The expressive power of structural operational semantics with explicit assumptions

The Expressive Power of Structural Operational Semantics with Explicit Assumptions*

Marino Miculan

Dipartimento di Matematica e Informatica
Università di Udine
via Zanon, 6 – I 33100 Udine – Italy
miculan@udmi5400.cineca.it

Abstract. We explore the expressive power of the formalism called *Natural Operational Semantics*, *NOS*, introduced by Burstall and Honsell for defining the operational semantics of programming languages. This formalism is derived from the Natural Semantics of Despeyroux and Kahn. It arises if we take seriously the possibility of deriving assertions in Natural Semantics under assumptions, i.e. using hypothetico-general premises in the sense of Martin-Löf. We investigate to what extent we can reduce to hypothetical premises the notions of store and environment of Plotkin's Structural Operational Semantics. We use this formalism to define the semantics of a functional language which features commands, blocks, procedures, complex declarations, structures and Abstract Data Types. We give the NOS together with the denotational semantics and prove the adequacy of the former w.r.t. the latter. We discuss some other difficulties which arose in the previous treatment of variables in connection with procedures.
Natural Operational Semantics can be easily encoded in formal systems based on λ-calculus type-checking, such as the Edinburgh Logical Framework. We briefly investigate this and discuss some of the design choices.

1 Introduction

In order to establish formally properties of programs, we have to represent formally their operational semantics. A very successful style of presenting operational semantics is the one introduced by Gordon Plotkin and known as *Structural Operational Semantics* (SOS) ([21]). The idea behind this approach is that all computational elaboration and evaluation processes can be construed as logical processes and hence can be reduced to the sole process of formal logical derivation within a formal system.

For example, the SOS of a functional language is a formal system for inferring assertions such as $\rho \vdash M \to m$, where m is the *value* of the *expression* M, and

* Work partially supported by Esprit BRA 6453, *Types for Proofs and Programs*, and italian MURST 40%, 60% grants.

ρ is the *environment* in which the evaluation is performed – usually a function mapping identifiers to values. The intended meaning of this proposition is "in the environment ρ, the evaluation of M gives m".

This style of specification does not have many of the defects of other formalisms (such as automata and definitional interpreters), since it is syntax-directed, abstract and easy to understand. It has been proved to be very successful in various areas of theoretical computer science. It was studied in depth by Kahn and many of his coworkers, and it has been used by Milner with the name of *Relational Semantics*. Nevertheless, the explicit presence of environments in propositions has some drawbacks in practical use:

- the abstraction power is limited: a function which maps identifiers to values amounts to von Neumann's computer's memory.
- Modularity is limited. Modularity of semantic descriptions is an ongoing area of research—see e.g. [17, 24]. Typically, in considering extensions of the language we may be forced to change the evaluation judgment itself. For instance, the judgment can take the form $\rho \vdash \langle M, \sigma\rangle \rightarrow \langle m, \sigma'\rangle$ in the case of expressions with side-effects [21]. Hence, previous rules and derivations are not any more compatible with the new assertion. However, even simple extensions which only introduce new kinds of identifiers and denotable objects (e.g. procedure identifiers and procedures) cause such problems, since the judgment can take the form: $\rho, \tau \vdash M \rightarrow m$.
- The system lacks conciseness: environments appear in all rules but are seldom used. For instance, in the "+" rule, $\frac{\rho\vdash N_1\rightarrow n_1 \quad \rho\vdash N_2\rightarrow n_2}{\rho\vdash N_1+N_2\rightarrow plus(n_1,n_2)}$, ρ plays no rôle: it is merely transferred from conclusion to premises (in a top-down proof development). The environment is effectively used only when we are dealing with identifiers, that is when we either declare an identifier or evaluate it, e.g. in the rule $\frac{\rho\vdash M\rightarrow m \quad [x\mapsto n]\rho\vdash N\rightarrow n}{\rho\vdash \textbf{let } x=M \textbf{ in } N} \rightarrow n$ and the axiom $\rho \vdash x \rightarrow \rho(x)$.
- In order to reason formally about properties of the operational semantics, it is necessary to encode the formal system into some proof-editor/checker. However, in most of the proof assistants, representation of functions (such as the environments) can be rather cumbersome, and mechanized reasoning about these encodings can be very hard.

A possible solution to these drawbacks is the *Natural Operational Semantics* formalism (NOS) introduced in [4] as a refinement of the Natural Semantics originally proposed by Kahn and his coworkers ([6, 12]). This formalism arises if we take seriously the possibility of deriving under assumptions assertions in Natural Semantics, i.e. using hypothetico-general judgments in the sense of Martin-Löf ([19]). It is based in fact on Gentzen's Natural Deduction style of proof ([8]): hypothetical premises are used to make assumptions about the values of variables. In this paper we investigate to what extent we can reduce to hypothetical premises the fragments of the store and environment of Plotkin's Structural Operational Semantics. Thus, instead of evaluating an expression within a given environment, we compute its value under a set of assumptions on the values of its free variables. In other words, we replace explicit environments with implicit contextual structures, that is, the hypothetical premises in Natural Deduction.

For example, consider a functional language with two syntactic classes, *Expr*, the class of expressions (ranged over by M, N), and *Id*, the class of identifiers (ranged over by x, y), the former including the latter. The SOS of this language is a system for deriving judgments of the form $\rho \vdash M \rightarrow m$. Instead, in the NOS paradigm the judgments can be simplified to those of the form $M \Rightarrow m$, whose reading is "the value of expression M is m". There are no more contextual structures: the predicate is $\Rightarrow \subset \mathit{Expr} \times \mathit{Expr}$.

These assertions can be inferred using a Natural Deduction style proof system, that is a set of rules of the form

$$\frac{\begin{array}{ccc}(\Delta_1) & \ldots & (\Delta_k) \\ \vdots & & \vdots \\ M_1 \Rightarrow m_1 & \ldots & M_k \Rightarrow m_k\end{array}}{M \Rightarrow m}(\text{possible side-condition})$$

where the sets of assertions $\Delta_1, \ldots \Delta_k$ are the *discharged assumptions*. Therefore, the evaluation of the expression M to the value m can be represented by the following derivation in N.D. style:

$$\Gamma = \{x_1 \Rightarrow n_1, \ldots, x_k \Rightarrow n_k\}$$
$$\begin{array}{c}\diagdown\!\!\diagup \\ \mathcal{D} \\ \diagup\!\!\diagdown \\ M \Rightarrow m\end{array} \qquad \text{written } \mathcal{D} : \Gamma \vdash M \Rightarrow m$$

where the hypotheses $\Gamma = \{x_1 \Rightarrow n_1, \ldots, x_k \Rightarrow n_k\}$ $(k \geq 0)$ can be interpreted as a set of variable bindings: the value of the variables involved in the evaluation of M. This derivation can be read as "in every environment which satisfies the assumptions in Γ, M is evaluated to m." This means that, given an environment ρ s.t. $\forall (x \Rightarrow m) \in \Gamma : \rho(x) = m$, there is a derivation of $\rho \vdash M \rightarrow m$ in the corresponding SOS proof system. An assumption about the value of a variable can be discharged when it is valid locally to a subcomputation. For instance, in the case of local declarations, in order to evaluate **let** $x = N$ **in** M, we can evaluate M assuming that the value of x is the same as that of N. This extra assumption is not necessary for evaluating **let** $x = N$ **in** M, so it can be discharged. The **let** rule is the rule in which the whole power of the ND style appears

$$\frac{\begin{array}{c}(x \Rightarrow n) \\ \vdots \\ N \Rightarrow n \quad M \Rightarrow m\end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}$$

whose reading is "if n is the value of N and, assuming the value of x is n then m is the value of M, then the value of **let** $x = N$ **in** M is m." (Unfortunately, the situation is not so simple, since this extra assumption can clash with a previous assumption on x which is valid globally. This will be discussed in detail in Sect.2.)

This truly N.D. approach has the benefit that all the rules which do not refer directly to identifiers appear in a simpler form than those in SOS style: no environment appears. For instance, the rule for the "+" function becomes

$$\frac{N_1 \Rightarrow n_1 \quad N_2 \Rightarrow n_2}{N_1 + N_2 \Rightarrow \mathit{plus}(n_1, n_2)}.$$

In this paper, we address the following question: what kind of programming languages can be treated conveniently using this formalism. We are interested in understanding to what extent we can reduce to assumptions the concepts of store, environment, binding and similar linear datatypes.

2 Analysis of the NOS style

In this section we try to convey briefly to the reader the main features of operational semantics in N.D. style. Recall that a N.D. style rule can be viewed as a concise description of a special kind of rule for deriving *sequents*, i.e. metapropositions of the form $\Gamma \vdash A$ ([8, 2]):

$$\frac{\begin{array}{c}(\Delta_1) \ldots (\Delta_k) \\ \vdots \qquad\quad \vdots \\ A_1 \quad \ldots \quad A_k\end{array}}{A} \text{ corresponds to } \frac{\Gamma, \Delta_1 \vdash A_1 \quad \ldots \quad \Gamma, \Delta_k \vdash A_k}{\Gamma \vdash A}$$

where $A, A_1, \ldots, A_k$ are propositions and $\Delta_1, \ldots, \Delta_k$ sets of propositions. Γ is any set of proposition. This rule means that, in order to prove that A is a consequence of a given Γ, we have to prove for $i = 1 \ldots k$ that A_i is a consequence of Γ, Δ_i. In other words, for proving each A_i we can use some local assumptions Δ_i, the global hypotheses Γ always remaining valid. This fact is at the core of the issues discussed in the following subsections.

2.1 The issue of local variables

Since the NOS rules are N.D. rules, if we have the following deduction $\mathcal{D}$:

$$\frac{\begin{array}{ccc}\Gamma, (\Delta_1) & & \Gamma, (\Delta_k) \\ \mathcal{D}_1 & \ldots & \mathcal{D}_k \\ M_1 \Rightarrow m_1 & & M_k \Rightarrow m_k\end{array}}{M \Rightarrow m}$$

all the bindings in Γ are available in evaluating M_i, for $i = 1 \ldots k$. As a consequence of this, the **let** rule showed in Sect.1 above, is incorrect because previous (global) assumptions on locally defined variables can be used during the subevaluation, e.g. as follows:

$$\frac{0 \Rightarrow 0 \quad \dfrac{1 \Rightarrow 1 \quad (x \Rightarrow 0)_{(1)}}{\textbf{let } x = 1 \textbf{ in } x \Rightarrow 0}}{\textbf{let } x = 0 \textbf{ in let } x = 1 \textbf{ in } x \Rightarrow 0}(1)$$

An attempt to overcome this problem could be that of using *higher-order syntax* à la Church. This technique originated with Church's idea to analyze $\forall x.P$ as $\forall(\lambda x.P)$ where $\forall$ has a higher order functionality: $\forall : (\mathit{Term} \to \mathit{Prop}) \to \mathit{Prop}$ [5]. It was further used by Martin-Löf [19] and thoroughly expanded in the Edinburgh Logical Framework [11]. This technique has been proved to work extremely well for pure functional languages (see [3] for a treatment of this in the context of λ-calculus and [9, 14] of more general functional languages). For example, the construct $\mathbf{lambda}\, x.M$ could be compiled to $\mathbf{lam}\ (\lambda x.M)$, where $\mathbf{lam} : (\mathit{Expr} \to \mathit{Expr}) \to \mathit{Expr}$. However, higher-order syntax cannot be used directly in the case of languages with imperative features. In fact, it easily yields semantic inconsistencies, since it treats identifiers as placeholders for expressions. This is correct in pure functional languages, but does not hold in imperative languages. For instance, the application of the λ-abstraction containing a command[2] $\mathbf{lambda}\ (\lambda x.[x := x + 1]x)$ to 0 would be reduced to the evaluation of $[0 := 0 + 1]0$, which is meaningless. Furthermore, there is no direct representation of loops like $\mathbf{while}\, b\, \mathbf{do}\, x := x + 1$. See [3] for more difficulties in handling Hoare's logic. Another problem is the absence of induction principles for encodings that employ higher-order syntax.

The difficulty of avoiding the capturing of local variables can be overcome by making explicit the textual substitution of variables in local evaluations [4]. This is similar to Gentzen's notion of *Eigenvariable* [8]. Recall the $\exists$-ELIM rule:

$$\dfrac{\exists x.A \quad \begin{array}{c}(A')\\ \vdots \\ B\end{array}}{B}$$

A' is obtained from A by replacing all the occurrences of x with x', where x' does not occur neither in any of A, x, B, nor in any assumption different from A'.

