Natural Semantics

G. Kahn

INRIA, Sophia–Antipolis

06565 Valbonne CEDEX, FRANCE†

In recent years, many researchers have begun to present semantic specifications in a style that has been strongly advocated by Plotkin in [19]. The purpose of this paper is to introduce in an intuitive manner the essential ideas of the method that we call now Natural Semantics, together with its connections to ideas in logic and computing. Natural Semantics is of interest *per se* and also because it is used as a semantics specification formalism for the interactive computer system CENTAUR that we are currently building at INRIA.

1. Introduction

It is interesting and illuminating to present several aspects of the semantics of programming languages in a unified manner: static semantics, dynamic semantics, and translation. It has been shown in earlier work [10]that it is possible to use denotational semantics to give a satisfactory account of all these semantic aspects. What is more, several researchers, following [16]process the resulting formal descriptions to obtain actual type-checkers, interpreters, and translators. One may wonder why it should be necessary to investigate yet another semantics specification formalism.

Several difficulties come up in a purely denotational definition.

- Coding up static semantics as semantics in a domain of types turns out to be a subterfuge. First, since type-checking aims at characterizing legal programs, the domain of type values must be artificially extended to include one (or several) values for "wrong type". Second, in a language with overloading, an identifier may *a priori* have several possible types. Should the overloading resolution algorithm be part of the formal specification? As discussed in [9], writing static semantics in this way is more akin to *programming* in a functional language than to writing a formal specification.
- A similar lack of abstraction and expressive uneasiness is felt when specifying translations. For example, one needs to have an extra parameter just for the purpose of generating new symbols in a denotational way and there is no elegant way to do backpatching. A consequence is that specifying translators is again considered a programming activity and proving properties of translations is made considerably harder.
- Denotational semantics is of course much better suited to express dynamic semantics. Two techniques of denotational semantics, known as currying and continuations, are notationally difficult for most language designers, but following the ideas of Mosses [16]this difficulty could be overcome. In the case of parallelism and non-determinism, denotational semantics becomes substantially more difficult. Somehow an operational definition is easier to understand and more convincing. The semantic definitions of many new proposals for parallel languages involve axiomatizing atomic transitions, or transactions, and

† This research is partially supported under ESPRIT, Project 348.

are easily expressed in Natural Semantics. But this is not to say that Natural Semantics should be identified with operational semantics.

The general idea of this sort of semantic definition is to provide axioms and inference rules that characterize the various semantic predicates to be defined on an expression E. To paraphrase Prawitz [18], "the inferences are broken down into atomic steps in such a way that each step involves only one language construct".

For example in static semantics, one wants to state that expression E has type τ in the environment ρ. Hence one axiomatizes

$$\rho \vdash \mathrm{E} : \tau$$

where the environment ρ is a collection of assumptions on the types of the variables of E. To specify a translation from language L_1 to language L_2, one gives rules of the form

$$\rho \vdash \mathrm{E}_1 \longrightarrow \mathrm{E}_2$$

where expression E_1 is in the source language, expression E_2 is in the target language and ρ records how identifiers should be translated. In dynamic semantics, there is a much greater variance in style, depending on the properties of the language to describe. In the simplest languages, it is sufficient to express that the evaluation of the program fragment E in state s_1 yields a new state s_2. This predicate is written

$$s_1 \vdash \mathrm{E} \Rightarrow s_2$$

where state s_1 records the values of the identifiers involved in E.

A semantic definition is a list of axioms and inference rules that define one of the predicates above. In other words, a semantic definition is identified with a logic, and reasoning with the language amounts to proving theorems within that logic. Computing (e.g. type-checking, intepreting) is seen as a way to solve certain equations. For example, given state s_1 and program E, is there a state s_2 s.t. $s_1 \vdash \mathrm{E} \Rightarrow s_2$ holds? Or given an initial type-environment ρ_0, is it possible for expression E to be assigned a type τ such that $\rho_0 \vdash \mathrm{E} : \tau$ holds?

This formulation suggests several remarks.

- Other kinds of equations could be of interest. For example, given E and τ, does there exist some ρ such that $\rho \vdash \mathrm{E} : \tau$? This is a type-inference problem.
- Since the presentation is inherently relational, rather than functional, non-deterministic computation will be the general case. Similarly, overloading in type-checking will arise naturally.
- Since several logics will be defined on the same object language (e.g one assigning types to programs and another assigning values to them), it will be interesting to examine relationships between these logics.

Now there are many ways of presenting a logic. In our experiments, on the left of the turnstile symbol $\vdash$ we always have a collection of assumptions on variables, not arbitrary formulae. This is most reminiscent of Natural Deduction. Rules for dealing with block structure, type-checking, or evaluating applications are very close to certain rules in Natural Deduction, except for the fact that our collections of assumptions or environments are not necessarily sets. Hence the name Natural Semantics was coined, rather than the more restrictive and less informative "Structural Operational Semantics" of Plotkin.

1.1. *Comparison with syntax*

The way we look at Natural Semantics is proof-theoretic: we think of axioms and rules of inferences as a way of generating new facts from existing facts. Inference rules allow the construction of new proof-trees from existing proof-trees. In this regard, our view is very close to the traditional use of context-free grammars in computer science.

A grammar is presented as a collection of grammar rules. The rules define legal parse trees, and as a consequence legal sentences. Analogously, axioms and inference rules serve in defining legal proof-trees, and hence the facts that may be derived from axioms using inference rules. A grammar may be used either as a generator or as a recognizer. There exist general algorithms to find a parse tree for a given sentence. Likewise, a semantic description may be thought of in two ways: as a generating facts or as a description of possible computations. Corresponding to the general recognizers of context-free grammars, we have the general interpreters of semantic descriptions. Grammars may be ambiguous, and this often considered a nuisance. But we *want* to model nondeterministic computations, and in a logical system it is generally the case that there are several proofs of the same fact.

There is however a major technical difference between grammars and logical systems. In a grammar, the non-terminals stand for sets of words. In an inference rule, or rule scheme, the variables stand for individuals (terms), and all occurrences of the same variable in the rule should be substituted with the same term.

2. The formalism

2.1. *Rules*

A semantic definition is an *unordered* collection of rules. A rule has basically two parts, a numerator and a denominator. *Variables* may occur both in the numerator and the denominator of a rule. These variables allow a rule to be instantiated. Usually, typographical conventions are used to indicate that the variables in a rule must have a certain type.

