ROSE '95

PROCEEDINGS

The 3rd Romanian Conference on Open Systems

Technical Sessions • Promotional Papers

1-4 November 1995, Bucharest, Romania

An Operational Semantics for NPL

Eneia Todoran
Department of Computer Engineering, Technical University of Cluj-Napoca
26 Baritiu Street, 3400 Cluj-Napoca, Romania
e-mail: eneia@utcluj.ro

Abstract. NPL is a concurrent programming language. NPL combines in a uniform framework basic control flow concepts from sequential logic programming with backtracking and cut and from parallel logic programming with committed choice. This paper presents some sample programs and an operational semantics for NPL in the style of the Structured Operational Semantics of Plotkin.

1. Introduction.

The language NPL was developed in a series of papers ([6],[7]). Intuitively, it is obtained by parameterizing a process algebra (the process layer) with Lisp expressions (the data layer). This paper focuses on the approach to non-determinism in NPL. In NPL the non-determinism is a union of behaviors. In this model it is possible to combine in a uniform framework basic control flow concepts from sequential logic programming with backtracking and cut and from parallel logic programming with committed choice. In this paper we present some sample programs and an operational semantics for NPL in the style of Plotkin's Structured Operational Semantics. The transition system presented in section 3. has been implemented as an executable Prolog prototype and it has been tested on a large number of examples (including the ones presented in section 2.)

2. An informal introduction to the Language

The syntax of NPL is very intuitive: a mixture of Lisp and process algebra. The statements of the language are built by Lisp expressions, process names and a special process '!' named *select*, combined by means of operators on processes. Process names are distinguished from Lisp symbols by beginning with an uppercase letter (Lisp symbols must begin with a lower case letter; in this paper we only deal with parameterless processes in NPL - see also [7]). There is a special process named 'Root' which has the task to create all the other processes for a certain program. NPL provides (binary) operators for sequential composition (.), for parallel composition (||) (with a lower precedence than '.'; curly braces may be used to force the precedence in expressions), and two (n-ary) operators for non-deterministic choice (in the general notation: $\sum_i s_i = s_1 + ... + s_n$): $[s_1 + ... + s_n]$ (abbreviated as [+]) and $<s_1 + ... + s_n>$ (abbreviated as <+>). In NPL the non-determinism is a union of behaviors. The operator $[s_1 + ... + s_n]$ introduces an ordering relation (specified by the textual order) between the

non-deterministic alternatives (s_i). Operationally, this comes to a mechanism of backtracking. The operator <+> specifies a parallel evaluation of the non-deterministic alternatives. In both cases the non-deterministic alternatives operate on different data as in Prolog and Concurrent Prolog. There is a mechanism of "negation by failure" in NPL. The evaluation of any Lisp expression to *nil* produces the abandon of the continuation of the current non-deterministic alternative of the process. In NPL logic is a consequence of an interpretation of process behavior. The values of truth are inherited from Lisp (*nil* for *false* and everything else for *true*). The operators on processes allow for the following logic interpretation: [+] = OR, <+>=(parallel) OR, .=AND, ||=(parallel) AND. The Lisp in NPL is extended with a logic of work with free and bound variables, which is the basis for the dataflow mechanism (see [6]). Assuming that the variables x, y and z are initially unbound (undefined), the following program fragment has the predictable behavior of assigning them the values x=1, y=3 and z=2: (setf x 1)||(setf y (+ x z))||(setf z 2). The Lisp command *setf* is used in NPL as an assignment primitive. There are two types of elementary actions in NPL: simple expressions (without any side effect) and assignments.

As a starting programming example we present a process that computes the length of a list 'l'. It is based on a simple *while* loop. The program below is deterministic. The guards: 'l' and '(null l)' can be inspected in any order, so we could have used the operator <+> instead of [+].

```
Root=(setf l '(1 2 3 ...)).(setf n 0).While.(print n)
While=[(null l)+l.(list (setf l (cdr l))).(setf n (+ 1 n)).While]
```

