A Simple Applicative Language: Mini-ML

Dominique Clément
SEMA, Sophia-Antipolis
Joëlle Despeyroux
Thierry Despeyroux
Gilles Kahn
INRIA, Sophia-Antipolis
Route des Lucioles, 06565 Valbonne Cedex, France

Abstract.

This paper presents a formal description of the central part of the ML language in Natural Semantics. Static semantics, dynamic semantics, and translation to an abstract machine code are covered. The description has been tested on a computer and we explain why this is feasible. Several facts that one may want to prove about the language are expressed and proved within the formalism.

1. Introduction

ML is a programming language with very interesting characteristics from the standpoint of static and dynamic semantics.

- ML is a strongly typed language but there is no type declaration: expressions are typed *implicitly.*
- ML exhibits *polymorphism*: it is possible to define functions that work uniformly on arguments of many types.
- ML allows the definition of higher-order functions: the value of an ML expression may be a *closure.*

ML typechecking is the object of numerous discussions in the literature, e.g. [1], [4], [6], [13], and the use of an *inference system* to describe ML typing is now widely accepted. On the other hand recent work of Curien and Cousineau [3] has shown how to *compile* ML into code for an abstract machine, the Categorical Abstract Machine (CAM). Hence we are in a position to describe formally and completely three aspects of ML: typechecking, dynamic semantics, and translation into CAM code. Without loss of generality, we restrict ML to a central part christened Mini-ML, a simple typed λ-calculus with constants, products, conditionals, and recursive function definitions.

This work is partially supported under ESPRIT, p. 348

1.1. Sample programs

To illustrate Mini-ML, we introduce several examples in *concrete* syntax. First of course is how to write and use the factorial function:

$$\textbf{letrec } fact = \lambda x.\, \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * fact(x-1) \textbf{ in } fact\ 4$$

The next example shows the definition of a higher order function *twice*:

$$\begin{array}{l}\textbf{let } succ = \lambda x.x+1 \\ \textbf{in let } twice = \lambda f.\lambda x.(f\ (f\ x)) \\ \quad \textbf{in } ((twice\ succ)\ 0)\end{array}$$

This expression uses block structure:

$$\begin{array}{l}\textbf{let } i = 5 \\ \textbf{in let } i = i+1 \textbf{ in } i\end{array}$$

Here we have both simultaneous definitions and block structure:

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

This example has simultaneous recursive definitions:

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

Finally here is a typical example of polymorphism:

$$\textbf{let } f = \lambda x.x \textbf{ in } f\ f$$

The examples above, which we use throughout this paper, are believed to reflect the main intricacies in static and dynamic semantics of ML.

The remainder of this paper is divided into four parts. In Section 2, we discuss the static semantics of Mini-ML.

First, we present the Damas-Milner type inference system for the simple λ-calculus part of Mini-ML. Then we show how this system, which is non-deterministic, can be turned into a deterministic inference system. The latter system can be read as an *algorithm* to type-check Mini-ML expressions. Finally we give complementary rules to deal with products and *letrec*. In Section 3, the dynamic semantics of Mini-ML is also given as an inference system. The nicer points of this system are recursive function specification and handling of products. To compile Mini-ML, we use CAM as the target machine. The CAM is a very simple abstract machine. The formal system in Section 4 specifies the transitions of the machine. Finally Section 5 contains translation rules from Mini-ML to CAM, again in the same style.

1.2. Abstract Syntax of Mini-ML

The abstract syntax given below describes λ-calculus extended with let, letrec, if, and products. Furthermore, in an expression $\lambda P.e$, P may be either an identifier or a (tree-like) pattern. For example $\lambda(x,y).e$ is a valid expression and so is $\lambda(x,((y,z),t)).e'$. The constructor *mlpair* builds products of expressions, while the *pairpat* constructor serves in building patterns of identifiers. The *nullpat* constructor is used for the unit object (), which is both a pattern and an expression.

sorts EXP, IDENT, PAT, NULLPAT

subsorts EXP⊃ NULLPAT, IDENT
PAT⊃ NULLPAT, IDENT

constructors

'Patterns'

pairpat	:	PAT×PAT	→	PAT
nullpat	:		→	NULLPAT

'Expressions'

ident	:		→	IDENT
number	:		→	EXP
false	:		→	EXP
true	:		→	EXP
apply	:	EXP×EXP	→	EXP
mlpair	:	EXP×EXP	→	EXP
lambda	:	PAT×EXP	→	EXP
let	:	PAT×EXP×EXP	→	EXP
letrec	:	PAT×EXP×EXP	→	EXP
if	:	EXP×EXP×EXP	→	EXP

Figure 1. Abstract Syntax of Mini-ML

2. Static Semantics

Before introducing an inference system that assigns types in Mini-ML we must define the type language. In typed λ-calculus every object has a type. Thus the type language must be able to express *basic types* as well as *functional types*. For example, the type of the successor function $\lambda x.x+1$ is $int \to int$. In the same way the identity function $\lambda x.x$ for integers has type $int \to int$, but for booleans it has type $bool \to bool$. It is clear that the identity function may be defined without taking into account the type of its present parameter. To express this *abstraction* on the type of the parameter, the *type variable* α is bound by a quantifier: the polymorphic identity function has type $\forall\alpha.\alpha \to \alpha$.

2.1. The Type Language

The type language contains two syntactic categories, types and type schemes.

Types: a type τ is either

i. a basic type *int*, *bool*,

ii. a type variable α ,

iii. a functional type $\tau \to \tau'$, where τ and τ' are types,

iv. a product type $\tau \times \tau'$, where τ and τ' are types.

Type schemes: a type-scheme σ is either

i. a type τ,

ii. a type-scheme $\forall\alpha.\sigma$, where σ is a type-scheme.

Remark: quantifiers may occur only at the top level of type-schemes, they do not occur within type-schemes.

A type expression in this language may have both *free* and *bound* variables. Let us write $FV(\sigma)$ and $BV(\sigma)$ for the sets of free and bound variables of a type expression σ. Following [4], we now define two relations between type expressions that contain type variables.

Definition 2.1. *A type scheme σ' is called an instance of a type scheme σ if there exists a substitution S of types for free type variables such that:*

$$\sigma' = S\sigma.$$

Instantiation acts on *free* variables: if S is written $[\alpha_i \leftarrow \tau_i]$ with $\alpha_i \in FV(\sigma)$ then $S\sigma$ is obtained by replacing each free occurrence of α_i in σ by τ_i (renaming the bound variables of σ if necessary). The *domain* of S is written $D(S)$.

Definition 2.2. *A type scheme $\sigma = \forall\alpha_1 \cdots \alpha_m.\tau$ has a generic instance $\sigma' = \forall\beta_1 \cdots \beta_n.\tau'$, and we shall write $\sigma \succeq \sigma'$, if there exists a substitution S such that*

$$\tau' = S\tau \quad \text{with} \quad D(S) \subseteq \{\alpha_1 \cdots \alpha_m\}$$

and the β_i are not free in σ, i.e.

$$\beta_i \notin FV(\sigma) \qquad 1 \le i \le n.$$

Generic instantiation acts on *bound* variables. Note that if $\sigma \succeq \sigma'$ then for every substitution S, $S\sigma \succeq S\sigma'$. Note also

that if τ and τ' are types rather than type-schemes, then $\tau \succeq \tau'$ implies $\tau = \tau'$.