The numerator of a rule is again an *unordered* collection of formulae, the *premises* of the rule. Intuitively, if all premises hold, then the denominator, a single formula, holds. More formally, from proof-trees yielding the premises, we can obtain a new proof-tree yielding the denominator, or conclusion, of the rule.

2.2. *Sequents and conditions*

Formulae are divided in two kinds: *sequents* and *conditions*. The conclusion of a rule is necessarily a sequent. On the numerator, sequents are distinguished from conditions, that are placed slightly to the right of the inference rule. Conditions convey in general a restriction on the applicability of the rule: a variable may not occur free somewhere, a value must satisfy some predicate, some relation must hold between two variables. As boolean predicates, conditions are built with the help of logical connectives from atomic conditions. One may wish to axiomatize atomic conditions, for example in a separate set of rules.

A sequent has two parts, an *antecedent* (on the left) and a *consequent* (on the right), and we use the turnstile symbol $\vdash$ to separate these parts. The consequent is a predicate. Predicates come in several forms, indicated by various infix symbols. These infix symbols carry no reserved meaning, they just help us in memorizing what is being defined. The first argument of the consequent is called the *subject* of the sequent. Naturally, the subject of a rule is the subject of its conclusion.

A rule that contains no sequent on the numerator is called an *axiom*. Thus an axiom may be constrained by a condition.

2.3. *Judgements*

In a single semantic definition, sequents may have several forms depending on the syntactic nature of their subject. For example, in a typical Algol-like language, there are declarations, statements and expressions. The static semantics will contain sequents of the form

$$\rho_1 \vdash \text{DECL} : \rho_2$$

for the elaboration of declarations, of the form

$$\rho \vdash \text{STM}$$

to assert that statements are well typed, and also of the form

$$\rho \vdash \text{EXP} : \tau$$

to state that expression EXP has type τ. The various forms of sequents participating in the same semantic definition are called *judgements*. One reason for the elegance of the formalism is that several judgements are used without being given explicit names. In programming, overloading is used to the same end. Note that in our context like in programming, abuse of overloading leads to obscurity.

2.4. *Rule sets*

Some structure must be introduced in a collection of rules, if only to separate different semantic concerns. For example, in static semantics, one wishes to distinguish structural rules of consistency from the management of scope and the properties of type values. To this end, rules may be grouped into sets, with a given *name*. Sets of rules collect together rules or, recursively, rule sets. When one wishes to refer to a sequent that is axiomatized in a set of rules other than the textually enclosing one, the name of the set is indicated as a superscript of the sequent's turnstile.

2.5. *Abstract syntax, Use clauses*

Semantics tells us facts about the constructs of a language. These constructs taken together form the abstract syntax of the language, technically an order-sorted algebra. Intuitively, each construct has arguments and results belonging to syntactic categories, and some syntactic categories may be included in others. We indicate that language L is concerned with a definition by the declaration use L. In a translation, two languages are involved and we have to import two algebras. Other objects, such as environments or stores are often elements of algebras, and we will naturally modularize our definition by importing these algebras as well. When analyzing mechanically semantic rules, we will identify the various abstract syntax constructors. Ambiguity may arise if two algebras use the same name for an operator. Most of the time the context will be sufficient to resolve the ambiguity, but we may have to specify what operator we really mean.

Abstract syntax terms may occur in rules. They should of course be valid terms w.r.t their abstract syntax. Every such term is typed with a syntactic category (such as L.*expression*, or L.*statement*, or L.*declaration*). A language L includes all of its syntactic categories, and it is possible for two languages to share a given syntactic category. For example PASCAL and MODULA can share the category *expression*.

With an abstract syntax, we also import conventions on how to write abstract syntax trees in a linear fashion. Except where the notation is too ambiguous, we use systematically this readable way of denoting terms. For example, we will write

$$\text{while COND do STM}$$

rather than use the general notation for terms

$$\text{while(COND, STM)}$$

but the reader should be aware that the subject of a rule is never a string but a tree.

2.6. *Assigning types to rules*

The scope of variables is limited to the rule where they appear. Nevertheless, it is necessary to follow certain naming conventions to make a definition readable. For example, we want to assert that variables called ρ, possibly with indices or diacritical signs, are environments. For this we allow variable declarations. The scope of such declarations is the set of rules where they appear. It is not necessary to declare in this way all variables, because often their type may be inferred. In particular, it is practically never useful to declare variables that stand for abstract syntax fragments because these variables occur in the subject of rules. There, a language constructor dictates their type. Declarations and abstract syntax definitions serve then in typing sequents. It might also be wise to declare judgements, rather than merely infer how many judgement forms are involved in a definition.

2.7. *Typographical conventions*

In accordance with standard mathematical practice, it is convenient to associate different fonts to different types of variables. But it would be extremely painful to indicate these font changes as we enter the semantic definitions in a computer, with a keyboard that has a limited character set. Instead, we associate font information to types, and our type-checker is set-up to generate a text that is fully decorated with font changes. This text is then processed by TEX and either printed or examined on a high resolution display. It is clear that a sober use of fonts enhances the readability of semantic definitions.

2.8. *General strategy for execution*

As mentioned earlier, we want to use a computer to solve various kinds of equations on sequents. Typically, our unknown will be type values, states or generated code. But environments, or program fragments may also be unknown. To turn semantic definitions into executable code, there are probably many approaches. One is to compile rules into Prolog code, taking advantage of the similarity of Prolog variables and variables in inference rules. Roughly speaking, the conclusion of a rule maps to a clause head, and the premises to the clause body. Distinct judgements map to distinct Prolog predicates. Conditions, although written to the right of rules, are placed ahead of the rule body.

An equation is turned into a Prolog goal. Since pure Prolog attacks goals in a left to right manner, proofs of premises will also be attempted from left to right. This is not always reasonable, so that we need to use a version of Prolog that may postpone attacking goals until certain variables are instantiated. Conditions should be evaluated as soon as possible, to avoid building useless proof-trees. In our experiments, we have used Mu-Prolog [17]with success to that end.

2.9. *Actions*

It is useful to attach actions to rules, in a manner that is reminiscent of the way actions are associated to grammar-rules in YACC. Actions are triggered only after an inference rule is considered applicable. An action needs to access the rule's variable bindings, but it cannot under any circumstance interfere with the deduction process. Typically, actions are used to trace inference rules, to emit messages, to perform a variety of side effects. It is important to understand that searching for a proof may involve backtracking, so that if an inference has been used in a computation at some point, it does not necessarily participate in the final proof.