We continue with two processes that search an element in a tree. They only differ with respect to the operational strategy that they use (i.e. sequential or parallel). In both cases the search process is abandoned as soon as the element was found. When the special process ! is executed it selects a non-deterministic alternative in a set by destroying all the others. It is (syntactically) attached to the group of non-deterministic alternatives where it appears. For instance in the process 'SeqTreeSearch' below, the (three) non-deterministic alternatives are inspected in textual order. If 'elem' is equal with the root of the tree (tree1) then the search process stops. Otherwise the search process continues on the left subtree: '(car (cdr tree1))'. If 'elem' is found on the left subtree then the search of the right subtree is abandoned. In the case of the 'ParTreeSearch' process all the non-deterministic alternatives are inspected in parallel, and the first one which succeeds executes its (own) *select* process which destroys the other two alternatives. Both the processes use deep guards in performing their task.

```
SeqTreeSearch=tree1.[(equal elem (car tree1)).!+
        (setf tree1 (car (cdr tree1))).SeqTreeSearch.!+
        (setf tree1 (cdr (cdr tree1))).SeqTreeSearch]

ParTreeSearch=tree2.<(equal elem (car tree2)).!+
        (setf tree2 (car (cdr tree2))).ParTreeSearch.!+
        (setf tree2 (cdr (cdr tree2))).ParTreeSearch.!>
```

The operator '!' does not alter the compositionality of the semantics in the presence of the (sequential or of the) parallel processes. The two tree search processes can be performed in parallel (||). If both the searches are successfull ('elem' is present in both trees: 'tree1' and 'tree2') then, their (||=AND) parallel evaluation is succesfull. Otherwise the search process fails.

```
Root={(setf tree1 ...)||(setf tree2 ...)||(setf elem ...)}.{SeqTreeSearch||ParTreeSearch}
```

In the sequel we present three algorithms for graph search. The graph is represented as an association list indexed with the nodes of the graph. To each node it is attached the list of its neigh-

bors. The process 'Search' computes all all the paths between a start node and a goal node.

```
Root={Init||(setf path '(a))}.Search.(print path)
Init=(setf graph '((a b d nil nil)(b a d e nil)(c d e nil nil) (d a b c e)(e b d c nil)))
     ||(setf node 'a)||(setf goal 'e)
NotMb=<(null pp)+ pp.(not (equal (car pp) neighbor)). (list (setf pp (cdr pp))).NotMb>

Search=<(equal node goal)+(not (equal node goal)). (setf neighbors (cdr (assoc node graph))).
    Sel.(setf pp path).NotMb.(setf node neighbor).(setf path (cons neighbor path)).Search>
```

The effective operational strategy of search is specified by the process 'Sel' that (non-deterministically) selects a 'neighbor' for the current 'node'. We present three possible versions for this process. A Prolog-like depth-first search is obtained if the neighbors are inspected in a specified order (by means of [+]).

```
Sel=neighbors.[(setf neighbor (car neighbors))+ (setf neighbors (cdr neighbors)).Sel]
```

A parallel search is obtained by using the following process (that uses <+> instead of [+]).

```
Sel=neighbors.<(setf neighbor (car neighbors))+ (setf neighbors (cdr neighbors)).Sel>
```

In this case the solutions are returned in an unpredictable order.

The problem of search is in general NP-complete. The time complexity t(n) of a depth first search - where n is the "depth" of the search tree - is exponential in n, i.e. $t(n)=O(f^n)$ where f is the "fan-out" of the search tree. A parallel search can be very fast (t(n)=O(n)) but it assumes the availability of a number of processors that is exponential in n, $p(n)=O(f^n)$. For n=10 and f=4 (we assume that the search tree is uniform, i.e. each node has 4 neighbors), either t(n) or p(n) would be of the order of $10^6(=4^{10})$. The combination of the two operators for non-deterministic choice (in the process that selects a neighbor) can distribute the computing task among the two coordinates of the algorithm (time and number of processors). For n=10 and f=4 we obtain an algorithm with both t(n) and p(n) of the order of $10^3(=2^{10})$ by using the following process for neighbor selection.

```
Sel=<[(setf neighbor (car neighbors))+(setf neighbor (car (cdr (cdr (cdr neighbors)))))]+
   [(setf neighbor (car (cdr neighbors)))+(setf neighbor (car (cdr (cdr neighbors))))]>
```

3. Operational Semantics for NPL

In this section we present an operational semantics for NPL. It is based on a transition relation embedded in a deductive system in the style of Plotkin's Structured Operational Semantics [4]. The configurations that we use in the definition of the transition relation are partially ordered sets with a tree structure. The elementary actions - which are Lisp expressions in NPL - will be modeled by simple expressions over some data algebra and by assignment statements. The style of the presentation follows the one in [3], where semantics is studied in a metric framework. The Operational Semantics will be defined as the fixed point of a suitable contraction, and the semantic universe will be a complete metric space. We will use a so-called "linear-time" semantic domain. The elements of such a domain are sets of (possible) "traces".