2.2. The Damas-Milner Inference System

In programming languages the type of an expression depends on the type of identifiers that occur free in it. In other words, an expression e has type σ under a given set of assumptions A about the type of its free variables. In the following A is a list of assumptions $\{x : \sigma\}$ and A_x stands for the result of removing any assumption about x from A. The set $FV(A)$ of free variables of A is defined by extension

$$FV(A) = \bigcup_{\{x:\sigma\} \in A} FV(\sigma).$$

We say that the expression e *has type* σ if $A \vdash e : \sigma$ can be derived from the Damas-Milner inference system in Fig. 2. *Observe that the metavariables τ and τ' in this system range over types, while σ ranges over type-schemes.*

TAUT	$A \vdash x : \sigma \qquad (\{x : \sigma\} \in A)$
INST	$\dfrac{A \vdash e : \sigma}{A \vdash e : \sigma'} \qquad (\sigma \succeq \sigma')$
GEN	$\dfrac{A \vdash e : \sigma}{A \vdash e : \forall\alpha.\sigma} \qquad (\alpha \notin FV(A))$
APP	$\dfrac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash e\, e' : \tau}$
ABS	$\dfrac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$
LET	$\dfrac{A \vdash e' : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e : \tau}{A \vdash \text{let } x = e' \text{ in } e : \tau}$

Figure 2. The Damas-Milner Inference system DM

We will use the following lemma, which is proved in [4]:

Lemma 2.1. *If $\sigma \succeq \sigma'$ and $A_x \cup \{x : \sigma'\} \vdash e : \sigma_0$ then $A_x \cup \{x : \sigma\} \vdash e : \sigma_0$.*

Examples

Let us see now how these rules may be used to prove typings. For example, we can show that the identity function $\lambda x.x$ has type $\forall\alpha.\alpha \to \alpha$ by the following derivation tree:

$$\dfrac{\dfrac{\{x : \alpha\} \vdash x : \alpha \quad [\text{TAUT}]}{\vdash \lambda x.x : \alpha \to \alpha \quad [\text{ABS}]}}{\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha \quad [\text{GEN}]}$$

We can use that proof to get a specialized type for the identity function:

$$\dfrac{\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha}{\vdash \lambda x.x : \mathit{int} \to \mathit{int}} \quad [\text{INST}]$$

More directly we have also the following proof tree:

$$\dfrac{\{x : \mathit{int}\} \vdash x : \mathit{int} \quad [\text{TAUT}]}{\vdash \lambda x.x : \mathit{int} \to \mathit{int} \quad [\text{ABS}]}$$

We can go further now and show a proof tree for the typing of let $f = \lambda x.x$ in $f\, f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\{f : \forall\alpha.\alpha \to \alpha\} \vdash f : \forall\alpha.\alpha \to \alpha}{\{f : \forall\alpha.\alpha \to \alpha\} \vdash f : \beta \to \beta} \qquad \dfrac{\dfrac{\{x : \alpha\} \vdash x : \alpha}{\vdash \lambda x.x : \alpha \to \alpha}}{\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha}}{\{f : \forall\alpha.\alpha \to \alpha\} \vdash f : (\beta \to \beta) \to (\beta \to \beta)}}{\{f : \forall\alpha.\alpha \to \alpha\} \vdash f\, f : \beta \to \beta}}{\vdash \text{let } f = \lambda x.x \text{ in } f\, f : \beta \to \beta}$$

Remark: in λ-calculus the expression let $x = e'$ in e is considered equivalent to $(\lambda x.e)\, e'$. Hence one might wonder about the need for the LET rule. Indeed, the following proof tree is obtained without using the LET rule:

$$\dfrac{\dfrac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau} \qquad A \vdash e' : \tau'}{A \vdash (\lambda x.e)\, e' : \tau}$$

so that the following rule is derivable in DM:

$$\text{LET}^- \qquad \dfrac{A \vdash e' : \tau' \qquad A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash (\lambda x.e)\, e' : \tau}$$

In rule LET, variable x may be assigned a type-scheme. But in rule LET$^-$, it may only be assigned a type. The LET rule is the source of polymorphism in ML.

Now to *compute* the type of an expression with such an inference system, i.e. given A and e derive some σ such that $A \vdash e : \sigma$, we need an *algorithm* to build a proof of $A \vdash e : \sigma$. With the exception of GEN and INST, all rules are *exclusive*: there is only one rule stating how to type each syntactic construct. But rules GEN and INST may be invoked at any time, so that the strategy to follow in constructing a proof tree is not obvious.

2.3. A Deterministic Inference System

In the examples above we notice that:

- rules APP, ABS involve only types, not type-schemes. Likewise, a *let*-expression has a type τ, not a type-scheme.
- rule GEN is the only rule that introduces quantifiers. If the TAUT rule returns a type-scheme, this type-scheme was found in the assumptions A. It might come from a predefined type, or from an earlier use of the LET rule since ABS only enters types in assumptions.
- if σ is a type-scheme, to derive the premise $A_x \cup \{x : \sigma\} \vdash e : \tau$ of the LET rule, we must use INST after each use of TAUT for x if we want to use any other rule.

These observations suggest considering the slightly modified version of the Damas-Milner system in Fig. 3:

$$\text{TAUT}' \qquad A \vdash x : \tau \qquad (x : \sigma \in A,\ \sigma \succeq \tau)$$

$$\text{APP} \qquad \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash e\, e' : \tau}$$

$$\text{ABS} \qquad \frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

$$\text{LET}' \qquad \frac{A \vdash e' : \tau' \quad A_x \cup \{x : \sigma\} \vdash e : \tau}{A \vdash \text{let } x = e' \text{ in } e : \tau} \qquad (\sigma = gen(A, \tau'))$$

Figure 3. A variant DM′ of the Damas-Milner system

where $gen(A,\tau)$ is defined by

$$gen(A,\tau) = \begin{cases} \forall \alpha_1 \cdots \alpha_n.\tau & (FV(\tau) \setminus FV(A) = \{\alpha_1 \cdots \alpha_n\} \\ \tau & (FV(\tau) \setminus FV(A) = \emptyset) \end{cases}$$

We can use DM′ instead of DM thanks to the next result (see proof in appendix A.1).

Theorem 2.1. *The system DM′ is equivalent to DM in the following sense:*

$$A \overset{\text{DM}'}{\vdash} e : \tau \Longrightarrow A \overset{\text{DM}}{\vdash} e : \tau,$$

$$\forall A, e\ \exists \tau \quad A \overset{\text{DM}}{\vdash} e : \sigma \Longrightarrow A \overset{\text{DM}'}{\vdash} e : \tau\ \&\ gen(A,\tau) \succeq \sigma.$$

But DM′ is well adapted to computing types, as we now see.

2.4. Type Inference

To understand why DM′ allows *computing* types, we examine how proof trees can be constructed systematically on several examples. First, we typecheck the function $\lambda x.succ\ x$. More precisely, we attempt to construct, for some τ, a proof of

$$\{succ : int \to int\} \vdash \lambda x.succ\ x : \tau$$

The last step of the proof *must* use the ABS rule, the only rule concerning λ-abstraction. We must now find a proof of

$$\{succ : int \to int\} \cup \{x : \tau_2\} \vdash succ\ x : \tau_1$$

with both τ_2 and τ_1 unknown. If we find this proof, then we will have succeeded with $\tau = \tau_2 \to \tau_1$, and our last proof step will be an instance of

$$\frac{\{succ : int \to int\} \cup \{x : \tau_2\} \vdash succ\ x : \tau_1}{\{succ : int \to int\} \vdash \lambda x.succ\ x : \tau_2 \to \tau_1}$$

where τ_2 and τ_1 are type metavariables. Now for $succ\ x$ we must use APP to obtain an instance of

$$\frac{\{succ : int \to int\} \cup \{x : \tau_2\} \vdash x : \tau_3 \qquad \{succ : int \to int\} \cup \{x : \tau_2\} \vdash succ : \tau_3 \to \tau_1}{\{succ : int \to int\} \cup \{x : \tau_2\} \vdash succ\ x : \tau_1}$$

The first premise must use TAUT′

$$\{succ : int \to int\} \cup \{x : \tau_2\} \vdash succ : int \to int$$

which gives the only type of *succ*. As a consequence $\tau_3 = int$ and $\tau_1 = int$. The second premise must be proved with TAUT′ again

$$\{succ : int \to int\} \cup \{x : \tau_2\} \vdash x : \tau_3 \qquad \tau_2 \succeq \tau_3$$

and since $\tau_3 = int$ we must also have $\tau_2 = int$. In other words, we have found the following proof tree:

$$\frac{\{succ : int \to int\} \cup \{x : int\} \vdash x : int \qquad \{succ : int \to int\} \cup \{x : int\} \vdash succ : int \to int}{\{succ : int \to int\} \cup \{x : int\} \vdash succ\ x : int}$$
$$\{succ : int \to int\} \vdash \lambda x.succ\ x : int \to int$$

