A Case Study of a Combinator-Based Compiler for a Language with Blocks and Recursive Functions

Marek J. Lao

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405

TECHNICAL REPORT NO. 150

A Case Study of a Combinator-Based Compiler for a Language with Blocks and Recursive Functions

by

Marek J. Lao

October, 1983

Research reported herein was supported in part by a Fulbright Fellowship and by the National Science Foundation under grant numbers MCS 79-04183 and MCS 83-03325.

1 Introduction

This paper presents a target machine and a compiler developed by transforming the denotational semantics of a language. The method has been originally used by Wand in [4]. It consists of the following steps:

- elimination of λ-variables from the semantic equations through the introduction of special-purpose combinators,
- since the combinators have some associative properties, a combinator tree can be rotated into an almost linear form,
- distributive properties of the combinators make it possible to distribute the symbol table information into instructions, the resulting code becomes more linear and resembles a code for a conventional machine,
- after some representation decisions, a target machine to interpret the code is built.

The method has been used for an applicative language ([4]) and a simple block-structured language ([3]). The source language chosen for this paper is a block-structured language with recursive functions. We are not concerned with syntactic issues, therefore an abstract syntax resembling the one of parse trees has been chosen for the purpose of this paper.

Our main concern is the parameter passing and storage management. We discuss parameters called by function, variable, value and result. These modes can be easily extended by introducing the inout mode from Ada. Adding the call-by-name mechanism will require some additional work.

A sample program studied thoroughly in this paper is the following one:

```
(block
  (fun f    (i j)
            (value result)
            (if (zero? i)
               (block
                  (assign result 1)
                  (assign j 0)     )
               (assign result
                  (times i (f (minus1 i) j)))  ))
  (block (var i)
    (assign i 1)
    (assign i (f i i))
    (print i)  )    )
```

In the declaration of the function:

- *f* is the name of the function,
- *i* and *j* are the names of local variables for parameters,
- **value** and **result** are the modes for passing these parameters, notice that the value assigned to *i* as a **result** parameter is overwritten in the assignment

statement,

– *result* is a standard variable storing the result of the function call.

The language is presented in Section 2. Sections 3 and 4 introduce the combinators to linearize the code and discuss their basic properties. Section 5 shows the static scoping and a way to bind identifiers at compile time. Then in Section 6 we present the first target machine, which is a version of a stack-display machine. We modify the compiler in Section 7 to produce somewhat more conventional code for a machine which uses Dijkstra's display mechanism ([1]).

2 Source language

The language we have chosen for this paper is presented in Tables 1–5. Due to its complexity, we owe the reader some explanations. First of all, we assume that nonterminals with indices have exactly the same syntax and meaning as those without. $\langle \mathit{Ident} \rangle$, $\langle \mathit{Const} \rangle$ and *empty* have the usual syntax and meaning. Operation symbols, i.e. $\langle \mathit{Binop} \rangle$, $\langle \mathit{Unop} \rangle$, $\langle \mathit{Binpred} \rangle$ and $\langle \mathit{Unpred} \rangle$, also remain unspecified throughout the paper. The main reason for introducing operation symbols to the language instead of pre-declared functions is to show the possibility of "optimized" code generation for parameter passing. The mechanism is different from the one used for applications because parameters of operations are always passed by (expressed) values.

$$
\begin{array}{rcl}
\langle \mathit{Program} \rangle & ::= & \langle \mathit{Block} \rangle \\
\langle \mathit{Block} \rangle & ::= & (\mathbf{block} \langle \mathit{Dec} \rangle \langle \mathit{Stmt\text{-}list} \rangle) \\
\langle \mathit{Dec} \rangle & ::= & \langle \mathit{Var\text{-}dec} \rangle \mid \langle \mathit{Fun\text{-}dec} \rangle \mid \mathit{empty} \\
\langle \mathit{Var\text{-}dec} \rangle & ::= & (\mathbf{var} \langle \mathit{Ident} \rangle_1 \ldots \langle \mathit{Ident} \rangle_n) \qquad (n \geq 1) \\
\langle \mathit{Fun\text{-}dec} \rangle & ::= & (\mathbf{fun} \langle \mathit{Ident} \rangle\, (\langle \mathit{Ident} \rangle_1 \ldots \langle \mathit{Ident} \rangle_n)\, \langle \mathit{Mode\text{-}list\text{-}body} \rangle) \qquad (n \geq 0) \\
\langle \mathit{Mode\text{-}list\text{-}body} \rangle & ::= & (\langle \mathit{Mode\text{-}list} \rangle) \langle \mathit{Stmt\text{-}list} \rangle \\
\langle \mathit{Mode\text{-}list} \rangle & ::= & \langle \mathit{In\text{-}mode} \rangle \langle \mathit{Mode\text{-}list} \rangle \mid \langle \mathit{Out\text{-}mode} \rangle \langle \mathit{Mode\text{-}list} \rangle \mid \mathit{empty} \\
\langle \mathit{In\text{-}mode} \rangle & ::= & \mathbf{var} \mid \mathbf{value} \mid \mathbf{fun} \\
\langle \mathit{Out\text{-}mode} \rangle & ::= & \mathbf{result} \\
\langle \mathit{Stmt\text{-}list} \rangle & ::= & \langle \mathit{Stmt} \rangle \langle \mathit{Stmt\text{-}list} \rangle \mid \mathit{empty} \\
\langle \mathit{Stmt} \rangle & ::= & (\mathbf{skip}) \mid (\mathbf{assign} \langle \mathit{Lhs} \rangle \langle \mathit{Exp} \rangle) \mid (\mathbf{if} \langle \mathit{Boolexp} \rangle \langle \mathit{Stmt} \rangle_1 \langle \mathit{Stmt} \rangle_2) \mid \\
 & & (\mathbf{while} \langle \mathit{Boolexp} \rangle \langle \mathit{Stmt} \rangle) \mid (\mathbf{read} \langle \mathit{Lhs} \rangle) \mid (\mathbf{print} \langle \mathit{Exp} \rangle) \mid \langle \mathit{Block} \rangle \\
\langle \mathit{Lhs} \rangle & ::= & \langle \mathit{Ident} \rangle \\
\langle \mathit{Exp} \rangle & ::= & \langle \mathit{Lhs} \rangle \mid \langle \mathit{Rexp} \rangle \\
\langle \mathit{Rexp} \rangle & ::= & \langle \mathit{Const} \rangle \mid (\langle \mathit{Binop} \rangle \langle \mathit{Exp} \rangle_1 \langle \mathit{Exp} \rangle_2) \mid (\langle \mathit{Unop} \rangle \langle \mathit{Exp} \rangle) \mid (\langle \mathit{Fun} \rangle \langle \mathit{Apar\text{-}list} \rangle) \\
\langle \mathit{Fun} \rangle & ::= & \langle \mathit{Ident} \rangle \\
\langle \mathit{Apar\text{-}list} \rangle & ::= & \langle \mathit{Apar} \rangle \langle \mathit{Apar\text{-}list} \rangle \mid \mathit{empty} \\
\langle \mathit{Apar} \rangle & ::= & \langle \mathit{Ident} \rangle \mid \langle \mathit{Rexp} \rangle \\
\langle \mathit{Boolexp} \rangle & ::= & (\langle \mathit{Binpred} \rangle \langle \mathit{Exp} \rangle_1 \langle \mathit{Exp} \rangle_2) \mid (\langle \mathit{Unpred} \rangle \langle \mathit{Exp} \rangle)
\end{array}
$$

Table 1. Syntax of the source language

We concentrate on semantic questions. Therefore we do not check whether identifiers in a list are distinct; we assume that they are. For the same reason, we do not check whether the number of formal parameters and the number of modes for passing them are equal. Both these problems may be solved, for instance, by using a two-level grammar (as for Algol 68). Thus the questions are of a syntactic nature, and these features may be checked by a parser for our language. The syntax of declarations does not contain any type specification; we assume that all variables and functions are of the basic type, type integer.

In order to simplify semantic equations, we distinguish different occurrences of identifiers. An identifier used in an expression or actual parameter may mean:

- an expression (value),
- a variable passed to a function,
- a function passed to another function.

Since all these occurrences imply different actions to be done, it is reasonable to introduce different categories in the syntax already. Therefore we have three nonterminals, i.e. $\langle Rexp \rangle$, $\langle Exp \rangle$ and $\langle Apar \rangle$, describing expressions. An identifier used in $\langle Exp \rangle$ always means a value stored in a variable, while an identifier used in $\langle Apar \rangle$ may mean not only a value or a variable but also a function. The precise meaning in the latter case cannot be determined during compilation because formal functions, which do not specify their parameters, are allowed (see discussion in Sec. 8).

For the same reason, we introduce $\langle$*Mode-list-body*$\rangle$. The action for a parameter is carried out before ($\langle$*In-mode*$\rangle$ parameter) or after ($\langle$*Out-mode*$\rangle$ parameter) evaluating the function body (cf. [2]). Therefore semantic equations for parameter passing must consider the function body as well.

Basic domains

$Msgs$		messages
B		truth values (β)
N		integers – basic values
E	$= N$	expressed values (v)
A	$= E^* \times Msgs$	answers
L		locations (l)
D	$= L + F$	denoted values
$\overline{D}$	$= D +$ '*undeclared*'	
V	$= E + D$	storable values (v)
$\overline{V}$	$= V +$ '*unused*' $+$ '*uninitialized*'	
F_n	$= Ec \rightarrow V^n \rightarrow K$	$(n \geq 0)$
F	$= F_0 + F_1 + F_2 + \cdots$	function values (f)
Env	$= \langle Ident \rangle \rightarrow \overline{D}$	environments (ρ)
S	$= E^* \times E^* \times [L \rightarrow \overline{V}]$	states (σ)

Continuations

K	$= S \rightarrow A$	command continuations (κ)
$Envc_n$	$= Env \rightarrow L^n \rightarrow K$	declaration continuations (χ) $(n \geq 0)$
Ec	$= E \rightarrow K$	expression continuations
Bc	$= B \rightarrow K$	boolean expression continuations
Lc	$= L \rightarrow K$	L-value continuations
Fc	$= F \rightarrow K$	F-value continuations
Vc	$= \overline{V} \rightarrow K$	V-value continuations
Dc	$= \overline{D} \rightarrow K$	D-value continuations
Pc_n	$= V^n \rightarrow K$	parameter passing continuations

A product of P^n is meant to be curried, i.e. $P^n \rightarrow X = P \rightarrow \cdots \rightarrow P \rightarrow X$

Table 2. Semantic domains

The first component of state S corresponds to the state of the input tape; the second, to the state of the output tape. The function $L \rightarrow \overline{V}$ describes the state of the memory. The domain F_n consists of all n-parameter functions; V^n in its definition is a vector of parameters. Parameters passed to a function are temporarily stored under the locations of local variables. So L^n in the definition of $\mathcal{M}\ell_n$ is a vector of the locations where the parameters have been stored and where their values (modified accordingly to the modes) are to be stored lateron. L and V in the definition of the meaning functions for modes denote the location of the local variable and the value of the passed parameter respectively. Notice that the location is marked *'uninitialized'* before the appriopriate action for $\langle In\text{-}mode \rangle$ or the function body is evaluated; it is especially important for $\langle Out\text{-}mode \rangle$ parameters.

In all semantic equations, we have assumed a standardized form of semantic functions which takes the environment as an argument. This makes it shorter and easier to change the semantic equations into a combinatorial form (compare with [3], [4]). Therefore the functions giving the meaning of constants, operators and modes require the environment. Their actions do not depend on its value, though.

$$
\begin{array}{l}
\mathcal{P} : \langle Program \rangle \to S \to A \\
\mathcal{B}\ell : \langle Block \rangle \to Env \to K \to K \\
\mathcal{D}_n : \langle Dec \rangle \to Env \to Envc_n \to K \qquad (n \geq 0) \\
\mathcal{S}\ell : \langle Stmt\text{-}list \rangle \to Env \to K \to K \\
\mathcal{S} : \langle Stmt \rangle \to Env \to K \to K \\
\mathcal{R} : \langle Exp \rangle \to Env \to Ec \to K \\
\mathcal{C} : \langle Const \rangle \to Env \to Ec \to K \\
\mathcal{L} : \langle Lhs \rangle \to Env \to Lc \to K \\
\mathcal{B} : \langle Boolexp \rangle \to Env \to Bc \to K \\
\mathcal{F} : \langle Fun \rangle \to Env \to Fc \to K \\
\mathcal{M}\ell_n : \langle Mode\text{-}list\text{-}body \rangle \to Env \to L^n \to K \qquad (n \geq 0) \\
\mathcal{A}\ell_n : \langle Apar\text{-}list \rangle \to Env \to Pc_n \to K \qquad (n \geq 0) \\
\mathcal{A} : \langle Apar \rangle \to Env \to Vc \to K \\
\mathcal{M}_{in} : \langle In\text{-}mode \rangle \to Env \to K \to L \to V \to K \\
\mathcal{M}_{out} : \langle Out\text{-}mode \rangle \to Env \to K \to L \to V \to K \\
\mathcal{O}_b : \langle Binop \rangle \to Env \to Ec \to E \to E \to K \\
\mathcal{O}_u : \langle Unop \rangle \to Env \to Ec \to E \to K \\
\mathcal{O}_{bp} : \langle Binpred \rangle \to Env \to Bc \to E \to E \to K \\
\mathcal{O}_{up} : \langle Unpred \rangle \to Env \to Bc \to E \to K
\end{array}
$$

Table 3. Meaning functions

The basic kind of a block is a declaration of variables and a list of statements to be evaluated in the new environment containing these variables. We have decided that after the evaluation of the statement list in a block, the locations of its variables are released. Therefore the continuation for the statements is of the form $release\text{-}block_n \rho' l_1 \ldots l_n \kappa$ and the declaration continuation requires the vector of the new locations. This shows us that a stack-wise implementation of the memory is possible.

As we have already noticed, the meaning of actual parameters must be determined by the applied function and, in the general case, it requires run-time checking. In order to have only one copy of the code that checks the correctness of actual parameters, we have decided to introduce a two-step application. The first step ($\mathcal{A}$ and $\mathcal{A}\ell$) evaluates actual parameters to expressed values, if the parameter is $\langle Rexp \rangle$, and to denoted values for identifiers. The next step ($\mathcal{M}\ell$) checks the correctness between actual parameters and the specification within the applied function. During this step dereferencing of the passed values is carried out if necessary. Thus the action for a **var** parameter (*L-pass*) stores the location of the passed variable in the local variable, for a **fun** parameter *F-pass* stores the *F*-value in the local variable. For a **value** parameter (*E-pass*), the local variable is initialized with the value of the expression; if a variable has been passed, it is

Programs

$$\mathcal{P}\llbracket p\rrbracket = \lambda\sigma_0.\mathcal{B}\ell\llbracket p\rrbracket\,\mathit{init\text{-}env}\ \mathit{init\text{-}cont}\ \sigma_0$$

Statements

$$\mathcal{S}\ell\llbracket \mathit{empty}\rrbracket = \lambda\rho\kappa.\kappa$$
$$\mathcal{S}\ell\llbracket s\ sl\rrbracket = \lambda\rho\kappa.\mathcal{S}\llbracket s\rrbracket\rho(\mathcal{S}\ell\llbracket sl\rrbracket\rho\kappa)$$
$$\mathcal{S}\llbracket(\mathbf{skip})\rrbracket = \lambda\rho\kappa.\kappa$$
$$\mathcal{S}\llbracket(\mathbf{assign}\ id\ e)\rrbracket = \lambda\rho\kappa.\mathcal{L}\llbracket id\rrbracket\rho(\lambda l.\mathcal{R}\llbracket e\rrbracket\rho(\mathit{store}\,\rho\kappa l))$$
$$\mathcal{S}\llbracket(\mathbf{if}\ b\ s_1\ s_2)\rrbracket = \lambda\rho\kappa.\mathcal{B}\llbracket b\rrbracket\rho(\lambda\beta.\beta\rightarrow\mathcal{S}\llbracket s_1\rrbracket\rho\kappa,\ \mathcal{S}\llbracket s_2\rrbracket\rho\kappa)$$
$$\mathcal{S}\llbracket(\mathbf{while}\ b\ s)\rrbracket = \lambda\rho\kappa.\mathit{fix}\ (\lambda\theta.\mathcal{B}\llbracket b\rrbracket\rho(\lambda\beta.\beta\rightarrow\mathcal{S}\llbracket s\rrbracket\rho\theta,\kappa))$$
$$\mathcal{S}\llbracket(\mathbf{read}\ id)\rrbracket = \lambda\rho\kappa.\mathcal{L}\llbracket id\rrbracket\rho(\lambda l.\mathit{do\text{-}read}\,\rho(\mathit{store}\,\rho\kappa l))$$
$$\mathcal{S}\llbracket(\mathbf{print}\ e)\rrbracket = \lambda\rho\kappa.\mathcal{R}\llbracket e\rrbracket\rho(\lambda v.\mathit{do\text{-}print}\,\rho\kappa v)$$
$$\mathcal{S}\llbracket \mathit{block}\rrbracket = \mathcal{B}\ell\llbracket \mathit{block}\rrbracket$$

Expressions

$$\mathcal{R}\llbracket id\rrbracket = \lambda\rho\eta.\mathcal{L}\llbracket id\rrbracket\rho(\mathit{fetch}\,\rho\eta)$$
$$\mathcal{R}\llbracket c\rrbracket = \mathcal{C}\llbracket c\rrbracket$$
$$\mathcal{R}\llbracket(\oplus\ e_1\ e_2)\rrbracket = \lambda\rho\eta.\mathcal{R}\llbracket e_1\rrbracket\rho(\lambda v_1.\mathcal{R}\llbracket e_2\rrbracket\rho(\lambda v_2.\mathcal{O}_b\llbracket\oplus\rrbracket\rho\eta v_1 v_2))$$
$$\mathcal{R}\llbracket(\ominus\ e)\rrbracket = \lambda\rho\eta.\mathcal{R}\llbracket e\rrbracket\rho(\lambda v.\mathcal{O}_u\llbracket\ominus\rrbracket\rho\eta v)$$
$$\mathcal{R}\llbracket(id\ a_1\ldots a_n)\rrbracket = \lambda\rho\eta.\mathcal{F}\llbracket id\rrbracket\rho(\mathit{check}_n\,\rho(\lambda f.\mathcal{A}\ell_n\llbracket a_1\ldots a_n\rrbracket\rho(\lambda v_1\ldots v_n.f\eta v_1\ldots v_n)))$$
$$\mathcal{B}\llbracket(\prec\ e_1\ e_2)\rrbracket = \lambda\rho\eta.\mathcal{R}\llbracket e_1\rrbracket\rho(\lambda v_1.\mathcal{R}\llbracket e_2\rrbracket\rho(\lambda v_2.\mathcal{O}_{bp}\llbracket\prec\rrbracket\rho\eta v_1 v_2))$$
$$\mathcal{B}\llbracket(?\ e)\rrbracket = \lambda\rho\eta.\mathcal{R}\llbracket e\rrbracket\rho(\lambda v.\mathcal{O}_{up}\llbracket ?\rrbracket\rho\eta v)$$
$$\mathcal{F}\llbracket id\rrbracket = \lambda\rho\eta.\mathit{lookup}\llbracket id\rrbracket\rho(\lambda v.v\in F\rightarrow\eta v,\ \mathit{terminate}\llbracket\text{“not a function applied”}\rrbracket)$$
$$\mathcal{L}\llbracket id\rrbracket = \lambda\rho\eta.\mathit{lookup}\llbracket id\rrbracket\rho(\lambda v.v\in L\rightarrow\eta v,\ \mathit{terminate}\llbracket\text{“not a variable”}\rrbracket)$$

