Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M. de Rooy

Concurrent evaluation of side-effects

Computer Science/Department of Software Technology Note CS-N8801 June

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Concurrent Evaluation of Side-Effects

M. de Rooy
Department of Mathematics and Computer Science
Free University, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

In this paper various semantics are given for an imperative and a functional language. In both languages the evaluation of expressions may have side-effects. Moreover, the languages support applicative concurrency. That is, the arguments of a function are evaluated concurrently. This may involve the concurrent evaluation of side-effects. For the imperative language an operational semantics and a denotational semantics are provided, based on the mathematical framework of complete metric spaces. Furthermore, metric tools are used to prove semantical equivalence. The functional language includes the feature of lazy single assignment, which is the basis for side-effects in the language. Both sequential evaluation and concurrent evaluation of the arguments of a function are considered. Operational, denotational and natural semantics are presented.

1982 CR Categories : F.3.2, F.3.3.
1985 Mathematics Subject Classification : 68Q10, 68Q55.
Keywords and Phrases : Concurrency, side-effects, imperative languages, functional languages, operational semantics, denotational semantics, natural semantics, continuations, processes, domain equations, metric spaces, lazy evaluation, single assignment, graph reduction, closures.
Note : This report describes graduate work done under the supervision of professor J.W. de Bakker.

Note CS-N8801
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. Introduction

In this paper various semantics are given for an imperative and a functional language. In both languages the evaluation of expressions may have side-effects. Traditionally, side-effects are modifications of the nonlocal environment, for instance assignments to nonlocal variables or variable parameters. However, we are primarily interested in the concurrent evaluation of side-effects. Therefore we regard every statement in an expression as a side-effect.

Concurrent evaluation of side-effects originates in the fact that both languages support applicative concurrency. Applicative concurrency is concurrency with regard to the way in which the arguments of a function are evaluated. The arguments of a function are evaluated concurrently, which may involve the concurrent evaluation of side-effects in the arguments.

Applicative concurrency is actually based on the fact that the order of evaluation of the arguments of a function is left unspecified. In order to evaluate the arguments of a function the atomic actions constituting the evaluation of the arguments are merged arbitrarily. In this manner the concurrent evaluation of the arguments of a function is modelled. The merging of atomic actions is done also in the collatoral elaboration in ALGOL 68.

We now proceed with an outline of this paper.

In chapter 2 some mathematical preliminaries are given. This chapter is especially concerned with the notions of complete metric spaces and domain equations. It provides the mathematical framework for chapter 3 in particular.

Chapter 3 describes the imperative language. It is a relatively simple language that supports the concurrent evaluation of side-effects in integer functional expressions and boolean functional expressions. Two interpretations of the language are considered, the latter based on an enlarged granularity of atomic actions. For each interpretation an operational semantics is given based on a Hennessy/Plotkin transition system and a metric denotational semantics is presented together with a proof of the equivalence of the operational and the denotational semantics. In these semantics the meaning of a statement is a process. A process is an element of a process domain. Process domains are obtained as solutions of domain equations. A process domain together with its metric is a complete metric space. Available metric techniques are used in order to simplify the semantical equivalence proof.

Chapter 4 describes the functional language. This language has been adopted from [Jo]. It is a functional language with the feature of lazy single assignment. This assignment provides the basis for side-effects. We present the denotational semantics of Josephs for the langauge. This semantics only supports sequential evaluation of the arguments of a function. We give a natural semantics for the language following a semantical definition method described in [Ka]. Furthermore, we provide operational semantics in order to support concurrent evaluation of arguments of functions and even the use of the so-called par combinator for parallel evaluation of expressions.

This paper concludes with a list of references.

Acknowledgement

I am grateful to professor J.W. de Bakker and dr. J.-J.Ch. Meyer for suggestions and discussions in preparing this paper.

2. Mathematical preliminaries

2.1. Notational conventions

A function f with domain X and range Y is denoted by $f: X \rightarrow Y$. The variant notation $f\{y/x\}$ with $x \in X$ and $y \in Y$ stands for a function from X to Y defined by:

$f\{y/x\}(x) = y$
$f\{y/x\}(x') = f(x')$ if $x \neq x'$

A set A with typical element α is denoted by $(\alpha \in)A$.

The powerset of a set A, the set of all subsets of A, is denoted by $\mathcal{P}(A)$. The set of all nonempty and closed subsets of A (with respect to the metric involved) is denoted by $\mathcal{P}_{nc}(A)$.

2.2. Metric spaces

Definition 2.1

A metric d on a set M is a mapping $d: M \times M \rightarrow \mathbb{R}$, which satisfies

(i) $\forall x,y \in M \quad d(x,y) >= 0$

(ii) $\forall x,y \in M \quad d(x,y) = d(y,x)$

(iii) $\forall x,y \in M \quad d(x,y) = 0 \Leftrightarrow x = y$

(iv) $\forall x,y,z \in M \quad d(x,z) <= d(x,y) + d(y,z)$

(M, d) is called a metric space.

Example

The discrete metric d on a set X is given by:

$d(x, x) = 0$

$d(x, y) = 1$ if $x \neq y$

for $x, y \in X$.

Definition 2.2

Let (M, d) be a metric space, $x \in M$.

a) A sequence $(x_i)_i$ in M is called a Cauchy sequence whenever

$\forall \epsilon > 0 \ \exists N \in \mathbb{N} \ \forall m,n > N \ d(x_m, x_n) < \epsilon$

b) A sequence $(x_i)_i$ converges to x whenever

$\forall \epsilon > 0 \ \exists N \in \mathbb{N} \ \forall n > N \ d(x, x_n) < \epsilon$

c) (M, d) is called complete whenever each Cauchy sequence converges to an element of M.

Definition 2.3

Let (M_1, d_1) and (M_2, d_2) be metric spaces.

A function $f: M_1 \to M_2$ is called contracting whenever

$\forall x,y \in M_1 \quad d_2(f(x), f(y)) \leqq \epsilon \cdot d(x,y) \qquad$ with $0 \leqq \epsilon < 1$

Proposition 2.4 (Banach's fixed point theorem)

Let (M, d) be a complete metric space ; $f : M \to M$ a contracting function. Then f has an unique fixed point. Or: $\exists ! x \in M \ [f(x)=x \ ]$.

Definition 2.5

A subset X of a complete metric space (M, d) is called closed when each Cauchy sequence in X converges to an element of X.

Definition 2.6

Let $(M, d),(M_1, d_1),\ldots,(M_n, d_n)$ be metric spaces.

a) We define $d_{M_1 \to M_2}$ as follows:

$\forall f_1, f_2 \in M_1 \to M_2 \quad d_{M_1 \to M_2}(f_1, f_2) = \sup_{x \in M_1} \{d_2(f_1(x), f_2(x))\}$

b) Let $M_1 \cup \ldots \cup M_n$ be the disjoint union of $M_1,\ldots,M_n$. We define $d_{M_1 \cup \ldots \cup M_n}$ as follows:

$\forall x,y \in M_1 \cup \ldots \cup M_n$

$d_{M_1 \cup \ldots \cup M_n}(x,y) = d_j(x,y)$ if $x,y \in M_j$, $1 \leqq j \leqq n$

$d_{M_1 \cup \ldots \cup M_n}(x,y) = 1$ otherwise

c) We define $d_{M_1 \times \ldots \times M_n}$ as follows:

$\forall (x_1,\ldots,x_n),(y_1,\ldots,y_n) \in M_1 \times \ldots \times M_n \quad d_{M_1 \times \ldots \times M_n}((x_1,\ldots,x_n),(y_1,\ldots,y_n)) = \max\{d_i(x_i,y_i)\}$

d) We define $d_{\mathcal{P}_{nc}(M)}$ as follows:

$\forall X,Y \in \mathcal{P}_{nc}(M) \quad d_{\mathcal{P}_{nc}(M)}(X,Y) = \max\{\sup_{x \in X}\{d(x,Y)\}, \sup_{y \in Y}\{d(y,X)\}\}$,

where $d(x,Z) = \inf_{z \in Z}\{d(x,z)\}$ for every $Z \subset M$, $x \in M$.

Proposition 2.7

Let $(M, d),(M_1, d_1),\ldots,(M_n, d_n)$ be as in the previous definition. Then

(i) $(M_1 \to M_2, d_{M_1 \to M_2})$,

(ii) $(M_1 \cup \ldots \cup M_n, d_{M_1 \cup \ldots \cup M_n})$,

(iii) $(M_1 \times \ldots \times M_n, d_{M_1 \times \ldots \times M_n})$,

(iv) $(\mathcal{P}_{nc}(M), d_{\mathcal{P}_{nc}(M)})$

are complete metric spaces.

2.3. Domain equations

A process is an element of a process domain. Process domains are obtained as solutions of domain equations. A general reflexive domain equation has the form:

$$P = F(P)$$

where F is a functor on the category of complete metric spaces. For an elaborate discussion of domain equations we refer the reader to [BZ] and [AB].

We only need nonessential variations of the following domain equation:

$$P = \{p_0\} \cup (A \rightarrow \mathcal{P}_{nc}(B \times id_{½}(P)))$$

The nil process is denoted by p_0. A and B are arbitrary sets. We assume for A and B the discrete metric. $id_{½}$ is a functor on the category of complete metric spaces given by:
$id_{½}(Q) = Q$, $id_{½}(d)(x,y) = ½ \cdot d(x,y)$.
An example of a process is $\lambda a.\{<b_1, p_1>, <b_2, p_2>\}$. On basis of an argument a this process can choose between b_1 as first step with resumption p_1 or b_2 as first step with resumption p_2.

The equation is solved as follows. Define

$$P_0 = (\{p_0\}, d_0)$$

$$P_{n+1} = (\{p_0\} \cup (A \rightarrow \mathcal{P}_{nc}(B \times id_{½}(P_n))), d_{n+1}) \quad \text{for } n \geqq 0$$

d_0 is the discrete metric. The metric d_{n+1} is defined along the lines of definition 2.6. We now put $(P_\omega, d_\omega) = (\bigcup_n P_n, \bigcup_n d_n)$ and define (P', d') as the completion of (P_ω, d_ω). The complete metric space (P', d') is the solution of the domain equation. For the proof we refer again to [BZ].

3. Concurrent evaluation of side-effects in an imperative language

An imperative language is a language that consists of sequences of commands. In this chapter we describe an imperative language with side-effects and applicative concurrency. We interpret the language in two ways. For each interpretation we give an operational and a metric denotational semantics and prove that they are equivalent.

A short outline of this chapter: in paragraph 3.1 we describe the syntax of the language, give an informal explanation and make clear in which two ways we view the language. In paragraph 3.2 the semantics corresponding with the first interpretation is given, in paragraph 3.3 the semantics corresponding with the second interpretation. Paragraph 3.4 concludes this chapter with a short comment.

3.1. Description of the language

3.1.1. Preliminary definitions

Definition 3.1

($m \in$) Icon is the syntactical set of integer constants.

($x \in$) Ivar is the syntactical set of integer variables.

($\xi \in$) Pvar is the syntactical set of procedure variables.

Definition 3.2

($\alpha \in$) V is the semantical set of integer values.

($\underline{w}\in$) W is the semantical set of boolean truth values {true, false}.

Definition 3.3

The state $\sigma\in\Sigma$ is a mapping from the integer variables to the integer values. Or: Σ = Ivar $\rightarrow$ V.

3.1.2. Syntax

Definition 3.4

A program $t\in$Prog is a pair $< D \mid s >$, with D$\in$Decl and $s\in$Stat. A declaration D$\in$Decl is a n-tuple $<\xi_1 \Leftarrow s_1^g, \ldots, \xi_n \Leftarrow s_n^g>$ or $<\xi_i \Leftarrow s_i^g>_i$, for short, with $\xi_i\in$Pvar and $s_i^g \in$Stat^g (guarded statements), for some $n \geqq 0$.

Definition 3.5

The class ($s\in$) Stat of statements is given by:

$s ::= x:=e \mid \xi \mid (s_1;s_2) \mid (s_1 \cup s_2) \mid$ if b then s_1 else s_2 fi

(where $x\in$Ivar and $\xi\in$Pvar)

The class ($s^g\in$) Stat^g of guarded statements is given by:

$s^g ::= x:=e^g \mid (s_1^g;s_2) \mid (s_1^g \cup s_2^g) \mid$ if B then s_1^g else s_2^g fi $\mid$ if β^g then s_1 else s_2 fi

Definition 3.6

The class ($e\in$) Exp of expressions is defined as follows:

$e ::= x \mid m \mid (s;e) \mid f(e_1,\ldots,e_n)$

(where $x\in$Ivar and $m\in$Icon)

The class ($e^g\in$) Exp^g of guarded expressions is defined as follows:

$e^g ::= x \mid m \mid (s^g;e) \mid f(e_1^g,\ldots,e_n^g)$

Definition 3.7

The class ($b\in$) Bexp of boolean expressions is defined as follows:

$b ::=$ true $\mid$ false $\mid$ not(b) $\mid$ and(b_1,b_2) $\mid$ or(b_1,b_2) $\mid$ relop(e_1,e_2)

(where relop is one of the following relational operators: $=, \neq, \leqq, \geqq, >, <$)

The class ($b^g\in$) Bexp^g of guarded boolean expressions is defined by:

$b^g ::=$ true $\mid$ false $\mid$ not(b^g) $\mid$ and(b_1^g,b_2^g) $\mid$ or(b_1^g,b_2^g) $\mid$ relop(e_1^g,e_2^g)

Definition 3.8

The class ($\epsilon\in$) Exp(ϵ) of nonsimple expressions is defined by:

$\epsilon ::= (s;e) \mid f(e_1,\ldots,e_{i-1},\epsilon_i,e_{i+1},\ldots,e_n)$

(where $1 \leqq i \leqq n$)

The class ($\epsilon^g \in$) Exp(ϵ^g) of guarded nonsimple expressions is defined by:

$\epsilon^g ::= (s^g ; e) \mid f(e_1^g, \ldots, e_{i-1}^g, \epsilon_i^g, e_{i+1}^g, \ldots, e_n^g)$

(where $1 \leqq i \leqq n$)

The class (E$\in$) Exp(E) of simple expressions is defined by:

$E ::= x \mid m \mid f(E_1, \ldots, E_n)$

Definition 3.9

The class ($\beta \in$) Bexp(β) of nonsimple boolean expressions is given by:

$\beta ::=$ relop(ϵ,e) | relop(e,ϵ) | and(β,b) | and(b,β) | or(β,b) | or(b,β) | not(β)

The class ($\beta^g \in$) Bexp(β^g) of guarded nonsimple boolean expressions is given by:

$\beta^g ::=$ relop(ϵ^g,e^g) | relop(e^g,ϵ^g) | and(β^g,b^g) | and(b^g,β^g) | or(β^g,b^g) | or(b^g,β^g) | *not*(β^g)

The class (B$\in$) Bexp(B) of simple boolean expressions is given by:

$B ::=$ true | false | relop(B_1,B_2) | and(B_1,B_2) | or(B_1,B_2) | not(B)

Lemma 3.10

a. The classes Exp(ϵ) and Exp(E) are disjoint and form together the class Exp.

b. The classes Bexp(β) and Bexp(B) are disjoint and form together the class Bexp.

Proof. Trivial.

3.1.3. Informal explanation

We have made use of the simultaneous declaration format for recursion. Instead of this we could have included $\mu\xi[s]$-constructs for recursion in the definition of the statements, the commands of this imperative language. However, the simultaneous declaration format makes the semantical equivalence proofs much easier.

The class of statements includes the assignment $x := e$ and the (possibly recursive) procedure call ξ. From other statements we construct statement composition ($s_1 ; s_2$), nondeterministic choice ($s_1 \cup s_2$) and the if-then-else statement. Notice nevertheless that an assignment $x := e$ possibly contains several substatements:

$x :=$ ($x := 2$; $y := 3$; $z := 1$; 2);

A condition b of an if-statement may also contain substatements:

if =($x := 2$; y, $y := 3$; z) then s_1 else s_2 fi

In the definition of a program we see that the body of a procedure in the declaration part is a guarded statement. A statement is guarded if none of its initial substatements is a (possibly recursive) procedure call. In this manner it is guaranteed that execution of a procedure call does not immediately proceed with just another procedure call. The guardedness requirement is needed later in lemma 3.22 and the semantical equivalence proof.

The class of expressions consists of the basic expressions m (an integer constant) and x (an integer

variable) and the composite expressions $(s;e)$ and $f(e_1,\ldots,e_n)$. The expressions are divided into the simple expressions which do not contain any statements and the nonsimple expressions which contain at least one statement. We consider any nonsimple expression as an expression with a side-effect. The expression $(s;e)$ called statement-expression composition provides the basis for side-effects. Evaluation of $(s;e)$ involves execution of s followed by evaluation of e.

In the expression $f(e_1,\ldots,e_n)$ f stands for a function from n-tuples of integers to integers. An order of evaluation is not imposed upon the arguments of f. The arguments are evaluated concurrently by merging their atomic actions. Concurrent evaluation of side-effects comes about if f has nonsimple arguments.

Because of the occurrence of statements in expressions, guardedness is also required for expressions. An expression is guarded if none of its initial substatements is a (possibly recursive) procedure call. A simple expression is thus guarded too.

The class of boolean expressions consist of the boolean constants true, false, the relational expression relop(e_1,e_2), not(b), and(b_1,b_2) and or(b_1,b_2). A boolean expression is nonsimple if it contains a statement, that is, if some subexpression is a relational expression relop(e_1,e_2) such that at least one of its arguments is a nonsimple expression. Again, the arguments of boolean functions are evaluated concurrently.

The condition of a guarded if-statement is a guarded boolean expression. If this guarded condition is simple, the statements following the condition must be guarded. If the condition is guarded and nonsimple, the statements following the condition do not need to be guarded. This explains the two instances of the if-statement in the definition of the guarded statements.

As said before, we interpret the language in two ways. The difference between these two interpretations is in the granularity of atomic actions. In the first interpretation the only atomic action is the assignment of a simple expression to a variable. The asssignment of a nonsimple expression to a variable now involves evaluation of the side-effects followed by the assignment of the resulting simple expression to the variable. Evaluation of a simple condition of an if-statement does not take a step.

In the second interpretation the evaluation of an integer constant, the evaluation of a boolean constant, the evaluation of an integer variable, the application of a function and the assignment of an integer value to a variable are atomic actions. As it turns out, this increases the possibilities of the language. We return to this in section 3.3.1.

This completes the informal description of the language.

3.2. Semantics of the first interpretation

A short outline of this paragraph: in section 3.2.1. the semantical valuations of simple (boolean) expressions are defined. Section 3.2.2. presents an operational semantics O_1. A denotational semantics D_1 is given in 3.2.3. A denotational semantics with continuations is provided in section 3.2.4. The semantical equivalence of O_1 and D_1 is proven in 3.2.5.

3.2.1. Preliminary definitions

In this section we define the semantical meaning of a simple expression and a simple boolean expression.

Definition 3.11

The semantical valuation V : Exp(E) $\rightarrow$ ($\Sigma \rightarrow$ V) is given by:

$V(m)(\sigma) = \alpha$ (where α is the integer denoted by m)

$V(x)(\sigma) = \sigma(x)$

$V(f(E_1,\ldots,E_n))(\sigma) = [f](V(E_1)(\sigma),\ldots,V(E_n)(\sigma))$

Remark. We denote by $[f]$ the semantical meaning of f, a function from V^n to V. We assume the intended meaning of f as known.

Definition 3.12

The semantical valuation W : Bexp(B)$\rightarrow(\Sigma\rightarrow W)$ is given by:

$W(\text{true})(\sigma) = \underline{\text{true}}$

$W(\text{false})(\sigma) = \underline{\text{false}}$

$W(\text{relop}(E_1,E_2)(\sigma) = [\text{relop}](V(E_1)(\sigma),V(E_2)(\sigma))$

$W(\text{not}(B))(\sigma) = \neg W(B)(\sigma)$

$W(\text{and}(B_1,B_2)(\sigma) = W(B_1)(\sigma) \wedge W(B_2)(\sigma)$

$W(\text{or}(B_1,B_2)(\sigma) = W(B_1)(\sigma) \vee W(B_2)(\sigma)$

Remark. [relop], the semantical counterpart of relop is a function from W^2 to W. Note that we used instead of [and],[or],[not] the logical symbols $\wedge, \vee, \neg$. They denote the wellknown functions from W^2 to W.

3.2.2. Operational semantics

An operational semantics is based on a transition relation $\rightarrow$. A transition relation is a relation between configurations. Suppose that c_1 and c_2 are configurations and $(c_1, c_2) \in \rightarrow$ (Notation: $c_1 \rightarrow c_2$). We then call $c_1 \rightarrow c_2$ a transition. The intuitive meaning of a transition is that one step can be taken from configuration c_1 to configuration c_2.

A transition relation is determined by a transition system. Formally, a transition system is a syntax directed deductive system for proving transitions. It consists of axioms that show what the basic transitions are, and rules for deducing new transitions from old ones. A transition relation consists of all provable transitions.

We now define the configurations of our operational semantics in

Definition 3.13

A configuration c $\in$ Conf is given by:

$c ::= \langle s, \sigma\rangle \mid \langle e, \sigma\rangle \mid \langle b, \sigma\rangle \mid \sigma$

where $s \in$ Stat, $e \in$ Exp, $b \in$ Bexp and $\sigma \in \Sigma$.

The used simultaneous declaration format requires that we define a transition relation for every declaration D. We therefore define a mapping T_1 : Decl $\rightarrow$ $\mathcal{P}$(Conf $\times$ Conf) such that T_1(D) is the transition relation determined by the transition system T_1(D). This transition relation is denoted by $\rightarrow_D$. We define the corresponding transition system in Definition 3.14.

The transition relation $\rightarrow_D$ consists of transitions of the form :

(1) $< s, \sigma > \rightarrow_D < s', \sigma' >$

(2) $< s, \sigma > \rightarrow_D \sigma'$

(3) $< e, \sigma > \rightarrow_D < e', \sigma' >$

(4) $< b, \sigma > \rightarrow_D < b', \sigma' >$.

The transitions have the following intuitive meaning : (1) Executing s one step changes the state from σ to σ', with s' still to be executed. (2) Executing s just takes one step, changing the state from σ to σ'. (3,4) Evaluation of a (boolean) expression e (b) one step changes the state from σ to σ', with e' (b') still to be evaluated.

We now define

Definition 3.14

The transition system $T_1(D)$ is defined by the following axiom and rules:

Axiom 1

$$< x := E, \sigma > \rightarrow_D \sigma\{V(E)(\sigma)/x\}$$

Rule 1

$$\frac{< s_1, \sigma_1 > \rightarrow_D < s_2, \sigma_2 > \mid \sigma_2}{< s_1 ; s, \sigma_1 > \rightarrow_D < s_2 ; s, \sigma_2 > \mid < s, \sigma_2 >}$$

$$< s_1 \cup s, \sigma_1 > \rightarrow_D < s_2, \sigma_2 > \mid \sigma_2$$

$$< s \cup s_1, \sigma_1 > \rightarrow_D < s_2, \sigma_2 > \mid \sigma_2$$

Rule 2

$$\frac{< s, \sigma > \rightarrow_D < s', \sigma' > \mid \sigma'}{< \xi, \sigma > \rightarrow_D < s', \sigma' > \mid \sigma'}$$

provided that $\xi = s$ in D

Rule 3

$$\frac{< s_1, \sigma > \rightarrow_D < s', \sigma' > \mid \sigma'}{< \text{if B then } s_1 \text{ else } s_2 \text{ fi}, \sigma > \rightarrow < s', \sigma' > \mid \sigma'}$$

provided that $W(B)(\sigma) = \underline{\text{true}}$

Rule 4

$$\frac{< s_2, \sigma > \rightarrow_D < s', \sigma' > \mid \sigma'}{< \text{if B then } s_1 \text{ else } s_2 \text{ fi}, \sigma > \rightarrow < s', \sigma' > \mid \sigma'}$$

provided that $W(B)(\sigma) = \underline{\text{false}}$

Rule 5

$$\frac{< s, \sigma > \rightarrow_D < s', \sigma' > \mid \sigma'}{< (s;e), \sigma > \rightarrow_D < (s';e), \sigma' > \mid < e, \sigma' >}$$

Rule 6

$$\frac{< e, \sigma> \rightarrow_D < e', \sigma'>}{< x:=e, \sigma> \rightarrow_D <x:=e', \sigma'>}$$

$$< f(e_1,\ldots,e_{i-1},e,e_{i+1},\ldots,e_n), \sigma> \rightarrow_D <f(e_1,\ldots,e_{i-1},e',e_{i+1},\ldots,e_n), \sigma'>$$

$$< \text{relop}(e,e_2), \sigma> \rightarrow_D < \text{relop}(e',e_2), \sigma'>$$

$$< \text{relop}(e_1,e), \sigma> \rightarrow_D < \text{relop}(e_1,e'), \sigma'>$$

Rule 7

$$\frac{< b, \sigma> \rightarrow_D < b', \sigma'>}{< \text{not}(b), \sigma> \rightarrow_D < \text{not}(b'), \sigma'>}$$

$$< \text{and}(b,b_2), \sigma> \rightarrow_D < \text{and}(b',b_2), \sigma'>$$

$$< \text{and}(b_1,b), \sigma> \rightarrow_D < \text{and}(b_1,b'), \sigma'>$$

$$< \text{or}(b,b_2), \sigma> \rightarrow_D < \text{or}(b',b_2), \sigma'>$$

$$< \text{or}(b_1,b), \sigma> \rightarrow_D < \text{or}(b_1,b'), \sigma'>$$

$$< \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma> \rightarrow_D < \text{if } b' \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma'>$$

Remark

We used

$$\frac{1 \rightarrow 2 \mid 3}{4 \rightarrow 5 \mid 6} \text{ as shorthand for } \frac{1 \rightarrow 2}{4 \rightarrow 5} \text{ and } \frac{1 \rightarrow 3}{4 \rightarrow 6},$$

$$\frac{1 \rightarrow 2}{3 \rightarrow 4} \text{ as shorthand for } \frac{1 \rightarrow 2}{3 \rightarrow 4} \text{ and } \frac{1 \rightarrow 2}{5 \rightarrow 6}.$$
$$5 \rightarrow 6$$

We call an assignment $x:=E$ a simple assignment. In this semantics a simple assignment is considered an atomic action. The only axiom represents the execution of this only atomic action. Every other transition eventually is based on this axiom. A transition sequence therefore represents in fact a sequence of simple assignments.

Rule 5 provides the basis for a transition $<e, \sigma> \rightarrow <e', \sigma'>$. In a similar way the two last conclusions of Rule 6 form the basis for satisfying the premiss of Rule 7.

Note further that for a configuration $<s,\sigma>$ there is always a transition $<s,\sigma> \rightarrow <s',\sigma'>|\sigma'$. For a configuration $<\epsilon,\sigma>$ there is a transition $<\epsilon,\sigma> \rightarrow <e',\sigma'>$ and for a configuration $<\beta,\sigma>$ a transition $<\beta,\sigma> \rightarrow <b',\sigma'>$.

We now proceed with the definition of the operational meaning of a program $t \equiv < D \mid s >$. We need the process domain given by the following domain equation:

$$P = \{p_0\} \cup (\Sigma \rightarrow \mathcal{P}_{nc}(\Sigma \times \text{id}_{½}(P)))$$

We denote by p a typical element of P. A typical element of $P\backslash\{p_0\}$ is denoted by p in italic font. We now define

Definition 3.15

a. The mapping O_1 : Prog $\rightarrow$ P is given by : $O_1 [< D \mid s >] = O_D [s]$

b. The mapping O_D : Stat $\rightarrow$ P is given by :

$$O_D [s] = \lambda\sigma.(\{< \sigma', O_D [s'] > \mid < s, \sigma> \rightarrow_D < s', \sigma'>\} \cup \{< \sigma', p_0> \mid < s, \sigma> \rightarrow_D \sigma' \})$$

Lemma 3.16

Let the operator Φ : Decl $\rightarrow$ ((Stat $\rightarrow$ P) $\rightarrow$ (Stat $\rightarrow$ P)) be defined as follows :

For N: Stat $\rightarrow$ P, D$\in$ Decl and $s \in$ Stat

$$\Phi(D)(N)(s)=\lambda\sigma.(\{< \sigma', N(s')> \mid < s, \sigma> \rightarrow_D < s', \sigma'>\} \cup \{< \sigma', p_0> \mid < s, \sigma> \rightarrow_D \sigma'\})$$

Then $\Phi(D)$ is contracting and $\Phi(D)$ has O_D as its fixed point.

