The Natural Dynamic Semantics of Mini-Standard ML

Dominique Clément
SEMA-METRA
INRIA SOPHIA-ANTIPOLIS
Rue Emile Hugues, 06560 Valbonne/France

Abstract

We describe how to express the dynamic semantics of a small subset of the Standard ML language in Natural Semantics. The present specification is based on a communication of R.Milner that describes the dynamic semantics of Standard ML in a structural style, and can be viewed as an example of the "programming effort" that is necessary to obtain an executable version of such a specification. The main aspects of Natural Semantics covered concern its relationships with typed inference systems and with some properties of natural deduction. The description has been tested on a computer but we do not give here details on the compilation techniques.

1. Introduction

The use of inference systems to specify the static and dynamic semantics of programming languages has its origin in the presentation of semantics in a structural axiomatic style in Plotkin[15]. For example, to express that a "phrase P evaluates to a value α in an environment e" we can write a formal sentence of the form:

$$e \vdash P : \alpha$$

where the evaluation predicate ":" is defined by a set of axioms and inference rules. Such a system formally defines the sound phrases (with respect to dynamic semantics) of a programming language as those that can be inferred from the system. In other words, the *evaluation* of a phrase P to a value α is defined by the existence of a derivation tree for $e \vdash P : \alpha$.

Natural Semantics is a specification formalism originating in Plotkin's structural semantics but with flavors of Gentzen's natural deduction[3], [10]. A specification in Natural Semantics is defined by inference rules involving several *judgements*. For dynamic semantics judgements are generally of the form $e \vdash P : \alpha$, meaning that the term P has value α in context e. Then it is possible to prove formal properties of these specifications: Natural Semantics has be used to prove the correctness of translations[6] for the central part of the ML language.

But beside this purely descriptive aspect, a key question of interest is the use of logical systems as *executable specification* formalisms. Natural semantics is one such executable specification formalism. Specifications are written in Typol[7], a language that implements Natural Semantics, and compiled to produce typecheckers, interpreters, and translators[4].

In this paper we present the natural dynamic semantics of Standard ML, or more exactly of a subset of Standard ML. As pointed out by Milner[12] the design of Standard ML is based on simple and well understood ideas that have been experimented with in previous versions of ML or in other functional languages. Furthermore Milner gives a formal definition of the dynamic semantics of core Standard ML[13]

This work is partially supported under ESPRIT, Project 348

in a structural axiomatic style. Hence we have the opportunity to use Natural Semantics on a completely specified language.

In the next section we present the main aspects of Natural Semantics that we use to specify the dynamic semantics of ML. Then we describe the subset of Standard ML used in the sequel of this paper together with the relevant semantic domains. In the following two sections we discuss two aspects of ML, namely exceptions and pattern matching, that need special attention. The first one is a direct application of the notion of judgements, while the second one is also related to the specification of *negation* within inference rules. Finally we give the semantics of expressions, followed by all rules that are necessary to complete the specification of the dynamic semantics of our subset of Standard ML.

2. Natural Semantics

A specification in Natural Semantics is an inference system, i.e. a collection of inference rules that have the following form:

$$\frac{hypotheses}{conclusion}$$

where both *hypotheses* and *conclusion* are of the form $\Gamma \vdash term : \alpha$, where Γ is a set of hypothesis on, at least, the variables of *term*. In such a formula, which is called a *sequent*, the context Γ, the subject *term*, and the value α are *abstract trees* that belong to a finite system of types called an abstract syntax definition. The identification of Natural Semantics objects with abstract syntax trees is a central aspect of this formalism.

First this identification implies that the tree terms used within Natural Semantics act as *type constructors*. Next this identification implies that every variable in Natural Semantics, also called meta-variable, stands for values that belong to some abstract syntax definition. In other words the meta-variables of a specification in Natural Semantics are *typed*. We illustrate these two aspects on simple examples.

2.1. Terms are type constructors

An abstract syntax is defined by a system of types with *sorts*, *subsorts*, and *functions*. The notion of subsorts is used to express containment relations between sorts.

Consider the system of types with two sorts VAR and EXP, with the relation $\text{VAR} \prec \text{EXP}$ to express that a variable is also an expression, and with two functions: $var :\rightarrow \text{VAR}$ and $application : \text{EXP} \times \text{EXP} \rightarrow \text{EXP}$. In this system of types a tree term such as $application(varF, varX)$ is of type EXP, while the two subtrees $varF$ and $varX$ are of type VAR. Hence in a sequent of the form $\Gamma \vdash application(varF, varX) : \alpha$, the subject stands for applications where the operator and the operand are both restricted to be object language variables.

Now consider the term $application(\text{OPERATOR}, \text{OPERAND})$, where OPERATOR and OPERAND are meta-variables. This term is of type EXP if and only if the two variables OPERATOR and OPERAND are of type EXP. In the sequent $\Gamma \vdash application(\text{OPERATOR}, \text{OPERAND}) : \alpha$, the subject now stands for applications where both the operator and the operand are general expressions. This means in particular that the values of these two variables must be tree terms that belong to the sort EXP (or to subsorts of EXP, such as the sort VAR).

