alpha} \tag{1}$$

$$s \vdash \emptyset : s \tag{2}$$

$$\frac{s \vdash \text{COM} : s_1 \qquad s_1 \vdash \text{COMS} : s_2}{s \vdash \text{COM}; \text{COMS} : s_2} \tag{3}$$

$$\alpha \cdot s \vdash quote(\text{X}) : \text{X} \cdot s \qquad (var(\text{X})) \tag{4}$$

$$\alpha \cdot s \vdash quote(int\,\text{N}) : \text{N} \cdot s \tag{5}$$

$$\alpha \cdot s \vdash quote(bool\,\text{T}) : \text{T} \cdot s \tag{6}$$

$$(\alpha, \beta) \cdot s \vdash car : \alpha \cdot s \tag{7}$$

$$(\alpha, \beta) \cdot s \vdash cdr : \beta \cdot s \tag{8}$$

$$\alpha \cdot \beta \cdot s \vdash cons : (\beta, \alpha) \cdot s \tag{9}$$

$$\alpha \cdot s \vdash push : \alpha \cdot \alpha \cdot s \tag{10}$$

$$\alpha \cdot \beta \cdot s \vdash swap : \beta \cdot \alpha \cdot s \tag{11}$$

$$\frac{\overset{\text{eval}}{\vdash}\ \text{OP}, (\alpha, \beta) : \gamma}{(\alpha, \beta) \cdot \text{S} \vdash op\,\text{OP} : \gamma \cdot \text{S}} \tag{12}$$

$$\frac{s \vdash \text{C}_1 : s_1}{true \cdot s \vdash branch(\text{C}_1, \text{C}_2) : s_1} \tag{13}$$

$$\frac{s \vdash \text{C}_2 : s_1}{false \cdot s \vdash branch(\text{C}_1, \text{C}_2) : s_1} \tag{14}$$

$$\rho \cdot s \vdash cur(\text{C}) : [\![\text{C}, \rho]\!] \cdot s \tag{15}$$

$$\frac{(\rho, \alpha) \cdot s \vdash \text{C} : s_1}{([\![\text{C}, \rho]\!], \alpha) \cdot s \vdash app : s_1} \tag{16}$$

$$\frac{(\rho, \alpha) \cdot s \vdash \text{C} : \alpha \cdot s}{\rho \cdot s \vdash rec(\text{C}) : \alpha \cdot s} \tag{17}$$

end CAM_DS

Figure 5. Definition of CAM

9. Translation from Mini-ML to CAM

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

9.1. *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:

$$\rho \vdash e \rightarrow c$$

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 ρ, expression e 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*

$c, c_1, c_2, c_3 : CAM;$

$\rho, \rho_1 : ENV;$

$$\frac{init\_pat \vdash \mathrm{E} \rightarrow c}{\vdash \mathrm{E} \rightarrow c} \tag{1}$$

$$\rho \vdash \mathrm{number}\,\mathrm{N} \rightarrow quote(int\,\mathrm{N}) \tag{2}$$

$$\rho \vdash \mathrm{true} \rightarrow quote(bool\ \text{“true”}) \tag{3}$$

$$\rho \vdash \mathrm{false} \rightarrow quote(bool\ \text{“false”}) \tag{4}$$

$$\frac{\rho \overset{\text{access}}{\vdash} \mathrm{ident}\,\mathrm{I} \mapsto c}{\rho \vdash \mathrm{ident}\,\mathrm{I} \rightarrow c} \tag{5}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad \rho \vdash \mathrm{E}_2 \rightarrow c_2 \qquad \rho \vdash \mathrm{E}_3 \rightarrow c_3}{\rho \vdash \mathrm{if}\ \mathrm{E}_1\ \mathrm{then}\ \mathrm{E}_2\ \mathrm{else}\ \mathrm{E}_3 \rightarrow push; c_1; branch(c_2, c_3)} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad \rho \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) \rightarrow push; c_1; swap; c_2; cons} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \mathrm{let}\ \mathrm{P} = \mathrm{E}_1\ \mathrm{in}\ \mathrm{E}_2 \rightarrow push; c_1; cons; c_2} \tag{8}$$

$$\frac{(\rho, \mathrm{P}) \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \mathrm{letrec}\ \mathrm{P} = \mathrm{E}_1\ \mathrm{in}\ \mathrm{E}_2 \rightarrow push; rec(c_1); cons; c_2} \tag{9}$$

$$\frac{(\rho, \mathrm{P}) \vdash \mathrm{E} \rightarrow c}{\rho \vdash \lambda \mathrm{P}.\mathrm{E} \rightarrow cur(c)} \tag{10}$$

$$\frac{\rho \vdash \mathrm{E}_2 \rightarrow c_2 \qquad \overset{\text{trans\_const}}{\vdash} \mathrm{E}_1 \rightarrow c_1}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 \rightarrow c_2; c_1} \quad (\ is\_const(\mathrm{E}_1)\ ) \tag{11}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad \rho \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 \rightarrow push; c_1; swap; c_2; cons; app} \tag{12}$$

end ML_CAM

Figure 6. Translation from Mini-ML to CAM

9.2. *Comments on the formal system*

Translation of an ML program is invoked, in rule 1, with an initial environment *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.

set ACCESS is

$\varphi, \varphi_1 : ENV;$

$$\frac{\rho \mapsto \emptyset \vdash \text{X} : c}{\rho \vdash \text{X} : c} \tag{1}$$

$$\text{ident}\,\text{X} \mapsto c \cdot \varphi \vdash \text{ident}\,\text{X} : c \tag{2}$$

$$\frac{\varphi \vdash \text{ident}\,\text{X} : c}{\text{ident}\,\text{Y} \mapsto c' \cdot \varphi \vdash \text{ident}\,\text{X} : c} \qquad (\text{Y} \neq \text{X}) \tag{3}$$

$$\frac{\rho_2 \mapsto c; cdr \cdot \rho_1 \mapsto c; car \cdot \varphi \vdash \text{X} : c'}{(\rho_1, \rho_2) \mapsto c \cdot \varphi \vdash \text{X} : c'} \tag{4}$$

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 shows, as in the dynamic semantics of Mini-ML, the similarity between a let and a letrec. The only difference is the use of the *rec* instruction in the translation of the letrec. The remaining rules deal with closures. Rule 10 merely generates the instruction *cur* that constructs closures. Rule 11 concerns predefined operators. Finally, rule 12 is the general case for an application.

10. 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.

10.1. *Translation of semantic values*

We need here two auxiliary definitions. Given ρ, an environment used by ML, consisting of a list of mapping of the form $((\text{P} \mapsto \alpha, \text{Q} \mapsto \beta), \text{R} \mapsto \gamma)$, we define:

Definition 9.1. *$\bar{\rho}$ is an environment used by the translation. It is the 'pattern' corresponding to ρ and defined by the rules:*

- $\overline{\emptyset} = \_$
- $\overline{\text{P} \mapsto \alpha} = \text{P}$
- $\overline{(\rho_1, \rho)} = (\bar{\rho}_1, \bar{\rho})$

Definition 9.2. *$\vec{\rho}$ is a value put on the stack used by CAM. It is the 'term' corresponding to ρ and defined by:*

- $\vec{\emptyset} = \mathit{0}$
- $\overline{\text{P} \mapsto \alpha} = t(\alpha)$
- $\overline{(\rho_1, \rho)} = (\vec{\rho}_1, \vec{\rho})$

Example. For $\rho = ((\text{P} \mapsto 1, \text{Q} \mapsto 2), \text{R} \mapsto 3)$, we have $\bar{\rho} = (((\_, \text{P}), \text{Q}), \text{R})$ and $\vec{\rho} = (((\mathit{0}, 1), 2), 3)$.