Actual parameters

$$\mathcal{A}\ell_0\llbracket \mathit{empty}\rrbracket = \lambda\rho\eta.\eta$$
$$\mathcal{A}\ell_n\llbracket a\ al\rrbracket = \lambda\rho\eta.\mathcal{A}\llbracket a\rrbracket\rho(\lambda v.\mathcal{A}\ell_{n-1}\llbracket al\rrbracket\rho(\eta v))$$
$$\mathcal{A}\llbracket id\rrbracket = \mathit{lookup}\llbracket id\rrbracket$$
$$\mathcal{A}\llbracket e\rrbracket = \mathcal{R}\llbracket e\rrbracket$$

Blocks

$$\mathcal{B}\ell\llbracket(\mathbf{block}\ sl)\rrbracket = \mathcal{S}\ell\llbracket sl\rrbracket$$
$$\mathcal{B}\ell\llbracket(\mathbf{block}\ (\mathbf{var}\ id_1\ldots id_n)\ sl)\rrbracket = \lambda\rho\kappa.\mathcal{D}_n\llbracket(\mathbf{var}\ id_1\ldots id_n)\rrbracket\rho(\lambda\rho' l_1\ldots l_n.\mathcal{S}\ell\llbracket sl\rrbracket\rho'\,(\mathit{release\text{-}block}_n\,\rho' l_1\ldots l_n\kappa))$$
$$\mathcal{B}\ell\llbracket(\mathbf{block}\ (\mathbf{fun}\ \alpha)\ sl)\rrbracket = \lambda\rho\kappa.\mathcal{D}_0\llbracket(\mathbf{fun}\ \alpha)\rrbracket\rho(\lambda\rho'.\mathcal{S}\ell\llbracket sl\rrbracket\rho'\kappa)$$

Variable declaration

$$\mathcal{D}_n\llbracket(\mathbf{var}\ id_1\ldots id_n)\rrbracket = \lambda\rho\chi\sigma.\mathit{let}\ (l_1,\ldots l_n) = \mathit{new}_n\sigma,\ \mathit{and}\ \sigma' = \mathit{adjoin}_n\,\sigma l_1\ldots l_n\ \mathit{in}\ \chi(\rho[l_1/id_1]\ldots[l_n/id_n])l_1\ldots l_n\sigma'$$

Function declaration

$$\mathcal{D}_0\llbracket(\mathbf{fun}\ id\ (id_1\ldots id_n)\ (m_1\ldots m_n)\ sl)\rrbracket = \lambda\rho\chi.\chi(\mathit{fix}\ (\lambda\rho'.[(f\rho')/id]))$$

where f is:

$$f = \lambda\rho'\eta v_1\ldots v_n\sigma.\mathit{let}\ (l_0,l_1,\ldots l_n) = \mathit{new}_{n+1}\sigma,\ \mathit{and}\ \rho'' = \rho'[l_0/\mathrm{result}][l_1/id_1]\ldots[l_n/id_n],$$
$$\mathit{and}\ \sigma' = \mathit{adjoin\text{-}init}_{n+1}\,\sigma l_0 l_1\ldots l_n\,\text{‘}\mathit{uninitialized}\text{’}\,v_1\ldots v_n$$
$$\mathit{in}\ \mathcal{M}\ell_n\llbracket(m_1\ldots m_n)\ sl\rrbracket\rho'' l_1\ldots l_n(\mathit{fetch}\,\rho''(\mathit{release\text{-}fun}_{n+1}\,\rho'' l_0 l_1\ldots l_n\eta)l_0)\sigma'$$

Parameter passing

$$\mathcal{M}\ell_0\llbracket(\mathit{empty})\ sl\rrbracket = \mathcal{S}\ell\llbracket sl\rrbracket$$
$$\mathcal{M}\ell_n\llbracket(im\ ml)\ sl\rrbracket = \lambda\rho l_1\ldots l_n\kappa\sigma.\mathcal{M}_{in}\llbracket im\rrbracket\rho(\mathcal{M}\ell_{n-1}\llbracket(ml)\ sl\rrbracket\rho l_2\ldots l_n\kappa)l_1(\sigma_3 l_1)\,\langle\sigma_1,\sigma_2,\sigma_3[\text{‘}\mathit{uninitialized}\text{’}/l_1]\rangle$$
$$\mathcal{M}\ell_n\llbracket(om\ ml)\ sl\rrbracket = \lambda\rho l_1\ldots l_n\kappa\sigma.\mathcal{M}\ell_{n-1}\llbracket(ml)\ sl\rrbracket\rho l_2\ldots l_n(\mathcal{M}_{out}\llbracket om\rrbracket\rho\kappa l_1(\sigma_3 l_1))\,\langle\sigma_1,\sigma_2,\sigma_3[\text{‘}\mathit{uninitialized}\text{’}/l_1]\rangle$$

Others

$\mathcal{C}\llbracket c\rrbracket = \mathit{const}\llbracket c\rrbracket$	$\mathcal{O}_b\llbracket\oplus\rrbracket = \mathit{binop}\llbracket\oplus\rrbracket$	$\mathcal{O}_u\llbracket\ominus\rrbracket = \mathit{unop}\llbracket\ominus\rrbracket$
$\mathcal{O}_{bp}\llbracket\prec\rrbracket = \mathit{binpred}\llbracket\prec\rrbracket$	$\mathcal{O}_{up}\llbracket ?\rrbracket = \mathit{unpred}\llbracket ?\rrbracket$	$\mathcal{M}_{in}\llbracket\mathbf{var}\rrbracket = \mathit{L\text{-}pass}$
$\mathcal{M}_{in}\llbracket\mathbf{value}\rrbracket = \mathit{E\text{-}pass}$	$\mathcal{M}_{in}\llbracket\mathbf{fun}\rrbracket = \mathit{F\text{-}pass}$	$\mathcal{M}_{out}\llbracket\mathbf{result}\rrbracket = \mathit{I\text{-}pass}$

Table 4. Semantic equations

$$
\begin{array}{l}
\mathit{terminate}[\![m]\!] = \lambda\sigma.(\sigma_2 \parallel m) \\
\mathit{init\text{-}env} = \lambda id.\,\textit{'undeclared'} \\
\mathit{init\text{-}cont} = \mathit{terminate}[\![\textit{"normal termination"}]\!] \\
\mathit{new}_n = \lambda\sigma.\text{some distinct } (l_1, \ldots l_n) \text{ such that } (\sigma_3 l_i) = \textit{'unused'} \\
\mathit{adjoin}_n = \lambda\sigma l_1 \ldots l_n.\langle\sigma_1, \sigma_2, \sigma_3[\textit{'uninitialized'}/l_1]\ldots[\textit{'uninitialized'}/l_n]\rangle \\
\mathit{adjoin\text{-}init}_n = \lambda\sigma l_1 \ldots l_n v_1 \ldots v_n.\langle\sigma_1, \sigma_2, \sigma_3[v_1/l_1]\ldots[v_n/l_n]\rangle \\
\mathit{release}_n = \lambda\sigma l_1 \ldots l_n.\langle\sigma_1, \sigma_2, \sigma_3[\textit{'unused'}/l_1]\ldots[\textit{'unused'}/l_n]\rangle \\
\mathit{release\text{-}block}_n = \lambda\rho l_1 \ldots l_n \kappa\sigma.\kappa(\mathit{release}_n \sigma l_1 \ldots l_n) \\
\mathit{release\text{-}fun}_n = \lambda\rho l_1 \ldots l_n \eta v\sigma.\eta v(\mathit{release}_n \sigma l_1 \ldots l_n) \\
\mathit{store} = \lambda\rho\kappa l v\sigma.\kappa\langle\sigma_1, \sigma_2, \sigma_3[v/l]\rangle \\
\mathit{do\text{-}read} = \lambda\rho\eta\sigma.\sigma_1 = \mathit{nil} \rightarrow \mathit{terminate}[\![\text{"eof encountered"}]\!]\sigma, \eta(\mathit{first}\ \sigma_1)\langle\mathit{rest}\ \sigma_1, \sigma_2, \sigma_3\rangle \\
\mathit{do\text{-}print} = \lambda\rho\kappa v\sigma.\kappa\langle\sigma_1, \sigma_2 \parallel v, \sigma_3\rangle \\
\mathit{fetch} = \lambda\rho\eta l\sigma.(\sigma_3 l) = \textit{'uninitialized'} \rightarrow \mathit{terminate}[\![\text{"unininitialized variable"}]\!]\sigma, \eta(\sigma_3 l)\sigma \\
\mathit{check}_n = \lambda\rho\eta f.f \in F_n \rightarrow \eta f, \mathit{terminate}[\![\text{"wrong number of parameters"}]\!] \\
\mathit{find}[\![id]\!] = \lambda\rho\eta.\eta(\rho\ id) \\
\mathit{deref?} = \lambda\rho\eta v\sigma.v \in L \rightarrow \eta(\sigma_3 v)\sigma, \eta v\sigma \\
\mathit{L\text{-}pass} = \lambda\rho\kappa l v.v \in L \rightarrow \mathit{store}\rho\kappa l v, \mathit{terminate}[\![\text{"not a variable passed"}]\!] \\
\mathit{E\text{-}pass} = \lambda\rho\kappa l v.\mathit{deref?}\rho(\lambda v'.v' \in E \rightarrow \mathit{store}\rho\kappa l v', \mathit{terminate}[\![\text{"not an expression passed"}]\!])v \\
\mathit{F\text{-}pass} = \lambda\rho\kappa l v.v \in DF \rightarrow \mathit{store}\rho\kappa l v, \mathit{terminate}[\![\text{"not a function passed"}]\!] \\
\mathit{I\text{-}pass} = \lambda\rho\kappa l v.v \in L \rightarrow \mathit{fetch}\rho(\mathit{store}\rho\kappa v)l, \mathit{terminate}[\![\text{"not a variable passed for result"}]\!] \\
\mathit{lookup}[\![id]\!] = \lambda\rho\eta.\mathit{find}[\![id]\!]\rho(\lambda d\sigma.d = \textit{'undeclared'} \rightarrow \mathit{terminate}[\![\text{"undeclared identifier"}]\!]\sigma, \\
\qquad\qquad d \in F \rightarrow \eta d\sigma, \qquad \text{— regular function} \\
\qquad\qquad (\sigma_3\, d) \in F \rightarrow \eta(\sigma_3\, d)\sigma, \qquad \text{— fun parameter} \\
\qquad\qquad (\sigma_3\, d) \in L \rightarrow \eta(\sigma_3\, d)\sigma, \qquad \text{— var parameter} \\
\qquad\qquad \eta d\sigma)
\end{array}
$$

Table 5. Auxiliary functions

dereferenced. For a **result** parameter (*I-pass*), the value assigned to the local variable (l) is stored under the passed location (v). The last action has to be done after evaluating the body; so *I-pass*'es are put on "the stack of continuations". Due to the order of parameters, the **result** parameters are assigned values from the right-hand side to the left. In our solution, the checking of the kind of a **result** parameter is done after evaluating the function body. Both the order of passing **result** parameters and the time of checking their kind can be easily changed.

Finally, let us discuss the equation for a function declaration. A function declaration itself changes the environment for the block and the function body only. The F-value in this new environment is quite complex. First of all, a function transfers its parameters to the locations of local variables (*adjoin-init*). Then the parameter passing and evaluating the function body ($\mathcal{M}\ell$) are to be done; $l_1, \ldots l_n$ passed to $\mathcal{M}\ell$ denote the locations of the local variables. The continuation for $\mathcal{M}\ell$ consists of:

- fetching the value of the function call; every function possesses a (predeclared) variable with the standard identifier *result*; the value assigned to this variable at the end of the evaluation of the function body becomes the value of the application* ,
- releasing the locations of local variables (*release-fun*).

*This idea comes from Loglan, a programming language designed and implemented at the Institute of Informatics, the University of Warsaw, Poland

Now we present two examples to be discussed in the paper.

Example 1

The following program will illustrate static scoping and the implementation of a loop:

```
(block   (var i j)
   (block   (var i)
     (assign i 1)
     (while (positive? i)
            (assign i (minus1 i))  )
     (assign j i)    )
   (assign i j)
   (print i)    )
```

The answer is, of course, the value of i from the outer block. The operators of *positive?* and *minus1* have the natural semantics.

Example 2

The following program contains a recursive function to compute the factorial of its first parameter. The other parameter is to illustrate the **result** mode.

```
(block
   (fun  f  (i j)
            (value result)
            (if (zero? i)
                (block
                  (assign result 1)
                  (assign j 0)  )
                (assign result
                  (times i  (f (minus1 i) j) ))    ))
   (block   (var i)
     (assign i 1)
     (assign i (f i i))
     (print i)    ))
```

The answer should be 1, i.e. the value of the factorial , despite the fact that i occurs as a **result** parameter and is assigned the value of 0 as well.

3 Combinatorial semantic equations

Now we rewrite the semantic equations eliminating λ-variables. The additional auxiliary functions are shown in Table 6. The equations are modified by the introduction of special-purpose combinators:

– a family of sequencing combinators (cf. [4])

$$D_n(\alpha,\beta) = \lambda\rho x_0 x_1 \ldots x_n.\alpha\rho(\beta\rho x_0 x_1 \ldots x_n)$$

$$
\begin{aligned}
&\mathit{return} = \lambda\rho\kappa.\kappa\\
&\mathit{test}\ f\,g = \lambda\rho\kappa\beta.\beta \rightarrow f\rho\kappa, g\rho\kappa\\
&\mathit{wloop}\ f = \lambda\rho\kappa.\mathit{fix}\,(f\rho\kappa)\\
&\mathit{wtest}\ f = \lambda\rho\kappa\theta\beta.\beta \rightarrow f\rho\theta, \kappa\\
&\mathit{table}_0 = \mathit{init\text{-}env}\\
&\mathit{ext}_n\ \mathit{id}_n \ldots \mathit{id}_1 = \lambda\rho l_1 \ldots l_n.\rho[l_1/\mathit{id}_1]\ldots[l_n/\mathit{id}_n]\\
&\mathit{block}_n\ f\,\mathit{env} = \lambda\kappa\sigma.\mathit{let}\,(l_1,\ldots l_n) = \mathit{new}_n\sigma \text{ and } \sigma' = \mathit{adjoin}_n\sigma l_1\ldots l_n\\
&\qquad\qquad\text{in } f(\mathit{env}\,l_1\ldots l_n)l_1\ldots l_n\kappa\sigma'\\
&\mathit{ext\text{-}fun}\ \mathit{id}\,f = \lambda\rho.\mathit{fix}\,(\lambda\rho'.\rho[(f\rho')/\mathit{id}])\\
&\mathit{function}_n\ f\,\mathit{env} = \lambda\eta v_1\ldots v_n\sigma.\mathit{let}\,(l_0,l_1,\ldots l_n) = \mathit{new}_n\sigma\\
&\qquad\qquad\qquad\text{and } \sigma' = \mathit{adjoin\text{-}init}_{n+1}\sigma l_0 l_1\ldots l_n\ \text{`uninitialized'}v_1\ldots v_n\\
&\qquad\qquad\qquad\text{in } f(\mathit{env}\,l_0 l_1\ldots l_n)l_1\ldots l_n l_0 l_1\ldots l_n\eta l_0\sigma'\\
&\mathit{pass}_n\ f = \lambda\rho a_0 a_1\ldots a_n\sigma.f\rho a_1\ldots a_n a_0(\sigma_3\, a_0)\langle\sigma_1,\sigma_2,\sigma_3[\text{`uninitialized'}/a_0]\rangle\\
&\mathit{apply}_n = \lambda\rho\eta f v_1\ldots v_n.f\eta v_1\ldots v_n
\end{aligned}
$$

Table 6. Additional auxiliary functions

- a family of binding combinators (cf. [4],[5])
$$B_p(\alpha,\beta) = \lambda a_1\ldots a_p.\alpha(\beta a_1\ldots a_p)$$
- a family of generalized D's ("post" combinators)
$$P_{m\,n}(\alpha,\beta) = \lambda\rho a_1\ldots a_m x_0 x_1\ldots x_n.\alpha\rho a_1\ldots a_m(\beta\rho x_0 x_1\ldots x_n)$$
- a transferring combinator
$$T(\alpha) = \lambda\rho\eta v.\alpha\rho(\eta v)$$

The D and B combinators have been presented and discussed by Wand in [4], [5]. In the original version, D was always used with a continuation in place of x_0 . Here we use this combinator in a little different way; we know that one of the x's is a continuation. It does not, however, change any of the properties of D discussed in [4].

The P combinators generalize D's in the sense that some of their parameters are bound to the first argument while the rest to the second. These combinators are extensively used in the equations for parameter passing. While the combinator tree is rotated, however, the combinators become of the form of $P_{0\,n}$ which is equivalent to D_n. Here also one of the x's is a continuation. In our applications, a's are the locations of formal parameters passed to the function body (α); x's are the parameters for passing results to the calling module as well as releasing locations, that is β corresponds to the action to be done after evaluating the function body, *I-pass, fetch, release-fun*.

The T combinator, due to Wand, transfers a value passed as a parameter onto the stack for continuation. Similarly to P, it disappears while the combinator tree is being linearized.

Now using the combinators and auxiliary functions we are able to rewrite the semantic equations without λ-variables. The results are shown in Table 7. Notice that checking whether an identifier has been declared can be done during compilation. It is also possible to determine the kind of an identifier at the same time. Therefore, the equations for $\mathcal{L}[\![id]\!]$, $\mathcal{F}[\![id]\!]$ and $\mathcal{A}[\![id]\!]$ are reduced to $\mathit{lookup}[\![id]\!]$.

