LFCS

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

A Natural Deduction treatment of Operational Semantics

A Natural Deduction treatment of Operational Semantics

by

Rod Burstall and Furio Honsell

ECS-LFCS-88-69
(also published as CSR-282-88)

LFCS Report Series

November 1988

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

A Natural Deduction treatment of Operational Semantics[1]

Rod Burstall[2] and Furio Honsell[3]
University of Edinburgh and University of Turin

Abstract
We show how Natural Deduction extended with two replacement operators can provide a framework for defining programming languages, a framework which is more expressive than the usual Operational Semantics presentation in that it permits hypothetical premises. This allows us to do without an explicit environment and store. Instead we use the hypothetical premises to make assumptions about the values of variables. We define the extended Natural Deduction logic using the Edinburgh Logical Framework.

1. Introduction

The Edinburgh Logical Framework (ELF) provides a formalism for defining Natural Deduction style logics [Harper et al. 1987]. Natural Deduction is rather more powerful than the notation which is commonly used to define programming languages in "inference-style" Operational Semantics, following Plotkin [1981] and others, for example Kahn [1987]. So one may ask

> "Can a Natural Deduction style be used with advantage to define programming languages?".

We show here that, with a slight extension, it can, and hence that the ELF can be used as a formal meta-language for defining programming languages. However ELF employs the "judgements as types" paradigm and takes the form of a higher order typed lambda calculus. We do not need all this power here, and in this paper we present a slight extension of Natural Deduction, simply typed, as a semantic notation for programming language definition. This extension can itself be defined in ELF.

The inspiration for using a meta-logic for Natural Deduction proofs comes from Martin-Löf. Our work benefited from that of Mason [1987] who did a proof system for Hoare logic in the ELF, encountering problems about the treatment of program variables. In particular he adopted the non-interference relation originally used by Reynolds [1978].

The main feature of Natural Deduction proofs, used in our semantics but not used in the usual style of Operational Semantics, is that the premises of a rule may be hypothetical: they may themselves be of the form "Q is derivable from P_1 and ... and P_n". We write a premise of this form thus

$$\begin{array}{c} (P_1, \ldots, P_n) \\ \vdots \\ Q \end{array}$$

$$R$$

---

[1] To appear as invited paper in Proc. 8th Conf. on Foundations of Software Technology, Pune, Springer LNCS. This is a revised version with some corrections. The main changes are (i) the rules for applying a closure or a primitive (ii) the example proofs (iii) removal of explicit equality in alpha logic (iv) the rule of substitution uses α.

[2] Dept of Computer Science, Edinburgh University, JCMB, King'sBdgs., Mayfield Rd., Edinburgh EH9 3JZ
[3] Dipartimento di Informatica, Universita di Torino, Corso Svizzeria 185, Torino, Italy

Using these techniques we are able to give an operational semantics which dispenses with the traditional notions of environment and store. This makes our semantic definitions and our proofs of evaluations appreciably simpler than the traditional ones. Our proofs are the same shape as the traditional ones, but each formula is simpler; the environment and store do not appear repeatedly in formulas as they do in the traditional system. Instead of environment and store we use the notions of expression with an attached substitution and evaluation of an expression after a command (compare Dynamic Logic). Instead of evaluating an expression M with respect to an environment we consider its value given some assumptions about the values of the variables which occur in it.

The main technical difficulty is to define the substitution operation; this plays a crucial role in our semantics. We need substitution both for expressions and for commands. This is not a textual substitution; it depends on the binding operators in the language being defined. The meaning of [m/x]M (substitute value m for identifier x in expression M) is that when we come to evaluate M we assume the value of x to be m and ignore any previous assumptions about the value of x. To express this we use the hypothetical premises of Natural Deduction, with side conditions to ensure that we have a fresh variable. However we are not able to express substitution purely in a Natural Deduction logic. The difficulty can be reduced to substituting a (new) identifier for an (old) one. Thus we have to add a primitive operator for identifier replacement, just textual replacement, which we call α. (It is named from α conversion in lambda calculus.) We define this with a special rule schema. We also need an "dual" to α which we call $\underline{\alpha}$.

In using an editor or a programming language you probably learn from a "Users' Manual" and then, having got some experience, look up the fine points in a "Reference Manual". For our style of doing operational semantics the "Reference Manual" is a formal description in ELF. We try to provide also a less formal "Users' Manual" which describes the form of the semantic rules and the criteria for proofs; this takes the form of "Alpha Logic". Alpha Logic will be described with about the degree of precision with which the rules of predicate logic might be defined in a text book. As usual we will give the Users' Manual first, asking the reader to suspend critical judgement while getting an intuitive feel for the style of semantics we propose.

We proceed in four steps.

- We define Alpha Logic: Natural Deduction extended with operators α and $\underline{\alpha}$ to replace variables.
- We specialise this to Evaluation Logic, by using Alpha Logic with evaluation predicates, =>, and substitution operators, [/], obeying suitable rules.
- We use Evaluation Logic to define the semantic rules for a sample programming language which features lambda expressions, commands, procedures and expressions with side effects.
- We give a formal definition of Alpha Logic in the Edinburgh Logical Framework.

