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 [10], "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 $\tau$ in the environment $\rho$. Hence one axiomatizes

$$\rho \vdash e : \tau$$

where the environment $\rho$ is a collection of assumptions on the types of the variables of $e$. To specify a translation from language $L_1$ to language $L_2$, on gives rules of the form

$$\rho \vdash e_1 \rightarrow e_2$$

where $\rho$ records assumptions on identifiers, expression $e_1$ is in the source language and $e_2$ is in the target language. 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 expression $e$ in state $s_1$ yields a new state $s_2$. This predicate is written

$$s_1 \vdash e \Rightarrow s_2$$

where $s_1$ 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 is proving theorems within that logic. Converting (e.g., type-checking, interpreting) is seen as a way to solve equations. For example, given state $s_1$ and program $p$, is there a state $s_2$ s.t. $s_1 \vdash e \Rightarrow s_2$ holds? Or given an initial type-environment $\rho_0$, is it possible for expression $e$ to be assigned a type $\tau$ such that $\rho_0 \vdash e : \tau$ holds?

This formulation suggests several remarks.

- Other kinds of equations could be of interest. For example, given $\tau$ and $\rho$, does there exist some $e$ such that $\rho \vdash e : \tau$? 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 formulse. 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.

1.2. Comparison with syntax

The way we look at Natural Semantics is proof-theoretic: we think of axioms and rules of inference 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 schema, 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 → 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

\[ ρ_1 \vdash \text{DECL} : ρ_2 \]

for the elaboration of declarations, of the form

\[ ρ \vdash \text{STM} \]

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

\[ ρ \vdash \text{EXP} : τ \]

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 algebraic systems as well. When analysing 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 a rule 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 p, 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 variable information to types 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 with grammar-rules in YACC. Actions are triggered only after an inference rule is considered applicable. An action needs to be attached to 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 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 type-checking 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 all, of course is how to write the factorial function:

\[
\text{letrec } fact = \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x \times \text{fact}(x - 1) \text{ in } \text{fact} 4
\]

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

\[
\text{let } \text{suc}\!c = \lambda x. x + 1 \\
\text{in } \text{let } \text{twice } = \lambda f. \lambda x. f(f(x)) \\
\text{in } ((\text{twice suc}) 0)
\]

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

\[
\text{let } i = 5 \\
\text{in } \text{let } i = i + 1 \text{ in } i
\]

Here we have both simultaneous definitions and block structure:

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

and this last example involves simultaneous recursive definitions:

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

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 constructs, together with their syntactic types. The abstract syntax of Mini-ML is given in 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 \( \text{pair}\!p \) builds products of expressions, while the \( \text{pair}\!p \) constructor serves in building patterns of identifiers. The \( \text{nullp}\!c \) constructor is used for the unit object \( () \), which is both a pattern and an expression.

<table>
<thead>
<tr>
<th>sorts</th>
<th>EXP, IDENT, PAT, NULLPAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsorts</td>
<td>EXP ⊅ NULLPAT, IDENT</td>
</tr>
<tr>
<td>PAT ⊅ NULLPAT, IDENT</td>
<td></td>
</tr>
<tr>
<td>constructors</td>
<td></td>
</tr>
<tr>
<td>Patterns</td>
<td>pairpat : PAT × EXP → PAT</td>
</tr>
<tr>
<td>nullpat : → NULLPAT</td>
<td></td>
</tr>
<tr>
<td>Expressions</td>
<td>number : → EXP</td>
</tr>
<tr>
<td>false : → EXP</td>
<td></td>
</tr>
<tr>
<td>true : → EXP</td>
<td></td>
</tr>
<tr>
<td>ident : → IDENT</td>
<td></td>
</tr>
<tr>
<td>lambda : EXP → EXP</td>
<td></td>
</tr>
<tr>
<td>if : EXP × EXP → EXP</td>
<td></td>
</tr>
<tr>
<td>mpair : EXP × EXP → EXP</td>
<td></td>
</tr>
<tr>
<td>apply : EXP × EXP → EXP</td>
<td></td>
</tr>
<tr>
<td>let : EXP × EXP → EXP</td>
<td></td>
</tr>
<tr>
<td>letrec : EXP × EXP → EXP</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ \rho \vdash e \Rightarrow a \]

where \( e \) is a Mini-ML expression, \( \rho \) is an environment and \( a \) is the result of the evaluation of \( e \) in \( \rho \). 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 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 \( \text{true} \) and \( \text{false} \), in italics to distinguish them from the literals \( \text{true} \) and \( \text{false} \).