If we represent this proof tree as a term, this term has the structure ABS(APP(TAUT′, TAUT′)),
or more precisely $\text{ABS}_x(\text{APP}(\text{TAUT}'_{succ}, \text{TAUT}'_x))$,
or even $\lambda x.\ succ\ x$. This fact is an instance of the Principal Type Theorem [8] and it is completely general:

Theorem 2.2. *If $A \vdash e : \tau$ has a proof in DM′ then the structure of this proof is e.*

Proof. The proof is by induction on the structure of e. □

In the example above, we have reasoned both on constructing the proof tree and on resolving constraints concerning type metavariables. We proceed now more systematically since by Theorem 2.2 constructing the proof tree is trivial. Consider the identity function $\lambda x.x$. Using ABS and TAUT′, we obtain the proof tree

$$\frac{\{x : \tau'\} \vdash x : \tau}{\vdash \lambda x.x : \tau' \to \tau}$$

with the constraint $\tau' \succeq \tau$. But since τ and τ' are types we have $\tau = \tau'$ so that all proof trees are instances of

$$\frac{\{x:\tau\} \vdash x:\tau}{\vdash \lambda x.x:\tau \to \tau}$$

without any more constraint on the type metavariable τ. In particular we have $\vdash \lambda x.x : \alpha \to \alpha$, with α a type variable. Since types are first order terms, any instance of α is an instance of τ, and conversely. It is very tempting to identify type variables with type metavariables.

To understand whether this is possible in general, we try to find a type for the polymorphic function

$$\text{let } i = \lambda x.x \text{ in } i\,i.$$

First we use the LET′ rule, together with the earlier result $\vdash \lambda x.x : \alpha \to \alpha$:

$$\frac{\vdash \lambda x.x : \alpha \to \alpha \qquad \{i : \forall\alpha \cdot \alpha \to \alpha\} \vdash i\,i : \tau}{\vdash \text{let } i = \lambda x.x \text{ in } i\,i : \tau}$$

To compute the type of $i\,i$ under the assumption $\{i : \forall\alpha \cdot \alpha \to \alpha\}$ we use APP, and for each occurrence of i we use TAUT′. But now, in the assumption, identifier i is associated to a type-scheme. We must find types τ_1 and τ_2 that are generic instances of $\{i : \forall\alpha \cdot \alpha \to \alpha\}$. Summing up, we have the following constraints:

$$\begin{gathered}\forall\alpha \cdot \alpha \to \alpha \succeq \tau_1 \\ \forall\alpha \cdot \alpha \to \alpha \succeq \tau_2 \\ \tau_1 = \tau_2 \to \tau\end{gathered}$$

Solving this yields $\tau_1 = \tau_3 \to \tau_3$, then $\tau_2 = \tau_4 \to \tau_4$, and the third equation yields $\tau_3 = \tau_2 = \tau_4 \to \tau_4 = \tau$. The instance of APP to use is of the form

$$\frac{\{i : \forall\alpha \cdot \alpha \to \alpha\} \vdash i : \tau_4 \to \tau_4 \qquad \{i : \forall\alpha \cdot \alpha \to \alpha\} \vdash i : (\tau_4 \to \tau_4) \to (\tau_4 \to \tau_4)}{\{i : \forall\alpha \cdot \alpha \to \alpha\} \vdash i\,i : \tau_4 \to \tau_4}$$

and without loss of generality we can take $\tau_4 = \beta$ resulting in

$$\vdash \text{let } i = \lambda x.x \text{ in } i\,i : \beta \to \beta$$

To see that λ-bound variables and let-bound variables are treated differently, we conclude with a last example. Assume that the innocuous axiom INT

$$A \vdash \text{int x} : \mathit{int}$$

has been added to DM′, and try to find some τ such that

$$\vdash \lambda x.\text{let } i = x \text{ in } i\,1 : \tau$$

We produce mechanically the proof tree

$$\frac{\{x:\tau_1\} \vdash x:\tau_3 \qquad \dfrac{\{i:\sigma\} \vdash i:\tau_5 \qquad \vdash 1:\tau_4}{\{i:\sigma\} \vdash i\,1:\tau_2}}{\dfrac{\{x:\tau_1\} \vdash \text{let } i = x \text{ in } i\,1 : \tau_2}{\vdash \lambda x.\text{let } i = x \text{ in } i\,1 : \tau}}$$

and the constraints:

$$\frac{\tau_1 \succeq \tau_3 \qquad \dfrac{\sigma \succeq \tau_5 \qquad \tau_4 = \mathit{int}}{\tau_5 = \tau_4 \to \tau_2}}{\dfrac{\sigma = \mathit{gen}(\{x:\tau_1\},\tau_3)}{\tau = \tau_1 \to \tau_2}}$$

Each use of an axiom or an inference rule gives rise to precisely one constraint. Equalities between meta-variables define an equivalence relation. The remaining constraints are of the form: $\tau = \tau_1 \to \tau_2$, $\sigma = \mathit{gen}(A,\tau)$, and $\sigma \succeq \tau$. Now to solve the constraints at one level, we first solve the constraints for the above sub-trees (computing the type of an expression from the types of its sub-expressions), except for the LET′ rule. There the constraints are solved in mid-order: first solve the constraints for $A \vdash e' : \tau'$, then compute $\sigma = \mathit{gen}(A,\tau')$, and finally solve the constraints for $A_x \cup \{x:\sigma\} \vdash e : \tau$. To compute $\mathit{gen}(A,\tau')$, every type metavariable of τ' that does not occur in $FV(A)$ is considered as a type variable and generalised. Thanks to the lemma 2.1 (and 2.2 in appendix A.1), this complete generalisation does not restrict the resolution of the constraints for the second subtree, i.e. for $A_x \cup \{x:\sigma\} \vdash e : \tau$.

Returning to our example, from $\tau_1 \succeq \tau_3$ we deduce $\tau_1 = \tau_3$ so that $\sigma = \mathit{gen}(\{x:\tau_1\},\tau_1) = \tau_1 = \tau_3$. Now from $\sigma \succeq \tau_5$ we obtain $\tau_1 = \tau_3 = \sigma = \tau_5$. But $\tau_5 = \tau_4 \to \tau_2 = \mathit{int} \to \tau_2$ so that $\tau = (\mathit{int} \to \tau_2) \to \tau_2$. Hence we obtain

$$\vdash \lambda x.\text{let } i = x \text{ in } i\,1 : (\mathit{int} \to \alpha) \to \alpha$$

Regrouping Theorems 2.1, 2.2, and what we have learnt from the examples above, we have now:

Theorem 2.3. *For an expression e of size n, the construction of a proof of $A \vdash e : \tau$ yields exactly n constraints. The constraints are solved with the same principle that is used to build the proof tree, and their most general solution yields the most general type for τ.*

Remarks:

- Unification is pervasive in this algorithm. It is inherently present in the inference rule formalism, and it is used to solve constraints as well.
- In the process of solving constraints, we find a type for all bound variables. This is why the algorithm is said to perform type inference.

Corollary 2.1. *The system DM′ can be understood and executed as a Prolog program. Given A and e, if some τ is found, and if τ is a finite term, then $\mathit{gen}(A,\tau)$ is the principal type of e.*

Remark: there is only one way in which we can fail to find a type for e in DM′: the constraints do not have any solution that is a finite type (e.g. $e = \lambda x.x\,x$). As soon as we add to DM′ other axioms, such as INT, we can fail for

incompatibility as well (e.g. $e = \lambda x.1\,x$). These two cases of failure are, of course, familiar in first-order unification.