This last step puts Alpha Logic on a firm foundation and shows the connection of our approach with the ELF treatment of logical languages.

Comparison with the Edinburgh Logical Framework

Our original aim was to use the ELF as a definition medium for Programming Languages. We achieve this in the last step above. Our semantic rules can be written in ELF notation by mere transliteration. However the rules do not use all the power of the ELF except in defining the substitution operations. It seemed to us better to present Evaluation Logic as a simpler framework for semantics. It is not higher order and does not use dependent types, instead it makes use of the primitive operators α and $\underline{\alpha}$. The semantic rules do not even mention α or $\underline{\alpha}$, once we have defined substitution. In short

ELF approach: If you understand lambda calculus with dependent types you can define many logics formally.

Alpha Logic approach: If you understand Natural Deduction with variable replacement you can define many programming languages formally.

In alpha logic we lose the ELF advantage of a built-in type checker for dependent types, so that we have to write a separate static semantics (not treated in this paper). Even in ELF we may have to write a static semantics, for example to handle phenomena such as the polymorphism in ML.

Acknowledgements

We thank the UK Science and Engineering Research Council for support (RB and FH) and the Italian Ministry of Public Instruction for support (FH). Much of the work was carried out while FH was at Edinburgh University, and we thank our colleagues at the Laboratory for Foundations of Computer Science for stimulating interaction. We would like to thank Randy Pollack who coded our semantics in his LEGO ELF system and thereby brought to light two errors in our earlier version of the paper. RB also thanks Butler Lampson of DECSRC for experience gained in their joint work on the semantics of the Pebble language, and he thanks the Conference Programme Committee for the invitation to present a paper.

2. Definition of a Framework for Operational Semantics

Alpha logic is about syntactic entities, namely identifiers and expressions.

2.1.Alpha logic

Let Σ be a first order signature with two sorts, Id (identifiers) and Expr (expressions), some predicates P and some function symbols F. We introduce two distinguished function symbols α, $\underline{\alpha}$: Id * Id * Expr -> Expr; α denotes replacement of all occurrences of an identifier in a term by another identifier, whilst $\underline{\alpha}$ is "dual" to it.

We define Terms and (atomic) Formulas over the signature and variables as usual and Rules as in Natural Deduction.

An alpha logic over a signature Σ has the sorts Id and Expr with the distinguished function symbols α and $\underline{\alpha}$ subject to a set of rules $\mathbf{R} \cup \underline{\mathbf{R}}$ which we now define.

An element of **R** is a well formed instance of the following schema:

$$\frac{C[\,J\,]}{C[\,\alpha_{j/i}\,I\,]}$$

where C[ ] is any context (formula with a hole) and where J is the term obtained from the term I by replacing all occurrences of identifier i with j.

Thus to prove $\alpha_{j/i}(i+k) < 3$ we have to prove $j+k < 3$.

An element of $\underline{\mathbf{R}}$ is a well formed instance of the following schema:

$$\frac{C[\,J\,]}{C[\,\underline{\alpha}_{j/i}\,I\,]}$$

where C[ ] is any context (formula with a hole) and where I is the term obtained from the term J by replacing all occurrences of identifier j with i.

Thus if we have proved $i+j < 3$ we may conclude $\underline{\alpha}_{j/i}(i+i) < 3$. (We may reach the same conclusion from $i+i < 3$ or $j+i < 3$ or $j+j < 3$.) The reader may like to think of $\underline{\alpha}_{j/i}$ as dual to $\alpha_{i/j}$.

A proof in alpha logic is a natural deduction proof (Prawitz 1966). Later we will give a formal account of the notion of alpha proof in the Edinburgh Logical Framework.

More generally a "many sorted alpha logic" has sorts Id_{S1}, ..., Id_{Sn} and $Expr_{t1}$, ..., $Expr_{tn}$ with function symbols α_{st}, $\underline{\alpha}_{st}$: $Id_S * Id_S * Expr_t \rightarrow Expr_t$.

2.2.Evaluation logic - 1

We now specialise the alpha logic to "evaluation logic". We consider first the case with two sorts Id (identifiers) and Expr (expressions). An evaluation logic is an alpha logic whose signature contains two distinguished predicates, a distinguished function and certain rules which they obey.

The distinguished predicates are evaluation and value
_ => _ over Expr * Expr
Value _ over Expr

The distinguished functions are expr (to convert Id to expr) and substitution
expr : Id -> Expr
[_ / _] : Expr * Id * Expr -> Expr

For ease of reading we will usually omit the conversion function expr hereafter, treating Id as a subsort of Expr.

We use the following symbols (possibly primed)
x, y, z for identifiers
M, N, m, n, p for expressions

There are two rules are, first

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ \text{value } n, \qquad \alpha_{x'/x}\, M \Rightarrow \alpha_{x'/x}\, m\end{array}}{[n/x]\, M \Rightarrow m} \quad x' \text{ is a new variable}$$

By "x' is a new variable" we mean that is it does not occur in n, x, M and m nor does it occur in any assumption except those of the form x' => n ; such assumptions are discharged by an application of this rule.