In terms of style, actions should be used with parsimony. For example, when specifying a translation, it is mandatory to *axiomatize it* rather than have actions generate output code. On the other hand, in the context of type-checking, it is more appropriate to have actions filter error messages, rather than introduce strange type values to handle various erroneous situations.

A significant use of actions is in debugging inference rules. We want to follow what inference rules are applied, but also *where* they are applied. In other words, when a rule is used we want to know where the subject of the rule is, with respect to the subject of the initial equation. When executing a dynamic semantics specification, we follow execution very precisely in this way.

To solve this problem, we imagine that each variable that stands for an abstract syntax tree is in fact a pair made of tree and a tree address. Our rule compiler then keeps track of the tree-offsets of the variables introduced in the subject of the rule, relative to the tree address of the subject. Within actions, the user can refer to the tree-address of the rule's subject via a standard variable.

3. A small functional language

As an example, we are going to write semantic specifications in Natural Semantics for a very small functional language. This language, called Mini-ML, is a simple typed λ-calculus with constants, products, conditionals, and recursive function definitions. Of ML [11], it retains call-by-value. The language is strongly typed, but there are no type declarations, types are inferred from the context. It is possible to define functions that work uniformly on arguments of many types: one construct introduces ML-polymorphism.

The dynamic semantics of ML is fairly simple to describe. The only difficulty resides in handling elegantly mutually recursive definitions. To illustrate compilation, we use as target code the the Categorical Abstract Machine (CAM) of Cousineau and Curien [4]. It is interesting to see how convenient Natural Semantics is to specify such a translation and some of its properties. ML typechecking is the object of numerous discussions in the literature, e.g. [6]. Using an *inference system* to describe typing goes back at least to Curry [5]. Reynolds [20]is a remarkable presentation in this spirit.

We begin with an intuitive presentation of the language.

3.1. *Sample programs*

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

$$\begin{array}{l}\text{letrec } \ \mathit{fact}{=}\ \lambda x.\,\text{if } x = 0\,\text{then } \ 1\,\text{else } \ x * \mathit{fact}(x-1)\\ \text{in } \mathit{fact}\ 4\end{array}$$

Next, we define and use the higher order function *twice*:

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

The language has block structure, so that the following expression evaluates to 6:

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

Here we have both simultaneous definitions and block structure:

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

and this last example involves simultaneous recursive definitions:

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

3.2. *Abstract Syntax of Mini-ML*

An abstract syntax is an order-sorted algebra. It is given by a set of sorts, a description of their inclusion relations, and the list of all language constructors, together with their syntactic types. The abstract syntax of Mini-ML is given[1] on Fig. 1. It defines a 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$\supset$ nullpat, ident   pat$\supset$ nullpat, ident

constructors

Patterns

pairpat	:	pat$\times$pat	$\rightarrow$	pat
nullpat	:		$\rightarrow$	nullpat

Expressions

number	:		$\rightarrow$	exp
false	:		$\rightarrow$	exp
true	:		$\rightarrow$	exp
ident	:		$\rightarrow$	ident
lambda	:	pat$\times$exp	$\rightarrow$	exp
if	:	exp$\times$exp$\times$exp	$\rightarrow$	exp
mlpair	:	exp$\times$exp	$\rightarrow$	exp
apply	:	exp$\times$exp	$\rightarrow$	exp
let	:	pat$\times$exp$\times$exp	$\rightarrow$	exp
letrec	:	pat$\times$exp$\times$exp	$\rightarrow$	exp

Figure 1. Abstract Syntax of mini-ML

4. Dynamic Semantics

In Mini-ML, the evaluation of a sub-expression always yields a value, so that we have to axiomatize the single judgement

$$\rho \vdash \text{E} \Rightarrow \alpha$$

where E is a Mini-ML expression, ρ is an environment and α is the result of the evaluation of E in ρ. Functions can be manipulated as any other object in the language. For example a function may be passed as parameter to another function, or returned as the value of

[1] An abstract syntax may also be presented as a set of inference rules. Sort inclusions give then rise to *inheritance* rules.

an expression. Thus the domain of semantic values is slightly more complicated than for a traditional Algol-like language.

4.1. *Semantic values, environments*

Values in Mini-ML are either:

- integer values in $\mathbb{N}$
- boolean values *true* and *false*, in italics to distinguish them from the literals true and false.
- closures of the form $[\![\lambda \text{P.E}, \rho]\!]$, where E is an expression and ρ is an environment. A closure is just a pair of a λ-expression denoting a function and an environment.
- *opaque* closures, i.e. closures whose contents cannot be inspected. These closures are associated to predefined functions.
- pairs of semantic values of the form (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. These values are recorded in the environment. A Mini-ML *environment* ρ is an ordered list of pairs $P \mapsto \alpha$ where P is pattern and α a value. The symbol $\cdot$ separates the elements of this list. Here is an example of environment:

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

The environment is intuitively scanned *from right to left* so that in this example, x is associated to 1, y is associated to 5, and no other identifier has a value associated to it. The typical equation we want to solve with the formal system in Fig. 2. is

$$\rho_0 \vdash \text{E} \Rightarrow \alpha$$

where α is the unknown and ρ_0 is an initial environment. The initial environment associates opaque closures to a few predefined operators, such as $+$, $-$, etc.

4.2. *Semantic rules*

The semantic rules of Mini-ML are shown on Figure 2. Axioms 1 to 3 state that integer and boolean literals are in normal form, i.e. yield immediately a semantic value.

Axiom 4 asserts that an expression of the form λP.E is also in normal form. The value obtained is a closure, pairing this expression with the environment. This is in sharp contrast with λ-calculus, where E would be further reduced. Axiom 5 simply says that the value associated to an identifier is to be looked up in the environment. But since the environment is somewhat complex, we choose to axiomatize this process with auxiliary rules, the set VAL_OF. These rules are described later. Evaluation of conditional expressions is specified in the next two rules 6 and 7. Note that these rules have the same conclusion but that they are mutually exclusive. They are also exhaustive provided, as we shall assume, we evaluate only expressions that are type correct. The rules show that evaluation of E_1, E_2 and E_3 takes place in the same environment, and has no side effect. Hence it could be done in parallel.