Simlarly, in evaluating $\mathbf{let}\ x = N\ \mathbf{in}\ M$, we have to replace all the occurrences of x in M with a new identifier never used before, say x', which will be bound to the value of n. The **let** rule is definable as follows:

$$\dfrac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ N \Rightarrow n \quad M' \Rightarrow m'\end{array}}{\mathbf{let}\ x = N\ \mathbf{in}\ M \Rightarrow m}\ E(x, M, m)$$

where $E(x, M, m)$ is a typographic abbreviation for the *Eigenvariable Condition*: $E(x, M, m) \equiv$ "M', m' are obtained from M, m respectively by replacing *all* the occurrences of x with x', which does not appears neither in x, M, m nor in any assumption different from $(x' \Rightarrow n)$".

This treatment of local variables correctly obeys the standard stack discipline of languages with static scoping: when we have to define a local variable, we allocate a new cell (represented by x', a new variable) where we store the local value (this is achieved by assuming $x' \Rightarrow n$). This allocation is active only during the evaluation of M (the derivation tree of $M' \Rightarrow m'$); then, the cell is disposed (x' does not occur in any other place).

[2] $[\cdot]\cdot : \mathit{Comm} \times \mathit{Expr} \to \mathit{Expr}$ applies commands to expressions; see Sect.3.1,4 and [4]

2.2 What informations can assumptions represent?

The structural rules implicit in N.D. systems of monotonicity of hypothesis have another immediate consequence. We can reduce to assumptions only informations which can be dealt with using a static scoping discipline. In particular, a side-effect assignment of pointers which induces variables aliasing (or sharing) is difficult to encode, since we would necessitate of a vector. In fact, we cannot retrace all the bindings which are involved on a set of shared variables whenever one of them changes its value.

However, in languages which do not allow sharing, assignments can be reduced to definitions of new variables. Therefore, we focus on this kind of languages. Namely, those whose semantics can be defined without using both environment and store. These comprise all purely functional languages, but also some interesting extensions of these which have genuinely imperative features. This is in fact our thesis: only languages whose denotational semantics is definable by using only the notion of environment can be conveniently handled using NOS. In the following we describe some of these languages.

3 The language $\mathcal{L}_P$

In this section, we examine a functional language extended with imperative features as assignments which give it an imperative flavor. Its semantics can been successfully described by using the NOS paradigm. We give its syntax, its NOS and denotational semantics (App.A, B, C) and we prove that the former is adequate w.r.t. the latter. Finally, we will discuss the relation between the NOS and a SOS description.

3.1 Syntax

$\mathcal{L}_P$ is an untyped λ-calculus extended by a set of structured commands. These commands are embedded into expressions using the "modal" operator $[\mathbf{on}\cdot\,\mathbf{do}\cdot]\cdot$. The expression $[\mathbf{on}\, x_1 = M_1; \ldots; x_k = M_k\ \mathbf{do}\, C]M$ can be read as:

"execute C in the environment formed only by the bindings $x_1 = M_1; \ldots; x_k = M_k$; use resulting values of these identifiers to extend the global environment in which M has to be evaluated, obtaining the value of the entire expression."

C cannot have access to "external" variables other than $x_1 \ldots x_k$, so all possible side effects are concerned with only these variables. Moreover, the entire **on-do** expression above does not have any side effect: all environment changes due to C's execution are local to M.

$\mathcal{L}_P$ allows us to declare and use procedures. For the sake of simplicity, but w.l.o.g., these procedures will take exactly two arguments. The first argument is passed by value/result, the second by value [21]. Furthermore, the body of a procedure cannot access global variables, but only its formal parameters (and locally defined identifiers, of course). This means that when $P(x, M)$ is executed within the scope of the declaration $\mathbf{proc}\, P(y, z) = C\ \mathbf{in}\, D$, C is executed in the

environment formed by only two bindings: $\{y \Rightarrow n, z \Rightarrow m\}$, where n, m are the values of x, M respectively. After C's execution, the new value of y is copied back into x. So, $P(x, M)$ can effect only x.

The restriction on global access forbids sharing of identifiers, so there is no need for a store. This does not drastically reduce the expressiveness of the imperative language. Donahue has shown that in this case, the call-by-value/result is a "good" simulation of the usual call-by-reference [7].

In [4] a different definition of procedure is given. There, procedures parameters are passed only by value, but procedures can have access to global variables. However, there is a problem with this approach, since the N.D. treatment of procedures does not immediately lend itself to support side effects on global variables. That approach does not work; for instance, the expression $[\mathbf{on}\ x = 0\ \mathbf{do\ proc}\ P(z) = (x := z)\ \mathbf{in}\ x := 1; P(nil)]x$ would be evaluated to 1 instead of *nil*. This is due to the fact that the assignment made by $P(nil)$ on the global variable x is local to the environment of the procedure itself. In fact, executions of such procedures leave the global environment unchanged.

3.2 Natural Operational Semantics

The complete NOS formal system for $\mathcal{L}_P$ consists of 67 rules; see App.B.1.

In order to deal with λ-closures, command checking and execution, procedure checking and bookkeeping, we need to introduce some new constructors besides those of Sect.3.1 and new predicates besides $\Rightarrow$. As in [4], the use of these new constructors is reserved: a programmer cannot directly utilize these constructors to write down a program. Below we list the constructors and predicates, and we briefly describe the most important ones. For their formal meaning, see Theor.3.

Constructor	Functionality
$[\_/\_]\_$	$: Expr \times Id \times Expr \rightarrow Expr$
$\_ \cdot \_$	$: Expr \times Expr \rightarrow Expr$
$[\_\|\_]\_$	$: Declarations \times Commands \times Expr \rightarrow Expr$
$[\_/\_]_c\_$	$: Commands \times Id \times Commands \rightarrow Commands$
lambda	$: Id \times Id \times Commands \rightarrow Procedures$
$[\_/\_]_{pe}\_$	$: Procedures \times ProcId \times Expr \rightarrow Expr$
$[\_/\_]_{pc}\_$	$: Procedures \times ProcId \times Commands \rightarrow Commands$

Judgment	Type	Judgment	Type
$\Rightarrow$	$\subset Expr \times Expr$	$\Rightarrow_p$	$\subset ProcId \times Procedures$
$value$	$\subset Expr$	$free_e$	$\subset Expr \times IdSet$
$closed$	$\subset Expr$	$free_c$	$\subset Commands \times IdSet$
$closed_p$	$\subset ProcId$	$\triangleright$	$\subset Declarations \times IdSet$

where *Procedures* is a new syntactic class defined as $q ::= \mathbf{lambda}\, x, y.C$, and *IdSet* is the subset of *Expr* defined as $l ::= nil \mid x \mid l_1 :: l_2$

The intuitive meaning of $[n/x]M$ is "the expression obtained from M by replacing all free occurrences of x with n." Just as for the **let** discussed in Sect.2,

in order to evaluate $[n/x]M$ we have to evaluate M under the assumption that the value of x is n, and hence any previous assumption on x must be ignored. This is implemented by the substitution rule, no.1, which is very similar to the **let** rule of Sect.2. This rule is the core of the evaluation system. Many other evaluation rules, e.g. the one for **let**, are reduced to the substitution evaluation (rule no.3). In NOS, to each sort of identifiers and substitution operators (e.g. *Id* and $[\_/\_]\_$, *ProcId* and $[\_/\_]_{pe}\_$, *Id* and $[\_/\_]_c\_$, etc.) there corresponds a specific substitution rule, similar in shape to rule no.1. In fact, this mechanism is used whenever one has to deal with standard static scoping. One can even think of these rules as a polymorphic variant of the same set of rules. Of course, minor adjustments have to be accounted for (rules no.24, 33, 29) (see [4, 15]).

The operator $[\_/\_]\_$ is also used to record local environments in values of **lambda**-abstractions, i.e. the *closures*. An expression like $\mathbf{lambda}\, x.M$ is evaluated into $[n_1/x_1]\ldots[n_k/x_k]\mathbf{lambda}\, x.M$, where $x_1, \ldots, x_k$ are all the free identifiers of M but x, and $n_1, \ldots, n_k$ are their respective values. The construction of closures is performed by rules no.4 and no.5; its application by rules no.7, no.8

The intuitive meaning of $[R|C]M$ is the same as of $[\mathbf{on}\ R\ \mathbf{do}\ C]M$. This expression is introduced in order to apply the declaration R until it is empty (rule no.16); then, the command C is executed.

The judgment *value* encodes the assertion that an expression is a value, and so it cannot be further reduced nor its meaning is affected by a substitution.

The judgments *closed*, $closed_p$ are used during closure construction, in order to determine the bindings that we have to record (rules no.4, no.5, no.31). Informally, we can derive *closed* M if and only if M has no free variables. These judgments belong to static semantics: their rules do not use evaluation rules.

The judgments *free*, $free_c$, $\rhd$ are used to check that command expressions $[\mathbf{on}\, D\ \mathbf{do}\, C]M$ and procedures do not access global variable. Informally, we can infer $\Gamma \vdash$ *free* $M\ l$ if and only if all the free variables of M appear at the leaves of the tree l (see Theor.3). On the other hand, $\rhd$ collects the variables defined by a declaration D into a set (represented by a tree of identifiers).

The judgment $\Rightarrow_p$ is used for bookkeeping the bindings between procedure identifiers and procedural abstraction (see rules no.28, no.30).

3.3 Denotational Semantics

In appendix C.1 we sketch the denotational semantics for the language $\mathcal{L}_P$. Domains are introduced to represent all the entities we have defined. This semantics is self-explanatory. We follow the usual syntax ([23]); $\underline{\lambda}$ denotes the *strict abstraction*: for each meta-expression M with free variable x on pointed domain D, $(\underline{\lambda}x.M)\bot = \bot$. Furthermore, $\underline{\underline{\lambda}}$ is the *double-strict abstraction*: for each meta-expression $M \neq \bot$ with free variable x on pointed domain D with both $\bot$ and $\top$, $(\underline{\underline{\lambda}}x.M)\bot = \bot, (\underline{\underline{\lambda}}x.M)\top = \top$.

Moreover we use the standard domains without giving their definition. The domains used are *Unit* (the set composed by only one point), $\mathbb{T}$ (the boolean set composed by two points, *true* and *false*), $\mathbb{N}$ (the set of natural numbers).

3.4 Adequacy

In this section we will show that the NOS description of $\mathcal{L}_P$ appearing in appendix B.1 is adequate w.r.t. the denotational semantics; that is, we will give soundness and completeness results of one semantics w.r.t. the other. Due to lack of space, we will only sketch the proofs; for further details see [15].

Definition 1. A set of formulæ Γ is a *canonical hypothesis* if

- it contains only formulæ like "$x \Rightarrow n, P \Rightarrow_p q, \mathit{closed}(x), \mathit{closed}_p(P)$";
- if $x \Rightarrow n, x \Rightarrow m \in \Gamma$ then m and n are syntactically the same expression;
- if $P \Rightarrow_p q, P \Rightarrow_p q' \in \Gamma$ then q and q' are syntactically the same procedure;

where $x \in \mathit{Id}, P \in \mathit{ProcId}$ and $m, n \in \mathit{Expr}, q, q' \in \mathit{Procedures}$.

In the rest of section, Γ will denote a generic canonical hypothesis. Let G be a formula; with $\Gamma \vdash G$ we denote the N.D. derivation of G, whose undischarged assumptions are in Γ.

Definition 2. Let $M \in \mathit{Expr}, R \in \mathit{Declarations}, \Gamma$ a canonical hypothesis; then
1. the set of *free identifiers of* M is denoted by $\mathrm{FV}(M) \subset \mathit{Id} \cup \mathit{ProcId}$. FV is naturally extended to *Commands*, bearing in mind that $\mathrm{FV}(x := M) = \mathrm{FV}(M)$;
2. the set of *variables defined by* R is $\mathrm{DV}(M) \subset \mathit{Id}$, defined as $\mathrm{DV}(x_1 = M_1; \ldots; x_k = M_k) = \{x_1, \ldots, x_k\}$;
3. the set of Γ-*closed identifiers* $\mathrm{C}(\Gamma)$ is $\mathrm{C}(\Gamma) \stackrel{\mathrm{def}}{=} \{x \in \mathit{Id} \mid \mathit{closed}(x) \in \Gamma\} \cup \{P \in \mathit{ProcId} \mid \mathit{closed}_p(P) \in \Gamma\}$;
4. the *closure* of Γ is $\overline{\Gamma} = \Gamma \cup \{\mathit{closed}(x) \mid (y \Rightarrow n) \in \Gamma, x \in \mathrm{FV}(n)\} \cup \{\mathit{closed}_p(P) \mid (y \Rightarrow n) \in \Gamma, P \in \mathrm{FV}(n)\}$;
5. Γ is a *well-formed hypothesis, wfh,* if $\Gamma = \overline{\Gamma}$.