This rule is really no more complex than the usual rule for, say, existential elimination.

The second rule is

$$\frac{\text{value } n}{n \Rightarrow n}$$

The rule for substitution is the key definition. It encapsulates the way we handle object language variables. One of the virtues of the Natural Deduction framework used in evaluation logic is that it provides uniformity in handling scopes of variables, avoiding side conditions on rules. This is important in practice because handling variables is a traditional pitfall in defining logics or programming languages. Our approach is to define substitution in terms of evaluation.

A naive version of the substitution rule might have been

$$\frac{\begin{array}{c}(x \Rightarrow n)\\ \vdots \\ \text{value } n, \qquad M \Rightarrow m\end{array}}{[n/x]\, M \Rightarrow m}$$

but the evaluation of M in the hypothetical premise could make use not only of x => n, but also of any other statement about the value of x in the context in which the rule is used; we might be evaluating [n/x] M in a context in which x => n' and this should not affect the value. Thus we need to introduce a new variable x' local to the hypothetical premise. We have to replace x by x' in M in the right subproof. Just in case the resulting value m in the subproof contains an x' we have to replace the x' by an x (This will occur in our example language only for closures).

Syntactic sugar
We will mostly use infix syntax for expressions, e.g.
let x = M in N for let(x, M, N)
When we introduce a syntax for a particular object language in what follows this is to be understood as syntactic sugar.

Example proof
We now give a proof in alpha-logic with the substitution rule above,
constants 1,2,3, an operation + and three extra rules
M => 1, N => 2

$$\frac{}{M+N \Rightarrow 3} \qquad \frac{}{\text{value } 1} \qquad \frac{}{\text{value } 2}$$

We will show that [1/x][2/y] x+y => 3

$$\dfrac{\text{value } 1 \qquad \dfrac{\text{value } 2 \qquad \dfrac{\dfrac{[x' \Rightarrow 1]_{(1)},\ [y' \Rightarrow 2]_{(2)}}{x'+y' \Rightarrow 3}}{\alpha_{y'/y}\, x'+y \Rightarrow \underline{\alpha}_{y'/y}\, 3}}{[2/y]\, x'+y \Rightarrow 3}\ (2)\ y'}{[1/x][2/y]\, x+y \Rightarrow 3}\ (1)\ x'$$

The square brackets enclose hypotheses which are discharged at the level indicated by the subscript. The scope of a variable is shown by writing it at the end of the line below the scope.

We could write this proof more briefly omitting the applications of the α and $\underline{\alpha}$ rules. We may think of the α expression brought in by the substitution rule being immediately reduced. A proof editor could do this step automatically. We recommend this style of proof display. Introducing α and $\underline{\alpha}$ is a technical device for making the machinery of substitution explicit. They do not appear in the semantic rules once we have defined substitution, and they can well be omitted from the proofs.

$$\dfrac{\text{value } 1 \qquad \dfrac{\text{value } 2 \qquad \dfrac{[x' \Rightarrow 1]_{(1)},\ [y' \Rightarrow 2]_{(2)}}{x'+y' \Rightarrow 3}}{[2/y]\, x'+y \Rightarrow 3}\ (2)\ y'}{[1/x][2/y]\, x+y \Rightarrow 3}\ (1)\ x'$$

We have treated the case with one sort Id and one sort Expr, in a many sorted case these would be indexed families Id_S and $Expr_t$. The distinguished functions and predicates, when required, would be indexed accordingly.

2.3. Conventions for syntax

To extend the evaluation logic signature (Id, Expr, => value and [/]) we will use the usual conventions of syntax. These allow us to introduce function symbols together with the infix notation for them. We will use the names allocated to schematic variables for the syntax classes. For example using

x, y, z for identifiers
M, N, m, n, p for expressions

the syntax definition

M ::= K | **let** x = M **in** N | **lambda** x . M

introduces the new function symbols

K: Expr
let: Id * Expr * Expr -> Expr
lambda: Id * Expr -> Expr

3. Example semantics

3.1. A basic functional language

In this section we will give the semantics for a simple functional language as a signature and a set of rules in evaluation logic.

Signature

We use a signature with sorts Id and Expr.
We use the following symbols (possibly primed)
x, y, z for identifiers
M, N, m, n, p, f, k for expressions
M ::= 0 | succ | plus | Y | ... | M N | **let** x = M in N | **lambda** x . M | m.n
To be explicit, M N means apply(M, N). We explain m.n below.

We need a new unary predicate over expressions
closed M - (informally) M has no free variables, except ones assumed to be closed.

The evaluation of a **let** and the application of a **lambda** expression (assuming call by value) are formulated in terms of our substitution operation.