In Natural Deduction terminology, rules 2 and 3 are *introduction rules* for the boolean values *true* and *false*. Rules 6 and 7 are *elimination* rules, they tell how boolean values may be consumed. Putting together rule 2 and 6 (resp. 3 and 7)we obtain two unexciting derived inference rules

$$\frac{\rho \vdash \text{E}_2 \Rightarrow \alpha}{\rho \vdash \text{if true then E}_2 \text{ else E}_3 \Rightarrow \alpha} \qquad (6')$$

$$\rho \vdash \text{number}\ \textsc{n} \Rightarrow \textsc{n} \tag{1}$$

$$\rho \vdash \text{true} \Rightarrow \mathit{true} \tag{2}$$

$$\rho \vdash \text{false} \Rightarrow \mathit{false} \tag{3}$$

$$\rho \vdash \lambda\textsc{p}.\textsc{e} \Rightarrow [\![\lambda\textsc{p}.\textsc{e}, \rho]\!] \tag{4}$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}\ \textsc{i} \mapsto \alpha}{\rho \vdash \text{ident}\ \textsc{i} \Rightarrow \alpha} \tag{5}$$

$$\frac{\rho \vdash \textsc{e}_1 \Rightarrow \mathit{true} \qquad \rho \vdash \textsc{e}_2 \Rightarrow \alpha}{\rho \vdash \text{if}\ \textsc{e}_1\ \text{then}\ \textsc{e}_2\ \text{else}\ \textsc{e}_3 \Rightarrow \alpha} \tag{6}$$

$$\frac{\rho \vdash \textsc{e}_1 \Rightarrow \mathit{false} \qquad \rho \vdash \textsc{e}_3 \Rightarrow \alpha}{\rho \vdash \text{if}\ \textsc{e}_1\ \text{then}\ \textsc{e}_2\ \text{else}\ \textsc{e}_3 \Rightarrow \alpha} \tag{7}$$

$$\frac{\rho \vdash \textsc{e}_1 \Rightarrow \alpha \qquad \rho \vdash \textsc{e}_2 \Rightarrow \beta}{\rho \vdash (\textsc{e}_1, \textsc{e}_2) \Rightarrow (\alpha, \beta)} \tag{8}$$

$$\frac{\rho \vdash \textsc{e}_1 \Rightarrow [\![\lambda\textsc{p}.\textsc{e}, \rho_1]\!] \qquad \rho \vdash \textsc{e}_2 \Rightarrow \alpha \qquad \rho_1 \cdot \textsc{p} \mapsto \alpha \vdash \textsc{e} \Rightarrow \beta}{\rho \vdash \textsc{e}_1\ \textsc{e}_2 \Rightarrow \beta} \tag{9}$$

$$\frac{\rho \vdash \textsc{e}_2 \Rightarrow \alpha \qquad \rho \cdot \textsc{p} \mapsto \alpha \vdash \textsc{e}_1 \Rightarrow \beta}{\rho \vdash \text{let}\ \textsc{p} = \textsc{e}_2\ \text{in}\ \textsc{e}_1 \Rightarrow \beta} \tag{10}$$

$$\frac{\rho \cdot \textsc{p} \mapsto \alpha \vdash \textsc{e}_2 \Rightarrow \alpha \qquad \rho \cdot \textsc{p} \mapsto \alpha \vdash \textsc{e}_1 \Rightarrow \beta}{\rho \vdash \text{letrec}\ \textsc{p} = \textsc{e}_2\ \text{in}\ \textsc{e}_1 \Rightarrow \beta} \tag{11}$$

Figure 2. The dynamic semantics of mini-ML

$$\frac{\rho \vdash \textsc{e}_3 \Rightarrow \alpha}{\rho \vdash \text{if false then}\ \textsc{e}_2\ \text{else}\ \textsc{e}_3 \Rightarrow \alpha} \tag{7$'$}$$

Rule 8 is the introduction rule for pairs of values. Both components of the pair must be evaluated. The lack of rules for value-pair elimination (left projection, right projection) might indicate a weakness in the design of Mini-ML. But a rule in the set VAL_OF serves that purpose.

The next rule 9 deals with function 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. The closure's body is evaluated in the closure's environment, to which the parameter association $\textsc{p} \mapsto \alpha$ has been added. Since $\textsc{e}_2$ is evaluated, Mini-ML uses call-by-value. The rule is valid whether $\textsc{p}$ is a pattern or a single variable. Note that the rule is departing from denotational semantics in that $\textsc{e}$ is *not* a subexpression of $\textsc{e}_1$ or $\textsc{e}_2$. It is a closure elimination rule, whereas rule 4 was the closure introduction rule. From rule 4 and 9 we deduce an interesting rule:

$$\frac{\rho \vdash \textsc{e}_2 \Rightarrow \alpha \qquad \rho \cdot \textsc{p} \mapsto \alpha \vdash \textsc{e}_1 \Rightarrow \beta}{\rho \vdash \lambda\textsc{p}.\textsc{e}_1\ \textsc{e}_2 \Rightarrow \beta} \tag{9$'$}$$

This rule can be added, as an optimization, to the semantics of mini-ML. In the case where the operator of an application is syntactically a λ-term, it is not necessary to build a closure that is to be taken apart immediately thereafter. If we compare this new rule with rule 10, we see that, at evaluation time,

$$\lambda\textsc{p}.\textsc{e}_1\ \textsc{e}_2 \equiv (\text{let}\ \textsc{p} = \textsc{e}_2\ \text{in}\ \textsc{e}_1).$$

Rule 10 is typical of Natural Deduction, as we can see in a logical presentation where ρ is implicit and emphasis is placed on discharging hypotheses:

$$\frac{\mathrm{E}_2 \Rightarrow \alpha \qquad \begin{array}{c}[\mathrm{P} \mapsto \alpha] \\ \mathrm{E}_1 \Rightarrow \beta\end{array}}{\text{let } \mathrm{P} = \mathrm{E}_2 \text{ in } \mathrm{E}_1 \Rightarrow \beta} \tag{10}$$

The last rule, rule 11, defines in one and the same way the simple recursive functions and the mutually recursive ones such as

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

The rule is very similar to rule 10, except that expression E_2 is evaluated in the same environment as E_1, rather than in ρ. We should keep in mind that in Mini-ML only functional objects may be defined recursively, so that α is either a closure, or a tree of closures. In both cases, it will contain a reference to itself and may be understood as a regular infinite tree.