Theorem 3. $\forall \Gamma, \forall M, m \in \mathit{Expr} \forall C \in \mathit{Commands}, \forall l \in \mathit{IdSet}$:
1. $\Gamma \vdash \mathit{closed}\ M \iff \mathrm{FV}(M) \subseteq \mathrm{C}(\Gamma)$
2. $\Gamma \vdash \mathit{free}\ m\ l \iff \mathrm{FV}(m) \cap \mathit{Id} \subseteq \mathit{leaves}(l) \wedge \mathrm{FV}(m) \cap \mathit{ProcId} \subseteq \mathrm{C}(\Gamma)$
   $\Gamma \vdash \mathit{free}\ C\ l \iff \mathrm{FV}(C) \cap \mathit{Id} \subseteq \mathit{leaves}(l) \wedge \mathrm{FV}(C) \cap \mathit{ProcId} \subseteq \mathrm{C}(\Gamma)$
   where $\mathit{leaves}(\mathit{nil}) = \emptyset, \mathit{leaves}(x) = \{x\}, \mathit{leaves}(l_1 :: l_2) = \mathit{leaves}(l_1) \cup \mathit{leaves}(l_2)$
3. $\Gamma \vdash \mathit{value}\ m \implies \overline{\Gamma} \vdash \mathit{closed}\ m$

Proof. By induction on the derivations and on the syntax of M, m, C. □

Note that not all closed expressions are values; e.g., ((**lambda** $x.x$) 0).

Definition 4. Let $I \subseteq \mathit{Id} \cup \mathit{ProcId}$ and let $\rho, \rho' \in \mathbb{E}$. We say that ρ and ρ' *agree on* I ($\rho \equiv_I \rho'$) if $\forall x \in I \cap \mathit{Id} : (\mathit{access}\ [\![x]\!]\ \rho = \mathit{access}\ [\![x]\!]\ \rho')$ and $\forall P \in I \cap \mathit{ProcId} : (\mathit{procaccess}\ [\![P]\!]\ \rho = \mathit{procaccess}\ [\![P]\!]\ \rho')$.

Note that all ρ, ρ' agree on the empty set, that is $\forall \rho, \rho' \in \mathbb{E} : \rho \equiv_\emptyset \rho'$.

Theorem 5. $\forall m \in \mathit{Expr}, \forall R \in \mathit{Declarations}, \forall \rho, \rho' \in \mathbb{E}$:
1. $\rho \equiv_{\mathrm{FV}(m)} \rho' \Rightarrow \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$
2. $\rho \equiv_{\mathrm{FV}(C)} \rho' \Rightarrow \mathcal{C}[\![C]\!]\rho = \mathcal{C}[\![C]\!]\rho'$
3. $\rho \equiv_{\mathrm{FV}(R)} \rho' \Rightarrow \mathcal{D}[\![R]\!]\rho = \mathcal{D}[\![R]\!]\rho'$

Proof. By simultaneous induction on the syntax of m, C, R. □

Theorem 6. $\forall \Gamma, \forall \rho, \rho' \in \mathbb{E}, \forall m \in \mathit{Expr}, \forall l \in \mathit{IdSet}$:
1. $\rho \equiv_{\mathrm{C}(\Gamma)} \rho' \wedge \Gamma \vdash \mathit{closed}\ m \Longrightarrow \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$
2. $\rho \equiv_{\mathit{leaves}(l)} \rho' \wedge \Gamma \vdash \mathit{free}\ C\ l \Longrightarrow \mathcal{C}[\![C]\!]\rho = \mathcal{C}[\![C]\!]\rho'$
3. $\rho \equiv_{\mathrm{C}(\Gamma)} \rho' \wedge \Gamma \vdash \mathit{value}\ m \Longrightarrow \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$

Proof. Follows from Theor.3, Theor.5. □

Corollary 7. $\Gamma \vdash \mathit{value}\ m \wedge \mathrm{C}(\Gamma) = \emptyset \Longrightarrow \forall \rho, \rho' \in \mathbb{E} : \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$

Definition 8. We say that $\rho \in \mathbb{E}$ *satisfies* Γ ($\rho \models \Gamma$) if $\forall (x \Rightarrow n) \in \Gamma$: $\mathit{access}\ [\![x]\!]\ \rho = \mathcal{E}[\![n]\!]\rho$, and $\forall (P \Rightarrow q) \in \Gamma : \mathit{procaccess}\ [\![x]\!]\ \rho = \mathcal{Q}[\![q]\!]\rho$.

This is another place where the conciseness of the N.D. formalism comes into play. The domain of environments satisfying a given Γ can be much larger than the set of variables which occur on the left of assumptions in Γ.

Theorem 9. $\forall M, m, \forall \Gamma\ \mathit{wfh}, \forall \rho : \rho \models \Gamma \wedge \Gamma \vdash M \Rightarrow m \implies \mathcal{E}[\![M]\!]\rho = \mathcal{E}[\![m]\!]\rho$

Proof. By induction on the structure of derivation, using the previous results.□

Corollary 10 Soundness of NOS wrt DS. $\forall M, m \in \mathit{Expr} :\vdash M \Rightarrow m \Longrightarrow \mathcal{E}[\![M]\!] = \mathcal{E}[\![m]\!]$

Proof. Put $\Gamma = \emptyset$ in Theor.9, and notice that $\emptyset$ is a wfh and $\forall \rho \in \mathbb{E} : \rho \models \emptyset$. □

A completeness result is something like an "inverse" of Corollary 10. However, a literal converse of Corollary 10 cannot hold: for $M = m = (\mathbf{lambda}\ x.x0)$ it is $\mathcal{E}[\![M]\!] = \mathcal{E}[\![m]\!]$ but of course $\not\vdash M \Rightarrow m$. In fact, only some expressions can appear as values (see Theor.3). We need a new definition:

Definition 11. Let $M \in \mathit{Expr}$. An hypothesis Γ is *suitable for* M (M-suit(Γ)), if $\forall x, P \in \mathrm{FV}(M) \exists (x \Rightarrow n), (P \Rightarrow_p q) \in \Gamma$ such that $\mathrm{FV}(n), \mathrm{FV}(q) \subseteq \mathrm{C}(\Gamma)$.

A hypothesis is suitable for M if it contains enough bindings to evaluate M.

Theorem 12 Completeness of NOS wrt DS. $\forall M \in \mathit{Expr}, \forall \Gamma\ \mathit{wfh}, \forall \rho \in \mathbb{E}$: *if* $\mathcal{E}[\![M]\!]\rho \neq \bot, \top$, M-suit(Γ) *and* $\rho \models \Gamma$, *then* $\exists m \in \mathit{Expr} : \Gamma \vdash M \Rightarrow m$.

Proof. The difficulty in the proof is this: given an expression M whose meaning, in a given environment, is a proper point of $\mathbb{V}$, and a suitable hypothesis Γ, we have to build up a deduction $\Gamma \vdash M \Rightarrow m$, for some m.[3] This cannot be done by induction on the syntactic structure of M, since our language is higher order; in fact, the evaluation of M can use M itself, and not only its subterms (see e.g. rule no.21). Nevertheless, the theorem can be proved by using the technique of *inclusive predicates*, developed by Milner and Plotkin [22]. □

[3] By Theor.9, this m has the same meaning as M

3.5 Adequacy w.r.t. the Structural Operational Semantics

In the previous subsection we have proved the adequacy of the NOS specification of $\mathcal{L}_P$ w.r.t. the denotational semantics. Actually, the same adequacy can be proved w.r.t. a Structural Operational Semantics (*à la Plotkin*, [21]). One can define a complete "input-output" SOS system for $\mathcal{L}_P$, that is a system for deriving two kinds of judgments: *evaluation of expressions*, $\rho \vdash_{SOS} M \to m$, and *execution of commands*, $\rho \vdash_{SOS} C \to \rho'$ where ρ, ρ' are finite environments, i.e. they are defined on a finite number of identifiers, and m is a value. In such a SOS system, there is no problem in handling substitutions, since we merely update the environment function in the subderivation:

$$\frac{\rho[x \mapsto n] \vdash_{SOS} M \to m}{\rho \vdash_{SOS} [n/x]M \to m}$$

Of course, this is not a linearized N.D. style system since we may delete a previous binding on x from the environment. These systems are equivalent, that is, $\forall \rho$ finite environment, $\forall \Gamma, \forall M, m \in \mathit{Expr}$:
1. if $\forall (x \Rightarrow n) \in \Gamma : \rho(x) = n$ and $\Gamma \vdash M \Rightarrow m$, then $\rho \vdash_{SOS} M \to m$
2. if $\rho \vdash_{SOS} M \to m$ and $\forall x \in \mathrm{FV}(M) : (x \Rightarrow \rho(x)) \in \Gamma$, then $\Gamma \vdash M \Rightarrow m$.
This is provable using techniques similar to those of previous subsection. Moreover, the completeness result (2) does not require the technique of inclusive predicates, but only a simpler structural induction on the derivation $\rho \vdash_{SOS} M \to m$.

4 Some remarks about language design

$\mathcal{L}_P$ is quite different from the language considered in [4]. There are several reasons for these changes. In some cases these are motivated by the desire to have a natural soundness result (see section 3.1 for remarks concerning procedures).

In our language, commands are embedded into expressions by the **on-do** construct. A simpler formalism for applying directly commands to expressions is used in [4], i.e. the "modal" operator $[\ ] : \mathit{Commands} \times \mathit{Expr} \to \mathit{Expr}$, so that $[C]M$ is an expression if $M \in \mathit{Expr}, C \in \mathit{Commands}$. Informally, the value of $[C]M$ is the value of M after the execution of C; C can affect any variable which is defined before its execution. Furthermore, as all expressions, $[C]M$ has no side-effects, that is evaluating $[C]M$ does not change the global environment any more than evaluating 0 or *nil*. C affects only the local environment which is used to evaluate M, but its side effects are not "filtered" by a declaration of accessible variables. In order to appreciate the difference in notation between the two approaches compare the following semantically equivalent expressions:

in the system of [4]: $\mathbf{let}\ x = 0\ \mathbf{in}\ [x := \mathit{nil}]x$; in $\mathcal{L}_P$: $[\mathbf{on}\ x = 0\ \mathbf{do}\ x := \mathit{nil}]x$

At first it seems that the latter is more complex and nothing has been gained. But the former expression might lead us to think that we can define functions with local state variables and more interesting expressions objects, but this is not the case! For instance, we can try to model a bank account defining a function `withdraw` which takes the amount to be withdrawn (an example taken from [1]):

```
let bal = 100; withdraw = lambda a.[bal:=bal-a]bal in
   let remaining = (withdraw 50) in
      (withdraw 30)
```

The system in [4] will evaluate it to 70 instead 20: the first withdraw has no effect. The reason is that in the closure of `withdraw`, `bal` is bound to 100, and this binding is reapplied to the local environment whenever `withdraw` is applied; this "reinitializes" `bal` to 100 each time (see rules no.5, no.6, no.8). Therefore, an application of `withdraw` cannot affect any following application.

Thus, [4]'s system may lead to misunderstanding the meaning of some expressions. We decide to avoid this by writing explicitly the variables which a command can affect, and making explicit that such variables are always reinitialized whenever the command is executed. By writing $[\mathbf{on}\, x_1 = M_1; \ldots; x_k = M_k\, \mathbf{do} C]M$ we immediately know that, *before* C is executed, the "interface variables" $x_1 \ldots x_k$ are initialized. Therefore, an obscure program, like the `withdraw` one, cannot be written in $\mathcal{L}_P$:the `withdraw` function should be declared as

```
let withdraw = lambda a.[on bal = 100 do bal:=bal-a]bal in ...
```

and hence its meaning is clearer. This aspect is however a major problem: neither in [4] system nor in $\mathcal{L}_P$ the `withdraw` function with the intended meaning of [1] can be written. We'll elaborate on this in Sect.7.