$\mathcal{S}\ell\llbracket empty \rrbracket = \mathit{return}$

$\mathcal{S}\ell\llbracket sl\ sl \rrbracket = D_0(\mathcal{S}\llbracket s \rrbracket, \mathcal{S}\ell\llbracket sl \rrbracket)$

$\mathcal{S}\llbracket(\mathbf{skip})\rrbracket = \mathit{return}$

$\mathcal{S}\llbracket(\mathbf{assign}\ id\ e)\rrbracket = D_0(\mathcal{L}\llbracket id \rrbracket, D_1(\mathcal{R}\llbracket e \rrbracket, \mathit{store}))$

$\mathcal{S}\llbracket(\mathbf{if}\ b\ s_1\ s_2)\rrbracket = D_0(\mathcal{B}\llbracket b \rrbracket, \mathit{test}\ \mathcal{S}\llbracket s_1 \rrbracket\ \mathcal{S}\llbracket s_2 \rrbracket)$

$\mathcal{S}\llbracket(\mathbf{while}\ b\ s)\rrbracket = \mathit{wloop}\ D_1(\mathcal{B}\llbracket b \rrbracket, \mathit{wtest}\ D_0(\mathcal{S}\llbracket s \rrbracket, \mathit{return}))$

$\mathcal{S}\llbracket(\mathbf{read}\ id)\rrbracket = D_0(\mathcal{L}\llbracket id \rrbracket, D_1(\mathit{do\text{-}read}, \mathit{store}))$

$\mathcal{S}\llbracket(\mathbf{print}\ e)\rrbracket = D_0(\mathcal{R}\llbracket e \rrbracket, \mathit{do\text{-}print})$

$\mathcal{S}\llbracket block \rrbracket = \mathcal{B}\ell\llbracket block \rrbracket$

$\mathcal{R}\llbracket id \rrbracket = D_0(\mathcal{L}\llbracket id \rrbracket, \mathit{fetch})$

$\mathcal{R}\llbracket c \rrbracket = \mathit{const}\llbracket c \rrbracket$

$\mathcal{R}\llbracket(\oplus\ e_1\ e_2)\rrbracket = D_0(\mathcal{R}\llbracket e_1 \rrbracket, D_1(\mathcal{R}\llbracket e_2 \rrbracket, \mathit{binop}\llbracket \oplus \rrbracket))$

$\mathcal{R}\llbracket(\ominus\ e)\rrbracket = D_0(\mathcal{R}\llbracket e \rrbracket, \mathit{unop}\llbracket \ominus \rrbracket)$

$\mathcal{R}\llbracket(id\ a_1 \ldots a_n)\rrbracket = D_0(\mathcal{F}\llbracket id \rrbracket, D_0(\mathit{check}_n, D_1(\mathcal{A}\ell_n\llbracket a_1 \ldots a_n \rrbracket, \mathit{apply}_n))$

$\mathcal{B}\llbracket(\prec\ e_1\ e_2)\rrbracket = D_0(\mathcal{R}\llbracket e_1 \rrbracket, D_1(\mathcal{R}\llbracket e_2 \rrbracket, \mathit{binpred}\llbracket \prec \rrbracket))$

$\mathcal{B}\llbracket(?\ e)\rrbracket = D_0(\mathcal{R}\llbracket e \rrbracket, \mathit{unpred}\llbracket ? \rrbracket)$

$\mathcal{L}\llbracket id \rrbracket = \mathit{lookup}\llbracket id \rrbracket$

$\mathcal{F}\llbracket id \rrbracket = \mathit{lookup}\llbracket id \rrbracket$

$\mathcal{A}\ell_0\llbracket empty \rrbracket = \mathit{return}$

$\mathcal{A}\ell_n\llbracket a\ al \rrbracket = D_0(\mathcal{A}\llbracket a \rrbracket, T(\mathcal{A}\ell_{n-1}\llbracket al \rrbracket))$

$\mathcal{A}\llbracket id \rrbracket = \mathit{lookup}\llbracket id \rrbracket$

$\mathcal{A}\llbracket e \rrbracket = \mathcal{R}\llbracket e \rrbracket$

$\mathcal{B}\ell\llbracket(\mathrm{block}\ sl)\rrbracket = \mathcal{S}\ell\llbracket sl \rrbracket$

$\mathcal{B}\ell\llbracket(\mathrm{block}\ (\mathrm{var}\ id_1 \ldots id_n)\ sl)\rrbracket = B_1(\mathit{block}_n\ D_n(\mathcal{S}\ell\llbracket sl \rrbracket, \mathit{release\text{-}block}_n), \mathit{ext}_n\ id_n \ldots id_1)$

$\mathcal{B}\ell\llbracket(\mathrm{block}\ (\mathrm{fun}\ id\ (id_1 \ldots id_n)\ (ml)\ sl_1)\ sl_2)\rrbracket = B_1(\mathcal{S}\ell\llbracket sl_2 \rrbracket, \mathit{ext\text{-}fun}\ id\ f)$

where f is:

$\quad f = B_1(\mathit{function}_n\ P_{n\,n+2}(\mathcal{M}\ell_n\llbracket(ml)\ sl_1\rrbracket, D_{n+1}(\mathit{fetch}, \mathit{release\text{-}fun}_{n+1})), \mathit{ext}_{n+1}\ id_n \ldots id_1\ \mathrm{result})$

$\mathcal{M}\ell_0\llbracket(empty)\ sl \rrbracket = \mathcal{S}\ell\llbracket sl \rrbracket$

$\mathcal{M}\ell_n\llbracket(im\ ml)\ sl \rrbracket = \mathit{pass}_n(D_{n-1}(\mathcal{M}_{in}\llbracket im \rrbracket, \mathcal{M}\ell_{n-1}\llbracket(ml)\ sl \rrbracket))$

$\mathcal{M}\ell_n\llbracket(om\ ml)\ sl \rrbracket = \mathit{pass}_n(P_{n-1\,2}(\mathcal{M}\ell_{n-1}\llbracket(ml)\ sl \rrbracket, \mathcal{M}_{out}\llbracket om \rrbracket))$

Table 7. Combinatorial form of semantic equations

Since the modification of the equations is generally not difficult, we present the required transformations for only a few of them.

The equation for a block declaring variables may be modified as follows:

$$
\begin{aligned}
&\mathcal{B}\ell\llbracket(\mathbf{block}\ (\mathbf{var}\ id_1 \ldots id_n)\ sl)\rrbracket \\
&\qquad = \lambda\rho\kappa.\mathcal{D}_n\llbracket(\mathbf{var}\ id_1 \ldots id_n)\rrbracket\rho(\lambda\rho' l_1 \ldots l_n.D_n(\mathcal{S}\ell\llbracket sl \rrbracket, \mathit{release\text{-}block}_n)\rho' l_1 \ldots l_n \kappa) \\
&\qquad = \lambda\rho\kappa\sigma.\mathit{let}\ (l_1, \ldots l_n) = \mathit{new}_n\sigma\ \mathit{and}\ \sigma' = \mathit{adjoin}_n\sigma l_1 \ldots l_n \\
&\qquad\qquad \mathit{in}\ D_n(\mathcal{S}\ell\llbracket sl \rrbracket, \mathit{release\text{-}block}_n)(\rho[l_1/id_1]\ldots[l_n/id_n])l_1 \ldots l_n\kappa\sigma' \\
&\qquad = \lambda\rho.[\lambda env\kappa\sigma.\mathit{let}\ (l_1, \ldots l_n) = \mathit{new}_n\sigma\ \mathit{and}\ \sigma' = \mathit{adjoin}_n\sigma l_1 \ldots l_n \\
&\qquad\qquad\quad \mathit{in}\ D_n(\mathcal{S}\ell\llbracket sl \rrbracket, \mathit{release\text{-}block}_n)(env\ l_1 \ldots l_n)l_1 \ldots l_n\kappa\sigma']\ (\mathit{ext}_n\, id_n \ldots id_n\rho) \\
&\qquad = \lambda\rho.\mathit{block}_n D_n(\mathcal{S}\ell\llbracket sl \rrbracket, \mathit{release\text{-}block}_n)(\mathit{ext}_n\ id_n \ldots id_1\rho) \\
&\qquad = B_n(\mathit{block}_n\ D_n(\mathcal{S}\ell\llbracket sl \rrbracket, \mathit{release\text{-}block}_n)), \mathit{ext}_n\ id_n \ldots id_1)
\end{aligned}
$$

The transformations required for a function declaration are more complicated and are defined below:

$$\mathcal{D}_0\llbracket(\mathbf{fun}\ id\ (id_1 \ldots id_n)\ (ml)\ sl)\rrbracket = \lambda\rho\chi.\chi((\mathit{ext\text{-}fun}\ id f)\rho)$$

where f is:

$$
\begin{aligned}
f = {}&\lambda\rho'\eta v_1 \ldots v_n\sigma.\mathit{let}\ (l_0, l_1, \ldots l_n) = \mathit{new}_{n+1}\sigma\ \mathit{and}\ \rho'' = \rho'[l_0/\mathrm{result}][l_1/id_1]\ldots[l_n/id_n] \\
&\mathit{and}\ \sigma' = \mathit{adjoin\text{-}init}_{n+1}\sigma l_0 l_1 \ldots l_n\ \text{`}\mathit{uninitialized}\text{'} v_1 \ldots v_n \\
&\mathit{in}\ \mathcal{M}\ell_n\llbracket(ml)\ sl\rrbracket\rho'' l_1 \ldots l_n(\mathit{fetch}\rho''(\mathit{release\text{-}fun}_{n+1}\rho'' l_0 l_1 \ldots l_n\eta)l_0)\sigma'
\end{aligned}
$$

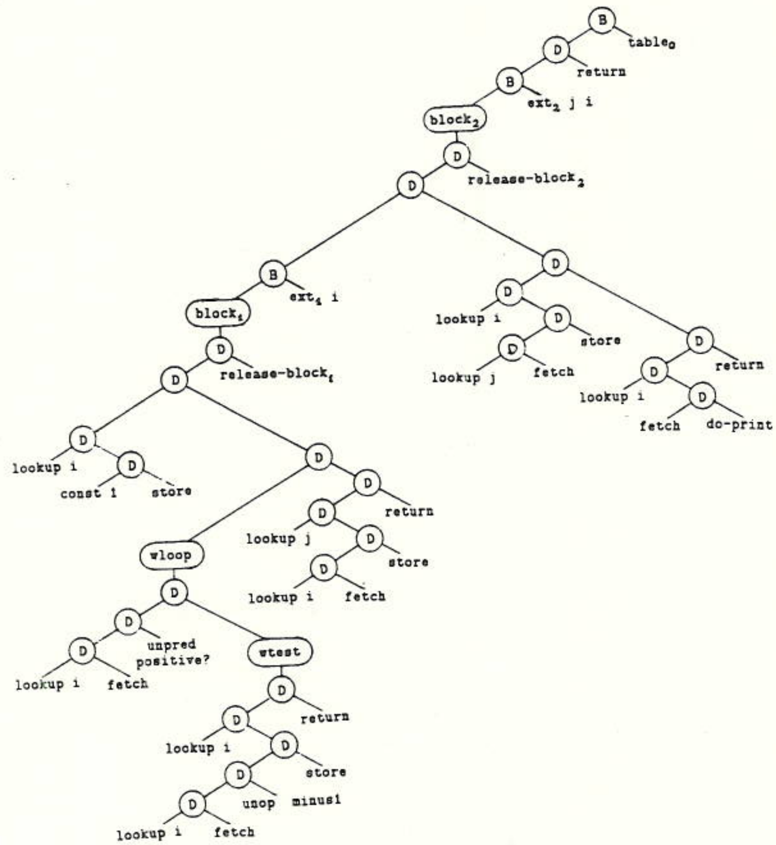

Figure 1. Example 1 – the "naive" code

$$
\begin{aligned}
&= \lambda\rho'\eta v_1\ldots v_n\sigma.\mathit{let}\,(l_0,l_1,\ldots l_n) = \mathit{new}_{n+1}\sigma \ \mathit{and}\ \rho'' = \rho'[l_0/\mathrm{result}][l_1/id_1]\ldots[l_n/id_n]\\
&\qquad\qquad \mathit{and}\ \sigma' = \mathit{adjoin\text{-}init}_{n+1}\sigma l_0 l_1\ldots l_n\ \text{`}\mathit{uninitialized}\text{'} v_1\ldots v_n\\
&\qquad\qquad \mathit{in}\ \mathcal{M}\ell_n[\![(ml)\ sl]\!]\rho'' l_1\ldots l_n(D_n(\mathit{fetch}, \mathit{release\text{-}fun}_{n+1})\rho'' l_0 l_1\ldots l_n\eta l_0)\sigma'\\
&= \lambda\rho'\eta v_1\ldots v_n\sigma.\mathit{let}\,(l_0,l_1,\ldots l_n) = \mathit{new}_{n+1}\sigma \ \mathit{and}\ \rho'' = \rho'[l_0/\mathrm{result}][l_1/id_1]\ldots[l_n/id_n]\\
&\qquad\qquad \mathit{and}\ \sigma' = \mathit{adjoin\text{-}init}_{n+1}\sigma l_0 l_1\ldots l_n\ \text{`}\mathit{uninitialized}\text{'} v_1\ldots v_n\\
&\qquad\qquad \mathit{in}\ P_{n\,n+2}(\mathcal{M}\ell_n[\![(ml)\ sl]\!], D_{n+1}(\mathit{fetch}, \mathit{release\text{-}fun}_{n+1}))\\
&\qquad\qquad\qquad \rho'' l_1\ldots l_n l_0 l_1\ldots l_n\eta l_0\sigma'
\end{aligned}
$$

that is

$$
\begin{aligned}
f &= \lambda\rho'.\mathit{function}_n\ P_{n\,n+2}(\mathcal{M}\ell_n[\![(ml)\ sl]\!], D_{n+1}(\mathit{fetch}, \mathit{release\text{-}fun}_{n+1}))\\
&\qquad\qquad (\mathit{ext}_{n+1}\ id_n\ldots id_1\,\mathrm{result}\,\rho')\\
&= B_1(\mathit{function}_n\ P_{n\,n+2}(\mathcal{M}\ell_n[\![(ml)\ sl]\!], D_{n+1}(\mathit{fetch}, \mathit{release\text{-}fun}_{n+1})),\\
&\qquad \mathit{ext}_{n+1}\ id_n\ldots id_1\,\mathrm{result})
\end{aligned}
$$

And then combining these results together with the definition of a block, we have:

$$
\begin{aligned}
&\mathcal{B}\ell[\![(\mathbf{block}\ (\mathbf{fun}\ id\,(id_1\ldots id_n)\,(ml)\ sl_1)\ sl_2)]\!]\\
&\quad = \lambda\rho\kappa.\mathcal{S}\ell[\![sl_2]\!]((\mathit{ext\text{-}fun}\ id\ B_1(\mathit{function}_n\ldots)\rho)\kappa\\
&\quad = B_1(\mathcal{S}\ell[\![sl_2]\!], \mathit{ext\text{-}fun}\ id\ B_1(\mathit{function}_n\ldots))
\end{aligned}
$$

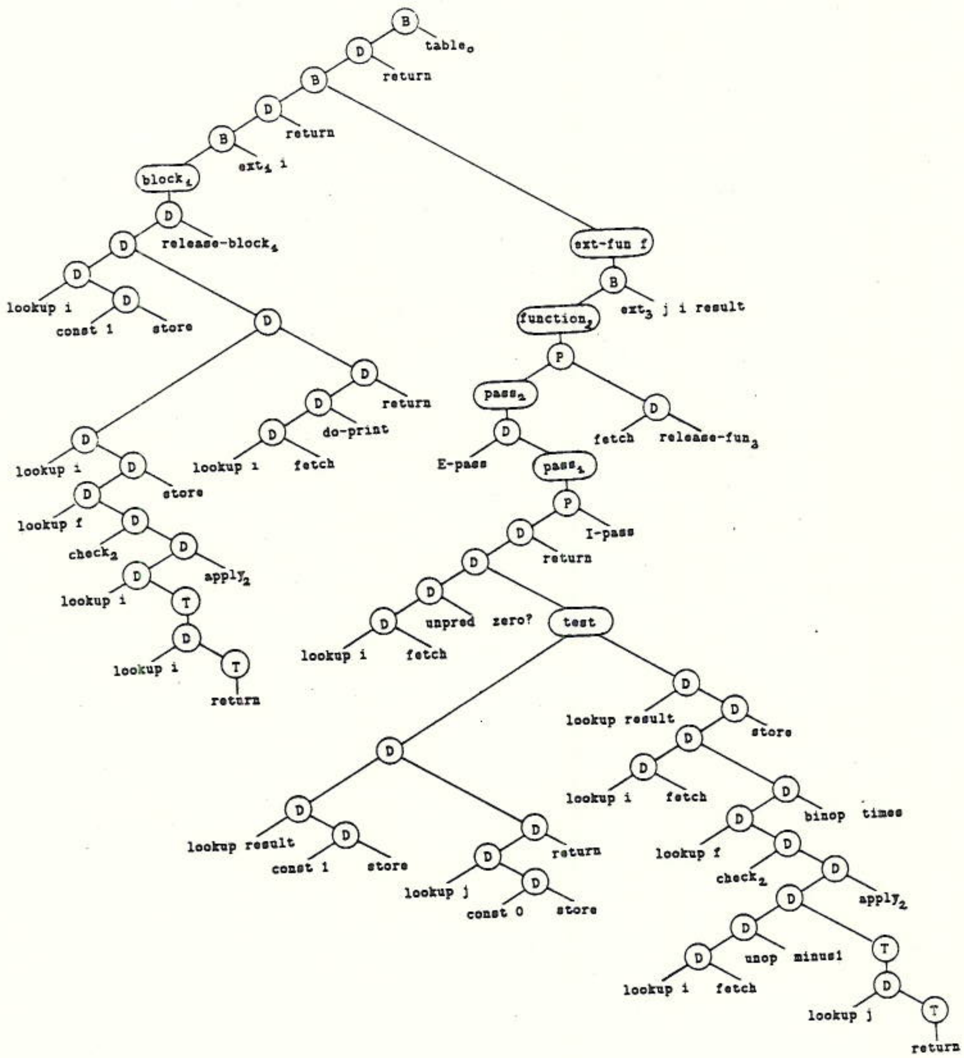

Figure 2. Example 2 – the "naive" code

The modifications of the equations for parameter passing are straightforward:

$$\begin{aligned}
&\mathcal{M}\ell_n[\![(im\ ml)\ sl]\!]\\
&\quad= \lambda\rho l_1\ldots l_n\kappa\sigma.\mathcal{M}_{in}[\![im]\!]\rho(\mathcal{M}\ell_{n-1}[\![(ml)\ sl]\!]\rho l_2\ldots l_n\kappa)l_1(\sigma_3\ l_1)\\
&\qquad\qquad\qquad\qquad\langle\sigma_1,\sigma_2,\sigma_3[\textit{'uninitialized'}/l_1]\rangle\\
&\quad= \lambda\rho l_1\ldots l_n\kappa\sigma.D_{n-1}(\mathcal{M}_{in}[\![im]\!],\mathcal{M}\ell_{n-1}[\![(ml)\ sl]\!])\rho l_2\ldots l_n\kappa l_1(\sigma_3\ l_1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\langle\sigma_1,\sigma_2,\sigma_3[\textit{'uninitialized'}/l_1]\rangle\\
&\quad= pass_n\ D_{n-1}(\mathcal{M}_{in}[\![im]\!],\mathcal{M}\ell_{n-1}[\![(ml)\ sl]\!])
\end{aligned}$$

and

$$
\begin{aligned}
&\mathcal{M}\ell_n[\![(om\ ml)\ sl]\!] \\
&\quad = \lambda\rho l_1 \ldots l_n \kappa\sigma.\mathcal{M}\ell_{n-1}[\![(ml)\ sl]\!]\rho l_2 \ldots l_n(\mathcal{M}_{out}[\![om]\!]\rho\kappa l_1(\sigma_3\ l_1)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle\sigma_1,\sigma_2,\sigma_3[\text{'uninitialized'}/l_1]\rangle \\
&\quad = \lambda\rho l_1 \ldots l_n \kappa\sigma.P_{n-1\,2}(\mathcal{M}\ell_{n-1}[\![(ml)\ sl]\!], \mathcal{M}_{out}[\![om]\!])\rho l_2 \ldots l_n\kappa l_1(\sigma_3\ l_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle\sigma_1,\sigma_2,\sigma_3[\text{'uninitialized'}/l_1]\rangle \\
&\quad = pass_n\ P_{n-1\,2}(\mathcal{M}\ell_{n-1}[\![(ml)\ sl]\!], \mathcal{M}_{out}[\![om]\!])
\end{aligned}
$$

The last transformation to be presented is the equation for computing actual parameters:

$$
\begin{aligned}
\mathcal{A}\ell_n[\![a\ al]\!] &= \lambda\rho\eta.\mathcal{A}[\![a]\!]\rho(\lambda v.\mathcal{A}\ell_{n-1}[\![al]\!]\rho(\eta\, v)) \\
&= D_0(\mathcal{A}[\![a]\!], (\lambda\rho\eta v.\mathcal{A}\ell_{n-1}[\![al]\!]\rho(\eta\, v)) \\
&= D_0(\mathcal{A}[\![a]\!], T(\mathcal{A}\ell_{n-1}[\![al]\!]))
\end{aligned}
$$

Now we may derive the first version of code – a "naive" code represented as a tree in which the internal nodes are labelled with B's, D's, T's, P's and some of the auxiliary functions. The leaves are labelled with actions, i.e. auxiliary functions like $lookup[\![id]\!]$, $fetch$, $ext_n\ id_n \ldots id_1$, and so on. We may omit almost all the indices since they can easily be computed from the information contained in the leaves. We keep indices only with such operations as $block_n$, $function_n$, $pass_n$, $check_n$ and $apply_n$.