For clarity, we had omitted the rule dealing with predefined function symbols:

$$\frac{\rho \vdash \mathrm{E}_1 \Rightarrow \text{opaque } \mathrm{OP} \qquad \rho \vdash \mathrm{E}_2 \Rightarrow \alpha \qquad \overset{\text{eval}}{\vdash} \;\mathrm{OP}, \alpha \Rightarrow \beta}{\rho \vdash \mathrm{E}_1 \; \mathrm{E}_2 \Rightarrow \beta} \tag{12}$$

This rule is an elimination rule for opaque closures. When the operator of an application evaluates to an opaque closure, we assume that there is an 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. There is no introduction rule for opaque closures: we assume that they come solely from the initial environment.

4.3. *Searching the environment*

The separate set VAL_OF (see Fig. 3.) gives rules to associate values to identifiers, given some environment. There are two problems to solve. First, we must traverse the environment from right to left to take into account block structure. This is achieved by the rules 1 and 2. Rule 1 could be called a *tautology* rule, and rule 2 is a *thinning* rule. The second problem is that the environment contains associations of the form $P \mapsto \alpha$ where P is a pattern, i.e. a tree of identifiers. From typechecking we know that P is bound to a value of the same shape, and rule 3 traverses P.

set VAL_OF is

$$\rho \cdot \text{ident } \mathrm{I} \mapsto \alpha \vdash \text{ident } \mathrm{I} \mapsto \alpha \tag{1}$$

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

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

end VAL_OF

Figure 3. The ML environment rules

Rule 3 is a sort of pair elimination rule. It is an astonishingly simple method to keep rules 10 and 11 valid when P is a pattern.

4.4. *Executing the definition*

To solve the equation in the unknown α

$$\rho_0 \vdash \text{E} \Rightarrow \alpha$$

we search for the last step of the proof of this fact. The structure of E forces this step in general. Rules 6 and 7 on the one hand, rules 9 and 12 on the other can lead to backtracking. This situation is very general and analyzed in [2]. It is interesting to remark that the derived inference rules $6'$ $7'$ and $9'$ can be added to the definition, since they can only infer valid facts. But they should systematically be preferred to 6 and 7, or to 9 and 12 because they are "faster". Intuitively, they should be preferred because their subject is more *specific*.

Non-termination in Mini-ML may occur because of rule 11. In the process of solving the initial equation, it is possible to grow the candidate proof tree endlessly from the bottom up. Then the equation will have no solution. Remark also that the Mini-ML we have described uses call-by-value. It is not difficult to write rules for a lazy Mini-ML, along the lines of [15]. The idea is to create new introduction rules for a different kind of closure called *suspensions*, and new elimination rules for these suspensions.

The fact that we used a functional language to illustrate dynamic semantics should not leave the impression that imperative languages cannot be described. Several experiments with Algol-like languages are reported in [13].

5. Translation

Translation from one language to another is heavily guided by the structure of the source language. Hence it is clear that our formalism is well suited for specifying translations.

Recently, Cousineau and Curien have proposed a very ingenious abstract machine for the compilation of ML [4]. The complete semantics of the machine is described first. Then we specify the translation from Mini-ML to CAM.

5.1. *Specifying the Categorical Abstract Machine (CAM)*

The Categorical Abstract Machine 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 *rec* is used to implement recursion. Predefined operations (such as addition, subtraction, division, etc.) may be added with the *op* instruction.

5.1.1. *Machine code and Machine state*

The abstract syntax of CAM code is given in Fig. 4.

The state of the CAM machine is a stack, whose top element may be viewed as a register. If s is a stack and α is a value, then pushing α onto s yields $s \cdot \alpha$. Several kinds of values may be pushed on the stack. Atomic values

- integers in $\mathbb{N}$,
- truth values *true* and *false*,

but also closures and environments since the machine is designed for higher-order functional languages sorts

  value, com, program, coms

subsorts

  com$\supset$coms

constructors

Programs

program	:	coms	$\rightarrow$	program
coms	:	com*	$\rightarrow$	coms

Commands

quote	:	value	$\rightarrow$	com
car	:		$\rightarrow$	com
cdr	:		$\rightarrow$	com
cons	:		$\rightarrow$	com
push	:		$\rightarrow$	com
swap	:		$\rightarrow$	com
op	:		$\rightarrow$	com
branch	:	coms$\times$coms	$\rightarrow$	com
cur	:	coms	$\rightarrow$	com
app	:		$\rightarrow$	com
rec	:	coms	$\rightarrow$	com

Values

int	:		$\rightarrow$	value
bool	:		$\rightarrow$	value

Figure 4. Abstract syntax of CAM code

- closures of the form $[\![\mathrm{C}, \rho]\!]_{cam}$, where C is a fragment of CAM code and ρ is a value that is meant to denote an environment,
- pairs of semantic values, and hence recursively trees of such values.

The pair constructor is used in particular to build environments. A special value () denotes the empty environment.

5.1.2. *Transition rules*

The rules describing the transitions of the CAM machine appear in Fig. 5. Two judgements are used. Only rule 1 involves the judgement

$$s \vdash \mathrm{C} \Rightarrow \alpha$$

meaning that program C, started in state s returns α as result. All other rules involve only sequents of the form

$$s \vdash \mathrm{C} \Rightarrow s'$$

where C is CAM-code and s and s' are states of the CAM machine. The sequent may be read as *executing code* C *when the machine is in state* s *takes it to state* s'.

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 *init_stack* contains a single element, the initial environment. This environment includes the closures corresponding to the predefined operators. For example, we might have

$$init\_stack = \big(\big(\big((), [\![cdr; op+,\ ()]\!]_{cam}\big), [\![cdr; op-,\ ()]\!]_{cam}\big), [\![cdr; op*,\ ()]\!]_{cam}\big).$$

$$\frac{\mathit{init\_stack} \vdash \text{COMS} \Rightarrow s \cdot \alpha}{\vdash \mathit{program}(\text{COMS}) \Rightarrow \alpha} \tag{1}$$

