LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

On Natural Deduction Style Semantics, Environments and Stores

On Natural Deduction Style Semantics, Environments and Stores

by

Christian-Emil Ore

ECS-LFCS-89-88

LFCS Report Series (also published as CSR-305-89)

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

June 1989

On natural deduction style semantics, environments and stores

Christian-Emil Ore *
Department of Computer Science
University of Edinburgh

June 30, 1989

1 Introduction

The origin for the work presented in this paper was a wish to compare the rather different approaches to natural deduction style semantics presented by Burstall and Honsell [BurHon88] and by Hannan and Miller [HanMil88], and see which class of programming languages could be presented in the formalisms in a natural way. The semantics in the two papers differ from ordinary operatinal semantics in that the semantics are presented as natural deduction systems ([Praw65]).

I was also interested how the two methods should be implemented in the Edinburgh Logical Framework (ELF) [Har87]. The implementations in this paper are done in Pollack's Lego system [Pol88]. This system is a proof system for an extended version of Theory of Construction [CoqHuet88], in which ELF can be seen as a subcalculus. In the implementations only the ELF part of the calculus is used, although a full use of the second order features would have made the implementation smoother. It has to be mentioned that implementations are described in both [BurHon88] and [HanMil88], the implementations are done in ELF and λ-prolog respectively.

In the following I will also use the shorter term 'natural semantics'. This term was introduced by Kahn et al. ([Kahn87]) to denote a formalization of operational semantics originated from the work of Plotkin ([Plot81]). Their use of the term is misleading since, as we will see, standard operational semantics is a sequent calculus. In this paper the term 'natural semantics' will be used in the sense "natural deduction style operational semantics".

Section 2 consists of a survey of the different kind of semantical systems. The example language is a simple purely functional language. In section 3 various kinds of assignments and side-effects are introduced. In section 4 the implementations of the semantics are discussed, and a module for representing stores is presented.

*On leave from University of Oslo, the author's address: University of Oslo, Department of Computer Science, P.O. box 1080 Blindern, 0316 Oslo 3, Norway

2 Natural deduction style semantics

2.1 Operational semantics

The usual sample language is a kind of mini ML containing a choice operator, declaration, abstraction and application. Assignment and other sequential features will be included in section 3.

Syntax :

$e ::= x \mid v \mid \texttt{IF}\ e\ \texttt{THEN}\ e\ \texttt{ELSE}\ e \mid (e)e \mid \texttt{LET}\ e\ \texttt{IN}\ e \mid \Lambda x.e$

The usual dynamic operational semantic is given below.

Rules for IF and LET :

$$\frac{\rho \vdash e_1 \Rightarrow \mathit{true} \quad \rho \vdash e_2 \Rightarrow \alpha}{\rho \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\texttt{ELSE}\ e_3 \Rightarrow \alpha} \qquad \frac{\rho \vdash e_1 \Rightarrow \mathit{false} \quad \rho \vdash e_3 \Rightarrow \alpha}{\rho \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\texttt{ELSE}\ e_3 \Rightarrow \alpha}$$

$$\frac{\rho \vdash e_1 \Rightarrow \alpha \quad \rho[x \mapsto \alpha] \vdash e_2 \Rightarrow \beta}{\rho \vdash \texttt{LET}\ x = e_1\ \texttt{IN}\ e_2 \Rightarrow \beta}$$

Rule for'by value' application:

$$\frac{\rho \vdash e_1 \Rightarrow [\Lambda x.e, \rho'] \quad \rho \vdash e_2 \Rightarrow \alpha \quad \rho'[x \mapsto \alpha] \vdash e \Rightarrow \beta}{\rho \vdash (e_1)e_2 \Rightarrow \beta}$$

Rule for closures:

$\rho \vdash \Lambda x.e \Rightarrow [\Lambda x.e, \rho]$

The above rules can be seen as definition rules for a three place predicate '$\_ \vdash \_ \Rightarrow \_$'. Under this interpretation they constitute a part of a (extended) Prawitz natural deduction system, and the semantics is a natural deduction style semantics.

However, the turn stile can be taken seriously as a deduction symbol. In this case the above rules, with some extensions, will define a logic in which the theorems are the evaluations of closed programs, that is, programs that can be evaluated in an empty environment. An environment can therefore be seen as a set of hypothetic evaluations of the variables. The turn stile symbol represents a consequence relation between such hypothetical evaluations and the evaluation of a program term.

The above logic is defined as a sequent calculus. Is this sequent calculus a disguised natural deduction system? In other words, is it possible to see the environment ρ in $\rho \vdash M \Rightarrow m$ as the set of the undischarged assumptions in a proof tree with $M \Rightarrow m$ as root?

The above system can be seen as a natural deduction system written as sequents, that is, the step from $\rho[x \mapsto \alpha]$ to ρ expresses the assumption discharging. This natural deduction system will, however, not only describe the semantics of the languages, but a lot of unwanted results as well. The following example illustrate this problem.

Example 2.1 Assume that the above semantics describes a natural deduction system on sequent form and that it is extended with some δ-rules for numbers. The following deductions are possible in the system. (The indices on the assumptions show at which point in the deduction they are discharged.)

$$\dfrac{3 \Rightarrow 3 \qquad \dfrac{2 \Rightarrow 2 \qquad \dfrac{(x \Rightarrow 3)_1 \quad (x \Rightarrow 2)_2}{x \Rightarrow 2}}{\texttt{LET}\ x = 2\ \texttt{IN}\ x \Rightarrow 2}\,{}^{2}}{\texttt{LET}\ x = 3\ \texttt{IN LET}\ x = 2\ \texttt{IN}\ x \Rightarrow 2}\,{}^{1} \qquad \dfrac{3 \Rightarrow 3 \qquad \dfrac{2 \Rightarrow 2 \qquad \dfrac{(x \Rightarrow 3)_1 \quad (x \Rightarrow 2)_2}{x \Rightarrow 3}}{\texttt{LET}\ x = 2\ \texttt{IN}\ x \Rightarrow 3}\,{}^{2}}{\texttt{LET}\ x = 3\ \texttt{IN LET}\ x = 2\ \texttt{IN}\ x \Rightarrow 3}\,{}^{1}$$

The left deduction is what we want. The deduction on the the right hand side states a fact which is false according to the intended semantics.
End of example.