Proof

We first prove that $\Phi(D)$ is contracting :

$$d_{Stat\rightarrow P} (\Phi(D)(N_1), \Phi(D)(N_2)) \leqq a \cdot d_{Stat\rightarrow P} (N_1, N_2) \qquad (\text{with } 0 \leqq a < 1)$$

$d_{Stat\rightarrow P} (\Phi(D)(N_1), \Phi(D)(N_2)) = \sup_s d_P (\Phi(D)(N_1)(s), \Phi(D)(N_2)(s)) =^{df}$

$\sup_s d_P (p_1, p_2) = \sup_s \sup_\sigma d_{P_{nc}(\Sigma \times id\,(P))} (p_1(\sigma), p_2(\sigma)) =$

$\sup_s \sup_\sigma \max (\sup_{< \sigma_1, p> \in p_1(\sigma)} \inf_{< \sigma_2, p'> \in p_2(\sigma)} d_{\Sigma \times id\,(P)} (< \sigma_1, p>, < \sigma_2, p'>) ,$

$\sup_{< \sigma_2, p'> \in p_2(\sigma)} \inf_{< \sigma_1, p> \in p_1(\sigma)} d_{\Sigma \times id\,(P)} (< \sigma_2, p'>, < \sigma_1, p>)) =$

$\sup_s \sup_\sigma \max (\sup_{< \sigma_1, p> \in p_1(\sigma)} \inf_{< \sigma_1, p'> \in p_2(\sigma)} d_{\Sigma \times id\,(P)} (< \sigma_1, p>, < \sigma_1, p'>) ,$

$\sup_{< \sigma_2, p'> \in p_2(\sigma)} \inf_{< \sigma_2, p> \in p_1(\sigma)} d_{\Sigma \times id\,(P)} (< \sigma_2, p'>, < \sigma_2, p>)) \leqq$

$\sup_s \sup_\sigma \max (\sup_{< \sigma_1, N_1(s_1)> \in p_1(\sigma)} \tfrac{1}{2} \cdot d_P (N_1(s_1), N_2(s_1)) ,$

$\sup_{< \sigma_2, N_2(s_2)> \in p_2(\sigma)} \tfrac{1}{2} \cdot d_P (N_2(s_2), N_1(s_2))) =$

$\tfrac{1}{2} \cdot d_{Stat\rightarrow P} (N_1, N_2)$

where

$$p_1 = \lambda\sigma.(\{<\sigma', N_1(s')> \mid <s, \sigma> \rightarrow_D <s', \sigma'>\} \cup \{<\sigma', p_0> \mid <s, \sigma> \rightarrow_D \sigma'\})$$

and

$$p_2 = \lambda\sigma.(\{<\sigma', N_2(s')> \mid <s, \sigma> \rightarrow_D <s', \sigma'>\} \cup \{<\sigma', p_0> \mid <s, \sigma> \rightarrow_D \sigma'\})$$

Before we can use Banach's theorem to prove that $\Phi(D)$ has O_D as its unique fixed point, we must prove that (Stat$\rightarrow$P, $d_{Stat\rightarrow P}$) is a complete metric space.

We define d_{Stat} as the discrete metric on Stat. (Stat, d_{Stat}) now is a complete metric space; every Cauchy sequence converges. We refer the reader to [BZ] for completeness of (P, d_P) and now apply proposition 2.7 to prove that (Stat$\rightarrow$P, $d_{Stat\rightarrow P}$) is complete.

Banach's theorem now gives us that $\Phi(D)$ has a unique fixed point. As can be seen, O_D is that fixed point;

$$\forall s \quad \Phi(D)(O_D)(s) = \lambda\sigma.(\{< \sigma', O_D(s')> \mid < s, \sigma> \rightarrow_D < s', \sigma'>\} \cup \{< \sigma', p_0> \mid < s, \sigma> \rightarrow_D \sigma'\})$$
$$= O_D(s)$$

Thus:

$$\Phi(D)(O_D) = O_D$$

□

We also define the operational meaning of an expression and a boolean expression. We need them when proving semantical equivalence with the denotational semantics D_1.

First we define the process domains Q and R by the domain equations:

$$Q = (\Sigma \to V) \cup (\Sigma \to \mathcal{P}_{nc}(\Sigma \times id_{1/2}(Q)))$$

$$R = (\Sigma \to W) \cup (\Sigma \to \mathcal{P}_{nc}(\Sigma \times id_{1/2}(R)))$$

A typical element of Q resp. R is denoted by q resp. r. A typical element of $(\Sigma \to V)$ resp. $(\Sigma \to W)$ is denoted by τ resp. v. We denote by q resp. r in italic font typical elements of $Q\backslash(\Sigma \to V)$ resp. $R\backslash(\Sigma \to W)$.

Definition 3.17

The mapping F_D : Exp → Q is given by :

$$F_D(\epsilon) = \lambda\sigma.\{< \sigma', F_D(e') > \mid < \epsilon, \sigma > \to_D < e', \sigma' >\}$$

$$F_D(E) = \lambda\sigma.V(E)(\sigma)$$

Lemma 3.18

Let Φ_e : Decl → ((Exp → Q) → (Exp → Q)) be given by :

For G : Exp → Q and D∈ Decl

$$\Phi_e(D)(G)(\epsilon) = \lambda\sigma.\{< \sigma', G(e') > \mid < \epsilon, \sigma > \to_D < e', \sigma' >\}$$

$$\Phi_e(D)(G)(E) = \lambda\sigma.V(E)(\sigma)$$

Then $\Phi_e(D)$ is contracting and $\Phi_e(D)$ has F_D as its fixed point.

Definition 3.19

The mapping A_D : Bexp → R is given by :

$$A_D(\beta) = \lambda\sigma.\{< \sigma', A_D(b') > \mid < \beta, \sigma > \to_D < b', \sigma' >\}$$

$$A_D(B) = \lambda\sigma.W(B)(\sigma)$$

Lemma 3.20

Let Φ_b : Decl → ((Bexp → R) → (Bexp → R)) be given by :

For C : Bexp → R and D∈ Decl

$$\Phi_b(D)(C)(\beta) = \lambda\sigma.\{< \sigma', C(b') > \mid < \beta, \sigma > \to_D < b', \sigma' >\}$$

$$\Phi_b(D)(C)(B) = \lambda\sigma.W(B)(\sigma)$$

Then $\Phi_b(D)$ is contracting and $\Phi_b(D)$ has A_D as its fixed point.

The lemmas 3.18 and 3.20 can be proven in a similar way as lemma 3.16.

3.2.3. Denotational semantics

According to [BZ], a denotational semantics is characterized by : (i) the systematic use of mathematical models which are used as range for the valuation mappings assigning meaning to the various programming constructs, (ii) the systematic way of adhering to the compositionality principle.

In the following denotational semantics the mathematical framework of complete metric spaces is used. Moreover, it is compositional in the sense that the meaning of a composite construct is built up from the meanings of its components.

First we define Env = Pvar $\rightarrow$ P to be the set of environments γ, i.e. mappings from procedure variables to their meanings. Furthermore we recall

$$P=\{p_0\}\cup(\Sigma\rightarrow\mathcal{P}_{nc}(\Sigma\times id_{\frac{1}{2}}(P)))$$

$$Q=(\Sigma\rightarrow V)\cup(\Sigma\rightarrow\mathcal{P}_{nc}(\Sigma\times id_{\frac{1}{2}}(Q)))$$

$$R=(\Sigma\rightarrow W)\cup(\Sigma\rightarrow\mathcal{P}_{nc}(\Sigma\times id_{\frac{1}{2}}(R)))$$

We now define the meaning of a program.

Definition 3.21

The mapping M_1 : Prog $\rightarrow$ P is given by:

$M_1[< D \mid s >] = D_1(\gamma_D)(s)$,

where $\gamma_D = \gamma\{p_i/\xi_i\}_{i=1}^n$

where , for $D \equiv < \xi_i \Leftarrow s_i^g >_i$, we put $<p_1,\ldots,p_n>$ = fixed point $<\Phi_1,\ldots,\Phi_n>$

where $\Phi_j : P^n \rightarrow P$ is given by:

$\Phi_j(p_1') \cdots (p_n') = D_1(\gamma\{p_i'/\xi_i\}_i)(s_j^g)$

Lemma 3.22

The unique fixed point (in Definition 3.21) exists by the guardedness requirement which ensures contractivity of Φ_j $(1 \leqq j \leqq n)$.

Outline of the proof

The function $<\Phi_1,\ldots,\Phi_n>:P^n\rightarrow P^n$, given by

$$<\Phi_1,\ldots,\Phi_n>(p_1,\ldots,p_n) = <\Phi_1(p_1,\ldots,p_n),\ldots,\Phi_n(p_1,\ldots,p_n)>,$$

is contracting.

$$d_{P^n}(<\Phi_1(p_1,\ldots,p_n),\ldots,\Phi_n(p_1,\ldots,p_n)>, <\Phi_1(q_1,\ldots,q_n),\ldots,\Phi_n(q_1,\ldots,q_n)>) =$$

$$\max_i d_P(\Phi_i(p_1,\ldots,p_n),\Phi_i(q_1,\ldots,q_n)) =$$

$$\max_i d_P(D_1(\gamma\{p_j/\xi_j\}_j)(s_i^g), D_1(\gamma\{q_j/\xi_j\}_j)(s_i^g)) \leqq \qquad (*)$$

$$\tfrac{1}{2} \cdot \max_k d_P(p_k,q_k) = \tfrac{1}{2} \cdot d_{P^n}((p_1,\ldots,p_n),(q_1,\ldots,q_n))$$

Furthermore, (P^n, d_{P^n}) is a complete metric space. We now use Banach's theorem to prove that $<\Phi_1,\ldots,\Phi_n>$ has a unique fixed point.

Remark The step (*) can be proven in detail by induction on the complexity of s_1^g. We refer the reader to [ABKR2] for similar proofs.

We now give the definition of the denotational semantics D_1. The definition of the various used operators is given in the definitions 3.24 - 3.27.

Definition 3.23

a. The mapping D_1 : Env → (Stat → P) is given by:

$$D_1(\gamma)(s_1;s_2) = D_1(\gamma)(s_2) \circ_1 D_1(\gamma)(s_1)$$
$$D_1(\gamma)(s_1 \cup s_2) = D_1(\gamma)(s_1) \cup D_1(\gamma)(s_2)$$
$$D_1(\gamma)(\xi) = \gamma(\xi)$$
$$D_1(\gamma)(x:=e) = \mathrm{F}(x, E_1(\gamma)(e))$$
$$D_1(\gamma)(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) = [[B_1(\gamma)(b), D_1(\gamma)(s_1), D_1(\gamma)(s_2)]]$$

b. The mapping E_1 : Env → (Exp → Q) is defined by:

$$E_1(\gamma)(x) = \lambda\sigma.\sigma(x)$$
$$E_1(\gamma)(m) = \lambda\sigma.V(m)(\sigma)$$
$$E_1(\gamma)((s;e)) = E_1(\gamma)(e) \circ_2 D_1(\gamma)(s)$$
$$E_1(\gamma)(f(e_1, \ldots, e_n)) = \underline{f}([E_1(\gamma)(e_1),\ldots,E_1(\gamma)(e_n)])$$

c. The mapping B_1 : Env → (Bexp → R) is defined by :

$$B_1(\gamma)(\text{true}) = \lambda\sigma.\underline{\text{true}}$$
$$B_1(\gamma)(\text{false}) = \lambda\sigma.\underline{\text{false}}$$
$$B_1(\gamma)(\text{not}(b)) = \underline{\text{not}}(B_1(\gamma)(b))$$
$$B_1(\gamma)(\text{relop}(e_1,e_2)) = \underline{\text{relop}}([E_1(\gamma)(e_1),E_1(\gamma)(e_2)])$$
$$B_1(\gamma)(\text{and}(b_1,b_2)) = \underline{\text{and}}([B_1(\gamma)(b_1),B_1(\gamma)(b_2)]_2)$$
$$B_1(\gamma)(\text{or}(b_1,b_2)) = \underline{\text{or}}([B_1(\gamma)(b_1),B_1(\gamma)(b_2)]_2)$$

We proceed with the definitions of the operators in the above definition. We start with the definition of composition, union and the assignment-operator.

Definition 3.24

a. The (first) composition operator $\circ_1$: P×P → P is given by:

$$p \circ_1 \bar{p} = \lambda\sigma. \bigcup_{<\sigma',p'> \in \bar{p}(\sigma)} \{<\sigma', p \circ_1 p'>\}$$
$$p \circ_1 p_0 = p$$

b. The union operator $\cup$: P×P → P is defined by:

$p \cup p'$ is the set-theoretic union in case $p \neq p_0$ and $p' \neq p_0$. $p \cup p_0 = p_0 \cup p = p$.

c. The assignment-operator F : Ivar × Q → P is defined by:

$$F(x,q) = \lambda\sigma.\bigcup_{<\sigma',q'>\in q(\sigma)} \{<\sigma',F(x,q')>\}$$
$$F(x,\tau) = \lambda\sigma.\{<\sigma\{\tau(\sigma)/x\},p_0>\}$$

d. The second composition operator $\circ_2$: $Q\times P \to Q$ is given by:

$$q \circ_2 p = \lambda\sigma.\bigcup_{<\sigma',p'>\in p(\sigma)} \{<\sigma',q\circ_2 p'>\}$$
$$q \circ_2 p_0 = q$$

We omit the proof that the above operators are well-defined. We now come to the definition of the 'merge'-operator [$\cdot\cdot\cdot$] for handling concurrent evaluation of the arguments of a function or a boolean function. Therefore we need two intermediary domains:

$$(\phi\in)\ Q_n' = (\Sigma\to V^n)\cup(\Sigma\to P_{nc}(\Sigma\times id_{½}(Q_n'))),$$
$$R' = (\Sigma\to W\times W)\cup(\Sigma\to P_{nc}(\Sigma\times id_{½}(R'))).$$

Definition 3.25

a. The merge operator [$\cdot\cdot\cdot$]: $Q^n\to Q_n'$ is given by:

$$[q_1,\ldots,q_n] = \lambda\sigma.\{<\sigma',[q_1,\ldots,q_i',\ldots,q_n]> \mid 1\leqq i\leqq n,\ q_i\notin\Sigma\to V,\ <\sigma',q_i'>\in q_i(\sigma)\}$$
where $\exists\ j$ such that $q_j \notin \Sigma\to V$.
$$[\tau_1,\ldots,\tau_n] = \lambda\sigma.(\tau_1(\sigma),\ldots,\tau_n(\sigma)).$$

b. The operator [$\cdot\cdot\cdot,\cdot\cdot\cdot$]$_2$: $R\times R\to R'$ is given by:

$$[r_1,r_2]_2 = \lambda\sigma.(\{<\sigma',[r_1',r_2]_2> \mid r_1\notin\Sigma\to W,\ <\sigma',r_1'>\in r_1(\sigma)\}$$
$$\cup\{<\sigma',[r_1,r_2']_2> \mid r_2\notin\Sigma\to W,\ <\sigma',r_2'>\in r_2(\sigma)\})$$
where $\exists\ j$ such that $r_j\notin\Sigma\to W$.
$$[v_1,v_2]_2 = \lambda\sigma.(v_1(\sigma),v_2(\sigma))$$

Examples

$$[\lambda\sigma.\{<\sigma\{1/x\},\lambda\sigma.\{<\sigma\{2/y\},\lambda\sigma.3>\}>\},\ \lambda\sigma.\{<\sigma\{4/x\},\lambda\sigma.5>\}] =$$
$$\lambda\sigma.\{<\sigma\{1/x\},\lambda\sigma.\{<\sigma\{2/y\},\lambda\sigma.\{<\sigma\{4/x\},\lambda\sigma.(3,5)>\}>,$$
$$<\sigma\{4/x\},\lambda\sigma.\{<\sigma\{2/y\},\lambda\sigma.(3,5)>\}>\}>,$$
$$<\sigma\{4/x\},\lambda\sigma.\{<\sigma\{1/x\},\lambda\sigma.\{<\sigma\{2/y\},\lambda\sigma.(3,5)>\}>\}>\}.$$

$$[\lambda\sigma.\sigma(z),\ \lambda\bar{\sigma}.4,\ \lambda\sigma'.\sigma'(y)] = \lambda\tilde{\sigma}.(\tilde{\sigma}(z),\ 4,\ \tilde{\sigma}(y))$$

Operational spirit of the merge. The way E handles the construct $f(e_1,\ldots,e_n)$ is denotational: (i) the meaning of the construct is composed of the meanings of its subcomponents $f,e_1,e_2,\ldots,e_n$ and (ii) in giving several constructs a meaning we use a nice methematical model. However, a closer look at our

merge-operator reveals more. The intuition behind this merge-operator is rather operational. In merging x processes we nondeterministically choose a process to take one single step and go from state σ to σ'. (Compare this with a transition). After that we still have to merge the resumption of the chosen process and the $n-1$ other processes. Our merge operator now takes all possible 'choices' together and collects them in a resulting process, the merge of the n processes. (Compare this with taking all transition sequences together to form the operational meaning).

About the last step of the merge. The last step consists of applying the second clause of Definition 3.25a: $[\tau_1, \ldots, \tau_n] = \lambda\sigma.(\tau_1(\sigma), \ldots, \tau_n(\sigma))$. Each τ_i delivers a value in V, when given a state σ. What happens here? The same state σ is given to every τ_i. This corresponds to the immediate evaluation of the resulting simple expressions in the same state. That state is the state reached after evaluation of the side-effects.

We now turn to the meaning of the underlined f,relop,and,or,not. In what is mentioned below (including definition 3.26) we restrict ourselves to the underlined f. The case of the underlined boolean functions is handled very similarly. In fact, the underline is an operator with as first argument a function f and as second argument an element of Q_n'. We use the shorthand $\underline{f}$.

Definition 3.26

$\underline{f} : Q_n' \to Q$ is defined as follows:

for $\phi \in Q_n' \setminus (\Sigma \to V^n)$

$\underline{f}(\phi) = \lambda\sigma. \bigcup_{<\sigma',\phi'> \in \phi(\sigma)} \{<\sigma', \underline{f}(\phi')>\}$

for $\phi \in (\Sigma \to V^n)$

$\underline{f}(\phi) = \lambda\sigma.[f](\phi(\sigma))$

We did not yet define the semantic if-then-else operator:

Definition 3.27

The operator $[[\ldots, \ldots, \ldots]] : R \times P \times P \to P$ is defined by:

- $[[r,p_1,p_2]] = \lambda\sigma. \bigcup_{<\sigma',\bar{r}> \in r(\sigma)} \{<\sigma',[[\bar{r},p_1,p_2]]>\}$
- $[[v,p_1,p_2]] = \lambda\sigma.$ if $v(\sigma)$ then $p_1(\sigma)$ else $p_2(\sigma)$ fi,

where $p_1 \neq p_0$ and $p_2 \neq p_0$.

- $[[v,p_0,p]] = \lambda\sigma.$ if $v(\sigma)$ then $\{<\sigma,p_0>\}$ else $p(\sigma)$ fi,

where $p \neq p_0$.

- $[[v,p,p_0]] = \lambda\sigma.$ if $v(\sigma)$ then $p(\sigma)$ else $\{<\sigma,p_0>\}$ fi,

where $p \neq p_0$.

- $[[v,p_0,p_0]] = \lambda\sigma. \{<\sigma,p_0>\}$

The last clause in this definition could also have been $[[\cdots]] = p_0$. The choice is somewhat arbitrary. The following example explains our choice.

Example

$[[\lambda\sigma.\underline{\text{false}},p,p_0]] = \lambda\sigma.$if $\underline{\text{false}}$ then $p(\sigma)$ else $\{<\sigma,p_0>\}$ fi $= \lambda\sigma.\{<\sigma,p_0>\}$

$$[[\lambda\sigma.\underline{\text{false}},p_0,p_0]] = \lambda\sigma.\{<\sigma,p_0>\}$$

The outcome in this case of equal first argument $\lambda\sigma.\underline{\text{false}}$ and third argument p_0 now is not dependent on the second argument. The idea here is that if you see that the boolean condition of the if-construct always evaluates to false, you can ignore the second argument, the 'then'-part.

Remark

The last three clauses of the previous definition are actually not used. The semantical meaning of a statement is never equal to the empty process p_0.

Before proving semantic equivalence of O_1 and M_1 in section 3.2.5, we describe in 3.2.4 another (equivalent) denotational semantics M_1'.

3.2.4. Denotational semantics with continuations

A continuation describes what will happen after the execution of the current statement (or (boolean) expression). It is given as argument to the semantical functions. The continuation method provides an elegant mechanism to handle compositional constructs. Several compositional operators of the previous section have now been eliminated.

We recall

$$P = \{p_0\} \cup (\Sigma \to \mathcal{P}_{nc}(\Sigma \times id_{\frac{1}{2}}(P)))$$

$$Env = Pvar \to P$$

We define

Definition 3.28

The mapping $M_1' : Prog \to P$ is given by:

$$M_1'[< D \mid s >] = D_1'(\gamma_D)(s)(p_0) ,$$

where $\gamma_D = \gamma\{p_i/\xi_i\}_{i=1}^n$

where , for $D \equiv < \xi_i \Leftarrow s_i^g >_i$, we put $<p_1,\ldots,p_n> =$ fixed point $<\Phi_1,\ldots,\Phi_n>$

where $\Phi_j : P^n \to P$ is given by:

$$\Phi_j(p_1') \cdots (p_n') = D_1'(\gamma\{p_i'/\xi_i\}_i)(s_j^g)(p_0)$$

Lemma 3.29

The unique fixed point in the above definition exists by the guardedness requirement which ensures contractivity of the Φ_j.

The class of continuations ($p \in$) $Cont_s$ is equal to P. The class of expression continuations is given by

$$(c_e \in)\ Cont_e = ((\Sigma \to V) \to P) \cup \{i_e\}$$

The class of boolean expression continuations is given by

$$(c_b \in)\ Cont_b = ((\Sigma \to W) \to P) \cup \{i_b\}$$

i_e and i_b denote the empty expression continuation respectively the empty boolean expression continuation. i_e is given by

$$i_e : (\Sigma \to V) \to (\Sigma \to V) \text{ such that } i_e(\tau) = \tau$$

Analogously, i_b is given by

$$i_b : (\Sigma \to W) \to (\Sigma \to W) \text{ such that } i_b(\upsilon) = \upsilon$$

We recall

$$Q = (\Sigma \to V) \cup (\Sigma \to \mathcal{P}_{nc}(\Sigma \times id_{\frac{1}{2}}(Q)))$$

$$R = (\Sigma \to W) \cup (\Sigma \to \mathcal{P}_{nc}(\Sigma \times id_{\frac{1}{2}}(R)))$$

We now define

Definition 3.30

a. The mapping $D_1' : Env \to Stat \to Cont_s \to P$ is given by:

$$D_1'(\gamma)(s_1;s_2)(p) = D_1'(\gamma)(s_1)(D_1'(\gamma)(s_2)(p))$$

$$D_1'(\gamma)(s_1 \cup s_2)(p) = D_1'(\gamma)(s_1)(p) \cup D_1'(\gamma)(s_2)(p)$$

$$D_1'(\gamma)(\xi)(p) = C(\gamma(\xi))(p)$$

$$D_1'(\gamma)(x:=e)(p) = E_1'(\gamma)(e)(\lambda\tau.\lambda\sigma.\{<\sigma\{\tau(\sigma)/x\}, p>\})$$

$$D_1'(\gamma)(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi})(p) = B_1'(\gamma)(b)(\lambda\upsilon.\lambda\sigma.\text{if } \upsilon(\sigma) \text{ then } D_1'(\gamma)(s_1)(p)(\sigma) \text{ else } D_1'(\gamma)(s_2)(p)(\sigma))$$

b. The mapping $E_1' : Env \to Exp \to ((((\Sigma \to V) \to P) \to P) \cup \{i_e\} \to Q)$ is defined by:

$$E_1'(\gamma)(x)(c_e) = c_e(\lambda\sigma.\sigma(x))$$

$$E_1'(\gamma)(m)(c_e) = c_e(\lambda\sigma.V(m)(\sigma))$$

$$E_1'(\gamma)((s;e))(c_e) = D_1'(\gamma)(s)(E_1'(\gamma)(e)(c_e))$$

$$E_1'(\gamma)(f(e_1,\ldots,e_n))(c_e) = \underline{f}([E_1'(\gamma)(e_1)(i_e),\ldots,E_1'(\gamma)(e_n)(i_e)])(c_e)$$

c. The mapping $B_1' : Env \to Bexp \to ((((\Sigma \to W) \to P) \to P) \cup \{i_b\} \to R)$ is defined by:

$$B_1'(\gamma)(\text{true})(c_b) = c_b(\lambda\sigma.\underline{\text{true}})$$

$$B_1'(\gamma)(\text{false})(c_b) = c_b(\lambda\sigma.\underline{\text{false}})$$

$$B_1'(\gamma)(\text{not}(b))(c_b) = \underline{\text{not}}(B_1'(\gamma)(b)(i_b))(c_b)$$

$$B_1'(\gamma)(\text{relop}(e_1,e_2))(c_b) = \underline{\text{relop}}([E_1'(\gamma)(e_1)(i_e), E_1'(\gamma)(e_2)(i_e)])(c_b)$$

$$B_1'(\gamma)(\text{and}(b_1,b_2))(c_b) = \underline{\text{and}}([B_1'(\gamma)(b_1)(i_b), B_1'(\gamma)(b_2)(i_b)]_2)(c_b)$$

$$B_1'(\gamma)(\text{or}(b_1,b_2))(c_b) = \underline{\text{or}}([B_1'(\gamma)(b_1)(i_b), B_1'(\gamma)(b_2)(i_b)]_2)(c_b)$$

The definition of the union operator $\cup : P \times P \to P$ has been given in definition 3.24b. The definition of the merge operator $[\cdots] : Q^n \to Q_n'$ has been given in definition 3.25, just as the definition of the merge operator $[\cdots]_2 : R \times R \to R'$. The definition of the if-then-else operator is assumed to be known. Note that $D_1'(\gamma)(s)(p)$ is never equal to p_0.

The operator C is defined in

Definition 3.31

The operator C: P × P → P is given by:

$C(\bar{p})(p) = \lambda\sigma.\ \bigcup_{<\sigma', p'> \in \bar{p}(\sigma)} \{<\sigma', C(p')(p)>\}$

$C(p_0)(p) = p$

The reader may have noticed that C(p)(p') actually is $p' \circ_1 p$, as defined in definition 3.24a.

Recall that

$$(\phi \in)\ Q_n' = (\Sigma \to V^n) \cup (\Sigma \to \mathcal{P}_{nc}(\Sigma \times \mathrm{id}_{\frac{1}{2}}(Q_n')))$$

The underline operator is now given by the following definition. Again, we only consider the case of the function f.

Definition 3.32

$\underline{f} : Q_n' \to ((((\Sigma \to V) \to P) \to P) \cup \{i_e\} \to Q)$ is defined by:

if $\phi \in Q_n' \setminus (\Sigma \to V^n)$

$\underline{f}(\phi)(c_e) = \lambda\sigma.\ \bigcup_{<\sigma', \phi'> \in \phi(\sigma)} \{<\sigma', \underline{f}(\phi')(c_e)>\}$

if $\phi \in (\Sigma \to V^n)$

$\underline{f}(\phi)(c_e) = c_e(\lambda\sigma.[f](\phi(\sigma)))$

Finally, we state without proving it: $M_1'(t) = M_1(t)$ for $t \in$ Prog

3.2.5. Semantic equivalence

O_1 and D_1 have been defined so that the semantic equivalence proof now rather exactly can follow the method as described in [KR, BM].

Theorem 3.33 For all t ∈ Prog : $O_1[\,t\,] = M_1[\,t\,]$.

Proof

It is sufficient to show that

●1 $E_1(\gamma_D)(e) = \Phi_e(D)(E_1(\gamma_D))(e)$,

●2 $B_1(\gamma_D)(b) = \Phi_b(D)(B_1(\gamma_D))(b)$,

●3 $D_1(\gamma_D)(s) = \Phi(D)(D_1(\gamma_D))(s)$.

We prove this by simultaneous induction on the complexity of (boolean) expressions and statements. The proof proceeds in two stages; first for guarded (boolean) expressions and statements, and next for general (boolean) expressions and statements.

Measure of complexity.

The complexity of a statement s, a boolean expression b and an expression e is the number of operators (composition, union) plus the number of functional constructs (if-then-else, functions f, boolean functions) it consists of. We write $c(n)$ for a language construct c with complexity n.

Stage 1.

Basis for induction.

●1

$e^g(0) ::= m \mid x$

$\Phi_e(\mathrm{D})(E_1(\gamma_\mathrm{D}))(x)=\lambda\sigma.V(x)(\sigma)=\lambda\sigma.\sigma(x)=E_1(\gamma_\mathrm{D})(x)$

$\Phi_e(\mathrm{D})(E_1(\gamma_\mathrm{D}))(m)=\lambda\sigma.V(m)(\sigma)=E_1(\gamma_\mathrm{D})(m)$

●2

$b^g(0) ::= \mathrm{true} \mid \mathrm{false}$

$\Phi_b(\mathrm{D})(B_1(\gamma_\mathrm{D}))(\mathrm{true})=\lambda\sigma.W(\mathrm{true})(\sigma)=B_1(\gamma_\mathrm{D})(\mathrm{true})$

$\Phi_b(\mathrm{D})(B_1(\gamma_\mathrm{D}))(\mathrm{false})=B_1(\gamma_\mathrm{D})(\mathrm{false})$

●3

$s^g ::= x:=e^g(0)$

$\Phi(\mathrm{D})(D_1(\gamma_\mathrm{D}))(x := e^g(0)) = \lambda\sigma.\ \{<\sigma', \mathrm{p}_0> \mid <x := e^g(0), \sigma> \rightarrow_\mathrm{D} \sigma'\} =$

$\mathrm{F}(x, \lambda\sigma.V(e^g(0))(\sigma)) = D_1(\gamma_\mathrm{D})(x := e^g(0))$

Induction hypothesis: Assume the hypothesis for complexity less than n.

Induction step: We prove the hypothesis for complexity equal to n.

●1

$\Phi_e(\mathrm{D})(E_1(\gamma_\mathrm{D}))((s^g\ ; e)) = \lambda\sigma.\ \{<\sigma', E_1(\gamma_\mathrm{D})(e')> \mid <(s^g\ ; e), \sigma> \rightarrow_\mathrm{D} <e', \sigma'>\} =$ (*1)

$\lambda\sigma.\ \{<\sigma', E_1(\gamma_\mathrm{D})(e) \circ_2 D_1(\gamma_\mathrm{D})(s')> \mid <s^g, \sigma> \rightarrow_\mathrm{D} <s', \sigma'>\} =$

$E_1(\gamma_\mathrm{D})(e) \circ \lambda\sigma.\ \{<\sigma', D_1(\gamma_\mathrm{D})(s')> \mid <s^g, \sigma> \rightarrow_\mathrm{D} <s', \sigma'>\} =$ (ind)

$E_1(\gamma_\mathrm{D})(e) \circ D_1(\gamma_\mathrm{D})(s^g) = E_1(\gamma_\mathrm{D})(s^g\ ; e)$

(*1) Case A is $e' \equiv s'\ ; e$ and case B is $e' \equiv e$. We treat case A ; case B is similar.