Hence the abstract syntax tree terms used within Natural Semantics act as type constructors. Furthermore the meta-variables that occur strictly within a tree term are implicitely *typed* by an abstract

syntax definition. In fact every meta-variable within a specification in Natural Semantics is typed, and this typing is of primary importance for the style of Natural Semantics specifications, as we explain now.

2.2. Variables are typed

As in other first order logic languages, a variable in Natural Semantics is used to impose equality constraints among subterms or to share information between different objects. But a variable occurrence in a Natural Semantic inference rule also expresses a *constraint* on the values that can be substituted to that variable.

Consider the following rule that could be used to specify the evaluation of an ML expression when it is reduced to an ML variable:

$$e \vdash x : \alpha \qquad (e(x) = \alpha)$$

Assume that the variable e denotes an environment that is a mapping of ML variables to ML values, and that the variable α denotes an ML value. Now to express that this rule is restricted to ML variables, i.e. on tree terms of the form *var* X, the meta-variable x must be declared of type VAR. Without such a containment on the values of the meta-variable x, any ML expression could be substituted to x.

Hence a meta-variable in Natural Semantics can be used as an *instantiation filter* to restrict the domain of validity of an inference rule. But this filtering on the values of meta-variables is directly computed from the abstract syntax definitions. Define the *phylum* associated to a sort as the set of functions obtained from the partial order defined by the subsort relations. With our example of system of types with two sorts VAR and EXP, the phylum associated to the sort VAR is reduced to the set $\{var\}$, while the phylum associated to the sort EXP is the set $\{var, application\}$. Then a meta-variable of type S denotes a variable that can only be substituted by a term whose root symbol belongs to the phylum P identified with type S, i.e. it is equivalent to an untyped variable v that satisfies a boolean predicate of the form $v \in$ P.

The typing of meta-variables presented here provides a nice modularization mechanism for Natural Semantics specifications. In particular the type information on meta-variables can be used to distinguish rules that express different evaluations of the same construct of a programming language. As we shall see later, this situation is central in the specification of the dynamic semantics of Standard ML because of the ML exception mechanism. Note that this typing is essentially a matter of style. For instance in the example above, it is possible to use the tree term *var* X instead of the meta-variable x to restrict the use of the inference rule to ML variables. More generally any inference rule with type information can also be expressed as an inference rule without type information but with auxiliary boolean predicates.

3. Abstract Syntaxes

The Standard ML language defined in [12] is a quite complete functional programming language, even without Input/Output primitives nor Modules for separate compilation. For the purpose of this paper we only consider a subset of the Core language that includes the most relevant features of Standard ML in the context of their specification in Natural Semantics. A complete specification of the dynamic semantics of the full Standard ML language has also been done[5].

From Standard ML we keep the following constructs:

+ Declarations: value and exception declarations using value bindings and exception bindings,
+ Expressions: application, raising and handling exceptions, and function abstraction,

+ Patterns: to create value bindings by pattern matching.

but we omit the following:

- type, datatype, abstract datatype declarations, and type expressions (all of them are relevant of static semantics),
- local declarations, both in declarations (using **local**) and in expressions (using **let**),
- sequences in value bindings and exception bindings,
- recursive value bindings,
- labelled records and the layered pattern construct,
- side-effect constructs, i.e. references and assignment.

3.1. Abstract Syntax of Core Mini-ML

The principal syntax classes of our Mini-Standard ML language are defined in terms of the three disjoint primitive classes given in Figure 1.

sorts VAR, CON, EXN

functions

var	:	$\rightarrow$	VAR	value variables
con	:	$\rightarrow$	CON	value constructors
exn	:	$\rightarrow$	EXN	exception names

Figure 1. Primitive Classes

An exception name is always completely determined by its occurrence in abstract syntax trees. But this is not true for value variables and value constructors for which the scope of datatype bindings must be taken into account. We assume that any ambiguity on the class of an identifier has been solved (by a type-checker for example), i.e. we assume in the following that ML abstract syntax trees are always well formed. Examples of value constructors are: booleans *true* and *false*, list constant *nil* and list constructor "::". Value variables occur in value bindings such as "$x = 1$".

The abstract syntax of declarations, value and exception bindings, patterns, and expressions is given in Figure 2.

The purpose of declarations is to bind identifiers to values. Value bindings are used to declare value variables while exception bindings are used to declare exceptions. An exception binding is either a simple exception binding or an exception name: the sort EXN is a subsort of the sort EXCBIND.

Patterns are linear terms containing only variables and value constructors. In our subset of Standard ML the unique compound pattern is the *construction* of the form "*con pat*". Finally, atomic expressions are value variables and value constructors (see the subsort section). Compound expressions are the function abstraction **fun**, the application *exp exp'*, raising exceptions with **raise**, and handling exceptions with **handle**.