Since our language allows a lambda expression to appear as the result of evaluating an expression, we must ensure that it carries with it enough information about the values of its free variables to enable it to be applied in any context, so we use Landin's notion of "closure" (Landin [1964]). Our Natural Deduction technique really corresponds to stack discipline and we cannot expect it to handle function values without some further device. We do not have an explicit environment, instead we give rules which successively bind the variables appearing in the body until the body is "closed". We have to define closed, essentially by induction on the syntax. This means that for each syntactic constructor (let, lambda and so on) we need not only a rule for => but also a rule for closed. The predicate closed conveys the binding nature of the operators; it really belongs to the static semantics.

We have to give rules for the application of closures and primitive functions to argument values. For this we introduce a new syntactic construct "m.n" for such applications with appropriate rules. These rules allow us to take a closure to pieces and apply its body in the correct context .

An illustrative proof follows the rules.

Evaluation rules

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

$$\frac{M \Rightarrow m, \quad N \Rightarrow n, \quad m.n \Rightarrow p}{MN \Rightarrow p}$$

$$\frac{\text{value } n, \quad [p/y]\,(f\,.\,n) \Rightarrow m}{([p/y]\,f)\,.\,n \Rightarrow m}$$

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

Example of "delta rules" for primitive functions :-

$$\frac{}{\text{value } 0}$$

$$\frac{}{\text{value succ}} \qquad \frac{\text{value } n}{\text{succ}\,.\,n \Rightarrow \text{succ}\,.\,n}$$

$$\frac{}{\text{value plus}} \qquad \frac{\text{value } m}{\text{plus}\,.\,m \Rightarrow \text{plus}\,.\,m}$$

$$\frac{\text{value } n}{(\text{plus}\,.\,0)\,.\,n \Rightarrow n} \qquad \frac{(\text{plus}\,.\,m)\,.\,n \Rightarrow p}{(\text{plus}\,.\,(\text{succ}\,.\,m))\,.\,n \Rightarrow \text{succ}\,.\,p}$$

$$\frac{}{\text{value } Y}$$

$$\frac{\text{value } f}{Y\,.\,f \Rightarrow Y\,.\,f} \qquad \frac{(f\,.\,(Y\,.\,f))\,.\,m \Rightarrow n}{(Y\,.\,f)\,.\,m \Rightarrow n}$$

Rules for evaluating lambda expressions to closures :-

$$\frac{\begin{array}{c}(\text{closed } x)\\ \vdots\\ \text{closed } M\end{array}}{\textbf{lambda } x.\ M \Rightarrow (\textbf{lambda } x.\ M)}$$

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

The reader might like to compare these with the usual treatment using an explicit environment, ρ . For example

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

$$\frac{\rho \vdash M \Rightarrow m, \quad \rho \vdash N \Rightarrow n, \quad m.n \Rightarrow p}{\rho \vdash M\,N \Rightarrow p}$$

In the latter rule the environment is not used; it is simply passed down. This is implicit in out Natural Deduction formulation. We only mention the environment when it is used.

Rules for closed

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

$$\frac{}{\text{closed } 0} \quad \text{and similarly for the other constants}$$

$$\frac{\text{closed } M, \quad \text{closed } N}{\text{closed}(M\,N)}$$

$$\frac{\text{closed } m, \quad \text{closed } n}{\text{closed}(m.n)}$$

$$\frac{\begin{array}{c}(\text{closed } x)\\ \vdots\\ \text{closed } N, \qquad \text{closed } M\end{array}}{\text{closed}(\textbf{let } x = N \textbf{ in } M)}$$

$$\frac{\begin{array}{c}(\text{closed } x)\\ \vdots\\ \text{closed } M\end{array}}{\text{closed}(\textbf{lambda } x.\ M)}$$

Rule for value

$$\frac{M \Rightarrow m}{\text{value } m}$$

Example of evaluation

As an example of proofs using these rules we evaluate
(**let** y = 2 in **lambda** x.x+y) 1
We use two Lemmas, A and B, to simplify the layout. They are proved below. We will omit the proofs of such exciting facts as "value 1" and "1+2 => 3".

Main evaluation proof

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{[y' \Rightarrow 2]_{(1)}}{A}}{\text{value } 2, \quad \textbf{lambda } x.x+y' \Rightarrow [2/y'](\textbf{lambda } x.x+y')}\,(1)}{2 \Rightarrow 2, \ [2/y](\textbf{lambda } x.x+y) \Rightarrow [2/y](\textbf{lambda } x.x+y)} \qquad B}{\textbf{let } y=2 \textbf{ in } \textbf{lambda } x.x+y \Rightarrow [2/y](\textbf{lambda } x.x+y), \quad 1 \Rightarrow 1, \quad ([2/y]\textbf{lambda } x.x+y).1 \Rightarrow 3}}{(\textbf{let } y = 2 \textbf{ in } \textbf{lambda } x.x+y)\ 1 \Rightarrow 3}$$