$\Phi_e(D)(E_1(\gamma_D))(f(e_1^g,\ldots,e_n^g)) =$

$\lambda\sigma.\ \{<\sigma', E_1(\gamma_D)(e')> \mid <f(e_1^g,\ldots,e_n^g), \sigma> \rightarrow_D <e', \sigma'>\} =$ (*2)

$\lambda\sigma.\ \{<\sigma', f([E_1(\gamma_D)(e_1^g),\ldots,E_1(\gamma_D)(e_i'),\ldots,E_1(\gamma_D)(e_n^g)])> \mid <f(e_1^g,\ldots,e_n^g), \sigma> \rightarrow_D <e', \sigma'>\} =$

$\lambda\sigma.\ \{<\sigma', f([E_1(\gamma_D)(e_1^g),\ldots,E_1(\gamma_D)(e_i'),\ldots,E_1(\gamma_D)(e_n^g)])> \mid <e_i^g, \sigma> \rightarrow_D <e_i', \sigma'>\} =$

$f(\lambda\sigma.\ \{<\sigma', [E_1(\gamma_D)(e_1^g),\ldots,E_1(\gamma_D)(e_i'),\ldots,E_1(\gamma_D)(e_n^g)]> \mid <e_i^g, \sigma> \rightarrow_D <e_i', \sigma'>\}) =$ (ind)

$f([E_1(\gamma_D)(e_1^g),\ldots,E_1(\gamma_D)(e_i^g),\ldots,E_1(\gamma_D)(e_n^g)]) =$

$E_1(\gamma_D)(f(e_1^g,\ldots,e_n^g))$

(*2) e' has the form $f(e_1^g,\ldots,e_i',\ldots,e_n^g)$

●2

$\Phi_b(D)(B_1(\gamma_D))(\text{and}(b_1^g, b_2^g)) =$

$\lambda\sigma.\ \{<\sigma', B_1(\gamma_D)(\text{and}(b_1', b_2^g))> \mid <\text{and}(b_1^g, b_2^g), \sigma> \rightarrow_D <\text{and}(b_1', b_2^g), \sigma'>\} =$ (*3)

$\lambda\sigma.\ \{<\sigma', \underline{\text{and}}([B_1(\gamma_D)(b_1'), B_1(\gamma_D)(b_2^g)]_2)> \mid <b_1^g, \sigma> \rightarrow_D <b_1', \sigma'>\} =$

$\underline{\text{and}}(\lambda\sigma.\ \{<\sigma', [B_1(\gamma_D)(b_1', B_1(\gamma_D)(b_2^g)]_2> \mid <b_1^g, \sigma> \rightarrow_D <b_1', \sigma'>\}) =$ (ind)

$\underline{\text{and}}([B_1(\gamma_D)(b_1^g), B_1(\gamma_D)(b_2^g)]_2) =$

$B_1(\gamma_D)(\text{and}(b_1^g, b_2^g))$

(*3) The other case and(b_1^g,b_2') is similar.

Analogous:

$\Phi_b(D)(B_1(\gamma_D))(\text{or}(b_1^g, b_2^g)) = B_1(\gamma_D)(\text{or}(b_1^g, b_2^g))$

$\Phi_b(D)(B_1(\gamma_D))(\text{not}(b^g)) = B_1(\gamma_D)(\text{not}(b^g))$

$\Phi_b(D)(B_1(\gamma_D))(\text{relop}(e_1^g, e_2^g)) =$

$\lambda\sigma.\ \{<\sigma', B_1(\gamma_D)(\text{relop}(e_1^g, e_2'))> \mid <\text{relop}(e_1^g, e_2^g), \sigma> \rightarrow_D <\text{relop}(e_1^g, e_2'), \sigma'>\} =$ (*4)

$\lambda\sigma.\ \{<\sigma', \underline{\text{relop}}([E_1(\gamma_D)(e_1^g), E_1(\gamma_D)(e_2')])> \mid <e_2^g, \sigma> \rightarrow_D <e_2', \sigma'>\} =$

$\underline{\text{relop}}(\lambda\sigma.\ \{<\sigma', [E_1(\gamma_D)(e_1^g), E_1(\gamma_D)(e_2')]> \mid <e_2^g, \sigma> \rightarrow_D <e_2', \sigma'>\}) =$ (ind)

$\underline{\text{relop}}([E_1(\gamma_D)(e_1^g), E_1(\gamma_D)(e_2^g)]) =$

$B_1(\gamma_D)(\text{relop}(e_1^g, e_2^g))$

(*4) The case $\text{relop}(e_1', e_2^g)$ is analogous.

●3

$\Phi(D)(D_1(\gamma_D))(s_1^g ; s_2) =$

$\lambda\sigma.\ \{<\sigma', D_1(\gamma_D)(s')> \mid <s_1^g ; s_2, \sigma> \rightarrow_D <s', \sigma'>\} =$ (*5)

$\lambda\sigma.\ \{<\sigma', D_1(\gamma_D)(s_2)> \mid <s_1^g ; s_2, \sigma> \rightarrow_D <s_2, \sigma'>\} =$

$D_1(\gamma_D)(s_2) \circ \lambda\sigma.\ \{<\sigma', p_0> \mid <s_1^g, \sigma> \rightarrow_D \sigma'\} =$ (ind)

$D_1(\gamma_D)(s_2) \circ D_1(\gamma_D)(s_1^g) =$

$D_1(\gamma_D)(s_1^g ; s_2)$

(*5) Case A is $s' \equiv s_1' ; s_2$; case B is $s' \equiv s_2$. We treat case B.

Analogous:

$\Phi(D)(D_1(\gamma_D))(s_1^g \cup s_2^g) = D_1(\gamma_D)(s_1^g \cup s_2^g)$

$\Phi(D)(D_1(\gamma_D))(\text{if B then } s_1^g \text{ else } s_2^g \text{ fi}) = D_1(\gamma_D)(\text{if B then } s_1^g \text{ else } s_2^g \text{ fi})$

$\Phi(D)(D_1(\gamma_D))(x := \epsilon^g) = \lambda\sigma.\ \{<\sigma', D_1(\gamma_D)(x := e')> \mid <x := \epsilon^g, \sigma> \rightarrow_D <x := e', \sigma'>\} =$

$\lambda\sigma.\ \{<\sigma', F(x, E_1(\gamma_D)(e'))> \mid <x := \epsilon^g, \sigma> \rightarrow_D <x := e', \sigma'>\} =$

$F(x, \lambda\sigma.\ \{<\sigma', E_1(\gamma_D)(e')> \mid <\epsilon^g, \sigma> \rightarrow_D <e', \sigma'>\}) =$ (*6)

$F(x, E_1(\gamma_D)(\epsilon^g)) = D_1(\gamma_D)(x := \epsilon^g)$

(*6) We already used: $\Phi_e(D)(E_1(\gamma_D))(e^g) = E_1(\gamma_D)(e^g)$.

$\Phi(D)(D_1(\gamma_D))(\text{if } \beta^g \text{ then } s_1 \text{ else } s_2 \text{ fi}) =$

$\lambda\sigma.\ \{<\sigma', D_1(\gamma_D)(s')> \mid <\text{if } \beta^g \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma> \rightarrow_D <s', \sigma'>\} =$ (*7)

$\lambda\sigma.\ \{<\sigma', [[B_1(\gamma_D)(b'), D_1(\gamma_D)(s_1), D_1(\gamma_D)(s_2)]]> \mid <\beta^g, \sigma> \rightarrow_D <b', \sigma'>\} =$ (*8)

$[[B_1(\gamma_D)(\beta^g), D_1(\gamma_D)(s_1), D_1(\gamma_D)(s_2)]] =$

$D_1(\gamma_D)(\text{if } \beta^g \text{ then } s_1 \text{ else } s_2 \text{ fi})$

(*7) $s' \equiv$ if b' then s_1 else s_2 fi

(*8) We already used: $\Phi_b(D)(B_1(\gamma_D))(b^g) = B_1(\gamma_D)(b^g)$.

Stage 2.

We prove by induction on the complexity. All cases are as in the first stage, except the case that $s \equiv \xi$ with $\xi \Leftarrow s_i^g$ in D.

$\Phi(D)(D_1(\gamma_D))(\xi) = \lambda\sigma.\ \{<\sigma', D_1(\gamma_D)(s')> \mid <\xi, \sigma> \rightarrow_D <s', \sigma'>\} =$

$\lambda\sigma.\ \{<\sigma', D_1(\gamma_D)(s')> \mid <s_i^g, \sigma> \rightarrow_D <s', \sigma'>\} =$ (stage 1)

$D_1(\gamma_D)(s_i^g) = D_1(\gamma_D)(\xi)$

This completes our semantic equivalence proof.

3.3. Semantics of the second interpretation

This paragraph is organized the same way as the previous paragraph. Section 3.3.1 is an introduction. An operational semantics is given in 3.3.2, a denotational semantics in 3.3.3. In section 3.3.4 a denotational semantics with continuations is presented. In 3.3.5 the semantical equivalence theorem is stated.

3.3.1. Introduction

The second interpretation is based on an enlarged granularity of atomic actions (in other words a smaller size). Now, atomic actions are: evaluation of an integer constant, a boolean constant, an integer variable; application of a function; the assignment of an integer value to an integer variable.

We illustrate the consequences in the following example.

Example

Consider the statement

$x :=$ plus($x:=1; y, y:=2; x$).

In the first interpretation execution of this statement consists of three atomic actions, namely $x:=1$, $y := 2$ and $x :=$ plus(y, x). The first action is either $x:=1$ or $y:=2$. The second action is now fixed by the first choice. The last action is $x :=$ plus(y, x).

According to the semantics D_1:

$D_1(\gamma_D)(x :=$ plus($x:=1; y, y:=2; x)) =$

$$\lambda\sigma.\ \{<\sigma\{1/x\}, \lambda\sigma.\ \{<\sigma\{2/y\}, \lambda\sigma.\ \{<\sigma\{3/x\}, p_0>\}>\}> ,$$
$$<\sigma\{2/y\}, \lambda\sigma.\ \{<\sigma\{1/x\}, \lambda\sigma.\ \{<\sigma\{3/x\}, p_0>\}>\}>\}$$

In the second interpretation execution of the above statement consists of eight atomic actions, namely evaluation of the integer constant 1 (eval(1)) , assignment of the integer value 1 to x , evaluation of the integer variable y (eval(y)) , eval(2) , assignment of the integer value 2 to y , eval(x) , application of the function plus and assignment of the resulting value to x. This results in 20 possible merges.

Suppose that σ is the state before execution. Then the possible final states (after execution) are:

1) $\sigma\{\ 2/y, 3/x\ \}$, (in the first interpretation the only final state)

2) $\sigma\{\ 2/y, \sigma(x)+2/x\ \}$,

3) $\sigma\{\ 2/y, \sigma(y)+1/x\ \}$.

In the next three sections the semantics are accomodated to support the enlarged granularity of atomic actions.

3.3.2. Operational semantics

We start with the definitions of operational statement, operational expression and operational boolean expression. They are needed in the configurations of the operational semantics.

Definition 3.34

The class of operational statements ($\tilde{s} \in$) OpStat is given by:

$\tilde{s} ::= s \mid x:=\tilde{e} \mid \tilde{s}_1;s_2 \mid$ if $\tilde{b}$ then s_1 else s_2 fi

where $s, s_1, s_2 \in$ Stat, $\tilde{e} \in$ OpExp and $\tilde{b} \in$ OpBexp

Definition 3.35

The class of operational expressions ($\tilde{e} \in$) OpExp is given by:

$\tilde{e} ::= e \mid \alpha \mid \tilde{s};e \mid f(\tilde{e}_1,\ldots,\tilde{e}_n)$

where $e \in$ Exp, $\alpha \in$ V and $\tilde{s} \in$ OpStat

Definition 3.36

The class of operational boolean expressions ($\tilde{b} \in$) OpBexp is given by:

$\tilde{b} ::= b \mid \underline{w} \mid \text{not}(\tilde{b}) \mid \text{and}(\tilde{b_1}, \tilde{b_2}) \mid \text{or}(\tilde{b_1}, \tilde{b_2}) \mid \text{relop}(\tilde{e_1}, \tilde{e_2})$

where $b \in$Bexp, $\underline{w} \in$ W and $\tilde{e}_1, \tilde{e}_2 \in$OpExp

Now we can define the configurations in

Definition 3.37

A configuration $c \in \text{Conf}_2$ is defined by:

$c ::= <\tilde{s}, \sigma> \mid \sigma \mid <\tilde{e}, \sigma> \mid \alpha \mid <\tilde{b}, \sigma> \mid \underline{w}$

where $\sigma \in \Sigma$, $\alpha \in V$, $\underline{w} \in W$; $\tilde{s} \in$ OpStat, $\tilde{e} \in$ OpExp and $\tilde{b} \in$ OpBexp

The transitions have the form:

$$<\tilde{s}, \sigma> \rightarrow_D <\tilde{s}', \sigma'> \mid \sigma'$$
$$<\tilde{e}, \sigma> \rightarrow_D <\tilde{e}', \sigma'> \mid \alpha$$
$$<\tilde{b}, \sigma> \rightarrow_D <\tilde{b}', \sigma'> \mid \underline{w}$$

We now give the transition system $T_2(D)$.

Definition 3.38

The transition system $T_2(D)$ is defined by the following axioms and rules:

Axiom 1

$$< x := \alpha, \sigma> \rightarrow_D \sigma\{\alpha/x\}$$

Axiom 2

$$< m, \sigma> \rightarrow_D \alpha$$

provided that α is the integer denoted by m

Axiom 3

$$<\text{true}, \sigma> \rightarrow_D \underline{\text{true}}$$

Axiom 4

$$<\text{false}, \sigma> \rightarrow_D \underline{\text{false}}$$

Axiom 5

$$< x, \sigma> \rightarrow_D \sigma(x)$$

Axiom 6

$$< f(\alpha_1, \ldots, \alpha_n), \sigma> \rightarrow_D \alpha$$

provided that $\alpha = f(\alpha_1, \ldots, \alpha_n)$

Axiom 7

$$<\text{relop}(\alpha_1, \alpha_2), \sigma> \rightarrow_D \underline{w}$$

provided that $\underline{w} =$ [relop] (α_1, α_2)

Axiom 8

$$<\text{not}(\underline{w}_1), \sigma> \rightarrow_D \underline{w}$$

provided that $\underline{w} = [\text{not}]\ (\underline{w}_1)$

Axiom 9

$$< \text{and}(\underline{w}_1, \underline{w}_2), \sigma > \rightarrow_D \underline{w}$$

provided that $\underline{w} = [\text{and}]\ (\underline{w}_1, \underline{w}_2)$

Axiom 10

$$< \text{or}(\underline{w}_1, \underline{w}_2), \sigma > \rightarrow_D \underline{w}$$

provided that $\underline{w} = [\text{or}]\ (\underline{w}_1, \underline{w}_2)$

Rule 1

$$\frac{< \tilde{s}, \sigma > \rightarrow_D < \tilde{s}', \sigma' > \mid \sigma'}{\begin{array}{l} < \tilde{s}\ ; s_2, \sigma > \rightarrow_D < \tilde{s}'\ ; s_2, \sigma' > \mid < s_2, \sigma' > \\ < (\tilde{s}\ ; e), \sigma > \rightarrow_D < (\tilde{s}'\ ; e), \sigma' > \mid < e, \sigma' > \end{array}}$$

Rule 2

$$\frac{< s, \sigma > \rightarrow_D < \tilde{s}', \sigma' >}{\begin{array}{l} < \text{if } \underline{\text{true}} \text{ then } s \text{ else } s_2 \text{ fi}, \sigma > \rightarrow_D < \tilde{s}', \sigma' > \\ < \text{if } \underline{\text{false}} \text{ then } s_1 \text{ else } s \text{ fi}, \sigma > \rightarrow_D < \tilde{s}', \sigma' > \\ < s \cup s_2, \sigma > \rightarrow_D < \tilde{s}', \sigma' > \\ < s_1 \cup s, \sigma > \rightarrow_D < \tilde{s}', \sigma' > \\ < \xi, \sigma > \rightarrow_D < \tilde{s}', \sigma' > \end{array}}$$

provided that in the last conclusion $\xi \Leftarrow s$ in D

Rule 3

$$\frac{< \tilde{e}, \sigma > \rightarrow_D < \tilde{e}', \sigma' > \mid \alpha}{\begin{array}{l} < x := \tilde{e}, \sigma > \rightarrow_D < x := \tilde{e}', \sigma' > \mid < x := \alpha, \sigma > \\ < f(\tilde{e}_1, \ldots, \tilde{e}_{i-1}, \tilde{e}, \tilde{e}_{i+1}, \ldots, \tilde{e}_n), \sigma > \rightarrow_D < f(\tilde{e}_1, \ldots, \tilde{e}_{i-1}, \tilde{e}', \tilde{e}_{i+1}, \ldots, \tilde{e}_n), \sigma' > \mid < f(\tilde{e}_1, \ldots, \tilde{e}_{i-1}, \alpha, \tilde{e}_{i+1}, \ldots, \tilde{e}_n), \sigma > \\ < \text{relop}(\tilde{e}, \tilde{e}_2), \sigma > \rightarrow_D < \text{relop}(\tilde{e}', \tilde{e}_2), \sigma' > \mid < \text{relop}(\alpha, \tilde{e}_2), \sigma > \\ < \text{relop}(\tilde{e}_1, \tilde{e}), \sigma > \rightarrow_D < \text{relop}(\tilde{e}_1, \tilde{e}'), \sigma' > \mid < \text{relop}(\tilde{e}_1, \alpha), \sigma > \end{array}}$$

Rule 4

$$\frac{< \tilde{b}, \sigma > \rightarrow_D < \tilde{b}', \sigma' > \mid \underline{w}}{\begin{array}{l} < \text{not}(\tilde{b}), \sigma > \rightarrow_D < \text{not}(\tilde{b}'), \sigma' > \mid < \text{not}(\underline{w}), \sigma > \\ < \text{and}(\tilde{b}, \tilde{b}_2), \sigma > \rightarrow_D < \text{and}(\tilde{b}', \tilde{b}_2), \sigma' > \mid < \text{and}(\underline{w}, \tilde{b}_2), \sigma > \\ < \text{and}(\tilde{b}_1, \tilde{b}), \sigma > \rightarrow_D < \text{and}(\tilde{b}_1, \tilde{b}'), \sigma' > \mid < \text{and}(\tilde{b}_1, \underline{w}), \sigma > \\ < \text{or}(\tilde{b}, \tilde{b}_2), \sigma > \rightarrow_D < \text{or}(\tilde{b}', \tilde{b}_2), \sigma' > \mid < \text{or}(\underline{w}, \tilde{b}_2), \sigma > \\ < \text{or}(\tilde{b}_1, \tilde{b}), \sigma > \rightarrow_D < \text{or}(\tilde{b}_1, \tilde{b}'), \sigma' > \mid < \text{or}(\tilde{b}_1, \underline{w}), \sigma > \\ < \text{if } \tilde{b} \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma > \rightarrow_D < \text{if } \tilde{b}' \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma' > \mid < \text{if } \underline{w} \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma > \end{array}}$$

The ten axioms represent all atomic actions. The rules are very similar to the rules in transition system $T_1(D)$. Note that for every configuration $<\tilde{s}, \sigma>$ there is a transition $<\tilde{s}, \sigma> \rightarrow_D <\tilde{s}', \sigma'>$. A similar remark holds for configurations $<\tilde{e}, \sigma>$ with $\tilde{e} \notin V$ and $<\tilde{b}, \sigma>$ with $\tilde{b} \notin W$.

We define the operational meaning of a program in definition 3.39. In the sequel we use the following process domains.

$$(p\in)\ P = \{p_0\} \cup (\Sigma \rightarrow \mathcal{P}_{nc}(\Sigma \times id_{1/2}(P)))$$
$$(q\in)\ \overline{Q} = V \cup (\Sigma \rightarrow \mathcal{P}_{nc}(\Sigma \times id_{1/2}(\overline{Q})))$$
$$(r\in)\ \overline{R} = W \cup (\Sigma \rightarrow \mathcal{P}_{nc}(\Sigma \times id_{1/2}(\overline{R})))$$

Definition 3.39

a. The mapping O_2 : Prog $\rightarrow$ P is given by : $O_2 [< D \mid s >] = O_D' [s]$

b. The mapping O_D' : OpStat $\rightarrow$ P is given by :

$$O_D' [s] = \lambda\sigma.(\{< \sigma', O_D' [s'] > \mid < s, \sigma> \rightarrow_D < s', \sigma'>\} \cup \{< \sigma', p_0> \mid < s, \sigma> \rightarrow_D \sigma' \})$$

Lemma 3.40

Let the operator Φ' : Decl $\rightarrow$ ((OpStat $\rightarrow$ P) $\rightarrow$ (OpStat $\rightarrow$ P)) be defined as follows :

For N : OpStat $\rightarrow$ P and D$\in$ Decl:

$$\Phi'(D)(N)(s) = \lambda\sigma.(\{< \sigma', N(s')> \mid < s, \sigma> \rightarrow_D < s', \sigma'>\} \cup \{< \sigma', p_0> \mid < s, \sigma> \rightarrow_D \sigma'\})$$

Then $\Phi'(D)$ is contracting and $\Phi'(D)$ has O'_D as its fixed point.

We now define the operational meaning of an expression and a boolean expression.

Definition 3.41

The mapping F_D' : OpExp $\rightarrow \overline{Q}$ is given by :

For $\tilde{e} \notin V$:

$$F_D'(\tilde{e}) = \lambda\sigma.\{< \sigma', F_D'(\tilde{e}')> \mid < \tilde{e}, \sigma> \rightarrow_D < \tilde{e}', \sigma'>\}$$

for $\alpha \in V$

$$F_D'(\alpha) = \alpha$$

Lemma 3.42

Let Φ_e' : Decl $\rightarrow$ ((OpExp $\rightarrow \overline{Q}$) $\rightarrow$ (OpExp $\rightarrow \overline{Q}$)) be given by :

For G : OpExp $\rightarrow \overline{Q}$ and D$\in$ Decl

For $\tilde{e} \notin V$

$$\Phi_e'(D)(G)(\tilde{e}) = \lambda\sigma.\{< \sigma', G(\tilde{e}')> \mid < \tilde{e}, \sigma> \rightarrow_D < \tilde{e}', \sigma'>\}$$

for $\alpha \in V$

$$\Phi_e'(D)(G)(\alpha) = \alpha$$

Then $\Phi_e'(D)$ is contracting and $\Phi_e'(D)$ has F_D' as its fixed point.

Definition 3.43

The mapping A_D' : OpBexp $\rightarrow \overline{R}$ is given by :

For $\tilde{b} \notin W$

$$A_D'(\tilde{b}) = \lambda\sigma.\{< \sigma', A_D'(\tilde{b}')> \mid < \tilde{b}, \sigma> \rightarrow_D < \tilde{b}', \sigma'>\}$$

for $\underline{w} \in W$

$$A_D'(\underline{w}) = \underline{w}$$

Lemma 3.44

Let Φ_b' : Decl $\rightarrow$ ((OpBexp $\rightarrow \overline{R}$) $\rightarrow$ (OpBexp $\rightarrow \overline{R}$)) be given by :

For C : Bexp $\rightarrow \overline{R}$ and $D \in$ Decl

For $\tilde{b} \notin W$

$$\Phi_b'(D)(C)(\tilde{b}) = \lambda\sigma.\{< \sigma', C(\tilde{b}')> \mid < \tilde{b}, \sigma> \rightarrow_D < \tilde{b}', \sigma'>\}$$

for $\underline{w} \in W$

$$\Phi_b'(D)(C)(\underline{w}) = \underline{w}$$

Then $\Phi_b'(D)$ is contracting and $\Phi_b'(D)$ has A_D' as its fixed point.

3.3.3. Denotational semantics

The denotational semantics to be presented is very similar to the one given in section 3.2.3. Just some minor modifications are needed to support the enlarged granularity of atomic actions of the second interpretation.

First we recall

$$Env = Pvar \rightarrow P$$

$$P = \{p_0\} \cup (\Sigma \rightarrow \mathcal{P}_{nc}(\Sigma \times id_{½}(P)))$$

$$\overline{Q} = V \cup (\Sigma \rightarrow \mathcal{P}_{nc}(\Sigma \times id_{½}(\overline{Q})))$$

$$\overline{R} = W \cup (\Sigma \rightarrow \mathcal{P}_{nc}(\Sigma \times id_{½}(\overline{R})))$$

We now define the meaning of a program.

Definition 3.45

The mapping M_2 : Prog $\rightarrow$ P is given by:

$$M_2[< D \mid s >] = D_2(\gamma_D)(s) ,$$

where $\gamma_D = \gamma\{p_i/\xi_i\}_{i=1}^n$

where , for $D \equiv < \xi_i \Leftarrow s_i^g >_i$, we put $<p_1,\ldots,p_n> =$ fixed point $<\Phi_1,\ldots,\Phi_n>$

where Φ_j : $P^n \rightarrow P$ is given by:

$$\Phi_j(p_1') \cdots (p_n') = D_2(\gamma\{p_i'/\xi_i\}_i)(s_j^g)$$

Lemma 3.46

The unique fixed point (in Definition 3.45) exists by the guardedness requirement which ensures contractivity of Φ_j $(1 \leqq j \leqq n)$.

We now give the definition of the denotational semantics D_2, followed by the definitions of the new operators. The first two lines of the definition of E_2 in 3.47b are essentially different from the definition of E_1. Evaluation now takes a (silent) step.

Definition 3.47

a. The mapping D_2 : Env $\rightarrow$ (Stat $\rightarrow$ P) is given by:

$$D_2(\gamma)(s_1;s_2) = D_2(\gamma)(s_2) \circ_1 D_2(\gamma)(s_1)$$
$$D_2(\gamma)(s_1 \cup s_2) = D_2(\gamma)(s_1) \cup D_2(\gamma)(s_2)$$
$$D_2(\gamma)(\xi) = \gamma(\xi)$$
$$D_2(\gamma)(x:=e) = \mathrm{F}'(x, E_2(\gamma)(e))$$
$$D_2(\gamma)(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) = [[B_2(\gamma)(b), D_2(\gamma)(s_1), D_2(\gamma)(s_2)]]_2$$

b. The mapping E_2 : Env $\rightarrow$ (Exp $\rightarrow \overline{\mathrm{Q}}$) is defined by:

$$E_2(\gamma)(x) = \lambda\sigma.\ \{<\sigma, \sigma(x)>\}$$
$$E_2(\gamma)(m) = \lambda\sigma.\ \{<\sigma, V(m)(\sigma)>\}$$
$$E_2(\gamma)((s;e)) = E_2(\gamma)(e) \circ_3 D_2(\gamma)(s)$$
$$E_2(\gamma)(f(e_1,\ldots,e_n)) = \underline{f}([E_2(\gamma)(e_1),\ldots,E_2(\gamma)(e_n)]_3)$$

c. The mapping B_2 : Env $\rightarrow$ (Bexp $\rightarrow \overline{\mathrm{R}}$) is defined by :

$$B_2(\gamma)(\text{true}) = \underline{\text{true}}$$
$$B_2(\gamma)(\text{false}) = \underline{\text{false}}$$
$$B_2(\gamma)(\text{not}(b)) = \underline{\text{not}}(B_2(\gamma)(b))$$
$$B_2(\gamma)(\text{relop}(e_1,e_2)) = \underline{\text{relop}}([E_2(\gamma)(e_1),E_2(\gamma)(e_2)]_3)$$
$$B_2(\gamma)(\text{and}(b_1,b_2)) = \underline{\text{and}}([B_2(\gamma)(b_1),B_2(\gamma)(b_2)]_4)$$
$$B_2(\gamma)(\text{or}(b_1,b_2)) = \underline{\text{or}}([B_2(\gamma)(b_1),B_2(\gamma)(b_2)]_4)$$

The composition operator $\circ_1$ and the union operator $\cup$ have been defined in definition 3.24.

Definition 3.48

a. The assignment-operator F': Ivar $\times \overline{\mathrm{Q}} \rightarrow$ P is defined by:

For $q \notin V$

$$\mathrm{F}'(x, q) = \lambda\sigma. \bigcup_{<\sigma',q'> \in q(\sigma)} \{<\sigma', \mathrm{F}'(x,q')>\}$$

for $\alpha \in V$

$$\mathrm{F}'(x, \alpha) = \lambda\sigma.\{<\sigma\{\alpha/x\}, p_0>\}$$

b. The composition operator $\circ_3 : \overline{\mathrm{Q}} \times \mathrm{P} \rightarrow \overline{\mathrm{Q}}$ is given by:

$$q \circ_3 p = \lambda\sigma. \bigcup_{<\sigma',p'> \in p(\sigma)} \{<\sigma', q \circ_3 p'>\}$$

$$q \circ_3 p_0 = q$$

We now define the merge-operators $[\ \cdots\ ]_3$ and $[\ \cdots\ ]_4$. Therefore we need two intermediary domains:

$$(\phi \in)\ \overline{Q}_n' = V^n \cup (\Sigma \to P_{nc}(\Sigma \times id_{\frac{1}{2}}(\overline{Q}_n')))$$
$$\overline{R}' = W \times W \cup (\Sigma \to P_{nc}(\Sigma \times id_{\frac{1}{2}}(\overline{R}')))$$

Definition 3.49

a. The merge operator $[\ \cdots\ ]_3 : \overline{Q}^n \to \overline{Q}_n'$ is given by:

$$[q_1,\ldots,q_n]_3 = \lambda\sigma.\{<\sigma', [q_1,\ldots,q_i',\ldots,q_n]> \mid 1 \leqq i \leqq n,\ q_i \notin V,\ <\sigma',q_i'> \in q_i(\sigma)\}$$

where $\exists\ j$ such that $q_j \notin V$.

$$[\alpha_1,\ldots,\alpha_n]_3 = (\alpha_1,\ldots,\alpha_n)$$

b. The operator $[\ \cdots\ ,\ \cdots\ ]_4 : \overline{R} \times \overline{R} \to \overline{R}'$ is given by:

$$[r_1,r_2]_4 = \lambda\sigma.(\{<\sigma', [r_1',r_2]_4> \mid r_1 \notin W,\ <\sigma',r_1'> \in r_1(\sigma)\}$$
$$\cup \{<\sigma', [r_1,r_2']_2> \mid r_2 \notin W,\ <\sigma',r_2'> \in r_2(\sigma)\})$$

where $\exists\ j$ such that $r_j \notin W$.

$$[\underline{w}_1,\underline{w}_2]_4 = (\underline{w}_1, \underline{w}_2)$$

Note that in the second lines of the above definitions of the merge operators there is no longer a dependency on the state σ. We now turn to the meaning of the underlined functions. We restrict ourselves to the underlined f. The case of the underlined boolean functions is handled very similarly. We recall that the eventual application of a function to a tuple of values takes a (silent) step.