The definition of t on semantic values is given in Fig. 8. It is quite natural, with the possible exception of closures.

set T is

$\alpha, \beta : VALUE;\ mix : CAML;$

$$\vdash t(\mathrm{N}) = \mathrm{N} \tag{1}$$

$$\vdash t(\mathrm{T}) = \mathrm{T} \tag{2}$$

$$\vdash t((\alpha, \beta)) = (t(\alpha), t(\beta)) \tag{3}$$

$$\frac{\overset{\text{trans\_const}}{\vdash} \quad \mathrm{OP} \to \mathrm{OP}'}{\vdash t(\mathrm{ident\,OP}) = [\![cdr; \mathrm{OP}', \mathit{0}]\!]} \tag{4}$$

$$\frac{\bar{\rho} \overset{\text{ml\_cam}}{\vdash} \lambda\mathrm{P.E} \to cur(c)}{\vdash t([\![\lambda\mathrm{P.E}, \rho]\!]) = [\![c, \vec{\rho}]\!]} \tag{5}$$

$$\vdash t(\bot) = \bot \tag{6}$$

end T

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 λ-expression is the corresponding closure(Rule 5). Finally t is the identity on $\bot$ (Rule 6). 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 $\vec{\rho}$ and $\vec{\rho}$ is defined in term of $t(\alpha)$. In fact, there is no mystery: the translation of a graph is a graph. For example, consider $\rho = \mathrm{P} \mapsto [\![\mathrm{E}, \rho]\!]$. We have $\bar{\rho} = \mathrm{P}$ and $\dfrac{[\mathrm{P}] \overset{\text{ml\_cam}}{\vdash} \mathrm{E} \to cur(c)}{\vec{\rho} = t([\![\mathrm{E}, \rho]\!]) = [\![c, \vec{\rho}]\!]}$. So $\alpha = [\![\mathrm{E}, \rho]\!] = [\![\mathrm{E}, \mathrm{P} \mapsto \alpha]\!]$ is a graph and $t(\alpha) = [\![c, t(\alpha)]\!]$ is also a graph.

We are now ready to proceed with the complete proof. Unfortunately, dealing with infinite terms have made our languages non-deterministic, so we have to prove that both Rule 1 and Rule 2' hold. We will see, however, that, as this non-determinism only concerns infinite terms, the proof of rule 2' is very short.

10.2. *Proof of rule 1*

We have to prove that the following inference rule:

$$\frac{\overset{\text{ml\_ds}}{\vdash} e : \alpha \qquad \overset{\text{ml\_cam}}{\vdash} e \to c}{\overset{\text{cam\_ds}}{\vdash} c : t(\alpha)}$$

is valid in the theory $\mathcal{T} = \mathrm{ml\_ds} \cup \mathrm{ml\_cam} \cup \mathrm{cam\_ds} \cup \mathrm{t}$. For each proof tree for the premisses we must exhibit a proof tree for the conclusion. Let's make one step in this direction. We must construct the following proof tree:

$$\dfrac{\dfrac{\dfrac{init\_env \overset{\text{ml\_ds}}{\vdash} e : \alpha}{\overset{\text{ml\_ds}}{\vdash} e : \alpha} \qquad \dfrac{init\_pat \overset{\text{ml\_cam}}{\vdash} e \to c}{\overset{\text{ml\_cam}}{\vdash} e \to c}}{init\_stack \overset{\text{cam\_ds}}{\vdash} c : t(\alpha)}}{\overset{\text{cam\_ds}}{\vdash} c : t(\alpha)}$$

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, $\overline{init\_env} = init\_pat$ and $\vec{init\_env} = init\_stack$. So we shall prove that, for all expressions e of Mini-ML, for all environment ρ, and for all stack s of CAM:

$$\frac{\rho \overset{\text{ml\_ds}}{\vdash} e : \alpha \qquad \bar{\rho} \overset{\text{ml\_cam}}{\vdash} e \rightarrow c}{\vec{\rho} \cdot \mathrm{s} \overset{\text{cam\_ds}}{\vdash} c : t(\alpha) \cdot \mathrm{s}}$$

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

For each step of the induction, we will 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 $\rho \overset{\text{ml\_ds}}{\vdash} e : \alpha$ and $\bar{\rho} \overset{\text{ml\_cam}}{\vdash} e \rightarrow c$. The most interesting case, for which structural induction on the source term is not powerfull enough, is the general case of an application (Rules ml_ds.10 & ml_cam.13).

The $\perp$ case is trivial:

$\perp$-Rules in ml_ds & ml_cam:

$$\frac{\rho \vdash e : \perp \qquad \bar{\rho} \vdash e \rightarrow c}{\vec{\rho} \cdot \mathrm{s} \vdash c : \perp \cdot \mathrm{s}}$$

For simple cases, when there is one inference rule in ml_ds and one in ml_cam (with the exception of the $\perp$ - rule) dealing with a given constructor of ML, this looks like structural induction.

Rules ml_ds.1 & ml_cam.2: $e = \text{number}\,\mathrm{N}$.

$$\frac{\rho \vdash \text{number}\,\mathrm{N} : \mathrm{N} \qquad \bar{\rho} \vdash \text{number}\,\mathrm{N} \rightarrow quote(int\,\mathrm{N})}{\vec{\rho} \cdot \mathrm{s} \vdash quote(int\,\mathrm{N}) : \mathrm{N} \cdot \mathrm{s}}$$

For this we have only used rules ml_ds.2 & ml_cam.2 and rules cam_ds.5. No induction was needed as the proof trees are of length 1.

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: $e = \lambda\mathrm{P.E}$.

$$\frac{\rho \vdash \lambda\mathrm{P.E} : [\![\lambda\mathrm{P.E}, \rho]\!] \qquad \dfrac{(\bar{\rho}, \mathrm{P}) \vdash \mathrm{E} \rightarrow c}{\bar{\rho} \vdash \lambda\mathrm{P.E} \rightarrow cur(c)}}{\vec{\rho} \cdot \mathrm{s} \vdash cur(c) : [\![c, \vec{\rho}]\!] \cdot \mathrm{s}}$$

Rules ml_ds.5 & ml_cam.5: $e = \text{ident}\,\mathrm{I}$.

$$\frac{\dfrac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}\,\mathrm{I} \mapsto \alpha}{\rho \vdash \text{ident}\,\mathrm{I} : \alpha} \qquad \dfrac{\bar{\rho} \overset{\text{access}}{\vdash} \text{ident}\,\mathrm{I} \mapsto c}{\bar{\rho} \vdash \text{ident}\,\mathrm{I} \rightarrow c}}{\vec{\rho} \cdot \mathrm{s} \vdash c : t(\alpha) \cdot \mathrm{s}} \quad \text{(lemma “environment simulation”)}$$

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 will 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’.