1 An abstract syntax may also be presented as a set of inference rules. Sort inclusions give then rise to inheritance rules.
in the environment. But since the environment is somewhat complex, we choose to axiomatize this process with auxiliary rules, the set Valop. 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 $e_1, e_2, e_3$ 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

$$\rho \vdash e_2 \Rightarrow \alpha$$

$$\rho \vdash \text{if true then } e_2 \text{ else } e_3 \Rightarrow \alpha$$

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 Valop 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 $\tau \Rightarrow \alpha$ has been added. Since $e_3$ is evaluated, Mini-ML uses call-by-value. The rule is valid whether $\tau$ is a pattern or a single variable. Note that the rule is departing from denotational semantics in that $e_3$ is not a subexpression of $e_1$ or $e_2$. It is a closure elimination rule, whereas rule 4 was the closure introduction rule. From rule 4 and 9 we deduce an interesting rule:

$$\rho \vdash E_2 \Rightarrow \alpha \quad P \cdot P \Rightarrow \alpha \vdash E_1 \Rightarrow \beta$$

$$\rho \vdash \lambda P . E_3, E_1 \Rightarrow \beta$$

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 $\lambda$-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,

$$\lambda P . E_3, E_1 \equiv (\text{let } P = E_3 \text{ in } E_1).$$

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

$$\rho \vdash E_3 \Rightarrow \alpha \quad E_3 \Rightarrow \alpha$$

$$\text{let } r = P \text{ in } E_3 \Rightarrow \beta$$

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

$$\text{letrec } f (x, y) = \langle (x, \cdots, f \cdots g \cdots h \cdots),$$

$$\lambda x . f \cdots g \cdots h \cdots \rangle$$

$$\text{in } E$$

The rule is very similar to rule 10, except that expression $E_3$ is evaluated in the same environment as $E_1$, rather than in $\rho$. We should keep in mind that in Mini-ML only functional objects may be defined recursively, so that $\alpha$ 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:

$$\rho \vdash E_3 \Rightarrow \text{Opaque } OP \quad \rho \vdash E_3 \Rightarrow \alpha \quad \rho \vdash \text{OP }, \alpha \Rightarrow \beta$$

$$\rho \vdash E_3, E_2 \Rightarrow \beta$$

Figure 2. The dynamic semantics of mini-ML

Axiom 4 asserts that an expression of the form $\lambda P . E$ is also in normal form. The value obtained is a closure, pairing this expression with the environment. This is in sharp contrast with $\lambda$-calculus, where $e$ would be further reduced. Axiom 5 simply says that the value associated to an identifier is to be looked up
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 \( \text{eval} \) that is capable of returning a value \( \beta \) corresponding to the argument \( \alpha \). 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 \( \text{val.of} \) (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 \( \text{tautology} \) rule, and rule 2 is a \( \text{thinning} \) rule. The second problem is that the environment contains associations of the form \( P' \rightarrow \alpha \) 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 \( P' \).

\[
\begin{align*}
\text{set \( \text{val.of} \) is} \\
\rho \cdot \text{ident} \xi & \rightarrow \alpha \vdash \text{ident} \xi \rightarrow \alpha \\
\rho \vdash \text{ident} \xi & \rightarrow \alpha \\
\rho \cdot \text{ident} x & \rightarrow \beta \vdash \text{ident} x \rightarrow \alpha \\
(\& \neq 1) \\
\rho \cdot \pi_i & \rightarrow \alpha \cdot \pi_j \rightarrow \beta \vdash \text{ident} i \rightarrow \gamma \\
\rho \cdot (\pi_i, \pi_j) & \rightarrow (\alpha, \beta) \vdash \text{ident} i \rightarrow \gamma \\
\end{align*}
\]

\[
\text{end \( \text{val.of} \)}
\]

\[\text{Figure 3. The ML environment rules}\]

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 \( \alpha \)

\[
\rho_\alpha \vdash \xi \rightarrow \alpha
\]

we search for the last step of the proof of this fact. The structure of \( \alpha \) 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 \( 6' \) 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 \( \text{suspending} \) 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 [19].