The combinator trees are not linear yet. Therefore they are not acceptable for designing a target machine. Figure 1 shows the "naive" code for the program from Example 1; Figure 2 – for the program from Example 2.

4 Code rotation

We can rotate the "naive" code into an almost linear form using the following properties of the combinators:

Property 1. Right-associativity of D's (cf. [4])

$$D_k(D_p(\alpha,\beta),\gamma) = D_{k+p}(\alpha, D_k(\beta,\gamma))$$

Property 2. Elimination of P's

$$P_{n\,m}(P_{n\,r}(\alpha,\beta),\gamma) = P_{n,m+r}(\alpha, D_m(\beta,\gamma))$$

Proof:

$$
\begin{aligned}
&P_{n\,m}(P_{n\,r}(\alpha,\beta),\gamma)\rho l_1 \ldots l_n x_0 x_1 \ldots x_m x_{m+1} \ldots x_{m+r} \\
&= P_{n\,r}(\alpha,\beta)\rho l_1 \ldots l_n(\gamma\rho x_0 x_1 \ldots x_m)x_{m+1} \ldots x_{m+r} \\
&= \alpha\rho l_1 \ldots l_n(\beta\rho(\gamma\rho x_0 x_1 \ldots x_m)x_{m+1} \ldots x_{m+r}) \\
&= \alpha\rho l_1 \ldots l_n(D_m(\beta,\gamma)\rho x_0 x_1 \ldots x_m x_{m+1} \ldots x_{m+r}) \\
&= P_{n,m+r}(\alpha, D_m(\beta,\gamma))\rho l_1 \ldots l_n x_0 x_1 \ldots x_m x_{m+1} \ldots x_{m+r} \qquad \blacksquare
\end{aligned}
$$

Property 3. Sequencing P's and D's

$$P_{n\,m}(D_{n+r}(\alpha,\beta),\gamma) = D_{n+m+r}(\alpha, P_{n\,m}(\beta,\gamma))$$

Proof:

$$\begin{aligned}
&P_{n\,m}(D_{n+r}(\alpha,\beta),\gamma)\rho l_1 \dots l_n x_0 x_1 \dots x_m x_{m+1} \dots x_{m+r} \\
&= D_{n+r}(\alpha,\beta)\rho l_1 \dots l_n(\gamma\rho x_0 x_1 \dots x_m) x_{m+1} \dots x_{m+r} \\
&= \alpha\rho(\beta\rho l_1 \dots l_n(\gamma\rho x_0 x_1 \dots x_m) x_{m+1} \dots x_{m+r}) \\
&= \alpha\rho(P_{n\,m}(\beta,\gamma)\rho l_1 \dots l_n x_0 x_1 \dots x_m x_{m+1} \dots x_{m+r}) \\
&= D_{n+m+r}(\alpha, P_{n\,m}(\beta,\gamma))\rho l_1 \dots l_n x_0 x_1 \dots x_m x_{m+1} \dots x_{m+r} \quad \blacksquare
\end{aligned}$$

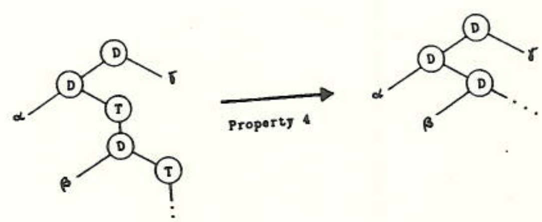

Figure 3. Elimination of T combinators

Property 4. Elimination of T's (due to Wand)

$$D_k(T(\alpha),\beta) = D_{k+1}(\alpha,\beta)$$

Proof:

$$\begin{aligned}
&D_k(T(\alpha),\beta)\rho x_0 x_1 \dots x_k x_{k+1} \\
&= T(\alpha)\rho(\beta\rho x_0 x_1 \dots x_k) x_{k+1} \\
&= \alpha\rho(\beta\rho x_0 x_1 \dots x_k x_{k+1}) \\
&= D_{k+1}(\alpha,\beta)\rho x_0 x_1 \dots x_k x_{k+1} \quad \blacksquare
\end{aligned}$$

Notice that Property 4 makes it possible to eliminate all of T's since they occur only in trees of the form shown in Fig. 3.

Property 5. Elimination of *return* (cf. [3])

$$D_k(\mathit{return},\gamma) = \gamma$$

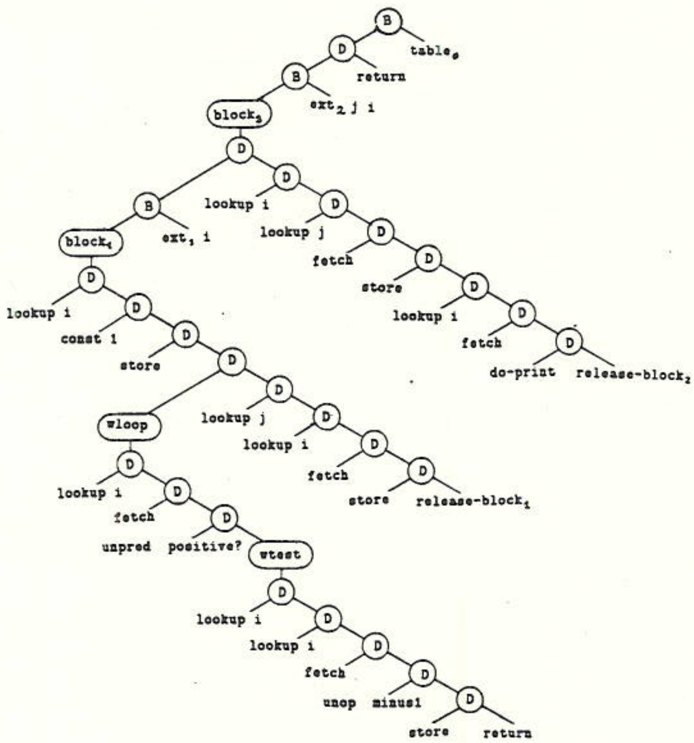

Figure 4. Example 1 – code after rotation

Property 6. Reducing indices at P's

$$P_{n\,m}(pass_{n+r}\ \alpha, \beta) = pass_{n+m+r}P_{n-1,m}(\alpha, \beta)$$

Proof:

$$\begin{aligned}
&P_{n\,m}(pass_{n+r}\ \alpha, \beta)\rho l_1 \ldots l_n x_0 x_1 \ldots x_m x_{m+1} \ldots x_{m+r}\sigma \\
&= pass_{n+r}(\alpha)\rho l_1 \ldots l_n(\beta\rho x_0 x_1 \ldots x_m)x_{m+1} \ldots x_{m+r}\sigma \\
&= \alpha\rho l_2 \ldots l_n(\beta\rho x_0 x_1 \ldots x_m)x_{m+1} \ldots x_{m+r}l_1(\sigma_3\, l_1)\langle\sigma_1, \sigma_2, \sigma_3[\text{'uninitialized'}/l_1]\rangle \\
&= P_{n-1,m}(\alpha, \beta)\rho l_2 \ldots l_n x_0 x_1 \ldots x_m x_{m+1} \ldots x_{m+r}l_1(\sigma_3\, l_1)\langle\sigma_1, \sigma_2, \sigma_3[\text{'uninitialized'}/l_1]\rangle \\
&= pass_{n+m+r}P_{n-1,m}(\alpha, \beta)\rho l_1 \ldots l_n x_0 x_1 \ldots x_m x_{m+1} \ldots x_{m+r}\sigma \quad \blacksquare
\end{aligned}$$

Since the value of n at $P_{n\,m}$ is equal to the number of *pass*'es following this P, then after pulling a P through all of D's and *pass*'es, the P becomes of the form $P_{0\,m} = D_m$ (by Properties 2, 3 and 6). Therefore it is possible to eliminate all P's from the code.

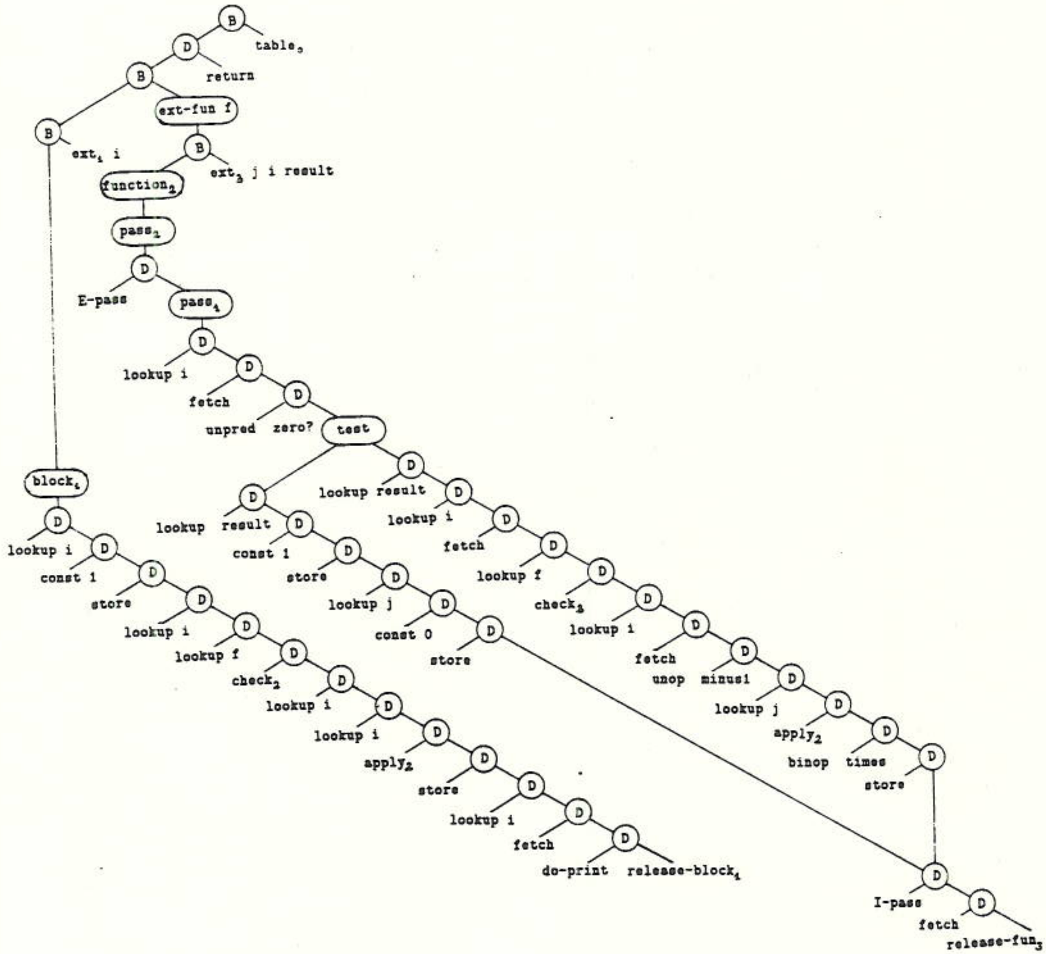

Figure 5. Example 2 – code after rotation

We may also sequence conditional statements with the rest of the code. We need, however, to introduce a new auxiliary function (cf. [6]):

$$test_k\ f\, g = \lambda \rho x_0 x_1 \ldots x_k \beta.\beta \rightarrow f \rho x_0 x_1 \ldots x_k, g \rho x_0 x_1 \ldots x_k$$

Notice that we could define $test_k$ by means of *test* and the *T* combinator. In that case, however, *T*'s would remain after rotation. Now we have:

Property 7. Sequencing *test*'s and the rest of the code (cf. [6])

$$D_k(test\ \alpha \beta, \gamma) = test_k\ (D_k(\alpha, \gamma), D_k(\beta, \gamma))$$

Of course, to avoid duplicating the code for γ, we represent the code after rotation as a dag instead of a tree (we may use the same code for both branches). We do not sequence the code of a **while** loop. This problem is discussed thoroughly in [6].

$rot[D\ [D\ \alpha\ \beta]\ \gamma] = rot[D\ \alpha[D\ \beta\ \gamma]]$	by Property 1
$rot[D\ [test\ \alpha\ \beta]\ \gamma] = [test\ rot[D\ \alpha\ \gamma]\ rot[D\ \beta\ \gamma]]$	by Property 7
$rot[D\ return\ \beta] = rot\beta$	by Property 5
$rot[D\ [T\ \alpha]\ \beta] = rot[D\ \alpha\ \beta]$	by Property 4
$rot[D\ \alpha\ \beta] = [D\ rot\alpha\ rot\beta]$	
$rot[D\ [P\ \alpha\ \beta]\ \gamma] = rot[D\ rot[P\ \alpha\ \beta]\ \gamma]$	
$rot[B\ \alpha\ \beta] = [B\ rot\alpha\ rot\beta]$	
$rot[wloop\ \alpha] = [wloop\ rot\alpha]$	
$rot[wtest\ \alpha] = [wtest\ rot\alpha]$	
$rot[block_n\ \alpha] = [block_n\ rot\alpha]$	
$rot[ext\text{-}fun\ id\ \alpha] = [ext\text{-}fun\ id\ rot\alpha]$	
$rot[P\ [P\ \alpha\ \beta]\ \gamma] = rot[P\ \alpha\ rot[D\ \beta\ \gamma]]$	by Property 2
$rot[P\ [D\ \alpha\ \beta]\ \gamma] = rot[D\ \alpha\ [P\ \beta\ \gamma]]$	by Property 3
$rot[P\ [pass\ \alpha]\ \beta] = [pass\ rot[P\ \alpha\ \beta]]$	by Property 6
$rot\alpha = \alpha$	for the rest (instructions)

Table 8. Structural definition of *rot*

Now we are able to define the function *rot* to rotate the "naive" code into an almost linear tree (dag). The definition is shown in Table 8. The rotated trees for Example 1 and Example 2 are of the form from Figures 4 and 5 respectively.

There are two rules that introduce double rotation of the code. This fact causes some problems like rotation of a tree which has been already rotated (corresponding rules are not shown in Table 8). One of these rules may be easily substituted by $rot[P\ [P\ \alpha\ \beta]\ \gamma] = rot[P\ \alpha[D\ \beta\ \gamma]]$. The new definition is equivalent to the old one since P becomes D at the end of rotation, and then we can rotate the result accordingly to Property 1. There is no such substitution for the rule $rot[D\ [P\ \alpha\ \beta]\ \gamma] = rot[D\ rot[P\ \alpha\ \beta]\ \gamma]$*. Therefore the code must still be rotated twice. It is possible, however, to restrict the double rotation to the code for parameter passing, and eliminate the P combinators before rotating the rest of the code.

5 Static scoping

In the same way as it has been done in [4], we may distribute the symbol table information to instructions. The combinator S is used to sequence instructions after this process:

$$S_{p\,n}(\alpha,\beta) = \lambda a_1 \ldots a_p \kappa x_1 \ldots x_n . \alpha a_1 \ldots a_p (\beta a_1 \ldots a_p \kappa x_1 \ldots x_n)$$

The a's correspond to the locations of variables visible in the environment, the x's create a local register file for evaluating expressions.

The symbol table information has been originally distributed by means of the B combinator. Here, however, we need to introduce a generalization of B to obtain the same goal:

$$A_{p\,i\,j}(\alpha,\beta) = \lambda a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j . \alpha(\beta a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j) c_1 \ldots c_j b_1 \ldots b_i c_1 \ldots c_j$$

All the λ-variables create a new display of locations; therefore they all are passed to the function defining the environment (β). The j top-most variables (c's) are locations of parameters to be passed. So they are passed as arguments to α, which probably contains

*We had not realized this difficulty until we were implementing this method in Scheme

$$
\begin{aligned}
&\mathit{test}_{pk}\ f\ g = \lambda a_1 \ldots a_p \kappa x_1 \ldots x_k \beta . \beta \rightarrow f a_1 \ldots a_p \kappa x_1 \ldots x_k, g a_1 \ldots a_p \kappa x_1 \ldots x_k \\
&\mathit{wloop}_p\ f = \lambda a_1 \ldots a_p \kappa . \mathit{fix}\,(f a_1 \ldots a_p \kappa) \\
&\mathit{wtest}_p\ f = \lambda a_1 \ldots a_p \kappa \theta \beta . \beta \rightarrow f a_1 \ldots a_p \theta, \kappa \\
&\mathit{block}_{pn}\ f = \lambda a_1 \ldots a_p \kappa \sigma . \mathit{let}\,(l_1, \ldots l_n) = \mathit{new}_n \sigma \text{ and } \sigma' = \mathit{adjoin}_n \sigma l_1 \ldots l_n \\
&\qquad\qquad\qquad\qquad\quad \text{in } f a_1 \ldots a_p l_1 \ldots l_n \kappa \sigma' \\
&\mathit{function}_{pn}\ f = \lambda a_1 \ldots a_p \eta x v_1 \ldots v_n \sigma . \mathit{let}\,(l_0, l_1, \ldots l_n) = \mathit{new}_{n+1} \sigma \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{and } \sigma' = \mathit{adjoin\text{-}init}_{n+1} \sigma l_0 l_1 \ldots l_n\, \text{`uninitialized'} v_1 \ldots v_n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } f a_1 \ldots a_p l_0 l_1 \ldots l_n \eta l_0 \\
&\mathit{apply}_{pn} = \lambda a_1 \ldots a_p \eta f v_1 \ldots v_n . f \eta f v_1 \ldots v_n \\
&\mathit{pass}_{p+mnm}\ f = \lambda a_1 \ldots a_p l_1 \ldots l_m \kappa x_1 \ldots x_n \sigma . f a_1 \ldots a_p l_1 \ldots l_m \kappa x_1 \ldots x_n l_1 (\sigma_3 l_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle \sigma_1, \sigma_2, \sigma_3[\text{`uninitialized'}/l_1] \rangle \\
&\mathit{release\text{-}block}_{p+nn} = \lambda a_1 \ldots a_p l_1 \ldots l_n \kappa \sigma . \kappa(\mathit{release}_n \sigma l_1 \ldots l_n) \\
&\mathit{release\text{-}fun}_{p+nn} = \lambda a_1 \ldots a_p l_1 \ldots l_n \eta v \sigma . \eta v(\mathit{release}_n \sigma l_1 \ldots l_n)
\end{aligned}
$$

Table 9. Auxiliaries for distributing the symbol table information

some of the *pass* instructions. The b's and c's together correspond to the newly created locations and they are passed to α as arguments for one of the *release* instructions. Notice that A_{p00} is equal to B_p.