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}\,\mathrm{I} \mapsto \alpha \qquad \bar{\rho} \overset{\text{access}}{\vdash} \text{ident}\,\mathrm{I} \mapsto c}{\vec{\rho} \cdot \mathrm{s} \vdash c : t(\alpha) \cdot \mathrm{s}}$$

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

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}\,\mathrm{I} \mapsto \alpha \qquad \varphi_{c'}(\bar{\rho}) \overset{\text{access}}{\vdash} \text{ident}\,\mathrm{I} \mapsto c'; c}{\vec{\rho} \cdot \mathrm{s} \vdash c : t(\alpha) \cdot \mathrm{s}}$$

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

- $\varphi_c(\emptyset) = \emptyset$
- $\varphi_c(\rho) = \rho \mapsto c$
- $\varphi_c(\rho, \rho_1) = \varphi_{c;cdr}(\rho_1) \cdot \varphi_{c;car}(\rho)$

For example, $\varphi_c[(((\_, \mathrm{R}), \mathrm{Q}), \mathrm{P})] = \mathrm{P} \mapsto c; cdr \cdot \mathrm{Q} \mapsto c; car; cdr \cdot \mathrm{R} \mapsto c; car; car$.

□ *Lemma "environment simulation"*

From now on we will 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.6 & ml_cam.6: $e = \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3$.

$$\dfrac{\dfrac{\rho \vdash \mathrm{E}_1 : \text{“true”} \qquad \rho \vdash \mathrm{E}_2 : \alpha}{\rho \vdash \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3 : \alpha} \quad \dfrac{\bar{\rho} \vdash \mathrm{E}_1 \to c_1 \qquad \bar{\rho} \vdash \mathrm{E}_2 \to c_2 \qquad \bar{\rho} \vdash \mathrm{E}_3 \to c_3}{\bar{\rho} \vdash \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3 \to push; c_1; branch(c_2, c_3)}}{\dfrac{\vec{\rho} \cdot \vec{\rho} \cdot \mathrm{s} \vdash c_1 : \text{“true”} \cdot \vec{\rho} \cdot \mathrm{s}\ \text{(induction)} \qquad \dfrac{\vec{\rho} \cdot \mathrm{s} \vdash c_2 : t(\alpha) \cdot \mathrm{s}\ \text{(induction)}}{\text{“true”} \cdot \vec{\rho} \cdot \mathrm{s} \vdash branch(c_2, c_3) : t(\alpha) \cdot \mathrm{s}}}{\vec{\rho} \cdot \mathrm{s} \vdash push; c_1; branch(c_2, c_3) : t(\alpha) \cdot \mathrm{s}}}$$

Rules ml_ds.7 & ml_cam.6: $e = \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3$.

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

Rules ml_ds.8 & ml_cam.7: $e = (\mathrm{E}_1, \mathrm{E}_2)$.

$$\dfrac{\dfrac{\rho \vdash \mathrm{E}_1 : \alpha \ \ \rho \vdash \mathrm{E}_2 : \beta}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) : (\alpha, \beta)} \ \dfrac{\bar{\rho} \vdash \mathrm{E}_1 \to c_1 \quad \bar{\rho} \vdash \mathrm{E}_2 \to c_2}{\bar{\rho} \vdash (\mathrm{E}_1, \mathrm{E}_2) \to push; c_1; swap; c_2; cons}}{\dfrac{\dfrac{\vec{\rho} \cdot \vec{\rho} \cdot \mathrm{s} \vdash c_1 : t(\alpha) \cdot \vec{\rho} \cdot \mathrm{s}\ \text{(induction)}}{\vec{\rho} \cdot t(\alpha) \cdot \mathrm{s} \vdash c_2 : t(\beta) \cdot t(\alpha) \cdot \mathrm{s}\ \text{(induction)}}}{\vec{\rho} \cdot \mathrm{s} \vdash push; c_1; swap; c_2; cons : (t(\alpha), t(\beta)) \cdot \mathrm{s}}}$$

For the next two cases, we shall use a Lemma on eval, which will be taken as hypothesis:

Lemma 'eval$_1$'.

$$\frac{\overset{\text{eval}}{\vdash} \mathrm{OP}, \alpha : \beta \qquad \overset{\text{trans\_const}}{\vdash} \text{ident}\,\mathrm{OP} \to c}{\overset{\text{eval}}{\vdash} c, t(\alpha) : t(\beta)}$$

Rules ml_ds.9 & ml_cam.11: $e = \mathrm{E}_1\ \mathrm{E}_2$ with $\mathrm{E}_1 = \text{ident}\,\mathrm{OP}$.

$$\dfrac{\dfrac{\rho \vdash \mathrm{E}_1 : \text{ident}\,\mathrm{OP} \quad \rho \vdash \mathrm{E}_2 : \alpha \quad \overset{\text{eval}}{\vdash} \mathrm{OP}, \alpha : \beta}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : \beta} \qquad \dfrac{\bar{\rho} \vdash \mathrm{E}_2 \to c_2 \qquad \overset{\text{trans\_const}}{\vdash} \mathrm{E}_1 \to c_1}{\bar{\rho} \vdash \mathrm{E}_1\ \mathrm{E}_2 \to c_2; c_1}}{\dfrac{\vec{\rho} \cdot \mathrm{s} \vdash c_2 : t(\alpha) \cdot \mathrm{s}\ \text{(induction)} \qquad \dfrac{\overset{\text{eval}}{\vdash} c_1, t(\alpha) : t(\beta)\ \text{(lemma “}eval_1\text{”)}}{t(\alpha) \cdot \mathrm{s} \vdash c_1 : t(\beta) \cdot \mathrm{s}}}{\vec{\rho} \cdot \mathrm{s} \vdash c_2; c_1 : t(\beta) \cdot \mathrm{s}}}$$

Rules ml_ds.9 & ml_cam.13: $e = \mathrm{E}_1\,\mathrm{E}_2$ with E_1 executes to $\mathrm{ident}\,\mathrm{OP}$.

$$\begin{array}{c}
\dfrac{\rho \vdash \mathrm{E}_1 : \mathrm{ident}\,\mathrm{OP} \quad \rho \vdash \mathrm{E}_2 : \alpha \quad \overset{\mathrm{eval}}{\vdash}\ \mathrm{OP}, \alpha : \beta}{\rho \vdash \mathrm{E}_1\,\mathrm{E}_2 : \beta} \qquad \dfrac{\begin{array}{c} \overset{\mathrm{trans_const}}{\vdash}\ \mathrm{ident}\,\mathrm{OP} \to t(op) \\ \bar{\rho} \vdash \mathrm{E}_1 \to c_1 \qquad \bar{\rho} \vdash \mathrm{E}_2 \to c_2 \end{array}}{\bar{\rho} \vdash \mathrm{E}_1\,\mathrm{E}_2 \to push; c_1; swap; c_2; cons; app} \\ \hline
\overset{\mathrm{eval}}{\vdash}\ t(op), t(\alpha) : t(\beta) \text{ (lemma ``eval'')} \\ \hline
t(\alpha) \cdot \mathrm{s} \vdash t(op) : t(\beta) \cdot \mathrm{s} \\ \hline
(0, t(\alpha)) \cdot \mathrm{s} \vdash cdr; t(op) : t(\beta) \cdot \mathrm{s} \\ \hline
([\![cdr; t(op), 0]\!], t(\alpha)) \cdot \mathrm{s} \vdash app : t(\beta) \cdot \mathrm{s} \\ \hline
\vec{\rho} \cdot [\![cdr; t(op), 0]\!] \cdot \mathrm{s} \vdash c_2 : t(\alpha) \cdot [\![cdr; t(op), 0]\!] \cdot \mathrm{s} \text{ (induction)} \\ \hline
\vec{\rho} \cdot \vec{\rho} \cdot \mathrm{s} \vdash c_1 : [\![cdr; t(op), 0]\!] \cdot \vec{\rho} \cdot \mathrm{s} \text{ (induction)} \\ \hline
\vec{\rho} \cdot \mathrm{s} \vdash push; c_1; swap; c_2; cons; app : t(\beta) \cdot \mathrm{s}
\end{array}$$