"Declarations"

sorts DEC

functions

val	:	VALBIND	→	DEC	**val** *valbind*
exception	:	EXCBIND	→	DEC	**exception** *excbind*

"Value Bindings"

sorts VALBIND

functions

simple_value	:	PAT×EXP	→	VALBIND	$pat = exp$

"Exception Bindings"

sorts EXCBIND

subsorts EXN ≺ EXCBIND

functions

simple_excbind	:	EXN×EXN	→	EXCBIND	$exn{=}exn'$

"Patterns"

sorts PAT

subsorts (VAR, CON) ≺ PAT

functions

construction	:	CON×PAT	→	PAT	*con pat*

"Expressions"

sorts EXP, MATCH, MRULE, HANDLER, HRULE

subsorts (VAR, CON) ≺ EXP

functions

fun	:	MATCH	→	EXP	**fun** *match*
application	:	EXP×EXP	→	EXP	*exp exp'*
raise	:	EXN×EXP	→	EXP	**raise** *exn* **with** *exp*
handle	:	EXP×HANDLER	→	EXP	*exp* **handle** *handler*
match	:	MRULE^+	→	MATCH	$mrule_1 \mid \cdots \mid mrule_n$
mrule	:	PAT×EXP	→	MRULE	$pat \Rightarrow exp$
handler	:	HRULE^+	→	HANDLER	$hrule_1 \parallel \cdots \parallel hrule_n$
with	:	EXN×MATCH	→	HRULE	*exn* **with** *match*

Figure 2. Abstract Syntax of ML

3.2. Abstract Syntaxes of Semantic domains

Now we need to define semantic domains such as the domain of values VAL and the environment domain ENV. To define the abstract syntaxes of these domains we will need four sorts of the abstract syntax definition of Mini Standard ML: the sort MATCH, the sort CON, the sort VAR, and the sort EXN. All these sorts are *imported*. This means that we import the language defined by the reflexive closure of each one of these four sorts.

The abstract syntax of values is given in Figure 3. A function value is a partial function represented as a **closure**. A closure is a pair of a function body, i.e. an ML match *match*, and of an environment *e*.

The value of a constructor is that value constructor, the sort CON is a subsort of VAL, and a construction value is a product of a value constructor and of an ML value. Following Milner, basic functions such as "+" and "−" are defined as members of the sort BASFUN, which is a subsort of VAL. But for each basic function f in BASFUN, **apply**(f, α) denotes the result of applying f to a value α. Such partial functions are members of the sort APPLY.

sorts VAL, BASFUN, APPLY
subsorts (CON, BASFUN, APPLY) $\prec$ VAL
functions

closure	:	MATCH×ENV	→	VAL	$[\![match, e]\!]$
prod	:	CON×VAL	→	VAL	(con, val)
apply	:	BASFUN×VAL	→	APPLY	$f(val)$

Figure 3. Values

To associate values to ML variables and exceptions to ML exception names we use an **environment** defined by the abstract syntax given in figure 4. This environment is a *list* of pairs.

sorts ENV, PAIR, VAR_PAIR, EXN_PAIR
subsorts (VAR_PAIR, EXN_PAIR) $\prec$ PAIR
functions

var_pair	:	VAR×VAL	→	VAR_PAIR	$var \mapsto val$
exn_pair	:	EXN×EXC	→	EXN_PAIR	$exn \mapsto exc$
env	:	PAIR*	→	ENV	environment e

Figure 4. Environments

An **exception** is an object that belongs to the sort EXC and to which an exception name *exn* may be associated. The nature of an exception is immaterial. A **packet** is a pair of an exception and an "excepted" value, and it is the unique operator of the sort PACK. Neither exceptions nor packets are values. Finally the singleton sort FAIL denotes failure (Figure 5).

sorts EXC, PACK, FAIL
functions

pack	:	EXC×VAL	→	PACK	packet *pack* or $<exc, val>$
exc	:		→	EXC	exception *exc*
fail	:		→	FAIL	failure *fail*

Figure 5. Packets, Exceptions, and Fail

3.3. Environment as context

Finally we need to define the manipulation primitives on the environment domain ENV. A first possibility is to give a functional definition of these environment manipulation primitives, as Milner does in [13]. First the environment is defined as the product of a value environment *ve* and of an exception environment *ee*. Each component of an environment is considered as a member of $MAP(S, S')$, the set of finite partial functions from a set S to a set S′. The basic operations on mappings are defined as follows:

Primitives on mappings.

i) If m belongs to $MAP(X,Y)$ then

$$m(x) = y \quad \Leftrightarrow \quad (x,y) \in m$$

ii) For two maps m and m', the map $m + m'$ is defined by:

$$m + m'(x) = \begin{cases} m'(x) & \text{if } m'(x) \text{ is defined,} \\ m(x) & \text{otherwise.} \end{cases}$$

Then environment primitives are defined in terms of these basic operations with the help of some notations: if $e = (ve, ee)$ and $e' = (ve', ee')$ then $e + e'$ denotes $(ve + ve', ee + ee')$; furthermore $e + ve'$ denotes $(ve + ve', ee)$ and $e + ee'$ denotes $(ve, ee + ee')$.