Now we formulate some useful properties of these combinators:

Property 8. Associativity of B's (cf. [5])

$$B_p(B_1(\alpha, \beta), \gamma) = B_p(\alpha, B_p(\beta, \gamma))$$

Property 9. Distribution law for B (cf. [5])

$$B_p(D_m(\alpha, \beta), \gamma) = S_{pm}(B_p(\alpha, \gamma), B_p(\beta, \gamma))$$

Property 10. Distribution law for A

$$A_{pij}(D_{m+i+2j}(\alpha, \beta), \gamma) = S_{p+i+j,m}(B_{p+i+j}(\alpha, \gamma), A_{pij}(\beta, \gamma))$$

Proof:

$$
\begin{aligned}
&A_{pij}(D_{m+i+2j}(\alpha, \beta), \gamma) a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j x_0 x_1 \ldots x_m \\
&\quad = D_{m+i+2j}(\alpha, \beta)(\gamma a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j) c_1 \ldots c_j b_1 \ldots b_i c_1 \ldots c_j x_0 x_1 \ldots x_m \\
&\quad = \alpha(\gamma a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j) \\
&\qquad\quad (\beta(\gamma a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j) c_1 \ldots c_j b_1 \ldots b_i c_1 \ldots c_j x_0 x_1 \ldots x_m) \\
&\quad = B_{p+i+j}(\alpha, \gamma) a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j (A_{pij}(\beta, \gamma) a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j x_0 x_1 \ldots x_m) \\
&\quad = S_{p+i+j,m}(B_{p+i+j}(\alpha, \gamma), A_{pij}(\beta, \gamma)) a_1 \ldots a_p b_1 \ldots b_i c_1 \ldots c_j x_0 x_1 \ldots x_m \qquad \blacksquare
\end{aligned}
$$

Before we reformulate the instructions, notice that, by the above properties, the only instructions that are bound to the symbol table information by the A combinator instead of B, are *test*, *release-block*, *release-fun* and *pass*. And even more, *test* and *release*'s are bound by means of A_{pi0} (see the reformulation of *pass* below).

$$
\begin{aligned}
&A_{p\,i\,0}(test_k\ f\ g, \tau) = test_{p+i,k}\ A_{p\,i\,0}(f, \tau)\ A_{p\,i\,0}(g, \tau)\\
&A_{p\,i\,0}(release\text{-}block_i, \tau) = release\text{-}block_{p+i,i}\\
&A_{p\,i\,0}(release\text{-}fun_i, \tau) = release\text{-}fun_{p+i,i}\\
&A_{p\,i\,j}(pass_{2j+i+n}, \tau) = pass_{p+i+j,n,j}\ A_{p\,i+1\,j-1}(f, \tau)\\
&B_p(wloop\ f, \tau) = wloop_p\ B_p(f, \tau)\\
&B_p(wtest\ f, \tau) = wtest_p B_p(f, \tau)\\
&B_p(block_n\ f, \tau) = block_{p\,n}\ A_{p\,n\,0}(f, \tau)\\
&B_p(function'_n\ f, \tau) = function_{p\,n}\ A_{p\,1\,n}(f, \tau)\\
&B_p(apply'_n, \tau) = apply_{p\,n}
\end{aligned}
$$

Table 10. Distributing the symbol table information

Let us define the environment-creating functions first:

$$
\begin{aligned}
table_p\ id\ \tau &= \lambda a_1 \ldots a_p.(\lambda id'.\ id' = id \rightarrow a_p, \tau a_1 \ldots a_{p-i}\ id'\\
fun\text{-}table_p\ id\ f\ \tau &= \lambda a_1 \ldots a_p.(\lambda id'.\ id' = id \rightarrow f a_1 \ldots a_p, \tau a_1 \ldots a_p\ id')
\end{aligned}
$$

By means of these functions, we have:

$$
\begin{aligned}
B_p(ext_n\ id_n \ldots id_1, \tau) &= table_{p+n}\ id_n \ldots (table_{p+1}\ id_1\ \tau) \ldots\\
B_p(ext\text{-}fun\ id\ f, \tau) &= fix\ (\lambda \tau'.fun\text{-}table_p\ id\ B_p(f, \tau')\ \tau)
\end{aligned}
$$

We use the definitions of the environment-creating functions to define the value of *lookup* only. And because of the static scoping, we may define other functions that give us the same answer and do not require the environment to be passed during run-time. These are:

$$
\begin{aligned}
selec_p\, j &= \lambda a_1 \ldots a_p \eta . \eta a_j\\
mk\text{-}fun_{p\,j} f &= \lambda a_1 \ldots a_p \eta . \eta (f a_1 \ldots a_j)
\end{aligned}
$$

Now we may define the value of $lookup[\![id]\!]$ with the symbol table information τ, given as a term defining τ, in the following way.

Let $j = \max\{k \mid \tau = \ldots(table_k\ id\ldots)\ldots$ or $\tau = \ldots(fun\text{-}table_k\ id\ f\ldots)\ldots\}$
Then:

- if $\tau = \ldots(table_j\ id\ldots)\ldots$ and $\tau \neq \ldots(fun\text{-}table_j\ id\ldots)\ldots$ and, in addition, id describes a regular variable, **value** or **result** parameter, then

$$
\begin{aligned}
B_p(lookup[\![id]\!], \tau) a_1 \ldots a_p \eta &= \eta(\tau a_1 \ldots a_p\ id) = \eta a_j\\
&= selec_p\, j\ a_1 \ldots a_p \eta
\end{aligned}
$$

so

$$B_p(lookup[\![id]\!], \tau) = selec_p\, j$$

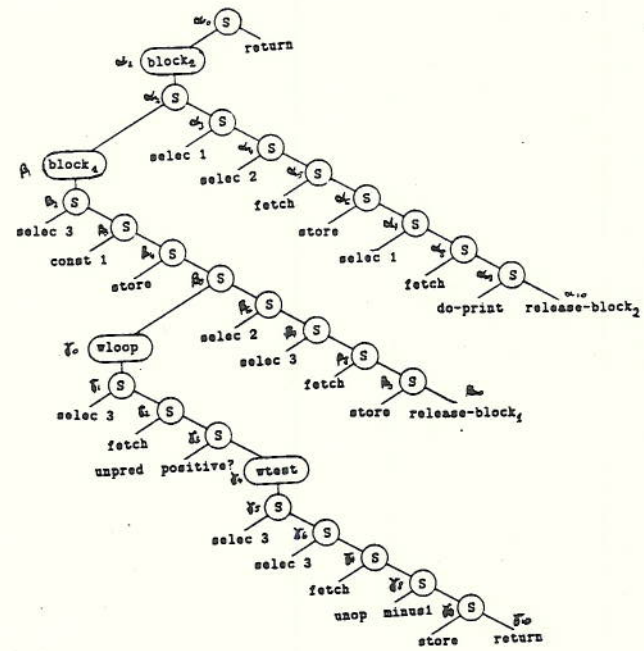

Figure 6. Example 1 – code after distribution

– if τ is defined as in the previous case, but *id* describes a **fun** or **var** parameter, then

$$\begin{aligned}
&S_{p\,n}(B_p(lookup[\![id]\!], \tau), \alpha)a_1 \ldots a_p\kappa x_1 \ldots x_n\sigma \\
&\quad = lookup[\![id]\!](\tau\, a_1 \ldots a_p)(\alpha a_1 \ldots a_p\kappa x_1 \ldots x_n)\sigma \\
&\quad = (\alpha a_1 \ldots a_p\kappa x_1 \ldots x_n)(\sigma_3\, a_j)\sigma \\
&\quad = fetch(\tau\, a_1 \ldots a_p)(\alpha a_1 \ldots a_p\kappa x_1 \ldots x_n)a_j\sigma \\
&\quad = S_{p\,n}(fetch_p, \alpha)a_1 \ldots a_p\kappa x_1 \ldots x_n a_j\sigma \\
&\quad = selec_p\, j\, a_1 \ldots a_p(S_{p\,n}(fetch_p, \alpha)a_1 \ldots a_p\kappa x_1 \ldots x_n)\sigma \\
&\quad = S_{p\,n}(selec_p\, j, S_{p\,n}(fetch_p, \alpha))a_1 \ldots a_p\kappa x_1 \ldots x_n\sigma
\end{aligned}$$

So in this case

$$S_{p\,n}(B_p(lookup[\![id]\!], \tau), \alpha) = S_{p\,n}(selec_p\, j, S_{p\,n}(fetch_p, \alpha))$$

– if $\tau = \ldots(\textit{fun-table}_j\ id\ f \ldots)\ldots$, i.e. *id* denotes a regular function, then

$$\begin{aligned}
B_p(lookup[\![id]\!], \tau)a_1 \ldots a_p\eta &= \eta(\tau a_1 \ldots a_p\, id) = \eta(f\, a_1 \ldots a_j) \\
&= \textit{mk-fun}_{p\,j}\ f\, a_1 \ldots a_p\eta
\end{aligned}$$

so

$$B_p(lookup[\![id]\!], \tau) = \textit{mk-fun}_{p\,j}\ f$$

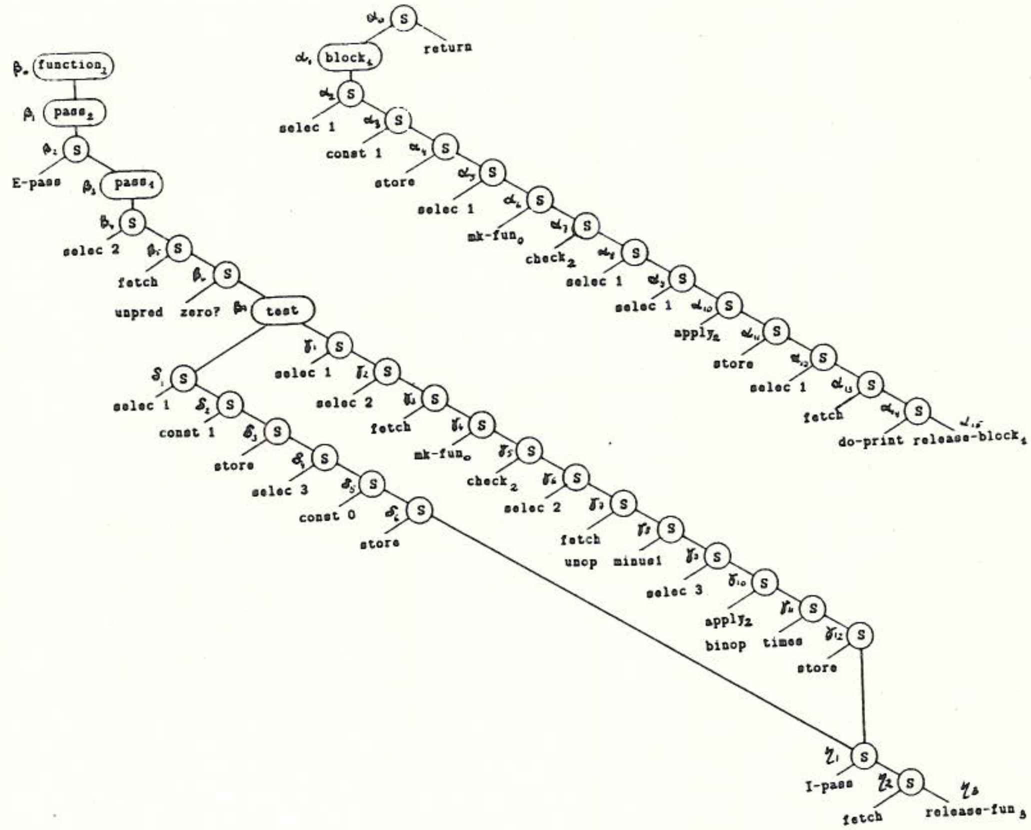

Figure 7. Example 2 – code after distribution

Now we may reformulate the basic instructions such as *store, fetch, const, do-print, L-pass* etc. The distribution of the symbol table information reaches these instructions as $B_p(\ldots,\tau)$. Since they ignore the information passed by τ, we may redefine all of them in the following way:

$$B_p(store,\tau) = store_p \qquad \text{for all } \tau\text{'s}$$

The other auxiliaries are also straightforward. Their new definitions are shown in Table 9. By these definitions we obtain the properties of the distribution collected in Table 10. For the purpose of an easier implementation of a target machine (see Section 6), we introduce an additional parameter to a function call. This parameter is ignored by the function; it is, however, removed from the stack for computing an expression. So we redefine the auxiliaries from the previous sections:

$$function'_n\ f\ env = \lambda\eta x.function_n\ f\ env\ \eta$$
$$apply'_n = \lambda\rho\eta f v_1\ldots v_n.f\eta f v_1\ldots v_n$$

and then we have the additional properties in Table 10.

Now we can define the function *distr* to distribute the symbol table information to the instructions, according to the presented rules. The definition is simple and therefore it is omitted. Then the compiler for the language is a function $compile[\![p]\!] = distr[B\ rot[D\ \mathcal{B}\ell[\![\mathrm{p}]\!]\ return]\ table_0]$. Figures 6 and 7 show the codes for Example 1 and 2 after the distribution.

6 Simple display machine

The machine that interpretes the code generated by the compiler consists of:

- the instruction sequence to be interpreted; let us denote the set of all instructions by *Ins*,
- the display, i.e. the locations of variables in the environment,
- the continuation register; let us denote all continuations by $\overline{K}$,
- the local register file for evaluating expressions.

The values stored in the local register file are:

- storable values when they result from evaluating expressions and actual parameters,
- truth values as results of evaluating boolean expressions,
- instruction sequences for implementing loops (another solution is presented in [6]).

So the local register file is built of elements from $W = V + B + Ins$. It is of the form of W^n.

The instructions and the instruction sequences to be interpreted are of the following functionality:

$$Ins_{p\,n} = L^p \to \overline{K} \to W^n \to S \to A$$

So $Ins = \bigcup Ins_{p\,n}$. Then, if Rep_α denotes the machine representations of elements from the domain α, and $Rep_{p\,n}$ denote the representations of $Ins_{p\,n}$, we may define the target machine – a function that interpretes instructions (cf. [5]):

$$M_{p\,n} : Rep_{p\,n} \to Rep_L^n \to Rep_{\overline{K}} \to Rep_W^n \to S \to A$$

Let us denote by $retpt_{p\,n}\beta a_1 \ldots a_p \kappa x_1 \ldots x_n = \beta a_1 \ldots a_p \kappa x_1 \ldots x_n$, i.e. the continuation to execute β with the given arguments. The interpretation of a program starts as $M_{00}\beta$ *init-cont* σ_0, where β is the instruction sequence of the program and σ_0 is the initial state.

Table 11 presents the interpretation of basic instructions. Here we show some of the more complicated. The "$\check{\ }$" operation is the abstraction function from representations to "real" values (cf. [5]).

$$
\begin{array}{lll}
M_{pn}[S\ \mathit{store}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma & & \\
\quad = M_{p\,n-2}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-2}\sigma' & \sigma' = \langle \sigma_1, \sigma_2, \sigma_3[x_n/x_{n-1}]\rangle & \\
M_{pn}[S\ \mathit{do\text{-}read}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma & & \\
\quad = M_{p\,n+1}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_n x_{n+1}\sigma' & x_{n+1} = \mathit{first}\,\sigma_1,\ \sigma' = \langle \mathit{rest}\,\sigma_1, \sigma_2, \sigma_3\rangle & \\
M_{pn}[S\ \mathit{do\text{-}print}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma & & \\
\quad = M_{p\,n-1}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-1}\sigma' & \sigma' = \langle \sigma_1, \sigma_2 \,\|\, x_n, \sigma_3\rangle & \\
\multicolumn{2}{l}{M_{pn}[S\ \mathit{fetch}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma = M_{pn}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-1}(\sigma_3\, x_n)\sigma} & \\
\multicolumn{2}{l}{M_{pn}[S\ [\mathit{const}\ i]\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma = M_{p\,n+1}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_n i \sigma} & \\
\multicolumn{2}{l}{M_{pn}[S\ [\mathit{selec}\ j]\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma = M_{pn}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_n a_j \sigma} & \\
\multicolumn{2}{l}{M_{pn}[S\ [\mathit{binop} \oplus]\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma = M_{p\,n-1}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-2} x \sigma} & x = \oplus x_{n-1} x_n \\
\multicolumn{2}{l}{M_{pn}[S\ [\mathit{unop} \ominus]\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma = M_{pn}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-1} x \sigma} & x = \ominus x_n \\
\multicolumn{2}{l}{M_{pn}[S\ [\mathit{binpred} \prec]\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma = M_{p\,n-1}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-2} x \sigma} & x = \prec x_{n-1} x_n \\
\multicolumn{2}{l}{M_{pn}[S\ [\mathit{unpred}\ ?]\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma = M_{pn}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-1} x \sigma} & x = ?x_n \\
\multicolumn{2}{l}{M_{p0}\,\mathit{return}\ a_1 \ldots a_p(\mathit{retpt}_{pn}\ \alpha a_1 \ldots a_p \kappa x_1 \ldots x_n)\sigma = M_{pn}\ \alpha a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma} & \\
\multicolumn{2}{l}{M_{00}\,\mathit{return}\ \mathit{init\text{-}cont}\ \sigma = \mathit{init\text{-}cont}\ \sigma = \sigma_2 \,\|\, \text{“normal termination”}} & \\
M_{pn}[S\ \mathit{L\text{-}pass}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma & & \\
\quad = \begin{cases} \sigma_2 \,\|\, \text{“not a variable passed”} \\ M_{p\,n-2}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-2}\sigma' \end{cases} & \begin{array}{l} \text{if } x_n \not\in L \\ \sigma' = \langle \sigma_1, \sigma_2, \sigma_3[x_n/x_{n-1}] \end{array} & \\
M_{pn}[S\ \mathit{E\text{-}pass}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma & & \\
\quad = \begin{cases} \sigma_2 \,\|\, \text{“not an expression passed”} \\ M_{p\,n-2}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-2}\sigma' \end{cases} & \begin{array}{l} \text{if } x_n \in F \\ \sigma' = \begin{cases} \langle \sigma_1, \sigma_2, \sigma_3[x_n/x_{n-1}]\rangle \\ \langle \sigma_1, \sigma_2, \sigma_3[(\sigma_3\ x_n)/x_{n-1}]\rangle \end{cases} \end{array} & \begin{array}{l} \\ \text{if } x_n \in E \\ \text{if } x_n \in L \end{array} \\
M_{pn}[S\ \mathit{F\text{-}pass}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma & & \\
\quad = \begin{cases} \sigma_2 \,\|\, \text{“not a function passed”} \\ M_{p\,n-2}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-2}\sigma' \end{cases} & \begin{array}{l} \text{if } x_n \not\in F \\ \sigma' = \langle \sigma_1, \sigma_2, \sigma_3[x_n/x_{n-1}]\rangle \end{array} & \\
M_{pn}[S\ \mathit{I\text{-}pass}\ \alpha]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma & & \\
\quad = \begin{cases} \sigma_2 \,\|\, \text{“not a variable passed for result”} \\ M_{p\,n-2}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-2}\sigma' \end{cases} & \begin{array}{l} \text{if } x_n \not\in L \\ \sigma' = \langle \sigma_1, \sigma_2, \sigma_3[(\sigma_3\ x_{n-1})/x_n]\rangle \end{array} &
\end{array}
$$