2.5. Executable Specifications

All formal specifications presented in this paper have been tested on a computer. To that end, we code inference rules in a computer formalism called Typol [2]. Transforming the system DM′ into a Typol program is straightforward, but except for the *gen* function and generic instantiation.

2.5.1. Operations on type-schemes

Function *gen* takes a type τ and an environment A and returns a type-scheme. It is built with the help of two auxiliary functions *freevars* and *setminus*. Function *freevars* builds the list of free variables $FV(\tau)$ that occur in a term τ. To build $FV(A)$ we use *freevars* on every assumption in A. Then with function *setminus* we build the list $\{\alpha_1, \alpha_2, \cdots \alpha_n\}$ of generic variables $FV(\tau) \setminus FV(A)$. The type-scheme $gen(A,\tau)$ is then obtained from τ in iterating the construction $bind(\forall, \alpha, \sigma)$ for all generic variables α_i, where *bind* is defined by $bind(\forall, \alpha, \sigma) = \forall\alpha \cdot \sigma$. Functions *freevars*, *setminus* and *bind* are of general interest in semantics and not particularly tied to the context of typechecking. They are provided in a generic way in Typol.

For predicate $\succeq$ we remark that the constraints involving it (generated by TAUT′) always have a type τ on the right hand side. So we can advantageously replace the predicate by a function *inst*, for generic instantiation, that strips all the quantifiers in a type-scheme and consistently replaces bound variables by fresh new free variables. Function *inst* is written in terms of the Typol generic primitive *unbind*, where *unbind* is defined by:

$$unbind(q,\tau',\tau) = \begin{cases} \tau_1[x\backslash\tau'] & (\tau = q_x.\tau_1) \\ \tau & \text{otherwise.} \end{cases}$$

where q denotes a quantifier.

2.5.2. Environment manipulations

To implement the system DM′ we must express in a constructive fashion both the set of assumptions A and the two manipulation operations $A_x \cup \{x:\sigma\}$ and $x:\sigma \in A$. First we implement the set of assumptions A as a list of pairs of the form $x:\sigma$. Then to add a new assumption, we use the *cons* operation on lists, written with an infix dot. But now our list of assumptions may contain several assumptions on the same identifier. Thus the condition $x:\sigma \in A$ of the TAUT′ rule has to be viewed as looking for the first occurrence of $x:\sigma$ in A. This operation is implemented with the following specialized Typol set TYPEOF.

set TYPEOF is

$$\{x:\sigma\} \cdot A \vdash x:\sigma \qquad (1)$$

$$\frac{A \vdash x:\sigma}{\{y:\sigma'\} \cdot A \vdash x:\sigma} \quad (y \neq x) \qquad (2)$$

end TYPEOF

Now we can write in Typol the system DM′ extended with constants as follows:

set TYPE is

$$A \vdash \text{true} : bool \qquad (1)$$

$$A \vdash \text{false} : bool \qquad (2)$$

$$A \vdash \text{number N} : int \qquad (3)$$

$$\frac{A \overset{\text{typeof}}{\vdash} \text{ident X} : \sigma}{A \vdash \text{ident X} : \tau} \quad (\tau = inst(\sigma)) \qquad (4)$$

$$\frac{A \vdash \text{E} : \tau' \to \tau \qquad A \vdash \text{E}' : \tau'}{A \vdash \text{E E}' : \tau} \qquad (5)$$

$$\frac{\{\text{X} : \tau'\} \cdot A \vdash \text{E} : \tau}{A \vdash \lambda\text{X.E} : \tau' \to \tau} \qquad (6)$$

$$\frac{A \vdash \text{E}' : \tau' \qquad \{\text{X} : \sigma\} \cdot A \vdash \text{E} : \tau}{A \vdash \text{let X} = \text{E}' \text{ in E} : \tau} \quad (\sigma = gen(A,\tau')) \qquad (7)$$

end TYPE

2.5.3. Products and Recursion

To complete our specification of Mini-ML, we have to include products, conditional expressions, and recursive definitions. With products it is possible to bind simultaneously several variables in an ML *pattern*. Consider for example the expression $\lambda(x,y).x + y$. Typechecking is not altered by this last feature, except that we must check that patterns are well formed, i.e. that they do not contain the same identifier repeated. An expression such as $\lambda(x,x).x + x$ is not valid. Furthermore we cannot add a global assumption on a pattern, but we must split it into components to associate a type to each identifier in the pattern. So we use the set MKENV to build a local environment, the exclusive union of $\{x:\tau\}$ for each identifier x in the pattern.

set MKENV is

$$\vdash \text{ident X}, \sigma : \{\text{ident X} : \sigma\} \qquad (1)$$

$$\frac{\vdash \text{P}_1, \sigma_1 : A_1 \qquad \vdash \text{P}_2, \sigma_2 : A_2 \qquad A_1 \overset{\cup}{\vdash} A_2 : A_3}{\vdash (\text{P}_1, \text{P}_2), \sigma_1 \times \sigma_2 : A_3} \qquad (2)$$

end MKENV

The Typol set ∪ specifies "exclusive union" on lists of assumptions. To add a new assumption on an identifier x into a list of assumptions A we must consider two cases:

1) the list does not contain any assumption upon x. This corresponds to the union of an empty list with the list $\{x:\sigma\}$ (rule 1).
2) else we iterate on the list (rule 2). But the two identifiers x and y must be different (to avoid multiple declarations).

Next the union of two lists is done one element at a time (rules 3 and 4).

set ∪ is

$$\emptyset \vdash \{x:\sigma\} : \{x:\sigma\} \quad (1)$$

$$\frac{A \vdash \{x:\sigma\} : A'}{\{y:\sigma'\}\cdot A \vdash \{x:\sigma\} : \{y:\sigma'\}\cdot A'} \ (y \neq x) \quad (2)$$

$$A \vdash \emptyset : A \quad (3)$$

$$\frac{A \vdash \{x:\sigma\} : A' \qquad A' \vdash A_1 : A''}{A \vdash \{x:\sigma\}\cdot A_1 : A''} \quad (4)$$

end ∪

In the rules involving binding (ABS and LET′), this local environment is prefixed to the current environment (concatenation is denoted by a semicolon).

Product expressions have product types. *Recursion* is handled by adding the polymorphic fixed-point operator to the initial environment:

$$\{fix : \forall\alpha.(\alpha \to \alpha) \to \alpha\}$$

and the rule 10 of Fig. 4. For the conditional expression *if* the condition must have type *bool* and both true and false part must have the same type (Rule 6).

The static semantics of Mini-ML is described by the Typol program on Fig. 4. To conclude this section, we see that to obtain an executable type checker, most of the work was done at level of inference rules (use DM′ rather than DM), rather than in converting inference rules into an algorithm.

3. Dynamic semantics of Mini-ML

In ML functions can be manipulated as any other object in the language. For example a function may be the parameter of another function: it is possible to define higher-order functions. Thus the domain of semantic values of ML is slightly more complicated than for a less expressive language.