Although such a functional definition of environment manipulations is perfectly meaningful, it is possible to give another definition that is more in the style of Natural Semantics. First the environment e is considered as a list of *propositions* of the form $x : \alpha$ or $exn : exc$, where the meta-variables x, α, exn, and exc denote respectively an ML variable, an ML value, an ML exception name, and an exception. PAIR* is the set of finite sequences of such propositions. We write an empty sequence "[]" (the empty environment) and $e[x : \alpha]$ the sequence obtained from the sequence e by adding one more assumption [10]. Then to look for the value associated to an ML variable x in an environment e, we define the sequent $\overset{\text{val\_of}}{\vdash}$ as follows:

set VAL_OF is

$$e[x : \alpha] \vdash x : \alpha \tag{1}$$

$$\frac{e \vdash x : \alpha}{e[y : \beta] \vdash x : \alpha} \qquad (y \neq x) \tag{2}$$

$$\frac{e \vdash x : \alpha}{e[exn : exc] \vdash x : \alpha} \tag{3}$$

end VAL_OF;

Rule 1 is very similar to the tautology $x : \alpha \vdash x : \alpha$, while rule 2 is akin of the thinning rule in Natural Deduction. The third rule is necessary to skip assumptions on exception names. To look for the exception associated to an exception name *exn*, we use the sequent $\overset{\text{exc\_of}}{\vdash}$, that is defined with the same kind of rules. The role of the environment in natural semantics is now quite explicit: it is the *context*, i.e. a set of hypothesises on identifiers, necessary to derive the value of a phrase. Note that we assume here that the environment contains at least one assumption on all free identifiers, ML variables and ML exception names, of the ML term that is evaluated. This consistency property should be proved as a lemma satisfied by well typed ML programs.

4. Specification of Exceptions

The principle followed by Milner[13] to specify the dynamic semantics of Standard ML is to define an *evaluation* relation "⇒" with inference rules, called *evaluation rules*, of the form:

$$\frac{e_1 \vdash \mathrm{P}_1 \Rightarrow r_1 \cdots e_n \vdash \mathrm{P}_n \Rightarrow r_n}{e \vdash \mathrm{P} \Rightarrow r}$$

A formal sentence such as $e \vdash \mathrm{P} \Rightarrow r$ expresses that, in a given environment e, the phrase P evaluates to a result r. Of course the nature of r depends on the syntax class of P. For example the result r is an environment when P is an ML declaration, but it is an ML value when P is an ML expression. So the evaluation of Standard ML is formally defined by an inference system, from which sentences may be inferred.

Then it is necessary to specify the exception mechanism of Standard ML, which is defined by the following principle of exception propagation[12]: whenever a sub-phrase evaluates to a packet then no further sub-evaluations occur and the exception is propagated, i.e. the packet is also the result of the main phrase. This is achieve by adding, for every evaluation rule, n further rules, one for each k with $1 \leq k \leq n$, called packet propagation rules. Generally packet propagation rules for a term P are of the form:

$$\frac{e_1 \vdash P_1 \Rightarrow r_1 \quad \cdots \quad e_k \vdash P_k \Rightarrow \mathit{pack}}{e \vdash P \Rightarrow \mathit{pack}}$$

where the variables r_i, with $1 \leq i \leq k-1$, are not packets.

In Natural Semantics we follow the same kind of approach, but we consider that the dynamic semantics of Standard ML is defined with different *judgements* on ML terms. First, we define the evaluation of ML phrases with judgements of the form $e \vdash \mathrm{P} : r$ where the result r is not a packet. Then we specify exceptions of Standard ML with judgements of the form $e \vdash \mathrm{P} : p$ where p can only be a packet.

For example the dynamic semantics of Mini-ML declarations is defined by the judgement:

$$e \vdash \mathrm{DEC} : e'$$

where both e and e' are of type ENV. Then, because a declaration may evaluate to a packet, we define another judgement:

$$e \vdash \mathrm{DEC} : \mathit{pack}$$

where the variable *pack* is of type PACK. Now the exception propagation mechanism of Standard ML can be defined as *transition rules* between these different kinds of judgements.

As a small digression, we would like to indicate how the type constructor interpretation of abstract terms is used to make our specification executable. Consider our judgements as ternary predicates "_ ⊢ _ : _", one for each form of judgements, that belong to a meta-system: such predicates can be evaluated within the meta-system in a Prolog like manner. Now the operator "_ ⊢ _ : _" is heavily overloaded, and to execute these predicates in the meta-system it is necessary to solve this overloading (this is one of the purposes of the compiler of Natural Semantics specifications). This is achieved by computing the type of each predicate from the abstract syntax definitions of the object terms. Note that, to avoid such an overloading, we could have defined two syntacticaly different judgements, one of the form $e \vdash \mathrm{P} : r$, the other of the form $e \vdash \mathrm{P} :: \mathit{pack}$. With this approach exception propagation rules are *coercion* rules between

these judgements. But we prefer to use the type information on meta-variables to distinguish judgements on terms that belong to the same sort, but on disjoint semantic domains.