We recall that given a set A, a *partial order* on A (often denoted by $\leq$ in analogy with familiar orderings on numbers) is a reflexive, transitive and antisymmetric relation on A. A *linear order* on

A is a partial order on A which satisfies: $a \geq b$ or $b \geq a$, for all $a,b \in A$. A linear ordering $\geq$ on a set A is a *well ordering*, if each non-empty $S \subseteq A$ has a least element, that is there is $s \in S$ such that $s' \geq s$, for all $s' \in S$. A *tree* $T=(T,\geq_T)$ is a partially ordered set with a least element (the *root*) such that, for every $x \in T$, the set of predecessors of x is well-ordered. We only consider trees such that the set of predecessors of each element in T is finite.

In the sequel the notation $(x \in)X$ introduces the set X with typical element x ranging over X. For X a set, we denote by $\mathcal{P}(X)$ the power set of X, i.e., the collection of all subsets of X. $\mathcal{P}_\pi(X)$ denotes the collection of all subsets of X which have the property π. For instance we note by $\mathcal{P}_{fin}(X)$ the set of all finite subsets of X. We introduce a number of syntactic and semantic notions.

Definition 3.1. (Expressions and states)

a) Let $Alg=<V;op_1,op_2,...>$ be a (data algebra) over some domain V (**R** might be an example). Let Var, with elements x,y,... be a class of simple variables. Let moreover $\sum$=Var->V$\cup${?}. We extend the definition of the operators op_i of Alg, so that an expression which has '?' as either of the arguments always evaluates to '?'. The elements $\sigma \in \sum$ have the intuitive meaning of machine states. When a variable x has the value '?' in the state σ ($\sigma(x)$='?') we say that the variable is free/unbound. When $\sigma(x) \in V$ we say that x is bound. Throughout the rest of this paper $\sigma_0 \in \sum$ will denote the state in which all the variables are unbound (i.e. $\sigma_0(x)$=?, $\forall x \in$Var).

b) Let $v \in V \cup \{?\}$, $\sigma \in \sum$, $x \in$Var. We use the "variant" notation turning the state σ into a state $\sigma\{v/x\}$ by putting:

$$\sigma\{v/x\}(y) = \begin{cases} v \ \underline{if} \ x \equiv y \\ \sigma(y) \ \underline{if} \ x \not\equiv y \end{cases}$$

c) We introduce the class $(e \in)$ Exp of expressions (over the data algebra Alg). We assume given the valuation Val: Exp -> ($\sum$ -> V), and an "interpretation" function B:V -> Bool$\cup${?} (Bool={T,F}) for the values in V$\cup${?}, such that B(?)=? (in NPL the interpretation function (which maps Lisp values in Bool$\cup${?}) is B(nil)=F, B(?)=? and for every other (Lisp) value v B(v)=T).

Definition 3.2. (NPL syntax)

a) (Statements). The class $(s \in)$ NPL of *statements* is given by:

s::=e | x:=e | X | ! | $s_1.s_2$ | $s_1 \parallel s_2$ | $[s_1+...+s_n]$ | $<s_1+...+s_n>$

with X$\in$Pvar, a set of procedure variables. We will assume that each program uses exactly the procedure variables in the initial segment $\mathcal{X}=\{X_0,...,X_n\}$ of Pvar, for some $n \geq 0$.

b) (Guarded statements). The class $(g \in)$ NPL^g of *guarded statements* is given by:

g:=e | x:=e | g.s | $g_1 \| g_2$ | $[g_1+...+g_n]$ | $<g_1+...+g_n>$

Note that '!' does not act as a guard.

c) (Declarations). The class $(D \in)$ NPL^D of *declarations* consists of n-tuples $D \equiv X_0=g_0,..,X_n=g_n$ or $<X_i=g_i>_i$ for short, with $X_i \in \mathcal{X}$ and $g_i \in NPL^g$, i=0...,n.

d) (Programs). The class $(\rho \in)NPL^P$ of programs consists of pairs $\rho \equiv <X_0|D>$, with $D \in NPL^D$ and $X_0 \in \mathcal{X}$ (X_0 plays the role of the process 'Root').