5 Some extensions of $\mathcal{L}_P$

In this section we briefly describe some further extensions of $\mathcal{L}_P$ concerning complex declarations, structures and imperative modules. Their semantics can be expressed without stores because there is no variable sharing. See appendix A, B and C for their syntax, NOS and DS respectively. We deal with each extension by itself, by simply adding new rules to the formal system without altering the previous ones. This illustrates modularity of NOS which allows us to add new rules for new constructs without changing the previous ones. For each extension, one can prove adequacy of NOS w.r.t. the denotational semantics ([15]), just by discussing only the new cases due to the extra rules.

5.1 Complex Declarations

$\mathcal{L}_D$ is obtained from $\mathcal{L}_P$ by adding expressions of the form $\mathbf{let} R\, \mathbf{in} M$ where R is a complex declaration like in Standard ML ([16]). In spite of the syntactic simplicity of these extensions, it appears to be unavoidable to define an entire evaluation system for declarations (rules no.72–88). The value of complex declarations are finite sets of bindings, represented by expressions called *syntactic environments*; they are trees whose leaves are of the form $x \mapsto n$ where $\mapsto$: $Id \times Expr \to Expr$ is a new local constructor. We need to introduce furthermore several constructors and a judgment for applying such syntactic environments to expressions and declarations ($\{\_\}\_, \{\_\}_d\_$) and for inferring expression closures ($\langle\_\rangle\_, \gg$). Informally,

one can derive $\Gamma \vdash R \gg I$ iff all expressions contained in R are closed in Γ and I is the set of identifiers defined by R. On the other hand, $\Gamma \vdash$ *closed* $\langle I \rangle M$ iff all free variables in M but the ones in I are *closed* in Γ. Once the rules will be laid down, these fact will be formally provable. Using this set of rules, we can define precisely when a complex **let** is closed without using any evaluation, since *closed* is a property belonging to static semantics. An adequacy theorem similar to Theor.3 can be proved for the system given in Sect.B.2. In [4] there is a simpler approach; it uses the complex declaration evaluation in order to determine the set of defined identifiers. This approach is not complete: there are closed expression whose *closed* property cannot be inferred in [4]'s system (e.g. **let** $o = (\textbf{lambda}\, x.xx); z = (oo)$ **in** z).

5.2 Structures and signatures

$\mathcal{L}_{M_F}$ extends $\mathcal{L}_P$ by adding a module system like that of Standard ML ([16]), where a module is "an environment turned into a manipulable object". Like SML, a module (here called *structure*) has a *signature*, and we can do *signature matching* in order to "cast" structures. However, there are some differences between SML and $\mathcal{L}_{M_F}$. First, in $\mathcal{L}_{M_F}$ structures and signatures are indeed expression. Therefore, they may be associated to identifiers with simple **let**s, without using special constructs. These **let**s can appear anywhere in expressions, not only at top level. Structures and signatures can be manipulated by common functions; however, there are not *functors* since the sharing specification is not implemented. The NOS should be self-explanatory.

5.3 Imperative modules (Abstract Data Types)

The extension $\mathcal{L}_{M_I}$ introduces modules *à la Morris* ([18]). In this formulation, a module is very close to an Abstract Data Type: it contains
1. a set of local variables, recording the *state* of the module; they are not accessible from outside the module;
2. some code for the initialization of the local variables above;
3. a set of procedures and functions which operate on these local variables and are the only part accessible from outside the module (the *interface*).

From outside a module we can only evaluate its functions, which do not produce side-effects, and execute its procedures, which can modify the state of the module (the value of local variables). In order to illustrate the idea, but w.l.o.g., we discuss only modules with exactly one local variable, one procedure with one argument (passed by value) and one function.

As for the previous languages, we do not need a representation of the store in defining the semantics of this kind of module ([7]). The rules for the specification of the imperative modules are certainly the most complex of those discussed in this paper. They are based on the principle of distributing as much as possible under the form of hypothetical assumption in deductions. In a module there are three informations: the state, the procedure and the function. Actually, only the state is subject to changes upon execution of the module procedure. We split

these three informations and record them using three different judgments (see rule no.114). The predicates of these assumptions are the following:
$\Rightarrow_m \subset ModId \times (Expr \times ModId)$ $\Rightarrow_{mp} \subset (ModId \times ProcId) \times \mathbb{Q}$
$\Rightarrow_{mf} \subset (ModId \times Id) \times Expr$
We use a lot of syntactic sugar; for instance, we write $T.P \Rightarrow_{mp} \boldsymbol{\lambda} x, y.C$ instead of $\Rightarrow_{mp} ((T, P), \mathbf{lambda}\, x, y.C)$.

When the state of a module changes (by executing its procedure), we have to substitute only the assumption involving $\Rightarrow_m$; the other two remain the same. Thus, while the procedure and the function are left associated to the original module identifier, the state becomes associated to a new *ModId*, and this substitution affects a part of the declaration to be evaluated (see rules no.116 no.115). The link between the new state and the procedures is maintained by the module identifier which appears on right of $\Rightarrow_m$ assumption: it is merely copied from the old assumption into the new one (rule no.116).

When a module procedure has to be executed ($T.P(M)$), first we look for the state of the module T, by requiring $T \Rightarrow_m (p, T')$. Here we find the original module identifier, T'. The invoked procedure is then associated to this identifier in the assumption $T'.P \Rightarrow_{mp} \mathbf{lambda}\, x, y.C$. After having bound x and y respectively to module variable value (p) and actual parameter (m), we execute C and get back the new value of the state variable. Finally, we substitute T with the new module state.

Function evaluation is similar to procedure call, but simpler (rule no.117).

We can successfully implement the bank account examined in Sect.4 by using this kind of modules, e.g. as follows:

```
module account is
  bal = 100;
  proc withdraw(amount) = bal := bal - amount;
  func balance = bal
in ...
```

Now we can withdraw an amount A by executing `account.withdraw(`A`)`, and know how much money we have left by evaluating `account.balance`.

However, even this notion of module is too weak to adequately model "functions with local state" as are necessary, for instance, in realizing memoized functions. In fact, as soon as an instance of a module is packaged within a λ abstraction, its connection with its parent (definition) is severed.

6 Encoding NOS in the Edinburgh LF

From a logician's point of view, the Natural Operational Semantics of a language is just a formal logical system in Natural Deduction style. Therefore, it can be easily encoded in interactive proof-checkers based on type-checking of typed λ-calculus, such as the Edinburgh Logical Framework (LF, [11]). This was actually one of the main motivations for introducing and investigating the systems of this

paper. A first outline about this can be found in [4]. In [15] a complete encoding of the NOS of the **while** subset of $\mathcal{L}_P$ appears.

The LF encoding of NOS has several significant consequences. When we encode the operational semantics in LF, we have to discuss details that are normally left out or too often taken for granted or even "swept under the rug". For instance, the complex side-condition of rule no.1 requires that "x is a new identifier", but we do not give a formal definition of this. When we encode NOS in LF, this condition has to be expressed formally and unambiguously. This is achieved by introducing two auxiliary judgments $in, notin : (Term\ Id) \to \prod_{S:Sorts}(Term\ S) \to Type$ where $Sorts : Type$ is the type of syntactic classes and $Term : Sorts \to Type$. The intuitive meaning of $(in\ x\ S\ p)$ is "the identifier x appears in the phrase p which belongs to the syntactic class S"; dually for *notin*. The rules for these judgments are given on the syntax of phrases in the obvious way—see [4, 15]. Thus, the complete substitution rule no.1 is as follows, where $Id, Expr : Sorts$:

$$M, m : Id \to Expr \quad \frac{value\ n \qquad \forall w : Id \begin{cases} (notin\ x\ Id\ w) \\ \vdots \\ notin\ x\ Expr\ (M\ w) \end{cases} \qquad \forall x' : Id \begin{cases} \left(\begin{array}{c} \forall w : Id \dfrac{in\ w\ Id\ x}{notin\ w\ Id\ x'} \\ \forall w : Id \dfrac{in\ w\ Expr\ (M\ x)}{notin\ w\ Id\ x'} \\ \forall w : Id \dfrac{in\ w\ Expr\ (m\ x)}{notin\ w\ Id\ x'} \\ x' \Rightarrow n \end{array}\right) \\ \vdots \\ (M\ x') \Rightarrow (m\ x') \end{cases}}{[n/x](M\ x) \Rightarrow (m\ x)}$$

The middle subderivation requires that x does not occur in the expression context M. Evaluation is performed in the right-hand subderivation; here, x is replaced by x' assuming x' does not occur in any of $x, (M\ x), (m\ x)$. This is achieved by the three discharged rules about *in*, *notin*. The full power of LF is exploited: rules are treated just as any other judgment.

We can use this encoding with *proof editors* based on LF, such as LEGO ([13]), and *logic programming languages* based on LF, such as Elf ([20]). LEGO can be successfully used to develop derivations (= computation traces) and to verify properties about the semantics themselves, e.g. equivalence between constructs. During the phase of operational semantics developing, we can try our rules and look for inconsistencies. Thus, we have immediately a powerful tool for semantic development and consistency checking.

On the other hand, logic programming languages such as Elf can be used to get an interpreter prototype for free: immediately after we have encoded in Elf the LF representation, we can ask queries like `?- True(eval` M `V).` where M is (the encoding of) an expression. In resolving this goal, Elf instantiates `V` to M's value, and develops a term which represent the deduction $\vdash M \Rightarrow V$, that is the computation trace of the evaluation of M.

In these systems we can prove several meta-results about semantics. In [15] the equivalence between two different NOS of the same **while**-language is developed. One of these semantics is "natural", clear but inefficient. The rules for

the **while** execution are the following:

$$\frac{M \Rightarrow \mathit{true} \quad [C]([\mathbf{while}\ M\ \mathbf{do}\ C]N) \Rightarrow n}{[\mathbf{while}\ M\ \mathbf{do}\ C]N \Rightarrow n} \qquad \frac{M \Rightarrow \mathit{false} \quad N \Rightarrow n}{[\mathbf{while}\ M\ \mathbf{do}\ C]N \Rightarrow n}$$

This semantics needs to backtrack and to re-evaluate the test expression if it does not match the required value in the assumption. The second semantics overcomes this drawback by introducing an auxiliary judgment, *dw* ("do while"):

$$\frac{M \Rightarrow_a m \quad \mathit{dw}\ m\ M\ C\ N\ n}{[\mathbf{while}\ M\ \mathbf{do}\ C]N \Rightarrow_a n} \qquad \frac{[C]([\mathbf{while}\ M\ \mathbf{do}\ C]N) \Rightarrow_a n}{\mathit{dw}\ \mathit{true}\ M\ C\ N\ n} \qquad \frac{N \Rightarrow_a n}{\mathit{dw}\ \mathit{false}\ M\ C\ N\ n}$$

Here, backtracking and double evaluation are not needed any more.

Adopting a technique used by Michaylov and Pfenning for functional languages ([14]), we can prove the equivalence between these two semantics by encoding in Elf a judgment, `naeq` : $\prod_{M,m \in Expr}(M \Rightarrow m) \rightarrow (M \Rightarrow_a m) \rightarrow$ Type. This judgment represents the `e`quivalence between "`n`atural" and "`a`lgorithmic" computation traces. Asking Elf about queries of the form `?- naeq` D D' where one of D, D' is instantiated to a derivation in one semantics, the system automatically gives us the equivalent derivation in the other semantics. In this way we have defined a bijection between the computation traces of the two semantics.

We can think the former semantic as the "theoretical" semantics of the language, and the latter as the real implementation. Thus, the formal equivalence proved in Elf between them can be seen as the backbone of a proof of compiler correctness. (For compiler correctness in LF see also [10]).

7 Concluding remarks

In this paper we have described the expressive power of the Natural Operational Semantics formalism. We have seen that this formalism handles successfully languages which do not allow variable aliasing, or sharing, We have shown some of these languages: functional languages extended with a restricted form of commands and procedures, blocks, complex declarations, modules *à la ML* (structures and signatures) and modules *à la Morris*.