Rules ml_ds.10 & ml_cam.13: $e = \mathrm{E}_1\,\mathrm{E}_2$, general case.

$$\begin{array}{c}
\dfrac{\rho \vdash \mathrm{E}_1 : [\![\lambda \mathrm{P}.\mathrm{E}, \rho_1]\!] \quad \rho \vdash \mathrm{E}_2 : \alpha \quad (\rho_1, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E} : \beta}{\rho \vdash \mathrm{E}_1\,\mathrm{E}_2 : \beta} \qquad \dfrac{\begin{array}{c} (\bar{\rho}_1, \mathrm{P}) \vdash \mathrm{E} \to c \\ \bar{\rho} \vdash \mathrm{E}_1 \to c_1 \qquad \bar{\rho} \vdash \mathrm{E}_2 \to c_2 \end{array}}{\bar{\rho} \vdash \mathrm{E}_1\,\mathrm{E}_2 \to push; c_1; swap; c_2; cons; app} \\ \hline
(\vec{\rho}_1, t(\alpha)) \cdot \mathrm{s} \vdash c : t(\beta) \cdot \mathrm{s} \text{ (induction)} \\ \hline
([\![c, \vec{\rho}_1]\!], t(\alpha)) \cdot \mathrm{s} \vdash app : t(\beta) \cdot \mathrm{s} \\ \hline
\vec{\rho} \cdot [\![c, \vec{\rho}_1]\!] \cdot \mathrm{s} \vdash c_2 : t(\alpha) \cdot [\![c, \vec{\rho}_1]\!] \cdot \mathrm{s} \text{ (induction)} \\ \hline
\vec{\rho} \cdot \vec{\rho} \cdot \mathrm{s} \vdash c_1 : [\![c, \vec{\rho}_1]\!] \cdot \vec{\rho} \cdot \mathrm{s} \text{ (induction)} \\ \hline
\vec{\rho} \cdot \mathrm{s} \vdash push; c_1; swap; c_2; cons; app : t(\beta) \cdot \mathrm{s}
\end{array}$$

Rules ml_ds.11 & ml_cam.8: $e = \mathrm{let}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2$.

$$\begin{array}{c}
\dfrac{\rho \vdash \mathrm{E}_1 : \alpha \qquad (\rho, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E}_2 : \beta}{\rho \vdash \mathrm{let}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2 : \beta} \qquad \dfrac{\bar{\rho} \vdash \mathrm{E}_1 \to c_1 \qquad (\bar{\rho}, \mathrm{P}) \vdash \mathrm{E}_2 \to c_2}{\bar{\rho} \vdash \mathrm{let}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2 \to push; c_1; cons; c_2} \\ \hline
\vec{\rho} \cdot \vec{\rho} \cdot \mathrm{s} \vdash c_1 : t(\alpha) \cdot \vec{\rho} \cdot \mathrm{s} \text{ (induction)} \qquad (\vec{\rho}, t(\alpha)) \cdot \mathrm{s} \vdash c_2 : t(\beta) \cdot \mathrm{s} \text{ (induction)} \\ \hline
\vec{\rho} \cdot \mathrm{s} \vdash push; c_1; cons; c_2 : t(\beta) \cdot \mathrm{s}
\end{array}$$

Rules ml_ds.12 & ml_cam.9: $e = \mathrm{letrec}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2$.

$$\begin{array}{c}
\dfrac{(\rho, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E}_1 : \alpha \qquad (\rho, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E}_2 : \beta}{\rho \vdash \mathrm{letrec}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2 : \beta} \qquad \dfrac{(\rho, \mathrm{P}) \vdash \mathrm{E}_1 \to c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \to c_2}{\rho \vdash \mathrm{letrec}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2 \to push; rec(c_1); cons; c_2} \\ \hline
(\vec{\rho}, \alpha) \cdot \vec{\rho} \cdot s \vdash c_1 : \alpha \cdot \vec{\rho} \cdot s \text{ (induction)} \\ \hline
\vec{\rho} \cdot \vec{\rho} \cdot s \vdash rec(c_1) : \alpha \cdot \vec{\rho} \cdot s \qquad (\vec{\rho}, \alpha) \cdot s \vdash c_2 : \beta \cdot s \text{ (induction)} \\ \hline
\vec{\rho} \cdot s \vdash push; rec(c_1); cons; c_2 : \beta \cdot s
\end{array}$$

□ *Rule 1*

10.3. *Proof of rule 2'*

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

$$\exists \alpha \qquad \dfrac{\overset{\mathrm{ml\_cam}}{\vdash}\ e \to c \qquad \overset{\mathrm{cam\_ds}}{\vdash}\ c : \alpha'}{\overset{\mathrm{ml\_ds}}{\vdash}\ e : \alpha \qquad \overset{\mathrm{t}}{\vdash} \alpha \to \alpha'}$$

As before, we shall prove a stronger rule:

$$\exists \alpha, \rho \qquad \dfrac{\bar{\rho}\ \overset{\mathrm{ml\_cam}}{\vdash}\ e \to c \qquad \vec{\rho} \cdot \mathrm{s}\ \overset{\mathrm{cam\_ds}}{\vdash}\ c : \alpha' \cdot \mathrm{s}}{\rho\ \overset{\mathrm{ml\_ds}}{\vdash}\ e : \alpha \qquad \overset{\mathrm{t}}{\vdash} \alpha \to \alpha'}$$

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:

$$\exists\alpha,\rho \qquad \frac{\bar\rho \overset{\text{ml\_cam}}{\vdash} e\rightarrow c \qquad \vec\rho\cdot \mathrm{s} \overset{\text{cam\_ds}}{\vdash} c:\alpha'\cdot \mathrm{s}}{\overset{\text{ml\_ds}\cup\text{t}}{\vdash} \rho\Rightarrow e:\alpha \;\&\; \alpha\rightarrow\alpha'}$$

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

Lemma Cam. Every Cam code translating an ML-expression preserves the stack. More formally:

$$\exists\beta \qquad \frac{\rho \overset{\text{l\_cam}}{\vdash} e\rightarrow c \qquad \alpha\cdot s \overset{\text{cam\_ds}}{\vdash} c:s'}{\alpha\cdot s \overset{\text{cam\_ds}}{\vdash} c:\beta\cdot s}$$

Proof. The proof is obvious, by induction on the length of the proof, except maybe for rule ml_cam.12:

$$\begin{array}{c}
\dfrac{\rho\vdash \mathrm{E}_1\rightarrow c_1 \qquad \begin{array}{c}\rho'\vdash \mathrm{E}_3\rightarrow c_3\\ \rho\vdash \mathrm{E}_2\rightarrow c_2\end{array}}{\rho\vdash \mathrm{E}_1\,\mathrm{E}_2\rightarrow push; c_1; swap; c_2; cons; app} \qquad \alpha\cdot\mathrm{s}\vdash push; c_1; swap; c_2; cons; app:\mathrm{s}' \\ \hline
(\vec\rho_1,\gamma)\cdot\mathrm{s}\vdash c_3:\delta\cdot\mathrm{s}\ \text{(induction)} \\ \hline
([\![c_3,\vec\rho_1]\!],\gamma)\cdot\mathrm{s}\vdash app:\delta\cdot\mathrm{s} \\ \hline
\alpha\cdot\beta\cdot\mathrm{s}\vdash c_2:\gamma\cdot\beta\cdot\mathrm{s}\ \text{(induction)} \\ \hline
\alpha\cdot\alpha\cdot\mathrm{s}\vdash c_1:\beta\cdot\alpha\cdot\mathrm{s}\ \text{(induction)} \\ \hline
\alpha\cdot\mathrm{s}\vdash push; c_1; swap; c_2; cons; app:\delta\cdot\mathrm{s}
\end{array}$$