Definition 3.3. (Configurations)

a) (Identifiers) The class $(\alpha, \beta \in)$Id, of *identifiers* is defined by: α::=i | α:i, for i$\in$N. We also use the class of finite sets of integers: $(I \in)Pid=\mathcal{P}_{fin}(N)$

b) (Parallel Processes) The class $(r \in)$ PP of *parallel processes* is defined by: $PP=\mathcal{P}_{fin}(Id \times Pid \times NPL)$. We will call an element $p \in r$ (for some $r \in PP$) a *process cell.*

c) (Parallel Non-deterministic alternatives) The class $(t \in)PN$ of *parallel non-deterministic alternatives* is defined by: $PN=\mathcal{P}_{fin}(Id \times Pid \times \sum \times PP)$. The elements $t \in PN$ will be the *configurations* of the transition system that defines the operational semantics for NPL. We will call an element $n \in t$ (for some $t \in PN$) a *non-deterministic alternative.*

Definition 3.4. (Consistent configurations)

We say that an element $r \in PP$ is *consistent* if the identifier component of each process cell in r

is unique among all the identifiers of the process cells in r. Similarly, we say that an element $t \in PN$ is a *consistent* if the identifier component of each non-deterministic alternative in t is unique among all the identifiers of the non-deterministic alternatives in t.

Definition 3.5. (Partial orderings on configurations)

On any consistent $t \in PN$ and consistent $r \in PP$ we define the following relations, $\geq_t$, $\geq_r$:

if $p_1, p_2 \in r$, $p_1 = <(\ldots(\beta:i_1):\ldots):i_k, I_1, s_1>$ (for $k \geq 0$), and $p_2 = <\beta, I_2, s_2>$ then $p_1 \geq_r p_2$

if $n_1, n_2 \in t$, $n_1 = <(\ldots(\alpha:i_1):\ldots):i_k, I_1, \sigma_1, r_1>$ (for $k \geq 0$), and $n_2 = <\alpha, I_2, \sigma_2, r_2>$ then $n_1 \geq_t n_2$.

We will sometimes refer to the computing entities of a consistent configuration by only mentioning their identifier.

We must point out here that both the non-deterministic alternatives in a configuration, and the process cells in a non-deterministic alternative are grouped in a tree structure in which only the leaves are active computing entities.

Definition 3.6. (Derivable configurations)

We note by '_' the components of the computing entities that are irrelevant for the exposure (and uniquely determined by the identifier).

1) An element $r \in PP$ is *derivable* if r is consistent and:

a) $p = <0, \_, \_> \in r$ and $\forall p' \in r\ p' \geq_r p$.

b) $<\beta:j, \_, \_> \in r \Rightarrow <\beta, I, \_> \in r$, and $j \in I$.

c) $<\beta, I, \_> \in r \Rightarrow \forall j \in I\ <\beta:j, \_, \_> \in r$.

If $p = <\beta, \emptyset, s>$ ($s \in NPL$) then we say that p is an *active* process cell.

2) An element $t \in PN$ is *derivable* if t is consistent, for every $<\alpha, I, \sigma, r> \in t$, r is derivable and:

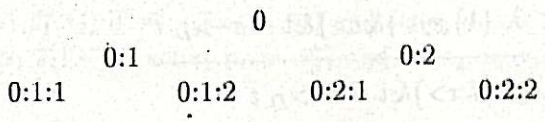
a) $n = <0, \_, \_, \_> \in t$ and $\forall n' \in t\ n' \geq_t n$.

b) $<\alpha:j, \_, \_, \_> \in t \Rightarrow <\alpha, I, \_, \_> \in t$ and $j \in I$.

c) $<\alpha, I, \_, \_> \in t \Rightarrow \forall j \in I\ <\alpha:j, \_, \_, \_> \in t$.

If $n = <\alpha, \emptyset, \sigma, r>$ ($\sigma \in \sum$, $r \in PP$) then we say that n is an *active* non-deterministic alternative.