This formalism improves abstractness and modularizability of Plotkin's Structural Operational Semantics and Kahn's Natural Semantics. Furthermore, such a operational description can be easily encoded in LF. Such encodings can be used within implementations of LF (LEGO and Elf), giving us powerful tools for developing language semantics formally, for checking correctness of translators and for proving semantic properties.

Unfortunately, so far we have not been able to give the semantics of a truly imperative language using this formalism. It seems that one cannot represent simultaneously both the store and the environment by means of assumptions. Without encoding a store we cannot describe usual imperative phenomena like side-effects with aliasing, argument passage of parameters by-reference and so on. Therefore, this formalism seems not general enough to deal with expressions

with side-effects, functions with local state variables or memoization, Pascal procedures (procedures with global variables and call-by-reference).

We think that exception handling can be added to $\mathcal{L}_{M_F}$ quite easily ([4]). The real lack of our languages w.r.t. ML is the absence of the store: ML is a store-based language. Therefore, in order to capture fully the semantics of ML (and encode it in LF) we have to find some representation of the store. This is a task remaining to be done. Ideally, we would like to extend the formalism as much as is needed to describe the semantics of an untyped λ-calculus extended by primitives for manipulating side-effects, like ML's `ref`, `!` and `:=` ([16]). The NOS of this language should be easily extended to that of ML.

A Syntax

A.1 Syntax of $\mathcal{L}_P$

Syntactic class *Id*
$x ::= i_0 \mid i_1 \mid i_2 \mid i_3 \mid \ldots$

Syntactic class *Expr*
$M ::= 0 \mid \mathit{succ} \mid \mathit{plus} \mid \mathit{true} \mid \mathit{false} \mid \leq$
$\mid \mathit{nil} \mid M :: N \mid \mathit{hd} \mid \mathit{tl}$
$\mid \textbf{lambda}\, x.M \mid MN$
$\mid \textbf{let}\, x = M\, \textbf{in}\, N$
$\mid \textbf{letrec}\, f(x) = M\, \textbf{in}\, N$
$\mid [\textbf{on}\, R\, \textbf{do}\, C]M$

Syntactic class *ProcId*
$P ::= p_1 \mid p_2 \mid p_3 \mid \ldots$

Syntactic class *Declarations*
$R ::= \langle\rangle \mid x = M; R$

Syntactic class *Commands*
$C ::= x := M \mid C; D \mid \textbf{nop}$
$\mid \textbf{if}\, M\, \textbf{then}\, C\, \textbf{else}\, D$
$\mid \textbf{beginnew}\, x = M;\ C\, \textbf{end}$
$\mid \textbf{proc}\, P(x, y) = C\, \textbf{in}\, D$
$\mid \textbf{while}\, M\, \textbf{do}\, C \mid P(x, M)$

A.2 Syntax of $\mathcal{L}_D$

Syntactic class *Expr*
$M ::= \ldots \mid \textbf{let}\, R\, \textbf{in}\, M$

Syntactic class *Declarations*
$R ::= \ldots \mid R; S \mid R\, \textbf{and}\, S$

A.3 Syntax of $\mathcal{L}_{M_F}$

Syntactic class *LongId*
$u ::= x \mid u.x$

Syntactic class *Commands*
$C ::= \ldots \mid u := M$

Syntactic class *Expr*
$M ::= \ldots \mid \textbf{sig}\, x_1 \ldots x_k\, \textbf{end}$
$\mid \textbf{struct}\, x_1 = M_1; \ldots; x_k = M_k\, \textbf{end}$
$\mid M : N \mid \textbf{open}\, u\, \textbf{in}\, M$

A.4 Syntax of $\mathcal{L}_{M_I}$

Syntactic class *ModId*
$T ::= t_1 \mid t_2 \mid t_3 \mid \ldots$

Syntactic class *Expr*
$M ::= \ldots \mid T.f$

Syntactic class *Commands*
$C ::= \ldots \mid T.P(M)$
$\quad \mid \textbf{module}\, T\, \textbf{is}\, x = M;$
$\quad\quad \textbf{proc}\, P(y) = C; \textbf{func}\, f = N$
$\quad \textbf{in}\, D$

B Rules for the NOS

B.1 NOS of $\mathcal{L}_P$

Rules for judgment $\Rightarrow$ For the meaning of $E(\cdot,\cdot,\cdot)$, see Sect.2.1. For typographical reasons, **lambda** will be sometimes abbreviated with $\boldsymbol{\lambda}$ and $[\langle\rangle|C]N$ with $[C]N$.

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots\\ \mathit{value}\ n\ M' \Rightarrow m'\end{array}}{[n/x]M \Rightarrow m}\ E(x,M,m) \quad (1)$$

$$\frac{\mathit{value}\ m}{m \Rightarrow m} \quad (2)$$

$$\frac{N \Rightarrow n \quad [n/x]M \Rightarrow m}{\textbf{let}\, x = N\, \textbf{in}\, M \Rightarrow m} \quad (3)$$

$$\frac{\begin{array}{c}(\mathit{closed}\ x)\\ \vdots\\ \mathit{closed}\ M\end{array}}{\textbf{lambda}\, x.M \Rightarrow \textbf{lambda}\, x.M} \quad (4)$$

$$\frac{\begin{array}{c}(\mathit{closed}\ y)\\ \vdots\\ y \Rightarrow n\ \textbf{lambda}\, x.M \Rightarrow m\end{array}}{\textbf{lambda}\, x.M \Rightarrow [n/y]m} \quad (5)$$

$$\frac{M \Rightarrow m \quad N \Rightarrow n \quad m \cdot n \Rightarrow p}{MN \Rightarrow p} \quad (6)$$

$$\frac{[n/x]M \Rightarrow p}{(\textbf{lambda}\, x.M) \cdot n \Rightarrow p} \quad (7)$$

$$\frac{\mathit{value}\ n \quad [m'/x](m \cdot n) \Rightarrow p}{([m'/x]m) \cdot n \Rightarrow p} \quad (8)$$

$$\frac{\mathit{value}\ n}{(\mathit{plus} \cdot 0) \cdot n \Rightarrow n} \quad (9)$$

$$\frac{(\mathit{plus} \cdot m) \cdot n \Rightarrow p}{(\mathit{plus} \cdot (\mathit{succ} \cdot m)) \cdot n \Rightarrow \mathit{succ} \cdot p} \quad (10)$$

$$\frac{\begin{array}{l}\textbf{let}\, f = (\textbf{lambda}\, x.\textbf{letrec}\\ \quad f(x) = N\ \textbf{in}\ N)\ \textbf{in}\ M \Rightarrow p\end{array}}{\textbf{letrec}\, f(x) = N\, \textbf{in}\, M \Rightarrow p} \quad (11)$$

$$\frac{M \Rightarrow m \quad N \Rightarrow n}{M :: N \Rightarrow m :: n} \quad (12)$$

$$\frac{\mathit{value}\ m}{\mathit{hd} \cdot (m :: n) \Rightarrow m} \quad (13)$$

$$\frac{\mathit{value}\ n}{\mathit{tl} \cdot (m :: n) \Rightarrow n} \quad (14)$$

$$\frac{D \rhd I \quad \mathit{free}\ C\ I \quad [D|C]N \Rightarrow n}{[\textbf{on}\, D\, \textbf{do}\, C]N \Rightarrow n} \quad (15)$$

$$\frac{M \Rightarrow m \quad [m/x]([D|C]N) \Rightarrow n}{[x = M; D|C]N \Rightarrow n} \quad (16)$$

$$\frac{N \Rightarrow n \quad [n/x]M \Rightarrow m}{[x := N]M \Rightarrow m} \quad (17)$$

$$\frac{[C]([D]M) \Rightarrow m}{[C; D]M \Rightarrow m} \quad (18)$$

$$\frac{M \Rightarrow \mathit{true} \quad [C]N \Rightarrow n}{[\textbf{if}\, M\, \textbf{then}\, C\, \textbf{else}\, D]N \Rightarrow n} \quad (19)$$

$$\frac{M \Rightarrow \mathit{false} \quad [D]N \Rightarrow n}{[\textbf{if}\, M\, \textbf{then}\, C\, \textbf{else}\, D]N \Rightarrow n} \quad (20)$$

$$\frac{\begin{array}{c}M \Rightarrow \mathit{true}\\ [C]([\textbf{while}\, M\, \textbf{do}\, C]N) \Rightarrow n\end{array}}{[\textbf{while}\, M\, \textbf{do}\, C]N \Rightarrow n} \quad (21)$$

$$\frac{M \Rightarrow \mathit{false} \quad N \Rightarrow n}{[\textbf{while}\, M\, \textbf{do}\, C]N \Rightarrow n} \quad (22)$$

$$\frac{N \Rightarrow n \quad [[n/x]_c C]M \Rightarrow m}{[\textbf{beginnew}\, x = N;\ C\ \textbf{end}]M \Rightarrow m} \quad (23)$$

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots\\ value\ n\ [C']M \Rightarrow m'\end{array}}{[[n/x]_c C]M \Rightarrow m}\ E(x, C, m) \quad (24)$$

$$\frac{value\ n}{(\leq \cdot 0) \cdot n \Rightarrow true} \quad (25)$$

$$\frac{value\ n}{(\leq \cdot(succ \cdot n)) \cdot 0 \Rightarrow false} \quad (26)$$

$$\frac{(\leq \cdot n) \cdot m \Rightarrow p}{(\leq \cdot(succ \cdot n)) \cdot (succ \cdot m) \Rightarrow p} \quad (27)$$

$$\frac{[[\mathbf{lambda}\, x, y.C/P]_{pc} D]M \Rightarrow m}{[\mathbf{proc}\, P(x, y) = C\ \mathbf{in}\ D]M \Rightarrow m} \quad (28)$$

$$\frac{\begin{array}{c}(P' \Rightarrow_p \boldsymbol{\lambda} x, y.C)\\ \vdots\\ free_c\ C\ (x, y) \quad [D']M \Rightarrow m'\end{array}}{[[\boldsymbol{\lambda} x, y.C/P]_{pc} D]M \Rightarrow m} E(P, D, m) \quad (29)$$

$$\frac{P \Rightarrow \boldsymbol{\lambda} x, y.C \quad M \Rightarrow m \quad z \Rightarrow p \quad [p/x][m/y][C]x \Rightarrow v \quad [v/z]N \Rightarrow n}{[P(z, M)]N \Rightarrow n} \quad (30)$$

$$\frac{\begin{array}{c}(closed\ P)\\ \vdots\\ free\ C\ (x, y) \quad P \Rightarrow_p \boldsymbol{\lambda} x, y.C\ \boldsymbol{\lambda} z.M \Rightarrow m\end{array}}{\boldsymbol{\lambda} z.M \Rightarrow [\boldsymbol{\lambda} x, y.C/P]_{pe} m} \quad (31)$$

$$\frac{value\ n \quad [Q/P]_{pe}(m \cdot n) \Rightarrow p}{([Q/P]_{pe} m) \cdot n \Rightarrow p} \quad (32)$$

$$\frac{\begin{array}{c}(P' \Rightarrow_p Q)\\ \vdots\\ free_c\ C\ (x, y) \quad M' \Rightarrow m'\end{array}}{[\boldsymbol{\lambda} x, y.C/P]_{pe} M \Rightarrow m} E(P, M, m) \quad (33)$$

Rules for judgment *value*

$$\frac{}{value\ m}\ m \text{ is a constant} \quad (34)$$

$$\frac{M \Rightarrow m}{value\ m} \quad (35)$$

Rules for judgment $\triangleright$

$$\frac{}{\langle\rangle\ \triangleright\ nil} \quad (36)$$

$$\frac{D_l\ \triangleright I}{x = M; D_l\ \triangleright\ x :: I} \quad (37)$$

Rules for judgment *closed*

$$\frac{}{closed\ m}\ m \text{ is a constant} \quad (38)$$

$$\frac{closed\ M \quad closed\ N}{closed(MN)} \quad (39)$$

$$\frac{closed\ m \quad closed\ n}{closed(m \cdot n)} \quad (40)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots\\ closed\ N\ \ closed\ M\end{array}}{closed(\mathbf{let}\ x = N\ \mathbf{in}\ M)} \quad (41)$$

$$\frac{\begin{array}{cc}(closed\ f, closed\ x) & (closed\ f)\\ \vdots & \vdots\\ closed\ N & closed\ M\end{array}}{closed(\mathbf{letrec}\ f(x) = N\ \mathbf{in}\ M)} \quad (42)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots\\ closed\ M\end{array}}{closed(\mathbf{lambda}\, x.M)} \quad (43)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots\\ closed\ n\ \ closed\ M\end{array}}{closed([n/x]M)} \quad (44)$$

$$\frac{closed\ m \quad closed\ n}{closed(m :: n)} \quad (45)$$

$$\frac{D\ \triangleright I \quad free\ C\ I \quad closed\ [D|\mathbf{nop}]M}{closed\ [\mathbf{on}\ D\ \mathbf{do}\ C]M} \quad (46)$$

$$\frac{closed\ M}{closed\ [\langle\rangle|\mathbf{nop}]M} \quad (47)$$

$$\frac{\begin{array}{c}(\mathit{closed}\ x)\\ \vdots\\ \mathit{closed}\ N\quad \mathit{closed}\ [R|\mathbf{nop}]M\end{array}}{\mathit{closed}\ [x=N;R|\mathbf{nop}]M} \tag{48}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y)\quad \mathit{closed}\ M\end{array}}{\mathit{closed}\ [\mathbf{lambda}\,x,y.C/P]_{pe}M} \tag{49}$$