We return now to the specification of exceptions. For Mini-Standard ML we have the following types for our judgements, classified according to the syntactic nature of the subject P:

1) **declarations:** $e \vdash \text{DEC} : e$ of type ENV × DEC × ENV and $e \vdash \text{DEC} : \mathit{pack}$ of type ENV × DEC × PACK,

2) **value bindings:** $e \vdash \text{VALBIND} : e'$ of type ENV × VALBIND × ENV and $e \vdash \text{VALBIND} : \mathit{pack}$ of type ENV × VALBIND × PACK,

3) **exception bindings:** $e \vdash \text{EXCBIND} : e'$ of type ENV × EXCBIND × ENV,

4) **expressions:** $e \vdash \text{EXP} : \alpha$ of type ENV × EXP × VAL and $e \vdash \text{EXP} : \mathit{pack}$ of type ENV × EXP × PACK.

To illustrate our approach, consider Mini-Standard ML declarations. First we define the judgement $e \vdash \text{DEC} : e'$, Figure 6. This judgement corresponds to the normal evaluation of an ML declaration.

$$\frac{e \vdash \text{VALBIND} : e'}{e \vdash \text{val VALBIND} : e'} \quad (1)$$

$$\frac{e \vdash \text{EXCBIND} : e'}{e \vdash \text{exception EXCBIND} : e'} \quad (2)$$

Figure 6. Evaluating a Declaration

The evaluation of a value declaration and of an exception declaration, rules 1 and 2, are expressed in terms of the evaluation of their bindings, respectively value binding and exception binding. They both evaluates to an environment e'. Then we define the judgement $e \vdash \text{DEC} : \mathit{pack}$ of type ENV × DEC × PACK for exceptions, Figure 7. It is only necessary to add a rule because ML exception declarations never evaluate to a packet.

$$\frac{e \vdash \text{VALBIND} : \mathit{pack}}{e \vdash \text{val VALBIND} : \mathit{pack}} \quad (3)$$

Figure 7. Exception in Declarations

5. Specification of Pattern-matching

Another interesting aspect of the dynamic semantics of Standard ML is the use of the same mechanism for value bindings in value declarations, **val** $x = 1$, parameter bindings in applications, (**fun** *match*) e, and exception handlings, *exn* **with** *match*. This is achieved with a pattern-matching mechanism between ML patterns and ML values. For example a conditional expression **if** e_1 **then** e_2 **else** e_3 is equivalent to the following application:

$$\begin{pmatrix} \textbf{fun} & \mathit{true} \Rightarrow e_2 \\ & \mathit{false} \Rightarrow e_3 \end{pmatrix} e_1$$

where $\mathit{true} \Rightarrow e_2$ and $\mathit{false} \Rightarrow e_3$ are the so-called *mrules* of the match.

Given a pattern PAT and an ML value α the first purpose of ML matching is to act as a filter between the structure of the pattern and the structure of the value. The second purpose of ML matching is to

bind the ML variables that occur in the pattern with ML values. Furthermore ML pattern matching is independent of the environment. Hence it is possible to describe the ML pattern-matching mechanism with a judgement $\vdash \text{PAT}, \alpha : e$ of type PAT × VAL × ENV. For our subset of Standard ML, this judgement is defined by the rules given in Figure 8.

$$\vdash x, \alpha : [x : \alpha] \tag{1}$$

$$\vdash con, con : \emptyset \tag{2}$$

$$\frac{\vdash \text{PAT}, \text{VAL} : e}{\vdash \text{CON PAT}, (\text{CON}, \text{VAL}) : e} \tag{3}$$

Figure 8. Matching a Pattern to a Value

The rule 1 shows the binding facet of ML pattern matching: an ML variable matches any value and the environment $[x : \alpha]$ is built. A value constructor matches only with the *same* value constructor, and no environment is built, rule 2. Finally the rule 3 specifies the matching of a construction with a product value. Both must have the same constructor as first component, then pattern-matching is recursively applied to their second components.

But this judgement only describes valid pattern matchings, i.e. matchings that do not *fail*. Indeed as a filter the matching fails for some pairs of patterns and values. Rather informally the ML pattern-matching fails when no proof tree can be obtained from the previous system. In some sense we are faced with the well known problem of negation in inference rules. To solve the difficulty we propose a solution based on rules conditioned by boolean predicates.