3.1. Semantic values and environment

Values in Mini-ML are either:

- integers: $\mathbb{N}$

program ML_TC is

use ML

$A, A' : ENV$;

$\tau, \tau' : TYPE$;

$\sigma : TYPE\_SCHEME$;

set TYPE is

$$A \vdash \text{number}_{\mathtt{N}} : int \quad (1)$$

$$A \vdash \text{true} : bool \quad (2)$$

$$A \vdash \text{false} : bool \quad (3)$$

$$\frac{\overset{\text{mkenv}}{\vdash} \mathtt{P}, \tau' : A' \qquad A'; A \vdash \mathtt{E} : \tau}{A \vdash \lambda \mathtt{P}.\mathtt{E} : \tau' \to \tau} \quad (4)$$

$$\frac{A \overset{\text{typeof}}{\vdash} \text{ident}_{\mathtt{X}} : \sigma}{A \vdash \text{ident}_{\mathtt{X}} : \tau} \quad (\tau = inst(\sigma)) \quad (5)$$

$$\frac{A \vdash \mathtt{E} : bool \qquad A \vdash \mathtt{E}' : \tau \qquad A \vdash \mathtt{E}'' : \tau}{A \vdash \text{if } \mathtt{E} \text{ then } \mathtt{E}' \text{ else } \mathtt{E}'' : \tau} \quad (6)$$

$$\frac{A \vdash \mathtt{E} : \tau \qquad A \vdash \mathtt{E}' : \tau'}{A \vdash (\mathtt{E}, \mathtt{E}') : \tau \times \tau'} \quad (7)$$

$$\frac{A \vdash \mathtt{E} : \tau' \to \tau \qquad A \vdash \mathtt{E}' : \tau'}{A \vdash \mathtt{E}\ \mathtt{E}' : \tau} \quad (8)$$

$$\frac{A \vdash \mathtt{E}' : \tau' \qquad \overset{\text{mkenv}}{\vdash} \mathtt{P}, \sigma : A' \qquad A'; A \vdash \mathtt{E} : \tau}{A \vdash \text{let } \mathtt{P} = \mathtt{E}' \text{ in } \mathtt{E} : \tau} \quad (9)$$
$$(\sigma = gen(A, \tau'))$$

$$\frac{A \vdash \text{let } \mathtt{P} = fix\, \lambda \mathtt{P}.\mathtt{E}' \text{ in } \mathtt{E} : \tau}{A \vdash \text{letrec } \mathtt{P} = \mathtt{E}' \text{ in } \mathtt{E} : \tau} \quad (10)$$

end TYPE

Figure 4. The static semantics of Mini-ML in TYPOL

- boolean values : *true*, *false*
- closures: $[\lambda \mathtt{P}.\mathtt{E}, \rho]$, where $\mathtt{E}$ is an expression and ρ is an environment. A closure is just a pair of a λ-expression and an environment.
- opaque closures, i.e. closures whose content cannot be inspected. These closures are associated to predefined functions.
- pairs of semantic values: (α, β) (which may in turn be pairs, so that trees of semantic values may be constructed).

Naturally the value of an expression e depends on the values of the identifiers that occur free in it. An *environment* ρ is an ordered list of pairs $P \mapsto \alpha$ where P is pattern

and α a value. Here is an example of environment:

$$x \mapsto 1 \cdot (x,y) \mapsto (true, 5)$$

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. 5. At the top-level, we assume that expressions are evaluated in an initial environment associating a few predefined operators to opaque closures.

program ML_DS is

use ML

$\rho, \rho_1 : ENV$;

$\alpha, \beta : VALUE$;

$$\rho \vdash \text{number}_N : N \tag{1}$$

$$\rho \vdash \text{true} : true \tag{2}$$

$$\rho \vdash \text{false} : false \tag{3}$$

$$\rho \vdash \lambda P.E : [\lambda P.E, \rho] \tag{4}$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}_I \mapsto \alpha}{\rho \vdash \text{ident}_I : \alpha} \tag{5}$$

$$\frac{\rho \vdash E_1 : true \qquad \rho \vdash E_2 : \alpha}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \alpha} \tag{6}$$

$$\frac{\rho \vdash E_1 : false \qquad \rho \vdash E_3 : \alpha}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \alpha} \tag{7}$$

$$\frac{\rho \vdash E_1 : \alpha \qquad \rho \vdash E_2 : \beta}{\rho \vdash (E_1, E_2) : (\alpha, \beta)} \tag{8}$$

$$\frac{\rho \vdash E_1 : \text{opaque}_{OP} \qquad \rho \vdash E_2 : \alpha \qquad \overset{\text{eval}}{\vdash} OP, \alpha : \beta}{\rho \vdash E_1\ E_2 : \beta} \tag{9}$$

$$\frac{\rho \vdash E_1 : [\lambda P.E, \rho_1] \qquad \rho \vdash E_2 : \alpha \qquad P \mapsto \alpha \cdot \rho_1 \vdash E : \beta}{\rho \vdash E_1\ E_2 : \beta} \tag{10}$$

$$\frac{\rho \vdash E_2 : \alpha \qquad P \mapsto \alpha \cdot \rho \vdash E_1 : \beta}{\rho \vdash \text{let } P = E_2 \text{ in } E_1 : \beta} \tag{11}$$

$$\frac{\rho_1 = P \mapsto [E_2, \rho_1] \cdot \rho \qquad \rho_1 \vdash E_1 : \alpha}{\rho \vdash \text{letrec } P = E_2 \text{ in } E_1 : \alpha} \tag{12}$$

end ML_DS

Figure 5. The dynamic semantics of Mini-ML

3.2. Semantic rules

In Figure 5, rules 1 to 3 associate values to integer or boolean literals. 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 expressions. We know from type-checking that the condition has type boolean, and thus its evaluation must return either *true* or *false*. Rule 8 is equally transparent.

The next rules deal with functional values. Rule 9 concerns opaque closures. When the operator of an application evaluates to an opaque closure, we assume that there is some evaluator EVAL that is capable of returning a value β corresponding to the argument α. Here, we could be a little more realistic and have opaque closures contain both the name of an operator *and* the name of an evaluator, to be invoked in this rule. 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 is a pattern or a single variable. From rule 4 and 10 we deduce

$$\frac{\rho \vdash E_2 : \alpha \qquad P \mapsto \alpha \cdot \rho \vdash E_1 : \beta}{\rho \vdash \lambda P.E_1\ E_2 : \beta} \tag{9'}$$

This rule can be added, as an optimization, to the semantics of Mini-ML: it saves building a closure that is to be taken apart immediately thereafter. Comparing it with rule 11, we see that, at evaluation time, $\lambda P.E_1\ E_2 = (\text{let } P = E_2 \text{ in } E_1)$.

The last rule, rule 12, defines in one and the same way the simple recursive functions and the mutually recursive ones such as

$$\begin{aligned} \text{letrec } (f,(g,h)) = (&\lambda x.\cdots f\cdots g\cdots h\cdots, \\ &(\lambda y.\cdots f\cdots g\cdots h\cdots, \\ &\lambda z.\cdots f\cdots g\cdots h\cdots)) \\ \text{in E} & \end{aligned}$$

The environment in which E_1 is evaluated is prefixed with a self-referencing closure.

Notice that since $\rho \vdash E_2 : \alpha$ is a premise of rule 10, we have an ML with call by value.

3.3. Searching the environment

The separate set VAL_OF (see Fig. 6.) 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 prefixed the environment with new associations.

set VAL_OF is

$$\text{ident I} \mapsto \alpha \cdot \rho \vdash \text{ident I} \mapsto \alpha \qquad (1)$$

$$\frac{\rho \vdash \text{ident I} \mapsto \alpha}{\text{ident X} \mapsto \beta \cdot \rho \vdash \text{ident I} \mapsto \alpha} \quad (\text{X} \neq \text{I}) \qquad (2)$$

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

$$\frac{\text{P}_1 \mapsto [\text{E}_1, \rho_1] \cdot \text{P}_2 \mapsto [\text{E}_2, \rho_1] \cdot \rho \vdash \text{ident I} \mapsto \alpha}{(\text{P}_1, \text{P}_2) \mapsto [(\text{E}_1, \text{E}_2), \rho_1] \cdot \rho \vdash \text{ident I} \mapsto \alpha} \qquad (4)$$

end VAL_OF

Figure 6. The ML environment rules

Rules 1 and 2 scan the environment until the first occurrence of an identifier is found, in a left to right scan. Type-checking guarantees that the identifier will be found. 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.