Definition 3.50

$\underline{f} : \overline{Q}_n' \to \overline{Q}$ is defined as follows:

For $\phi \in \overline{Q}_n' \setminus V^n$

$$\underline{f}(\phi) = \lambda\sigma.\ \cup_{<\sigma',\phi'> \in \phi(\sigma)} \{<\sigma', \underline{f}(\phi')>\}$$

for $\phi \in V^n$

$$\underline{f}(\phi) = \lambda\sigma.\{<\sigma, [f](\phi)>\}$$

The semantic if-then-else operator is defined in

Definition 3.51

The operator $[[\ldots, \ldots, \ldots]]_2 : \overline{R} \times P \times P \to P$ is defined by:

For $r \notin W$

$$- [[r,p_1,p_2]]_2 = \lambda\sigma.\ \cup_{<\sigma',\overline{r}> \in r(\sigma)} \{<\sigma', [[\overline{r},p_1,p_2]]>\}$$

for $\underline{w} \in W$

- $[[\underline{w},p_1,p_2]]_2 = \lambda\sigma.$ if $\underline{w}$ then $p_1(\sigma)$ else $p_2(\sigma)$ fi,

where $p_1 \neq p_0$ and $p_2 \neq p_0$.

- $[[\underline{w},p_0,p]]_2 = \lambda\sigma.$ if $\underline{w}$ then $\{<\sigma,p_0>\}$ else $p(\sigma)$ fi,

where $p \neq p_0$.

- $[[\underline{w},p,p_0]]_2 = \lambda\sigma.$ if $\underline{w}$ then $p(\sigma)$ else $\{<\sigma,p_0>\}$ fi,

where $p \neq p_0$.

- $[[\underline{w},p_0,p_0]]_2 = \lambda\sigma. \{<\sigma,p_0>\}$

3.3.4. Denotational semantics with continuations

As in section 3.2.4, we now present a denotational semantics with continuations. We state without proof that this semantics M_2' is equivalent to M_2.

We define

Definition 3.52

The mapping M_2' : Prog $\rightarrow$ P is given by:

$M_2'[< D \mid s >] = D_2'(\gamma_D)(s)(p_0)$,

where $\gamma_D = \gamma\{p_i/\xi_i\}_{i=1}^n$

where , for $D \equiv < \xi_i \Leftarrow s_i^g >_i$, we put $<p_1,\ldots,p_n>$ = fixed point $<\Phi_1,\ldots,\Phi_n>$

where Φ_j : $P^n \rightarrow P$ is given by:

$\Phi_j(p_1') \cdots (p_n') = D_2'(\gamma\{p_i'/\xi_i\}_i)(s_j^g)(p_0)$

Lemma 3.53

The unique fixed point in the above definition exists by the guardedness requirement which ensures contractivity of the Φ_j.

The class of continuations ($p \in$) $Cont_s$ is equal to P. The class of expression continuations is given by

($c_e \in$) $Cont_e = (V \rightarrow P) \cup \{i_e\}$

The class of boolean expression continuations is given by

($c_b \in$) $Cont_b = (W \rightarrow P) \cup \{i_b\}$

i_e and i_b denote the empty expression continuation respectively the empty boolean expression continuation. i_e is given by

$i_e : V \rightarrow V$ such that $i_e(\alpha) = \alpha$

Analogously, i_b is given by

$i_b : W \rightarrow W$ such that $i_b(\underline{w}) = \underline{w}$

We now define

Definition 3.54

a. The mapping D_2' : Env → Stat → Cont_s → P is given by:

$$D_2'(\gamma)(s_1;s_2)(\mathrm{p}) = D_2'(\gamma)(s_1)(\,D_2'(\gamma)(s_2)(\mathrm{p})\,)$$

$$D_2'(\gamma)(s_1 \cup s_2)(\mathrm{p}) = D_2'(\gamma)(s_1)(\mathrm{p}) \cup D_2'(\gamma)(s_2)(\mathrm{p})$$

$$D_2'(\gamma)(\xi)(\mathrm{p}) = C(\gamma(\xi))(\mathrm{p})$$

$$D_2'(\gamma)(x:=e)(\mathrm{p}) = E_2'(\gamma)(e)(\lambda\alpha.\lambda\sigma.\{<\sigma\{\alpha/x\}, \mathrm{p}>\})$$

$$D_2'(\gamma)(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi})(\mathrm{p}) = B_2'(\gamma)(b)(\lambda\underline{\mathrm{w}}.\ \text{if } \underline{\mathrm{w}} \text{ then } D_2'(\gamma)(s_1)(\mathrm{p}) \text{ else } D_2'(\gamma)(s_2)(\mathrm{p}))$$

b. The mapping E_2' : Env → Exp → (((V → P) → P) ∪ $\{\mathrm{i}_e\}$ → $\overline{\mathrm{Q}}$) is defined by:

$$E_1'(\gamma)(x)(\mathrm{c}_e) = \lambda\sigma.\{<\sigma, \mathrm{c}_e(\sigma(x))>\}$$

$$E_1'(\gamma)(m)(\mathrm{c}_e) = \lambda\sigma.\{<\sigma, \mathrm{c}_e(V(m)(\sigma))>\}$$

$$E_2'(\gamma)((s;e))(\mathrm{c}_e) = D_2'(\gamma)(s)(E_2'(\gamma)(e)(\mathrm{c}_e))$$

$$E_2'(\gamma)(f(e_1,\ldots,e_n))(\mathrm{c}_e) = \underline{f}([E_2'(\gamma)(e_1)(\mathrm{i}_e),\ldots,E_2'(\gamma)(e_n)(\mathrm{i}_e)]_3)(\mathrm{c}_e)$$

c. The mapping B_2' : Env → Bexp → (((W → P) → P) ∪ $\{\mathrm{i}_b\}$ → $\overline{\mathrm{R}}$) is defined by:

$$B_2'(\gamma)(\text{true})(\mathrm{c}_b) = \mathrm{c}_b(\underline{\text{true}})$$

$$B_2'(\gamma)(\text{false})(\mathrm{c}_b) = \mathrm{c}_b(\underline{\text{false}})$$

$$B_2'(\gamma)(\text{not}(b))(\mathrm{c}_b) = \underline{\text{not}}(B_2'(\gamma)(b)(\mathrm{i}_b))(\mathrm{c}_b)$$

$$B_2'(\gamma)(\text{relop}(e_1,e_2))(\mathrm{c}_b) = \underline{\text{relop}}([E_2'(\gamma)(e_1)(\mathrm{i}_e), E_2'(\gamma)(e_2)(\mathrm{i}_e)]_3)(\mathrm{c}_b)$$

$$B_2'(\gamma)(\text{and}(b_1,b_2))(\mathrm{c}_b) = \underline{\text{and}}([B_2'(\gamma)(b_1)(\mathrm{i}_b), B_2'(\gamma)(b_2)(\mathrm{i}_b)]_4)(\mathrm{c}_b)$$

$$B_2'(\gamma)(\text{or}(b_1,b_2))(\mathrm{c}_b) = \underline{\text{or}}([B_2'(\gamma)(b_1)(\mathrm{i}_b), B_2'(\gamma)(b_2)(\mathrm{i}_b)]_4)(\mathrm{c}_b)$$

The definition of the union operator ∪ : P × P → P has been given in definition 3.24b. The definition of the merge operators has been given in definition 3.49. The definition of the if-then-else operator : W × P × P → P is assumed to be known. The operator C has been defined in definition 3.31.

The underline operator is now given by the following definition. Again, we only consider the case of the function f.

Definition 3.55

$\underline{f}$: $\overline{\mathrm{Q}}_n'$ → (((V → P) → P) ∪ $\{\mathrm{i}_e\}$ → $\overline{\mathrm{Q}}$) is defined by:

if $\phi \in \overline{\mathrm{Q}}_n' \setminus \mathrm{V}^n$

$$\underline{f}(\phi)(\mathrm{c}_e) = \lambda\sigma.\ \cup_{<\sigma',\phi'>\in\phi(\sigma)} \{<\sigma', \underline{f}(\phi')(\mathrm{c}_e)>\}$$

if $\phi \in \mathrm{V}^n$

$$\underline{f}(\phi)(\mathrm{c}_e) = \lambda\sigma.\{<\sigma, \mathrm{c}_e([f](\phi))>\}$$

3.3.5. Semantic equivalence

In this section we state the semantic equivalence theorem; the proof of this theorem is not given. It is very similar to the proof given in section 3.2.5.

Theorem 3.56

For all t ∈ Prog : $O_2[\,t\,] = M_2[\,t\,]$.

We conclude this chapter with a short comment in the next paragraph.

3.4. Comment

In the first interpretation the applicative concurrency of the language is rather close to imperative concurrency. Why? Because the only atomic action is the simple assignment, evaluation of the arguments of a function amounts to the concurrent evaluation of the possible side-effects followed by the immediate evaluation of the resulting simple expressions. In fact, the statements in the arguments are collected and become arguments of some imperative concurrency operator.

In the second interpretation we can really speak about applicative concurrency. The arguments are evaluated in parallel. Even the evaluation of an integer constant is now an atomic action. The operational semantics given in this chapter have clearly shown the above. In the second interpretation especially the denotational semantics has been obtained by some little adjustments of the corresponding denotational semantics of the first interpretation.

The available metric framework has provided a neat environment for doing the semantical work. The semantical equivalence proof has been rather simple. Processes have been used in a 'natural' way to give meaning to the language constructs.

4. A functional language with side-effects

In this chapter we describe a language adopted from [Jo]. In [Jo] a style of programming called functional programming with side-effects has been introduced. In paragraph 4.1 some aspects of this functional programming with side-effects come up for discussion. In paragraph 4.2 a description of the particular language that we consider is given. It is followed in paragraph 4.3 by a detailed discussion of its denotational semantics as presented in [Jo].

The second part of this chapter is an addition to [Jo]. In paragraph 4.4 an operational semantics is provided for the language. Moreover, attention is paid to parallel evaluation of expressions and its semantics.

Furthermore, in paragraph 4.5 we go into a semantical specification formalism called natural semantics as described in [Ka] and then give a natural semantics for the described language. Paragraph 4.6 concludes this chapter with a short comment.

4.1 Functional programming with side-effects

Functional programming with side-effects is an extension of functional programming. Variables are made available for use together with the new feature of lazy single assignment. The notion of assignment (and side-effects) in a functional context seems somewhat controversial. However, Josephs provides evidence in [Jo] that

(i) Although functional programs are relatively easy to write because of the absence of side-effects, the possible presence of side-effects does not really complicate the development of functional programs with side-effects.

(ii) Despite the fact that subexpressions of a functional program with side-effects are not necessarily independent of each other, as is the case for subexpressions of a functional program, the programs are suitable for parallel execution. [More about this in paragraph 4.2 and 4.4.]

(iii) The lazy single assignment enriches functional programming with side-effects in an essential way. Some algorithms expressed in functional programs with side-effects are said to be unavailable to a purely functional programmer.

(iv) A gain in efficiency can be achieved in space and time complexity, whereby demand-driven graph reduction is used to implement functional programming with side-effects.

We already mentioned the notion of laziness. Functional programming with side-effects is based on

lazy evaluation. The lazy evaluation strategy is characterized by
(a) In case of the lazy single assignment

assign $x = E$; E'

the expression E is bound in unevaluated form to the uninstantiated variable x and is evaluated just in case of a demand for the value of x.
(b) The arguments to a function call are evaluated at most once and only if their values are required. For example, the evaluation of if true E_2 E_3 does not require the evaluation of E_3. The evaluation of $((\lambda x.(\text{add } x\ x))E)$ requires the evaluation of E only once.
(c) In case of a datastructure as (cons E_1 E_2) evaluation of E_1 and E_2 is delayed until their values are actually needed.

Finally, we like to point out that it is possible that evaluation of an expression deadlocks. An expression may contain uninstantiated variables. A demand for the value of such an uninstantiated variable suspends until the variable gets instantiated. A state of deadlock now arises when all demands for the values of expressions have been suspended. In fact, in case of sequential evaluation only one suspension already leads to deadlock.

4.2. Description of the language

In this paragraph we describe the particular functional language with side-effects that we consider. In section 4.2.1 the syntax of the language is given as described in chapter 4 of [Jo]. It is followed by the informal semantics of the language in section 4.2.2 and some examples in section 4.2.3.

4.2.1 Syntax

We start with a preliminary definition

Definition 4.1

($x \in$) Ide is the syntactical set of identifiers.

($m \in$) Icon is the syntactical set of integer constants.

($b \in$) Bcon is the syntactical set of boolean constants { true, false }.

Integer constants and boolean constants form a part of the language primitives or primitive expressions. Primitive expressions are defined in

Definition 4.2

The class of primitive expressions (p$\in$) Prim is defined by

p::= b | m | add | div | not | if | nil | cons | head | seq | val | Y

where $m \in$ Icon and $b \in$ Bcon.

We now give the syntax of the language.

Definition 4.3

The class of expressions ($E \in$) Exp is given by:

$$E ::= p \mid x \mid \text{var } x \text{ ; } E \mid \text{assign } x = E_1; E_2 \mid E_1\, E_2 \mid \lambda x.E$$

where $p \in$ Prim and $x \in$ Ide.

The language thus consists of expressions. The intuitive meaning of these expressions is explained in the next section.

4.2.2 Informal explanation

The intuitive meaning of the syntactical objects b and m is clear. The primitives add, div and not stand for the wellknown functions of the same name, defined on integer values respectively boolean truth values. We will call these primitive functions standard (primitive) functions. In the denotational semantics only the case of sequential evaluation of the arguments of standard functions is treated. In the operational semantics the case of concurrent evaluation is also considered. Furthermore, it is possible to extend the class of standard functions with functions as mul, sub, or, and, equal, etc. without radical consequences for the semantics. The primitive if denotes the function of three arguments with the usual meaning. Note that evaluation of its condition has possible side-effects, as was also the case in the previous chapter.

Nil denotes the empty list. The primitive function cons constructs a list of two elements or a pair. The arguments, normally in unevaluated form, are its respective elements. We stress that evaluation of the arguments is not forced by cons. Head is a primitive function of one argument, which value must be a list or pair. Head returns the value of the first element of the list. Moreover, the language can be extended with a similar primitive function tail.

The primitive seq represents a function of two arguments. Seq $E_1\ E_2$ stands for the sequential composition of the expressions E_1 and E_2. After evaluation of E_1 its value is discarded. Seq eventually returns the value of E_2. The primitive function val has two arguments. Val supports call-by-value. We here deviate from [Jo] and turn around the order of its two arguments. Now the first argument is evaluated first. The value of the second argument must be a function. This function is applied to the value of the first argument and val returns the resulting value. The primitive Y stands for Curry's paradoxical combinator. It is used to define recursive functions. We remind the reader of its definition:

$$Y \equiv \lambda y.\ (\lambda x.y(xx))\ (\lambda x.y(xx))$$

Besides parallel evaluation of the arguments of standard functions, the language can also be adapted to support parallel evaluation in a larger sense by means of the introduction of a primitive par. Par has two arguments E_1 and E_2. Evaluation of par $E_1\ E_2$ involves the concurrent evaluation of E_2 and E_1E_2. The value of E_1E_2 is the value that par returns. In this manner the language provides control annotations for both sequential and parallel evaluation. For instance, add $E_1\ E_2$ now stands for sequential addition and par (par add E_1) E_2 stands for parallel addition. In [Jo] no denotational semantics is given for the par combinator. In paragraph 4.4 an operational semantics is provided to support its use.

So far we have only occupied ourselves with primitive expressions and their intuitive meaning. We now continue with other expressions and their informal semantics.

Evaluation of an identifier x involves a demand for the value of x. If x is uninstantiated or already under evaluation, this demand is suspended. If x has been bound to an expression, this expression is evaluated and its value becomes the value of x. If x had already been evaluated, its value is directly returned. During the evaluation of an expression E that had been bound to x, x is marked 'under evaluation' so that the expression E just once is evaluated. In the meantime other demands for the value of x are suspended. This mechanism also detects some silly recursions. For instance, suppose x has been bound to the expression (add x 1). At some time during evaluation a demand is sent for the value of x. This demand is suspended, because x has been marked under evaluation. This suspension is fatal: it will not resume again.

Before an identifier can be used, it must be declared in a var or lambda construct. Evaluation of the expression var x; E involves the declaration of x as an uninstantiated variable with scope E. After that E is evaluated. The value of E is the value of the var expression.

The lazy single assignment assign $x = E_1$; E_2 has the following intuitive meaning: if x is an uninstantiated variable, x is bound to the expression E_1 in unevaluated form. Note that E_1 is an arbitrary expression. After that E_2 is evaluated and its value is the value of assign $x = E_1$; E_2. This lazy single assignment

provides the basis for side-effects in the language. If x has been instantiated already, evaluation of a lazy single assignment to x results in an error.

The expression E_1E_2 stands for the call-by-need application of E_1 to E_2. First E_1 is evaluated and its value must be a function. After that this function is applied to E_2. E_2 is evaluated as soon as its value is really required. Note however that if E_2 is an identifier, its value may have been evaluated already.

We now discuss briefly the notion of function. What stands in fact for a function? At first we have seen that some primitives denote a function. Secondly, a lambda expression stands for a function. Furthermore, a function applied to a number of arguments less than its arity is a (partially applied) function. An example of the latter category is the semantical meaning of (add 2). Note that the arguments of a partially applied function normally have not been evaluated yet.

A final remark concerns the application of a lambda expression to its argument. If the expression E' is not an identifier, evaluation of $(\lambda x.E\ E')$ has the same result as evaluation of assign $x = E'$; E. First x is bound to expression E' and evaluation then proceeds with E. However the evaluation of $(\lambda x.E\ y)$ differs from the evaluation of assign $x = y$; E. In the former case x is not bound to expression y, but x is 'identified' with y. That is, a demand for the value of x is actually a demand for the value of y. In case of an assignment to x as for instance in $(\lambda x.$ (assign $x = E_1$; E_2) $y)$, E_1 is in fact assigned to y, provided that y is uninstantiated.

In the next section we give some examples of the language constructs discussed. Among other things we show how the lack of a direct definition of functions by pattern-matching can be overcome.

4.2.3 Examples

Example 1

```
var x; ((seq ( assign x =2; var x; assign x =3; ((add x) 4) ) )
                ( (( λx.λy.((div x) y) 12) x) ) )
```

This example contains in fact three declarations of x. The scope of the first declared x includes the last occurrence of x. This x is bound to the expression 2. The value of the expression eventually is 6.

Example 2

```
var x; var y; ((add ( assign x =3; y )) ( assign y =2; x ))
```

Evaluation of this expression is suspended in case of sequential evaluation of the arguments of the standard primitive function add. Only in case of parallel evaluation of the two operands the value of the expression is equal to 5.

Example 3

Suppose that the expression E does not contain free occurrences of the identifier z and the expression E' does not contain free occurrences of y. The identifiers y and z are used as auxiliary variables in

(a) (Y $\lambda y.\lambda x.E$) E'

(b) var y; assign $y = \lambda x.E$; (y E')

(c) (($\lambda z.$($\lambda y.\lambda x.E$ (z z))) ($\lambda z.$($\lambda y.\lambda x.E$ (z z)))) E'

The expression $\lambda x.E$ here represents some function that must be applied to argument E'. In all three expressions above free occurrences of y in $\lambda x.E$ represent recursive function calls. The second expression (b) provides an alternative to using the Y-combinator. In the expression (c) the mathematical definition of the Y-combinator has been used.

Example 4

var x_r ; assign $x_r = \lambda x.\lambda y_1.\lambda y_2.\lambda y_3.\lambda y_4.$ (if (equal (tail x) nil)
(assign y_3 = tail y_1 ; assign y_2= (cons (head x) y_4) ; (head y_1))
(var z_1 ; var z_2 ; (cons (x_r (head x) y_1 y_2 z_1 z_2) (x_r (tail x) z_1 z_2 y_3 y_4)))) ; ...

A defiency of the language is that it does not support the direct definition of functions by pattern-matching. However, as Joseph states, a method can be found to transform pattern-matching into simpler constructs. In the above example the expression assigned to x_r denotes a function that had been defined originally by pattern-matching. Some minor modifications led to the above.

4.3. Denotational semantics according to [Jo]

This paragraph describes the denotational semantics for the language as can be found in [Jo], chapter 4. The semantic valuation can be thought of as an abstract model for a graph reduction machine. Section 4.3.1 briefly discusses the process of graph reduction. Section 4.3.2 contains the semantical definitions along with some commentaries.

4.3.1 Graph reduction

Expressions are viewed as graphs. Reduction rules are now performed on these expression-graphs in such a way that explicit copying of subexpressions can be avoided. Reduction rules are applied until the graph is in the so-called head normal form or lazy normal form.

A lambda expression, an expression denoting a partially applied function, boolean and integer expressions, the expression nil and an expression as cons E_1 E_2 are in head normal form. In reducing an expression to head normal form a normal order reduction strategy is employed: evaluation of E_1E_2 involves the evaluation of E_1 and after that applying the resulting function to E_2.
Finally, it can be noted that recursive constructs give rise to circular graphs.

4.3.2 Semantics

The already mentioned normal order reduction strategy implies that the following semantics will only support sequential evaluation of the arguments of (standard) functions.

We start with the definitions of several semantical domains:

Definition 4.4

($\underline{b} \in$) Bool is the semantical set of boolean truth values { true , false }.

($\underline{m} \in$) Num is the semantical set of integer values.

Definition 4.5

The class of basic values ($\beta \in$) Bv is given by:

$\beta ::= \underline{m} \mid \underline{b} \mid \underline{\text{nil}}$

where $\underline{m} \in$ Num and $\underline{b} \in$ Bool.

Definition 4.6

($\alpha \in$) Loc is the set of locations.

An environment $\varrho \in$ Env is a partial function from identifiers to locations.

A store $\sigma \in$ Store is a mapping from locations to stored values.

A location is assigned to every declared variable. The environment is used to maintain this information. A location can also be assigned in order to store an arbitrary expression. The location is then passed through as argument. The store returns the stored value of every location. A stored value may be an expression in unevaluated form or an expressed value, that is the value of an expression in head normal form. The store marks unused locations as free. The exact definition of stored values is given in definition 4.8.

Definition 4.7

An answer A $\in$ Ans is defined by:

$$A ::= \beta \mid (A_1, A_2) \mid \text{suspend} \mid \text{error}$$

A continuation $\theta \in$ Cont is an element of Store $\rightarrow$ Ans $\times$ Store.

An expression continuation $\varkappa \in$ Econt is a mapping from Ev to Cont.

The final value (not the expressed value!) of an expression is an answer. An answer can be a basic value or a pair (list) of (two) answers. If during evaluation an error is encountered, the answer error is reported. If a state of deadlock arises, the answer becomes suspend. We define Ev in

Definition 4.8

A closure $\nu \in$ Clo is a mapping from Econt to Cont.

An expressed value $\epsilon \in$ Ev is defined by:

$$\epsilon ::= \beta \mid (\alpha_1, \alpha_2) \mid \phi$$

where $\alpha_1, \alpha_2 \in$ Loc and ϕ denotes a typical element of Loc $\rightarrow$ Clo.

A stored value s $\in$ Sv is given by:

$$s ::= \nu \mid \epsilon \mid \text{unset} \mid \text{NotReady} \mid \text{free}$$

A closure represents an expression in unevaluated form. An expressed value corresponds to the head normal form of an expression. ϕ denotes a (partially applied) function. The expressed value (α_1, α_2) stands for the reduction of an expression to a list. The stored value NotReady indicates that the expression associated with the concerned location is under evaluation. Unset is only used for uninstantiated variables.

Two auxiliary functions are defined in

Definition 4.9

The function new: Store $\rightarrow$ Loc is a function that satisfies:

$$\sigma(\text{new}(\sigma)) = \text{free}$$

In other words, new returns a free location.

The function wrong: Cont is defined by:

$\text{wrong}(\sigma) = (\text{error}, \sigma)$

A graph reduction machine is usually driven by a printing routine. A printing routine is defined in

Definition 4.10

The function print: Ev $\rightarrow$ Cont is defined by:

$\text{print}\ \epsilon =$
$\epsilon \in (\text{Loc} \rightarrow \text{Clo}) \rightarrow \text{wrong}\ ;$
$\epsilon \in (\text{Loc} \times \text{Loc}) \rightarrow \lambda\sigma.((A_1, A_2), \sigma_2)$
where $(\alpha_1, \alpha_2) = \epsilon\ ;\ (A_1, \sigma_1) = \text{force}(\alpha_1)(\text{print})(\sigma)\ ;\ (A_2, \sigma_2) = \text{force}(\alpha_2)(\text{print})(\sigma_1)$
$\epsilon \in \text{Bv} \rightarrow \lambda\sigma.\ (\epsilon, \sigma)$

An expression may have been reduced to a list. The printing routine now forces the evaluation of the elements of the list. An attempt to print a list results in an error.

We now turn to the definition of the semantical valuation &. We remark that & is only defined for well-formed expressions. That is, expressions that do not contain occurrences of undeclared variables.

Definition 4.11

The semantic valuation mapping & : Exp $\rightarrow$ Env $\rightarrow$ Econt $\rightarrow$ Cont is defined by:

$\&(x)(\varrho)(\varkappa) = \text{force}(\varrho(x))(\varkappa)$

$\&(\text{p})(\varrho)(\varkappa) = \varkappa(\text{P(p)})$

$\&(E_1\ E_2)(\varrho)(\varkappa) = \&(E_1)(\varrho)(\varkappa')$ where $\varkappa' = A(E_2)(\varrho)(\varkappa)$

$\&(\lambda x.E)(\varrho)(\varkappa) = \varkappa(\ \lambda\alpha.\&(E)(\varrho\{\alpha/x\})\ )$

$\&(\text{var}\ x\ ;\ E)(\varrho)(\varkappa) = \lambda\sigma.\ \&(E)(\varrho\{\alpha/x\})(\varkappa)(\sigma\{\text{unset}/\alpha\})$ where $\alpha = \text{new}(\sigma)$

$\&(\text{assign}\ x = E_1\ ;\ E_2)(\varrho)(\varkappa) = \lambda\sigma.\ (\ \sigma(\alpha) = \text{unset} \rightarrow$
$\&(E_2)(\varrho)(\varkappa)(\sigma\{\ \&(E_1)(\varrho)/\alpha\})\ ,\ (\text{error}, \sigma))$ where $\alpha = \varrho(x)$

Definition 4.12

The function force: Loc $\rightarrow$ Econt $\rightarrow$ Cont is defined by:

$\text{force}(\alpha)(\varkappa) = \lambda\sigma.\ \text{case}\ \sigma(\alpha)\ \text{of}$
NotReady, unset : $(\ \text{suspend}, \sigma)$
ϵ : $\varkappa(\epsilon)(\sigma)$
ν : $\nu(\varkappa')(\sigma\{\text{NotReady}/\alpha\})$ *where* $\varkappa' = \lambda\epsilon\sigma'.\varkappa(\epsilon)(\sigma'\{\epsilon/\alpha\})$

Definition 4.13

The function A : Exp $\rightarrow$ Env $\rightarrow$ Econt $\rightarrow$ Econt is defined by:

$A(E)\varrho\varkappa = \lambda\epsilon.\ \epsilon \in (\text{Loc} \rightarrow \text{Clo}) \rightarrow \theta, \text{wrong}$
where $\theta = E \in \text{Ide} \rightarrow \theta_1, \theta_2$
$\theta_1 = \epsilon(\varrho(E))(\varkappa)$
$\theta_2 = \lambda\sigma.\ \epsilon(\alpha)(\varkappa)(\sigma')$

where $\alpha = \text{new}(\sigma)$ and $\sigma' = \sigma\{\,\&(E)(\varrho)/\alpha\}$

The demand for the value of an identifier is propagated to its location and handled by the function force. In case the stored value is a closure, its evaluation is forced. During evaluation the location is marked NotReady. If evaluation has been completed, the expressed value is stored. In case the stored value is already an expressed value, force directly return this value. In case the stored value is NotReady or unset, force suspends the demand.

A primitive expression is already in head normal form. The corresponding expressed value P[p] is given as argument to the expression continuation. P is defined in definition 4.14. Likewise, a lambda expression is in head normal form. The expression continuation is also applied to the corresponding expressed value, a (partially applied) function in Loc → Clo.

The use of expression continuations clearly emerges in the equation for the application E_1E_2. First E_1 is evaluated, say its expressed value is ϵ. After that the expression continuation $A(E_2)(\varrho)(\varkappa)$ is applied to ϵ. Suppose that during reduction of E the store has changed from σ to σ'. Computation is now given by:

$$\&(E_1E_2)(\varrho)(\varkappa)(\sigma)=\&(E_1)(\varrho)(\,A(E_2)(\varrho)(\varkappa)\,)(\sigma)=A(E_2)(\varrho)(\varkappa)(\epsilon)(\sigma')$$

The function A applies ϵ, if it is a function, to the location of expression E_2. If E_2 is not an identifier, it is stored at some new location and this location is given as argument to ϵ. If E_2 is an identifier, its own location is passed through to ϵ. This mechanism accurately implements the lazy evaluation strategy as described in paragraph 4.1 and 4.2.