The expression $\rho[x \mapsto \alpha]$ is usually read "extend the environment ρ such that the value α is bound to the variable x". At least for all non deterministic languages it is understood that α is now the only value bound to x. The above anomaly is caused by the interpretation of ρ as a set and $\rho[x \mapsto \alpha]$ as the union $\rho \cup \{x \Rightarrow \alpha\}$, which makes it possible to bind more than one value to a variable x.

The expression $\rho[x \mapsto \alpha]$ must be interpreted such that the above anomaly is avoided. The environment can still be seen as a set of assumptions, but the expression $\rho[x \mapsto \alpha]$ now denotes the set constructed by removing any element on the form $x \Rightarrow \alpha'$ and adding the new binding $x \Rightarrow \alpha$.

This necessary manipulation of the environment, that is, the assumptions, makes the rules for abstraction and declaration highly impure. The above use of sequents makes it possible to express this impurity in a proper manner, but it is not a sequent representation of a natural deduction system.

However, it is possible to define a natural deduction semantics for a language as above. This is done both in [BurHon88] and [HanMil88].

2.2 A natural semantics based on a substitution operator

In the semantics in the previous section the operator $\rho[x \mapsto \alpha]$ could not be defined as the union $\rho \cup \{x \Rightarrow \alpha\}$ since this could result in a non deterministic environment. However, this restriction can be lifted by doing some variable renaming in the terms on the right hand side of the turn stile.

Consider the LET rule. Assume that in the premise all the occurrences of the variable x in e_2 and β were replaced by a new variable x'. The rule could then be restated as

$$\frac{\rho \vdash e_1 \Rightarrow \alpha \quad \rho \cup \{x' \mapsto \alpha\} \vdash [x'/x]e_2 \Rightarrow [x'/x]\beta}{\rho \vdash \texttt{LET } x = e_1 \texttt{ IN } e_2 \Rightarrow \beta} \tag{1}$$

That is, the environment ρ is interpreted as a set and the expression $\rho[x' \mapsto \alpha]$ as the union $\rho \cup \{x' \Rightarrow \alpha\}$. The above rule is a sequent variant of a natural deduction style rule. One can say that the use of renaming in the rule simulates a kind of meta parenthesis in the deduction tree.

This is the main idea behind the system developed in [BurHon88]. Here first order natural deduction is extended with two new operators to what is called α-logic. These operators correspond roughly to the renaming in (1). The extended logic is used to define a substitution operator with semantics corresponding to the second sequent in the antecedent of (1). The aim is to show that the extended natural deduction system is an appropriate system for defining semantics of programming languages. It is claimed that by using substitution operators of the above kind it is possible to write the semantical rules without using neither explicit environments nor explicit stores. The former is clearly true, but the latter is, as we will see, only true for languages with restricted assignment (see section 4).

Some of the semantical rules for the simple sample language are shown below.

Rules for IF and LET :

$$\frac{L \Rightarrow true \quad M \Rightarrow m}{\texttt{IF } L \texttt{ THEN } M \texttt{ ELSE } N \Rightarrow m} \qquad \frac{L \Rightarrow false \quad N \Rightarrow n}{\texttt{IF } L \texttt{ THEN } M \texttt{ ELSE } N \Rightarrow n}$$

$$\frac{N \Rightarrow n \quad [n/x]M \Rightarrow m}{\texttt{LET } x = N \texttt{ IN } M \Rightarrow m}$$

Rules for 'by value' application:

$$\frac{M \Rightarrow m \quad N \Rightarrow n \quad m.n \Rightarrow p}{MN \Rightarrow p} \qquad \frac{value\ n \quad [p/y](f.n) \Rightarrow m}{([p/y]f).n \Rightarrow}$$

$$\frac{value\ n \quad [n/x]M \Rightarrow m}{(\Lambda\, x\, M).n \Rightarrow m}$$

Rules for closures:

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots \\ closed\ M\end{array}}{\Lambda\, x\, M \Rightarrow (\Lambda\, x\, M)} \qquad \frac{\begin{array}{c}(closed\ y)\\ \vdots \\ y \Rightarrow p \quad \Lambda\, x\, M \Rightarrow m\end{array}}{\Lambda\, x\, M \Rightarrow [p/y]m}$$

As mentioned above the terms on the form $[n/x]M$ do not indicate ordinary substitution. It is a shorthand notation for the kind of α-conversion and discharging of an assumption as we find in (1) above.

Assume that x' does not occur in M, m or x. Let $M^{x'}$ be M with all occurrences of x replaced by x' and let $m_{x'}$ be m with some occurrences of x replaced by x'. The rule for $[n/x]$ is as follows

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ M^{x'} \Rightarrow m_{x'}\end{array}}{[n/x]M \Rightarrow m}$$

where x' is not free in any assumptions other than $x' \Rightarrow n$

This rule is, as said in [BurHon88], not more complicated than the $\exists$ introduction rule. In fact, it is simpler. There is only a finite number of substitution possibilities. The above rule is in [BurHon88] defined by the use of two operators α and $\underline{\alpha}$ expressing the substitution in Mx' and $m_{x'}$. The use of two different renaming operators seems to be necessary from the way closures are defined (see also section 4).

2.3 A meta language approach to natural semantics

In [HanMil88] the angle of attack is quite different although the goal is to obtain a semantical system on a natural deduction form without use of explicit environments. They claim that second order features should be used to achieve this goal. The standard BNF syntax is replaced by a formalized syntax. This formalization of the syntax corresponds closely to the standard ELF way of syntax formalization (Avr87). The BNF syntax given in section 2.1 can be seen as a set of rules for the formation of strings of symbols and implicitly as a set of rules for the formation of parser trees. The latter aspect can be made explicit by writing the grammar as the definition of a data type '*term*' in e.g. ML. Application nodes will be described by the function $App : term \rightarrow term \rightarrow term$.