Lemma A

Show that if y' => 2 then (**lambda** x.x+y') => [2/y'](**lambda** x.x+y')

$$\frac{\dfrac{\dfrac{[\text{closed } (y')]_{(4)}, \ [\text{closed}(x)]_{(5)}}{\text{closed } (x+y')}\,(5)}{y' \Rightarrow 2, \quad \textbf{lambda } x.x+y' \Rightarrow \textbf{lambda } x.x+y'}\,(4)}{\textbf{lambda } x.x+y' \Rightarrow [2/y'](\textbf{lambda } x.x+y')}$$

Lemma B

Show ([2/y]**lambda** x.x+y) . 1 => 3

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{value } 1,\ [x' \Rightarrow 1]_{(1)},\ [y' \Rightarrow 2]_{(1)}}}{\text{value } 1,\ x'+y' \Rightarrow 3}}{\text{value } 1,\ [1/x](x+y') \Rightarrow 3}\ (1)}{\text{value } 2,\ (\textbf{lambda}\ x.x+y') . 1 \Rightarrow 3}$$

$$\dfrac{\text{value } 1,\ [2/y]((\textbf{lambda}\ x.x+y) . 1) \Rightarrow 3}{([2/y]\text{lambda}\ \mathbf{x.x+y)\ .\ 1\ \Rightarrow\ 3}}\ (2)$$

3.2. Complex declarations

We now consider allowing **let** to be followed by a complex declaration, formed with "**and**" (parallel declarations) and ";" (sequential declarations). Such declarations will form a new syntax class and will need to have as values just the environments which we have otherwise succeeded in eliminating. This seems unavoidable for languages, such as Standard ML, which permit such a declaration feature. However the "environments" only appear as values (declaration values).

Signature

We extend the previous signature as follows.
We introduce a new Expr-like sort Declaration with a new evaluation predicate =>.
We use the following symbols
R, S,r,s for declarations
R::= x=M | R **and** S | R; S
We generalise the syntax class Expr, introducing a new **let** and {}:
M ::= **let** R **in** N| {r}M
Finally we introduce the new rules for => and closed.

Evaluation rules

$$\frac{M \Rightarrow m}{x = M \Rightarrow m/x}$$

$$\frac{R \Rightarrow r,\ S \Rightarrow s}{R\ \textbf{and}\ S \Rightarrow r\ \textbf{and}\ s}$$

$$\frac{R \Rightarrow r,\ \{r\}S \Rightarrow s}{R\ ;\ S \Rightarrow r\ \textbf{and}\ s}$$

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

$$\frac{R \Rightarrow r, \quad \{r\}M \Rightarrow m}{\textbf{let } R \textbf{ in } M \Rightarrow m}$$

$$\frac{\{r\}(\{s\}M) \Rightarrow m}{\{r \textbf{ and } s\}M \Rightarrow m}$$

Rules for closed

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

$$\frac{R \Rightarrow r, \quad closed(\{r\}M)}{closed(\textbf{let } R \textbf{ in } M)}$$

$$\frac{closed\ (\{r\}(\{s\}M))}{closed\ (\{r \textbf{ and } s\}M}$$

3.3. An assignment language

Signature

We extend the previous signature as follows.
We introduce a new Expr-like sort Command.
We use the following symbols
C,D for commands.
C ::= x:=M | C; D | **if** M **do** C | **while** M **do** C
We extend the syntax class M
M ::= [C] M
The intended meaning of this new kind of expression is " evaluate M after doing the command C". For example if x≠y then

[x:=1; y:=2] x+y

evaluates to 3. These expressions have no side effect.
The use of [ ] for commands should not be confused with the notation for substitution. However there is a suggestive analogy.
We introduce a new predicate over Commands $closed_{comm}(C)$, which we will write simply as closed(C).
Finally we introduce the new rules for => and closed.

Evaluation rules

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

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

$$\frac{N \Rightarrow \text{true}\ ,\ [C]M \Rightarrow m}{[\textbf{if}\ N\ \textbf{do}\ C]\ M \Rightarrow m}$$

$$\frac{N \Rightarrow \text{false},\quad M \Rightarrow m}{[\textbf{if}\ N\ \textbf{do}\ C]\ M \Rightarrow m}$$

$$\frac{[\textbf{if}\ N\ \textbf{do}\ (C;\ \textbf{while}\ N\ \textbf{do}\ C)]\ M \Rightarrow m}{[\textbf{while}\ N\ \textbf{do}\ C]\ M \Rightarrow m}$$

Rules for closed

$$\frac{\text{closed}C,\ \text{closed}\,M}{\text{closed}\,[C]M}$$

$$\frac{\text{closed}\,M}{\text{closed}(\ x{:=}M\ )}$$

$$\frac{\text{closed}\,C,\ \text{closed}\,D}{\text{closed}(\ C;\ D)}$$

$$\frac{\text{closed}\,N,\quad \text{closed}\,C}{\text{closed}(\textbf{if}\ N\ \textbf{do}\ C)}$$

$$\frac{\text{closed}\,N,\quad \text{closed}\,C}{\text{closed}(\textbf{while}\ N\ \textbf{do}\ C)}$$