□ *Lemma Cam*

Rule ml_cam.12: $e = \mathrm{E}_1\,\mathrm{E}_2$, 1st case.

$$\begin{array}{c}
\dfrac{\bar\rho\vdash \mathrm{E}_1\rightarrow c_1 \qquad \dfrac{(\bar\rho_1,\mathrm{P})\vdash \mathrm{E}\rightarrow c}{\bar\rho\vdash \mathrm{E}_2\rightarrow c_2}}{\bar\rho\vdash \mathrm{E}_1\,\mathrm{E}_2\rightarrow push; c_1; swap; c_2; cons; app} \qquad
\begin{array}{c}
\overline{(\vec\rho_1,\beta')\cdot\mathrm{s}\vdash c:\gamma'\cdot\mathrm{s}\ \text{(Hyp.1)}} \\ \hline
(\alpha',\beta')\cdot\mathrm{s}\vdash app:\gamma'\cdot\mathrm{s} \\ \hline
\vec\rho\cdot\alpha'\cdot\mathrm{s}\vdash c_2:\beta'\cdot\alpha'\cdot\mathrm{s} \\ \hline
\vec\rho\cdot\vec\rho\cdot\mathrm{s}\vdash c_1:\alpha'\cdot\vec\rho\cdot\mathrm{s} \\ \hline
\vec\rho\cdot\mathrm{s}\vdash push; c_1; swap; c_2; cons; app:\gamma'\cdot\mathrm{s}
\end{array} \\ \hline
\rho\vdash \mathrm{E}_2:\beta \qquad \vdash\beta\rightarrow\beta'\ \text{(ind.)} \qquad \rho\vdash \mathrm{E}_1:\alpha=[\![\lambda\mathrm{P.E},\rho_1]\!] \qquad \vdash\alpha\rightarrow\alpha'\ \text{(ind. \& Hyp.1)} \\
\mathrm{P}\mapsto\beta\cdot\rho_1\vdash \mathrm{E}:\gamma \qquad \vdash\gamma\rightarrow\gamma'\ \text{(ind.)} \\ \hline
\rho\vdash \mathrm{E}_1\,\mathrm{E}_2:\gamma \qquad \vdash\gamma\rightarrow\gamma'
\end{array}$$

We made here one hypothesis on the form that can have α' for making the apply rule in cam_ds applicable. Hyp.1: $\alpha' = [\![c, v(\rho_1)]\!] \Rightarrow \alpha = [\![\lambda\mathrm{P.E},\rho_1]\!]$.

Rule ml_cam.12: $e = \mathrm{E}_1\,\mathrm{E}_2$, 2nd case. The alternative hypothesis says that α' is a closure again, but translation of an identifier. Hyp.2: $\alpha' = [\![cdr; top, 0]\!] \Rightarrow \alpha = \mathrm{ident}\,\mathrm{OP} \Rightarrow \beta' = \gamma'_1$. We use here a second lemma on eval, similar to the previous one, and taken as hypothesis as well:

Lemma eval$_2$.

$$\exists\alpha,\beta,\gamma \qquad \frac{\overset{\text{trans\_const}}{\vdash} \mathrm{ident}\,\mathrm{OP}\rightarrow c \qquad \overset{\text{eval}}{\vdash} c,\alpha':\beta'}{\overset{\text{eval}}{\vdash} \mathrm{OP},\alpha:\beta \ \ \vdash\alpha\rightarrow\alpha' \ \ \vdash\beta\rightarrow\beta'}$$

Then we can draw our proof tree.

$$
\dfrac{\dfrac{\dfrac{\bar{\rho} \vdash \textsc{e}_1 \rightarrow c_1 \qquad \bar{\rho} \vdash \textsc{e}_2 \rightarrow c_2}{\bar{\rho} \vdash \textsc{e}_1\ \textsc{e}_2 \rightarrow push; c_1; swap; c_2; cons; app} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{\text{eval}}{\vdash}\ top, \gamma_1' : \gamma'\ \text{(Hyp.2)}}{(\mathit{0}, \beta')\cdot \mathrm{s} \vdash cdr; top : \gamma' \cdot \mathrm{s}\ \text{(Hyp.2)}}}{(\alpha', \beta')\cdot \mathrm{s} \vdash app : \gamma' \cdot \mathrm{s}}}{\vec{\rho} \cdot \alpha' \cdot \mathrm{s} \vdash c_2 : \beta' \cdot \alpha' \cdot \mathrm{s}}}{\vec{\rho} \cdot \vec{\rho} \cdot \mathrm{s} \vdash c_1 : \alpha' \cdot \vec{\rho} \cdot \mathrm{s}}}{\vec{\rho} \cdot \mathrm{s} \vdash push; c_1; swap; c_2; cons; app : \gamma' \cdot \mathrm{s}}}{\rho \vdash \textsc{e}_2 : \beta \qquad \vdash \beta \rightarrow \beta'\ \text{(ind.)} \qquad \rho \vdash \textsc{e}_1 : \alpha = \text{ident}\ \textsc{op} \qquad \vdash \alpha \rightarrow \alpha'\ \text{(ind. \& Hyp.2)}}
}{\dfrac{\overset{\text{eval}}{\vdash}\ \textsc{op}, \gamma_1 : \gamma \qquad \vdash \gamma_1 \rightarrow \gamma_1' \qquad \vdash \gamma \rightarrow \gamma'\ \text{(lemma `}eval_2\text{')}}{\rho \vdash \textsc{e}_1\ \textsc{e}_2 : \gamma \qquad \vdash \gamma \rightarrow \gamma'}}
$$

□ *Rule 2'*

11. Execution of a mixed ML program

We will describe here the rules we choosed to add to the semantics of Mini-ML and CAM, in order to build the dynamic semantics of a 'mixed' ML program. We will then describe the new translation from Mini-ML to CAM. For each new formal system, we will give only the differences with the old one. The proof of correctness of this translation is given in the next section.

11.1. *The extand dynamic semantics of Mini-ML*

The new dynamic semantics of Mini-ML is partially given in Fig. 9.

program ML_DS is

use ML **use** ML_ENV

$\rho, \rho_1 : ENV;\ \alpha, \alpha_1, \beta : VALUE;$

$mix : CAML;$

$$\dfrac{init\_env \vdash mix : \alpha}{\vdash mix : \alpha} \qquad \text{(i)}$$

$$\dfrac{\rho \vdash \textsc{e}_1 : [\![\lambda \textsc{p}.mix, \rho_1]\!] \qquad \rho \vdash \textsc{e}_2 : \alpha \qquad (\rho_1, \textsc{p} \mapsto \alpha) \overset{\text{ml\_ds}}{\vdash} mix : \beta}{\rho \vdash \textsc{e}_1\ \textsc{e}_2 : \beta} \qquad (10)$$

$$\dfrac{[\rho] \overset{\text{cam\_ds}}{\vdash} c : [\alpha]}{\rho \vdash c : \alpha} \qquad (13)$$

$$\ldots$$

end ML_DS

Figure 9. Dynamic semantics of Mini-ML

We see that the main modification (from Fig. 2.) is the addition of rule 13, which enables the execution of a CAM-sub-expression in ML. ($[\rho]$ denotes a stack containing the single element ρ.) This rule is very simple, in the sense that we do not translate semantic values, nor environments, when switching from ML to CAM. That means that the CAM machine will treat a couple *pattern* $\mapsto$ *value* has if it was just a *value*. It also means that a CAM environment will be a tree rather than a 'comb' (totally unballenced tree), so that an ML environment and a CAM environment can be in one-to-one correspondance. This choice allows the compactness, and efficiency, of rule 13. Minor modifications are the changes from an ML-expression $\textsc{e}$ to a mixed expression mix, when desired. It is easy to see that the new semantics (say, ml_ds') extends the previous one (ml_ds), furthermore we have:

Fact. $\forall e \in ML \overset{ml\_ds}{\vdash} e : \alpha \Leftrightarrow \overset{ml\_ds'}{\vdash} e : \alpha.$

11.2. *The extand definition of CAM*

The new dynamic semantics of CAM is partially given in Fig. 10.

program CAM_DS is

use CAM **use** ML_ENV

$s, s_1, s_2 : STACK;$

$\alpha, \alpha_1, \beta, \rho, \rho_1 : VALUE;$

$mix : CAML;$

$$\rho \cdot s \vdash cur(\text{P}, \text{C}) : [\![\lambda \text{P}.\text{C}, \rho]\!]_{cam} \cdot s \tag{15}$$

$$\frac{(\rho, \text{P} \mapsto \alpha) \cdot s \vdash mix : s_1}{([\![\lambda \text{P}.mix, \rho]\!]_{\text{L}}, \alpha) \cdot s \vdash app : s_1} \tag{16}$$

$$\frac{\overset{\text{eval}}{\vdash} \text{OP}, \alpha : \beta}{(\text{ident}\,\text{OP}, \alpha) \cdot s \vdash app : \beta \cdot s} \tag{16'}$$

$$\frac{\rho \overset{\text{ml\_ds}}{\vdash} e : \alpha}{\rho \cdot s \vdash e : \alpha \cdot s} \tag{18}$$

$$x \mapsto \alpha \cdot s \vdash untag : \alpha \cdot s \tag{19}$$

$$\alpha \cdot s \vdash untag : \alpha \cdot s \qquad (\ \alpha \neq x \mapsto y\ ) \tag{20}$$

$$(\rho, \alpha) \cdot s \vdash tag(\text{P}) : (\rho, \text{P} \mapsto \alpha) \cdot s \tag{21}$$

$$\ldots$$

end CAM_DS

Figure 10. Definition of CAM

There are several modifications here, apart from those described in the previous subsection. Firstofall the cur instruction (rule 15) takes a pattern has an extra parameter, and produces a closure of a 'lambda-expression'. Then the app instruction (rule 16) takes such a closure, and uses the pattern to build the environment in which the execution goes on. This is necessary for maintaining the one-to-one correspondance between the CAM environment and the ML one. The easely way to build an extension of the previous semantics is to keep the old semantics of the cur and app instructions. They will be of no use for the new translation of an ML program (which will produce the new cur), but enable to execute the (old) translation of an ML program as before. The initial rule (1) has the same use. There is an additional rule for the app instruction in the case where the 'function' evaluates to a predefined operator (produced by ML). Finally, there are additional instructions untag and tag, whose use is to maintain, again, the correspondance between an ML environment and a CAM environment. As before, it is easy to prove the following:

Fact. $\forall c \in CAM,$ *(old) translation of an ML program,* $\overset{cam\_ds}{\vdash} c : \alpha \Leftrightarrow \overset{cam\_ds'}{\vdash} c : \alpha.$

11.3. *The mixed translation from Mini-ML to CAM*

The translation from Mini-ML to CAM has been substantialy modified (see Fig. 11.).

program ML_CAM is

use ML **use** *CAM*

$c, c_1, c_2, c_3 : CAM;\ \rho, \rho_1 : PAT;$

$e : ML;\ mix : CAML;$

$$\frac{null\_pat \vdash mix \rightarrow c}{\vdash mix \rightarrow c} \tag{1}$$

$$\rho \vdash \mathrm{ident}\,\mathrm{I} \rightarrow \mathrm{ident}\,\mathrm{I} \qquad (\ \mathrm{ident}\,\mathrm{I} \notin \rho\ ) \tag{5'}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad [\rho, \mathrm{P}] \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \mathrm{let}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2 \rightarrow push; c_1; cons; tag(\mathrm{P}); c_2} \tag{8}$$

$$\frac{[\rho, \mathrm{P}] \vdash \mathrm{E}_1 \rightarrow c_1 \qquad [\rho, \mathrm{P}] \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \mathrm{letrec}\,\mathrm{P} = \mathrm{E}_1 \,\mathrm{in}\, \mathrm{E}_2 \rightarrow push; rec(tag(\mathrm{P}); c_1); cons; tag(\mathrm{P}); c_2} \tag{9}$$

$$\frac{[\rho, \mathrm{P}] \vdash \mathrm{E} \rightarrow c}{\rho \vdash \lambda \mathrm{P}.\mathrm{E} \rightarrow cur(\mathrm{P}, c)} \tag{10}$$

$$\rho \vdash c \rightarrow c \tag{13}$$

...

end ML_CAM

Figure 11. Translation from Mini-ML to Cam

Firstofall, as we can translate a sub-expression, an identifier may be declared (by a let, or a letrec) outside of the expression being translated. We choosed to translate such an identifier by itself (rule 5′). [6] In rules 8 to 10, we generate the new tag instruction described in the previous subsection. The new rule (13) says that the translation of a CAM code is itself. We must say that this is a 'lasy' choice. Let see the disadvantage of this solution. Imagine we translate a sub-expression in which x is not declared. x is translated to itself. Now imagine we translate the all expression. X will stay in the code, despite the fact that it would have been replaced by an access path in the stack, if we had translated the all expression at once. This means that we (possibly) need to maintain an ML environment, even if we completely translate an ML expression into CAM. To overcome that, we would have written some special extra 'code', in the translation, to translate again a cam code, replacing, as possible, each ML identifier by its access path. This requires to 'recover' the Ml block structure corresponding to a translated code, which seems to be possible. Finally, the access paths for the identifiers are generated as before, with an additional rule (rule 4′) to generate the new untag instruction when needed (see Fig. 12.).

set ACCESS is

$\varphi, \varphi_1 : A\_ENV;$

$$\frac{\rho_2 \mapsto c; cdr; untag \cdot \rho_1 \mapsto c; car \cdot \varphi \vdash \mathrm{X} : c'}{[\rho_1, \rho_2] \mapsto c \cdot \varphi \vdash \mathrm{X} : c'} \tag{4'}$$

end ACCESS

Figure 12. Generating access paths for identifiers

[6] A (technical) consequence of that is that the translation must be done in an 'empty' environment (nullpat).

12. Proof of the partial translation from Mini-ML to CAM

We have already described: ml_ds, ml_cam and cam_ds. Now we must extend ml_cam to its corresponding partial translation, let's call it ml_cam', then we will give the formal system, t, describing the translation of semantic values.