3.4. Equivalent semantics of Mini-ML

The environment ρ_1 for a recursive declaration must satisfy the equation $\rho_1 = P \mapsto [E, \rho_1] \cdot \rho$ where E is the body of the function and ρ is the environment of definition. In the rule 12 of Fig. 5, we have tied a knot in the environment, i.e. we represent recursive environments by graphs. We can write a rule that does not use that artefact. Instead we introduce a new operator $\Leftarrow$ for recursively defined closures:

$$\frac{\text{P} \Leftarrow [\text{E}_2, \rho] \cdot \rho \vdash \text{E}_1 : \alpha}{\rho \vdash \text{letrec P} = \text{E}_2 \text{ in E}_1 : \alpha} \qquad (12')$$

If we use rule 12′ instead of rule 12, environments will contain components that are recursive associations. A new rule is necessary in the set VAL_OF to unfold such associations:

$$\frac{\text{P} \mapsto [\text{E}, \text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho_1] \cdot \rho \vdash \text{ident I} \mapsto \alpha}{\text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho \vdash \text{ident I} \mapsto \alpha}$$

Using rule (12′) and the extra rule above in set VAL_OF on obtains an equivalent semantics. This fact is proved in Appendix A.2.

4. Dynamic semantics of CAM

The Categorical Abstract Machine [3] 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 recursion. Predefined operations (such as addition, subtraction, division, etc.) may be added with the *op* instruction.

4.1. Machine code and Machine state

The abstract syntax of CAM code is given in Fig. 7.

sorts
VALUE, COM, PROGRAM, COMS

subsorts
COM⊃COMS

constructors

'Program'				
program	: COMS	→	PROGRAM	
coms	: COM*	→	COMS	
'Commands'				
quote	: VALUE	→	COM	
op	:	→	COM	
car	:	→	COM	
cdr	:	→	COM	
cons	:	→	COM	
push	:	→	COM	
swap	:	→	COM	
app	:	→	COM	
rplac	:	→	COM	
cur	: COMS	→	COM	
branch	: COMS×COMS	→	COM	
'Values'				
int	:	→	VALUE	
bool	:	→	VALUE	
null_value	:	→	VALUE	

Figure 7. Abstract syntax of CAM code

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 $[c, \rho]$, 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)

4.2. Transition rules

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. 8.

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. For example, we might have

$$init\_stack = (((), [cdr; op +, ()]), [cdr; op -, ()]).$$

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 is the less intuitive one. An *rplac* instruction takes a pair consisting of an environment ρ and a variable V, followed by an environment ρ_1 on the stack. It identifies V and ρ_1 and places the pair (ρ, ρ_1) on the stack. Notice that each occurrence of V 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. 9). In fact this rule can be written in a simpler, but perhaps less transparent, fashion:

$$(\rho, \rho_1) \cdot \rho_1 \cdot s \vdash rplac : (\rho, \rho_1) \cdot s \qquad (17')$$

4.3. Code equivalence

Thanks to the formal definition above, we can now reason about program equivalence for CAM code fragments.

Definition 4.1. *Two fragments of code c_1 and c_2 are equivalent if sequent $s \vdash c_1 : s'$ is provable iff sequent $s \vdash c_2 : s'$ is also provable in CAM_DS. This relation is written $c_1 \equiv c_2$.*

Remark: If c is non-empty and $s \vdash c : s'$ then s and s' are also non empty.

program CAM_DS is

use CAM

$s, s_1, s_2 : STACK;$

$\alpha, \beta : VALUE;$

$\rho, \rho_1 : ENV;$

$$\frac{init\_stack \vdash \text{COMS} : \alpha \cdot s}{\vdash program(\text{COMS}) : \alpha} \qquad (1)$$

$$s \vdash \emptyset : s \qquad (2)$$

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

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

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

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

$$(\alpha, \beta) \cdot s \vdash car : \alpha \cdot s \qquad (7)$$

$$(\alpha, \beta) \cdot s \vdash cdr : \beta \cdot s \qquad (8)$$

$$\alpha \cdot \beta \cdot s \vdash cons : (\beta, \alpha) \cdot s \qquad (9)$$

$$\alpha \cdot s \vdash push : \alpha \cdot \alpha \cdot s \qquad (10)$$

$$\alpha \cdot \beta \cdot s \vdash swap : \beta \cdot \alpha \cdot s \qquad (11)$$

$$\frac{\overset{eval}{\vdash} \text{OP}, \alpha : \beta}{\alpha \cdot s \vdash op\ \text{OP} : \beta \cdot s} \qquad (12)$$

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

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

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

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

$$\frac{\text{V} = \rho_1}{(\rho, \text{V}) \cdot \rho_1 \cdot s \vdash rplac : (\rho, \rho_1) \cdot s} \qquad (17)$$

end CAM_DS

Figure 8. The Categorical Abstract Machine

Definition 4.2. *A fragment of code c preserves the stack iff*

$$\forall \rho \exists \alpha \forall s : \quad \rho \cdot s \vdash c : s' \Longrightarrow s' = \alpha \cdot s$$

Examples: car and *cdr* preserve the stack, but *push* or *swap* do not preserve it.

To illustrate equivalence proofs, we establish the following lemma of interest in the next section:

Lemma 4.1. *If c_1 preserves the stack, then for any code c_2:*

$$push;cur(c_2);swap;c_1;cons;app \equiv push;c_1;cons;c_2$$

Proof: Since c_1 preserves the stack, we assume $\rho\cdot s \vdash c_1 : \alpha\cdot s$ for all s. Then

$$\begin{aligned}
&\rho\cdot s \vdash push;cur(c_2);swap;c_1;cons;app : s' \\
\iff\ &\rho\cdot\rho\cdot s \vdash cur(c_2);swap;c_1;cons;app : s' \\
\iff\ &[c_2,\rho]\cdot\rho\cdot s \vdash swap;c_1;cons;app : s' \\
\iff\ &\rho\cdot[c_2,\rho]\cdot s \vdash c_1;cons;app : s' \\
\iff\ &\alpha\cdot[c_2,\rho]\cdot s \vdash cons;app : s' \\
\iff\ &([c_2,\rho],\alpha)\cdot s \vdash app : s' \\
\iff\ &(\rho,\alpha)\cdot s \vdash c_2 : s' \\
\iff\ &\alpha\cdot\rho\cdot s \vdash cons;c_2 : s' \\
\iff\ &\rho\cdot\rho\cdot s \vdash c_1;cons;c_2 : s' \\
\iff\ &\rho\cdot s \vdash push;c_1;cons;c_2 : s'
\end{aligned}$$

5. Translating Mini-ML to CAM

We are now ready to generate CAM code for Mini-ML. In Fig. 9 is, in the traditional layout used for assembly code, what we produce for the factorial example in Section 1. The translation rules from Mini-ML to CAM† are given in Fig. 10. 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.

† The proof of correctness of this translation appears in [5]

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. 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

$\varphi, \varphi_1 : ENV;$

$$\frac{\rho \mapsto \emptyset \vdash \mathrm{X} : c}{\rho \vdash \mathrm{X} : c} \quad (1)$$

$$\mathrm{ident\,X} \mapsto c\cdot\varphi \vdash \mathrm{ident\,X} : c \quad (2)$$

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

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

end ACCESS

Figure 11. 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. The following lemma is proved in [5].

Lemma 5.1. *If c is the code generated for an ML-expression*

```
push;  quote(ρ)                                          letrec fact =
cons
push
       cur    (push                                      λx. if
                     push;  cdr                         (x
                     swap;  quote(0)                    ,0
                     cons;  op =                        ) =
              branch (quote(1),                         then 1 else
                     push;  cdr                         (x
                     swap;  push;  car;   cdr           ,(fact
                            swap;  push;  cdr           ,(x
                                   swap;  quote(1)      ,1
                                   cons;  op −          )−
                            cons;  app                  )call
                     cons;  op *))                      )*
swap;  rplac                                            in
       push;  cdr                                       (fact
       swap;  quote(4)                                  ,4
       cons;  app                                       )call
```

Figure 9. **Cam code for the factorial function**

program ML_CAM is

use ML

use *CAM*

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

$\rho, \rho_1 : ENV;$

$$\frac{init\_pat \vdash \mathrm{E} \rightarrow c}{\vdash \mathrm{E} \rightarrow program(c)} \quad (1)$$

$$\rho \vdash \text{number N} \rightarrow quote(int\ \mathrm{N}) \quad (2)$$

$$\rho \vdash \text{true} \rightarrow quote(bool\ \text{"true"}) \quad (3)$$

$$\rho \vdash \text{false} \rightarrow quote(bool\ \text{"false"}) \quad (4)$$

$$\frac{\rho \overset{access}{\vdash} \text{ident I} \mapsto c}{\rho \vdash \text{ident I} \rightarrow c} \quad (5)$$