We define another form of judgement $\vdash \text{PAT}, \alpha : \mathit{fail}$, of type PAT × VAL × FAIL, by the rules of Figure 9. Basicaly the ML pattern-matching only fails for incompatible pairs of ML pattern and of ML value. Note that an ML variable matches with any value: this matching never fails. For an ML constructor, the matching fails for every ML value that is not "equal" to this constructor, rule 4. Rule 5 expresses the same kind of condition for an ML construction. But there is another case of failure for ML constructions because the matching of the pattern PAT and of the value VAL may fails, rule 6.

$$\vdash con, val : \mathit{fail} \qquad (val \neq con) \tag{4}$$

$$\vdash \text{CON PAT}, val : \mathit{fail} \qquad (val \neq (\text{CON}, \text{VAL})) \tag{5}$$

$$\frac{\vdash \text{PAT}, \text{VAL} : \mathit{fail}}{\vdash \text{CON PAT}, (\text{CON}, \text{VAL}) : \mathit{fail}} \tag{6}$$

Figure 9. Failure of ML Matching

Remark: the rules that describe normal evaluation and exception propagation are sometimes rather similar. Consider the two rules 1 and 3 for declarations, figures 6 and 7, and the two rules 3 and 6 for expressions, figures 8 and 9. With some loss in modularity, it is possible to merge these rules as follows:

$$\frac{e \vdash \text{VALBIND} : e_p}{e \vdash \text{val VALBIND} : e_p} \qquad \frac{\vdash \text{PAT}, \text{VAL} : e_f}{\vdash \text{CON PAT}, (\text{CON}, \text{VAL}) : e_f}$$

where the meta-variables e_p and e_f are respectively of type ENV∪PACK and ENV∪FAIL. But now we have two new judgements of type ENV × DEC × (ENV ∪ PACK) and PAT × VAL × (ENV ∪ FAIL). This technique can be used for the purpose of concision.

6. Specification of Expressions and other classes

To complete the specification of the dynamic semantics of our subset of Standard ML we have to define judgements on matches, handlers, value bindings, exception bindings, and expressions. But all these specifications are done with the approach presented on ML declarations, and they are rather similar. Hence we will only give details on the dynamic semantics of Mini-Standard ML expressions.

6.1. *Expressions*

First consider the judgement $e \vdash \text{EXP} : \alpha$ on variables, value constructors, functions, and applications, Figure 10, but without considering exceptions.

$$\frac{e \overset{\text{val\_of}}{\vdash} x : \alpha}{e \vdash x : \alpha} \quad (1)$$

$$e \vdash \text{con} : \text{con} \quad (2)$$

$$e \vdash \text{fun MATCH} : [\![\text{MATCH}, e]\!] \quad (3)$$

$$\frac{e \vdash \text{EXP} : \text{con} \qquad e \vdash \text{EXP}' : \alpha'}{e \vdash (\text{EXP EXP}') : (\text{con}, \alpha')} \quad (4)$$

$$\frac{e \vdash \text{EXP} : f \qquad e \vdash \text{EXP}' : \alpha \qquad \overset{\text{apply}}{\vdash} f, \alpha : \alpha'}{e \vdash (\text{EXP EXP}') : \alpha'} \quad (5)$$

$$\frac{e \vdash \text{EXP} : [\![match, e']\!] \qquad e \vdash \text{EXP}' : \alpha' \qquad e' \vdash match, \alpha' : \alpha}{e \vdash (\text{EXP EXP}') : \alpha} \quad (6)$$

Figure 10. Evaluating an expression

The set of rules called **val_of**, rule 1, is used to look for the value α associated to the value variable x (this set is decribed in section 3.3). Rules 4, 5, and 6 describe the evaluation of an application according to the result of the evaluation of its operator. Note that the type information on meta-variables *con* and f is used to distinguish rules, rules 4 and 5, that have the same object term, but different premises, $e \vdash \text{EXP} : \text{con}$ and $e \vdash \text{EXP} : f$. In rule 5 the meta-variable f is of type BASFUN∪APPLY: this rule specifies the evaluation of basic functions. The set **apply** is defined as follows, where the meta-variable b is of type BASFUN:

set APPLY is

$$\vdash b, \alpha : b(\alpha) \quad (1)$$

$$\frac{\alpha'' = \text{eval}(b, \alpha, \alpha')}{\vdash b(\alpha), \alpha' : \alpha''} \quad (2)$$

end APPLY;

The **eval** function is considered as predefined evaluator that is capable of applying a basic function b to values α and α' and to return a value α''. Here the result depends only on the function b, which is for example addition on integers.

Then we define our judgements on expressions that deal with exceptions, i.e. the *raise* expression and the *handle* expression, Figure 11.

$$\frac{e \overset{\text{exc\_of}}{\vdash} \text{EXN} : exc \qquad e \vdash \text{EXP} : \alpha}{e \vdash \text{raise EXN with EXP} : <exc, \alpha>} \tag{7}$$

$$\frac{e \overset{\text{exc\_of}}{\vdash} \text{EXN} : exc \qquad e \vdash \text{EXP} : pack}{e \vdash \text{raise EXN with EXP} : pack} \tag{8}$$

$$\frac{e \vdash \text{EXP} : \alpha}{e \vdash \text{EXP handle HANDLER} : \alpha} \tag{9}$$

$$\frac{e \vdash \text{EXP} : pack \qquad e \vdash \text{HANDLER}, pack : \alpha}{e \vdash \text{EXP handle HANDLER} : \alpha} \tag{10}$$

Figure 11. Expressions with exceptions

A raise expression always evaluates to a packet, which is generated by the raise, rule 7, or which is the result of the evaluation of the expression, rule 8. The set **exc_of** is used to look for the exception *exc* associated to the exception name EXN (see section 3.3). The handler part of an handle expression is only used when the expression part of the handle evaluates to a packet, rule 10. The application of the handler to the packet is described with the judgement $e \vdash$ HANDLER, PACK : VAL.