3.4. Expressions with side effects

How can we extend the system to deal with expressions which may have side effects? Here are some tentative thoughts. In the above C had a side effect but with the given semantics for the functional language [C]M had no side effects. Now let us change the semantics of the functional language to allow expressions, M, to have side effects. To accomplish this we adopt the following device: write [M]N to mean the value of N after evaluating the expression M. Now for example

[M+N] P has the same value as [M]([N] P)
M+N => p if M=>m and [M]N => n and m+n => p

The revised semantic rules might be as follows. We do not give them all, just enough to illustrate the idea.

$$\frac{N \Rightarrow n}{[x]N \Rightarrow n}$$

$$\frac{N \Rightarrow n}{[0]N \Rightarrow n}$$ similarly for the other constants

$$\frac{N \Rightarrow n}{[\text{lambda } x.\ M]\ N \Rightarrow n}$$

The rule for evaluating an application becomes

$$\frac{M \Rightarrow m,\quad [M]\,N \Rightarrow n,\quad m.n \Rightarrow p}{MN \Rightarrow p}$$

$$\frac{P \Rightarrow p}{[m.n]\,P \Rightarrow p}$$ (m.n has no side effect)

$$\frac{[C]([M]\,N) \Rightarrow n}{[[C]\,M]\,N \Rightarrow n}$$

4. Framework for Operational Semantics - continued

4.1.Evaluation logic - 2

In a semantics we may have several sorts of identifiers and expressions and more than one substitution operation. For example we may have local variables in commands.

We may define an evaluation logic over a multi-sorted alpha logic with sorts Id_s and $Expr_t$ with more distinguished evaluation and value predicates and substitution functions. Thus we have

$\alpha_{ij}, \underline{\alpha}_{ij}$: $Id_i * Id_i * Expr_j \rightarrow Expr_j$
$expr_{ik}$: $Id_i \rightarrow Expr_k$
$\Rightarrow_k$ over $Expr_k * Expr_k$
$Value_k$ over $Expr_k$
$[\_/\_]\,ikm$: $Expr_k * Id_i * Expr_m \rightarrow Expr_m$

Corresponding to each substitution function an appropriate substitution rule schema is added, or even several rule schemas corresponding to different evaluation predicates. These schemas will always be of a similar pattern. (Since we do not specify such a pattern here the notion of evaluation logic is somewhat loose.)

Substitution rule
We consider as an example of a substitution rule the one needed for the language features introduced in the next section. We have
x for identifiers
N,m,n for expressions
C for commands (a second sort of expressions)

We want a rule for substituting expressions for identifiers in commands. The rule is (dropping subscripts)

$$\frac{\begin{array}{c}(x' \Rightarrow m)\\ \vdots \\ \text{value } n, \quad [\alpha_{x'/x}\, C]N \Rightarrow \underline{\alpha}_{x'/x}\, n\end{array}}{[[m/x]C]N \Rightarrow n} \quad x' \text{ is a new variable}$$

provided that x' is a new variable, that is it does not occur in any assumption except the ones on the top line nor in m, x, C, N and n.

5. Further example semantics

5.1. Local declarations and procedures with parameters

In this example we illustrate the semantics of a language with local variables in commands and with procedure declaration facilities. More precisely we discuss procedures with one parameter, passed by value, and possibly with local variables. We do not consider kinds of parameters other than value ones or procedures as parameters. We do not address the issue of recursive procedures here.

Signature
We extend the signature of the assignment language as follows.
A distinguished substitution function
[/] : Expr * Id * Command -> Expr
obeying the substitution rule introduced in the previous section. This will enable us to take care of local variables.
We introduce a new Expr-like sort Procedures and an Id-like sort Procedure_names, with a corresponding evaluation predicate ==>.
We use the following symbols
Q, h for Procedures
P for Procedure_names
P ::= P_0 | ... | P_k
Q ::= **lambda** x. C

We generalise the syntax class Command to
C ::= **begin new** x = M **in** D **end** | **proc** P(x) = C **in** D | P(M)

We use the substitution [m/x]C defined in the previous section. We omit the definition of the other substitution which we need, [h/P]C. The definition will be similar to [m/x]C, and

it will give us the mechanism for procedure call. We do not need to introduce procedure closures.
Finally we introduce the new rules for => and closed.

Evaluation rules

$$\frac{M \Rightarrow m, \qquad [[m/x]C]\ N \Rightarrow n}{[\textbf{begin new } x = M \textbf{ ; } C \textbf{ end}]\ N \Rightarrow n}$$

$$\frac{[\ [\textbf{lambda } x.\ C/P]\ D\ ]\ N \Rightarrow n}{[\textbf{proc } P(\ x) = C \textbf{ in } D]\ N \Rightarrow n}$$

$$\frac{P \Longrightarrow \textbf{lambda } x.\ C, \quad M \Rightarrow m, \quad [[m/x]C]\ N \Rightarrow n}{[P(M)]\ N \Rightarrow n}$$