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

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

$$s \cdot \alpha \vdash \mathit{quote}(\text{V}) \Rightarrow s \cdot \text{V} \tag{4}$$

$$s \cdot (\alpha, \beta) \vdash \mathit{car} \Rightarrow s \cdot \alpha \tag{5}$$

$$s \cdot (\alpha, \beta) \vdash \mathit{cdr} \Rightarrow s \cdot \beta \tag{6}$$

$$s \cdot \alpha \cdot \beta \vdash \mathit{cons} \Rightarrow s \cdot (\alpha, \beta) \tag{7}$$

$$s \cdot \alpha \vdash \mathit{push} \Rightarrow s \cdot \alpha \cdot \alpha \tag{8}$$

$$s \cdot \alpha \cdot \beta \vdash \mathit{swap} \Rightarrow s \cdot \beta \cdot \alpha \tag{9}$$

$$\frac{\overset{\text{eval}}{\vdash} \ \text{OP}, \alpha \Rightarrow \beta}{s \cdot \alpha \vdash \mathit{op}\, \text{OP} \Rightarrow s \cdot \beta} \tag{10}$$

$$\frac{s \vdash \text{C}_1 \Rightarrow s_1}{s \cdot \mathit{true} \vdash \mathit{branch}(\text{C}_1, \text{C}_2) \Rightarrow s_1} \tag{11}$$

$$\frac{s \vdash \text{C}_2 \Rightarrow s_1}{s \cdot \mathit{false} \vdash \mathit{branch}(\text{C}_1, \text{C}_2) \Rightarrow s_1} \tag{12}$$

$$s \cdot \rho \vdash \mathit{cur}(\text{C}) \Rightarrow s \cdot [\![\text{C}, \rho]\!]_{cam} \tag{13}$$

$$\frac{s \cdot (\rho, \alpha) \vdash \text{C} \Rightarrow s_1}{s \cdot ([\![\text{C}, \rho]\!]_{cam}, \alpha) \vdash \mathit{app} \Rightarrow s_1} \tag{14}$$

$$\frac{s \cdot (\rho, \rho_1) \vdash \text{C} \Rightarrow s \cdot \rho_1}{s \cdot \rho \vdash \mathit{rec}(\text{C}) \Rightarrow s \cdot \rho_1} \tag{15}$$

Figure 5. The definition of the Categorical Abstract Machine

Rules 2 and 3 specify sequential execution for a sequence of commands. Axioms 4 to 7 show that elementary instructions overwrite the top of the stack with their result. Instructions *push* and *swap*, described in rules 8 and 9 are useful to save the top of the stack. Rule 10 switches to an external evaluator EVAL for predefined operators. The external evaluator must be aware that it receives a single tree-stuctured argument.

Rule 11 and 12 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 13: *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 14 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. The *rec* instruction is used to build the self referencing environment ρ_1. Such an environment is necessary for the evaluation of recursive definitions. Notice the remarkable simplicity of rule 15.

5.1.3. *Remarks*

- Given some program P written in CAM code, to run it is to find an α such that $\vdash P \Rightarrow \alpha$. The general execution strategy works well and there are never any choices to build a proof. Rule 3 is the only one with two premises, and they must be treated from left to right.
- The Natural Deduction point of view does not seem to bring much insight here, but it is convenient to specify an abstract machine in the same formalism as a semantic definition. The equivalence of certain sequences of code can be proved easily and this is useful for optimisation [3].

5.2. *Generating CAM code for Mini-ML*

We are now ready to generate CAM code for mini-ML. Here is what we will produce for the factorial example in paragraph 3.1, laid out like an assembly code listing:

```
push;                                                letrec fact =
rec    (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 ∗)))                  )∗
cons                                                 in
       push;  cdr                                    (fact
       swap;  quote(4)                               ,4
       cons;  app                                    )call
```

The rules for translating Mini-ML to CAM[1] are given in Fig. 6. In these rules, except for rule 1, all sequents have the form:

$$\rho \vdash \text{E} \rightarrow c$$

where ρ is an environment, E is an expression in Mini-ML, and c is its translation into CAM-code. In words, the sequent reads: *in environment ρ, expression* E *is compiled into code c.* The notion of environment used in this translation is exactly the notion of a pattern in Mini-ML, i.e. a binary tree with identifiers at the leaves. The environment is used to decide what code to generate for variables.

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 6, 10, and 11). 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.

Rules 2, 3, and 4 generate code for literal values. Rule 5 generates an access path for an identifier. In rule 6, the body of a λ-term is compiled and then wrapped in a *cur* instruction.

[1] The proof of correctness of this translation is given in [7]

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

$$\rho \vdash \text{number}\ \mathrm{N} \rightarrow \mathit{quote}(\mathrm{N}) \tag{2}$$

$$\rho \vdash \text{true} \rightarrow \mathit{quote}(\mathit{true}) \tag{3}$$

$$\rho \vdash \text{false} \rightarrow \mathit{quote}(\mathit{false}) \tag{4}$$

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

$$\frac{(\rho, \mathrm{P}) \vdash \mathrm{E} \rightarrow c}{\rho \vdash \lambda \mathrm{P}.\mathrm{E} \rightarrow \mathit{cur}(c)} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad \rho \vdash \mathrm{E}_2 \rightarrow c_2 \qquad \rho \vdash \mathrm{E}_3 \rightarrow c_3}{\rho \vdash \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3 \rightarrow \mathit{push}; c_1; \mathit{branch}(c_2, c_3)} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad \rho \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) \rightarrow \mathit{push}; c_1; \mathit{swap}; c_2; \mathit{cons}} \tag{8}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad \rho \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 \rightarrow \mathit{push}; c_1; \mathit{swap}; c_2; \mathit{cons}; \mathit{app}} \tag{9}$$

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \text{let}\ \mathrm{P} = \mathrm{E}_1\ \text{in}\ \mathrm{E}_2 \rightarrow \mathit{push}; c_1; \mathit{cons}; c_2} \tag{10}$$

$$\frac{(\rho, \mathrm{P}) \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \text{letrec}\ \mathrm{P} = \mathrm{E}_1\ \text{in}\ \mathrm{E}_2 \rightarrow \mathit{push}; \mathit{rec}(c_1); \mathit{cons}; c_2} \tag{11}$$

Figure 6. Translation from mini-ML to CAM

To understand the next rules, the following inductive assertion is useful: *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 whenever necessary. If a temporary result is obtained, the *swap* instruction will push it on the stack, while bringing back on top the environment necessary for further computation. Now rule 7 and 8 become quite clear. Rule 9 builds a pair of values, the operator and the operand, and then generates an *app*. Rules 10 and 11 are very similar, just the way rules 10 and 11 were in dynamic semantics. A simple parameter binding is added to the environment in rule 10, while rule 11 adds a recursive binding. As indicated before, at execution time the *rec* instruction will build a self-referencing environment.