Remark that each derivable computing entity is a partially ordered set (with respect to the relations defined in 3.5.) with a tree structure. Indeed, by a) the structures have a root and all the computing entities have identifiers of the form: $\alpha = (\ldots((0:i):j)..):k$. By b) the predecessors of α are 0, 0:i, (0:i):j,... which clearly form a well ordered set. The representation that we use for the trees is suggested in the following picture:



Whenever we introduce a configuration, it will be tacitly assumed that it is derivable. This fact will be justified by lemma 3.9.

Let now $(\tau \in) Q = ((A \times \sum) \cup \{\varepsilon\}) - (a \in) A$ is the set of "elementary actions" in NPL: a::=e| x:=e. The operational semantics for NPL is based on a transition relation $\subseteq PN \times Q \times NPL^D \times PN$. We write the elements of such a transition relation in the notation:

$$t_1 \text{—}\tau\text{->}_D t_2 \qquad (1)$$

We *specify* the transition relation by a (finite) set of *axioms* (written in the form as in (1)) and *rules*, in the form:

$$t_1 \text{—}\tau\text{->}_D t'$$

$$\frac{}{t_2 \text{—}\tau\text{->}_D t'} \qquad (2)$$

The transitions of the system can be infered by *backward chaining*. In order to find the transitions of t_2 in (2) we try to infer the transitions of t_1, as any transition of t_1 is also a transition of t_2.

In the sequel we note by B&C, a non-empty set A such that A=B∪C and B∩C=∅. We (especially) use this notation as a mechanism for choice.

Definition 3.7.(Transition relation specification for NPL)

The transition relation for NPL is the smallest relation satisfying:

(End) $\{<0,\emptyset,\sigma,\emptyset>\} \text{—}\varepsilon\text{->} \emptyset$

(Elem1) $\{<\alpha,\emptyset,\sigma,\{<\beta\text{:i},\emptyset,\text{e}>,<\beta,\{\text{i}\}\&\text{I},\text{s}>\}\&\text{r}>\}\&\text{t} \text{—}[\text{e},\sigma]\text{->}_D \{<\alpha,\emptyset,\sigma,\{<\beta,\text{I},\text{s}>\}\&\text{r}>\}\&\text{t}$ if B(Val(e)(σ))=T

(Elem2) $\{<\alpha,\emptyset,\sigma,\{<\beta\text{:i},\emptyset,\text{x:=e}>,<\beta,\{\text{i}\}\&\text{I},\text{s}>\}\&\text{r}>\}\&\text{t} \text{—}[\text{x:=e},\sigma]\text{->}_D \{<\alpha,\emptyset,\sigma\{Val(e)(\sigma)/x\},\{<\beta,\text{I},\text{s}>\}\&\text{r}>\}\&\text{t}$ if B(Val(e)(σ))=T

(Fail1) $$\frac{\{<\alpha,I,\sigma_1,\text{r}_1>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}{\{<\alpha\text{:i},\emptyset,\sigma,\{<\beta,\emptyset,\text{c}>\}\&\text{r}>,<\alpha,\{\text{i}\}\&\text{I},\sigma_1,\text{r}_1>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}$$ if B(Val(e)(σ))=F

(Fail2) $$\frac{\{<\alpha,I,\sigma_1,\text{r}_1>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}{\{<\alpha\text{:i},\emptyset,\sigma,\{<\beta,\emptyset,\text{x:=c}>\}\&\text{r}>,<\alpha,\{\text{i}\}\&\text{I},\sigma_1,\text{r}_1>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}$$ if B(Val(e)(σ))=F

(Rec) $$\frac{\{<\alpha,\emptyset,\sigma,\{<\beta,\emptyset,\text{g}>\}\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}{\{<\alpha,\emptyset,\sigma,\{<\beta,\emptyset,\text{X}>\}\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}$$ if X=g ∈ D

(Seq) $$\frac{\{<\alpha,\emptyset,\sigma,\{<\beta\text{:1},\emptyset,\text{s}_1>,<\beta,\{1\},\text{s}_2>\}\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}{\{<\alpha,\emptyset,\sigma,<\beta,\emptyset,\text{s}_1.\text{s}_2>\}\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}$$