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

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

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

$$\frac{(\rho, \mathrm{P}) \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \text{letrec P} = \mathrm{E}_1 \text{ in } \mathrm{E}_2 \rightarrow \{push; quote(\rho_1); cons; push; c_1; swap; rplac; c_2\}} \quad (9)$$

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

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

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

end ML_CAM

Figure 10. Translation from Mini-ML to CAM

then c preserves the stack, i.e:

$$\rho \overset{ML\_CAM}{\vdash} e \rightarrow c, \quad \alpha \cdot s \overset{CAM\_DS}{\vdash} c : s' \quad \Longrightarrow \quad s' = \beta \cdot s$$

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 ρ_1 in the code. This is a technique for leaving a reference to be resolved at run time. The instruction $quote(\rho_1)$ 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 ρ_1. Instruction *rplac* will tie a knot, freezing the value of ρ_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 *cur* that constructs closures. Rule 11 concerns predefined operators. The predefined operators and their corresponding machine codes appear on Fig. 12. Of course, the list of predefined operators may be extended at will. Here we use for example $Predefined = \{+, -, *\}$.

set TRANS_CONST is

$$\vdash \text{ident "fst"} \rightarrow car \qquad \vdash \text{ident "snd"} \rightarrow cdr \quad (1)$$

$$\vdash \text{ident x} \rightarrow op\ \mathrm{x}\ (\mathrm{x} \in Predefined) \quad (2)$$

end TRANS_CONST;

Figure 12. Predefined operators

5.1. Code Optimisation

Rule 12 is the general case for an application. To illustrate code optimisation, consider again the situation where the operator of an application is an explicit λ-expression. From rules (10) and (12) we can *deduce* the following rule:

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \lambda \mathrm{P}.\mathrm{E}_2\ \mathrm{E}_1 \rightarrow push; c_1; swap; c_2; cons; app} \quad (13)$$

Now with the help of Lemma 4.1 from the previous section, we can obtain the optimised form:

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \lambda \mathrm{P}.\mathrm{E}_2\ \mathrm{E}_1 \rightarrow push; c_1; cons; c_2} \quad (13')$$

6. Conclusion

We have presented completely the semantic aspects of a small but non-trivial functional language: static semantics, dynamic semantics, and compilation to an abstract machine. We believe that the formalism we use can be read and understood by computer scientists who are not specialists in semantics. Furthermore, the formalism has definite technical advantages, and in particular it allows us to test formal definitions on a computer. Finally, many issues, such as mixing interpreted and compiled code or symbolic debugging — usually considered of a strictly pragmatic nature — can be understood in a completely formal manner in this context.

Acknowledgements. We are grateful to J. Incerpi for so much help in editing and typesetting this paper. G. Berry suggested the variant of paragraph 3.4. G. Cousineau made the CAM clear for us. G. Huet provided insights and encouragements.

† The idea of generating code containing free variables occurs already in [15]

REFERENCES

[1] CARDELLI L., "Basic Polymorphic Type-checking", Polymorphism, January 1985.

[2] CLÉMENT D., J. DESPEYROUX, T. DESPEYROUX, L. HASCOET, G.KAHN, "Natural Semantics on the Computer", INRIA Research Report RR 416, INRIA-Sophia-Antipolis, June 1985.

[3] COUSINEAU G., P.L. CURIEN, M. MAUNY, "The Categorical Abstract Machine", in *Functional Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 201, September 1985.

[4] DAMAS L., R. MILNER, "Principal type-schemes for functional programs", *Proceedings of the ACM Conference on Principles of Programming Languages 1982*, pp.207–212.

[5] DESPEYROUX J., "Proof of Translation in Natural Semantics", *Symposium on Logic in Computer Science*, Cambridge, Massachussets, June 1986.

[6] DESPEYROUX T., "Executable Specification of Static Semantics", *Semantics of Data Types*, Lecture Notes in Computer Science, Vol. 173, June 1984.

[7] GORDON M., R. MILNER, C. WADSWORTH, G. COUSINEAU, G.HUET, L. PAULSON, "The ML Handbook, Version 5.1", INRIA, October 1984.

[8] G. HUET "Computation and Deduction", Carnegie Mellon Course Notes, CMU, 1986.

[9] MACCRACKEN N. "The type checking of Programs with implicit type structure", *Semantics of Data Types*, Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[10] MACQUEEN D.B., "Modules for standard ML", *ACM Symposium on LISP and Functional Programming*, 1984, pp.198–207.

[11] MILNER R. "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, n.17, 1978, pp.348–375.

[12] MYCROFT A. "Polymorphic TYpe Schemes and Recursive Definitions", *Sixth International Symposium on Programming*, LNCS 167, 1984.

[13] PLOTKIN G.D., "A Structural Approach to Operational Semantics", DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[14] REYNOLDS J.C., "Three Approaches to Type Structure", *Proceedings TAPSOFT*, Lecture Notes in Computer Science, Vol. 185, March 1985.

[15] WARREN D.H.D., "Logic Programming and Compiler writing", *Software–Practice and Experience*, **10**, 1980, pp.97–125.

Appendix A.1. Equivalence between DM and DM'

Theorem 2.1. *The system DM' is equivalent to DM in the following sense:*

$$A \overset{DM'}{\vdash} e : \tau \Longrightarrow A \overset{DM}{\vdash} e : \tau$$

$$\forall A, e\ \exists \tau \quad A \overset{DM}{\vdash} e : \sigma \Longrightarrow A \overset{DM'}{\vdash} e : \tau\ \&\ gen(A,\tau) \succeq \sigma.$$

provided σ does not contain useless quantifier (i.e. σ is of the form $\forall \alpha_1 \cdots \alpha_n.\tau_1$ with α_i occuring in τ_1).

To prove the equivalence we use induction on the length of the proof and the lemma 2.1. We need induction on the length of the proof instead of structural induction, because of the GEN and INST rules of DM. First we consider the simpler case of the equivalence, i.e. the soundness of the system DM'.

($\Longleftarrow$) Soundness of the system DM'. For each proof tree in DM' which satisfy the condition on *gen*, we must exhibit a proof tree in DM. The method is the same for every proof tree, so we illustrate it only for proof trees ending with the TAUT' and the LET' rules.

Rule TAUT'. We have:

$$A \overset{DM'}{\vdash} x : \tau \qquad (x : \sigma \in A,\ \sigma \succeq \tau).$$

So we can build the following proof tree in DM:

$$\frac{A \vdash x : \sigma \qquad (x : \sigma \in A)}{A \vdash x : \tau \quad [INST]}$$

Rule LET'. We have:

$$\frac{A \overset{DM'}{\vdash} e' : \tau' \quad A_x \cup \{x : \sigma\} \overset{DM'}{\vdash} e : \tau}{A \overset{DM'}{\vdash} \text{let } x = e' \text{ in } e : \tau} \quad (\sigma = gen(A,\tau'))$$

So we have in DM:

$$\frac{\dfrac{A \vdash e' : \tau' \quad (ind)}{A \vdash e' : \sigma\ [GEN]} \qquad A_x \cup \{x : \sigma\} \vdash e : \tau \quad (ind)}{A \vdash \text{let } x = e' \text{ in } e : \tau \quad [LET]}$$

($\Longrightarrow$)Completeness of the system DM'. For each proof tree in DM we must exhibit a proof tree in DM' that satisfies the condition on *gen*. We do not give the proof in full details, but we only indicate its general outline. We will use the following property of *gen*:

Lemma 2.2. If $A_x \cup \{x : \tau'\} \vdash e : \tau_1$ with $gen(A_x \cup \{x : \tau'\}, \tau_1) \succeq \tau$ then $gen(A,\tau_1) \succeq \tau$.