Table 11. Evaluation of instructions

$$
\begin{aligned}
&M_{pn}[S\ [\mathit{block}_m\ \alpha]\beta]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma \\
&\quad = \mathit{block}_{p\,m}\check{\alpha}\check{a}_1 \ldots \check{a}_p(\check{\beta}\check{a}_1 \ldots \check{a}_p\check{\kappa}\check{x}_1 \ldots \check{x}_n)\sigma \\
&\quad = \check{\alpha}\check{a}_1 \ldots \check{a}_p\check{a}_{p+1} \ldots \check{a}_{p+m}(\check{\beta}\check{a}_1 \ldots \check{a}_p\check{\kappa}\check{x}_1 \ldots \check{x}_n)\sigma' \\
&\quad = M_{p+m\,0}\alpha a_1 \ldots a_p a_{p+1} \ldots a_{p+m}(\mathit{retpt}_{pn}\beta a_1 \ldots a_p \kappa x_1 \ldots x_n)\sigma' \\
&\quad\ \text{where: } (a_{p+1}, \ldots a_{p+m}) = \mathit{new}_m\sigma,\ \sigma' = \mathit{adjoin}_m\sigma a_{p+1} \ldots a_{p+m}
\end{aligned}
$$

$$
\begin{aligned}
&M_{pn}[\mathit{test}_{n-1}\ \alpha\ \beta]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma \\
&\quad = \mathit{test}_{p\,n-1}\check{\alpha}\check{\beta}\check{a}_1 \ldots \check{a}_p\check{x}_1 \ldots \check{x}_n\sigma \\
&\quad = \begin{cases} \check{\alpha}\check{a}_1 \ldots \check{a}_p\check{\kappa}\check{x}_1 \ldots \check{x}_{n-1}\sigma & \text{if } \check{x}_n = \check{\mathit{true}} \\ \check{\beta}\check{a}_1 \ldots \check{a}_p\check{\kappa}\check{x}_1 \ldots \check{x}_{n-1}\sigma & \text{if } \check{x}_n = \check{\mathit{false}} \end{cases} \\
&\quad = \begin{cases} M_{p\,n-1}\alpha a_1 \ldots a_p \kappa x_1 \ldots x_{n-1}\sigma & \text{if } x_n = \mathit{true} \\ M_{p\,n-1}\beta a_1 \ldots a_p \kappa x_1 \ldots x_{n-1}\sigma & \text{if } x_n = \mathit{false} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&M_{pn}[S\ [\mathit{wloop}\ \alpha]\ \beta]a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma \\
&\quad = M_{p0}[\mathit{wloop}\ \alpha]a_1 \ldots a_p(\mathit{retpt}_{pn}\ \beta a_1 \ldots a_p \kappa x_1 \ldots x_n)\sigma
\end{aligned}
$$

$$
\begin{aligned}
&M_{p0}[\mathit{wloop}\ \alpha]a_1\dots a_p\kappa\sigma\\
&\quad= \check{\alpha}\check{a}_1\dots\check{a}_p\check{\kappa}[\mathit{wloop}\ \alpha]\sigma\\
&\quad= M_{p1}\alpha a_1\dots a_p\kappa[\mathit{wloop}\ \alpha]
\end{aligned}
$$

$$
\begin{aligned}
&M_{p2}[\mathit{wtest}\ \alpha]a_1\dots a_p(\mathit{retpt}_{pn}\beta a_1\dots a_p\kappa x_1\dots x_n)\theta x\sigma\\
&\quad= \begin{cases} M_{p0}\alpha a_1\dots a_p(\mathit{retpt}_{p0}\theta a_1\dots a_p(\mathit{retpt}_{pn}\beta a_1\dots a_p x_1\dots x_n))\sigma & \text{if } x = \mathit{true}\\ M_{pn}\beta a_1\dots a_p\kappa x_1\dots x_n\sigma & \text{if } x = \mathit{false}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&M_{p+i,0}\ \textit{release-block}_i\ a_1\dots a_p l_1\dots l_i(\mathit{retpt}_{pn}\ \alpha a_1\dots a_p\kappa x_1\dots x_n)\sigma\\
&\quad= M_{pn}\ \alpha a_1\dots a_p\kappa x_1\dots x_n\sigma'\\
&\qquad \text{where } \sigma' = \langle\sigma_1,\sigma_2,\sigma_3[\text{‘}\mathit{unused}\text{'}/l_1]\dots[\text{‘}\mathit{unused}\text{'}/l_i]\rangle
\end{aligned}
$$

$$
\begin{aligned}
&M_{p+i,1}\ \textit{release-fun}_i\ a_1\dots a_p l_1\dots l_i(\mathit{retpt}_{mn}\ \alpha b_1\dots b_m\kappa x_1\dots x_n)x_{n+1}\sigma\\
&\quad= M_{m,n+1}\ \alpha b_1\dots b_m\kappa x_1\dots x_n x_{n+1}\sigma'\\
&\qquad \text{where } \sigma' = \langle\sigma_1,\sigma_2,\sigma_3[\text{‘}\mathit{unused}\text{'}/l_1]\dots[\text{‘}\mathit{unused}\text{'}/l_i]\rangle
\end{aligned}
$$

Notice that *release-fun* is the only instruction that changes all the elements in the display. Notice also that the memory may be arranged as a stack. The *release* instructions release the top-most elements in the display, that is those elements, which have been created after all other elements in the display. The locations stored in *retpt* continuations have been also created before the locations being released (see interpretation of *block* and *apply*).

$$
\begin{aligned}
&M_{p+n,m}[\mathit{pass}_n\ \alpha]a_1\dots a_p l_1\dots l_n\kappa x_1\dots x_m\sigma\\
&\quad= \mathit{pass}_{p+n,m,n}\ \check{\alpha}\check{a}_1\dots\check{a}_p\check{l}_1\dots\check{l}_n\check{\kappa}\check{x}_1\dots\check{x}_m\sigma\\
&\quad= \check{\alpha}\check{a}_1\dots\check{a}_p\check{l}_1\dots\check{l}_n\check{\kappa}\check{x}_1\dots\check{x}_m\check{l}_1(\check{\sigma_3}\ l_1)\sigma'\\
&\quad= M_{p+n,m+2}\ \alpha a_1\dots a_p l_1\dots l_n\kappa x_1\dots x_m l_1(\sigma_3\ l_1)\sigma'\\
&\qquad \text{where } \sigma' = \langle\sigma_1,\sigma_2,\sigma_3[\text{‘}\mathit{uninitialized}\text{'}/l_1]\rangle
\end{aligned}
$$

The only decision that still has to be made is the representation of a function value. We will let it be of the following form: $\langle \mathit{function}_{pn}\ \alpha, a_1, \dots a_p\rangle$. A function value is created while *mk-fun* is interpreted. Such a value is stored in the local register file and in local variables for **fun** parameters. The interpretation is as follows:

$$
\begin{aligned}
&M_{pn}[S[\textit{mk-fun}_j\ \langle\mathit{function}_{jm}\ \alpha\rangle]\ \beta]a_1\dots a_p\kappa x_1\dots x_n\sigma\\
&\quad= \textit{mk-fun}_{pj}\ \langle\check{\mathit{function}}_{jm}\ \alpha\rangle\check{a}_1\dots\check{a}_p(\check{\beta}\check{a}_1\dots\check{a}_p\check{\kappa}\check{x}_1\dots\check{x}_n)\sigma\\
&\quad= M_{p,n+1}\beta a_1\dots a_p\kappa x_1\dots x_n\langle\mathit{function}_{jm}\ \alpha, a_1,\dots a_j\rangle\sigma
\end{aligned}
$$

$$
\begin{aligned}
&M_{pn}[S\ \mathit{check}_m\ \alpha]a_1\dots a_p\kappa x_1\dots x_{n-1}\langle\mathit{function}_{jr}\ \beta, b_1,\dots b_j\rangle\sigma\\
&\quad= \begin{cases} \sigma_3 \parallel \text{“}\mathit{wrong\ number\ of\ arguments}\text{”} & \text{if } r \neq m\\ M_{pn}\alpha a_1\dots a_p\kappa x_1\dots x_{n-1}\langle\mathit{function}_{jr}\ \beta, b_1,\dots b_j\rangle\sigma & \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&M_{p,n+1+m}[S\ apply_m\ \alpha]a_1\ldots a_p\kappa x_1\ldots x_n\langle function_{jm}\ \beta, b_1,\ldots b_j\rangle v_1\ldots v_m\sigma\\
&\quad = apply_{pm}\check{a}_1\ldots\check{a}_p(retpt_{pn}\ \check{\alpha}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n)\langle \check{function}_{jm}\ \beta,\check{b}_1,\ldots\check{b}_j\rangle\check{v}_1\ldots\check{v}_m\sigma\\
&\quad = \langle \check{function}_{jm}\ \beta,\check{b}_1,\ldots\check{b}_j\rangle(retpt_{pn}\ \check{\alpha}\check{a}_1\ldots\check{a}_p\check{\kappa}\check{x}_1\ldots\check{x}_n)\langle \check{function}\ldots\rangle\check{v}_1\ldots\check{v}_m\sigma\\
&\quad = M_{j,m+1}\langle function_{jm}\ \beta\rangle b_1\ldots b_j(retpt_{pn}\ \alpha a_1\ldots a_p\kappa x_1\ldots x_n)\langle function\ldots\rangle v_1\ldots v_m\sigma
\end{aligned}
$$

$$
\begin{aligned}
&M_{p,n+1}\langle function_{pm}\ \alpha\rangle a_1\ldots a_p\kappa\langle function\ldots\rangle v_1\ldots v_n\sigma\\
&\quad = \check{\alpha}\check{a}_1\ldots\check{a}_p\check{a}_{p+1}\ldots\check{a}_{p+n+1}\check{\kappa}\check{a}_{p+1}\sigma'\\
&\quad = M_{p+n+1,1}\ \alpha a_1\ldots a_p a_{p+1}\ldots a_{p+n+1}\kappa a_{p+1}\sigma'\\
&\qquad \text{where } (a_{p+1},\ldots a_{p+n+1}) = new_{n+1}\sigma\\
&\qquad\quad \text{and } \sigma' = adjoin\text{-}init_{n+1}\sigma a_{p+1}\ldots a_{p+n+1}\ \text{`}uninitialized\text{'}v_1\ldots v_n
\end{aligned}
$$

Appendix A shows the interpretation of the code for Example 1, Appendix B for Example 2.

7 Standard display machine

We would like to obtain the standard display machine, discussed by Dijkstra in [1], as a result of our transformations. In the standard machine, locations of variables from a block or a function form a sequence of consecutive memory cells. The memory is ordered and there exists a function, *succ*, that for a given location returns the following one. Then a location of a variable can be expressed as a pair: the base, which is the location of the memory block, and the offset, the number of *succ*'s to be performed on the base in order to get the location corresponding to the variable. The display vector is shorter than that presented in the previous sections; it consists only of the bases of all visible blocks.

To be able to get the standard machine by transforming semantic equations for the language, we must already change the equations and introduce the block structure of the memory. It seems strange at first that this "representation" decision must be made when the language is being designed. It is, however, quite natural for languages with class objects, coroutines (Simula 67, Loglan), or pointers to dynamically created records (Pascal). Such languages introduce the notion of objects in their semantics already. Fortunately, this change does not require redefining the entire semantics of our language. The necessary changes are shown in Table 12. Notice the difference in indices of $\mathcal{M}\ell$ from Section 2 and $\mathcal{C}\ell$ here. The index of $\mathcal{M}\ell$ previously indicated the number of parameters still to be passed. The index of $\mathcal{C}\ell$ denotes the (consecutive) number of the parameter being passed, so it indicates the offset of this parameter in the memory block created for the function.

Table 13 shows the new combinatorial forms of the changed semantic equations. We need to replace P combinators by new ones; let us call them NP and define as:

$$NP_n(\alpha,\beta) = \lambda\rho l\kappa x_1\ldots x_n.\alpha\rho l(\beta\rho l\kappa x_1\ldots x_n)$$

They have the following properties:

Continuations

$Venvc = Env \to L \to K$ variable declaration continuations
$Fenvc = Env \to K$ function declaration continuations

Meaning functions

$\mathcal{D}_v : \langle Var\text{-}dec \rangle \to Env \to Venvc \to K$
$\mathcal{D}_f : \langle Fun\text{-}dec \rangle \to Env \to Fenvc \to K$
$\mathcal{C}\ell_n : \langle Mode\text{-}list\text{-}body \rangle \to Env \to L \to K$

Equations

$\mathcal{B}\ell[\![(\mathbf{block}\ (\mathbf{var}\ id_1 \dots id_n)\ sl)]\!] = \lambda\rho\kappa.\mathcal{D}_v[\![(\mathbf{var}\ id_1 \dots id_n)]\!]\rho(\lambda\rho' l.\mathcal{S}\ell[\![sl]\!]\rho'(release\text{-}block_n\, \rho' l \kappa))$
$\mathcal{B}\ell[\![(\mathbf{block}\ (\mathbf{fun}\ \alpha)\ sl)]\!] = \lambda\rho\kappa.\mathcal{D}_f[\![(\mathbf{fun}\ \alpha)]\!]\rho(\lambda\rho'.\mathcal{S}\ell[\![sl]\!]\rho'\kappa)$
$\mathcal{D}_v[\![(\mathbf{var}\ id_1 \dots id_n)]\!] = \lambda\rho\chi\sigma.\ let\ l = new_n\sigma\ and\ \sigma' = adjoin_n\sigma l$
$\quad and\ \rho' = \rho[l/id_1][(succ\, l)/id_2]\dots[(succ^{n-1}\, l)/id_{n-1}]$
$\quad in\ \chi\rho' l\sigma'$
$\mathcal{D}_f[\![(\mathbf{fun}\ id\ (id_1 \dots id_n)\ (ml)\ sl)]\!] = \lambda\rho\chi.\chi(fix\ (\lambda\rho'.\rho[(f\rho')/id]))$
where f is:
$f = \lambda\rho'\eta x v_1 \dots v_n\sigma.\ let\ l = new_{n+1}\sigma\ and\ \rho'' = \rho'[l/\mathrm{result}][(succ\, l)/id_1]\dots[(succ^n\, l)/id_n]$
$\quad and\ \sigma' = adjoin\text{-}init_{n+1}\sigma l\ \text{'}uninitialized\text{'} v_1 \dots v_n$
$\quad in\ \mathcal{C}\ell_1[\![(ml)\ sl]\!]\rho'' l(fetch\, \rho''(release\text{-}fun_{n+1}\, \rho'' l\eta)l)\sigma'$
$\mathcal{C}\ell_n[\![(empty)\ sl]\!] = body\ \mathcal{S}\ell[\![sl]\!]$
$\mathcal{C}\ell_n[\![(im\ ml)\ sl]\!] = \lambda\rho l\kappa\sigma.\ let\ l' = succ^n\, l$
$\quad in\ \mathcal{M}_{in}[\![im]\!]\rho(\mathcal{C}\ell_{n+1}[\![(ml)\ sl]\!]\rho l\kappa)l'(\sigma_3\, l')\langle\sigma_1, \sigma_2, \sigma_3[\text{'}uninitialized\text{'}/l']\rangle$
$\mathcal{C}\ell_n[\![(om\ ml)\ sl]\!] = \lambda\rho l\kappa\sigma.\ let\ l' = succ^n\, l$
$\quad in\ \mathcal{C}\ell_{n+1}[\![(ml)\ sl]\!]\rho l(\mathcal{M}_{out}[\![om]\!]\rho\kappa l'(\sigma_3\, l'))\langle\sigma_1, \sigma_2, \sigma_3[\text{'}uninitialized\text{'}/l']\rangle$

Auxiliaries

$body\, f = \lambda\rho l.f\rho$
$new_n\sigma =$ some l such that $(\sigma_3\, l), (\sigma_3(succ\, l)), \dots (\sigma_3(succ^{n-1}\, l))$ are 'unused'
$adjoin_n\, \sigma l = \langle\sigma_1, \sigma_2, \sigma_3[\text{'}uninitialized\text{'}/l][\text{'}uninitialized\text{'}/(succ\, l)]\dots[\text{'}uninitialized\text{'}/(succ^{n-1}\, l)]\rangle$
$adjoin\text{-}init_n\, \sigma l v_1 \dots v_n = \langle\sigma_1, \sigma_2, \sigma_3[v_1/l]\dots[v_n/(succ^{n-1}\, l)]\rangle$
$release\text{-}block_n = \lambda\rho l\kappa\sigma.\kappa\langle\sigma_1, \sigma_2, \sigma_3[\text{'}unused\text{'}/l]\dots[\text{'}unused\text{'}/(succ^{n-1}\, l)]\rangle$
$release\text{-}fun_n = \lambda\rho l\eta v\sigma.\eta v\langle\sigma_1, \sigma_2, \sigma_3[\text{'}unused\text{'}/l]\dots[\text{'}unused\text{'}/(succ^{n-1}\, l)]\rangle$

Table 12. Standard display – modified semantics

Property 11.

(1) $NP_n(NP_r(\alpha,\beta),\gamma) = NP_{n+r}(\alpha, NP_n(\beta,\gamma))$
(2) $NP_n(D_r(\alpha,\beta),\gamma) = D_{n+r}(\alpha, NP_n(\beta,\gamma))$
(3) $NP_n(pass_{0\,m}\ \alpha,\beta) = pass_{n\,m}\ NP_n(\alpha,\beta)$
(4) $NP_n(body\ \alpha,\beta) = D_{n+1}(\alpha,\beta)$

In the previous sections we have standardized the equations to take the environment as a parameter even when it has not been necessary. We could do almost the same now and add the base of the current block to all semantic functions. Then we would not need D combinators at all. We do not, however, demand this standardization; notice that after rotating, NP combinators do not occur in combinator trees (by Property 11(4)).