12.1. *Partial translation from Mini-ML to CAM*

This new translation, ml_cam', must say that for each compose expression we have the choice to compile it, using ml_cam, or recursively to 'translate' each of it's sub-expression, using ml_cam'. For that we just have to add the rules (partially !) presented in Fig. 13. to ml_cam.

program ML_CAM' is

use CAML

$e, e', e_1, e_1', e_2, e_2' : \mathit{CAML};$

$$\vdash \text{number}\,\mathrm{N} \rightarrow \text{number}\,\mathrm{N} \tag{1}$$

$$\vdash \text{true} \rightarrow \text{true} \tag{2}$$

...

$$\frac{\vdash e \rightarrow e'}{\vdash \lambda \mathrm{P}.e \rightarrow \lambda \mathrm{P}.e'} \tag{7}$$

$$\frac{\vdash e_1 \rightarrow e_1' \qquad \vdash e_2 \rightarrow e_2'}{\vdash \text{let}\,\mathrm{P} = e_1 \,\text{in}\, e_2 \rightarrow \text{let}\,\mathrm{P} = e_1' \,\text{in}\, e_2'} \tag{8}$$

...

end ML_CAM'

Figure 13. Partial translation from Mini-ML to CAM

12.2. *Translation of semantic values*

We need here three auxiliary definitions. Given ρ, an environment used by ML, consisting of a list of mapping of the form $((\mathrm{P} \mapsto \alpha, \mathrm{Q} \mapsto \beta), \mathrm{R} \mapsto \gamma)$, we define $\bar{\rho}$ as the 'pattern' used by the translation and $\vec{\rho}$ as the value put on the stack used by CAM, each of them corresponding to ρ and defined as it was defined for the (complete) translation from Mini-ML to CAM. The third definition is a new one. $\bar{\rho} - \bar{\rho}_0$ is an environment used by the proof of the translation. It is the pattern corresponding to ρ from which we 'remove' the pattern corresponding to ρ_0. ρ_0 must be the 'beginning' of ρ.

Example. For $\rho = ((\mathrm{P} \mapsto 1, \mathrm{Q} \mapsto 2), \mathrm{R} \mapsto 3)$ and $\rho_0 = (\mathrm{P} \mapsto 1)$ we have $\bar{\rho} = (((\_, \mathrm{P}), \mathrm{Q}), \mathrm{R})$, $\vec{\rho} = (((\mathit{0}, 1), 2), 3)$, $\bar{\rho} - \bar{\rho}_0 = ((\_, \mathrm{Q}), \mathrm{R})$.

The definition of t on semantic values is given in Fig. 14. It is, so to say, the identity plus the same as before. So, t is no longer a function but a relation.

The only maybe non-trivial rule is the one dealing with closures. It says that the translation of a closure may be the closure of the translation of its body, in any environment where it makes sens to translate this body. As before, of course the translation of a graph is a graph.

We are now ready to proceed with the proof. We will not give here yet another complete proof, but just give the sketch of it. As before, we have to prove that both Rule 1' and Rule 2' hold. We cannot prove Rule 1 in this case, because the translation is not a function on closures, though Cam is deterministic on lambda-expressions.

12.3. *Proof of rule 1'*

set T is

$\alpha, \alpha', \beta, \beta' : \mathit{VALUE};\ \mathit{mix}, \mathit{mix}' : \mathit{CAML};$

$$\overset{\text{t}}{\vdash} \alpha \rightarrow \alpha \tag{1}$$

$$\frac{\overset{\text{t}}{\vdash} \alpha \rightarrow \alpha' \qquad \overset{\text{t}}{\vdash} \beta \rightarrow \beta'}{\overset{\text{t}}{\vdash} (\alpha, \beta) \rightarrow (\alpha', \beta')} \tag{2}$$

$$\frac{(\bar{\rho} - \bar{\rho}_0, \text{P}) \overset{\text{ml\_cam}}{\vdash} \mathit{mix} \rightarrow \mathit{mix}' \qquad \overset{\text{t}}{\vdash} \rho \rightarrow \vec{\rho}}{\overset{\text{t}}{\vdash} [\![\lambda\text{P}.\mathit{mix}, \rho]\!] \rightarrow [\![\lambda\text{P}.\mathit{mix}', \vec{\rho}]\!]} \tag{3}$$

$$\frac{\overset{\text{t}}{\vdash} \alpha \rightarrow \alpha'}{\overset{\text{t}}{\vdash} (\text{P} \mapsto \alpha) \rightarrow (\text{P} \mapsto \alpha')} \tag{4}$$

end T

Figure 14. The definition of t on semantic values

We have to prove the validity of the following rule:

$$\frac{\overset{\text{ml\_ds}}{\vdash} e : \alpha \qquad \overset{\text{ml\_cam}}{\vdash} e \rightarrow e'}{\overset{\text{ml\_ds}}{\vdash} e' : \alpha' \qquad \overset{\text{t}}{\vdash} \alpha \rightarrow \alpha'}$$

in the theory $\mathcal{T} = \text{caml\_ds} \cup \text{ml\_cam} \cup \text{t}$, where caml_ds contains both ml_ds and cam_ds. For each proof tree for the premisses we must exhibit a proof tree for the conclusion. Let's make one step in this direction. We must construct the following proof tree:

$$\frac{\dfrac{\mathit{init\_env} \overset{\text{ml\_ds}}{\vdash} e : \alpha}{\overset{\text{ml\_ds}}{\vdash} e : \alpha} \qquad \overset{\mathit{ml\_cam}'}{\vdash} e \rightarrow e'}{\dfrac{\mathit{init\_env} \overset{\text{ml\_ds}}{\vdash} e' : \alpha'}{\overset{\text{ml\_ds}}{\vdash} e' : \alpha'} \qquad \overset{\text{t}}{\vdash} \alpha \rightarrow \alpha'}$$

This suggests what inference rule we should attempt to prove. We will prove that, for all expressions e of Mini-ML, for all environment ρ,

$$\frac{\rho \overset{\text{ml\_ds}}{\vdash} e : \alpha \qquad \bar{\rho} - \bar{\rho}_0 \overset{\text{ml\_cam}}{\vdash} e \rightarrow e' \qquad \overset{\text{t}}{\vdash} \rho \rightarrow \vec{\rho}}{\vec{\rho} \overset{\text{ml\_ds}}{\vdash} e' : \alpha' \qquad \overset{\text{t}}{\vdash} \alpha \rightarrow \alpha'}$$

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

For each step of the induction, we will draw a proof tree as we have done for the previous proof of translation. This time, the proof split into two parts depanding on the translation choosen for the node of expression e. Either it is translated to some Cam code, or to itself, according to the rules of *ml_cam′*. In the first case, one of the most interesting cases is again the general case of an application:

Rules ml_ds.10 & ml_cam.12: $e = \text{E}_1\,\text{E}_2$, general case.