Semantically the first argument is a function which is to be applied to the second, yielding the third. The data type can be redefined to include this semantical information by modeling function terms as terms of type $term \rightarrow term$. Application will then be described by the (meta) function $App : (term \rightarrow term) \rightarrow term \rightarrow term$. In the semantical rules applications will be defined by the use of meta-application. The formalized syntax is shown below.

$term : Type$

$true, false : term$
$\mathtt{IF} \ : term \rightarrow term \rightarrow term$
$\mathtt{LET} \ : term \rightarrow (term \rightarrow term) \rightarrow term$
$\Lambda : (term \rightarrow term) \rightarrow term$
$App : term \rightarrow term \rightarrow term$

Some semantical rules:

Rules for IF and LET :

$$\frac{e_1 \Rightarrow true \quad e_2 \Rightarrow \alpha}{(\mathtt{IF}\ e_1\ e_2\ e_3) \Rightarrow \alpha} \qquad \frac{e_1 \Rightarrow false \quad e_3 \Rightarrow \alpha}{(\mathtt{IF}\ e_1\ e_2\ e_3) \Rightarrow \alpha} \qquad \frac{e_1 \Rightarrow \alpha \quad e_2\alpha \Rightarrow \beta}{(\mathtt{LET}\ e_1\ e_2) \Rightarrow \beta}$$

Rules for closure and application

$$\Lambda\, e \Rightarrow \Lambda\, e \qquad \frac{e_1 \Rightarrow \Lambda\, e \quad e_2 \Rightarrow \alpha \quad e\alpha \Rightarrow \beta}{App(e_1\ e_2) \Rightarrow \beta}$$

The above semantical rules can easily be translated into ELF (see section 4).

3 Side effects and natural deduction style semantics

In the previous section we examined two different techniques for constructing natural deduction style semantics without any use of an explicit environment.

The first technique uses the idea that an environment can be seen as a set of hypothetical evaluations. The problem of expressing a sound extension of the environment is solved by mimicking metaparenthesis in the proof tree. That is, a kind of scope of assumptions is introduced. This is done by variable renaming combined with assumption discharging.

The second technique moves the burden of keeping an environment to the metalanguage. Application is simply modelled as application in the metalanguage. Declarations are modelled correspondingly through the standard transformation of '$\mathtt{LET}\ x = M\ \mathtt{IN}\ N$' into '$(\lambda\, x.\ N)M$'.

Both techniques were capable of describing the example language, which was purely functional. However, it should not be very surprising that it is possible to describe such a language in a pure natural deduction style semantics. The real problems lie in the description of languages with assignments and various kinds of side effects. Is it possible to remove the use of an explicit store from a natural deduction style semantics for such languages? The answer is both yes and no: For languages with a certain kind of simple assignments it is possible. But for most languages it is impossible. The problem is that an assignment changes the store and hence might change the world in which the variables are interpreted. In a natural deduction setting we can say that an assignment may change several assumptions.

3.1 Assignments and references

By slightly misusing the term 'assignment', programming languages can be divided into three categories F (functional), A (with assignment) and R (with general references):

F Languages with no assignment except binding of constants to identifiers. This is of course ordinary declaration of constants. In ML syntax this can be expressed as declarations on the form

```
val x=n
```

where n is some expression without occurrences of `ref`.

A Languages of type F extended with variables "containing" values of expressions mentioned above. In addition the languages contain an assignment operator for copying values from a variable to another. In ML syntax these languages have declarations on the form

```
val x=n
val x=ref(n)
```

where n is as above.

R Languages of type A extended with variables "containing" references to other variables. In ML syntax the declaration are on the form

```
val x=n
val x=ref(n)
val x=ref(ref(n))
etc.
```

The three main kinds of parameter passing, 'by value', 'by name' and 'by reference' ('by var' in PASCAL), constitute another important axis of characterization of (assignment) languages. However, a parameter passed 'by name' can be seen as a parameterless inline function passed by value. A semantical apparatus capable to describe 'by value' passing can always be used to describe 'by name'. Hence we will only be interested in the difference between 'by value' and 'by reference'.

Parameter passing 'by reference' is out of the question for purely functional languages, that is, of type F above. From the point of view of denotational semantics there is a major difference between languages of type A without and with parameter passing 'by reference'. In the latter languages there is disciplined form of aliasing, and in their description it is necessary to introduce the well known two step mapping from identifier to value, that is, separate environment and store. The former can be described with a one step mapping from identifier to value. There is no aliasing in such languages. Type R languages have references as first class values, and parameter passing 'by reference' can always be simulated if it is not available.

Parameter passing 'by reference' and the use of reference variables can result in what is known as aliasing. That is, two different identifiers denote the same location in the

store. An assignment of a value to one of the identifiers will also change the value bounded to the other. For type A languages with call 'by reference' the aliasing is a subject to a strict scoping discipline caused by the function bodies. For type R languages there is no such restriction on the aliasing. For a denotational semantics this has no consequences. The use of separate environment and store can cope with this without problems. However, when we now return to the natural deduction style semantics, we will see that the free use of aliasing in type R languages makes it impossible to describe these in a pure natural deduction style without introducing an explicit store in some form.

3.2 Type A languages

The sample language is now extended with an assignment operator and some other sequential features. The language is thought to be of type A although the exact syntax for declarations is not specified for the time being.

Syntax :

$e ::= x \mid$ basic values $\mid$ IF e THEN e ELSE $e \mid (e)e \mid$ LET e IN $e \mid \Lambda x.e \mid [k]e$

$k ::=$ IF e THEN $k \mid$ WHILE e DO $k \mid k;k \mid x := e$

The syntactical category k consists of the sequential part, the commands. The construction $[k]e$ can be read "compute the value of the expression e after the execution of the command k.

Some of the rules in a standard operational semantics are given below.