The equation for the expression var x; E shows that before E is evaluated a new location is assigned to x and marked as unset. Evaluation of assign $x = E_1$; E_2 results in an error, if x is an instantiated variable. If x is uninstantiated, E_1 is stored as a closure at a new location and evaluation proceeds with E_2.

We proceed with the definition of P.

Definition 4.14

The function P: Prim → Ev is defined by:

P(true) = true

P(false) = false

P(m) = $\underline{m}$ where $\underline{m}$ is the integer denoted by m

P(nil) = nil

P(not) = $\lambda\alpha\varkappa.$ force(α)($\lambda\epsilon.$ $\epsilon \in$ Bool → $\varkappa(\neg\epsilon)$, *wrong*)

P(head) = $\lambda\alpha\varkappa.$ force(α)($\lambda\epsilon.$ $\epsilon \in$ Loc×Loc → *force*(α_1)($\varkappa$) where $(\alpha_1, \alpha_2) = \epsilon$, wrong)

P(Y) = $\lambda\alpha\varkappa.$ force(α)($\lambda\epsilon.$ $\epsilon \in$ (Loc→Clo) → $\lambda\sigma.$ force(α')($\varkappa$)(σ'), wrong)

where $\alpha' = \text{new}(\sigma)$, $\sigma' = \sigma\{\,\epsilon(\alpha')/\alpha'\}$

P(cons) = $\lambda\alpha\varkappa.$ $\varkappa$(*cons* α)

P(add) = $\lambda\alpha\varkappa.$ $\varkappa$(*add* α)

P(div) = $\lambda\alpha\varkappa.$ $\varkappa$(*div* α)

P(val) = $\lambda\alpha\varkappa.$ $\varkappa$(*val* α)

P(seq) = $\lambda\alpha\varkappa.$ $\varkappa$(*seq* α)

P(if) = $\lambda\alpha\varkappa.$ $\varkappa$(*if* α)

where the functions in italic font are given by:

cons = $\lambda\alpha_1\alpha_2\varkappa.$ $\varkappa((\alpha_1,\alpha_2))$

add = $\lambda\alpha_1\alpha_2\varkappa.$ force(α_1)($\lambda\epsilon_1.$ $\epsilon_1 \in$ Num → θ, wrong)

where θ = force(α_2)($\lambda\epsilon_2.$ $\epsilon_2 \in$ Num → $\varkappa(\epsilon_1+\epsilon_2)$, wrong)

$div = \lambda\alpha_1\alpha_2\varkappa.\ \text{force}(\alpha_1)(\lambda\epsilon_1.\ \epsilon_1 \in \text{Num} \rightarrow \theta, \text{wrong})$

where $\theta = \text{force}(\alpha_2)(\lambda\epsilon_2.\ \epsilon_2 \in \text{Num and } (\epsilon_2 \neq 0) \rightarrow \varkappa(\epsilon_1 \div \epsilon_2)\ , \text{wrong})$

$val = \lambda\alpha_1\alpha_2\varkappa.\ \text{force}(\alpha_1)(\lambda\epsilon.\ \text{force}(\alpha_2)(\lambda\epsilon_2.\ \epsilon_2 \in (\text{Loc} \rightarrow \text{Clo}) \rightarrow \epsilon_2(\alpha_1)\varkappa\ , \text{wrong}))$

$seq = \lambda\alpha_1\alpha_2\varkappa.\ \text{force}(\alpha_1)(\lambda\epsilon_1.\ \text{force}(\alpha_2)(\varkappa))$

$if = \lambda\alpha_1\alpha_2\varkappa.\ \varkappa(if'\ \alpha_1\alpha_2)$

$if' = \lambda\alpha_1\alpha_2\alpha_3\varkappa.\ \text{force}(\alpha_1)(\lambda\epsilon.\ \epsilon \in \text{Bool} \rightarrow (\epsilon \rightarrow \text{force}(\alpha_2)(\varkappa)\ , \text{force}(\alpha_3)(\varkappa)), \text{wrong})$

The semantical meaning of the primitives is reasonably straightforward. Keep in mind that A passes through the locations of the arguments of primitive functions. Note that a circular structure is constructed in the equation of the semantical meaning of Y. Furthermore, the semantical meaning of cons is particularly simple. As can be seen, the standard functions evaluate their arguments sequentially. The equation of val has been adapted to the fact that its arguments have been turned around.

We further remark that the semantics can be accomodated in a very simple way to support the extension of the class of primitive functions with mul, sub, and, or, equal, etc.

Example

The semantical meaning of equal is given by:

$\text{P(equal)} = \lambda\alpha\varkappa.\ \varkappa(equal\,\alpha)$

$equal = \lambda\alpha_1\alpha_2\varkappa.\ \text{force}(\alpha_1)(\lambda\epsilon_1.\ \epsilon_1 \in \text{Bv} \rightarrow \theta, \text{wrong})$

where $\theta = \text{force}(\alpha_2)(\lambda\epsilon_2.\ \epsilon_2 \in \text{Bv} \rightarrow \varkappa(\epsilon_1 = \epsilon_2), \text{wrong})$

Equal returns true if its two arguments are identical basic values.

We conclude this paragraph with another

Example

This example shows a typical computation in which r_0 stands for the arid environment in which no variables are declared and σ_f denotes the store in which all locations are free. Furtermore, $\varkappa_0$ is the printing routine. However, it is not 'typical' but indeed possible that evaluation of an expression does not terminate, as is shown by the example. Notice that the expression does contain instances of every kind of language construct. We assume that the reader understands the shortened way the computation has been written down.

$\&\ (\ (\text{add}\ (\ \text{var}\ y;\ \text{assign}\ y = \lambda x.(\ y\ 0\ );\ (\ y\ 1\ )\ ))\ 3\ )\ (\ r_0\ )\ (\ \varkappa_0\ )\ (\ \sigma_f\ ) =$

$\&\ (\ \text{add}\ E_1\ )\ (\ r_0\ )\ (\ A\ (\ 3\ )\ (\ r_0\ )\ (\ \varkappa_0\ )\ )\ (\ \sigma_f\ ) =$

$\&\ (\ \text{add}\ )\ (\ r_0\ )\ (\ A\ (\ E_1\ )\ (r_0\ )\ (\ A\ (\ 3\ )\ (\ r_0\ )\ (\ \varkappa_0\ )\ )\ (\ \sigma_f\ ) =$

$A\ (\ E_1\ )\ (\ r_0\ )\ (\ \varkappa_1\ )\ (\ \text{P}\ (\ \text{add}\ )\ )\ (\ \sigma_f\ ) =$

$\text{P}\ (\ \text{add}\ )\ (\ \alpha_1\ )\ (\ \varkappa_1\ )\ (\ \sigma_f\ \{\ \&\ (\ E_1\ )\ (\ r_0\ )\ /\ \alpha_1\ \}\ ) =$

$\varkappa_1\ (\ add\ \alpha_1\ )\ (\ \sigma_1\ ) =$

$(\ add\ \alpha_1\ )\ (\ \alpha_2\ )\ (\ \varkappa_1\ )\ (\ \sigma_1\ \{\ \&\ (\ 3\ )\ (\ r_0\ )\ /\ \alpha_2\ \}\ ) =$

$\text{force}\ (\ \alpha_1\ )\ (\ \lambda\epsilon_1.\ \epsilon_1 \in \text{Num} \rightarrow \theta_1, \text{wrong}\ )\ (\ \sigma_2\ ) =$

$\&\ (\ \text{var}\ y;\ \text{assign}\ y = \lambda x.(\ y\ 0\ );\ (\ y\ 1\ )\ )\ (\ r_0\ )\ (\ \lambda\epsilon\sigma'.\ \varkappa_2\ (\ \epsilon\ )\ (\ \sigma'\ \{\ \epsilon\ /\ \alpha_1\ \}\ )\ )\ (\ \sigma_2\ \{\ \text{NotReady}\ /\ \alpha_1\ \}\ ) =$

$\&\ (\ \text{assign}\ y = \lambda x.(\ y\ 0\ );\ (\ y\ 1\ )\ )\ (\ r_0\ \{\ \alpha_3\ /\ y\ \}\ )\ (\ \sigma'_2\ \{\ \text{unset}\ /\ \alpha_3\ \}\ ) =$

$\&\ (\ (\ y\ 1\ )\ )\ (\ \varrho_1\ )\ (\ \varkappa_3\ )\ (\ \sigma'_2\ \{\ \&\ (\ \lambda x.(\ y\ 0\ )\ )\ (\ \varrho_1\ )\ /\ \alpha_3\ \}\ ) =$

$\&\ (\ y\ )\ (\ \varrho_1\ )\ (\ A\ (\ 1\ )\ (\ \varrho_1\ )\ (\ \varkappa_3\ )\ )\ (\ \sigma_3\ ) =$

force (α_3) ($\varkappa_4$) (σ_3) =

& ($\lambda x.$(y 0)) (ϱ_1) ($\lambda\epsilon\sigma'.$ $\varkappa_4$ (ϵ) (σ' { ϵ / α_3 })) (σ'_2 { NotReady / α_3 }) =

$\varkappa_5$ ($\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x })) (σ'_2 { NotReady / α_3 }) =

$\varkappa_4$ ($\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x })) (σ'_2 { $\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x }) / α_3 }) =

($\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x })) (α_4) ($\varkappa_3$) (σ'_2 { & ((y 0)) (ϱ_1 { α / x }) / α_3 , & (1) (ϱ_1) / α_4 }) =

& (y) (ϱ_1 { α_4 / x }) (A (0) (ϱ_1 { α_4 / x }) ($\varkappa_3$)) (σ_4) =

force (α_3) ($\varkappa_6$) (σ_4) = (*)

$\varkappa_6$ ($\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x })) (σ_4) =

($\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x })) (α_5) ($\varkappa_3$) (σ_4 { & (0) (ϱ_1 { α_4 / x }) / α_5 }) =

& (y) (ϱ_1 { α_5 / x }) (A (0) (ϱ_1 { α_5 / x }) ($\varkappa_3$)) (σ_5) =

force (α_3) ($\varkappa_7$) (σ_5) = (*)

$\varkappa_7$ ($\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x })) (σ_5) =

($\lambda\alpha.$ & ((y 0)) (ϱ_1 { α / x })) (α_6) ($\varkappa_3$) (σ_5 { & (0) (ϱ_1 { α_5 / x }) / α_6 }) =

& (y) (ϱ_1 { α_6 / x }) (A (0) (ϱ_1 { α_6 / x }) ($\varkappa_3$)) (σ_6) =

force (α_3) ($\varkappa_8$) (σ_6) = (*)

etc.

4.4. Operational semantics

This paragraph is organized as follows: section 4.4.1 contains several preparatory definitions. The operational semantics given in section 4.4.2 supports only sequential evaluation. The operational semantics of section 4.4.3 supports concurrent evaluation of the arguments of standard functions. In section 4.4.4 the language is extended with the par-combinator. Again, an operational semantics is given. Section 4.4.5 concludes this paragraph with some examples.

4.4.1. Preliminary definitions

In this section the operational counterparts of the notions of expressed value, closure, stored value and store are defined. Furthermore, operational expressions and operational environments are introduced.

First we recall

Definition 4.15

A basic value $\beta \in$ Bv is given by:

$\beta ::= \underline{b} \mid \underline{m} \mid \underline{\text{nil}}$

where $\underline{b} \in$ Bool and $\underline{m} \in$ Num

We define

Definition 4.16

A partially applied function $\phi \in$ Paf is given by:

$\phi ::= [\lambda x.E] \mid [\text{p}]\alpha_1 \ldots \alpha_m$

where $x \in$ Ide, $E \in$ Exp, $\alpha_1, \ldots, \alpha_m \in$ Loc and p is a primitive function with arity n ; $0 \leqq m < n$

The syntactical objects $[\lambda x.E]$ and [p] denote the expressed values of the expressions $\lambda x.E$ and p which are in head normal form. We further denote by <p> the actual semantical meaning of p. In this manner <add> for instance is the wellknown function + : Num × Num → Num.

Definition 4.17

The class of operational expressed values ($\epsilon \in$) Ev' is given by:

$\epsilon ::= \beta \mid (\alpha_1, \alpha_2) \mid \phi$

In other words: Ev' = Bv ∪ (Loc × Loc) ∪ Paf.

We denote by δ a typical element of Ev' \ Bv. Thus:

$\delta ::= (\alpha_1, \alpha_2) \mid [\lambda x.E] \mid [p]\alpha_1 \ldots \alpha_m$

We now give the general shape of the definitions of operational expressions. For each operational semantics it is indicated later on which subset is needed.

Definition 4.18

The class of operational expressions ($\tilde{E} \in$) Exp($\tilde{E}$) is defined by :

$\tilde{E} ::= E \mid \tilde{E}\ E \mid \tilde{E} \Rightarrow \alpha \mid \beta \Rightarrow \alpha \mid \delta \mid [p]\mathrm{arg}_1 \ldots \mathrm{arg}_n$

where $E \in$ Exp, $\alpha \in$ Loc, $\beta \in$ Bv, $\delta \in$ Ev'\Bv; p is a primitive function with arity n; arg_i satisfies:

$\mathrm{arg}_i ::= \alpha \mid \tilde{E} \Rightarrow \alpha \mid \epsilon|_\alpha$

where $\alpha \in$ Loc and $\epsilon \in$ Ev'

The operational expression $\tilde{E} \Rightarrow \alpha$ denotes an expression that is being evaluated and whose expressed value must be stored into location α afterwards. In this manner $\beta \Rightarrow \alpha$ represents a basic value that still must be stored into α. The operational expression $[p]\mathrm{arg}_1 \ldots \mathrm{arg}_n$ pictures the evaluation of the arguments of a primitive function. Sequential evaluation, concurrent evaluation of arguments of standard functions and concurrent evaluation of the arguments of the par-combinator are considered. In each case an adjusted definition of operational expression is given, that which amounts to imposing certain conditions upon arg_i ($1 \leqq i \leqq n$). The argument $\epsilon|_\alpha$ denotes an expressed value ϵ that is stored at location α.

We now proceed with

Definition 4.19

The class of operational environments $\varrho \in$ Env' is given by:

Env' = (Ide → Loc) ∪ (Env' × Env')

An operational environment is a partial function from identifiers to locations or a pair of operational environments. We denote by $\mathbf{r}_0$ the empty partial function, the arid environment in which no variables are declared.

Every subexpression of an expression needs its own environment. In, for instance, the expression (var x; E_1 E_2) the identifier x is known in E_1, not in E_2. In the denotational semantics the environment of a subexpression was passed through in the expression continuation:

$\&(\mathrm{var}\ x; E_1\ E_2)(\varrho)(\varkappa)(\sigma) =$

$\&(\mathrm{var}\ x; E_1)(\varrho)(A\,(E_2)(\varrho)(\varkappa))(\sigma) =$

$\&(E_1)(\varrho\{\alpha/x\})(A\,(E_2)(\varrho)(\varkappa))(\sigma\{\mathrm{unset}/\alpha\})$

In the operational semantics the operational environment keeps track of this information. Furthermore, the notion of operational environment is already accomodated to parallel evaluation.

As already said, an operational environment for an expression must contain an environment for each subexpression. In the following definition we make precise which environment fits which expression. We define a mapping from operational expressions to subsets of the operational environments. It maps an operational expression to the subset of the operational environments that consists of the operational environments that fit the expression.

Definition 4.20

The mapping Fit: $\mathrm{Exp}(\tilde{E}) \rightarrow \mathcal{P}(\mathrm{Env}')$ is recursively defined by:

a.

$\mathrm{Fit}(x) = \mathrm{Ide} \rightarrow \mathrm{Loc}$

$\mathrm{Fit}(\mathrm{p}) = \mathrm{Ide} \rightarrow \mathrm{Loc}$

$\mathrm{Fit}(\mathrm{var}\ x\,;\ E) = \mathrm{Fit}(E)$

$\mathrm{Fit}(\lambda x.E) = \mathrm{Fit}(E)$

$\mathrm{Fit}(E_1\ E_2) = \mathrm{Fit}(E_1) \times \mathrm{Fit}(E_2)$

$\mathrm{Fit}(\mathrm{assign}\ x = E_1;\ E_2) = (\mathrm{Fit}(x) \times \mathrm{Fit}(E_1)) \times \mathrm{Fit}(E_2)$

b.

$\mathrm{Fit}(\tilde{E}\ E) = \mathrm{Fit}(\tilde{E}) \times \mathrm{Fit}(E)$

c.

$\mathrm{Fit}(\tilde{E} \Rightarrow \alpha) = \mathrm{Fit}(\tilde{E})$

d.

$\mathrm{Fit}(\beta \Rightarrow \alpha) = \{\mathrm{r}_0\}$

e.

$\mathrm{Fit}((\alpha_1, \alpha_2)) = \{\mathrm{r}_0\} \times \{\mathrm{r}_0\}$

$\mathrm{Fit}([\lambda x.E]) = \mathrm{Fit}(\lambda x.E)$

$\mathrm{Fit}([\mathrm{p}]) = \{\mathrm{r}_0\}$

$\mathrm{Fit}([\mathrm{p}]\alpha_1 \ldots \alpha_m) = \mathrm{Fit}([\mathrm{p}]\alpha_1 \ldots \alpha_{m-1}) \times \{\mathrm{r}_0\} \qquad (\text{for } m > 0)$

f.

$\mathrm{Fit}([\mathrm{p}]\mathrm{arg}_1 \ldots \mathrm{arg}_n) = ((\{\mathrm{r}_0\} \times \mathrm{R}(\mathrm{arg}_1)) \times \ldots \times \mathrm{R}(\mathrm{arg}_n))$

where $\mathrm{R}(\mathrm{arg}_i)$ is given by:

$\mathrm{R}(\alpha) = \{\mathrm{r}_0\}$

$\mathrm{R}(\tilde{E} \Rightarrow \alpha) = \mathrm{Fit}(\tilde{E})$

$\mathrm{R}(\beta|_\alpha) = \{\mathrm{r}_0\}$

$\mathrm{R}(\delta|_\alpha) = \mathrm{Fit}(\delta)$

$\mathrm{Fit}(\tilde{E})$ consists of those operational environments which fit operational expression $\tilde{E}$.

Example

Consider the expression

$(\text{var } x\text{; assign } x = \lambda y.E_1\ ;\ E_2\ \ (\lambda y.E_3\ \ E_4))$

Suppose $\varrho_i \in \text{Fit}(E_i)$ for $i = 1,\ldots,4$ and $\hat{\varrho} \in \text{Ide} \rightarrow \text{Loc}$. Then

$(((\hat{\varrho}, \varrho_1), \varrho_2), (\varrho_3, \varrho_4))$

is an operational environment that fits the above expression. Suppose now that $\text{Fit}(E_i) = \text{Ide} \rightarrow \text{Loc}$ for $i = 1,\ldots,4$. We now call

$(((r_0, r_0), r_0), (r_0, r_0))$

the fitting arid operational environment of the given expression.

We now define

Definition 4.21

An operational closure $\nu \in \text{Clo}'$ is a pair (ϱ, E) such that $\varrho \in \text{Fit}(E)$ and $E \in \text{Exp}$.

Definition 4.22

a. The class of operational stored values $(s \in)\ \text{Sv}'$ is given by:

$s ::= \beta \mid (\delta, \varrho_1) \mid (\varrho_2, E) \mid \text{unset} \mid \text{NotReady} \mid \text{free}$

where $\beta \in \text{Bv}$, $\delta \in \text{Ev}' \backslash \text{Bv}$, $\varrho_1 \in \text{Fit}(\delta)$ and $(\varrho_2, E) \in \text{Clo}'$

b. The class of operational stores $(\sigma \in)\ \text{Store}'$ is given by:

$\text{Store}' = \text{Loc} \rightarrow \text{Sv}'$

Finally, we give the modified version of the definiton of new and define two other auxiliary functions.

Definition 4.23

The auxiliary function new : $\text{Store}' \rightarrow \text{Loc}$ satisfies:

$\sigma(\text{new}(\sigma)) = \text{free}$

Definition 4.24

Mod : $\text{Env}' \rightarrow (\text{Ide} \rightarrow (\text{Loc} \rightarrow \text{Env}'))$ is given by:

$\text{Mod}(\varrho)(x)(\alpha) = \varrho\{\alpha/x\}$ for $\varrho \in \text{Ide} \rightarrow \text{Loc}$

$\text{Mod}(\varrho)(x)(\alpha) = (\text{Mod}(\varrho_1)(x)(\alpha), \text{Mod}(\varrho_2)(x)(\alpha))$ for $\varrho = (\varrho_1, \varrho_2) \in \text{Env}' \times \text{Env}'$

Definition 4.25

Sel : $\mathbb{N} \rightarrow (\text{Env}' \rightarrow \text{Env}')$ is given by:

$\text{Sel}(1)((\varrho_1, \varrho_2)) = \varrho_2$

$\text{Sel}(i)((\varrho_1, \varrho_2)) = \text{Sel}(i-1)(\varrho_1)$ for $i > 1$

$\text{Sel}(k)(\varrho) = \varrho$ for $\varrho \in \text{Ide} \rightarrow \text{Loc}$ and $k \in \mathbb{N}$

Example

Suppose that $\varrho_0, \varrho_1 \in \text{Ide} \rightarrow \text{Loc}$; $\varrho_2 = \varrho_0\{\alpha/x\}$ and $\varrho_3 = \varrho_1\{\alpha/x\}$

Then

$\text{Sel}(1)(\text{Mod}((((\varrho_0, \varrho_1), \varrho_1), (\varrho_0, \varrho_0)))(x)(\alpha)) = (\varrho_2, \varrho_2)$

$\text{Sel}(2)(\text{Mod}((((\varrho_0, \varrho_1), \varrho_1), (\varrho_0, \varrho_0)))(x)(\alpha)) = \varrho_3$

$\text{Sel}(3)(\text{Mod}((((\varrho_0, \varrho_1), \varrho_1), (\varrho_0, \varrho_0)))(x)(\alpha)) = \varrho_3$

$\text{Sel}(4)(\text{Mod}((((\varrho_0, \varrho_1), \varrho_1), (\varrho_0, \varrho_0)))(x)(\alpha)) = \varrho_2$

$\text{Sel}(k)(\text{Mod}((((\varrho_0, \varrho_1), \varrho_1), (\varrho_0, \varrho_0)))(x)(\alpha)) = \varrho_2$ for $k > 4$

4.4.2. Sequential evaluation

In this section we present an operational semantics that only supports sequential evaluation of the arguments of a function just as the denotational semantics in paragraph 4.3.

First we define

Definition 4.26

The class of operational expressions ($\tilde{E} \in$) $\text{Exp}(\tilde{E}_1)$ for sequential evaluation is defined by :

$\tilde{E} ::= E \mid \tilde{E}\ E \mid \tilde{E} \Rightarrow \alpha \mid \beta \Rightarrow \alpha \mid \delta \mid [\text{p}]\text{arg}_1 \ldots \text{arg}_n$

where $E \in \text{Exp}$, $\alpha \in \text{Loc}$, $\beta \in \text{Bv}$, $\delta \in \text{Ev}' \backslash \text{Bv}$ and p is a primitive function with arity n;
arg_i $(1 \leqq i \leqq n)$ satisfies:

(i) $\text{arg}_i = \epsilon|_\alpha$ with $\epsilon \in \text{Ev}'$ and $\alpha \in \text{Loc}$ only if $i = 1$ or ($i > 1$ and arg_{i-1} satisfies (i))

or (ii) $\text{arg}_i = E' \Rightarrow \alpha$ with $E' \in \text{Exp}(\tilde{E}_1)$ and $\alpha \in \text{Loc}$ only if $i = 1$ or ($i > 1$ and arg_{i-1} satisfies (i))

or (iii) $\text{arg}_i \in \text{Loc}$.

The operational expression $[\text{p}]\text{arg}_1 \ldots \text{arg}_n$ amounts to

$[\text{p}]\ \epsilon_1|_{\alpha_1} \ldots \epsilon_{i-1}|_{\alpha_{i-1}} (\tilde{E}_i \Rightarrow \alpha_i)\ \alpha_{i+1} \ldots \alpha_n \qquad (\text{with } 0 \leqq i \leqq n)$

The first $i-1$ arguments have been evaluated already. Argument i is being evaluated. The arguments to the right have not been evaluated yet, but they are stored at location α_j $(i < j \leqq n)$.

We now define the configurations in

Definition 4.27

A configuration $c \in \text{Conf}_1$ is defined by:

$c ::= \langle \tilde{E}, \varrho, \sigma \rangle \mid \langle \beta, \sigma \rangle$

where $\tilde{E} \in \text{Exp}(\tilde{E}_1)$, $\varrho \in \text{Fit}(\tilde{E})$, $\sigma \in \text{Store}'$, $\beta \in \text{Bv}$.

Remark

The transitions have the form:

$\langle \tilde{E}, \varrho, \sigma \rangle \rightarrow \langle \beta, \sigma' \rangle$

or

$$< \tilde{E}, \varrho, \sigma > \rightarrow < \tilde{E}', \varrho', \sigma' >$$

Convention

In the transition system T_1 defined in definition 4.28 the following holds (unless stated otherwise):

$\varrho_i = \text{Sel}(n-i+1)(\varrho)$

We stress that this only holds for ϱ, not for ϱ', $\tilde{\varrho}$, etc.

We now define the transition system that determines the transition relation $\rightarrow$.

Definition 4.28

The transition system T_1 is defined by the following axioms and rules:

Axiom 1

$$< \text{true}, \varrho, \sigma > \rightarrow < \underline{\text{true}}, \sigma >$$

Axiom 2

$$< \text{false}, \varrho, \sigma > \rightarrow < \underline{\text{false}}, \sigma >$$

Axiom 3

$$< m, \varrho, \sigma > \rightarrow < \underline{m}, \sigma >$$

provided that
(i) $\underline{m}$ is the integer denoted by m

Axiom 4

$$< \text{nil}, \varrho, \sigma > \rightarrow < \underline{\text{nil}}, \sigma >$$

Axiom 5

$$< \text{var } x; E, \varrho, \sigma > \rightarrow < E, \varrho', \sigma\{\text{unset}/\alpha\} >$$

provided that
(i) $\alpha = \text{new}(\sigma)$
(ii) $\varrho' = \text{Mod}(\varrho)(x)(\alpha)$

Axiom 6

$$< \text{assign } x = E_1; E_2, \varrho, \sigma > \rightarrow < E_2, \varrho_2, \sigma\{(\varrho_1, E_1)/\alpha\} >$$

provided that

(i) $\varrho_0(x)=\alpha$
(ii) $\sigma(\alpha)$=unset

Axiom 7

$$< \lambda x.E, \varrho, \sigma> \rightarrow < [\lambda x.E], \varrho, \sigma>$$

Axiom 8

$$< p, \varrho, \sigma> \rightarrow < [p], r_0, \sigma>$$

provided that
(i) p is a primitive function of positive arity

Axiom 9

$$< [\lambda x.E]\ E', \varrho, \sigma> \rightarrow < E, \varrho', \sigma'>$$

provided that
(i) if $E' \in$ Ide then $\alpha=\varrho_1(E')$; $\sigma' = \sigma$
(ii) if $E' \notin$ Ide then α=new(σ) ; $\sigma'=\sigma\{ (\varrho_1, E')/\alpha\}$
(iii) ϱ'= Mod(ϱ_0)(x)(α)

Axiom 10

$$< [p]\alpha_1 \ldots \alpha_m E', \varrho, \sigma> \rightarrow < [p]\alpha_1 \ldots \alpha_m \alpha_{m+1}, \varrho', \sigma'>$$

provided that
(i) p is a primitive function with arity n; $0 \leqq m < n$
(ii) if $E' \in$ Ide then $\alpha_{m+1} = \varrho_{m+1}(E')$; $\sigma' = \sigma$
(iii) if $E' \notin$ Ide then α_{m+1} =new(σ) ; $\sigma' = \sigma\{ (\varrho_{m+1}, E')/\alpha_{m+1}\}$
(iv) ϱ' satisfies
(a) Sel(1)(ϱ') = r_0
(b) for $k>1$ Sel(k)(ϱ') = Sel(k)(ϱ)

Axiom 11

$$< x, \varrho, \sigma> \rightarrow < \beta, \sigma>$$

provided that
(i) $\varrho(x)=\alpha$
(ii) $\sigma(\alpha)=\beta$

Axiom 12

$$< x, \varrho, \sigma> \rightarrow < \delta, \varrho', \sigma>$$

provided that
(i) $\varrho(x)=\alpha$
(ii) $\sigma(\alpha)= (\delta, \varrho')$

Axiom 13

$$< x, \varrho, \sigma> \rightarrow < E \Rightarrow \alpha, \varrho', \sigma\{\text{NotReady}/\alpha\}>$$

provided that
(i) $\varrho(x)=\alpha$
(ii) $\sigma(\alpha)= (\varrho', E)$

Axiom 14

$$< \beta \Rightarrow \alpha, r_0, \sigma> \rightarrow < \beta, \sigma\{\beta/\alpha\}>$$

Axiom 15

$$< \delta \Rightarrow \alpha, \varrho, \sigma> \rightarrow < \delta, \varrho, \sigma\{(\delta, \varrho)/\alpha\}>$$

Axiom 16

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ (E_i \Rightarrow \alpha_i)\ arg_{i+1} \ldots arg_n, \varrho', \sigma'>$$

provided that
(i) p is a primitive function with arity n, cons excluded
(ii) if p≡if then $i=1$
(iii) $\sigma(\alpha_i)= (\varrho'_i, E_i)$
(iv) ϱ' satisfies:
(a) for $k \neq n-i+1$ $\text{Sel}(k)(\varrho') = \text{Sel}(k)(\varrho)$
(b) $\text{Sel}(n-i+1)(\varrho') = \varrho'_i$
(v) $\sigma' = \sigma\{\text{NotReady}/\alpha_i\}$