$$
\begin{array}{c}
\dfrac{\rho \vdash \text{E}_1 : [\![\lambda \text{P.E}, \rho_1]\!] \quad \rho \vdash \text{E}_2 : \alpha \quad (\rho_1, \text{P} \mapsto \alpha) \vdash \text{E} : \beta}{\rho \vdash \text{E}_1\,\text{E}_2 : \beta}
\quad
\dfrac{\begin{array}{c}(\bar\rho_1 - \bar\rho_0, \text{P}) \vdash \text{E} \to \text{E}' \qquad \vdash \rho \to \vec\rho \\ \bar\rho - \bar\rho_0 \vdash \text{E}_1 \to c_1 \qquad \bar\rho - \bar\rho_0 \vdash \text{E}_2 \to c_2\end{array}}{\bar\rho - \bar\rho_0 \vdash \text{E}_1\,\text{E}_2 \to push; c_1; swap; c_2; cons; app} \\
\hline
(\vec\rho_1, \text{P} \mapsto \alpha') \overset{ml\_ds}{\vdash} \text{E}' : \beta' \text{ (induction)} \qquad \vdash \beta \to \beta' \\
\hline
(\vec\rho_1, \text{P} \mapsto \alpha') \overset{cam\_ds}{\vdash} \text{E}' : \beta' \\
\hline
([\![\lambda \text{P.E}', \vec\rho_1]\!], \alpha') \vdash app : \beta' \\
\hline
\vec\rho \cdot [\![\lambda \text{P.E}', \vec\rho_1]\!] \vdash c_2 : \alpha' \cdot [\![\lambda \text{P.E}', \vec\rho_1]\!] \text{ (induction)} \qquad \vdash \alpha \to \alpha' \\
\hline
\vec\rho \cdot \vec\rho \vdash c_1 : [\![\lambda \text{P.E}', \vec\rho_1]\!] \cdot \vec\rho \text{ (induction)} \qquad \vdash \rho_1 \to \vec\rho_1 \\
\hline
\vec\rho \overset{\text{cam\_ds}}{\vdash} push; c_1; swap; c_2; cons; app : \beta' \\
\hline
\vec\rho \overset{\text{ml\_ds}}{\vdash} push; c_1; swap; c_2; cons; app : \beta' \qquad \vdash \beta \to \beta'
\end{array}
$$

The first rule used to swhitch from cam to ml (in the backward mode) is a rule of cam_ds, if E' is in ML. If E' is in Cam, then the rule used is the 'reverse' of the rule dealing with a Cam-sub-expression in ML. The reverse of this rule is valid as it is the only rule that enables to execute a Cam-sub-expression.

The second interested case is the case of the swhitching rule.

Rules ml_ds.13 & ml_cam.13: $e = c$.

$$
\dfrac{\dfrac{[\rho] \overset{\text{cam\_ds}}{\vdash} c : [\alpha]}{\rho \vdash c : \alpha} \qquad \bar\rho - \bar\rho_0 \vdash c \to c}{\rho \overset{\text{ml\_ds}}{\vdash} c : \alpha' \qquad \vdash \alpha \to \alpha'}
$$

This case is a sub-case of the second part of the proof, where we choose to translate a ML-node by itself. Let's see again the general case of an application.

Rules ml_ds.10 & ml_cam'.12: $e = \text{E}_1\,\text{E}_2$, general case.

$$
\dfrac{\dfrac{\rho \vdash \text{E}_1 : [\![\lambda \text{P.E}, \rho_1]\!] \quad \rho \vdash \text{E}_2 : \alpha \quad (\rho_1, \text{P} \mapsto \alpha) \vdash \text{E} : \beta}{\rho \vdash \text{E}_1\,\text{E}_2 : \beta}
\quad
\dfrac{\begin{array}{c}(\bar\rho_1 - \bar\rho_0, \text{P}) \vdash \text{E} \to \text{E}' \qquad \vdash \rho \to \vec\rho \\ \bar\rho - \bar\rho_0 \vdash \text{E}_1 \to \text{E}'_1 \qquad \bar\rho - \bar\rho_0 \vdash \text{E}_2 \to \text{E}'_2\end{array}}{\bar\rho - \bar\rho_0 \vdash \text{E}_1\,\text{E}_2 \to \text{E}'_1\,\text{E}'_2}}{\dfrac{\vec\rho \vdash \text{E}'_1 : [\![\lambda \text{P.E}', \vec\rho_1]\!] \quad \vdash \rho_1 \to \vec\rho_1 \quad \vec\rho \vdash \text{E}'_2 : \alpha' \quad \vdash \alpha \to \alpha' \quad (\vec\rho_1, \text{P} \mapsto \alpha') \vdash \text{E}' : \beta' \quad \vdash \beta \to \beta'}{\vec\rho \vdash \text{E}'_1\,\text{E}'_2 : \beta' \qquad \vdash \beta \to \beta'}}
$$

$\square$ *Rule 1'*

13. 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 ($\rho \vdash e : \alpha$, $\rho \vdash e \to c$ ...) in formal systems, and proofs of translation deal with validity of inference rules in the union of these formal systems. The partial translation of a program can be proved correct in the same way as we do for a complete translation.

References

[LNCSn stands for Vol n, Lecture Notes in Computer Science, Springer-Verlag].

[1] J. Barwise, "The Handbook of Mathematical Logic", North-Holland, Amsterdam, 1977, reprinted in 1983.

[2] D. Clement, J. Despeyroux, T. Despeyroux, L. Hascoet, G. Kahn, "Natural semantics on the computer", I.N.R.I.A Report RR 416, June 85.

[3] D. Clement, J. Despeyroux, T. Despeyroux, G. Kahn, "A Simple Applicative Language: Mini-ML", INRIA Report 529, May 1986. also in the Proc. of the ACM Lisp and Functional Programming, MIT, August 1986.

[4] D. Clement, "The Natural Dynamic Semantics of Mini-Standard-ML", in the proc. of the Conf. on Functional and Logic Programming, TAPSOFT'87.

[5] G. Cousineau, P. L. Curien, M. Mauny, "The categorical Abstract machine", LITP Report 85-8, University Paris VII, january 1985. also Proc. of the IFIP conference on "Functional Programming Languages and Computer Architecture", Nancy, France, Sept. 1985, LNCS 201.

[6] J. Despeyroux, "Proof of translation in natural semantics", Inria Report 514, april 1986. also in the Proc. of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986.

[7] T. Despeyroux, "Executable Specification of Static Semantics", Semantics of Data Types, 1984, LNCS 173.

[8] T. Despeyroux, "Typol: a formalism to implement Natural Semantics", to appear as an INRIA Report.

[9] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Melese, "Document stucture and modularity in Mentor", Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software development Environments, Pittsburgh, april 1984.

[10] G. Gentzen, "The Collected Papers of Gerhard Gentzen", E. Szabo, North-Holland, Amsterdam, 1969.

[11] M.J. Gordon, A.J. Milner, Christopher P. Wadsworth, "Edinburgh LCF", LNCS 78, 1979.

[12] L. Hascoët "Transformations automatiques de spécifications sémantiques. Application: un vérificateur de types incremental.", Thèsis, Nice university, March 1987.

[13] G. Huet "Formal Structures for Computation and Deduction", Courses Notes at CMU, May 1986.

[14] G. Kahn "Natural Semantics", Proc. of Symp. on Theorical Aspects of Computer Science, Passau, Germany, february 1987.

[15] J.J. Levy, "Réductions sures dans le lambda-calcul", Thèse de 3ième cycle, Paris VII, France, 1974.

[16] W. Li, "An operational approach to semantics and translation for concurent programming languages", Ph.D., Edinburgh, 1983.

[17] M. Mauny, "Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML.", Thèse de 3ième cycle, Paris VII, France, 1985.

[18] F.L. Morris, "Advice on structuring compilers and proving them correct", Principles Of Programming Languages, 1973.

[19] G.D. Plotkin, "A structural approach to operational semantics", Aarhus Report DAIMI FN-19, 1981.

[20] G.D. Plotkin, "A powerdomain for Countable Non Determinism", Proc. of the ICALP conference, Aarhus, Denmark, 1982, LNCS 140.

[21] D. Prawitz, "Natural Deduction, a Proof-Theoretical Study", Almqvist & Wiksell, Stockholm, 1965.

[22] D. Prawitz, "Ideas and results in Proof Theory", Proc. of the 2nd. Scand. Logic Congress, North Holland, 1971.

[23] C.P. WADSWORTH, "The relation between computational and denotational properties for Scott's D_∞ - Models of the lambda calculus", Siam Journal on Computing, 1976, Vol. 5.

Table of Contents