(Par) $$\frac{\{<\alpha,\emptyset,\sigma,\{<\beta\text{:1},\emptyset,\text{s}_1>,<\beta\text{:2},\emptyset,\text{s}_2>,<\beta,\{1,2\},\text{E}>\}\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}{\{<\alpha,\emptyset,\sigma,<\beta,\emptyset,\text{s}_1\|\text{s}_2>\}\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}$$

(Pend) $$\frac{\{<\alpha,\emptyset,\sigma,<\beta,\text{l},\text{s}>\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}{\{<\alpha,\emptyset,\sigma,\{<\beta\text{:i},\emptyset,\text{E}>,<\beta,\{\text{i}\}\&\text{I},\text{s}>\}\&\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}$$

(Nend) $$\frac{\{<\alpha,\text{I},\sigma_1,\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}{\{<\alpha\text{:i},\emptyset,\sigma,\emptyset>,<\alpha,\{\text{i}\}\&\text{I},\sigma_1,\text{r}>\}\&\text{t} \text{—}\tau\text{->}_D \tilde{t}}$$

$$\frac{\{<\alpha:\overbrace{1:\ldots:1}^{n\ times},\emptyset,\sigma,\{<\beta,\emptyset,\overline{s_1}>\}\&r>,\ldots,<\alpha:\overbrace{1:\ldots:1}^{(n-i)times},\{1\},\sigma,\{<\beta,\emptyset,\overline{s_i}>\}\&r>,\ldots,<\alpha,\{1\},\sigma_0,\emptyset>\}\&t \xrightarrow{\tau}_D \tilde{t}}{\{<\alpha,\emptyset,\sigma,\{<\beta,\emptyset,[s_1+\ldots+s_n]>\}\&r>\}\&t \xrightarrow{\tau}_D \tilde{t}}\ (SeqNed)$$

where $sel(s_i,\alpha,\alpha:\overbrace{1:\ldots:1}^{(n-i)times},\overline{s_i}>$ for i=1,...,n

$$\frac{\{<\alpha{:}1,\emptyset,\sigma,\{<\beta,\emptyset,\overline{s_1}>\}\&r>,\ldots,<\alpha{:}n,\emptyset,\sigma,\{<\beta,\emptyset,\overline{s_n}>\}\&r>,<\alpha,\{1,\ldots,n\},\sigma_0,\emptyset>\}\&t \xrightarrow{\tau}_D \tilde{t}}{\{<\alpha,\emptyset,\sigma,\{<\beta,\emptyset,<s_1+\ldots+s_n>>\}\&r>\}\&t \xrightarrow{\tau}_D \tilde{t}}\ (ParNed)$$

where $sel(s_i,\alpha,\alpha{:}i,\overline{s_i})$ for i=1,...,n

$$\frac{\{<\alpha,\emptyset,\sigma,r>\}\&t' \xrightarrow{\tau}_D \tilde{t}}{\{<\alpha,\emptyset,\sigma,\{<\beta,\emptyset,!(\alpha_1,\alpha_2)>\}\&r>\}\&t \xrightarrow{\tau}_D \tilde{t}}\ (Select)$$

where $t'=(t - t_0)\cup t_1$

$t0=\{<\alpha_i,l_i,\sigma_i,r_i> \mid <\alpha_i,l_i,\sigma_i,r_i> \in t$ and $\alpha_i \geq \alpha_1$ and not$(\alpha_i \geq \alpha_2)\}$

$t1=\{<\alpha_i,l_i,\sigma_0,\emptyset> \mid <\alpha_i,l_i,\sigma_i,r_i> \in t$ and $\alpha_i \geq \alpha_1$ and not$(\alpha_i \geq \alpha_2)\}$

The predicate 'sel' is:

$sel(!,\alpha_0,\alpha_1,!(\alpha_0,\alpha_1))$.
$sel(a,\alpha_0,\alpha_1,a)$.
$sel(X,\alpha_0,\alpha_1,X)$.
$sel([s_1+\ldots+s_n],\alpha_0,\alpha_1,[s_1+\ldots+s_n])$.
$sel(<s_1+\ldots+s_n>,\alpha_0,\alpha_1,<s_1+\ldots+s_n>)$.
$sel(s_1.s_2,\alpha_0,\alpha_1,\overline{s_1}.\overline{s_2})$ if $sel(s_1,\alpha_0,\alpha_1,\overline{s_1})$ and $sel(s_2,\alpha_0,\alpha_1,\overline{s_2})$.
$sel(s_1\|s_2,\alpha_0,\alpha_1,\overline{s_1}\|\overline{s_2})$ if $sel(s_1,\alpha_0,\alpha_1,\overline{s_1})$ and $sel(s_2,\alpha_0,\alpha_1,\overline{s_2})$.