From the identity that is valid at run-time

$$\lambda \mathrm{P}.\mathrm{E}_2\ \mathrm{E}_1 \equiv (\text{let}\ \mathrm{P} = \mathrm{E}_1\ \text{in}\ \mathrm{E}_2)$$

we deduce that the following inference rule should be valid

$$\frac{\rho \vdash \mathrm{E}_1 \rightarrow c_1 \qquad (\rho, \mathrm{P}) \vdash \mathrm{E}_2 \rightarrow c_2}{\rho \vdash \lambda \mathrm{P}.\mathrm{E}_2\ \mathrm{E}_1 \rightarrow \mathit{push}; c_1; \mathit{cons}; c_2} \tag{12}$$

Indeed, it is shown in [3]that rule 12 may be *deduced* from rules 6 and 9, once a few basic lemmas on CAM code have been established. Rule 12 is a code optimization rule. It should be added to the other rules and preferred when applicable.

To generate code for identifiers, the set ACCESS in Figure 7. is used. Here again, we assume that the program that is to be translated type-checks, and in paticular that it contains no undeclared variables. The rules in this set are similar to the rules in the set VAL_OF, except that while searching for an identifier, one constructs simultaneously its access path.

set ACCESS is

$$\frac{\rho \mapsto \emptyset \vdash \text{ident}\ \textsc{x} : c}{\rho \vdash \text{ident}\ \textsc{x} : c} \qquad (0)$$

$$\rho \cdot \text{ident}\ \textsc{x} \mapsto c \vdash \text{ident}\ \textsc{x} : c \qquad (1)$$

$$\frac{\rho \vdash \text{ident}\ \textsc{x} : c}{\rho \cdot \text{ident}\ \textsc{y} \mapsto c' \vdash \text{ident}\ \textsc{x} : c} \quad (\textsc{y} \neq \textsc{x}) \qquad (2)$$

$$\frac{\rho \cdot \rho_1 \mapsto c; car \cdot \rho_2 \mapsto c; cdr \vdash \text{ident}\ \textsc{x} : c'}{\rho \cdot (\rho_1, \rho_2) \mapsto c \vdash \text{ident}\ \textsc{x} : c'} \qquad (3)$$

end ACCESS

Figure 7. Generating access paths for identifiers

6. Type-checking Mini-ML

6.1. *General framework*

The semantic specifications of the previous sections rely on the hypothesis that they are applied to well-typed program fragments. From that angle, being well typed is a constraint on abstract syntax trees – what is sometimes called *context-sensitive* syntax. But if an expression E has type τ, it also means that the possible values that E may take during execution are characterized by τ. When axiomatizing

$$\rho \vdash \textsc{e} : \tau$$

where ρ is a type environment, both viewpoints are present. If the sequent has no proof, then E does not type-check. If it can be proved, then E has type τ. The relationship between the type system and the dynamic semantics must be established, by showing a variant of the Subject Reduction Theorem of [5].

Before introducing an inference system that assigns types in Mini-ML, the type language has to be explained. 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 $int \rightarrow int$. In the same way the identity function $\lambda x.x$ for integers has type $int \rightarrow int$, but for booleans it has type $bool \rightarrow bool$. 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 $\forall\alpha.\alpha \rightarrow \alpha$.

6.2. *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 $\tau \rightarrow \tau'$, 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 $FV(\sigma)$ for the set of free variables of a type expression σ. We now define two relations between type expressions that contain type variables.

Definition. *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. *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.

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

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 $\tau = \tau'$.

6.3. *The typing rules*

The rules for typing Mini-ML programs are given in Figure 8. Two judgements are used. The main judgement has the form

$$\rho \vdash \text{E} : \tau$$

where the environment ρ is a list of assumptions of the form $x : \sigma$. The environment is to be scanned from right to left. Concatenation of two environments ρ and ρ' is noted $\rho + \rho'$. An auxiliary judgement is necessary to handle the declaration of patterns correctly:

$$\vdash \text{P}, \tau : \rho.$$

This formula means that declaring P with type τ creates the type environment ρ. For example we have:

$$\vdash (x,(y,z)),(\tau_1 \times (\tau_2 \times \tau_3)) : \ x : \tau_1 \cdot y : \tau_2 \cdot z : \tau_3$$

$$\rho \vdash \text{number } \mathrm{N} : \mathit{int} \tag{1}$$

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

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

$$\frac{\vdash \mathrm{P}, \tau' : \rho' \qquad \rho + \rho' \vdash \mathrm{E} : \tau}{\rho \vdash \lambda \mathrm{P}.\mathrm{E} : \tau' \rightarrow \tau} \tag{4}$$

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{ident } \mathrm{X} : \sigma}{\rho \vdash \text{ident } \mathrm{X} : \tau} \quad (\tau = \mathit{inst}(\sigma)) \tag{5}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathit{bool} \qquad \rho \vdash \mathrm{E}_2 : \tau \qquad \rho \vdash \mathrm{E}_3 : \tau}{\rho \vdash \text{if } \mathrm{E}_1 \text{ then } \mathrm{E}_2 \text{ else } \mathrm{E}_3 : \tau} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \tau_1 \qquad \rho \vdash \mathrm{E}_2 : \tau_2}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) : \tau_1 \times \tau_2} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \tau' \rightarrow \tau \qquad \rho \vdash \mathrm{E}_2 : \tau'}{\rho \vdash \mathrm{E}_1 \ \mathrm{E}_2 : \tau} \tag{8}$$

$$\frac{\vdash \mathrm{P}, \tau_2 : \rho' \qquad \rho \vdash \mathrm{E}_2 : \tau_2 \qquad \rho + \rho'' \vdash \mathrm{E}_1 : \tau_1}{\rho \vdash \text{let } \mathrm{P} = \mathrm{E}_2 \text{ in } \mathrm{E}_1 : \tau_1} \quad (\rho'' = \mathit{gen}(\rho, \rho')) \tag{9}$$

$$\frac{\vdash \mathrm{P}, \tau_2 : \rho' \qquad \rho + \rho' \vdash \mathrm{E}_2 : \tau_2 \qquad \rho + \rho' \vdash \mathrm{E}_1 : \tau_1}{\rho \vdash \text{letrec } \mathrm{P} = \mathrm{E}_2 \text{ in } \mathrm{E}_1 : \tau_1} \tag{10}$$

$$\vdash \text{ident } \mathrm{X}, \tau : \ \text{ident } \mathrm{X} : \tau \tag{11}$$

$$\frac{\vdash \mathrm{P}_1, \tau_1 : \rho_1 \qquad \vdash \mathrm{P}_2, \tau_2 : \rho_2}{\vdash (\mathrm{P}_1, \mathrm{P}_2), \tau_1 \times \tau_2 : \ \rho_1 + \rho_2} \quad (\rho_1 \cap \rho_2 = \emptyset) \tag{12}$$

Figure 8. Rules for type assignment in Mini-ML

The first three axioms are the introduction rules for basic types. Rule 4 is the introduction rule for $\rightarrow$. The environment may associate a generic type σ to an identifier, but in ML an occurrence of an identifier has a type τ. So the condition in rule 5 specifies to create a generic instance of τ of the type scheme σ.