Rules for judgment *free*

$$\frac{}{\mathit{free}\ x\ (x,m)} \tag{50}$$

$$\frac{\mathit{free}\ x\ m}{\mathit{free}\ x\ (y,m)} \tag{51}$$

$$\frac{\mathit{free}\ M\ m\quad \mathit{free}\ N\ m}{\mathit{free}\ (M\ N)\ m} \tag{52}$$

$$\frac{\mathit{free}\ M\ m\quad \mathit{free}\ N\ x::m}{\mathit{free}\ (\mathbf{let}\,x=M\ \mathbf{in}\,N)\ m} \tag{53}$$

$$\frac{\mathit{free}\ M\ x::m}{\mathit{free}\ (\mathbf{lambda}\,x.M)\ m} \tag{54}$$

$$\frac{\mathit{free}\ M\ m\quad \mathit{free}\ N\ m}{\mathit{free}\ (M\cdot N)\ m} \tag{55}$$

$$\frac{\mathit{free}\ M\ m\quad \mathit{free}\ N\ m}{\mathit{free}\ (M::N)\ m} \tag{56}$$

$$\frac{\mathit{free}\ n\ m\quad \mathit{free}\ M\ x::m}{\mathit{free}\ ([n/x]M)\ m} \tag{57}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y,m)\qquad \mathit{free}\ M\ m\end{array}}{\mathit{free}\ ([\mathbf{lambda}\,x,y.C/P]M)\ m} \tag{58}$$

$$\frac{\mathit{free}\ C\ m\quad \mathit{free}\ M\ m}{\mathit{free}\ ([C]M)\ m} \tag{59}$$

$$\frac{\mathit{free}\ C\ m\quad \mathit{free}\ D\ m}{\mathit{free}\ (C;D)\ m} \tag{60}$$

$$\frac{\mathit{free}\ M\ m\quad \mathit{free}\ C\ m\quad \mathit{free}\ D\ m}{\mathit{free}\ (\mathbf{if}\ M\ \mathbf{then}\,C\ \mathbf{else}\,D)\ m} \tag{61}$$

$$\frac{\mathit{free}\ M\ m\quad \mathit{free}\ C\ m}{\mathit{free}\ (\mathbf{while}\,M\ \mathbf{do}\,C)\ m} \tag{62}$$

$$\frac{\mathit{free}\ M\ m\quad \mathit{free}\ C\ x::m}{\mathit{free}\ (\mathbf{beginnew}\,x=M;C\ \mathbf{end}\,)\ m} \tag{63}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y,m)\qquad \mathit{free}\ D\ m\end{array}}{\mathit{free}\ (\mathbf{proc}\,P(x,y)=C\ \mathbf{in}\,D)\ m} \tag{64}$$

$$\frac{\mathit{closed}_p(P)\quad \mathit{free}\ x\ m\quad \mathit{free}\ M\ m}{\mathit{free}\ (P(x,M))\ m} \tag{65}$$

$$\frac{\mathit{free}\ n\ m\quad \mathit{free}\ C\ x::m}{\mathit{free}\ ([n/x]C)\ m} \tag{66}$$

$$\frac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y,m)\qquad \mathit{free}\ D\ m\end{array}}{\mathit{free}\ ([\mathbf{lambda}\,x,y.C/P]D)\ m} \tag{67}$$

B.2 NOS of $\mathcal{L}_D$

Rules for judgment $\Rightarrow$

$$\frac{R\Rightarrow_d r\quad \{r\}M\Rightarrow m}{\mathbf{let}\,R\ \mathbf{in}\,M\Rightarrow m} \tag{68}$$

$$\frac{M\Rightarrow m}{\{nil\}M\Rightarrow m} \tag{69}$$

$$\frac{[n/x]M\Rightarrow m}{\{x\mapsto n\}M\Rightarrow m} \tag{70}$$

$$\frac{\{r\}(\{s\}M)\Rightarrow m}{\{r::s\}M\Rightarrow m} \tag{71}$$

Rules for judgment $\Rightarrow_d$

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots\\ value\ n\ R' \Rightarrow_d m\end{array}}{[n/x]_d R \Rightarrow_d m}\ E(x,n,R,m) \quad (72)$$

$$\frac{M \Rightarrow m}{x = M \Rightarrow_d x \mapsto m} \quad (73)$$

$$\frac{R \Rightarrow_d r \quad S \Rightarrow_d s}{R\ \mathbf{and}\ S \Rightarrow_d r :: s} \quad (74)$$

$$\frac{R \Rightarrow_d r \quad \{r\}_d S \Rightarrow_d s}{R; S \Rightarrow_d r :: s} \quad (75)$$

$$\frac{R \Rightarrow_d r}{\{nil\}_d R \Rightarrow_d r} \quad (76)$$

$$\frac{[n/x]_d R \Rightarrow_d r}{\{x \mapsto n\}_d R \Rightarrow_d r} \quad (77)$$

$$\frac{\{r\}_d(\{s\}_d R) \Rightarrow_d r}{\{r :: s\}_d R \Rightarrow_d r} \quad (78)$$

Rules for judgment *closed*

$$\frac{closed\ M}{closed\ \{nil\}M} \quad (79)$$

$$\frac{closed\ [n/x]M}{closed\ \{x \mapsto n\}M} \quad (80)$$

$$\frac{closed\ \{r\}\{s\}M}{closed\ \{r :: s\}M} \quad (81)$$

$$\frac{closed\ m}{closed\ x \mapsto m} \quad (82)$$

$$\frac{R \gg m \quad closed\ \langle m\rangle M}{closed(\mathbf{let}\ R\ \mathbf{in}\ M)} \quad (83)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots\\ closed\ M\end{array}}{closed\ \langle x\rangle M} \quad (84)$$

$$\frac{closed\ \langle m\rangle(\langle n\rangle M)}{closed\ \langle m :: n\rangle M} \quad (85)$$

Rules for judgment $\gg$

$$\frac{}{\langle\rangle \gg nil} \quad (86)$$

$$\frac{R \gg m \quad S \gg n}{R\ \mathbf{and}\ S \gg m :: n} \quad (87)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots\\ closed\ M \quad R \gg n\end{array}}{x = M; R \gg x :: n} \quad (88)$$

B.3 NOS of $\mathcal{L}_{M_F}$

Rules for judgment $\Rightarrow$

$$\frac{}{\mathbf{struct}\ \ \mathbf{end} \Rightarrow nil} \quad (89)$$

$$\frac{M \Rightarrow m \quad [m/x]\mathbf{struct}\ B_{str} \Rightarrow l}{\mathbf{struct}\ x = M\ B_{str} \Rightarrow (x \mapsto m, l)} \quad (90)$$

$$\frac{M \Rightarrow m \quad N \Rightarrow t \quad proj\ m\ (t)\ n}{M : N \Rightarrow n} \quad (91)$$

$$\frac{u \Rightarrow m \quad (x \mapsto p)\ in\ m}{u.x \Rightarrow p} \quad (92)$$

$$\frac{u \Rightarrow l \quad \{l\}M \Rightarrow m}{\mathbf{open}\ u\ \mathbf{in}\ M \Rightarrow m} \quad (93)$$

$$\frac{\begin{array}{c}M \Rightarrow m \quad u \Rightarrow l\\ upd\ l\ x\ m\ l' \quad [u := l']N \Rightarrow n\end{array}}{[u.x := M]N \Rightarrow n} \quad (94)$$

Rules for judgment *value*

$$\frac{}{value(\mathbf{sig}\ B_{sig})} \quad (95)$$

Rules for judgment *closed*

$$\frac{}{closed\ \mathbf{sig}\ B_{sig}} \tag{96}$$

$$\frac{closed\ M \quad closed\ N}{closed\ M : N} \tag{97}$$

$$\frac{}{closed\ \mathbf{struct}\ \ \mathbf{end}} \tag{98}$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots\end{array}}{closed\ M\ closed(\mathbf{struct}\ B_{str})} \tag{99}$$
$$closed(\mathbf{struct}\ x = M\ B_{str})$$

$$\frac{closed\ u}{closed\ u.x} \tag{100}$$

$$\frac{closed\ u \quad closed\ M}{closed(\mathbf{open}\ u\ \mathbf{in}\ M)} \tag{101}$$

Rules for judgments *in*, *proj*, *upd*

$$\frac{}{m\ in\ (m :: l)} \tag{102}$$

$$\frac{m\ in\ l}{m\ in\ (p :: l)} \tag{103}$$

$$\frac{}{proj\ l\ (\mathbf{sig}\ \ \mathbf{end}\ )\ nil} \tag{104}$$

$$\frac{(x \mapsto m)\ in\ l \quad proj\ l\ (\mathbf{sig}\ B_{sig})\ l'}{proj\ l\ (\mathbf{sig}\ x\ B_{sig})\ (x \mapsto m, l')} \tag{105}$$

$$\frac{}{upd\ (x \mapsto n, l)\ x\ m\ (x \mapsto m, l)} \tag{106}$$

$$\frac{upd\ l\ x\ m\ l'}{upd\ (y \mapsto n, l)\ x\ m\ (y \mapsto n, l')}\ x \neq y \tag{107}$$

B.4 NOS of $\mathcal{L}_{M_I}$

Rules for judgment *closed*

$$\frac{closed\ T}{closed\ T.f} \tag{108}$$

Rules for judgment $\gg$

$$\frac{\begin{array}{c}(closed\ T)\\ \vdots\\ R \gg I\end{array}}{[p/T]_m R \gg I} \tag{109}$$

Rules for judgment *free*

$$\frac{free\ R\ m \quad free\ M\ m}{free\ R.P(M)} \tag{110}$$

$$\frac{free\ R\ m}{free\ R.f\ m} \tag{111}$$

$$\frac{free\ p\ m \quad free\ N\ (R, m)}{free\ [p/R]_m N\ m} \tag{112}$$

$$\frac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ (x,y) \quad \mathit{free}\ N\ (x) \quad \mathit{free}\ D\ (R,m)}{\mathit{free}\ (\mathbf{module}\ R\ \mathbf{is}\ x = M;\ \mathbf{proc}\ P(y) = C;\ \mathbf{func}\ f = N \mathbf{in}\ D)\ m} \tag{113}$$

Rules for judgment $\Rightarrow$

$$\frac{\begin{pmatrix} R' \Rightarrow_m (m, R') \\ (R', P) \Rightarrow_{mp} \boldsymbol{\lambda} x, y.C \\ (R', f) \Rightarrow_{mf} \boldsymbol{\lambda} x.N \end{pmatrix} \quad \vdots \quad M \Rightarrow m \quad \mathit{free}\ C\ (x,y) \quad \mathit{free}\ N\ (x) \quad [D']N \Rightarrow n'}{[\mathbf{module}\ R\ \mathbf{is}\ x = M_1;\ \mathbf{proc}\ P(y) = C;\ \mathbf{func}\ f = M_2 \mathbf{in}\ D]N \Rightarrow n}\ E(R,D,m) \tag{114}$$

$$\frac{R \Rightarrow_m (p, R') \quad (R', P) \Rightarrow_{mp} \mathbf{lambda}\, x, y.C \quad M \Rightarrow m \quad [p/x][m/y][C]x \Rightarrow p' \quad [p'/R]_m N \Rightarrow n}{[R.P(M)]N \Rightarrow n} \tag{115}$$

$$\frac{(R' \Rightarrow_m (p, T)) \quad \vdots \quad \mathit{value}\ p \quad R \Rightarrow_m (\_, T) \quad N' \Rightarrow n'}{[p/R]_m N \Rightarrow n}\ E(R,N,n) \tag{116}$$

$$\frac{R \Rightarrow_m (p,T) \quad (T,f) \Rightarrow_{mf} \mathbf{lambda}\, x.M \quad [p/x]M \Rightarrow m}{R.f \Rightarrow m} \tag{117}$$