Some comments are necessary. We begin with a definition.

Definition 3.8.

For $t_1,t_2 \in PN$, the relation $t_1 \longmapsto t_2$ holds between t_1 and t_2, whenever (for some $\tau \in Q$, $D \in NPL^D$, and $t' \in PN$) we have:

$$\frac{t_2 \xrightarrow{\tau}_D t'}{t_1 \xrightarrow{\tau}_D t'}$$

$\longmapsto^*$ denotes the reflexive and trensitive closure of $\longmapsto$.

An element $t \in PN$ performs an ε-transition only if $t \longmapsto^* \{<0,\emptyset,\sigma,\emptyset>\}$. $t \xrightarrow{\varepsilon} \emptyset$ (specified in rule

(End)) is the final transition for every finite trace: $\emptyset$ has no further transitions. The rules (Elem1) and (Elem2) specify the semantics of the elementary actions. Remark that there is no transition if the expression 'e' evaluates to an undefined value: '?'. The execution of the elementary actions is postponed until the necessary information becomes available. In this way we model the dataflow behavior in NPL. The rules (Fail1) and (Fail2) model the mechanism of "negation by failure" in NPL. An expression that is interpreted (by B) as *false* in the language produces the "failure" of the non-deterministic alternative in which it appears. The rule (Rec) embodies procedure execution by body replacement. The rules (Seq) and (Par) create processes and puts them in an appropriate order for further execution. The same thing hapness in the rules (SeqNed) and (ParNed), but in these cases entire non-deterministic alternatives are involved (entire contexts, consisting of a common state and a tree of processes are copied in each non-deterministic alternative). In the rules (SeqNed) and (ParNed) the (syntactically) corresponding instances (see the definition of the predicate 'sel') of the special process *select* are marked with two (semantic) addresses in the configuration tree as suggested in the picture below.

(SeqNed)

$$[\alpha]$$
$$\downarrow$$
$$[\alpha{:}1 \rightarrow !(\alpha,\alpha{:}1)]$$
$$\downarrow$$
$$\dots$$
$$\downarrow$$
$$[\alpha:\overbrace{1:\dots:1}^{n\ times} \rightarrow !(\alpha,\alpha:\overbrace{1:\dots:1}^{n\ times})]$$

(ParNed)

$$[\alpha]$$
$$\dots \downarrow \dots$$
$$\dots\ [\alpha{:}i \rightarrow !(\alpha,\alpha{:}i)]\ \dots$$

It is easy to see that the process $!(\alpha_1,\alpha_2)$ will be executed by a process in a non-deterministic alternative in the subtree rooted at α_2. Moreover (by (SeqNed) and (ParNed)), $\alpha_2 \geq \alpha_1$, i.e. α_2 is a subtree of α_1. By the rule (Select) the special process ! performs the transformation: $< \alpha_i,I_i,\sigma_i,r_i > \rightarrow < \alpha,I_i,\sigma_0, \emptyset >$ on all the non-deterministic alternatives in the tree rooted at α_1, except for those in the subtree rooted at α_2. By rule (Nend) a non-deterministic alternative of the form $< \alpha_i,I_i,\sigma, \emptyset >$ terminates immediately without performing any transition. Thus, given a configuration tree, the special process '!' *selects* one of its subtree. This remark sugests us another (intuitive) interpretation for the process operators in NPL based on operations on sets (of behaviors). In this interpretation [+] and <+> stand for set union, and . and || for set intersection. This interpretation is useful for the understanding of the semantics of the special process '!' (*select*) that performs a subset selection. Remark also that rule (Select) keeps unchanged the (tree) structure of the configurations. The rules (Pend) and (Nend) model "normal" process and non-deterministic choice termination. The constant E does not belong to the language. It is necessary for the creation of a dummy process that we use only to keep the configurations derivable.