Rule 6 is the elimination rule for the basic type *bool*. It is interesting to note that there are two occurrences of τ in the premises, so that the rule unifies the types of the two expressions E_2 and E_3. Rule 7 introduces product types. Rule 8 is the elimination rule for $\rightarrow$. It performs unification on τ' in the same way as rule 6.

From rules 4 and 8 we deduce

$$\frac{\vdash \mathrm{P}, \tau_2 : \rho' \qquad \rho \vdash \mathrm{E}_2 : \tau_2 \qquad \rho + \rho' \vdash \mathrm{E}_1 : \tau_1}{\rho \vdash \lambda \mathrm{P}.\mathrm{E}_1 \ \mathrm{E}_2 : \tau_1} \tag{9'}$$

which differs from rule 9. So from the point of view of types,

$$\lambda \mathrm{P}.\mathrm{E}_1 \ \mathrm{E}_2 \not\equiv (\text{let } \mathrm{P} = \mathrm{E}_2 \text{ in } \mathrm{E}_1).$$

Rule 9 is the source of polymorphism in Mini-ML: the environment ρ'' is obtained from ρ' by generalization with respect to ρ. It is important to understand that generalizing ρ' creates several independent type schemes, while generalizing τ_2 would give a single type scheme. Rule 10 defines recursively τ_2 in terms of itself and it exhibits the usual similarity with rule $9'$. There is no point in trying to generalize ρ' with respect to $\rho + \rho'$.

The last two rules concern declarations. Rule 11 builds a singleton environment, rule 12 concatenates environments, while checking that they do not intersect. This is to exclude patterns with repeated occurrences of the same variable.

To search the type environment, the familiar rules of tautology and thinning are shown again in Figure 9.

set TYPE_OF is

$$\rho \cdot x : \sigma \vdash x : \sigma \qquad (1)$$

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

end TYPE_OF

Figure 9. Searching the type environment

6.4. *Remarks*

- Strictly speaking, the definition above is too generous. It computes recursive types for certain Mini-ML expressions that are not supposed to have any type, such as $\lambda x.x\,x$.
- In Mini-ML, the Subject Construction Theorem is valid [5]. Consider the proof tree for

$$\rho \vdash \text{E} : \tau.$$

  If we label each inference step with the constructor of its subject, we see that this proof tree is isomorphic to E. This result does not seem specific of Mini-ML.
- Many interesting ideas in type systems such as inheritance, overloading, coercions etc... are not present in Mini-ML. It is our experience that such ideas can be described fairly simply with Natural Semantics.
- The type-checker obtained from a definition in Natural Semantics is correct but not readily usable: it fails to type-check as soon as there is one type error. It is shown in [12]how to transform mechanically such a definition into a friendly type-checker that emits error messages and carries on. Further mechanical transformations yield in certain conditions an incremental type-checker.

7. Conclusion

In this presentation of Natural Semantics, we hope to have shown a simple and mathematically tractable method of writing semantic descriptions. At the present time, we investigate how to incorporate these ideas in a most faithful way in the construction of a complete interactive environment [13]. Much work is still needed of course, but the results of the first few years indicate that it is possible to create an elegant and reasonably efficient system.

ACKNOWLEDGMENTS

This paper owes much to earlier contributions of D. Clément and J. Despeyroux. Th. Despeyroux has built a large part of the system that permits testing semantic definitions. G. Berry, G. Cousineau and G. Huet have given sound advice in various discussions.

REFERENCES

[1] Cardelli L., “Basic Polymorphic Type-checking”, Polymorphism, January 1985.

[2] Clément D. “The Natural Dynamic Semantics of Mini-Standard ML”, to appear in *Proceedings CFLP*, Pisa, March 1987.

[3] Clément D., J. Despeyroux, Th. Despeyroux, G. Kahn, “A simple applicative language: Mini-ML”, *Proceedings of the ACM Conference on Lisp and Functional Programming 1986.*

[4] 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.

[5] Curry H.B., R. Feys, Combinatory Logic, Volume I, North-Holland Publishing Company, 1958.

[6] Damas L., R. Milner, “Principal type-schemes for functional programs”, *Proceedings of the ACM Conference on Principles of Programming Languages 1982*, pp.207–212.

[7] Despeyroux J., “Proof of Translation in Natural Semantics”, *Proceedings of the First ACM Conference on Logic in Computer Science, LICS 1986.*

[8] Despeyroux T., “Executable Specification of Static Semantics”, *Semantics of Data Types*, Lecture Notes in Computer Science, Vol. 173, June 1984.

[9] Despeyroux T., “Spécifications sémantiques dans le système MENTOR”, Thèse, Université Paris XI, 1983.

[10] Donzeau-Gouge V., “Utilisation de la sémantique dénotationnelle pour la description d’interprétations non-standard: application à la validation et à l’optimisation des programmes”, *Proceedings of the 3rd International Symposium on Programming*, Dunod, Paris, 1978.

[11] Gordon M., R. Milner, C. Wadsworth, G. Cousineau, G.Huet, L. Paulson, “The ML Handbook, Version 5.1”, INRIA, October 1984.

[12] Hascoet L., “Transformations automatiques de spécifications sémantiques. Application: un vérificateur de types incrémental” Thèse , To appear, Université de Nice, 1987.

[13] Heering J., J. Sidi, A. Verhoog (eds), “Generation of interactive programming environments - GIPE intermediate report”, CWI Report CS-R8620, Amsterdam, May 1986.

[14] MacQueen D.B., “Modules for standard ML”, *ACM Symposium on LISP and Functional Programming*, 1984, pp.198–207.

[15] Mauny M. “Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML”, Thèse, Université Paris 7, 1985.

[16] Mosses P., “SIS: a compiler generator system using denotational semantics”, DAIMI, University of Aarhus, August 1979.

[17] Naish L., Negation and Control in Prolog, Lecture Notes in Computer Science, Vol. 238, 1986.

[18] Prawitz D., “Ideas and results in proof theory”, *Proceedings of the Second Scandinavian Logic Symposium*, 1971, North-Holland.

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

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

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

Table of Contents