Finally the judgements $e \vdash \text{EXP} : \alpha$ and $e \vdash \text{EXP} : pack$ are used to specify the exception propagation mechanism in expressions, Figure 12.

$$\frac{e \vdash \text{EXP} : \alpha \qquad e \vdash \text{EXP}' : pack}{e \vdash (\text{EXP EXP}') : pack} \tag{11}$$

$$\frac{e \vdash \text{EXP} : pack}{e \vdash (\text{EXP EXP}') : pack} \tag{12}$$

$$\frac{e \vdash \text{EXP} : f \qquad e \vdash \text{EXP}' : \alpha \qquad \overset{\text{apply}}{\vdash} f, \alpha : pack}{e \vdash (\text{EXP EXP}') : pack} \tag{13}$$

$$\frac{e \vdash \text{EXP} : [\![match, e']\!] \qquad e \vdash \text{EXP}' : \alpha' \qquad e' \vdash match, \alpha' : pack}{e \vdash (\text{EXP EXP}') : pack} \tag{14}$$

$$\frac{e \vdash \text{EXP} : pack \qquad e \vdash \text{HANDLER}, pack : pack}{e \vdash \text{EXP handle HANDLER} : pack} \tag{15}$$

Figure 12. Exception propagation for expressions

The rule 13 is necessary because some basic functions, such as "+" may raise so-called standard exceptions. This implies that a new rule must be added to the set **apply**:

$$\frac{pack = eval(b, \alpha, \alpha')}{\vdash b(\alpha), \alpha' : pack}$$

In the last two rules 14 and 15, both the application of a match to a value and the application of a handler to a packet may evaluate to a packet. Note that a packet which is not trapped by a handler is propagated as it is, rule 15.

6.2. *Applying a match*

In the application of a match $\text{PAT}_1 \Rightarrow \text{EXP}_1 \mid \cdots \mid \text{PAT}_n \Rightarrow \text{EXP}_n$ to a value α, each component of the match, the so-called *mrule* $\text{PAT}_i \Rightarrow \text{EXP}_i$, is applied to the value α from left to right until one succeeds. This is described by the rules 1 and 2 of Figure 13. If none succeeds, then the packet $<ematch, ()>$ is returned, rule 3, where *ematch* is a predefined exception bound to the exception identifier "match". The application of an *mrule* to a value α is described by the rules 4 and 5. When the pattern PAT matches with the value α, the mrule evaluates as the expression EXP, rule 4. But the application fails when the pattern and the value do not match, rule 5.

$$\frac{e \vdash \text{MRULE}, \alpha : \alpha_p}{e \vdash \text{MRULE} \mid \text{MRULE\_S}, \alpha : \alpha_p} \quad (1)$$

$$\frac{e \vdash \text{MRULE}, \alpha : fail \qquad e \vdash \text{MRULE\_S}, \alpha : \alpha_p}{e \vdash \text{MRULE} \mid \text{MRULE\_S}, \alpha : \alpha_p} \quad (2)$$

$$e \vdash \text{match}[], \alpha : <ematch, ()> \quad (3)$$

$$\frac{\vdash \text{PAT}, \alpha : e' \qquad e ; e' \vdash \text{EXP} : \alpha_p}{e \vdash \text{PAT} \Rightarrow \text{EXP}, \alpha : \alpha_p} \quad (4)$$

$$\frac{\vdash \text{PAT}, \alpha : fail}{e \vdash \text{PAT} \Rightarrow \text{EXP}, \alpha : fail} \quad (5)$$

Figure 13. Applying a match

6.3. *Applying a handler*

The application of a handler, $hrule_1 \parallel \cdots \parallel hrule_n$, to a packet *pack* evaluates in a rather similar manner than the application of a match to a value. But the packet is propagated when none of the hrules matches with that packet, rule 3. To apply a **with** to a packet, the exception *exc'* associated to the exception name EXN in the environment *e* must be "equal" to the first component *exc* of the packet, rule 4. Hotherwise the application fails, rule 5. The boolean predicates *eqexc* and *neqexc* are used to test equality on exceptions.

6.4. *Evaluating a value binding*

We give in figure 15 the rules for value bindings. In rule 1 the environment e' is obtained by pattern-matching between the pattern and the value α of the expression. When this matching fails the packet $<ebind, ()>$ is returned, where *ebind* is a predefined exception bound to the exception identifier "ebind". The last rule, rule 3, describes packet propagation.