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 \( \text{car} \) and \( \text{cdr} \) serve in accessing data in the stack and the special instruction \( \text{rec} \) is used to implement recursion. Predefined operations (such as add, subtract, 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.

\[
\begin{align*}
\text{sorts} & \quad \text{VALUE, COM, PROGRAM, OOMS} \\
\text{subsorts} & \quad \text{COM} \rightarrow \text{OOMS} \\
\text{constructors} & \quad \text{Programs} \\
& \quad \text{coms : COM} \rightarrow \text{OOMS} \\
& \quad \text{Commands} \\
& \quad \text{quote : VALUE} \rightarrow \text{COM} \\
& \quad \text{car :} \rightarrow \text{COM} \\
& \quad \text{cdr :} \rightarrow \text{COM} \\
& \quad \text{cons :} \rightarrow \text{COM} \\
& \quad \text{push :} \rightarrow \text{COM} \\
& \quad \text{swap :} \rightarrow \text{COM} \\
& \quad \text{op :} \rightarrow \text{COM} \\
& \quad \text{branch : COMS} \times \text{OOMS} \rightarrow \text{COM} \\
& \quad \text{cur :} \rightarrow \text{COM} \\
& \quad \text{app :} \rightarrow \text{COM} \\
& \quad \text{rec :} \rightarrow \text{COM} \\
\text{Values} & \quad \text{int} : \rightarrow \text{VALUE} \\
& \quad \text{bool} : \rightarrow \text{VALUE} \\
\end{align*}
\]

\[\text{Figure 4. Abstract syntax of CAM code}\]

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

- integers in \( \mathbb{N} \),
- truth values \( \text{true and false} \),

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

- closures of the form \( [\sigma, \rho]_{\text{cam}} \), where \( \sigma \) is a fragment of CAM code and \( \rho \) 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

\[
s \vdash \sigma \Rightarrow \alpha
\]
meaning that program \( \sigma \), started in state \( s \) returns \( \alpha \) as result. All other rules involve only sequents of the form

\[
\sigma \vdash \alpha \Rightarrow \sigma'
\]

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

\[
\begin{align*}
\text{init.stack} & \vdash \text{COMS} \Rightarrow s \cdot \alpha \\
\vdash \text{program}(\text{COMS}) \Rightarrow \alpha \\
\sigma & \vdash \sigma \Rightarrow s \\
\sigma & \vdash \text{COM} \Rightarrow s_1 \\
\sigma_1 & \vdash \text{COMS} \Rightarrow s_2 \\
\sigma & \vdash \alpha \Rightarrow \text{quote}(v) \Rightarrow s \cdot v \\
\sigma & \vdash (\alpha, \beta) \Rightarrow \text{car} \Rightarrow s \cdot \alpha \\
\sigma & \vdash (\alpha, \beta) \Rightarrow \text{cdr} \Rightarrow s \cdot \beta \\
\sigma & \vdash \alpha \cdot \beta \Rightarrow \text{cons} \Rightarrow s \cdot (\alpha, \beta) \\
\sigma & \vdash \alpha \cdot \beta \Rightarrow \text{push} \Rightarrow s \cdot \alpha \\
\sigma & \vdash \alpha \cdot \beta \Rightarrow \text{swap} \Rightarrow s \cdot \beta \\
\sigma & \vdash \text{eval} \Rightarrow s' \cdot \beta \\
\sigma & \vdash \text{op} \Rightarrow \text{op} \Rightarrow s \cdot \beta \\
\sigma & \vdash \text{true} \Rightarrow \text{branch}(0, 0) \Rightarrow s_1 \\
\sigma & \vdash \text{false} \Rightarrow \text{branch}(0, 0) \Rightarrow s_1 \\
\sigma & \vdash \text{cur}(c) \Rightarrow \text{com} \Rightarrow s_1 \\
\sigma & \vdash (\rho, \alpha) \Rightarrow c \Rightarrow s_1 \\
\sigma & \vdash (\rho, \alpha) \Rightarrow \text{app} \Rightarrow s_1 \\
\sigma & \vdash \text{rec}(c) \Rightarrow c \Rightarrow s_1 \\
\sigma & \vdash \text{rec}(c) \Rightarrow c \Rightarrow s_1 \\
\sigma & \vdash \text{letrec}(\text{fact} = ) \\
\end{align*}
\]