Lemma 3.9. (Well definedness of Definition 3.7.)

a) $\{<0{:}1,\emptyset,\sigma_0,\{<0,\emptyset,X_0>\}>, <0,\{0{:}1\},\sigma_0,\emptyset >\}$ (this is the initial configuration of the system) is a derivable configuration.

b) If t_1 is derivable and $t_1 \longmapsto t_2$ then t_2 is derivable.

c) If t_1 is derivable and $t_1 \text{—}[a,\sigma]\text{->}_D t_2$ then t_2 is derivable.

Proof. Clear by the various axioms and rules of the transition relation specification for NPL.

The way we define the operational semantics for NPL is completely inspired by [3]. [3] is a necessary prerequisite for an accurate understanding of the rest of this paper, that concludes with Definition 3.11. Let now $(\theta \in)R=Q^* \cup Q^\omega$, i.e. R is the set of all finite and infinite sequences over Q. If '·' is the usual concatenation operator, then with ε (see the definition of Q) we compute as follows: $\varepsilon \cdot \theta = \theta \cdot \varepsilon = \theta$. We define a metric on R. Let for $\theta \in R$, $\theta(n)$ denote the prefix of θ of length n, in case $length(\theta) \geq n$, and θ otherwise. We put $d(\theta_1,\theta_2)=2^{-sup\{n|\theta_1(n)=\theta_2(n)\}}$, with the convention that $2^{-\infty}=0$. Then (R,d) is an ultrametric space.

Definition 3.10. (Semantic Universe)

We define the semantic domain for the operational semantics of NPL by $(\Theta \in)P=\mathcal{P}_{closed}(R)$, the set of all closed subsets of R (we have that (P,d_H) is a complete ultrametric space, where d_H is the Hausdorff metric on P induced by the metric d defined above).

We use the notation $\tau \cdot \Theta$ to denote the prefixing of all the sequences in Θ by τ. For example $\tau \cdot \{\tau_1, \tau_2\tau_3\} = \{\tau\tau_1, \tau\tau_2\tau_3\}$.

Definition 3.11. (Operational Semantics for NPL)

a) The mapping $\mathcal{O}$:$NPL^P \rightarrow P$ is:

$$\mathcal{O}[\![<X_0|D>]\!] = \mathcal{O}_D[\![\{<0{:}1,\emptyset,\sigma_0,\{<0,\emptyset,X_0\}>,<0,\{0{:}1\},\sigma_0,\emptyset>\}]\!]$$

b) We formally define $\mathcal{O}_D$ as the (unique) fixed point of the (higher order) operator Φ_D:$(PN \rightarrow P) \rightarrow (PN \rightarrow P)$, which for any $F \in PN \rightarrow P$ is defined by:

$$\Phi_D(F)(t) = \{\varepsilon \mid t \text{—}\varepsilon\text{->}_D \emptyset\} \cup \{[a,\sigma]\cdot F(t') \mid t \text{—}[a,\sigma]\text{->}_D t'\}.$$

4. Conclusions

The paper has presented some basic design considerations and an operational semantics for NPL in the style of the Structured Operational Semantics of Plotkin. The style of the semantic description follows the one in [3]. There is however, a distinctive part in our presentation. More precisely, the configurations of the transition system (that is the basis for the operational semantics) are partially ordered sets with a tree structure. The development of a denotational semantics, and a comparison between the two semantics ask for further research work.

References

[1] S.G. Akl. *Design and Analysis of Parallel Algorithms.* Prentice Hall, (1989).

[2] J.C.M. Baeten, C. Weijland. *Process Algebra.* Cambridge University Press, (1990).

[3] J.W. De Bakker, J.J.M.M. Rutten, (editors). *Ten Years of Concurrency Semantics,* World Scientific Publishing Co. Pte Ltd. (1992).

[4] G.D. Plotkin. *A Structural Approach to Operational Semantics,* Report DAIMI FN-19, Comp. Sci. Dept., Aarhus Univ, (1981).

[5] V. Stoltenberg-Hansen, I. Lindstrom, E.R. Griffor. *Mathematical Theory of Domains,* Cambridge University Press (1994).

[6] E. Todoran. *Dataflow Semantics in NPL,* in Proceedings of the Conference on Control and Technical Informatics, Timisoara, Romania, (1994).

[7] E. Todoran. *Non-determinism and Logic Programming in NPL,* ACAM vol 3., no. 2, (1994).