Rule for IF :

$$\frac{\rho \vdash e_1, s_1 \Rightarrow true, s_3 \qquad \rho \vdash e_2, s_3 \Rightarrow \alpha, s_2}{\rho \vdash \texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3, s_1 \Rightarrow \alpha, s_2}$$

Some rules for the sequential part:

$$\frac{\rho \vdash [\texttt{IF } e_1 \texttt{ THEN } (k;\ \texttt{WHILE } e_1 \texttt{ DO } k)]e_2, s_1 \Rightarrow \alpha, s_2}{\rho \vdash [\texttt{WHILE } e_1 \texttt{ DO } k]e_2, s_1 \Rightarrow \alpha, s_2} \qquad \frac{[k_1]([k_2]e), s_1 \Rightarrow \alpha, s_2}{\rho \vdash [k_1; k_2]e, s_1 \Rightarrow \alpha, s_2}$$

$$\frac{\rho \vdash e_1, s_1 \Rightarrow \alpha, s_3 \qquad \rho \vdash e_2, s_3[x \mapsto \alpha] \Rightarrow \beta, s_2}{\rho \vdash [x := e_1]e_1, s_1 \Rightarrow \beta, s_2}$$

The explicit stores in the rule for IF are not really used, they only indicate the order of execution. Compared with the rules of the substitution semantics in section 2, the above rules are rather messy.

3.2.1 The substitution approach and type A languages

As indicated in [BurHon88] it is possible to give a pure natural deduction style semantics for this language without explicit stores. The idea used in [BurHon88] is simple. A naïve understanding of the expression "$[x := e_1]e_2$" is "bind identifier x to the value of e_1 in e_2", that is, assignment can be seen as a kind of declaration. Some of the new semantical rules are given below.

Rule for IF :

$$\frac{e_1 \Rightarrow \mathit{true} \quad [\cdot e_1 \cdot]e_2 \Rightarrow \alpha}{\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \Rightarrow \alpha}$$

Some rules for the sequential part:

$$\frac{[\texttt{IF } e_1 \texttt{ THEN } (k;\ \texttt{WHILE } e_1 \texttt{ DO } k)]e_2 \Rightarrow \alpha}{[\texttt{WHILE } e_1 \texttt{ DO } k]e_2 \Rightarrow \alpha} \qquad \frac{[k_1]([k_2]e) \Rightarrow \alpha}{[k_1; k_2]e \Rightarrow \alpha}$$

$$\frac{e_1 \Rightarrow \alpha \quad [\cdot e_1 \cdot]([\alpha/x]e_2) \Rightarrow \beta}{[x := e_1]e_1, \Rightarrow \beta}$$

In the above rules the construction $[\cdot e_1 \cdot]e_2$ means "evaluate the expression e_2 after the execution of the sequential pars of e_1". It is necessary to introduce such a construct to ensure that the side effects of e_1,that is, variable name changes, are reflected in e_2 as well. Formally $\cdot e_1 \cdot$ denotes the conversion of e_1 into a command. In a complete semantics it is necessary to introduce a rule for '$\cdot\_\cdot$' for each e-production in the grammar. That is, all the e-rules must be duplicated. One of the new rules is given below.

Rule for IF expression as command:

$$\frac{e_1 \Rightarrow \mathit{true} \quad [\cdot e_1 \cdot\ ;\ \cdot e_2 \cdot]e_4 \Rightarrow \alpha}{[\cdot \texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3 \cdot]e_4 \Rightarrow \alpha}$$

The assignment rule also needs some explanation. We see that the substitution operator occurs in the interior of an expression. This is sound since the operator is a textual entity and does not denote a physical substitution.

It seems evident that the substitution operator cannot be used directly to model type A languages with parameter passing 'by reference', as the following example suggests.

Example 3.1 Extend the above syntax with the production $e ::= \Lambda x : \mathbf{ref}.e$. In the natural deduction style semantics it should be possible to prove that

$$\texttt{LET } y = 3 \texttt{ IN } \ (\Lambda x : \mathbf{ref}.[x := 2](x + y))y \ \Rightarrow \ 4$$

Assume that application is modelled in the same way as before. A refinement proof will proceed as sketched below (in sequent form),

$$\vdash \texttt{LET } y = 3 \texttt{ IN } \ (\Lambda x : \mathbf{ref}.[x := 2](x + y))y \ \Rightarrow \ 4$$

$$y' \Rightarrow 3, x' \Rightarrow y' \vdash [x' := 2](x' + y') \ \Rightarrow \ 4$$

$$y' \Rightarrow 3, x' \Rightarrow y', x'' \Rightarrow 2 \vdash (x'' + y') \Rightarrow 4$$

which is impossible to prove.

It would make no difference if a two step mapping were introduced, see also the discussion of type R languages below.

End of example.

The problem in the above example is caused by the simulated assignment of y to x. Instead of introducing an assumption $x' \Rightarrow y'$ it is possible to textually replace x by y' in the function body. However, only free occurrences of the parameter variable can be replaced by the actual parameter, that is, α-conversion. If this restriction is not introduced, then in the proof of a proposition on the form

$$\texttt{LET } y = 3 \texttt{ IN } (\Lambda x : \texttt{ref}. \ldots(\Lambda x.\Lambda y. \ldots)\ldots)y \Rightarrow n$$

the subexpression $(\Lambda x.\Lambda y. \ldots)$ will be changed into $(\Lambda y.\Lambda y. \ldots)$.

To describe parameter passing 'by reference' the following rule has to be introduced.

$$\frac{M \Rightarrow y \quad [\cdot M \cdot](N[y/x]_\alpha) \Rightarrow n}{(\Lambda x : \texttt{ref}.N)M \Rightarrow n}$$

Here $(N[y/x]_\alpha)$ stands for the expression N' obtained by replacing all free occurrences of the identifier x by y. The definition of 'free occurrence' of an identifier will of course differ from object language to object language. The operator must therefore be tailor-made for each language. In this the operator differs from the substitution operator in [BurHon88]. The above rule differs from the call 'by value' rule in that no assumptions are discharged. We also have to insist that the argument evaluates to a variable.

The replacement of the free occurrences of the formal parameter by the actual one is sound for the following reason: Under the execution of the body of the function, the formal parameter identifier will always be bounded to the same location in the store as the actual parameter identifier. This situation can only be changed by a declaration, in the scope of which the formal parameter identifier no longer is free.

The conclusion is that by adding pure α-conversion to the substitution operator in [BurHon88] it is possible to give natural deduction style semantics without explicit stores for type A languages with parameter passing 'by reference'.