\textbf{Figure 5.} The definition of the Categorical Abstract Machine

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 \text{init.stack} contains a single element, the initial environment. This environment includes the closures corresponding to the predefined operators. For example, we might have

\[
\text{init.stack} = \{ ((), [\text{cdr}; \text{op} + , (\text{com})], [\text{cdr}; \text{op} - , (), \text{com}], [\text{cdr}; \text{op} * , (), \text{com}] \}.
\]

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 \text{eval} for predefined operators. The external evaluator must be aware that it receives a single tree-structured argument.

Rule 11 and 12 define the \text{branch} instruction. It takes its (evaluated) condition from the top of the stack, and continues with either the true or the false part. The \text{cur} instruction is described in rule 13: \text{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 \text{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 \text{rec} instruction is used to build the self referencing environment \( \rho_1 \). Such an environment is necessary for the evaluation of recursive definitions. Notice the remarkable simplicity of rule 15.

\textbf{5.1.3. Remarks}

- Given some program \( P \) written in CAM code, to run it is to find an \( \alpha \) 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 formality as a semantic definition. The equivalence of certain sequences of code can be proved easily and this is useful for optimisation [8].

\textbf{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:

\[
\begin{align*}
\text{push} & ; \\
\text{letrec} \text{fact} & = \lambda x. \text{if} \\
& (x) \\
& \text{then} 1 \text{ else} \\
& (x) \\
\text{swap} & ; \\
\text{cons} & ; \text{app} \\
\text{call} & \\
\text{push} & ; \\
\text{cdr} & ; \\
\text{cons} & ; \text{app} \\
\text{call} & \\
\end{align*}
\]

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

\[
\rho \vdash s \Rightarrow e
\]

where \( \rho \) is an environment, \( s \) is an expression in Mini-ML, and \( e \) is its translation into CAM-code. In words, the sequent states: in environment \( \rho \), expression \( s \) is compiled into code \( e \). 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 \text{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 \text{car} and \text{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 \( \lambda \)-term is compiled and then wrapped in a \text{cur} instruction. To understand the next rules, the following inductive assertion is useful: the code for an expression expects its evaluation environment on

\(^1\) The proof of correctness of this translation is given in [7]
Figure 6. Translation from mini-ML to CAM

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

\[ \lambda x. x \equiv (\text{let } p = x, \text{ in } x) \]

we deduce that the following inference rule should be valid

\[ \rho \vdash x_1 \rightarrow c_1 \quad (p, \rho) \vdash x_2 \rightarrow c_2 \]

Indeed, it is shown in [5] 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 the the program that is to be translated type-checks, and in particular that it contains no undeclared variables. The rules in this set are similar to the rules in the set VAL.OP, except that while searching for an identifier, one constructs simultaneously its access path.

6. Type-checking Mini-ML

6.1. General framework

Figure 7. Generating access paths for identifiers

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 \( \epsilon \) has type \( r \), it also means that the possible values that \( \epsilon \) may take during execution are characterized by \( r \). When axiomatizing

\[ \rho \vdash \epsilon : r \]

where \( \rho \) is a type environment, both viewpoints are present. If the sequent has no proof, then \( \epsilon \) does not type-check. If it can be proved, then \( \epsilon \) has type \( r \). 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 \( \lambda \)-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 \( \text{int} \rightarrow \text{int} \). In the same way the identity function \( \lambda x.x \) for integers has type \( \text{int} \rightarrow \text{int} \), but for booleans it has type \( \text{bool} \rightarrow \text{bool} \). 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 \( \alpha \) 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 \( r \) is either
  i. a basic type \( \text{int}, \text{bool} \)
  ii. a type variable \( \alpha \)
  iii. a functional type \( r' \rightarrow r'' \), where \( r \) and \( r' \) are types,
  iv. a product type \( r \times r' \), where \( r \) and \( r' \) are types.