Axiom 17

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \beta_i|_{\alpha_i}\ arg_{i+1} \ldots arg_n, \varrho, \sigma>$$

provided that
(i) p is a primitive function with arity n, cons excluded
(ii) if p≡if then $i=1$
(iii) $\sigma(\alpha_i)= \beta_i$

Axiom 18

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \delta_i|_{\alpha_i}\ arg_{i+1} \ldots arg_n, \varrho', \sigma>$$

provided that
(i) p is one of the primitive functions seq, val, head or Y
(ii) n is the arity of p
(iii) $\sigma(\alpha_i)= (\delta_i, \varrho'_i)$
(iv) ϱ' satisfies:
(a) for $k \neq n-i+1$ $\text{Sel}(k)(\varrho') = \text{Sel}(k)(\varrho)$
(b) $\text{Sel}(n-i+1)(\varrho') = \varrho'_i$

Axiom 19

$$< [p]\ arg_1 \ldots arg_{i-1}\ (\epsilon \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho, \sigma > \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \epsilon|_\alpha\ arg_{i+1} \ldots arg_n, \varrho, \sigma' >$$

provided that
(i) p is a primitive function with arity n
(ii) if $\epsilon \in$ Bv then $\sigma' = \sigma\{\epsilon/\alpha\}$
(iii) if $\epsilon \in$ Ev' \ Bv then $\sigma' = \sigma\{(\epsilon, \varrho_i)/\alpha\}$

Before giving additional axioms for specific primitive functions, we give the only two rules:

Rule 1

$$\frac{< \tilde{E}, \tilde{\varrho}, \sigma > \rightarrow < \tilde{E}', \tilde{\varrho}', \sigma' >}{< \tilde{E}\ E'', (\tilde{\varrho}, \varrho''), \sigma > \rightarrow < \tilde{E}'\ E'', (\tilde{\varrho}', \varrho''), \sigma' >}$$

$$< \tilde{E} \Rightarrow \alpha, \tilde{\varrho}, \sigma > \rightarrow < \tilde{E}' \Rightarrow \alpha, \tilde{\varrho}', \sigma' >$$

$$< [p]\ arg_1 \ldots arg_{i-1}\ (\tilde{E} \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho, \sigma > \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ (\tilde{E}' \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho', \sigma' >$$

provided that in the third conclusion
(i) p is a primitive function with arity n
(ii) $Sel(n-i+1)(\varrho) = \tilde{\varrho}$
(iii) ϱ' satisfies:
(a) for $k \neq n-i+1$ $Sel(k)(\varrho') = Sel(k)(\varrho)$
(b) $Sel(n-i+1)(\varrho') = \tilde{\varrho}'$

Rule 2

$$\frac{< \tilde{E}, \tilde{\varrho}, \sigma > \rightarrow < \beta, \sigma' >}{< \tilde{E} \Rightarrow \alpha, \tilde{\varrho}, \sigma > \rightarrow < \beta \Rightarrow \alpha, r_0, \sigma' >}$$

$$< [p]\ arg_1 \ldots arg_{i-1}\ (\tilde{E} \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho, \sigma > \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ (\beta \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho', \sigma' >$$

provided that in the second conclusion
(i) p is a primitive function with arity n
(ii) $Sel(n-i+1)(\varrho) = \tilde{\varrho}$
(iii) ϱ' satisfies:
(a) for $k \neq n-i+1$ $Sel(k)(\varrho') = Sel(k)(\varrho)$
(b) $Sel(n-i+1)(\varrho') = r_0$

Axiom 20

$$< [p]\ \beta_1|_{\alpha_1} \ldots \beta_n|_{\alpha_n}, \varrho, \sigma > \rightarrow < \beta, \sigma >$$

provided that
(i) p is a standard primitive function such as add, div, or not
(ii) n is the arity of p
(iii) $\beta = <p> (\beta_1, \ldots, \beta_n)$

Axiom 21

$$< [seq]\ \epsilon_1|_{\alpha_1}\ \epsilon_2|_{\alpha_2}, \varrho, \sigma > \rightarrow < \epsilon_2, \sigma >_1 \ |\ < \epsilon_2, \varrho_2, \sigma >_2$$

provided that
(i) in case 1 : $\epsilon_2 \in Bv$
(ii) in case 2 : $\epsilon_2 \in Ev' \setminus Bv$

Axiom 22

$$< [val]\ \epsilon_1|_{\alpha_1}\ [\lambda x.E]\ |_{\alpha_2}\ , \varrho, \sigma> \rightarrow < E, \varrho', \sigma>$$

provided that
(i) $\varrho' = Mod(\varrho_2)(x)(\alpha_1)$

Axiom 23

$$< [val]\ \epsilon_1|_{\alpha}\ ([p]\alpha_1 \ldots \alpha_m)\ |_{\alpha'}\ , \varrho, \sigma> \rightarrow < [p]\alpha_1 \ldots \alpha_m \alpha, (\varrho_2, r_0), \sigma>$$

provided that
(i) p is a primitive function with arity n ; $0 \leqq m < n$

Axiom 24

$$< [if]\ \beta|_{\alpha_1}\ \alpha_2\ \alpha_3, \varrho, \sigma> \rightarrow < \beta_i, \sigma>_1\ |\ < \delta_i, \varrho'_i, \sigma>_2\ |\ < E_i \Rightarrow \alpha_i, \varrho'_i, \sigma\{\text{NotReady}/\alpha_i\}>_3$$

provided that
(i) ($\beta = tt$ and $i=2$) or ($\beta = ff$ and $i=3$)
(ii) in case 1 : $\sigma(\alpha_i) = \beta_i$
(iii) in case 2 : $\sigma(\alpha_i) = (\delta_i, \varrho'_i)$
(iv) in case 3 : $\sigma(\alpha_i) = (\varrho'_i, E_i)$

The cases 1,2,3 correspond with the configurations with subscripts 1,2,3.

Axiom 25

$$< [cons]\ \alpha_1\ \alpha_2, \varrho, \sigma> \rightarrow < (\alpha_1, \alpha_2), (r_0, r_0), \sigma>$$

Axiom 26

$$< [head]\ (\alpha_1, \alpha_2)\ |_{\alpha}, \varrho, \sigma> \rightarrow < \beta_1, \sigma>_1\ |\ < \delta_1, \varrho'_1, \sigma>_2\ |\ < E_1 \Rightarrow \alpha_1, \varrho'_1, \sigma\{\text{NotReady}/\alpha_1\}>_3$$

provided that
(i) in case 1 : $\sigma(\alpha_1) = \beta_1$
(ii) in case 2 : $\sigma(\alpha_1) = (\delta_1, \varrho'_1)$
(iii) in case 3 : $\sigma(\alpha_1) = (\varrho'_1, E_1)$

Axiom 27

$$< [Y]\ [\lambda x.E]\ |_{\alpha_1}\ , \varrho, \sigma> \rightarrow < E \Rightarrow \alpha, \varrho', \sigma\{\text{NotReady}/\alpha\}>$$

provided that
(i) $\alpha = new(\sigma)$
(ii) $\varrho' = Mod(\varrho_1)(x)(\alpha)$

Axiom 28

$$< [Y] ([p]\alpha_1 \ldots \alpha_m) \mid_{\alpha} , \varrho, \sigma> \rightarrow < [p]\alpha_1 \ldots \alpha_m \alpha' \Rightarrow \alpha', (\varrho_1, r_0), \sigma\{ \text{NotReady}/ \alpha'\}>$$

provided that
(i) p is a primitive function with arity n ; $0 \leqq m < n$
(ii) $\alpha' = \text{new}(\sigma)$

Note that the transition system accurately follows 'the steps' of the denotational semantics in 4.3, that is, it models graph reduction.

We define the operational meaning of an expression in definition 4.29. $< \tilde{E}, \varrho, \sigma> \not\rightarrow$ denotes that there is no transition $< \tilde{E}, \varrho, \sigma> \rightarrow c$ for any $c \in \text{Conf}_1$. Furthermore, we remark that for $c_1, c_2 \in \text{Conf}_1$ such that $c_1 \neq c_2$ it is not possible that $< \tilde{E}, \varrho, \sigma> \rightarrow c_1$ and $< \tilde{E}, \varrho, \sigma> \rightarrow c_2$.

Definition 4.29

a. The mapping O_1: Exp $\rightarrow$ (Store' $\rightarrow$ Ans $\times$ Store') is defined by:

$O_1(E)(\sigma) = \overline{O}_1(< E, \varrho, \sigma>)$

where ϱ is the fitting arid operational environment of E

b. The mapping $\overline{O}_1$: $\text{Conf}_1 \rightarrow$ Ans $\times$ Store' is defined by:

$\overline{O}_1(< \beta, \sigma>) = (\beta, \sigma)$

$\overline{O}_1(< E, \varrho, \sigma>)$ is given by:

if there is a finite transition sequence $< E, \varrho, \sigma> = c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_n \not\rightarrow$ then

in case $c_n = < \beta, \sigma'>$ then

$\overline{O}_1(c_0) = (\beta, \sigma')$

in case $c_n = < \phi, \varrho', \sigma'>$ then

$\overline{O}_1(c_0) = (\text{error}, \sigma')$

in case $c_n = < (\alpha_1, \alpha_2), (r_0, r_0), \sigma'>$ then

$\overline{O}_1(c_0) = ((A_1, A_2), \sigma_2)$

where $(A_1, \sigma_1) = \mathcal{K}(\alpha_1, \sigma)$ and $(A_2, \sigma_2) = \mathcal{K}(\alpha_2, \sigma_1)$

otherwise: say $c_n = < \tilde{E}', \varrho', \sigma'>$

$\overline{O}_1(c_0) = (\text{suspend}, \sigma')$

else if there is an infinite transition sequence $<E, \varrho, \sigma> = c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_n \rightarrow \ldots$ then

$\overline{O}_1(c_0) = (\text{error}, \sigma)$

c. The mapping $\mathcal{K}$: Loc $\times$ Store' $\rightarrow$ Ans $\times$ Store' is defined by:

$\mathcal{K}(\alpha, \sigma) =$

if $\sigma(\alpha) =$ unset, NotReady, free then (suspend, σ)
if $\sigma(\alpha) = \beta$ then (β, σ)
if $\sigma(\alpha) = (\phi, \varrho)$ then (error, σ)
if $\sigma(\alpha) = (\varrho, E)$ then $\overline{O}_1(< E, \varrho, \sigma>)$
if $\sigma(\alpha) = (\alpha_1, \alpha_2)$ then $((A_1, A_2), \overline{\sigma}_2)$
where $(A_1, \overline{\sigma}_1) = \mathcal{K}(\alpha_1, \sigma)$ and $(A_2, \overline{\sigma}_2) = \mathcal{K}(\alpha_2, \overline{\sigma}_1)$

4.4.3. Concurrent evaluation of arguments of standard functions

We now accomodate the operational semantics to support concurrent evaluation of arguments of standard functions. Some minor modifications are sufficient.

Definition 4.30

The class of operational expressions ($\tilde{E} \in$) $\mathrm{Exp}(\tilde{E}_2)$ for concurrent evaluation of arguments of standard functions is defined by :

$$\tilde{E} ::= E \mid \tilde{E}\ E \mid \tilde{E} \Rightarrow \alpha \mid \beta \Rightarrow \alpha \mid \delta \mid [\mathrm{p}]\mathrm{arg}_1 \ldots \mathrm{arg}_n$$

where $E \in \mathrm{Exp}$, $\alpha \in \mathrm{Loc}$, $\beta \in \mathrm{Bv}$, $\delta \in \mathrm{Ev}' \backslash \mathrm{Bv}$ and p is a primitive function with arity n; arg_i is given by:

$$\mathrm{arg}_i ::= \alpha \mid \tilde{E} \Rightarrow \alpha \mid \epsilon|_\alpha$$

where $\alpha \in$ Loc and $\epsilon \in$ Ev'

The operational expression $[\mathrm{p}]\mathrm{arg}_1 \ldots \mathrm{arg}_n$ is now used to represent both concurrent evaluation of the arguments of standard functions and sequential evaluation of the arguments of other primitive functions. The latter is established by moving the conditions of arg_i in section 4.4.2 inside the transition system T_2.

Definition 4.31

A configuration $c \in \mathrm{Conf}_2$ is defined by:

$$c ::= < \tilde{E}, \varrho, \sigma > \mid < \beta, \sigma >$$

where $\tilde{E} \in \mathrm{Exp}(\tilde{E}_2)$, $\varrho \in$ Fit($\tilde{E}$), $\sigma \in$ Store', $\beta \in$ Bv.

Remark

The transitions have the form:

$$< \tilde{E}, \varrho, \sigma > \rightarrow < \beta, \sigma' >$$

or

$$< \tilde{E}, \varrho, \sigma > \rightarrow < \tilde{E}', \varrho', \sigma' >$$

Definition 4.32

The transition system T_2 is defined by the axioms 1-15, 20-28 of transition system T_1 and the following axioms and rules:

Axiom 16 (*)

$$< [\mathrm{p}]\ \mathrm{arg}_1 \ldots \mathrm{arg}_{i-1}\ \alpha_i\ \mathrm{arg}_{i+1} \ldots \mathrm{arg}_n, \varrho, \sigma > \rightarrow < [\mathrm{p}]\ \mathrm{arg}_1 \ldots \mathrm{arg}_{i-1}\ (E_i \Rightarrow \alpha_i)\ \mathrm{arg}_{i+1} \ldots \mathrm{arg}_n, \varrho', \sigma' >$$

provided that
(i) p is a primitive function with arity n, cons excluded
(ii) in case p ≡ seq or p ≡ val then: ($i = 2 \Rightarrow \mathrm{arg}_1 = \epsilon_1|_{\alpha_1}$ for $\epsilon_1 \in$ Ev' and $\alpha_1 \in$ Loc
(iii) if p ≡ if then $i = 1$
(iv) $\sigma(\alpha_i) = (\varrho'_i, E_i)$
(v) ϱ' satisfies:
(a) for $k \neq n - i + 1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n - i + 1)(\varrho') = \varrho'_i$
(vi) $\sigma' = \sigma\{\ \mathrm{NotReady}/\alpha_i\ \}$

Axiom 17 (*)

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \beta_i|_{\alpha_i}\ arg_{i+1} \ldots arg_n, \varrho, \sigma>$$

provided that
(i) p is a primitive function with arity n, cons excluded
(ii) in case p≡seq or p≡val then: ($i=2 \Rightarrow arg_1 = \epsilon_1|_{\alpha_1}$ for $\epsilon_1 \in Ev'$ and $\alpha_1 \in Loc$
(iii) if p≡if then $i=1$
(iv) $\sigma(\alpha_i) = \beta_i$

Axiom 18 (*)

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \delta_i|_{\alpha_i}\ arg_{i+1} \ldots arg_n, \varrho', \sigma>$$

provided that
(i) p is one of the primitive functions seq, val, head or Y
(ii) in case p≡seq or p≡val then: ($i=2 \Rightarrow arg_1 = \epsilon_1|_{\alpha_1}$ for $\epsilon_1 \in Ev'$ and $\alpha_1 \in Loc$
(iii) n is the arity of p
(iv) $\sigma(\alpha_i) = (\delta_i, \varrho'_i)$
(v) ϱ' satisfies:
(a) for $k \neq n-i+1$ $Sel(k)(\varrho') = Sel(k)(\varrho)$
(b) $Sel(n-i+1)(\varrho') = \varrho'_i$

Axiom 19

$$< [p]\ arg_1 \ldots arg_{i-1}\ (\epsilon \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \epsilon|_{\alpha}\ arg_{i+1} \ldots arg_n, \varrho, \sigma'>$$

provided that
(i) p is a primitive function with arity n
(ii) if $\epsilon \in Bv$ then $\sigma' = \sigma\{\epsilon/\alpha\}$
(iii) if $\epsilon \in Ev' \setminus Bv$ then $\sigma' = \sigma\{(\epsilon, \varrho_i)/\alpha\}$

Rule 1

$$\frac{< \tilde{E}, \tilde{\varrho}, \sigma> \rightarrow < \tilde{E}', \tilde{\varrho}', \sigma'>}{< \tilde{E}\ E'', (\tilde{\varrho}, \varrho''), \sigma> \rightarrow < \tilde{E}'\ E'', (\tilde{\varrho}', \varrho''), \sigma'>}$$

$$< \tilde{E} \Rightarrow \alpha, \tilde{\varrho}, \sigma> \rightarrow < \tilde{E}' \Rightarrow \alpha, \tilde{\varrho}', \sigma'>$$

$$< [p]\ arg_1 \ldots arg_{i-1}\ (\tilde{E} \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ (\tilde{E}' \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho', \sigma'>$$

provided that in the third conclusion
(i) p is a primitive function with arity n
(ii) $Sel(n-i+1)(\varrho) = \tilde{\varrho}$
(iii) ϱ' satisfies:
(a) for $k \neq n-i+1$ $Sel(k)(\varrho') = Sel(k)(\varrho)$
(b) $Sel(n-i+1)(\varrho') = \tilde{\varrho}'$

Rule 2

$$\frac{< \tilde{E}, \tilde{\varrho}, \sigma> \rightarrow < \beta, \sigma'>}{< \tilde{E} \Rightarrow \alpha, \tilde{\varrho}, \sigma> \rightarrow < \beta \Rightarrow \alpha, r_0, \sigma'>}$$

$$< [p]\ arg_1 \ldots arg_{i-1}\ (\tilde{E} \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho, \sigma> \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ (\beta \Rightarrow \alpha)\ arg_{i+1} \ldots arg_n, \varrho', \sigma'>$$

provided that in the second conclusion
(i) p is a primitive function with arity n
(ii) $\mathrm{Sel}(n-i+1)(\varrho) = \tilde{\varrho}$
(iii) ϱ' satisfies:
(a) for $k \neq n-i+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n-i+1)(\varrho') = \mathbf{r}_0$

Note that, although the 'copied' axioms are literally identical, the configurations may be different.

We now define the operational meaning of an expression.

Definition 4.33

a. The mapping O_2: Exp $\rightarrow$ (Store' $\rightarrow$ $\mathcal{P}$(Ans $\times$ Store')) is defined by:

$O_2(E)(\sigma) = \overline{O}_2(< E, \varrho, \sigma>)$

where ϱ is the fitting arid operational environment of E

b. The mapping $\overline{O}_2$: $\mathrm{Conf}_2 \rightarrow \mathcal{P}$(Ans $\times$ Store') is defined by:

$\overline{O}_2(<\beta, \sigma>) = (\beta, \sigma)$

$\overline{O}_2(< E, \varrho, \sigma>)$ is given by:

if there is a finite transition sequence $< E, \varrho, \sigma> = \mathrm{c}_0 \rightarrow \mathrm{c}_1 \rightarrow \ldots \rightarrow \mathrm{c}_n \rightarrow /$ then

in case $\mathrm{c}_n = <\beta, \sigma'>$ then

$(\beta, \sigma') \in \overline{O}_2(\mathrm{c}_0)$

in case $\mathrm{c}_n = <\phi, \varrho', \sigma'>$ then

(error, σ') $\in \overline{O}_2(\mathrm{c}_0)$

in case $\mathrm{c}_n = <(\alpha_1, \alpha_2), (\mathbf{r}_0, \mathbf{r}_0), \sigma'>$ then

$((\mathrm{A}_1, \mathrm{A}_2), \sigma_2) \in \overline{O}_2(\mathrm{c}_0)$

where $(\mathrm{A}_1, \sigma_1) \in \mathcal{K}_2(\alpha_1, \sigma)$ and $(\mathrm{A}_2, \sigma_2) \in \mathcal{K}_2(\alpha_2, \sigma_1)$

otherwise: $\mathrm{c}_n = < \tilde{E}', \varrho', \sigma'>$

(suspend, σ') $\in \overline{O}_2(\mathrm{c}_0)$

else if there is an infinite transition sequence $<E, \varrho, \sigma> = \mathrm{c}_0 \rightarrow \mathrm{c}_1 \rightarrow \ldots \rightarrow \mathrm{c}_n \rightarrow \ldots$ then

(error, σ) $\in \overline{O}_2(\mathrm{c}_0)$

c. The mapping $\mathcal{K}_2$: Loc $\times$ Store' $\rightarrow$ $\mathcal{P}$(Ans $\times$ Store') is defined by:

$\mathcal{K}_2(\alpha, \sigma) =$

if $\sigma(\alpha)$ = unset, NotReady, free then (suspend, σ)
if $\sigma(\alpha) = \beta$ then (β, σ)
if $\sigma(\alpha) = (\phi, \varrho)$ then (error, σ)
if $\sigma(\alpha) = (\varrho, E)$ then $\overline{O}_2(< E, \varrho, \sigma>)$
if $\sigma(\alpha) = (\alpha_1, \alpha_2)$ then $((A_1, A_2), \overline{\sigma}_2)$
where $(A_1, \overline{\sigma}_1) \in \mathcal{K}_2(\alpha_1, \sigma)$ and $(A_2, \overline{\sigma}_2) \in \mathcal{K}_2(\alpha_2, \overline{\sigma}_1)$

4.4.4. The par-combinator

We now extend the language with the primitive function par. Par is a function of two arguments E_1 and E_2. par E_1 E_2 stands for the concurrent reduction of E_1 E_2 and E_2. The value of E_1 E_2 is returned. We now have a specific control annotation for parallel evaluation. Therefore the arguments of other primitive functions are evaluated sequentially. Thus add E_1 E_2 is sequential addition and (par (par add E_1) E_2) is parallel addition.

We return to the transition system of sequential evaluation and add some axioms to support the use of the par-combinator. First the definition of operational expression must be adjusted.

Definition 4.34

The class of operational expressions ($\tilde{E} \in$) $\mathrm{Exp}(\tilde{E}_3)$ for sequential evaluation combined with concurrent evaluation of the arguments of par is defined by :

$\tilde{E} ::= E \mid \tilde{E}\ E \mid \tilde{E} \Rightarrow \alpha \mid \beta \Rightarrow \alpha \mid \delta \mid [\mathrm{p}]\mathrm{arg}_1 \ldots \mathrm{arg}_n$

where $E \in \mathrm{Exp}$, $\alpha \in \mathrm{Loc}$, $\beta \in \mathrm{Bv}$, $\delta \in \mathrm{Ev}' \backslash \mathrm{Bv}$ and p is a primitive function with arity n;
arg_i $(1 \leqq i \leqq n)$ satisfies:

if p $\not\equiv$ par then

(i) $\mathrm{arg}_i = \epsilon|_\alpha$ with $\epsilon \in$ Ev′ and $\alpha \in$ Loc only if $i = 1$ or ($i > 1$ and arg_{i-1} satisfies (i))

or (ii) $\mathrm{arg}_i = E' \Rightarrow \alpha$ with $E' \in \mathrm{Exp}(\tilde{E}_1)$ and $\alpha \in$ Loc only if $i = 1$ or ($i > 1$ and arg_{i-1} satisfies (i))

or (iii) $\mathrm{arg}_i \in$ Loc.

if p $\equiv$ par then

$\mathrm{arg}_i ::= \alpha \mid \epsilon|_\alpha \mid \tilde{E} \Rightarrow \alpha$

where $\alpha \in$ Loc and $\epsilon \in$ Ev′

Note that Exp now includes the primitive par.

Definition 4.35

A configuration $c \in \mathrm{Conf}_3$ is defined by:

$c ::= < \tilde{E}, \varrho, \sigma > \mid < \beta, \sigma >$

where $\tilde{E} \in \mathrm{Exp}(\tilde{E}_3)$, $\varrho \in \mathrm{Fit}(\tilde{E})$, $\sigma \in$ Store′, $\beta \in$ Bv.

Remark

The transitions have the form:

$< \tilde{E}, \varrho, \sigma > \rightarrow < \beta, \sigma' >$

or

$< \tilde{E}, \varrho, \sigma > \rightarrow < \tilde{E}', \varrho', \sigma' >$

Definition 4.36

The transition system T_3 is defined by the axioms 1-15, 19-28 and rules 1 and 2 of transition system T_1 and the following axioms:

Axiom 16 (*)

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma > \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ (E_i \Rightarrow \alpha_i)\ arg_{i+1} \ldots arg_n, \varrho', \sigma' >$$

provided that
(i) p is a primitive function with arity n, cons and par excluded
(ii) if p ≡ if then $i = 1$
(iii) $\sigma(\alpha_i) = (\varrho'_i, E_i)$
(iv) ϱ' satisfies:
(a) for $k \neq n-i+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n-i+1)(\varrho') = \varrho'_i$
(v) $\sigma' = \sigma\{\mathrm{NotReady}/\alpha_i\}$

Axiom 17 (*)

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma > \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \beta_i|_{\alpha_i}\ arg_{i+1} \ldots arg_n, \varrho, \sigma >$$

provided that
(i) p is a primitive function with arity n, cons and par excluded
(ii) if p ≡ if then $i = 1$
(iii) $\sigma(\alpha_i) = \beta_i$

Axiom 18

$$< [p]\ arg_1 \ldots arg_{i-1}\ \alpha_i\ arg_{i+1} \ldots arg_n, \varrho, \sigma > \rightarrow < [p]\ arg_1 \ldots arg_{i-1}\ \delta_i|_{\alpha_i}\ arg_{i+1} \ldots arg_n, \varrho', \sigma >$$

provided that
(i) p is one of the primitive functions seq, val, head or Y
(ii) n is the arity of p
(iii) $\sigma(\alpha_i) = (\delta_i, \varrho'_i)$
(iv) ϱ' satisfies:
(a) for $k \neq n-i+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n-i+1)(\varrho') = \varrho'_i$

Axiom 29 (*)

$$< [par]\ arg_1\ \alpha_2, \varrho, \sigma > \rightarrow < [par]\ arg_1\ \epsilon_2|_{\alpha_2}, \varrho', \sigma >_1 \ |\ < [par]\ arg_1\ (E_2 \Rightarrow \alpha_2), \varrho'', \sigma'' >_2$$

provided that
(i) in case 1
(a) if $\epsilon_2 \in$ Bv then $\sigma(\alpha_2) = \epsilon_2$; $\varrho' = \varrho$
(b) if $\epsilon_2 \in$ Ev'\Bv then $\sigma(\alpha_2) = (\epsilon_2, \varrho_2')$; $\varrho' = ((\varrho_0, \varrho_1), \varrho_2')$
(ii) in case 2
(a) $\sigma(\alpha_2) = (\overline{\varrho}_2, E_2)$
(b) $\varrho'' = ((\varrho_0, \varrho_1), \overline{\varrho}_2)$
(c) $\sigma'' = \sigma\{\mathrm{NotReady}/\alpha_2\}$

Axiom 30 (*)

$$< [par]\ \alpha_1\ arg_2, \varrho, \sigma > \rightarrow < [par]\ (\delta_1\ x) \Rightarrow \alpha_3\ arg_2, \varrho', \sigma' >_1 \ |\ < [par]\ (E_1 \Rightarrow \alpha_1\ x) \Rightarrow \alpha_3\ arg_2, \varrho'', \sigma'' >_2$$

provided that
(i) $arg_2 ::= \alpha_2 \ |\ \tilde{E}_2 \Rightarrow \alpha_2 \ |\ \epsilon_2|_{\alpha_2}$
(ii) in case 1

(a) $\sigma(\alpha_1) = (\delta_1, \varrho_1')$
(b) $\varrho' = ((\varrho_0, (\varrho_1', \hat{\varrho})), \varrho_2)$
(c) $\hat{\varrho}$ satisfies (i) $\hat{\varrho} \in \text{Ide} \rightarrow \text{Loc}$ and (ii) $\hat{\varrho}(x) = \alpha_2$
(d) $\sigma' = \sigma\{\text{NotReady}/\alpha_3\}$
(iii) in case 2
(a) $\sigma(\alpha_1) = (\overline{\varrho}_1, E_1)$
(b) $\varrho'' = ((\varrho_0, (\overline{\varrho}_1, \tilde{\varrho})), \varrho_2)$
(c) $\tilde{\varrho}$ satisfies (i) $\tilde{\varrho} \in \text{Ide} \rightarrow \text{Loc}$ and (ii) $\tilde{\varrho}(x) = \alpha_2$
(d) $\sigma'' = \sigma\{\text{NotReady}/\alpha_1, \alpha_3\}$
(iv) $\alpha_3 = \text{new}(\sigma)$

Note that we use x here as an auxiliary variable; its environment is $\hat{\varrho}$ resp. $\tilde{\varrho}$.

Axiom 31

$$< [\text{par}]\ \epsilon_1|_{\alpha_1}\ \epsilon_2|_{\alpha_2}, \varrho, \sigma> \rightarrow < \epsilon_1, \sigma>_1 \mid < \epsilon_1, \varrho_1, \sigma>_2$$

provided that
(i) in case 1: $\epsilon_1 \in \text{Bv}$
(ii) in case 2: $\epsilon_2 \in \text{Ev}'\backslash\text{Bv}$

We now give the operational meaning of an expression.