Rules for closed

$$\frac{\begin{array}{c}\text{closed } x \\ \vdots \\ \text{closed } n, \quad \text{closed } C\end{array}}{\text{closed}([n/x]\ C)}$$

$$\frac{\text{closed M,} \qquad \begin{array}{c}\text{closed } x \\ \vdots \\ \text{closed } C\end{array}}{\text{closed}(\textbf{begin new } x = M \textbf{ ; } C \textbf{ end})}$$

$$\frac{\text{closed } [[\textbf{lambda } x.\ C/P]D]}{\text{closed } [\textbf{proc } P(\ x) = C \textbf{ in } D]}$$

$$\frac{\text{closed } M\ , \quad \text{closed } N}{\text{closed } [P(M)]\ N}$$

$$\frac{\begin{array}{c}\text{closed}(x) \\ \vdots \\ \text{closed}(\ C), \quad \text{closed}([\ D\ ]\ N)\end{array}}{\text{closed}(\ [[\textbf{lambda } x.\ C/P]\ D\ ]\ N)}$$

6. Definition of Alpha Logic in the Edinburgh Logical Framework

In this section we outline a definition of the minimal signature of Alpha Logic in the Edinburgh Logical Framework. This is a signature with two sorts Id and Expr and two function symbols

$\alpha, \underline{\alpha}$: Id –> Id –> Expr-> Expr

In order to encode in ELF such an instance of Alpha Logic we proceed as follows. First of all we will introduce an ELF type corresponding to the collection of sorts and a type constructor, Term, defined on sorts.

Sorts : Type
Term : Sorts -> Type

We will introduce constants corresponding to Id and Expr and two new constants. The first $\supset$, is intended to denote the higher order sort constructor (written as an infix and bracketed to the right), while the second is intended to denote syntactic application

Id : Sorts
Expr : Sorts
$\supset$: Sorts -> Sorts -> Sorts
app : Π s,t : Sorts. Term(s$\supset$t) -> Term(s) -> Term(t)

Corresponding to a function in the signature we declare a constant of type Term(s) for the appropriate s, that is

$\alpha, \underline{\alpha}$: Term(Id $\supset$ Id $\supset$ Expr $\supset$ Expr).

Corresponding to the class of formulae we introduce an ELF type
Form : Type.
Corresponding to a predicate in the signature we would introduce a constant over types Term(s) for appropriate sorts s.

Finally we introduce a judgement forming operator, asserting the truth of Alpha Logic formulas

True : Form -> Type.

We now encode the rules for the replacement operators α and $\underline{\alpha}$. This is the most elaborate part. We have to introduce in the ELF signature a number of new predicates, new constants, new formula forming operators, and new rules governing the provability of these new formulae. More precisely we define

$\in$: Π s : Sorts. Term(Id) -> Term(s) -> Type

$\notin$: Π s : Sorts. Term(Id) -> Term(s) -> Type

The judgement x $\in$ M says that the identifier x occurs in M, while x $\notin$ M says that it does not occur. To translate the rules of Alpha Logic we introduce constants of the types which follow. (For ease of reading we will write rules using a somewhat more suggestive notation than the ELF Π and ->. We will subscript arguments which are sorts)

$$\frac{x \in_s M}{x \in_t \mathrm{app}_{s\,t}\, N\, M} \quad s,t\text{: Sorts, } x\text{:Term}_{Id}\text{, } M\text{: Term}_s\text{, } N\text{:Term}_{s \supset t}$$

$$\frac{x \in_{s \supset t} N}{x \in_t \mathrm{app}_{s\,t}\, N\, M} \quad s,t\text{: Sorts } x\text{:Term}_{Id}\text{, } M\text{: Term}_s\text{, } N\text{:Term}_{s \supset t}$$

$$\frac{x \notin_s M,\ x \notin_{s \supset t} N}{x \notin_t \mathrm{app}_{s\,t}\, N\, M} \quad s,t\text{: Sorts } x\text{:Term}_{Id}\text{, } M\text{: Term}_s\text{, } N\text{:Term}_{s \supset t}$$

$$\frac{}{x \in_{Id} x} \quad x\text{ :Term}_{Id}$$

$$\frac{y \notin_{Id} x}{x \notin_{Id} y} \quad x,y\text{ :Term}_{Id}$$

$$\frac{}{x \notin_{Id \supset Id \supset Exp \supset Expr} \alpha} \quad x\text{: Term}_{Id} \quad \text{(and similarly for } \underline{\alpha})$$

When we come to add identifier constants to the signature, for each pair of distinct constants, i,j in Term_{Id}, we will need a rule

$$\frac{}{i \notin_{Id} j}$$

We are now ready to illustrate how to encode the set of rules about the α and $\underline{\alpha}$ operators in ELF. We introduce constants of the following ELF types

$$\frac{x \notin_{Expr} F(y),\ \mathrm{True}(G(F(y)))}{\mathrm{True}(G(\alpha_{y/x}\, F(x)))} \quad \begin{array}{l} F\text{:Term}_{Id}\text{->Term}_{Expr}\text{, } G\text{:Term}_{Expr}\text{->Form,} \\ x,y\text{: Term}_{Id} \end{array}$$

$$\frac{y \notin_{Expr} F(x),\ \mathrm{True}(G(F(y)))}{\mathrm{True}(G(\underline{\alpha}_{y/x}\, F(x)))} \quad \begin{array}{l} F\text{:Term}_{Id}\text{->Term}_{Expr}\text{, } G\text{:Term}_{Expr}\text{->Form,} \\ x,y\text{: Term}_{Id} \end{array}$$

In the last rule we have omitted app before α and $\underline{\alpha}$ for ease of reading.