C Denotational semantics

C.1 Denotational semantics of $\mathcal{L}_P$

Semantic domains

$\mathbb{V} = (\mathbb{N} + \mathbb{T} + \mathbb{U} + \mathbb{P} + \mathbb{F})^{\top}_{\perp}$	$\mathbb{P} = \mathbb{V} \times \mathbb{V}$
$\mathbb{N} = \mathit{Nat}$ (the domain of natural numbers)	$\mathbb{F} = \mathbb{V} \to \mathbb{V}$
$\mathbb{T} = \mathit{Truth}$ (the domain of truth values)	$\mathbb{E} = ((\mathit{Id} \to \mathbb{V}) \times (\mathit{ProcId} \to \mathbb{Q}))^{\top}$
$\mathbb{U} = \mathit{Unit}$ (the one-element domain)	$\mathbb{Q} = (\mathit{Id} \to \mathbb{V} \to \mathbb{E} \to \mathbb{E})^{\top}$

Operators

newenv	$= (\lambda x.\top, \lambda p.\top)$	$: \mathbb{E}$
update	$= \lambda x.\lambda n.\underline{\lambda}(\rho_v, \rho_p).([x \mapsto n]\rho_v, \rho_p)$	$: \mathit{Id} \to \mathbb{V} \to \mathbb{E} \to \mathbb{E}$
access	$= \lambda x.\underline{\lambda}(\rho_v, \rho_p).\rho_v(x)$	$: \mathit{Id} \to \mathbb{E} \to \mathbb{V}$
$\mathit{procupdate}$	$= \lambda p.\lambda q.\underline{\lambda}(\rho_v, \rho_p).(\rho_v, [p \mapsto q]\rho_p)$	$: \mathit{ProcId} \to \mathbb{Q} \to \mathbb{E} \to \mathbb{E}$
$\mathit{procaccess}$	$= \lambda p.\underline{\lambda}(\rho_v, \rho_p).\rho_p(p)$	$: \mathit{ProcId} \to \mathbb{E} \to \mathbb{Q}$
$\mathit{overlay}$	$= \underline{\lambda}\rho_1.\underline{\lambda}\rho_2.\lambda x.\mathbf{if}\ \mathrm{is}\top(\rho_2(x)) \to \rho_1(x)\ [\!]\ \rho_2(x)$	$: \mathbb{E} \to \mathbb{E} \to \mathbb{E}$

Semantic functions

$$\mathcal{E} : \mathit{Expr} \to \mathbb{E} \to \mathbb{V} \qquad \mathcal{D} : \mathit{Declarations} \to \mathbb{E} \to \mathbb{E}$$
$$\mathcal{C} : \mathit{Commands} \to \mathbb{E} \to \mathbb{E} \quad \mathcal{Q} : \mathit{Procedures} \to \mathbb{E} \to \mathbb{Q}$$

$$\mathcal{E}[\![x]\!] = \underline{\lambda}\rho.\mathit{access}[\![x]\!]\rho \qquad \mathcal{E}[\![0]\!] = \underline{\lambda}\rho.\mathrm{in}\mathbb{N}(\mathit{zero}) \qquad \mathcal{E}[\![\mathit{nil}]\!] = \underline{\lambda}\rho.\mathrm{in}\mathbb{U}()$$

$$\mathcal{E}[\![\mathit{true}]\!] = \underline{\lambda}\rho.\mathrm{in}\mathbb{T}(\mathit{true}) \qquad \mathcal{E}[\![\mathit{false}]\!] = \underline{\lambda}\rho.\mathrm{in}\mathbb{T}(\mathit{false})$$

$\mathcal{E}[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ \mathcal{E}[\![N]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)$

$\mathcal{E}[\![\mathbf{letrec}\ f(x) = M\ \mathbf{in}\ N]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ g = \mathit{fix}(\underline{\lambda}g.\underline{\lambda}v.\mathcal{E}[\![M]\!](\mathit{update}\ [\![f]\!]\ g\ \rho))\ \mathbf{in}$
$\mathcal{E}[\![N]\!](\mathit{update}\ [\![f]\!]\ g\ \rho)$

$\mathcal{E}[\![\mathbf{lambda}\ x.M]\!] = \ \underline{\lambda}\rho.\mathrm{in}\mathbb{F}(\underline{\lambda}v.\mathcal{E}[\![M]\!](\mathit{update}\ [\![x]\!]\ v\ \rho))$

$\mathcal{E}[\![M\ N]\!] = \ \underline{\lambda}\rho.\mathbf{cases}\ \mathcal{E}[\![M]\!]\rho\ \mathbf{of}\ \mathrm{is}\mathbb{F}(f) \to f(\mathcal{E}[\![N]\!]\rho)[\!]\ \top\mathbf{end}$

$\mathcal{E}[\![M :: N]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ v_1 = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ \mathbf{let}\ v_2 = \mathcal{E}[\![N]\!]\rho\ \mathbf{in}\ \mathrm{in}\mathbb{P}((v_1, v_2))$

$\mathcal{E}[\![[m/x]N]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ v = \mathcal{E}[\![m]\!]\rho\ \mathbf{in}\ \mathcal{E}[\![N]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)$

$\mathcal{E}[\![m \cdot n]\!] = \ \underline{\lambda}\rho.\mathbf{cases}\ \mathcal{E}[\![m]\!]\ \mathbf{of}\ \mathrm{is}\mathbb{F}(f) \to f(\mathcal{E}[\![n]\!]\rho)[\!]\ \top\mathbf{end}$

$\mathcal{E}[\![[\mathbf{on}\ \bar{x} = \bar{M}\ \mathbf{do}\ C]N]\!] = \ \underline{\lambda}\rho.\mathbf{if}\ (\mathit{maxfree}\ [\![C]\!](\bar{x})) \to \mathcal{C}[\![C]\!](\mathcal{D}[\![\bar{x} = \bar{M}]\!]\rho)[\!]\ \top$

where $\mathit{maxfree} : \mathit{Commands} \to \mathit{Id}^* \to \mathbb{T}$; the meaning of
"$\mathit{maxfree}\ [\![C]\!]\ s = \mathit{true}$" is simply "every free identifier of C is in s". $\mathit{maxfree}$ is trivially defined on the syntactic structure of commands; we omit its definition.

$\mathcal{D}[\![\langle\rangle]\!] = \ \underline{\lambda}\rho.\mathit{newenv}$

$\mathcal{D}[\![x = M; R]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}$
$\mathbf{let}\ \tau = \mathcal{D}[\![R]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)\ \mathbf{in}$
$\mathit{overlay}\ \tau\ (\mathit{update}\ [\![x]\!]\ v\ \mathit{newenv})$

$\mathcal{D}[\![[n/x]_d R]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ v = \mathcal{E}[\![n]\!]\rho\ \mathbf{in}\ \mathcal{D}[\![R]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)$

$\mathcal{C}[\![x := M]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ \mathit{update}\ [\![x]\!]\ v\ \rho$

$\mathcal{C}[\![\mathbf{while}\ M\ \mathbf{do}\ C]\!] = \ \mathit{fix}(F)$

where $F : (\mathbb{E} \to \mathbb{E}) \to (\mathbb{E} \to \mathbb{E})$ $F = \ \underline{\lambda}f.\underline{\lambda}\rho.\mathbf{cases}\ \mathcal{E}[\![M]\!]\rho\ \mathbf{of}$
$\mathrm{is}\mathbb{T}(t) \to \mathbf{if}\ t \to f(\mathcal{C}[\![C]\!]\rho)\ [\!]\ \rho$
$[\!]\ \top$
$\mathbf{end}$

$\mathcal{C}[\![\mathbf{beginnew}\ x = M;\ C\ \mathbf{end}]\!] = \ \underline{\lambda}\rho.\mathbf{let}\ \rho' = \mathit{update}\ [\![x]\!]\ (\mathcal{E}[\![M]\!]\rho)\ \rho\ \mathbf{in}$
$\mathbf{let}\ \rho'' = \mathcal{C}[\![C]\!]\rho'\ \mathbf{in}$
$\mathit{update}\ [\![x]\!]\ (\mathit{access}\ [\![x]\!]\ \rho)\ \rho''$

$\mathcal{C}[\![\mathbf{proc}\ P(x, y) = C\ \mathbf{in}\ D]\!] =$
$\underline{\lambda}\rho.\mathbf{let}\ \rho' = \mathit{procupdate}\ [\![P]\!]\ (\mathcal{Q}[\![\mathbf{lambda}\ x, y.C]\!]\rho)\ \rho\ \mathbf{in}$
$\mathbf{let}\ \rho'' = \mathcal{C}[\![D]\!]\rho'\ \mathbf{in}$
$\mathit{procupdate}\ [\![P]\!]\ (\mathit{procaccess}\ [\![P]\!]\ \rho)\ \rho''$

$$\mathcal{C}[\![P(x,M)]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ ((procaccess\ [\![P]\!]\ \rho)\ [\![x]\!]\ v\ \rho)$$

$$\begin{array}{l}\mathcal{Q}[\![\mathbf{lambda}\ x,y.C]\!] = \\ \underline{\underline{\lambda}}\rho.\mathbf{if}\ maxfree\ [\![C]\!]\ ([\![x]\!],[\![y]\!])emptysign)) \rightarrow \\ \quad \lambda i.\underline{\underline{\lambda}}v_y.\underline{\underline{\lambda}}\tau.\mathbf{let}\ v_x = (access\ i\ \tau)\ \mathbf{in} \\ \quad\quad \mathbf{let}\ \rho' = \mathcal{C}[\![C]\!](update\ [\![y]\!]\ v_y\ (update\ [\![x]\!]\ v_x\ \rho))\ \mathbf{in} \\ \quad\quad\quad update\ i\ (access\ [\![x]\!]\ \rho')\ \tau \\ \quad [\![\top\end{array}$$

$$\mathcal{E}[\![[Q/P]_{pe}M]\!] = \underline{\underline{\lambda}}\rho.\mathcal{E}[\![M]\!](procupdate\ [\![P]\!]\ \mathcal{Q}[\![Q]\!]\rho\ \rho)$$

C.2 Denotational semantics of $\mathcal{L}_D$

Semantic functions $\mathcal{O} : SyntEnvir \rightarrow \mathbb{E} \rightarrow \mathbb{E}$

$$\mathcal{E}[\![\mathbf{let}\ R\ \mathbf{in}\ N]\!] = \underline{\underline{\lambda}}\rho.\mathcal{E}[\![N]\!](overlay\ (\mathcal{D}[\![R]\!]\rho)\ \rho)$$

$$\mathcal{E}[\![\{r\}M]\!] = \underline{\underline{\lambda}}\rho.\mathcal{E}[\![M]\!](\mathcal{O}[\![r]\!]\rho) = \mathcal{E}[\![M]\!] \circ \mathcal{O}[\![r]\!]$$

$$\mathcal{D}[\![R\ \mathbf{and}\ S]\!] = \underline{\underline{\lambda}}\rho.overlay\ (\mathcal{D}[\![S]\!]\rho)\ (\mathcal{D}[\![R]\!]\rho)$$

$$\mathcal{O}[\![x \mapsto n]\!] = \underline{\underline{\lambda}}\rho.update\ (\mathcal{E}[\![n]\!]\rho)\ \rho$$

$$\mathcal{O}[\![r :: s]\!] = \mathcal{O}[\![s]\!] \circ \mathcal{O}[\![r]\!]$$

C.3 Denotational semantics of $\mathcal{L}_{M_F}$

Semantic domains

$$\begin{array}{llll}\mathbb{V} = (\mathbb{N}+\mathbb{U}+\mathbb{P}+\mathbb{F}+\mathbb{B}+\mathbb{S})^{\top}_{\perp} & & \mathbb{B} = Id \times \mathbb{V} \\ \mathbb{U} = Unit & & \mathbb{S} = Id^* = \mathbb{ES} + \mathbb{CS} \\ \mathbb{P} = \mathbb{V} \times \mathbb{V} & & \mathbb{ES} = Unit \\ \mathbb{F} = \mathbb{V} \rightarrow \mathbb{V} & & \mathbb{CS} = Id \times \mathbb{S}\end{array}$$