3.2.2 The meta-language approach and type A languages

In [HanMil88] there is no attempt to use the meta-language approach to model languages with assignment and side effects. This can clearly be done by introducing an explicit store in the semantics rules.

The explicit environments were removed by using the implicit environment mechanism of the meta-language, that is, application and declaration were modelled by use of application in the meta-language. It is possible in some sense to remove the stores in a corresponding way. An expression $[x := M]N$ can be modelled as an application $(\lambda x.N)M$. This is insufficient since the side effects in M are ignored. To model the side effects, N must be converted into a function taking all variables that can change value in M as argument. Hence M has to be evaluated to a pair consisting of the value of M and a list of the changed variable values. This tuple is simply a restricted store. In general the meaning of M must be a function taking a tuple of values of the variables changed earlier in the program and producing a pair of value and changed store. The definition of the translation of M into this function will be defined as an operational semantics with explicit store (see section 3.3).

The best way (or least worse way) of removing the explicit stores is by using some of the ideas in [BurHon88]. That is, an α-conversion operator has to be introduced. This operator can be seen as a first order meta-operator expressible in the meta-language. The operator can be implemented in same way as the substitution operator in [BurHon88], see section 4.

Declarations and applications are modelled in the same way as before, while assignment is modelled as in section 3.2.1 above. Some of the new rules are shown below.

Rule for application ('by value'):

$$\frac{e_1 \Rightarrow \Lambda\, e \qquad [\cdot e_1 \cdot]e_2 \Rightarrow \gamma \qquad [\cdot e_1 \cdot; \cdot e_2 \cdot](e\gamma) \Rightarrow \beta}{App(e_1\ e_2) \Rightarrow \beta}$$

Some rules for the sequential part:

$$\frac{\texttt{[IF } e_1 \texttt{ THEN } (k\texttt{; WHILE } e_1 \texttt{ DO } k)]e_2 \Rightarrow \alpha}{\texttt{[WHILE } e_1 \texttt{ DO } k]e_2 \Rightarrow \alpha} \qquad \frac{[k_1]([k_2]e) \Rightarrow \alpha}{[k_1; k_2]e \Rightarrow \alpha}$$

$$\frac{e_1 \Rightarrow \gamma \qquad \forall x'.[\cdot e_1 \cdot](e_2[x'/x]_\alpha) \Rightarrow (\beta[x'/x]_\alpha)}{[x := e_1]e_1, \Rightarrow \beta}$$

In the rule for assignment '$(e_2[x'/x]_\alpha)$' is the expression e_2 with all free occurrences of x replaced by x', correspondingly for '$(\beta[x'/x]_\alpha)$'. Parameter passing 'by reference' is modelled as in the previous section.

The combined method sketched above, has the same limitations as the pure substitution method in [BurHon88].

One might argue that the different way of modelling assignment, and application and declaration makes the semantics somewhat impure. In my opinion, however, assignment is different from the other two mechanisms, and the above semantics reflects this fact.

3.3 Type R languages

The free use of references makes the situation much more complicated for type R languages. There is no tidy system of nested scopes for aliasing. The use of reference assignment can introduce aliasing anywhere in a program, and existing aliasing can be changed by a new reference assignment. It is impossible to use the above α-conversion, because when used the variable name disappears.

An introduction of a two step mapping from identifier to value a lá denotational semantics will not solve the problem.

Example 3.2 Assume that x and y denote the same location r, which in turn 'contains' the value 2. Suppose we have the following situation in a refinement proof (shown on sequent form)

$$y \Rightarrow r, r \Rightarrow 2, x \Rightarrow r \vdash [x := 3]M \Rightarrow m \tag{2}$$

In the substitution/renaming semantics the next two steps will give the situation

$$y \Rightarrow r, r \Rightarrow 2, x \Rightarrow r, x' \Rightarrow r', r' \Rightarrow 3 \vdash M' \Rightarrow m'$$

The assumptions no longer reflect the connection between x and y.
End of example.

The problem in the above example is similar to that in the modeling of parameter passing 'by reference' for type A languages. This is not surprising since such a parameter passing contains a concealed reference assignment. What is needed in the above example is to be able to go from (2) to the situation

$$y \Rightarrow r', r \Rightarrow 2, x \Rightarrow r', r' \Rightarrow 3 \vdash M' \Rightarrow m' \tag{3}$$

Without introducing explicit stores, this problem can only be solved by introducing some metaoperator Φ that for any given r and r' changes all the assumptions on the form $z \Rightarrow r$ to assumptions on the form $z \Rightarrow r'$. Hence we are back in sequent calculus style semantics. In fact, the step from (2) to (3) above corresponds to the application of the rule

$$\frac{\rho \vdash N \Rightarrow n \quad \rho[r \mapsto n] \vdash [\cdot N \cdot]M \Rightarrow m}{\rho \vdash [x := N]M \Rightarrow m}$$

where $\rho[r \mapsto n]$ is defined as in section 2 and ρ is a combined environment and store (with a two step mapping from identifier to value).

The conclusion after this tour through assignment is that the pure metalanguage approach cannot be used to remove stores from the semantic rules. The substitution technique can be used to model type A languages without use of explicit stores. Neither of the techniques can be used to model type R languages without introducing explicit stores. Hence a general framework for implementing/describing operational semantics must have tools for handling stores, e.g. lists.

A reasonable basis for implementing natural deduction style semantics can be either the metalanguage or the substitution approach. But in both cases the evaluation relation '$\Rightarrow$' must be a relation between pairs consisting of a program expression and a store. This fits nicely into Moggi's categorical semantic framework based on monads ([Mog89]). Here programs with side effects are modelled as functions from an initial store to a pair of a value and a final store. The translation map from (abstract) program text to functions (objects in a Kleisli category) corresponds, roughly speaking, to the map obtained by regarding the above evaluation relation as a function in the program expression in the left hand side pair.

4 Implementations

The meta-language semantics and the substitution semantics can, as mentioned before, be implemented in ELF. In this section I will give the implementations of both. However, there is no attempt to implement the extended semantics for type A languages as described in section 3.2. Instead a complete implementation module for stores is presented.