Type schemes: a type-scheme \( \sigma \) is either
  i. a type \( r \),
  ii. a type-scheme \( \forall \alpha. \sigma \), where \( \sigma \) 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 \( FV(\epsilon) \) for the set of free variables of a type expression \( \epsilon \). We now define two relations between type expressions that contain type variables.

Definition. A type scheme \( \sigma' \) is called an instance of a type scheme \( \sigma \) if there exists a substitution \( S \) of types for free type variables such that:

\[ \sigma' = S \sigma. \]
Definition. A type scheme \( \sigma = \forall a_1 \ldots \forall a_n. \tau \) has a generic instance \( \sigma' = \forall \beta_1 \ldots \forall \beta_n. \tau' \), and we shall write \( \sigma \succeq \sigma' \), if there exists a substitution \( S \) such that
\[
r' = S \tau \quad \text{with} \quad D(S) \subseteq \{a_1 \ldots a_n\}
\]
and the \( \beta_i \) are not free in \( \sigma \), i.e.
\[
\beta_i \notin FV(\sigma) \quad 1 \leq i \leq n.
\]

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 \( r \) and \( r' \) are types rather than type-schemes, then \( r \succeq r' \) implies \( r = r' \).

6.3. The typing rules

The rules for typing Mini-ML programs are given in Figure 8. Two judgements are used. The main judgement for typing Mini-ML programs has the form
\[
\rho \vdash e : r
\]
where the environment \( \rho \) is a list of assumptions of the form \( x : \sigma \). The environment is to be scanned from right to left. Concatenation of two environments \( \rho \) and \( \rho' \) is noted \( \rho \cdot \rho' \). An auxiliary judgement is necessary to handle the declaration of patterns correctly:
\[
\vdash p, r : \rho.
\]
This formula means that declaring \( p \) with type \( r \) creates the type environment \( \rho \). For example we have:
\[
\vdash (x, (y, z)), (r_1 \times (r_2 \times r_3)) : x : r_1, y : r_2, z : r_3
\]

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 \( \sigma \) to an identifier, but in ML an occurrence of an identifier has a type \( r \). So the condition in rule 5 specifies to create a generic instance of \( r \) of the type scheme \( \sigma \).

Rule 6 is the elimination rule for the basic type \( \text{bool} \). It is interesting to note that there are two occurrences of \( r \) in the premises, so that the rule unifies the types of the two expressions \( e_3 \) and \( e_4 \). Rule 7 introduces product types. Rule 9 is the elimination rule for \( \rightarrow \). It performs unification on \( r' \) in the same way as rule 6.

From rules 4 and 8 we deduce
\[
\vdash p, r : \rho, r' \vdash e : r_3 \quad \rho \vdash \text{let } p = e_3 \text{ in } r_1
\]
which differs from rule 9. So from the point of view of types,
\[
\rho \vdash e : r \quad \text{let } p = e_3 \text{ in } r_1
\]

Rule 9 is the source of polymorphism in Mini-ML: the environment \( \rho'' \) is obtained from \( \rho' \) by generalisation with respect to \( \rho \). It is important to understand that generalisation \( \rho' \) creates several independent type schemes, while generalizing \( r_3 \) would give a single type scheme. Rule 10 defines recursively \( r_3 \) in terms of itself and it exhibits the usual similarity with rule \( \rho' \). There is no point in trying to generalise \( \rho' \) with respect to \( \rho \cdot \rho' \).

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.

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 \( \lambda x.x \).

- In Mini-ML, the Subject Construction Theorem is valid [5]. Consider the proof tree for
\[
\rho \vdash e : r
\]

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 mecha-
cally 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