Operators

$$\begin{array}{lll}emptystruct & = \mathrm{in}\mathbb{U}() & : \mathbb{V} \\ consstruct & = \lambda x.\underline{\underline{\lambda}}v.\underline{\underline{\lambda}}c.\mathrm{in}\mathbb{P}(\mathrm{in}\mathbb{B}([\![x]\!],v),c) & : Id \rightarrow \mathbb{V} \rightarrow \mathbb{V} \rightarrow \mathbb{V} \\ emptysign & = \mathrm{in}\mathbb{ES}() & : \mathbb{V} \\ conssign & = \lambda i.\lambda t.\mathrm{in}\mathbb{CS}((i,t)) & : Id \rightarrow \mathbb{S} \rightarrow \mathbb{S} \\ accessstruct & = \text{see below} & : Id \rightarrow \mathbb{V} \rightarrow \mathbb{V} \\ applystruct & = \text{see below} & : \mathbb{V} \rightarrow \mathbb{E} \rightarrow \mathbb{E} \\ projection & = \text{see below} & : \mathbb{V} \rightarrow \mathbb{S} \rightarrow \mathbb{V} \\ longupdate & = \text{see below} & : LongId \rightarrow \mathbb{V} \rightarrow \mathbb{E} \rightarrow \mathbb{E}\end{array}$$

$$\begin{array}{l}projection = \underline{\underline{\lambda}}s.\lambda t.\mathbf{cases}\ t\ \mathbf{of} \\ \qquad\qquad \mathrm{is}\mathbb{ES}() \rightarrow \mathrm{in}\mathbb{U}() \\ \qquad\qquad [\![\ \mathrm{is}\mathbb{CS}(i,t') \rightarrow \\ \qquad\qquad\quad \mathbf{let}\ v = accessstruct\ i\ s\ \mathbf{in} \\ \qquad\qquad\qquad \mathbf{let}\ s' = projection\ s\ t'\ \mathbf{in}\ consstruct\ i\ v\ s' \\ \qquad\quad \mathbf{end}\end{array}$$

Semantic functions $\mathcal{E}[\![\textbf{struct end}\,]\!] = \underline{\underline{\lambda}}\rho.emptystruct$

$\mathcal{E}[\![x \mapsto n]\!] = \underline{\underline{\lambda}}\rho.\textbf{let}\ v = \mathcal{E}[\![n]\!]\rho\ \textbf{in}\ \mathrm{in}\mathbb{B}([\![x]\!], v)$

$$\mathcal{E}[\![\textbf{struct}\ x = M\ B_{str}]\!] = \underline{\underline{\lambda}}\rho.\textbf{let}\ v_1 = \mathcal{E}[\![M]\!]\rho\ \textbf{in}\ \textbf{let}\ v_2 = \mathcal{E}[\![\textbf{struct}\ B_{str}]\!](update\ [\![x]\!]\ v_1\ \rho)\ \textbf{in}\ consstruct\ [\![x]\!]\ v_1\ v_2$$

$\mathcal{E}[\![\textbf{sig end}\,]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{S}(emptysign)$

$$\mathcal{E}[\![\textbf{sig}\ x\ B_{sig}]\!] = \underline{\underline{\lambda}}\rho.\textbf{cases}\ \mathcal{E}[\![\textbf{sig}\ B_{sig}]\!]\rho\ \textbf{of}\ \mathrm{is}\mathbb{S}(s) \to \mathrm{in}\mathbb{S}(conssig\ [\![x]\!]\ s)\ [\!]\ \top\ \textbf{end}$$

$$\mathcal{E}[\![M : N]\!] = \underline{\underline{\lambda}}\rho.\textbf{let}\ s = \mathcal{E}[\![M]\!]\rho\ \textbf{in}\ \textbf{cases}\ \mathcal{E}[\![N]\!]\rho\ \textbf{of}\ \mathrm{is}\mathbb{S}(t) \to projection\ s\ t\ [\!]\ \top\ \textbf{end}$$

$\mathcal{E}[\![\textbf{open}\ u\ \textbf{in}\ M]\!] = \underline{\underline{\lambda}}\rho.\textbf{let}\ s = \mathcal{E}[\![u]\!]\rho\ \textbf{in}\ \mathcal{E}[\![M]\!](applystruct\ s\ \rho)$

$\mathcal{E}[\![u.x]\!] = \underline{\underline{\lambda}}\rho.accessstruct\ [\![x]\!]\ (\mathcal{E}[\![u]\!]\rho)$

C.4 Denotational semantics of $\mathcal{L}_{M_I}$

Semantic domains

$$\begin{array}{rlrl} \mathbb{E} &= (\mathbb{IM} \times \mathbb{PM} \times \mathbb{MM})^\top & \mathbb{MM} &= ModId \to \mathbb{M} \\ \mathbb{IM} &= Id \to \mathbb{V} & \mathbb{M} &= (\mathbb{V} \times \mathbb{Q}_M \times \mathbb{F}_M)^\top \\ \mathbb{PM} &= ProcId \to \mathbb{Q} & \mathbb{Q}_M &= \mathbb{V} \to \mathbb{V} \to \mathbb{V} \\ \mathbb{Q} &= (Id \to \mathbb{V} \to \mathbb{E} \to \mathbb{E})^\top & \mathbb{F}_M &= \mathbb{V} \to \mathbb{V} = \mathbb{F} \end{array}$$

Operators

$$\begin{array}{lll} newenv &= (\lambda x.\top, \lambda p.\top, \lambda r.\top) & : \mathbb{E} \\ update &= \lambda x.\lambda n.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).([x \mapsto n]\rho_v, \rho_p, \rho_m) & : Id \to \mathbb{V} \to \mathbb{E} \to \mathbb{E} \\ access &= \lambda x.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_v(x) & : Id \to \mathbb{E} \to \mathbb{V} \\ procupdate &= \lambda p.\lambda q.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).(\rho_v, [p \mapsto q]\rho_p, \rho_m) & : ProcId \to \mathbb{Q} \to \mathbb{E} \to \mathbb{E} \\ procaccess &= \lambda p.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_p(p) & : ProcId \to \mathbb{E} \to \mathbb{Q} \\ modupdate &= \lambda r.\lambda q.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).(\rho_v, \rho_p, [r \mapsto m]\rho_m) & : ProcId \to \mathbb{Q} \to \mathbb{E} \to \mathbb{E} \\ modaccess &= \lambda r.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_m(r) & : ProcId \to \mathbb{E} \to \mathbb{Q} \end{array}$$

Semantic functions

$$\mathcal{C}[\![\textbf{module}\ R\ \textbf{is}\ x = M;\ \textbf{proc}\ P(y) = C;\ \textbf{func}\ f = N\ \textbf{in}\ D]\!] = \underline{\underline{\lambda}}\rho.\textbf{if}\ maxfree\ [\![C]\!]\ (conssign\ [\![x]\!]\ (conssign\ [\![y]\!] emptysign)) \to \textbf{if}\ maxfree\ [\![N]\!]\ (conssign\ [\![x]\!]\ emptysign) \to$$

$$\begin{array}{l}
\textbf{let } m = \mathcal{E}[\![M]\!]\rho \textbf{ in} \\
\quad \textbf{let } q = \underline{\underline{\lambda}} v_x.\underline{\underline{\lambda}} v_y.access\ [\![x]\!]\ \mathcal{C}[\![C]\!](update\ [\![y]\!]\ v_y\ (update\ [\![x]\!]\ v_x\ \rho)) \textbf{ in} \\
\quad\quad \textbf{let } g = \underline{\underline{\lambda}} v_x.\mathcal{E}[\![N]\!](update\ [\![x]\!]\ v_x\ \rho) \textbf{ in} \\
\quad\quad\quad \textbf{let } \rho' = modupdate\ [\![R]\!]\ (m, q, g)\ \rho \textbf{ in} \\
\quad\quad\quad\quad modupdate\ [\![R]\!]\ (modaccess\ [\![R]\!]\ \rho)\ \mathcal{C}[\![D]\!]\rho'
\end{array}$$

$$]\!]\ \top$$

$$]\!]\ \top$$

$$\begin{array}{rl}
\mathcal{C}[\![R.P(M)]\!] = & \underline{\underline{\lambda}}\rho.\textbf{let } (n, q, g) = modaccess\ [\![R]\!]\ \rho \textbf{ in} \\
& \quad \textbf{let } n' = q\ n\ (\mathcal{E}[\![M]\!]\rho) \textbf{ in} \\
& \quad\quad modupdate\ [\![R]\!]\ (n', q, g)\ \rho
\end{array}$$

$$\mathcal{E}[\![R.f]\!] = \underline{\underline{\lambda}}\rho.\textbf{let } (n, q, g) = modaccess\ [\![R]\!]\ \rho \textbf{ in } (g\ n)$$

$$\begin{array}{rl}
\mathcal{E}[\![[n/R]_m M]\!] = & \underline{\underline{\lambda}}\rho.\textbf{let } (_, q, g) = modaccess\ [\![R]\!]\ \rho \textbf{ in} \\
& \quad \textbf{let } m' = \mathcal{E}[\![n]\!]\rho \textbf{ in } \mathcal{E}[\![M]\!](modupdate\ [\![R]\!]\ (m', q, g)\ \rho)
\end{array}$$

Acknowledgments

Many of the ideas in this paper were first presented in [15]. I am most grateful to Furio Honsell for his invaluable and patient support and for directing my thesis work. I wish to acknowledge Randy Pollack and Frank Pfenning for several useful discussions and for making accessible via `ftp` the LEGO and Elf systems respectively. I would like to thank also the anonymous referees for the useful comments on the earlier version of this paper.

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1985.
2. A. Avron. Simple consequence relations. *Information and Computation*, 92:105–139, Jan. 1991.
3. A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using Typed Lambda Calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.
4. R. Burstall and F. Honsell. Operational semantics in a natural deduction setting. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 185–214. Cambridge University Press, June 1990.
5. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
6. J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First Conference on Logic in Computer Science*, pages 193–205. The Association for Computing Machinery, 1986.
7. J. E. Donahue. Locations considered unnecessary. *Acta Informatica*, 8:221–242, July 1977.

8. G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North Holland, 1969.
9. J. J. Hannan. Proof-theoretical methods for analysis of functional programs. Technical Report MS–CIS–89–07, Dep. of Computer and Information Science, University of Pennsylvania, Dec. 1988.
10. J. J. Hannan and F. Pfenning. Compiler verification in LF. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.
11. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.
12. G. Kahn. Natural Semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39. Springer-Verlag, 1987.
13. Z. Luo, R. Pollack, and P. Taylor. *How to use LEGO (A Preliminary User's Manual)*. Department of Computer Science, University of Edinburgh, Oct. 1989.
14. S. Michaylov and F. Pfenning. Natural Semantics and some of its Meta-Theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, number 596 in LNAI, pages 299–344, Stockolm, Sweden, Jan. 1991. Springer-Verlag.
15. M. Miculan. Semantica operazionale strutturata ad ambienti distribuiti – teoria e sperimentazione. Undergraduate thesis, Università di Udine, Udine, Italy, July 1992. In italian.
16. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.
17. E. Moggi. Notions of computation and monads. *Information and Computation*, 1, 1993.
18. J. H. Morris, Jr. Types are not sets. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 120–124, Boston, Oct. 1973. The Association for Computing Machinery.
19. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monograph on Computer Science*. Oxford University Press, 1990.
20. F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE, June 1989. Also available as ERGO Report 89–067, School of Computer Science, Carnegie Mellon Univ., Pittsburgh.
21. G. D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Computer Science Department, Århus University, Århus, Denmark, Sept. 1981.
22. G. D. Plotkin. Notes about semantics. Unpublished notes given at CSLI, Stanford, Aug. 1985.
23. D. A. Schmidt. *Denotational Semantics*. Allyn & Bacon, 1986.
24. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91–160, rev.2, Department of Computer Science, Rice University, Houston, Texas, 1991.

This article was processed using the LaTeX macro package with LLNCS style