4.1 Semantics for functional features

The meta-language semantics represents a formalization almost equal to the standard way of doing implementations in ELF. The rules can be written directly in ELF. The formalization of the syntax and some of the rules are shown in figure 1.

Most of the rules in the substitution semantics are easy to implement in ELF. In fact, the only rule that causes complications is the rule for the substitution operator:

$$\frac{\begin{array}{c}(x' \Rightarrow n) \\ \vdots \\ M^{x'} \Rightarrow m_{x'}\end{array}}{[n/x]M \Rightarrow m}$$

The rule is highly impure. Some of the side conditions are however easy to implement in ELF. The requirement that x' is a new variable not occurring in other assumptions than $x' \Rightarrow n$ can easily be expressed by use of the universal quantifier and implication:

$$\frac{\forall x'.x' \Rightarrow n \rightarrow M^{x'} \Rightarrow m_{x'}}{[n/x]M \Rightarrow m}$$

The main obstacle is the side condition demanding that all occurrences of x is to be replaced by x' in M. The α-conversion is not a part of the λ-calculus and has to be defined explicitly. The standard way to do this is by converting the term in question into a function term and express the substitution as application. To ensure that all occurrences are replaced, the function term has to satisfy a non occurrence predicate. By using implication and conjunction this can be expressed in the following rule:

$$\frac{x \; noc \; M' \quad \forall x'.(x' \; noc \; x) \rightarrow (x' \Rightarrow n) \rightarrow (M'x' \Rightarrow m'x') \wedge (x' \; noc \; N') \wedge (x' \; noc \; n')}{[n/x]M'x \Rightarrow m'x}$$

In the above rule M' and n' are function terms of type $var \rightarrow exp$. The term $x' \; noc \; m'$ expresses that x' does not occur in M'. The predicate noc must be defined by separate rules. The rule does not require that all occurrences of x in n should be replaced by x'. Hence it is not necessary to include $x \; noc \; n'$.

The use of (higher-order) conjunction makes it impossible to express the above rule directly in ELF, since these higher order features are not a part of ELF. In the Theory of Constructions, however, this would be straight forward. In [BurHon88] the α-logic is introduced to avoid this use of higher order features. Higher order features are used to implement this logic, though.

The substitution semantics is a nice way to describe functional features of a language, but the use of the α-logic to express this semantics is, in my opinion, not a good idea. The α-rules are neither well suited to explain the side condition on the substitution rule nor to be used as a tool in the implementation. It seems best to state the side conditions informally. The above rule can be implemented in ELF by introducing a conditional evaluation judgement in addition to the standard one. This can be seen as a hackers way to solve the problem, but it also reflects the inability of ELF to express substitution directly. The above rule is replaced by two new rules:

$$\frac{x \; noc \; M' \quad \forall x'.(x' \; noc \; x) \rightarrow (x' \Rightarrow m) \rightarrow condeval(x', N', n')}{[n/x]M'x \Rightarrow m'x}$$

$$\frac{x' \; noc \; N' \quad x' \; noc \; n' \quad N'x' \Rightarrow n'x'}{condeval(x', N', n')}$$

The formalization of the syntax and the implementation of some of the rules are shown in figure 2.

Finally, it has to be remarked that an implementation of the pure α-conversion mentioned in connection with assignment (section 3.2) is even more complicated. For this conversion an additional judgement representing the concept of a free variable has to be introduced.

```
[exp:Type];


[if:exp -> exp -> exp -> exp];                          (* Syntax *)
[App:exp -> (exp -> exp)];
[lamb: (exp -> exp) ->exp];
[Let: (exp -> exp) -> exp ->exp];

[eval:exp -> exp ->Type];                              (* Evaluation Judgement *)


[evalif1:                                              (* If true rule *)
{e1,e2,e3,a:exp}(eval e1 true) -> (eval e2 a) ->
(eval (if e1 e2 e3) a)
];

[evalif2:                                              (* If false rule *)
{e1,e2,e3,a:exp}(eval e1 false) -> (eval e3 a) ->
 (eval (if e1 e2 e3) a)
];

[evallamb:                                             (* Closure rule *)
{M:exp -> exp} (eval (lamb M) (lamb M))
];

[evalappl:                                             (* Application rule *)
{e1, e2, a1, a2 :exp}{M:exp->exp}
(eval e1 (lamb M)) -> (eval e2 a2) -> (eval (M a2)  a1) ->
(eval (app e1 e2) a1)
```

Figure 1: The ELF implementation of the meta-language semantics, the formalized syntax and some rules

```
[exp,var:Type];


[if:exp -> exp -> exp -> exp];                    (* Syntax *)
[App:exp -> exp -> exp];
[lambda: var -> exp ->exp];
[Let: var -> exp -> exp -> exp];
[Sub: exp -> var -> exp -> exp];
[Dot: exp -> exp -> exp];
[ive:var ->exp]                                  (* Embedding of var into exp *)


[eval:exp -> exp ->Type];                        (* Evaluation Judgement  *)
[condeval:var->(var->exp)->(var->exp)->Type];    (* Judgement used in the *)
                                                 (* Substitution rule     *)
[noc:var -> (exp -> exp) -> Type];               (* Judgements for non    *)
[noce:var -> exp -> Type];                       (* Eccurrence relation   *)


[subst:                                          (* Substitution rule *)
{x:var}{m:exp}{N',n':var -> exp}
(noc x N') ->
({x':var} (noce x' (ive x)) -> (eval x' m) -> (condeval x' N' n')) ->
(eval (Sub x m N'x) n'x)
];

[cond:                                           (* Auxiliary rule for subst. *)
{x':var}{N',n':var -> exp}
(noc x' N') -> (noc x' n') -> (eval N'x' n'x') ->
(condeval x' N' n')
];

[LetEval:                                        (* Rule for Let *)
{x:var}{M,N,m,n:exp}
(eval M m) -> (eval (Sub m x N)  n) ->
(eval (Let x M N) n)
];
```

Figure 2: The ELF implementation of the substitution semantics

4.2 The semantical approaches and implementations of functional languages

The most central part of an implementation of λ-calculus and functional programming languages is the treatment of application. Application has to implemented by the use of substitution in some form. As pointed out in [Cur88] "substitution is not a magic, meta-level, one-step operation, but a process which has to be carried along the structure". Curien also divides, according to the treatment of application, the implementation of functional languages into two groups, the graph reduction machines and the environment machines. In the former the environment is baked into the graph. In the latter the environment is kept separate from the program to be evaluated and is accessed as a database.