Definition 4.37

a. The mapping O_3: Exp $\rightarrow$ (Store' $\rightarrow$ $\mathcal{P}$(Ans $\times$ Store')) is defined by:

$O_3(E)(\sigma) = \overline{O}_3(< E, \varrho, \sigma>)$

where ϱ is the fitting arid operational environment of E

b. The mapping $\overline{O}_3$: $\text{Conf}_3 \rightarrow \mathcal{P}$(Ans $\times$ Store') is defined by:

$\overline{O}_3(< \beta, \sigma>) = (\beta, \sigma)$

$\overline{O}_3(< E, \varrho, \sigma>)$ is given by

if there is a finite transition sequence $< E, \varrho, \sigma> = c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_n \rightarrow /$ then

in case $c_n = < \beta, \sigma'>$ then

$(\beta, \sigma') \in \overline{O}_3(c_0)$

in case $c_n = < \phi, \varrho', \sigma'>$ then

$(\text{error}, \sigma') \in \overline{O}_3(c_0)$

in case $c_n = < (\alpha_1, \alpha_2), (r_0, r_0), \sigma'>$ then

$((A_1, A_2), \sigma_2) \in \overline{O}_3(c_0)$

where $(A_1, \sigma_1) \in \mathcal{C}_3(\alpha_1, \sigma)$ and $(A_2, \sigma_2) \in \mathcal{C}_3(\alpha_2, \sigma_1)$

otherwise: $c_n = < \tilde{E}', \varrho', \sigma'>$

$(\text{suspend}, \sigma') \in \overline{O}_3(c_0)$

else if there is an infinite transition sequence $<E, \varrho, \sigma> = c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_n \rightarrow \ldots$ then

$(\text{error}, \sigma) \in \overline{O}_3(c_0)$

c. The mapping $\mathfrak{IC}_3$: Loc × Store′ → $\mathcal{P}$(Ans × Store′) is defined by:

$\mathfrak{IC}_3(\alpha, \sigma)$ =

if $\sigma(\alpha)$ = unset, NotReady, free then (suspend, σ)
if $\sigma(\alpha) = \beta$ then (β, σ)
if $\sigma(\alpha) = (\phi, \varrho)$ then (error, σ)
if $\sigma(\alpha) = (\varrho, E)$ then $O_3(\leq E, \varrho, \sigma>)$
if $\sigma(\alpha) = (\alpha_1, \alpha_2)$ then $((A_1, A_2), \bar{\sigma}_2)$
where $(A_1, \bar{\sigma}_1) \in \mathfrak{IC}_3(\alpha_1, \sigma)$ and $(A_2, \bar{\sigma}_2) \in \mathfrak{IC}_3(\alpha_2, \bar{\sigma}_1)$

4.4.5. Examples

Example 1

A transition sequence in T_1. We consider the expression

add (assign $x=2$; 3) (assign $y=3$; x)

Suppose that $\varrho_i(x)=\alpha_3$ and $\varrho_i(y)=\alpha_4$ for $i=0,\ldots,6$; $\sigma(\alpha_3)=\sigma(\alpha_4)=$unset and $\sigma(\alpha_1)=\sigma(\alpha_2)=$free.

< add (assign $x=2$; 3) (assign $y=3$; x), $((\varrho_0, ((\varrho_1, \varrho_2), \varrho_3)), ((\varrho_4, \varrho_5), \varrho_6)), \sigma$> →

< [add] E_1 E_2, $((r_0, \tilde{\varrho}_1), \tilde{\varrho}_2), \sigma$> →

< [add] α_1 E_2, $((r_0, r_0), \tilde{\varrho}_2), \sigma\{(\tilde{\varrho}_1, E_1)/\alpha_1\}$> →

< [add] α_1 α_2, $((r_0, r_0), r_0), \sigma_1\{(\tilde{\varrho}_2, E_2)/\alpha_2\}$> →

< [add] $(E_1 \Rightarrow \alpha_1)$ α_2, $((r_0, \tilde{\varrho}_1), r_0), \sigma_2\{$ NotReady/$\alpha_1\}$> →

< [add] $(3 \Rightarrow \alpha_1)$ α_2, $((r_0, \varrho_3), r_0), \sigma_3\{(\varrho_2, 2)/\alpha_3\}$> →

< [add] $(\underline{3} \Rightarrow \alpha_1)$ α_2, $((r_0, r_0), r_0), \sigma_4$> →

< [add] $\underline{3}$ α_2, $((r_0, r_0), r_0), \sigma_4\{\underline{3}/\alpha_1\}$> →

< [add] $\underline{3}$ $(E_2 \Rightarrow \alpha_2)$, $((r_0, r_0), \tilde{\varrho}_2), \sigma_5\{$ NotReady/$\alpha_2\}$> →

< [add] $\underline{3}$ $(x \Rightarrow \alpha_2)$, $((r_0, r_0), \varrho_6), \sigma_6\{(\varrho_5, 3)/\alpha_4\}$> →

< [add] $\underline{3}$ $(2 \Rightarrow \alpha_3 \Rightarrow \alpha_2)$, $((r_0, r_0), \varrho_2), \sigma_7\{$ NotReady/$\alpha_3\}$> →

< [add] $\underline{3}$ $(\underline{2} \Rightarrow \alpha_3 \Rightarrow \alpha_2)$, $((r_0, r_0), r_0), \sigma_8$> →

< [add] $\underline{3}$ $(\underline{2} \Rightarrow \alpha_2)$, $((r_0, r_0), r_0), \sigma_8\{\underline{2}/\alpha_3\}$> →

< [add] $\underline{3}$ $\underline{2}$, $((r_0, r_0), r_0), \sigma_9\{\underline{2}/\alpha_2\}$> →

< $\underline{5}$, σ_{10}>

Example 2

The denotational meaning of the expression of example 1.

We assume that $\varrho(x)=\alpha_3$, $\varrho(y)=\alpha_4$, $\sigma(\alpha_3)=\sigma(\alpha_4)=$unset and $\sigma(\alpha_1)=\alpha_2)=$free.

$\mathcal{E}$(add (assign $x=2$; 3) (assign $y=3$; x))(ϱ)(print)(σ) =

$\mathcal{E}$(add E_1)(ϱ)($\mathcal{A}(E_2)(\varrho)$(print))($\sigma$) =

$\&(\text{add})(\varrho)(A(E_1)(\varrho)(A(E_2)(\varrho)(\text{print})))(\sigma) =$

$x_1(\text{P}(\text{add}))(\sigma) =$

$\text{P}(\text{add})(\alpha_1)(A(E_2)(\varrho)(\text{print}))(\sigma\{\&(E_1)(\varrho)/\alpha_1\}) =$

$A(E_2)(\varrho)(\text{print})(\mathit{add}\ \alpha_1)(\sigma_1) =$

$(\mathit{add}\ \alpha_1)(\alpha_2)(\text{print})(\sigma_1\{\&(E_2)(\varrho)/\alpha_2\}) =$

$\text{force}(\alpha_1)(\lambda\epsilon_1.\ \epsilon_1 \in \text{Num} \rightarrow \theta_1, \text{wrong})(\sigma_2) =$

$\&(E_1)(\varrho)(\lambda\epsilon\sigma'.x_2(\epsilon)(\sigma'\{\epsilon/\alpha_1\}))(\sigma_2\{\text{NotReady}/\alpha_1\}) =$

$\&(3)(\varrho)(x_3)(\sigma_3\{\&(2)(\varrho)/\alpha_3\}) =$

$x_3(\underline{3})(\sigma_4) =$

$x_2(\underline{3})(\sigma_4\{\underline{3}/\alpha_1\}) =$

$\text{force}(\alpha_2)(\lambda\epsilon_2.\ \epsilon_2 \in \text{Num} \rightarrow \text{print}(\underline{3}+\epsilon_2), \mathit{wrong})(\sigma_5) =$

$\&(E_2)(\varrho)(\lambda\epsilon\sigma'.x_4(\epsilon)(\sigma'\{\epsilon/\alpha_2\}))(\sigma_5\{\text{NotReady}/\alpha_2\}) =$

$\&(x)(\varrho)(x_5)(\sigma_6\{\&(3)(\varrho)/\alpha_4\}) =$

$\text{force}(\alpha_3)(x_5)(\sigma_7) =$

$\&(2)(\varrho)(\lambda\epsilon\sigma'.x_5(\epsilon)(\sigma'\{\epsilon/\alpha_3\}))(\sigma_7\{\text{NotReady}/\alpha_3\}) =$

$x_6(\underline{2})(\sigma_8) =$

$x_5(\underline{2})(\sigma_8\{\underline{2}/\alpha_3\}) =$

$x_4(\underline{2})(\sigma_9\{\underline{2}/\alpha_2\}) =$

$\text{print}(\underline{3}+\underline{2})(\sigma_{10}) =$

$(\underline{5}, \sigma_{10})$

Example 3

A transition sequence in transition system T_2. We consider the expression:

add (assign $x=2$; y) (assign $y=3$; x)

This expression is very similar to the expression of the previous examples. However, sequential evaluation of this expression deadlocks. Concurrent evaluation of the arguments does not deadlock. Just one of the possible sequences is given.

Suppose that $\varrho_i(x)=\alpha_3$ and $\varrho_i(y)=\alpha_4$ for $i=0,\ldots,6$; $\sigma(\alpha_3)=\sigma(\alpha_4)=\text{unset}$ and $\sigma(\alpha_1)=\sigma(\alpha_2)=\text{free}$.

$< \text{add (assign } x=2; y) \text{ (assign } y=3; x), ((\varrho_0, ((\varrho_1, \varrho_2), \varrho_3)), ((\varrho_4, \varrho_5), \varrho_6)), \sigma > \rightarrow$

$< [\text{add}]\ E_1\ E_2, ((r_0, \tilde{\varrho}_1), \tilde{\varrho}_2), \sigma > \rightarrow$

$< [\text{add}]\ \alpha_1\ E_2, ((r_0, r_0), \tilde{\varrho}_2), \sigma\{(\tilde{\varrho}_1, E_1)/\alpha_1\} > \rightarrow$

$< [\text{add}]\ \alpha_1\ \alpha_2, ((r_0, r_0), r_0), \sigma_1\{(\tilde{\varrho}_2, E_2)/\alpha_2\} > \rightarrow$

$< [\text{add}]\ (E_1 \Rightarrow \alpha_1)\ \alpha_2, ((r_0, \tilde{\varrho}_1), r_0), \sigma_2\{\text{NotReady}/\alpha_1\} > \rightarrow$

$< [\text{add}]\ (y \Rightarrow \alpha_1)\ \alpha_2, ((r_0, \varrho_3), r_0), \sigma_3\{(\varrho_2, 2)/\alpha_3\} > \rightarrow$

$< [\text{add}]\ (y \Rightarrow \alpha_1)\ (E_2 \Rightarrow \alpha_2), ((r_0, \varrho_3), \tilde{\varrho}_2), \sigma_4\{\text{NotReady}/\alpha_2\} > \rightarrow$

$< [\text{add}]\ (y \Rightarrow \alpha_1)\ (x \Rightarrow \alpha_2), ((r_0, \varrho_3), \varrho_6), \sigma_5\{(\varrho_5, 3)/\alpha_4\} > \rightarrow$

$< [\text{add}]\ (3 \Rightarrow \alpha_4 \Rightarrow \alpha_1)\ (x \Rightarrow \alpha_2), ((r_0, \varrho_5), \varrho_6), \sigma_6\{\text{NotReady}/\alpha_4\} > \rightarrow$

$< [\text{add}]\ (3 \Rightarrow \alpha_4 \Rightarrow \alpha_1)\ (2 \Rightarrow \alpha_3 \Rightarrow \alpha_2), ((r_0, \varrho_5), \varrho_2), \sigma_7\{\text{NotReady}/\alpha_3\} > \rightarrow$

$< [\text{add}]\ (3 \Rightarrow \alpha_4 \Rightarrow \alpha_1)\ (\underline{2} \Rightarrow \alpha_3 \Rightarrow \alpha_2), ((r_0, \varrho_5), r_0), \sigma_8 > \rightarrow$

$< [\text{add}]\ (3 \Rightarrow \alpha_4 \Rightarrow \alpha_1)\ (\underline{2} \Rightarrow \alpha_2), ((r_0, \varrho_5), r_0), \sigma_8\{\underline{2}/\alpha_3\} > \rightarrow$

$< [\text{add}]\ (3 \Rightarrow \alpha_4 \Rightarrow \alpha_1)\ \underline{2}, ((r_0, \varrho_5), r_0), \sigma_9\{\underline{2}/\alpha_2\} > \rightarrow$

$< [\text{add}]\ (\underline{3} \Rightarrow \alpha_4 \Rightarrow \alpha_1)\ \underline{2}, ((r_0, r_0), r_0), \sigma_{10} > \rightarrow$

$< [\text{add}]\ (\underline{3} \Rightarrow \alpha_1)\ \underline{2}, ((r_0, r_0), r_0), \sigma_{10}\{\underline{3}/\alpha_4\} > \rightarrow$

$< [\text{add}]\ \underline{3}\ \underline{2}, ((r_0, r_0), r_0), \sigma_{11}\{\underline{3}/\alpha_1\} > \rightarrow$

$< \underline{5}, \sigma_{12} >$

Example 4

A transition sequence in T_3. We consider the expression:

(par (par z assign z=add; assign x=2; 1) assign y=3; 4)

Note that evaluation of the expression

((z assign z=add; assign x=2; 1) assign y=3; 4)

deadlocks.

Just one of the possible sequences is shown.

Suppose that $\varrho_i(x)=\alpha_3$, $\varrho_i(y)=\alpha_4$ and $\varrho_i(z)=\alpha_6$ for $i=0,\ldots,10$; $\sigma(\alpha_j)$=unset for j=3,4,6 and $\sigma(\alpha_k)$=free for j=1,2,5,7.

$<$ par (par z assign z=add; assign x=2; 1) (assign y=3; 4),

$((\varrho_0, ((\varrho_1, \varrho_2), ((\varrho_3, \varrho_4), ((\varrho_5, \varrho_6), \varrho_7)))), ((\varrho_8, \varrho_9), \varrho_{10})), \sigma > \rightarrow$

$< [\text{par}]\ E_1\ E_2, ((r_0, \tilde{\varrho}_1), \tilde{\varrho}_2), \sigma > \rightarrow$

$< [\text{par}]\ \alpha_1\ E_2, ((r_0, r_0), \tilde{\varrho}_2), \sigma\{(\tilde{\varrho}_1, E_1)/\alpha_1\} > \rightarrow$

$< [\text{par}]\ \alpha_1\ \alpha_2, ((r_0, r_0), r_0), \sigma_1\{(\tilde{\varrho}_2, E_2)/\alpha_2\} > \rightarrow$

$< [\text{par}]\ (E_1 \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ \alpha_2, ((r_0, (\tilde{\varrho}_1, \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_2\{\text{NotReady}/\alpha_1, \alpha_5\} > \rightarrow$

$< [\text{par}]\ (([\text{par}]\ E_3\ E_4) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ \alpha_2,$

$((r_0, (((r_0, \varrho_2), ((\varrho_3, \varrho_4), ((\varrho_5, \varrho_6), \varrho_7))), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_3 > \rightarrow$

$< [\text{par}]\ (([\text{par}]\ \alpha_6\ E_4) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ \alpha_2,$

$((r_0, (((r_0, r_0), ((\varrho_3, \varrho_4), ((\varrho_5, \varrho_6), \varrho_7))), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_3 > \rightarrow$

$< [\text{par}]\ (([\text{par}]\ \alpha_6\ \alpha_7) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ \alpha_2,$

$((r_0, (((r_0, r_0), r_0), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_3\{(\tilde{\varrho}_4, E_4)/\alpha_7\} > \rightarrow$

$< [\text{par}]\ (([\text{par}]\ \alpha_6\ (E_4 \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ \alpha_2,$

$((r_0, (((r_0, r_0), \tilde{\varrho}_4), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_3\{\text{NotReady}/\alpha_7\} > \rightarrow$

$< [\text{par}]\ (([\text{par}]\ \alpha_6\ ((\text{assign}\ x=2; 1) \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ \alpha_2,$

$((r_0, (((r_0, r_0), ((\varrho_5, \varrho_6), \varrho_7)), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_4\{(\varrho_4, \text{add})/\alpha_6\} > \rightarrow$

$< [\text{par}]\ (([\text{par}]\ \alpha_6\ (1 \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ \alpha_2,$

$((r_0, (((r_0, r_0), \varrho_7), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_5\{(\varrho_6, 2)/\alpha_3\} > \rightarrow$

$< [\text{par}]\ (([\text{par}]\ \alpha_6\ (1 \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (E_2 \Rightarrow \alpha_2),$

$((r_0, (((r_0, r_0), \varrho_7), \tilde{\varrho}\{\alpha_2/x\})), \tilde{\varrho}_2), \sigma_6\{\text{NotReady}/\alpha_2\} > \rightarrow$

$< [par]\ (([par]\ \alpha_6\ (1 \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, r_0), \varrho_7), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_7\{(\varrho_9, 3)/\alpha_4\}> \rightarrow$

$< [par]\ (([par]\ (add \Rightarrow \alpha_6\ x) \Rightarrow \alpha_8\ (1 \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, (\varrho_4, \hat{\varrho}\{\alpha_7/x\})), \varrho_7), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_8\{NotReady/\alpha_6, \alpha_8\}> \rightarrow$

$< [par]\ (([par]\ ([add] \Rightarrow \alpha_6\ x) \Rightarrow \alpha_8\ (\underline{1} \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, (r_0, \hat{\varrho}\{\alpha_7/x\})), \varrho_7), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_9> \rightarrow$

$< [par]\ (([par]\ ([add]\ x) \Rightarrow \alpha_8\ (1 \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, (r_0, \hat{\varrho}\{\alpha_7/x\})), \varrho_7), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_9\{([add], r_0)/\alpha_6\}> \rightarrow$

$< [par]\ (([par]\ ([add]\ x) \Rightarrow \alpha_8\ (\underline{1} \Rightarrow \alpha_7)) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, (r_0, \hat{\varrho}\{\alpha_7/x\})), r_0), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_{10}> \rightarrow$

$< [par]\ (([par]\ ([add]\ x) \Rightarrow \alpha_8\ \underline{1}\,|_{\alpha_7}) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, (r_0, \hat{\varrho}\{\alpha_7/x\})), r_0), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_{10}\{\underline{1}/\alpha_7\}> \rightarrow$

$< [par]\ (([par]\ ([add]\ \alpha_7) \Rightarrow \alpha_8\ \underline{1}\,|_{\alpha_7}) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, (r_0, r_0)), r_0), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_{11}> \rightarrow$

$< [par]\ (([par]\ ([add]\ \alpha_7)\,|_{\alpha_8}\ \underline{1}\,|_{\alpha_7}) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, (((r_0, (r_0, r_0)), r_0), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_{11}\{([add]\ \alpha_7, (r_0, r_0))/\alpha_8\}> \rightarrow$

$< [par]\ (([add]\ \alpha_7) \Rightarrow \alpha_1\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, ((r_0, r_0), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_{12}> \rightarrow$

$< [par]\ (([add]\ \alpha_7)\ x) \Rightarrow \alpha_5\ (4 \Rightarrow \alpha_2),$

$((r_0, ((r_0, r_0), \tilde{\varrho}\{\alpha_2/x\})), \varrho_{10}), \sigma_{12}\{([add]\ \alpha_7, (r_0, r_0))/\alpha_1\}> \rightarrow$

$< [par]\ (([add]\ \alpha_7)\ x) \Rightarrow \alpha_5\ (\underline{4} \Rightarrow \alpha_2), ((r_0, ((r_0, r_0), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_{13}> \rightarrow$

$< [par]\ (([add]\ \alpha_7)\ x) \Rightarrow \alpha_5\ \underline{4}\,|_{\alpha_2}, ((r_0, ((r_0, r_0), \tilde{\varrho}\{\alpha_2/x\})), r_0), \sigma_{13}\{\underline{4}/\alpha_2\}> \rightarrow$

$< [par]\ (([add]\ \alpha_7)\ \alpha_2) \Rightarrow \alpha_5\ \underline{4}\,|_{\alpha_2}, ((r_0, ((r_0, r_0), r_0)), r_0), \sigma_{14}> \rightarrow$

$< [par]\ (([add]\ \underline{1})\ \alpha_2) \Rightarrow \alpha_5\ \underline{4}\,|_{\alpha_2}, ((r_0, ((r_0, r_0), r_0)), r_0), \sigma_{14}> \rightarrow$

$< [par]\ (([add]\ \underline{1})\ \underline{4}) \Rightarrow \alpha_5\ \underline{4}\,|_{\alpha_2}, ((r_0, ((r_0, r_0), r_0)), r_0), \sigma_{14}> \rightarrow$

$< [par]\ (\underline{5} \Rightarrow \alpha_5)\ \underline{4}\,|_{\alpha_2}, ((r_0, r_0), r_0), \sigma_{14}> \rightarrow$

$< [par]\ \underline{5}\,|_{\alpha_5}\ \underline{4}\,|_{\alpha_2}, ((r_0, r_0), r_0), \sigma_{14}\{\underline{5}/\alpha_5\}> \rightarrow$

$< \underline{5}, \sigma_{15}>$

4.5. Natural semantics

In [Ka] natural semantics have been introduced. We briefly discuss some aspects of this semantic specification formalism.

A natural semantics is based on an inference system. Axioms and inference rules are provided to characterize various semantic predicates to be defined on an expression. For instance,

$$\varrho \vdash^{\circ} E \Rightarrow \alpha$$

expresses that expression E in environment ϱ evaluates to α. The turnstile is used, because a natural semantics is identified with a logic. On the left of the turnstile collections of assumptions on variables

occur, not arbitrary formulae.

A semantic definition is an unordered collection of rules. A rule has basically two parts, a numerator and a denominator. The numerator is again an unordered collection of formulae. Formulae are either sequents or conditions. The denominator of a rule is a sequent. Atomic conditions may be axiomatized in a separate set of rules. A sequent has an antecedent and a consequent separated by the turnstile. The consequent is a predicate.

A rule that contains no sequent on the numerator is an axiom. The various forms of sequents in a semantic definition are called judgements.

We now return to our language. First we define

Definition 4.38

The class of partially applied functions ($\phi \in$) Paf' is given by:

$\phi ::= \lambda x.E \mid pE_1 \ldots E_m$

where $E, E_1, \ldots, E_m \in$ Exp; p is a primitive function with arity n; $0 \leqq m < n$

Note that Paf' $\subset$ Exp. Furthermore we define

Definition 4.39

The class of expressed values ($\epsilon \in$) Ev'' is given by:

$\epsilon ::= \beta \mid (\alpha_1, \alpha_2) \mid \phi$

where $\beta \in$ Bv, $\alpha_1, \alpha_2 \in$ Loc and $\phi \in$ Paf'.

We denote by δ an element of Ev''\Bv. Thus:

$\delta ::= (\alpha_1, \alpha_2) \mid \lambda x.E \mid pE_1 \ldots E_m$

Definition 4.40

The class of stored values (s$\in$) Sv'' is given by:

s ::= β | (δ, ϱ_1) | (ϱ_2, E) | unset | NotReady | free

where $\beta \in$ Bv, $\delta \in$ Ev''\Bv, $\varrho_1 \in$ Fit(δ), (ϱ_2, E) $\in$ Clo'.

The class of stores ($\sigma \in$) Store'' is given by:

Store'' = Loc $\rightarrow$ Sv''

We use the following judgements:

1) $\varrho, \sigma \vdash E \Rightarrow (\beta, \sigma')$

where $\varrho \in$ Fit(E), $\sigma, \sigma' \in$ Store'', $E \in$ Exp and $\beta \in$ Bv,

2) $\varrho, \sigma \vdash E \Rightarrow (\delta, \varrho', \sigma')$

where $\varrho' \in$ Fit(δ) and $\delta \in$ Ev''\Bv.
We prefer to write

$$< E, \varrho, \sigma > \rightarrow < \beta, \sigma' > \mid < \delta, \varrho', \sigma' >$$

In our semantic definition we do not use conditions in the strict sense of the natural semantics of Kahn. Furthermore, formulae in the numerator of a rule have an implicit order.

We now define

Definition 4.41

The inference system is defined by the following axioms and rules:

Axiom 1

$$< \text{true}, \varrho, \sigma > \rightarrow < \underline{\text{true}}, \sigma >$$

provided that
(i) $\varrho \in \text{Ide} \rightarrow \text{Loc}$

Axiom 2

$$< \text{false}, \varrho, \sigma > \rightarrow < \underline{\text{false}}, \sigma >$$

provided that
(i) $\varrho \in \text{Ide} \rightarrow \text{Loc}$

Axiom 3

$$< m, \varrho, \sigma > \rightarrow < \underline{m}, \sigma >$$

provided that
(i) $\varrho \in \text{Ide} \rightarrow \text{Loc}$

Axiom 4

$$< \text{nil}, \varrho, \sigma > \rightarrow < \underline{\text{nil}}, \sigma >$$

provided that
(i) $\varrho \in \text{Ide} \rightarrow \text{Loc}$

Axiom 5

$$< \text{cons } E_1 \, E_2, \varrho, \sigma\{ \text{free}/\alpha_i \}_{i \in I} > \rightarrow < (\alpha_1, \alpha_2), (r_0, r_0), \sigma\{ (\varrho_i, E_i) /\alpha_i \}_{i \in I} >$$

provided that
(i) the index set I is equal to $\{ i \in \{1,2\} \mid E_i \notin \text{Ide} \}$
(ii) $\varrho \in \text{Fit}(\text{cons } E_1 \, E_2)$

Axiom 6

$$< \lambda x.E, \varrho, \sigma > \rightarrow < \lambda x.E, \varrho, \sigma >$$

provided that
(i) $\varrho \in \text{Fit}(E)$

Axiom 7

$$< pE_1 \ldots E_m, \varrho, \sigma > \rightarrow < pE_1 \ldots E_m, \varrho, \sigma >$$

provided that
(i) p is a primitive function with arity n
(ii) $0 \leqq m < n$
(iii) $\varrho \in \mathrm{Fit}(pE_1 \ldots E_m)$

Axiom 8

$$< x, \varrho, \sigma> \rightarrow < \beta, \sigma>_1 \mid < \delta, \bar{\varrho}, \sigma>_2$$

provided that
(i) $\varrho \in \mathrm{Ide} \rightarrow \mathrm{Loc}$ and $\varrho(x) = \alpha$
(ii) in case 1 : $\sigma(\alpha) = \beta$
(iii) in case 2 : $\sigma(\alpha) = (\delta, \bar{\varrho})$

Rule 1 (Var)

$$\frac{< ((E_0\, E_1)\ \cdots\ E_n), \varrho, \sigma\{\mathrm{unset}/\alpha\}> \rightarrow < \beta, \bar{\sigma}> \mid < \delta, \bar{\varrho}, \bar{\sigma}>}{< (((\mathrm{var}\, x\, ;\, E_0)\, E_1)\ \cdots\ E_n), \varrho', \sigma\{\mathrm{free}/\alpha\}> \rightarrow < \beta, \bar{\sigma}> \mid < \delta, \bar{\varrho}, \bar{\sigma}>}$$

provided that
(i) $n \geqq 0$
(ii) $\mathrm{Mod}(\varrho_0)(x)(\alpha) = \varrho_0$
(iii) ϱ' satisfies:
(a) for $k < n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) for $k > n$ $\mathrm{Mod}(\mathrm{Sel}(k)(\varrho'))(x)(\alpha) = \mathrm{Sel}(k)(\varrho)$

Rule 2 (Assign)

$$\frac{< ((E_0\, E_1)\ \cdots\ E_n), \varrho, \sigma\{(\tilde{\varrho}, E)/\alpha\}> \rightarrow < \beta, \bar{\sigma}> \mid < \delta, \bar{\varrho}, \bar{\sigma}>}{< (((\mathrm{assign}\, x = E\, ;\, E_0)\, E_1)\ \cdots\ E_n), \varrho', \sigma\{\mathrm{unset}/\alpha\}> \rightarrow < \beta, \bar{\sigma}> \mid < \delta, \bar{\varrho}, \bar{\sigma}>}$$

provided that
(i) $n \geqq 0$
(ii) $\mathrm{Mod}(\varrho_0)(x)(\alpha) = \varrho_0$ and $\mathrm{Mod}(\tilde{\varrho})(x)(\alpha) = \tilde{\varrho}$
(iii) ϱ' satisfies:
(a) for $k < n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') = \varrho_0$
(c) $\mathrm{Sel}(n+2)(\varrho') = \tilde{\varrho}$
(d) $\mathrm{Sel}(n+3)(\varrho') \in \mathrm{Ide} \rightarrow \mathrm{Loc}$ such that $\mathrm{Sel}(n+3)(\varrho')(x) = \alpha$
(e) for $k > n+3$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(n+3)(\varrho')$

Rule 3 (Lambda - argument no variable)

$$\frac{< ((E_0\, E_1)\ \cdots\ E_n), \varrho, \sigma\{(\tilde{\varrho}, E')/\alpha\}> \rightarrow < \beta, \bar{\sigma}> \mid < \delta, \bar{\varrho}, \bar{\sigma}>}{< (((\lambda x.E_0\, E')\, E_1)\ \cdots\ E_n), \varrho', \sigma\{\mathrm{free}/\alpha\}> \rightarrow < \beta, \bar{\sigma}> \mid < \delta, \bar{\varrho}, \bar{\sigma}>}$$

provided that
(i) $n \geqq 0$ and $E' \notin Ide$
(ii) $\mathrm{Mod}(\varrho_0)(x)(\alpha) = \varrho_0$
(iii) ϱ' satisfies:
(a) for $k < n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') = \tilde{\varrho}$
(c) for $k > n+1$ $\mathrm{Mod}(\mathrm{Sel}(k)(\varrho'))(x)(\alpha) = \mathrm{Sel}(k-1)(\varrho)$

Rule 4 (Lambda - argument a variable)

$$\frac{< ((E_0\ E_1)\ \cdots\ E_n),\ \varrho,\ \sigma> \rightarrow < \beta,\ \bar{\sigma}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}>}{< (((\lambda x.E_0\ y)\ E_1)\ \cdots\ E_n),\ \varrho',\ \sigma> \rightarrow < \beta,\ \bar{\sigma}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}>}$$

provided that
(i) $n \geqq 0$
(ii) $\mathrm{Mod}(\varrho_0)(x)(\alpha) = \varrho_0$
(iii) ϱ' satisfies:
(a) for $k<n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') \in \mathrm{Ide} \rightarrow \mathrm{Loc}$ such that $\mathrm{Sel}(n+1)(\varrho')(y) = \alpha$
(c) for $k>n+1$ $\mathrm{Mod}(\ \mathrm{Sel}(k)(\varrho')\ )(x)(\alpha) = \mathrm{Sel}(k-1)(\varrho)$