6.5. *Evaluating an exception binding*

The rules for exception binding are given in Figure 16. In rule 1 the exception identifier exn X is associated to a *new* exception *exc*. But in rule 2 the exception *exc* was already associated to the exception name EXN'.

$$\frac{e \vdash \text{HRULE}, pack : \alpha_p}{e \vdash \text{HRULE} \parallel \text{HRULE\_S}, pack : \alpha_p} \quad (1)$$

$$\frac{e \vdash \text{HRULE}, pack : fail \qquad e \vdash \text{HRULE\_S}, pack : \alpha_p}{e \vdash \text{HRULE} \parallel \text{HRULE\_S}, pack : \alpha_p} \quad (2)$$

$$e \vdash \text{handler}[], pack : pack \quad (3)$$

$$\frac{e \overset{\text{exc\_of}}{\vdash} \text{EXN} : exc' \qquad e \vdash \text{MATCH}, \alpha : \alpha_p}{e \vdash \text{EXN with MATCH}, <exc, \alpha> : \alpha_p} \quad (eqexc(exc', exc)) \quad (4)$$

$$\frac{e \overset{\text{exc\_of}}{\vdash} \text{EXN} : exc'}{e \vdash \text{EXN with MATCH}, <exc, \alpha> : fail} \quad (neqexc(exc', exc)) \quad (5)$$

Figure 14. Applying a handler

$$\frac{e \vdash \text{EXP} : \alpha \qquad \vdash \text{PAT}, \alpha : e'}{e \vdash \text{PAT} = \text{EXP} : e'} \quad (1)$$

$$\frac{e \vdash \text{EXP} : \alpha \qquad \vdash \text{PAT}, \alpha : fail}{e \vdash \text{PAT} = \text{EXP} : <ebind, ()>} \quad (2)$$

$$\frac{e \vdash \text{EXP} : pack}{e \vdash \text{PAT} = \text{EXP} : pack} \quad (3)$$

Figure 15. Evaluating a value binding

$$e \vdash \text{exn X} : \{\text{exn X} \mapsto exc\} \quad (1)$$

$$\frac{e \overset{\text{exc\_of}}{\vdash} \text{exn X}' \mapsto exc}{e \vdash \text{exn X} = \text{exn X}' : \{\text{exn X} \mapsto exc\}} \quad (2)$$

Figure 16. Evaluating an exception binding

7. Related work and conclusion

The use of type information presented in this paper can be related to the work of Aït-Kaci[1] and of Mycroft[14] on typed logic. The approach proposed in this paper is a more modest effort that does not use all the generality of the typing inclusion mechanism developped by Aït-Kaci[1] in the Login language. But our motivations are slightly different: Login is intended to be a programming language for database applications while Natural Semantics is a formal specification formalism. For instance in Natural Semantics we do not seem to need type inheritance.

We have illustrated the use of Natural Semantics to specify the dynamic semantics of an applicative language with exception and pattern-matching. The most important features of Natural Semantics that have been used are the identification of tree terms with abstract syntax trees, tree terms are used as instantiation filters, together with the use of type information on meta-variables. We found that last aspect the key to produce modular and readable Natural Semantic descriptions.

Although we have not considered the implementation aspect of such a specification, the existence of a compiler for Natural Semantics makes it feasible to execute the specification of the dynamic semantics presented in this paper. Techniques to obtain an efficient implementation, and that have been omitted from this paper, are still under development.

Acknowledgements: I would like to thank to J.Despeyroux for fruitful discussions and G. Kahn for detailed suggestions and corrections.

REFERENCES

[1] H. AÏT-KACI, AND R. NASR "Logic and Inheritance", ACM Journal of Logic Programming, pp 219-228, 1986

[2] H. AÏT-KACI, "A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures", Ph.D Thesis, University of Pensylvania, 1984.

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

[4] CLÉMENT D., J. DESPEYROUX, T. DESPEYROUX, G. KAHN, "A Simple Applicative Language: Mini-ML", Conference on Lisp and Functional Programming, 1986.

[5] CLÉMENT D. "The Natural Dynamic Semantics of Standard ML", *to appear as Inria report.*

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

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

[8] GENTZEN G. "The Collected Papers of Gerhard Gentzen", E.Szabo, Noth-Holland, Amsterdam, 1969.

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

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

[11] KAHN G., "Natural Semantics", Proc. of Symp. on Theoritical Aspects of Computer Science, Passau, Germany, February 1987.

[12] MILNER R., "The Standard ML Core Language", Polymorphism, Volume II, Number 2, October, 1985.

[13] MILNER R., "The Dynamic Operational Semantics of Standard ML", Department of Computer Science, University of Edinburgh, Edinburgh, England, April 1985 *Private communication.*

[14] MYCROFT A. AND O'KEEFE R.A., "A Polymorphic Type System for Prolog", Journal of Artificial Intelligence 23(3), pp 295-307, 1984.

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

[16] PRAWITZ D., "Natural Deduction, a Proof-Theoritical Study", Almqvist & Wiksell, Stockholm, 1965.

[17] WATT D. A., "Executable Semantic Descriptions", Software-practice and experience, Vol. 16(1), p.13-43, january 1986.