The ELF implementations make it clear that the two approaches are based on these two different implementation ideas. The meta-language semantics is based on graph reduction. The formalized syntax indicates how the object programs are transformed into a structure of combinators (meta-language functions). Application is modelled by graph reduction (application in the meta-language). All the complexity of the graph reduction is hidden and in this approach the substitution operator is indeed a one step meta-level operator. This, in my opinion, is not a bad thing. It is not desirable that semantics should express all the bookkeeping needed in an implementation.

The substitution semantics is a true representative for the environment approach adapted to implementation in a system like ELF. There is no substitution but renaming of variables. All applications are broken down to basic computations represented by δ-rules. The environment is implemented by the use of the environment mechanism in the meta-language and is for this reason invisible in the semantical rules. However, to avoid name clashes α-conversion must be used in an environment approach. Renaming is only a meta level operator in λ-calculus. Hence a renaming operator must be defined explicitly. This must be done for each new language and is the main drawback of the substitution semantics. However, by the use of tactics this need not complicate the actual use of the implemented semantics. But it requires more initial work for each new language.

The semantics can be seen as logics. The above implementations are done in a polymorphic λ-calculus. A proof will produce a term of the type corresponding to the theorem. What kind of information does such a term contain? Is it possible to use the information in the proof terms to construct, say, a kind of compiler compiler? The theorems state facts about evaluations of program terms and the proof terms will in fact describe the execution of the program. That is, if we want to show that $M \Rightarrow m$ the proof corresponds to the execution of M. If M is a non terminating program, the proving process will never end. Hence the proof term itself only describes the execution history of the program. However, each step in the proving process adds something to this history. One might say that the prover is the executer of the program. A complete set of automatic tactics will therefore be an interpreter for the language in question.

4.3 Stores, assignment and reference expressions

The necessary features for describing all kinds of assignment languages constitute a module which can be defined independently of the choice of semantical approach for the

functional part.

A store is some structure in which locations are bound to values. A store can be represented as a function, as a list or as any other appropriate structure. There are three common operations needed to manipulate a store:

- *update* — binds a new value to a given location.
- *get* — produces the value bound to a given location.
- *new* — produces a new unused location.

A store will be modelled as a list of pairs consisting of a location and a value, e.g.

$$s = [(r_1, v_1), \ldots, (r_m, v_m)]$$

The term $s[r \mapsto n]$ will in the following mean "*update* s". This will be implemented by simply appending the pair (r, n) to the list s. It is of course possible to implement a physical replacement, but this will not simplify the structures needed and will definitely not simplify the use of the semantics.

The *get* operator must be defined according to the implementation of the *update* operator:

$$get(s[r_1 \mapsto n], r_2) = if\ r_1 = r_2\ then\ n\ else\ get(s, r_2)$$

This operator is not defined for the empty list. This corresponds to the fact that in the ELF implementation the type (*get empty r n*) is empty for all r and n.

The *new* operator is implemented by the use of a non occurrence predicate *noc*. r *noc* s is true if there is no term on the form (r, n) in the list s. The ELF implementation of stores is shown (in Lego syntax) in figure 3.

The lack of a deallocation operator implies that for any program the initial store will always be an initial segment of the final store. In this way the final store will contain the assignment history of the program.

In the substitution semantics the instantiated variable identifiers have a scope limited by the subtree with the corresponding substitution operator expression as root:

$$\frac{M \Rightarrow m \qquad \dfrac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ N^{x'} \Rightarrow n_{x'}\end{array}}{[n/x]N \Rightarrow n}}{[x := M]N \Rightarrow n}$$

This makes it possible to use the universal quantifier to bind x' in the rule for "$[n/x]N \Rightarrow n$" (see above).

Unfortunately, the locations do not have such a nice scope property. The scope of a location will always be from the point of the allocation to the end of the program

$$\frac{(N, s_1) \Rightarrow (n, s_2) \quad r\ noc\ s_2 \quad (M, s_2[r \mapsto n]) \Rightarrow (m, s_2)}{(\texttt{LET}\ x = \texttt{ref}(N)\ \texttt{IN}\ M,\ s_1) \Rightarrow (m, s_2)}$$

In the above simplified rule for variable declaration r will occur in all stores in the right subtree as well as in s_3. This will be the case for every location allocated in the left subtree. Hence the locations can only be bounded on the outermost level. In any practical use of the semantics it seems necessary to declare the necessary amount of locations beforehand. The non occurrence relation must also be predefined for the locations in question.

It has to be remarked that even with a deallocation operator it would be necessary for many languages to explicitly define the needed locations.

In programming languages there are three main operations on the store: assignment, constructing a pointer (reference) to a value and value retrieval. The ML syntax for the three operators is

$$M := N$$
$$\mathbf{ref}(N)$$
$$!M$$

where M has to be of type $ref(\cdots)$ and n can be of any type. In our environmentless setting the dynamic semantics will be as follows:

Assignment :

$$\frac{(L, s_1) \Rightarrow (r, s_2) \quad (M, s_2) \Rightarrow (m, s_3) \quad (N, s_3[r \mapsto m]) \Rightarrow (n, s_4)}{([L := M]N, s_1) \Rightarrow (n, s_4)}$$

Pointer construction :

$$\frac{(N, s_1) \Rightarrow (n, s_2) \quad r \; noc \; s_2}{(\mathbf{ref}(N), s_1) \Rightarrow (r, s_2[r \mapsto n])}$$

Value retrieval :

$$\frac{(M, s_1) \Rightarrow (r, s_2) \quad get(s_2, r) = m}{(!M, s_1) \Rightarrow (m, s_2)}$$

5 Conclusion

In this paper we have seen three methods for construction operational semantical rules — the standard, the substitution based and the meta-language methods.