Rule 5 (Expressed value)

$$\frac{< ((\phi\ E_1)\ \cdots\ E_n),\ \varrho,\ \sigma\{\ (\phi,\ \varrho_0)/\alpha\}> \rightarrow < \beta,\ \bar{\sigma}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}>}{< ((x\ E_1)\ \cdots\ E_n),\ \varrho',\ \sigma\{\ (\phi,\ \varrho_0)/\alpha\}> \rightarrow < \beta,\ \bar{\sigma}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}>}$$

provided that
(i) $n>0$; (note that case n=0 is treated by Axiom 8)
(ii) ϱ' satisfies:
(a) for $k<n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') \in \mathrm{Ide} \rightarrow \mathrm{Loc}$ such that $\mathrm{Sel}(n+1)(\varrho')(x) = \alpha$
(c) for $k>n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(n+1)(\varrho')$

Rule 6 (Simple closure)

$$\frac{< E,\ \varrho,\ \sigma\{\ \mathrm{NotReady}/\alpha\}> \rightarrow < \beta,\ \bar{\sigma}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}>}{< x,\ \varrho',\ \sigma\{\ (\varrho,\ E)/\alpha\}> \rightarrow < \beta,\ \bar{\sigma}\{\ \beta/\alpha\}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}\{\ (\delta,\ \bar{\varrho})/\alpha\}>}$$

provided that
(i) $\varrho' \in \mathrm{Ide} \rightarrow \mathrm{Loc}$ such that $\varrho'(x)=\alpha$

Rule 7 (Closure)

$$\frac{\begin{array}{l}< E,\ \tilde{\varrho},\ \sigma_1\{\ \mathrm{NotReady}/\alpha\}> \rightarrow < \phi,\ \hat{\varrho},\ \sigma_2> \\ < ((\phi\ E_1)\ \cdots\ E_n),\ \varrho,\ \sigma_2\{\ (\phi,\ \hat{\varrho})/\alpha\}> \rightarrow < \beta,\ \bar{\sigma}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}>\end{array}}{< ((x\ E_1)\ \cdots\ E_n),\ \varrho',\ \sigma_1\{\ (\tilde{\varrho},\ E)/\alpha\}> \rightarrow < \beta,\ \bar{\sigma}> \mid < \delta,\ \bar{\varrho},\ \bar{\sigma}>}$$

provided that
(i) $n>0$; (note that case n=0 is treated by rule 6)
(ii) for $k>0$ $\mathrm{Sel}(n+k)(\varrho) = \mathrm{Sel}(k)(\hat{\varrho})$; in other words $\varrho_0 = \hat{\varrho}$
(iii) ϱ' satisfies:
(a) for $k<n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') \in \mathrm{Ide} \rightarrow \mathrm{Loc}$ such that $\mathrm{Sel}(n+1)(\varrho')(x) = \alpha$
(c) for $k>n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(n+1)(\varrho')$

Rule 8 (Standard functions)

$$< E_1,\ \varrho_1,\ \sigma_1\{\ \mathrm{NotReady}/\alpha_i\}_{i\in I} > \rightarrow < \beta_1,\ \bar{\sigma}_1>$$

$$< E_2,\ \varrho_2,\ \sigma_2> \rightarrow < \beta_2,\ \bar{\sigma}_2>$$

$$< E_i,\ \varrho_i,\ \sigma_i> \rightarrow < \beta_i,\ \bar{\sigma}_i> \qquad (2<i<n)$$

$$\frac{< E_n, \varrho_n, \sigma_n > \rightarrow < \beta_n, \bar{\sigma}_n >}{< pE_1 \ldots E_n, \varrho, \sigma_1\{ \text{free}/\alpha_i \}_{i \in I} > \rightarrow < \beta, \sigma_{n+1} >}$$

provided that
(i) p is a standard primitive function such as add, div or not with arity $n > 0$
(ii) the index set I is equal to $\{ i \in \{1,\ldots,n\} \mid E_i \notin \text{Ide} \}$
(iii) for i=1,...,n : if $i \in I$ then $\sigma_{i+1} = \bar{\sigma}_i\{ \beta_i/\alpha_i \}$ else $\sigma_{i+1} = \bar{\sigma}_i$ fi
(iv) $\beta = [p](\beta_1, \ldots, \beta_n)$
(v) $\varrho_0 \in \text{Ide} \rightarrow \text{Loc}$

Rule 9 (Seq)

$$\frac{\begin{array}{l} < \bar{E}_1, \varrho'_1, \sigma_1\{ \text{NotReady}/\alpha_i \}_{i \in I} > \rightarrow < \beta_1, \bar{\sigma}_1 >_{1A} \mid < \delta_1, \bar{\varrho}_1, \bar{\sigma}_1 >_{1B} \\ < \bar{E}_2, \varrho'_2, \sigma_2 > \rightarrow < \beta_2, \bar{\sigma}_2 >_{2A} \mid < \delta_2, \bar{\varrho}_2, \bar{\sigma}_2 >_{2B} \mid < \phi, \bar{\varrho}_2, \bar{\sigma}_2 >_{2C} \\ [< \phi E_1 \ldots E_n, \varrho, \sigma_3 > \rightarrow < \beta_3, \bar{\sigma}_3 > \mid < \delta_3, \bar{\varrho}_3, \bar{\sigma}_3 >]_{2C} \end{array}}{< \text{seq } \bar{E}_1 \bar{E}_2 E_1 \ldots E_n, \varrho', \sigma_1\{ \text{free}/\alpha_i \}_{i \in I} > \rightarrow < \beta_j, \sigma_4 > \mid < \delta_j, \bar{\varrho}_j, \sigma_4 >}$$

provided that
(i) the index set is equal to $\{ i \in \{1,2\} \mid \bar{E}_i \notin \text{Ide} \}$
(ii) if $1 \in I$ then
in case 1A : $\sigma_2 = \bar{\sigma}_1\{ \beta_1/\alpha_1 \}$
in case 1B : $\sigma_2 = \bar{\sigma}_1\{ (\delta_1, \bar{\varrho}_1)/\alpha_2 \}$
else $\sigma_2 = \bar{\sigma}_1$ fi
(iii) in case 2A,2B : $n=0$, $j=2$
if $2 \in I$ then
in case 2A : $\sigma_4 = \bar{\sigma}_2\{ \beta_2/\alpha_2 \}$
in case 2B : $\sigma_4 = \bar{\sigma}_2\{ (\delta_2, \bar{\varrho}_2)/\alpha_2 \}$
else $\sigma_4 = \bar{\sigma}_2$ fi
(iv) in case 2C : $n > 0$, $j=3$
if $2 \in I$ then $\sigma_3 = \bar{\sigma}_2\{ (\phi, \bar{\varrho}_2)/\alpha_2 \}$ else $\sigma_3 = \bar{\sigma}_2$ fi
$\sigma_4 = \bar{\sigma}_3$
for $k > 0$ $\text{Sel}(n+k)(\varrho) = \text{Sel}(k)(\bar{\varrho}_2)$, in other words $\varrho_0 = \bar{\varrho}_2$
(v) ϱ' satisfies:
(a),in case 2C : for $k < n+1$ $\text{Sel}(k)(\varrho') = \text{Sel}(k)(\varrho)$
(b) $\text{Sel}(n+1)(\varrho') = \varrho'_2$
(c) $\text{Sel}(n+2)(\varrho') = \varrho'_1$
(d) $\text{Sel}(n+3)(\varrho') \in \text{Ide} \rightarrow \text{Loc}$
(e) for $k > n+3$ $\text{Sel}(k)(\varrho') = \text{Sel}(n+3)(\varrho')$

Rule 10 (Val)

$$\frac{\begin{array}{l} < \bar{E}_1, \varrho'_1, \sigma_1\{ \text{NotReady}/\alpha_i \}_{i \in I} > \rightarrow < \epsilon_1, \bar{\sigma}_1 >_{1A} \mid < \epsilon_1, \bar{\varrho}_1, \bar{\sigma}_1 >_{1B} \\ < \bar{E}_2, \varrho'_2, \sigma_2 > \rightarrow < \phi, \bar{\varrho}_2, \bar{\sigma}_2 > \\ < \phi x E_1 \ldots E_n, \hat{\varrho}, \sigma_3 > \rightarrow < \beta_3, \bar{\sigma}_3 > \mid < \delta_3, \bar{\varrho}_3, \bar{\sigma}_3 > \end{array}}{< \text{val } \bar{E}_1 \bar{E}_2 E_1 \ldots E_n, \varrho', \sigma_1\{ \text{free}/\alpha_i \}_{i \in I} > \rightarrow < \beta_3, \bar{\sigma}_3 > \mid < \delta_3, \bar{\varrho}_3, \bar{\sigma}_3 >}$$

provided that
(i) $n \geqq 0$
(ii) the index set I is equal to $\{ i \in \{1,2\} \mid \bar{E}_i \notin \text{Ide} \}$
(iii) if $1 \in I$ then
in case 1A : $\sigma_2 = \bar{\sigma}_1\{ \epsilon_1/\alpha_1 \}$ with $\epsilon_1 \in \text{Bv}$
in case 1B : $\sigma_2 = \bar{\sigma}_1\{ (\epsilon_1, \bar{\varrho}_1)/\alpha_1 \}$ with $\epsilon_1 \in \text{Ev} \setminus \text{Bv}$
else $\sigma_2 = \bar{\sigma}_1$ fi
(iv) if $2 \in I$ then $\sigma_3 = \bar{\sigma}_2\{ (\phi, \bar{\varrho}_2)/\alpha_2 \}$ else $\sigma_3 = \bar{\sigma}_2$ fi

(v) if $l \notin I$ then $\alpha_l = \varrho'_1(\bar{E}_1)$
(vi) $Sel(n+1)(\hat{\varrho})(x) = \alpha_l$
(vii) for $k>0$ $Sel(n+k+1)(\hat{\varrho}) = Sel(k)(\bar{\varrho}_2)$
(vii) ϱ' satisfies:
(a) for $k<n+1$ $Sel(k)(\varrho') = Sel(k)(\hat{\varrho})$
(b) $Sel(n+1)(\varrho') = \varrho'_2$
(c) $Sel(n+2)(\varrho') = \varrho'_1$
(d) $Sel(n+3)(\varrho') \in Ide \rightarrow Loc$
(e) for $k>n+3$ $Sel(k)(\varrho') = Sel(n+3)(\varrho')$

Rule 11 (If)

$$\frac{\begin{array}{l} < \bar{E}_1, \varrho'_1, \sigma_1\{ \text{NotReady}/\alpha_i \}_{i\in I} > \rightarrow < \beta_1, \bar{\sigma}_1 > \\ < \bar{E}_l, \varrho'_l, \sigma_l > \rightarrow < \beta_l, \bar{\sigma}_l >_{2A} \mid < \delta_l, \bar{\varrho}_l, \bar{\sigma}_l >_{2B} \mid < \phi, \bar{\varrho}_l, \bar{\sigma}_l >_{2C} \\ \quad [< \phi E_1 \ldots E_n, \varrho, \sigma_4 > \rightarrow < \beta_4, \bar{\sigma}_4 > \mid < \delta_4, \bar{\varrho}_4, \bar{\sigma}_4 >]_{2C} \end{array}}{< \text{if } \bar{E}_1\bar{E}_2\bar{E}_3E_1 \ldots E_n, \varrho', \sigma_1\{ \text{free}/\alpha_i \}_{i\in I} > \rightarrow < \beta_j, \sigma_5 > \mid < \delta_j, \bar{\varrho}_j, \sigma_5 >}$$

provided that
(i) the index set is equal to $\{ i \in \{1,2,3\} \mid \bar{E}_i \notin Ide \}$
(ii) ($\beta_1 =$ true and $l = 2$) or ($\beta_1 =$ false and $l = 3$)
(iii) if $l \in I$ then
in case $\bar{E}_{5-l} \notin Ide$: $\sigma_l = \bar{\sigma}_1\{ \beta_1/\alpha_1, (\varrho'_{5-l}, \bar{E}_{5-l})/\alpha_{5-l} \}$
in case $\bar{E}_{5-l} \in Ide$: $\sigma_l = \bar{\sigma}_1\{ \beta_1/\alpha_1 \}$
else
in case $\bar{E}_{5-l} \notin Ide$: $\sigma_l = \bar{\sigma}_1\{ (\varrho'_{5-l}, \bar{E}_{5-l})/\alpha_{5-l} \}$
in case $\bar{E}_{5-l} \in Ide$: $\sigma_l = \bar{\sigma}_1$
fi
(iv) in case lA,lB : $n=0$, $j = l$
if $l \in I$ then
in case lA : $\sigma_5 = \bar{\sigma}_l\{ \beta_l/\alpha_l \}$
in case lB : $\sigma_5 = \bar{\sigma}_l\{ (\delta_l, \bar{\varrho}_l)/\alpha_l \}$
else $\sigma_5 = \bar{\sigma}_l$ fi
(v) in case lC : $n>0$, $j=4$
if $l \in I$ then $\sigma_4 = \bar{\sigma}_l\{ (\phi, \bar{\varrho}_l)/\alpha_l \}$ else $\sigma_4 = \bar{\sigma}_l$ fi
$\sigma_5 = \bar{\sigma}_4$
$\varrho_0 = \bar{\varrho}_l$
(vi) ϱ' satisfies:
(a) in case 2C : for $k<n+1$ $Sel(k)(\varrho') = Sel(k)(\varrho)$
(b) $Sel(n+1)(\varrho') = \varrho'_3$
(c) $Sel(n+2)(\varrho') = \varrho'_2$
(d) $Sel(n+3)(\varrho') = \varrho'_1$
(e) $Sel(n+4)(\varrho') \in Ide \rightarrow Loc$
(f) for $k>n+4$ $Sel(k)(\varrho') = Sel(n+4)(\varrho')$

Rule 12 (Head)

$$\frac{\begin{array}{l} < \bar{E}_1, \varrho'_1, \sigma_1\{ \text{NotReady}/\alpha_i \}_{i\in I} > \rightarrow < (\alpha, \alpha'), (r_0, r_0), \bar{\sigma}_1 > \\ \quad < xE_1 \ldots E_n, \varrho, \sigma_2 > \rightarrow < \beta_2, \bar{\sigma}_2 > \mid < \delta_2, \bar{\varrho}_2, \bar{\sigma}_2 > \end{array}}{< \text{head } \bar{E}_1E_1 \ldots E_n, \varrho', \sigma_1\{ \text{free}/\alpha_i \}_{i\in I} > \rightarrow < \beta_2, \bar{\sigma}_2 > \mid < \delta_2, \bar{\varrho}_2, \bar{\sigma}_2 >}$$

provided that
(i) the index set I is equal to $\{ i \in \{1\} \mid \bar{E}_i \notin Ide \}$
(ii) $n \geqq 0$
(iii) if $1 \in I$ then $\sigma_2 = \bar{\sigma}_1\{ (\alpha, \alpha')/\alpha_1 \}$ else $\sigma_2 = \bar{\sigma}_1$ fi

(iv) $\varrho_0 \in$ Ide $\rightarrow$ Loc such that $\varrho_0(x)=\alpha$
(v) ϱ' satisfies:
(a) for $k<n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') = \varrho'_1$
(c) $\mathrm{Sel}(n+2)(\varrho') \in$ Ide $\rightarrow$ Loc
(d) for $k>n+2$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(n+2)(\varrho')$

Rule 13A (Y-combinator)

$$\frac{\begin{array}{l} < \bar{E}_1, \varrho'_1, \sigma_1\{ \text{NotReady}/\alpha_i \}_{i\in I} > \rightarrow < \lambda x.E, \bar{\varrho}_1, \bar{\sigma}_1 > \\ < E, \varrho'_2, \sigma_2 > \rightarrow < \beta_2, \bar{\sigma}_2 >_{2A} \mid < \delta_2, \bar{\varrho}_2, \bar{\sigma}_2 >_{2B} \mid < \phi, \bar{\varrho}_2, \bar{\sigma}_2 >_{2C} \\ \quad [< \phi E_1 \ldots E_n, \varrho, \sigma_3 > \rightarrow < \beta_3, \bar{\sigma}_3 > \mid < \delta_3, \bar{\varrho}_3, \bar{\sigma}_3 >]_{2C} \end{array}}{< \mathrm{Y}\, \bar{E}_1 E_1 \ldots E_n, \varrho', \sigma_1\{ \text{free}/\alpha_i \}_{i\in I} > \rightarrow < \beta_j, \sigma_4 > \mid < \delta_j, \bar{\varrho}_j, \sigma_4 >}$$

provided that
(i) the index set is equal to $\{2\} \cup \{ i \in \{1\} \mid \bar{E}_i \notin \text{Ide} \}$
(ii) if $1 \in I$ then $\sigma_2 = \sigma_1\{ (\lambda x.E, \bar{\varrho}_1)/\alpha_1 \}$ else $\sigma_2 = \bar{\sigma}_1$ fi
(iii) $\varrho'_2 = \mathrm{Mod}(\bar{\varrho}_1)(x)(\alpha_2)$
(iv) in case 2A,2B : $n=0$, $j=2$
in case 2A : $\sigma_4 = \bar{\sigma}_2\{ \beta_2/\alpha_2 \}$
in case 2B : $\sigma_4 = \bar{\sigma}_2\{ (\delta_2, \bar{\varrho}_2)/\alpha_2 \}$
(v) in case 2C : $n>0$, $j=3$
$\sigma_3 = \bar{\sigma}_2\{ (\phi, \bar{\varrho}_2)/\alpha_2 \}$
$\sigma_4 = \bar{\sigma}_3$
$\varrho_0 = \bar{\varrho}_2$
(vi) ϱ' satisfies:
(a) in case 2C : for $k<n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') = \varrho'_1$
(c) $\mathrm{Sel}(n+2)(\varrho') \in$ Ide $\rightarrow$ Loc
(d) for $k>n+2$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(n+2)(\varrho')$

Rule 13B (Y-combinator)

$$\frac{\begin{array}{l} < \bar{E}_1, \varrho'_1, \sigma_1\{ \text{NotReady}/\alpha_i \}_{i\in I} > \rightarrow < \mathrm{p}E'_1 \ldots E'_m, \bar{\varrho}_1, \bar{\sigma}_1 > \\ < \mathrm{p}E'_1 \ldots E'_m x, \varrho'_2, \sigma_2 > \rightarrow < \beta_2, \bar{\sigma}_2 >_{2A} \mid < \delta_2, \bar{\varrho}_2, \bar{\sigma}_2 >_{2B} \mid < \phi, \bar{\varrho}_2, \bar{\sigma}_2 >_{2C} \\ \quad [< \phi E_1 \ldots E_n, \varrho, \sigma_3 > \rightarrow < \beta_3, \bar{\sigma}_3 > \mid < \delta_3, \bar{\varrho}_3, \bar{\sigma}_3 >]_{2C} \end{array}}{< \mathrm{Y}\, \bar{E}_1 E_1 \ldots E_n, \varrho', \sigma_1\{ \text{free}/\alpha_i \}_{i\in I} > \rightarrow < \beta_j, \sigma_4 > \mid < \delta_j, \bar{\varrho}_j, \sigma_4 >}$$

provided that
(i) the index set is equal to $\{2\} \cup \{ i \in \{1\} \mid \bar{E}_i \notin \text{Ide} \}$
(ii) if $1 \in I$ then $\sigma_2 = \sigma_1\{ (\mathrm{p}E'_1 \ldots E'_m, \bar{\varrho}_1)/\alpha_1 \}$ else $\sigma_2 = \bar{\sigma}_1$ fi
(iii) ϱ'_2 satisfies:
(a) $\mathrm{Sel}(1)(\varrho'_2) \in$ Ide $\rightarrow$ Loc such that $\mathrm{Sel}(1)(\varrho'_2)(x) = \alpha_2$
(b) for $k>1$ $\mathrm{Sel}(k)(\varrho'_2) = \mathrm{Sel}(k-1)(\bar{\varrho}_1)$
(iv) in case 2A,2B : $n=0$, $j=2$
in case 2A : $\sigma_4 = \bar{\sigma}_2\{ \beta_2/\alpha_2 \}$
in case 2B : $\sigma_4 = \bar{\sigma}_2\{ (\delta_2, \bar{\varrho}_2)/\alpha_2 \}$
(v) in case 2C : $n>0$, $j=3$
$\sigma_3 = \bar{\sigma}_2\{ (\phi, \bar{\varrho}_2)/\alpha_2 \}$
$\sigma_4 = \bar{\sigma}_3$
$\varrho_0 = \bar{\varrho}_2$
(vi) ϱ' satisfies:
(a) in case 2C : for $k<n+1$ $\mathrm{Sel}(k)(\varrho') = \mathrm{Sel}(k)(\varrho)$
(b) $\mathrm{Sel}(n+1)(\varrho') = \varrho'_1$

(c) $Sel(n+2)(\varrho') \in Ide \rightarrow Loc$
(d) for $k > n+2$ $Sel(k)(\varrho') = Sel(n+2)(\varrho')$

The meaning of an expression is defined in

Definition 4.42

a. The mapping N: Exp $\rightarrow$ (Store'' $\rightarrow$ Ans $\times$ Store'') is defined by:

$N(E)(\sigma) = \overline{N}(< E, \varrho, \sigma >)$

where ϱ is the fitting arid operational environment of E

b. The mapping $\overline{N}$: Exp $\times$ Env' $\times$ Store'' $\rightarrow$ Ans $\times$ Store'' is defined by:

$\overline{N}(< E, \varrho, \sigma >) =$

if $< E, \varrho, \sigma > \rightarrow < \beta, \sigma' >$ then

(β, σ')

if $< E, \varrho, \sigma > \rightarrow < \phi, \varrho', \sigma' >$ then

(error, σ')

if $< E, \varrho, \sigma > \rightarrow < (\alpha_1, \alpha_2), (\mathrm{r}_0, \mathrm{r}_0), \sigma' >$ then

$((A_1, A_2), \sigma_2)$

where $(A_1, \sigma_1) = \mathcal{K}'(\alpha_1, \sigma)$ and $(A_2, \sigma_2) = \mathcal{K}'(\alpha_2, \sigma_1)$

otherwise

(error, σ)

c. The mapping $\mathcal{K}'$: Loc $\times$ Store'' $\rightarrow$ Ans $\times$ Store'' is defined by:

$\mathcal{K}'(\alpha, \sigma) =$

if $\sigma(\alpha) =$ unset, NotReady, free then (suspend, σ)
if $\sigma(\alpha) = \beta$ then (β, σ)
if $\sigma(\alpha) = (\phi, \varrho)$ then (error, σ)
if $\sigma(\alpha) = (\varrho, E)$ then $\overline{N}(\leq E, \varrho, \sigma >)$
if $\sigma(\alpha) = (\alpha_1, \alpha_2)$ then $((A_1, A_2), \overline{\sigma}_2)$
where $(A_1, \overline{\sigma}_1) = \mathcal{K}'(\alpha_1, \sigma)$ and $(A_2, \overline{\sigma}_2) = \mathcal{K}'(\alpha_2, \overline{\sigma}_1)$

Example

The 'natural' meaning of the expression

var x; var y; (add (assign $x = 2$; 3) (assign $y = 3$; x)

Note that the expression resembles the expression of the examples 1 and 2 in section 4.4.5.

Say $\varrho = ((\mathrm{r}_0, ((\mathrm{r}_0, \mathrm{r}_0), \mathrm{r}_0)), ((\mathrm{r}_0, \mathrm{r}_0), \mathrm{r}_0))$

Assume that $\sigma(\alpha_i) =$ free for $i = 1, \ldots, 4$.

We give the inference:

1) $< 3, \varrho_4, \sigma_4 > \rightarrow < \underline{3}, \sigma_4 >$ (A1)

where $\varrho_4 = \text{Sel}(1)(\varrho_3)$
where $\sigma_4 = \sigma\{ \text{NotReady}/\alpha_1, \text{NotReady}/\alpha_2, (\text{Sel}(2)(\varrho_3), 2)/\alpha_3, \text{unset}/\alpha_4\}$

2) $<$ assign x =2; 3, $\varrho_3, \sigma_3 > \rightarrow < \underline{3}, \sigma_4 >$ (1, R2)

where $\varrho_3 = \text{Sel}(2)(\varrho_2)$
where $\sigma_3 = \sigma\{ \text{NotReady}/\alpha_1, \text{NotReady}/\alpha_2, \text{unset}/\alpha_3, \text{unset}/\alpha_4\}$

3) $< 2, \varrho_7, \sigma_7 > \rightarrow < \underline{2}, \sigma_7 >$ (A1)

where $\varrho_7 = \text{Sel}(2)(\varrho_3)$
where $\sigma_7 = \sigma\{ \underline{3}/\alpha_1, \text{NotReady}/\alpha_2, \text{NotReady}/\alpha_3, (\text{Sel}(2)(\varrho_5), 3)/\alpha_4\}$

4) $< x, \varrho_6, \sigma_6 > \rightarrow < \underline{2}, \sigma_8 >$ (3, R6)

where $\varrho_6 = \text{Sel}(1)(\varrho_5)$
where $\sigma_6 = \sigma\{ \underline{3}/\alpha_1, \text{NotReady}/\alpha_2, (\text{Sel}(2)(\varrho_3), 2)/\alpha_3, (\text{Sel}(2)(\varrho_5), 3)/\alpha_4\}$
where $\sigma_8 = \sigma\{ \underline{3}/\alpha_1, \text{NotReady}/\alpha_2, \underline{2}/\alpha_3, (\text{Sel}(2)(\varrho_5), 3)/\alpha_4\}$

5) $<$ assign y =3; x, $\varrho_5, \sigma_5 > \rightarrow < \underline{2}, \sigma_8 >$ (4, R2)

where $\varrho_5 = \text{Sel}(1)(\varrho_2)$
where $\sigma_5 = \sigma\{ \underline{3}/\alpha_1, \text{NotReady}/\alpha_2, (\text{Sel}(2)(\varrho_3), 2)/\alpha_3, \text{unset}/\alpha_4\}$

6) $<$ add E_1 E_2, ϱ_2 $\sigma_2 > \rightarrow <\underline{5}, \sigma_9 >$ (2, 5, R8)

where $\varrho_2 = \text{Mod}(\varrho_1)(y)(\alpha_4)$
where $\sigma_2 = \sigma\{ \text{free}/\alpha_1, \text{free}/\alpha_2, \text{unset}/\alpha_3, \text{unset}/\alpha_4\}$
where $\sigma_9 = \sigma\{ \underline{3}/\alpha_1, \underline{2}/\alpha_2, \underline{2}/\alpha_3, (\text{Sel}(2)(\varrho_5), 3)/\alpha_4\}$

7) $<$ var y; (add E_1 E_2), $\varrho_1, \sigma_1 > \rightarrow < \underline{5}, \sigma_9 >$ (6, R1)

where $\varrho_1 = \text{Mod}(\varrho)(x)(\alpha_3)$
where $\sigma_1 = \sigma\{ \text{free}/\alpha_1, \text{free}/\alpha_2, \text{unset}/\alpha_3, \text{free}/\alpha_4\}$

8) $<$ var x; var y; (add E_1 E_2), $\varrho, \sigma > \rightarrow < \underline{5}, \sigma_9 >$ (7, R1)

Thus: $N(E)(\sigma) = \bar{N}(< E, \varrho, \sigma >) = (\underline{5}, \sigma_9)$.

4.6. Comment

Eventually, the operational semantics has required more preparatory work than the natural semantics. This is obviously an advantage of the more elegant natural semantics. However, the examples show that the natural semantics has more theoretical value than practical value, unless a general strategy for execution is provided to solve the various kinds of 'equations' on sequents. We further refer to [Ka] for this subject.

There is obviously a correspondence between &, O_1 and N, but it has not been given in this paper. Further progress can be made in accomodating the natural and denotational semantics to parallel evaluation. Furtermore, it would be very nice, if semantics could be given based on processes as in chapter 3. This requires a solid mathematical framework. Once established, semantical equivalence proofs and the like are within reach.

We conclude this paper with a list of references in chapter 5.

5. References

[AB] P. America, J.W. de Bakker, Designing equivalent semantic models for process creation, in: Proc. Advanced School on Mathematical Models for the Semantics of Parallellism (M. Venturini Zilli, ed.), LNCS 280, Springer (1987) 21-80.

[ABKR1] P. America, J.W. de Bakker, J.N. Kok, J.J.M.M. Rutten, Operational semantics of a parallel object-oriented language, 13th ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 13-15, 1986, pp. 194-208.

[ABKR2] P. America, J.W. de Bakker, J.N. Kok, J.J.M.M. Rutten, Denotational semantics of a parallel object-oriented language, Report CS-R8626, Centre for Mathematics and Computer Science, Amsterdam, 1986. To appear in Information and Computation.

[BKMOZ] J.W. de Bakker, J.N. Kok, J.-J.Ch. Meyer, E.-R. Olderog, J.I. Zucker, Contrasting themes in the semantics of imperative concurrency, in: Current Trends in Concurrency. Overviews and Tutorials (J.W. de Bakker, W.P. de Roever, G. Rozenberg, eds.), LNCS 224, Springer (1986) 51-121.

[BM] J.W. de Bakker, J.-J.Ch. Meyer, Metric semantics for concurrency, Report CS-R8803, Centre for Mathematics and Computer Science, Amsterdam, 1988. To appear in BIT.

[BZ] J.W. de Bakker, J.I. Zucker, Processes and the denotational semantics of concurrency, Information and Control 54 (1982) 70-120.

[Jo] M.B. Josephs, Functional programming with side-effects, Technical Monograph PRG-55, Oxford University Computing Laboratory, 1986.

[Ka] G. Kahn, Natural Semantics, in: Proc. of Symp. on Theoretical Aspects of Computer Science (F.J. Brandenburg, G. Vidal-Naquet, M. Wirsing, eds.) , Passau, Federal Republic of Germany, February 1987, LNCS 247, Springer (1987) 22-39.

[KR] J.N. Kok, J.J.M.M. Rutten, Contractions in comparing concurrency semantics, Report CS-R8755, Centre for Mathematics and Computer Science, Amsterdam, 1987. To appear in Proc. 15th ICALP, Tampere, 1988.