To distribute the symbol table information to instructions, we need still another modification of B combinators. The new combinators are simpler than A's and may be defined as:

$$NA_p(\alpha,\beta) = \lambda a_1 \dots a_p.\alpha(\beta a_1 \dots a_p)a_p$$

Auxiliaries

$\mathit{pass}_{m\,n}\ f \quad = \lambda\rho l\kappa x_1 \ldots x_m\sigma.\ \mathit{let}\ l' = \mathit{succ}^n\, l$
$\qquad \mathit{in}\ f\rho l\kappa x_1 \ldots x_m l'(\sigma_3 l')\langle\sigma_1, \sigma_2, \sigma_3[\text{'uninitialized'}/l']\rangle$
$\mathit{block}_n\ f\ \mathit{env} \quad = \lambda\kappa\sigma.\ \mathit{let}\ l = \mathit{new}_n\sigma\ \mathit{and}\ \sigma' = \mathit{adjoin}_n\sigma l$
$\qquad \mathit{in}\ f(\mathit{env}\, l)l\kappa\sigma'$
$\mathit{ext}_n\ \mathit{id}_n \ldots \mathit{id}_1 = \lambda\rho l.\rho[l/\mathit{id}_1]\ldots[(\mathit{succ}^{n-1}\, l)/\mathit{id}_n]$
$\mathit{function}_n\ f\ \mathit{env} = \lambda\eta x v_1 \ldots v_n\sigma.\ \mathit{let}\ l = \mathit{new}_{n+1}\sigma\ \mathit{and}\ \sigma' = \mathit{adjoin\text{-}init}_{n+1}\sigma l\ \text{'uninitialized'}v_1 \ldots v_n$
$\qquad \mathit{in}\ f(\mathit{env}\, l)l\eta l\sigma'$

Equations

$\mathcal{B\ell}[\![(\mathbf{block}\ (\mathbf{var}\ \mathit{id}_1 \ldots \mathit{id}_n)\ \mathit{sl})]\!] = B_1(\mathit{block}_n\, D_1(\mathcal{S\ell}[\![\mathit{sl}]\!], \mathit{release\text{-}block}_n), \mathit{ext}_n\, \mathit{id}_n \ldots \mathit{id}_1)$
$\mathcal{B\ell}[\![(\mathbf{block}\ (\mathbf{fun}\ \mathit{id}\, (\mathit{id}_1 \ldots \mathit{id}_n)\ (\mathit{ml})\ \mathit{sl}_1)\ \mathit{sl}_2)]\!] = B_1(\mathcal{S\ell}[\![\mathit{sl}_2]\!], \mathit{ext\text{-}fun}\, \mathit{id}\, f)$
where f is:
$\qquad f = B_1(\mathit{function}_n\, NP_1(\mathcal{C\ell}_1[\![(\mathit{ml})\ \mathit{sl}_1]\!], D_1(\mathit{fetch}, \mathit{release\text{-}fun}_{n+1})), \mathit{ext}_{n+1}\, \mathit{id}_n \ldots \mathit{id}_1\ \text{result})$
$\mathcal{C\ell}_n[\![(\mathit{empty})\ \mathit{sl}]\!] = \mathit{body}\ \mathcal{S\ell}[\![\mathit{sl}]\!]$
$\mathcal{C\ell}_n[\![(\mathit{im\,ml})\ \mathit{sl}]\!] = \mathit{pass}_{0\,n}\, D_1(\mathcal{M}_{in}[\![\mathit{im}]\!], \mathcal{C\ell}_{n+1}[\![(\mathit{ml})\ \mathit{sl}]\!])$
$\mathcal{C\ell}_n[\![(\mathit{om\ ml})\ \mathit{sl}]\!] = \mathit{pass}_{0\,n}\, NP_n(\mathcal{C\ell}_{n+1}[\![(\mathit{ml})\ \mathit{sl}]\!], \mathcal{M}_{out}[\![\mathit{om}]\!])$

Table 13. Standard display – combinatorial equations

Auxiliaries

$\mathit{table}_p\ \mathit{id}\ i\,\tau = \lambda a_1 \ldots a_p\, \mathit{id}'.\mathit{id}' = \mathit{id} \rightarrow \mathit{succ}^i\, a_p, \tau a_1 \ldots a_p\, \mathit{id}'$
$\mathit{new\text{-}frame}_p\ \tau = \lambda a_1 \ldots a_p\, \mathit{id}'.\tau a_1 \ldots a_{p-1}\, \mathit{id}'$
$B_p(\mathit{ext}_n\ \mathit{id}_n \ldots \mathit{id}_1, \tau) = \mathit{table}_p\ \mathit{id}_n\, n-1 \ldots (\mathit{table}_p\ \mathit{id}_1\, 0(\mathit{new\text{-}frame}_p\ \tau))\ldots$
$\mathit{selec}_p(j, i) = \lambda a_1 \ldots a_p.\mathit{succ}^i\, a_j$
$\mathit{pass}_{p\,m\,n}\ f = \lambda a_1 \ldots a_p\kappa x_1 \ldots x_m\sigma.$
$\qquad \mathit{let}\ l = \mathit{succ}^n\, a_p,\ \mathit{Comband}\ \sigma' = \langle\sigma_1, \sigma_2, \sigma_3[\text{'uninitialized'}/l]\rangle$
$\qquad \mathit{in}\ f a_1 \ldots a_p\kappa x_1 \ldots x_m l(\sigma_3\, l)\sigma'$
$\mathit{block}_{p\,n}\ f = \lambda a_1 \ldots a_p\kappa\sigma.\mathit{let}\ l = \mathit{new}_n\sigma\ \mathit{and}\ \sigma' = \mathit{adjoin}_n\sigma l$
$\qquad \mathit{in}\ f a_1 \ldots a_p l\kappa\sigma'$
$\mathit{function}_{p\,n}\ f = \lambda a_1 \ldots a_p\eta x v_1 \ldots v_n\sigma$
$\qquad \mathit{let}\ l = \mathit{new}_{n+1}\sigma\ \mathit{and}\ \sigma' = \mathit{adjoin\text{-}init}_{n+1}\sigma l\ \text{'uninitialized'}v_1 \ldots v_n$
$\qquad \mathit{in}\ f a_1 \ldots a_p l\eta l\sigma'$
$\mathit{release}_n\ \sigma = \lambda l.\langle\sigma_1, \sigma_2, \sigma_3[\text{'unused'}/l]\ldots[\text{'unused'}/(\mathit{succ}^{n-1}\, l)]\rangle$
$\mathit{release\text{-}block}_{p\,n} = \lambda a_1 \ldots a_p\kappa\sigma.\kappa(\mathit{release}_n\sigma a_p)$
$\mathit{release\text{-}fun}_{p\,n} = \lambda a_1 \ldots a_p\eta v\sigma.\eta v(\mathit{release}_n\sigma a_p)$

Distribution

let $\tau = \ldots(\mathit{table}_j\ \mathit{id}\, i\ldots)\ldots$ or $\tau = \ldots(\mathit{fun\text{-}table}_j\ \mathit{id}\ f\ldots)\ldots$
if id denotes a variable without dereferencing
$\qquad S_{p\,n}(B_p(\mathit{lookup}[\![\mathit{id}]\!], \tau), \beta) = S_{p\,n}(\mathit{selec}_p(j, i), \beta)$
if id denotes a **var** or **fun** parameter
$\qquad S_{p\,n}(B_p(\mathit{lookup}[\![\mathit{id}]\!], \tau), \beta) = S_{p\,n}(\mathit{selec}_p(j, i), S_{p\,n}(\mathit{fetch}, \beta))$
if id denotes a regular function
$\qquad S_{p\,n}(\mathit{lookup}[\![\mathit{id}]\!], \tau), \beta) = S_{p\,n}(\mathit{mk\text{-}fun}_{p\,j}\ f, \beta)$
$NA_p(\mathit{test}_k\ f g, \tau) = \mathit{test}_{p\,k}\ NA_p(f, \tau)\ NA_p(g, \tau)$
$NA_p(\mathit{release\text{-}block}_i, \tau) = \mathit{release\text{-}block}_{p\,i}$
$NA_p(\mathit{release\text{-}fun}_{p\,i}, \tau) = \mathit{release\text{-}fun}_{p\,i}$
$NA_p(\mathit{pass}_{m\,n}\ f, \tau) = \mathit{pass}_{p\,m\,n}\ NA_p(f, \tau)$
$B_p(\mathit{block}_n\ f, \tau) = \mathit{block}_{p\,n}\ NA_p(f, \tau)$
$B_p(\mathit{function}_n\ f, \tau) = \mathit{function}_{p\,n}\ NA_p(f, \tau)$

Table 14. Standard display – distribution

that is NA copies the last parameter which is the base of the current block. NA combinators have the following property:

Property 12.

$$NA_p(D_{m+1}(\alpha, \beta), \gamma) = S_{p\,m}(B_p(\alpha, \beta), NA_p(\beta, \gamma))$$

The new environment-creating functions and the values of the auxiliaries applied to NA are shown in Table 14. The pair (i, j) in the new version of *selec* means the base (i) and the offset (j) of the variable. The transformations are simpler for the standard display, and, as we have already said, they would be even simpler if our equations were in the new standardized form. We think that the machine architecture as well as the interpreting function of M are trivial to define. Therefore we will not define them here.

8 Final remarks

Further modifications may be done in order to obtain a target-machine code that resembles a conventional assembly-language program. One of these modifications has been presented by Wand in [6]. He has introduced a binding operator *label* that makes it possible to implement loops in the standard way, i.e. not storing continuations in the stack (or local register file). This operator is statically scoped and it is a subject for rotation, that is it has the following property ([6]):

$$rot(D_k(while\text{-}loop(x, y), z)) = label\theta.\,rot(D_k(x, test_k(D_k(y, \theta), z)))$$

We can prove that we still are able to distribute the symbol table information to this code, and by introducing pointers, we can eliminate the variable θ entirely ([6]).

After this modification, our code is not linear; it is also not a dag. To be able to store the code in a linear memory, we should introduce "jumps". They may be of the form similar to *retpt*, that is:

$$cont_k\ \theta = \lambda\rho\kappa x_1 \ldots x_k.\theta\rho\kappa x_1 \ldots x_k$$

Notice that because the *label* operator is closed and the value produced by evaluating the loop condition is absorbed by *test*, the first occurrence of θ is of the same functionality as that bound by *cont*. So the number of parameters passed to θ remains the same. The *cont* combinator may be useful for conditionals as well.

Another modification, also suggested by Wand, is introduction of "complex" combinators that eliminate S. We may define them in the following way:

$$STORE_{p\,n}\ \beta = S_{p\,n}(store_n, \beta)$$

These combinators together with *cont* produce a code of a "linear" form. The exceptions are *test* and *block*. A "linear" representation of $test_n\ \alpha\beta$ is easy; one may store this instruction as $(TEST_{p\,n}\ \alpha)\ \beta$, where α is a pointer of the same form as this used by *cont*. Of course, the *cont* combinator may be also used at the end of the β sequence.

To linearize blocks we may introduce a combinator of the following form:

$$back_{pn}\ \alpha\ \beta = \lambda a_1 \ldots a_p \kappa x_1 \ldots x_n . \alpha a_1 \ldots a_p (\beta a_1 \ldots a_{p-1} \kappa x_1 \ldots x_n)$$

and then show that

$$back_{pn}\ S_{pm}(\alpha, \beta)\ \gamma = S_{p\,n+m}(\alpha, back_{pn}\ \beta\ \gamma)$$

Since for the solutions presented so far the local register file is empty when a *release-block* instruction is being evaluated, we may also have:

$$\begin{aligned} RELEASE\text{-}BLOCK_{pni}\ \alpha = \lambda a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma . \alpha a_1 \ldots a_{p-1} \kappa x_1 \ldots x_n \\ \langle \sigma_1, \sigma_2, \sigma_3[\text{'unused'}/a_p] \ldots [\text{'unused'}/(succ^{i-1}\ a_p)] \rangle \end{aligned}$$

and then:

$$back_{pn}\ (release\text{-}block_{pi}\ \alpha) = RELEASE\text{-}BLOCK_{pni}\ \alpha$$

By these properties we may define:

$$BLOCK_{pni}\ \alpha = \lambda a_1 \ldots a_p \kappa x_1 \ldots x_n \sigma . \alpha a_1 \ldots a_p a_{p+1} \kappa x_1 \ldots x_n \sigma'$$

where $a_{p+1} = new_i \sigma$ and $\sigma' = adjoin_i \sigma a_{p+1}$ and

$$S_{pn}(block_{pi}\ \alpha, \beta) = BLOCK_{pni}(back_{p+1\,n}\ \alpha\ \beta)$$

In this way we eventually obtain a linear code of the desired form. Notice that we do not need the *retpt* combinator to enter a block anymore.

Still another question concerns when checking of parameters passed to a function takes place. It is, however, connected with the language design, not the method discussed here. If the formal function had its parameters specified, then it could be possible to carry out all the checking of *L-pass*, *E-pass* and *I-pass* during compilation. *F-pass*, however, would contain a complex checker of function specifications or the formal functions would have to be restricted in some way. That is:

- the current solution specifies (**fun** $f(i\ g)$ (**var fun**) ...) and applications of g need a run-time checking of the number of parameters as well as the kind of parameters passed,
- when specifying g in the same way as f, i.e. (**fun** f $(i\ g)$ (**var fun**(**var fun** (**value**))) ...) we avoid the need of any run-time checking; in this case, however, it is impossible to specify a function that may be applied to itself (infinite chain of **fun** specifications),
- specification of g does not define the second-level function's parameters (as in Loglan), i.e. (**fun** $f(i\ g)$ (**var fun**(**var fun**)) ...) where the **fun** parameter of g does not contain any specification of its parameters; then the run-time checking is needed for *F-pass* only.

These solutions would result in different codes generated by the compiler. For languages which are not strongly-typed (Simula 67, Loglan) *L-pass, E-pass, I-pass* and *F-pass* may serve as the type checker for applications, and in the general case must be carried out when the program is being executed.

References

[1] Dijkstra, E.W. "Recursive Programming", *Numerische Mathematik* **2**,1960 also in *Programming Systems and Languages*, S.Rosen (Ed.), McGraw–Hill, New York, 1967

[2] Gordon, M.J.C. "The Denotational Description of Programming Languages", Springer–Verlag, New York, 1979

[3] Li-Mei Wu, C–616 Project, Indiana University, 1982 (manuscript)

[4] Wand, M. "Semantics-Directed Machine Architecture", *Conf. Rec. 9th ACM Symp. on Principles of Programming Languages* (1982), 234–241

[5] Wand, M. "Deriving Target Code as a Representation of Continuation Semantics" *ACM Trans. on Prog. Lang. and Systems*, **4**, 3, July, 1982, 496–517

[6] Wand, M. "Loops in Combinator-Based Compilers" *Conf. Rec. 10th ACM Symp. on Principles of Prog. Lang.* (1983), 190–196

Appendix A. Interpretation of Example 1

Let σ_0 be the initial state. Since we are not to read and the **print** statement occurs only at the and of the program, we show the changes of the third component of the state only. The locations are bold-typed integers. The labels used are the labels from Figure 6. We also label continuations.

$M_{00}(\ \alpha_0\colon S\,[\mathit{block}_2\ \alpha_2]\ \mathit{return}])\mathit{init\text{-}cont}\,\sigma_0$

$\to\ M_{20}(\alpha_2\colon [S\,[\mathit{block}_1\ \beta_2]\ \alpha_3])\mathbf{12}\kappa_1 : (\mathit{retpt}_{00}\ \mathit{return}\ \mathit{init\text{-}cont})\sigma_1$
  where $\sigma_1 : \mathbf{1} \to$ *'uninitialized'*, $\mathbf{2} \to$ *'uninitialized'*

$\to\ M_{30}(\beta_2\colon [S\,[\mathit{selec}\ 3]\ \beta_3])\mathbf{123}\kappa_2 : \mathit{retpt}_{20}\ \alpha_3\ \mathbf{12}\kappa_1)\sigma_2$
  where $\sigma_2 : \mathbf{1} \to$ *'uninitialized'*, $\mathbf{2} \to$ *'uninitialized'*, $\mathbf{3} \to$ *'uninitialized'*

$\to\ M_{31}(\beta_3\colon [S\,[\mathit{cons}\ 1]\ \beta_4])\mathbf{123}\kappa_2\,\mathbf{3}\sigma_2$

$\to\ M_{32}(\beta_4\colon [S\ \mathit{store}\ \beta_5])\mathbf{123}\kappa_2\,\mathbf{3}1\sigma_2$

$\to\ M_{30}(\beta_5\colon [S\,[\mathit{wloop}\ \gamma_2]\ \beta_6])\mathbf{123}\kappa_2\sigma_3$
  where σ_3; $\mathbf{1} \to$ *'uninitialized'*, $\mathbf{2} \to$ *'uninitialized'*, $\mathbf{3} \to 1$

$\to\ M_{30}(\gamma_1\colon [\mathit{wloop}\ \gamma_2])\mathbf{123}\kappa_3 : (\mathit{retpt}_{30}\ \beta_6\ \mathbf{123}\kappa_2)\sigma_3$

$\to\ M_{31}(\gamma_2\colon [S\,[\mathit{selec}\ 3]\ \gamma_3])\mathbf{123}\kappa_3\gamma_1\sigma_3$

$\to\ M_{32}(\gamma_3\colon [S\ \mathit{fetch}\ \gamma_4])\mathbf{123}\kappa_3\gamma_1\,\mathbf{3}\sigma_3$

$\to\ M_{32}(\gamma_4\colon [S\,[\mathit{unpred\ positive?}]\ \gamma_5]))\mathbf{123}\kappa_3\gamma_1\,1\sigma_3$

$\to\ M_{32}(\gamma_5\colon [\mathit{wtest}\ \gamma_6])\mathbf{123}\kappa_3\gamma_1\ \mathit{true}\,\sigma_3$

$\to\ M_{30}(\gamma_6\colon [S\,[\mathit{selec}\ 3]\ \gamma_7])\mathbf{123}\kappa_4 : (\mathit{retpt}_{30}\ \gamma_1\ \mathbf{123}\kappa_3)\sigma_3$

$\to\ M_{31}(\gamma_7\colon [S\,[\mathit{selec}\ 3]\ \gamma_8])\mathbf{123}\kappa_4\,\mathbf{3}\sigma_3$

$\to\ M_{32}(\gamma_8\colon [S\ \mathit{fetch}\ \gamma_9])\mathbf{123}\kappa_4\,\mathbf{33}\sigma_3$

$\to\ M_{32}(\gamma_9\colon [S\,[\mathit{unop\ minus1}]\ \gamma_{10}])\mathbf{123}\kappa_4\,\mathbf{3}1\sigma_3$

$\to\ M_{32}(\gamma_{10}\colon [S\ \mathit{store}\ \gamma_{11}])\mathbf{123}\kappa_4\,\mathbf{3}0\sigma_3$

$\to\ M_{30}(\gamma_{11}\colon \mathit{return})\mathbf{123}\kappa_4 : (\mathit{retpt}_{30}\ \gamma_1\ \mathbf{123}\kappa_3)\sigma_4$
  where $\sigma_4 : \mathbf{1} \to$ *'uninitialized'*, $\mathbf{2} \to$ *'uninitialized'*, $\mathbf{3} \to 0$

$\to\ M_{30}(\gamma_1\colon [\mathit{wloop}\ \gamma_2])\mathbf{123}\kappa_3\sigma_4$