It is not shown how the standard method can be implemented in ELF, but this is straight forward. The environments are represented in the same way as the stores, and the turn stile is regarded as a two place predicate or combined with $\Rightarrow$ to a three (five) place predicate.

The two other methods are more interesting in the fact that here the explicit use of environment is sought removed from the rules. Both methods are capable to describe purely functional programming languages without use of explicit environments in the semantical rules. It is only the meta-language approach though that fully gets rid of the environment by transforming the object programs into a set of combinators. The substitution operator approach removes the environments from the semantical rules by viewing the environment as assumptions in a natural deduction system. As said above the methods represent respectively the two main ways of implementing functional languages.

The metalevel approach is the easiest of the two to implement in ELF. However, the substitution approach can be as easy to use if some work is invested in writing tactics for

Figure 3: ELF implementation of stores (in Lego syntax)

```
[Loc,Map:Type];                    (* Exp:Type declared in the functional module*)

[empty:Store];                     (* Implementation of stores as lists        *)
[ext:Store->Map->Store];           (* update is modelled by ext                 *)
[map:Loc->Exp->Map];
[ire:Loc->Exp];

[noc:Loc->Store->Type];            (* New location is modelled by noc and nec   *)
[nec:Loc->Loc->Type];
[get:Store -> Loc->Exp->Type];     (* The value 'in' a location is modelled     *)
                                   (* by the get Judgement                      *)

[get1:{s:Store}{r:Loc}{e:Exp} get (ext s (map r e)) r e];

[get2:{s:Store}{r1,r2:Loc}{e1,e2:Exp} (get s r1 e2) -> (nec r1 r2) ->
                                      (get (ext s (map r1 e1))  r2  e2)];

[necswap:{r1,r2:Loc} (nec r1 r2) -> (nec r2 r1)];
[nectrans:{r1,r2,r3:Loc} (nec r1 r2) -> (nec r2 r3) -> (nec r1 r2)];

[nocempty:{r:Loc} noc r empty];
[noccomp:{r1,r2:Loc}{s:Store}{e:Exp} (noc r1 s) -> (nec r1 r2) ->
                                     (noc r1 (ext s (map r2 e)))]
```

Figure 4: The rules for assignment and reference expressions in ELF (Lego syntax)

```
[evalassign:
{L,N,M,n,m|exp}{s1,s2,s3,s4|Store}{r:Loc}
(eval (pair L s1) (pair (ire r) s2))  ->
(eval (pair M s2) (pair m s3))  ->
n(eval (pair N (ext s3 (map r m))) (pair n s4))->
(eval (pair (PREFIX (ASSIGN L M) N) s1) (pair n s4))
]

[evalREF:
{N,n,m|exp}{s1,s2,s3|Store}{r:Loc}
(eval (pair N s1) (pair n s2)) -> (noc r s2) ->
(eval (pair(ire r) (ext s2 (map r n))) (pair m s3)) ->
(eval (pair (REF N) s1) (pair m s3))
]

[evalGET:
{N,n|exp}{s1,s2|Store}{r|Loc}
(eval (pair N s1) (pair (ire r) s2)) -> (get s2 r n)
-> (eval (pair (GET N) s1) (pair n s2))
]
```

the necessary book-keeping of non occurrence of variables. The substitution approach is clearly best for manual use.

Neither of the methods can be used to describe programming languages with side-effects without using explicit stores, although the substitution approach can be used for languages with a very simple assignment system, that is, without pointers. In fact, this is the class of languages it is possible to describe by Hoare-logic, at least in a not too complicated way (see [Dahl86]). The introduction of pointers and sharing represents a big increase in complexity both in many formal descriptions and in implementation of programming languages.

6 Acknowledgements

I would like to thank Randy Pollack for introducing me to his Lego system and to the work of J.Hannah and D.Miller. I would also like to thank Furio Honsell for introducing me to the substitution semantics and finally Rod Burstall for many interesting discussions and useful comments.

References

[Avr87] Avron, A., Honsell, F., Mason, I.
Using Typed Lambda Calculus to Implement Formal Systems on a Machine
ECS-LFCS-87-31, University of Edinburgh 1987

[deBruijn72] de Bruijn, N.
Lambda Calculus Notation with Nameless Dummies
Indag. Math. 34(5), pp. 381-393, 1972

[BurHon88] Burstall, R. & Honsell, F.
A Natural Deduction treatment of Operational Semantics
Proc. 8th Conf. on Foundations of Software Technology, Pune, Springer LNCS 1988

[BurHon88b] Burstall, R. & Honsell, F.
A Natural Deduction treatment of Operational Semantics
Report ECS-LFCS-88-69 University of Edinburgh, 1988
(the same as the one above, but with some corrections)

[CoqHuet88] Coquand, Th., Huet, G.
The Calculus of Constructions
Information and Computation, 2/3, vol.76 , 1988

[Cur88] Curien, P.-L.
The $\lambda\rho$-calculus: an Abstract Framework for Environment machines
Preprint, October 1988

[Dahl86] Dahl, O.-J., Langmyhr, D., Owe, O.
Preliminary Report on the Specification and Programming Language ABEL
Research Report, Dept. of Comp. Sc. University of Oslo 1986

[HanMil88] Hannan, J. & Miller, D.
Enriching a Meta-Language with Higher-Order Features
In proceedings of the Workshop on Meta-Programming in Logic programming, Bristol, June 1988, to appear

[Har87] Harper, R. Honsell, F., Plotkin, G.
A Framwork for Defining Logics
In Proceedings of LICS 87, Cornell 1987

[Kahn87] Kahn, G.
Natural Semantics
Report de Recherche 601, INRIA 1987

[Mog89] Moggi E.
Computational lambda-calculus and monads
In Proceedings of LICS 89, to appear

[Plot81] Plotkin, G.
A Structural Approach to Operational Semantics
DAIMI FN-19, Aarhus University, 1981

[Pol88] Pollack, R.
Lego...
Unpublished manuscript, Edinburgh 1988

[Praw65] Prawitz, D.
Natural Deduction, A Proof-Theoretical Study
Almqvist&Wiksell, Stockholm 1965