Evaluation Logic in ELF

We will now consider a more elaborate example: translating the minimal signature of Evaluation Logic.
Besides the standard ELF types and constants we will introduce here a judgement forming operator corresponding to => , one corresponding to value and constants for the expression constructors.
The first part of the ELF signature will then be as before:

Sorts : Type
Form : Type
Term : Sorts -> Type
True :Form -> Type
Id : Sorts
Expr : Sorts
$\supset$: Sorts - > Sorts ->Sorts
app : Π s,t : Sorts. Term(s$\supset$t) -> Term(s) -> Term(t)
α, $\underline{\alpha}$: Term(Id $\supset$ Id $\supset$ Expr $\supset$ Expr)
$\in$, $\notin$: Π s : Sorts.Term(Id) -> Term(s) -> Type

with the addition of
expr : Term(Id $\supset$ Expr)
[/] : Term(Expr $\supset$ Id $\supset$ Expr $\supset$ Expr)
value: Term(Expr) -> Form
=> : Term(Expr) -> Term(Expr) -> Form

A few more rules specific to the particular Σ of Evaluation Logic have to be introduced in addition to the ones we had before. As before we will limit ourselves to giving the type

$$\frac{}{x \notin \mathrm{Id} \supset \mathrm{Expr} \supset \mathrm{Expr} \supset \mathrm{Expr}\ [/]} \quad x: \mathrm{Term}_{\mathrm{Id}}$$

$$\frac{}{\mathrm{T}x \notin \mathrm{Id} \supset \mathrm{Expr}\ \mathrm{expr}} \quad x: \mathrm{Term}_{\mathrm{Id}}$$

The translations of the rules specific to Evaluation Logic in ELF are straightforward:

$$\frac{\mathrm{True}(\mathrm{value}(n))}{\mathrm{True}(n \Rightarrow n)} \quad n:\mathrm{Term}_{\mathrm{Expr}}$$

and

$$\frac{w \in_{Id} x}{w:Term_{Id}} \quad \frac{w \in_{Expr} M}{w:Term_{Id}} \quad \frac{w \in_{Expr} m}{w:Term_{Id}}$$
$$w \notin_{Id} x', \qquad w \notin_{Id} x', \qquad w \notin_{Id} x', \qquad True(x' \Rightarrow n)$$
$$\frac{}{x':Term_{Id}}$$

$$\frac{True(value(n)) \qquad True(\alpha_{x'/x}\, M \Rightarrow \underline{\alpha}_{x'/x}\, m)}{True([x/n]\ M \Rightarrow m)} \quad x:Term_{Id},\ M,m,n:Term_{Expr}$$

In the last rule for ease of reading we have omitted expr, also app before α, $\underline{\alpha}$ and [/].

The extension to Evaluation Logic for commands will be analogous.

Concluding remarks

We have shown how to define semantics of a simple but non-trivial language in our Natural Deduction style. We have not treated reference variables, exceptions or data types, nor have we defined the type discipline by a static semantics. These remain to be investigated. Another area for exploration would be the application of the technique to defining logics. We would also like to consider program verification and transformation in this formalism .

Although our system relies on the Edinburgh Logical Framework for a formal definition, it can be applied without explicit reference to ELF, basing it on Alpha Logic.

References

Avron, A. ,Honsell, F, Mason, I, (1987) Using Typed Lambda Calculus to Implement Formal Systems on a Machine, Report LFCS87-31, Comp. Sci Dept. Edinburgh Univ. UK

Kahn, G. (1987) Natural Semantics, Rapport de Recherche N. 601, INRIA, France

Harper, R. , Honsell, F., Plotkin, G. (1987) A Framework for Defining Logics, Proceedings of the Second Annual Conference on Logic in Computer Science, Cornell, USA

Landin, P.J. (1964) The Mechanical Evaluation of Expressions, Computer Journal, 6.

Mason, I. (1987) Hoare's Logic in the LF, Report LFCS-87-32, Comp. Science Dept. Edinburgh University, UK

Plotkin, G.(1981) A Structural Approach to Operational Semantics, DAIMI FN-19, Computer Science Department ,Aarhus University, Denmark

Prawitz, D. (1965) Natural Deduction: A Proof-Theoretic study, Almqvist & Wiksel, Stockholm

Reynolds J.C. (1978) Syntactic Control of Interference, 5th Annual Symp. on Principles Of Prog. Langs., Tucson, ACM