$\to\ M_{31}(\gamma_2\colon [S\,[\mathit{selec}\ 3]\ \gamma_3])\mathbf{123}\kappa_3\gamma_1\sigma_4$

$\to\ M_{32}(\gamma_3\colon [S\ \mathit{fetch}\ \gamma_4])\mathbf{123}\kappa_3\gamma_1\,\mathbf{3}\sigma_4$

$\to\ M_{32}(\gamma_4\colon [S\,[\mathit{unpred\ positive?}]\ \gamma_5])\mathbf{123}\kappa_3\gamma_1\,0\sigma_4$

$\to\ M_{32}(\gamma_5\colon [\mathit{wtest}\ \gamma_6])\mathbf{123}\kappa_3 : (\mathit{retpt}_{30}\ \beta_6\ \mathbf{123}\kappa_2)\gamma_1\,\mathit{false}\,\sigma_4$

$\to\ M_{30}(\beta_6\colon [S\,[\mathit{selec}\ 2]\ \beta_7])\mathbf{123}\kappa_2\sigma_4$

$\to\ M_{31}(\beta_7\colon [S\,[\mathit{selec}\ 3]\ \beta_8])\mathbf{123}\kappa_2\sigma_4$

$\to\ M_{32}(\beta_8\colon [S\ \mathit{fetch}\ \beta_9])\mathbf{123}\kappa_2\,\mathbf{2}\sigma_4$

$\to\ M_{32}(\beta_9\colon [S\ \mathit{store}\ \beta_{10}])\mathbf{123}\kappa_2\,\mathbf{2}0\sigma_4$

$\to\ M_{30}(\beta_{10}\colon \mathit{release\text{-}block}_1)\mathbf{123}\kappa_2 : (\mathit{retpt}_{20}\ \alpha_3\ \mathbf{12}\kappa_1)\sigma_5$
  where $\sigma_5 : \mathbf{1} \to$ *'uninitialized'*, $\mathbf{2} \to 0$, $\mathbf{3} \to 0$

$\to\ M_{20}(\alpha_3\colon [S\,[\mathit{selec}\ 1]\ \alpha_4])\mathbf{12}\kappa_1\sigma_6$
  where $\sigma_6 : \mathbf{1} \to$ *'uninitialized'*, $\mathbf{2} \to 0$, $\mathbf{3} \to$ *'unused'*

$\to\ M_{21}(\alpha_4\colon [S\,[\mathit{selec}\ 2]\ \alpha_5])\mathbf{12}\kappa_1\,\mathbf{1}\sigma_6$

$\to\ M_{22}(\alpha_5\colon [S\ \mathit{fetch}\ \alpha_6])\mathbf{12}\kappa_1\,\mathbf{12}\sigma_6$

$\to\ M_{22}(\alpha_6\colon [S\ \mathit{store}\ \alpha_7])\mathbf{12}\kappa_1\,\mathbf{1}0\sigma_6$

$\to\ M_{20}(\alpha_7\colon [S\,[\mathit{selec}\ 1]\ \alpha_8])\mathbf{12}\kappa_1\sigma_7$
  where $\sigma_7 : \mathbf{1} \to 0$, $\mathbf{2} \to 0$

$\to\ M_{21}(\alpha_8\colon [S\ \mathit{fetch}\ \alpha_9])\mathbf{12}\kappa_1\,\mathbf{1}\sigma_7$

$\to\ M_{21}(\alpha_9\colon [S\ \mathit{do\text{-}print}\ \alpha_{10}])\mathbf{12}\kappa_1\,0\sigma_7$

$\to\ M_{20}(\alpha_{10}\colon \mathit{release\text{-}block}_2)\mathbf{12}\kappa_1 : (\mathit{retpt}_{00}\ \mathit{return}\ \mathit{init\text{-}cont})\sigma_7$
  do-print prints 0 into the second component of the state

$\to\ M_{00}(\mathit{return})\,\mathit{init\text{-}cont}\,\sigma_8$
  where $\sigma_8 : \mathbf{1} \to$ *'unused'*, $\mathbf{2} \to$ *'unused'*

$\rightarrow$ $\mathit{init\text{-}cont}\,\sigma_8$ produces the answer: 0 "normal termination"

Appendix B. Interpretation of Example 2

This example shows how functions and function calls are interpreted. As in Appendix A, we show only the changes of the third component of σ. Since we have to distinguish the values from different components of V, we denote locations as bold-typed integers – it may be implemented by using tag bits for values in V. The labels for instruction sequences come from Figure 7; σ_0 is the initial state.

$M_{00}(\alpha_0\colon [S\,[\mathit{block}_1\ \alpha_2]\ \mathit{return}])\ \mathit{init\text{-}cont}\,\sigma_0$

$\rightarrow\quad M_{10}(\alpha_2\colon [S\,[\mathit{selec}\ 1]\ \alpha_3])\mathbf{1}\kappa_1 : (\mathit{retpt}_{00}\,\mathit{return}\,\mathit{init\text{-}cont})\sigma_1$

where $\sigma_1 : \mathbf{1} \rightarrow$ *'uninitialized'*

$\rightarrow\quad M_{11}(\alpha_3\colon [S\,[\mathit{const}\ 1]\ \alpha_4])\mathbf{1}\kappa_1\,\mathbf{1}\sigma_1$

$\rightarrow\quad M_{12}(\alpha_4\colon [S\ \mathit{store}\ \alpha_5])\mathbf{1}\kappa_1\,\mathbf{11}\sigma_1$

$\rightarrow\quad M_{10}(\alpha_5\colon [S\,[\mathit{selec}\ 1]\ \alpha_6])\mathbf{1}\kappa_1\sigma_2$

where $\sigma_2 : \mathbf{1} \rightarrow 1$

$\rightarrow\quad M_{11}(\alpha_6\colon [S\,[\mathit{mk\text{-}fun}_0\ \beta_0]\ \alpha_7])\mathbf{1}\kappa_1\,\mathbf{1}\sigma_2$

$\rightarrow\quad M_{12}(\alpha_7\colon [S\ \mathit{check}_2\ \alpha_8])\mathbf{1}\kappa_1\,\mathbf{1}\boldsymbol{f} : \langle \mathit{function}_{02}\ \beta_1\rangle\sigma_2$

$\rightarrow\quad M_{12}(\alpha_8\colon [S\,[\mathit{selec}\ 1]\ \alpha_9])\mathbf{1}\kappa_1\,\mathbf{1}\boldsymbol{f}\sigma_2$

$\rightarrow\quad M_{13}(\alpha_9\colon [S\,[\mathit{selec}\ 1]\ \alpha_{10}])\mathbf{1}\kappa_1\,\mathbf{1}\boldsymbol{f}\,\mathbf{1}\sigma_2$

$\rightarrow\quad M_{14}(\alpha_{10}\colon [S\ \mathit{apply}_2\ \alpha_{11}])\mathbf{1}\kappa_1\,\mathbf{1}\boldsymbol{f}\,\mathbf{11}\sigma_2$

$\rightarrow\quad M_{03}(\beta_0\colon \mathit{function}_{02}\ \beta_1)\kappa_2 : (\mathit{retpt}_{11}\,\alpha_{11}\,\mathbf{1}\kappa_1\,\mathbf{1})\boldsymbol{f}\,\mathbf{11}\sigma_2$

$\rightarrow\quad M_{31}(\beta_1\colon [\mathit{pass}_2\ \beta_2])\mathbf{234}\kappa_2\,\mathbf{2}\sigma_3$

where $\sigma_3 : \mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow \mathbf{1}, \mathbf{4} \rightarrow \mathbf{1}$

$\rightarrow\quad M_{33}(\beta_2\colon [S\ \mathit{E\text{-}pass}\ \beta_3])\mathbf{234}\kappa_2\,\mathbf{231}\sigma_4$

where $\sigma_4 : \mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow$ *'uninitialized'*, $\mathbf{4} \rightarrow \mathbf{1}$

$\rightarrow\quad M_{31}(\beta_3\colon [\mathit{pass}_1\ \beta_4])\mathbf{234}\kappa_2\,\mathbf{2}\sigma_5$

where $\sigma_5 : \mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow \mathbf{1}$

$\rightarrow\quad M_{33}(\beta_4\colon [S\,[\mathit{selec}\ 2]\ \beta_5])\mathbf{234}\kappa_2\,\mathbf{241}\sigma_6$

where $\sigma_6 : \mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow$ *'uninitialized'*

$\rightarrow\quad M_{34}(\beta_5\colon [S\ \mathit{fetch}\ \beta_6])\mathbf{234}\kappa_2\,\mathbf{2413}\sigma_6$

$\rightarrow\quad M_{34}(\beta_6\colon [S\,[\mathit{unpred\ zero?}]\ \beta_7])\mathbf{234}\kappa_2\,\mathbf{2411}\sigma_6$

$\rightarrow\quad M_{34}(\beta_7\colon [\mathit{test}\ \delta_1\ \gamma_1])\mathbf{234}\kappa_2\,\mathbf{241}\mathit{false}\sigma_6$

$\rightarrow\quad M_{33}(\gamma_1\colon [S\,[\mathit{selec}\ 1]\ \gamma_2])\mathbf{234}\kappa_2\,\mathbf{241}\sigma_6$

$\rightarrow\quad M_{34}(\gamma_2\colon [S\,[\mathit{selec}\ 2]\ \gamma_3])\mathbf{234}\kappa_2\,\mathbf{2412}\sigma_6$

$\rightarrow\quad M_{35}(\gamma_3\colon [S\ \mathit{fetch}\ \sigma_4])\mathbf{234}\kappa_2\,\mathbf{24123}\sigma_6$

$\rightarrow\quad M_{35}(\gamma_4\colon [S\,[\mathit{mk\text{-}fun}_0\ \beta_0]\ \gamma_5])\mathbf{234}\kappa_2\,\mathbf{24121}\sigma_6$

$\rightarrow\quad M_{36}(\gamma_5\colon [S\ \mathit{check}_2\ \gamma_6])\mathbf{234}\kappa_2\,\mathbf{24121}\boldsymbol{f} : \langle \mathit{function}_{02}\ \beta_1\rangle\sigma_6$

$\rightarrow\quad M_{36}(\gamma_6\colon [S\,[\mathit{selec}\ 2]\ \gamma_7])\mathbf{234}\kappa_2\,\mathbf{24121}\boldsymbol{f}\sigma_6$

$\rightarrow\quad M_{37}(\gamma_7\colon [S\ \mathit{fetch}\ \gamma_8])\mathbf{234}\kappa_2\,\mathbf{24121}\boldsymbol{f}\,\mathbf{3}\sigma_6$

$\rightarrow\quad M_{37}(\gamma_8\colon [S\,[\mathit{unop\ minus1}]\ \gamma_9])\mathbf{234}\kappa_2\,\mathbf{24121}\boldsymbol{f}\,1\sigma_6$

$\rightarrow\quad M_{37}(\gamma_9\colon [S\,[\mathit{selec}\ 3]\ \gamma_{10}])\mathbf{234}\kappa_2\,\mathbf{24121}\boldsymbol{f}\,0\sigma_6$

$\rightarrow\quad M_{38}(\gamma_{10}\colon [S\ \mathit{apply}_2\ \gamma_{11}])\mathbf{234}\kappa_2\,\mathbf{24121}\boldsymbol{f}\,0\,\mathbf{4}\sigma_6$

$\rightarrow\quad M_{03}(\beta_0\colon \mathit{function}_{02}\beta_1)\kappa_3 : (\mathit{retpt}_{35}\,\gamma_{11}\,\mathbf{234}\kappa_2\,\mathbf{24121})\boldsymbol{f}\,00\mathbf{4}\sigma_6$

$\rightarrow\quad M_{31}(\beta_1\colon [\mathit{pass}_2\ \beta_2])\mathbf{567}\kappa_3\,\mathbf{5}\sigma_7$

where $\sigma_7 : \mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4}, \mathbf{5} \rightarrow$ *'uninitialized'*,

$\mathbf{6} \rightarrow 0, \mathbf{7} \rightarrow \mathbf{4}$

$\rightarrow\quad M_{33}(\beta_2\colon [S\ \mathit{E\text{-}pass}\ \beta_3])\mathbf{567}\kappa_3\,\mathbf{560}\sigma_8$

where $\sigma_8 : \mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4}, \mathbf{5}, \mathbf{6} \rightarrow$ *'uninitialized'*,

$\mathbf{7} \rightarrow \mathbf{4}$

$\rightarrow\quad M_{31}(\beta_3\colon [\mathit{pass}_1\ \beta_4])\mathbf{567}\kappa_3\,\mathbf{5}\sigma_9$

where $\sigma_9 : \mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4}, \mathbf{5} \rightarrow$ *'uninitialized'*,

$\mathbf{6} \rightarrow 0, \mathbf{7} \rightarrow \mathbf{4}$

$\rightarrow$ $M_{33}(\beta_4$: $[S\,[selec\ 2]\ \beta_5])\mathbf{567}\kappa_3\,\mathbf{574}\sigma_{10}$
where σ_{10} : $\mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4}, \mathbf{5} \rightarrow$ *'uninitialized'*,
$\mathbf{6} \rightarrow 0, \mathbf{7} \rightarrow$ *'uninitialized'*
$\rightarrow$ $M_{34}(\beta_5$: $[S\,fetch\ \beta_6])\mathbf{567}\kappa_3\,\mathbf{5746}\sigma_{10}$
$\rightarrow$ $M_{34}(\beta_6$: $[S\,[unpred\ zero?]\ \beta_7])\mathbf{567}\kappa_3\,\mathbf{5740}\sigma_{10}$
$\rightarrow$ $M_{34}(\beta_7$: $[test\ \delta_1\ \gamma_1])\mathbf{567}\kappa_3\,\mathbf{574}\,true\sigma_{10}$
$\rightarrow$ $M_{33}(\delta_1$: $[S\,[selec\ 1]\ \delta_2])\mathbf{567}\kappa_3\,\mathbf{574}\sigma_{10}$
$\rightarrow$ $M_{34}(\delta_2$: $[S\,[const\ 1]\ \delta_3])\mathbf{567}\kappa_3\,\mathbf{5745}\sigma_{10}$
$\rightarrow$ $M_{35}(\delta_3$: $[S\,store\ \delta_4])\mathbf{567}\kappa_3\,\mathbf{57451}\sigma_{10}$
$\rightarrow$ $M_{33}(\delta_4$: $[S\,[selec\ 3]\ \delta_5])\mathbf{567}\kappa_3\,\mathbf{574}\sigma_{11}$
where σ_{11} : $\mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow$ *'uninitialized'*,
$\mathbf{5} \rightarrow 1, \mathbf{6} \rightarrow 0, \mathbf{7} \rightarrow$ *'uninitialized'*
$\rightarrow$ $M_{34}(\delta_5$: $[S\,[const\ 0]\ \delta_6])\mathbf{567}\kappa_3\,\mathbf{5747}\sigma_{11}$
$\rightarrow$ $M_{35}(\delta_6$: $[S\,store\ \eta_1)\mathbf{567}\kappa_3\mathbf{57470}\sigma_{11}$
$\rightarrow$ $M_{33}(\eta_1$: $[S\,I\text{-}pass\ \eta_2])\mathbf{567}\kappa_3\,\mathbf{574}\sigma_{12}$
where σ_{12} : $\mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow$ *'uninitialized'*, $\mathbf{5} \rightarrow 1$,
$\mathbf{6} \rightarrow 0, \mathbf{7} \rightarrow 0$
$\rightarrow$ $M_{31}(\eta_2$: $[S\,fetch\ \eta_3])\mathbf{567}\kappa_3\,\mathbf{5}\sigma_{13}$
where σ_{13} : $\mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow 0, \mathbf{5} \rightarrow 1, \mathbf{6} \rightarrow 0$,
$\mathbf{7} \rightarrow 0$
$\rightarrow$ $M_{31}(\eta_3$: $release\text{-}fun_3)\mathbf{567}\kappa_3 : (retpt_{35}\,\gamma_{11}\,\mathbf{234}\kappa_2\,\mathbf{24121})1\sigma_{13}$
$\rightarrow$ $M_{36}(\gamma_{11}$: $[S\,[binop\ times]\ \gamma_{12}])\mathbf{234}\kappa_2\,\mathbf{241211}\sigma_{14}$
where σ_{14} : $\mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow$ *'uninitialized'*, $\mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow 0, \mathbf{5}, \mathbf{6}, \mathbf{7} \rightarrow$ *'unused'*
$\rightarrow$ $M_{35}(\gamma_{12}$: $[S\,store\ \eta_1])\mathbf{234}\kappa_2\,\mathbf{24121}\sigma_{14}$
$\rightarrow$ $M_{33}(\eta_1$: $[S\,I\text{-}pass\ \eta_2])\mathbf{234}\kappa_2\,\mathbf{241}\sigma_{15}$
where σ_{15} : $\mathbf{1} \rightarrow 1, \mathbf{2} \rightarrow 1, \mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow 0$
$\rightarrow$ $M_{31}(\eta_2$: $[S\,fetch\ \eta_3])\mathbf{234}\kappa_2\,\mathbf{2}\sigma_{16}$
where σ_{16} : $\mathbf{1} \rightarrow 0, \mathbf{2} \rightarrow 1, \mathbf{3} \rightarrow 1, \mathbf{4} \rightarrow 0$
$\rightarrow$ $M_{31}(\eta_3$: $release\text{-}fun_3)\mathbf{234}\kappa_2 : (retpt_{11}\,\alpha_{11}\,\mathbf{1}\kappa_1\,\mathbf{1})1\sigma_{16}$
$\rightarrow$ $M_{12}(\alpha_{11}$: $[S\,store\ \alpha_{12}])\mathbf{1}\kappa_1\,\mathbf{11}\sigma_{17}$
where σ_{17} : $\mathbf{1} \rightarrow 0, \mathbf{2}, \mathbf{3}, \mathbf{4} \rightarrow$ *'unused'*
$\rightarrow$ $M_{10}(\alpha_{12}$: $[S\,[selec\ 1]\ \alpha_{13}])\mathbf{1}\kappa_1\sigma_{18}$
where σ_{18} : $\mathbf{1} \rightarrow 1$
$\rightarrow$ $M_{11}(\alpha_{13}$: $[S\,fetch\ \alpha_{14}])\mathbf{1}\kappa_1\,\mathbf{1}\sigma_{18}$
$\rightarrow$ $M_{11}(\alpha_{14}$: $[S\,print\ \alpha_{15}])\mathbf{1}\kappa_1\,1\sigma_18$
$\rightarrow$ $M_{10}(\alpha_{15}$: $release\text{-}block_1)\mathbf{1}\kappa_1 : (retpt_{00}\ return\ init\text{-}cont)\sigma_{19}$
The second component of σ_{19} contains 1
$\rightarrow$ $M_{00}(return)\ init\text{-}cont\sigma_{20}$ σ_{20} : $\mathbf{1} \rightarrow$ *'unused'*
$\rightarrow$ $init\text{-}cont\ \sigma_{20}$ produces the answer: 1 "normal termination"

Notice that the value of the function call is transmitted after all **result** parameters (states σ_{17} and σ_{18} for instance).