This lemma expresses a kind of monotonicity of the *gen* operation. If $A_x \cup \{x : \tau'\} \vdash e : \tau_1$ then the generalisation

of the type expression τ_1 over A cannot be more restrictive than its generalisation over $A_x \cup \{x : \tau'\}$.

Rule TAUT. We have:

$$A \vdash x : \sigma \qquad (x : \sigma \in A)$$

So we have the following proof tree in DM′:

$$A \vdash' x : \tau \qquad (x : \sigma \in A, \sigma \succeq \tau)$$

where τ is the generic instance of σ obtained as follows:

$$\tau = \begin{cases} S\tau' & \text{if } \sigma = \forall\alpha_1 \cdots \alpha_n.\tau', \\ \sigma & \text{otherwise.} \end{cases}$$

with S of the form $[\alpha_i \leftarrow \beta_i]$ $(1 \leq i \leq n)$, where the β_i are new type variables. By definition of *gen*, we have $gen(A, \tau) \succeq \sigma$.

Rule INST. We have:

$$\frac{A \vdash e : \sigma}{A \vdash e : \sigma'} \qquad (\sigma \succeq \sigma')$$

So, using the transitivity of $\succeq$, we can build the following proof tree in DM′:

$$\frac{\dfrac{A \vdash' e : \tau \qquad gen(A, \tau) \succeq \sigma \ (ind)}{A \vdash' e : \tau}}{gen(A, \tau) \succeq \sigma' \ (trans)}$$

Rule GEN. We have:

$$\frac{A \vdash e : \sigma}{A \vdash e : \forall\alpha.\sigma} \qquad (\alpha \notin FV(A))$$

By induction on the numerator, $A \overset{\text{DM}'}{\vdash} e : \tau$ with $gen(A, \tau) \succeq \sigma$. Now, with the assumption on σ, the condition $\alpha \notin FV(A)$ implies that $gen(A, \tau) \succeq \forall\alpha.\sigma$.

Rule APP. We have:

$$\frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash e\, e' : \tau}$$

So we can build the following proof tree in DM′:

$$\dfrac{\begin{array}{ll} A \vdash' e : \tau_1' \to \tau_1 \ (ind) & \dfrac{A \vdash' e' : \tau_2' \qquad gen(A, \tau_2') \succeq \tau' \ (ind)}{A \vdash' e' : \tau_1'} \\ gen(A, \tau_1' \to \tau_1) \succeq \tau' \to \tau & \end{array}}{\dfrac{}{gen(A, \tau_1') \succeq \tau' \ (unif)}} \\ A \vdash' e\, e' : \tau_1 \quad [APP]$$

with $gen(A, \tau_1) \succeq \tau$.

Rule ABS. We have:

$$\frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

So we can build in DM′:

$$\frac{A_x \cup \{x : \tau'\} \vdash' e : \tau_1 \qquad gen(A_x \cup \{x : \tau'\}, \tau_1) \succeq \tau \ (ind)}{A \vdash' \lambda x.e : \tau' \to \tau_1 \quad [ABS]}$$

with $gen(A, \tau' \to \tau_1) \succeq \tau' \to \tau$ (*lemma2.2.*).

Rule LET. We have:

$$\frac{A \vdash e' : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e : \tau}{A \vdash \text{let } x = e' \text{ in } e : \tau}$$

So we can build the following proof tree:

$$\dfrac{\begin{array}{ll} A \vdash' e' : \tau' \ (ind) & \dfrac{\dfrac{A_x \cup \{x : \sigma\} \vdash e : \tau}{A_x \cup \{x : gen(A, \tau')\} \vdash e : \tau \ (2.1.)}}{A_x \cup \{x : gen(A, \tau')\} \vdash' e : \tau_1 \ (ind)} \\ gen(A, \tau') \succeq \sigma & \end{array}}{\dfrac{}{gen(A_x \cup \{x : gen(A, \tau')\}, \tau_1) \succeq \tau}} \\ A \vdash' \text{let } x = e' \text{ in } e : \tau_1 \ [LET'] \\ gen(A, \tau_1) \succeq \tau \ (2.2)$$

□ Theorem 2.1.

Appendix A.2.
Equivalence between graph and unfolding

Theorem 3.1. $\vdash e : t(\alpha) \iff \vdash' e : \alpha$

Where $\vdash$ stands for $\overset{\text{ml\_ds}}{\vdash}$ with graphs and $\vdash'$ stands for $\overset{\text{ml\_ds}'}{\vdash}$ with the unrolling operator $\Leftarrow$. The function t goes from environments and values with the $\Leftarrow$ operator to environments and values with graphs.

First we define a function t which goes from environments and values with the $\Leftarrow$ operator to environments and values with graphs.

Definition 3.1. *For environments we define:*

- $t(\emptyset) = \emptyset$
- $t(P \mapsto \alpha \cdot \rho) = P \mapsto t(\alpha) \cdot t(\rho)$
- $t(P \Leftarrow [E, \rho_1] \cdot \rho) = P \mapsto [E, \rho_2] \cdot t(\rho)$ *with* $\rho_2 = P \mapsto [E, \rho_2] \cdot t(\rho_1)$

For values, we define:

- $t(N) = N$
- $t(B) = B$
- $t(\text{ident OP}) = \text{ident OP}$
- $t((\alpha, \beta)) = (t(\alpha), \beta))$
- $t([E, \rho]) = [E, t(\rho)]$

Now we shall prove that:

$$t(\rho) \vdash e : t(\alpha) \iff \rho \vdash' e : \alpha,$$

by induction on the length of the proof. We do not give it in full detail, but we consider only the most significant rules. Note that we have also $\rho \vdash e : \alpha \iff t'(\rho) \vdash' e : t'(\alpha)$ where the function t' goes from environments and values with graphs to environments and values with the $\Leftarrow$ operator.

($\Longrightarrow$) The rules are the same in both systems and the induction is quite obvious, except for the "letrec" rule.

Rule ml_ds.12: $e = \mathit{letrec}\ P = E_2\ \mathit{in}\ E_1$

$$\frac{\rho_1 = \text{P} \mapsto [\text{E}_2, \rho_1] \cdot t(\rho) \qquad \rho_1 \vdash \text{E}_1 : t(\alpha)}{t(\rho) \vdash \text{letrec P} = \text{E}_2 \text{ in E}_1 : t(\alpha)}$$

by definition of t, $\rho_1 = t(\text{P} \Leftarrow [\text{E}_2, \rho] \cdot \rho)$, so by induction:

$$\frac{\text{P} \Leftarrow [\text{E}_2, \rho] \cdot \rho \vdash' \text{E}_1 : \alpha}{\rho \vdash' \text{letrec P} = \text{E}_2 \text{ in E}_1 : \alpha}$$

Set Val_of: The proof is trivial since all inference rules of $\overset{\text{val\_of}}{\vdash}$ are also in $\overset{\text{val\_of}'}{\vdash}$

($\Longleftarrow$) Proofs are the same, except for the VAL_OF set because of the new unfolding rule.

$$\frac{\text{P} \mapsto [\text{E}, \text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho_1] \cdot \rho \vdash' \text{ident I} \mapsto \alpha}{\text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho \vdash' \text{ident I} \mapsto \alpha}$$

So we can build in $\vdash$:

$$\frac{\text{P} \mapsto [\text{E}, \text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho_1] \cdot \rho \vdash' \text{ident I} \mapsto \alpha}{\text{P} \mapsto [\text{E}, t(\text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho_1)] \cdot t(\rho) \vdash \text{ident I} \mapsto t(\alpha)\ (ind, t)}$$
$$t(\text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho) \vdash \text{ident I} \mapsto t(\alpha)$$

because, by definition of t:

$$t(\text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho_1) = \text{P} \Leftarrow [\text{E}, \rho_2] \cdot t(\rho_1) = \rho_2$$

and finally:

$$\text{P} \mapsto [\text{E}, \rho_2] \cdot t(\rho) = t(\text{P} \Leftarrow [\text{E}, \rho_1] \cdot \rho)$$

□ Theorem 3.1.