Direct Denotational Semantics:

Combinator-Based Compilation of Goto's

May, 1984

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 156

Direct Denotational Semantics:

Combinator-Based Compilation of Goto's

May, 1984

by

Marek J. Lao

This material is based on work supported by the National Science Foundation under grants MCS 79-04183 and MCS 83-03325.

Direct denotational semantics: combinator-based compilation of **goto**'s

Marek J. Lao*
Computer Science Department
Indiana University
Bloomington, IN 47405

Abstract

Direct denotational semantics is a simple method for defining programming languages. However the semantics of **goto** commands must be rather elaborate and far from real implementations. When a **goto** command is encountered, the abstract machine defining the meaning of programs switches to a searching mode and begins to scan the code to find the corresponding label. The same command in real machines is implemented as a direct jump (a pointer) to the appropriate place in the executable code. This paper presents a compilation of programs defined by a direct semantics to the usual form of a target code. Wand's technique of combinator-based compilation is used, so every transformation during compilation preserves the meaning of the program. This means that the entire compiler is correct and no additional proofs are needed.

Key Words and Phrases:

programming languages, semantics, denotational semantics, direct semantics, compilers, lambda calculus, combinators

1 Introduction

Blikle and Tarlecki in [2] present a "naive" direct (i.e. non-continuational) denotational semantics for a block-structured language with **goto**'s. The semantics is very simple and does not require reflexive Scott domains ([5,6]) to define the meaning of programs. Many of the domains used can be simple sets without additional features. Definitions of semantic functions are expressed by very common combinators like composition and conditional composition. Therefore the semantics can be easily understood by programmers and can be useful in proving correctness of programs.

The simplicity of "naive" semantics must have a drawback. The semantics of **goto**'s is easily expressible in the continuational style, while for the direct style the existence of **goto**'s forces the language designers to introduce an additional component to the abstract machine implementing the language. This component is a *mode*. The machine is always in one of two modes: executing or searching for a label. The idea is that when the machine is in executing mode, all non-**goto** commands transform the state in the usual way. If a **goto** command is encountered, the machine switches to

*Author's permanent address: Institute of Informatics, University of Warsaw, PKiN, P.O.Box 1210, 00-901 Warsaw, Poland

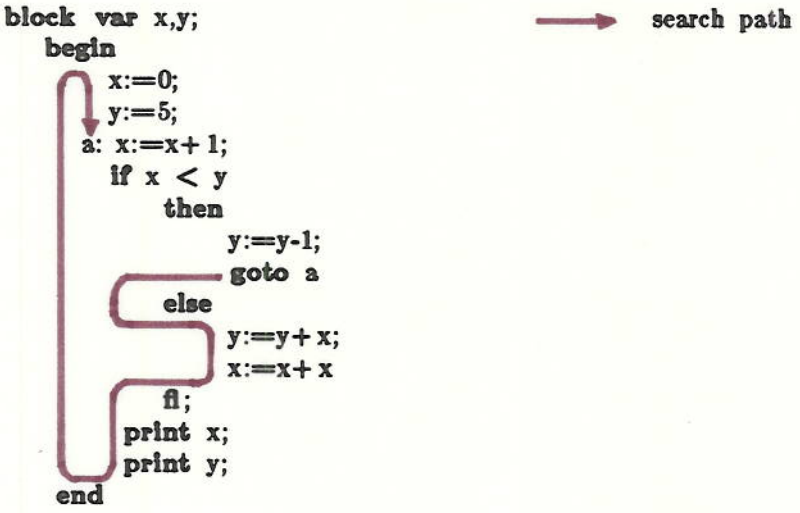


Figure 1. A search path in a program

a searching mode and begins scanning commands until a corresponding label is found. Then it switches back to the executing mode. Figure 1 presents a simple program with a search path shown.

The authors admit ([2]) that their "... main concern is to say *what* programs are doing rather than *how* we execute them". The abstract implementation is not optimal, although it is easy to think of a program defined in a traditional real-machine code which produces the same answers and is optimal in the sense that all commands are always executed (not scanned). Our task is to build a compiler which given a program defined in the naive semantics produces such a code. Usually one shows that the original program and the target code produced by a compiler are equivalent, that is they produce the same answers for the same data. This proof may be quite difficult and therefore most compiler designers omit any formal arguments of this equivalence.

Wand in his papers ([8,9,10]) presents an approach that makes this question trivial. His combinator-based compilers modify the code by equality-preserving transformations. Hence every elementary step (processing a command, expression etc.) can be performed separately and the result will still give correct answers. Due to this fact, the compiler must be correct. In this paper, we apply Wand's technique to build a compiler for a block-structured language with **goto**'s defined in almost the same way as introduced by Blikle and Tarlecki. The target machine is always in the executing mode, so no scanning of the code is needed to find a label when a program runs.

The method consists of the following steps:

- introduction of special purpose combinators that enable us to abstract from the details of the code,
- linearization of the combinator code,
- introduction of new fixed-point equations for each label in order to handle jumps backwards,

– elimination of **goto**'s, that is elimination of the searching mode.

For the program from Figure 1, we could informally denote the part with the label as FIX($\lambda\theta$....**if**...**then** $y := y - 1; \theta$...). A real compiler built by our method must therefore be able to introduce unique variables (like θ in the above "denotation") and unique labels for compiling conditionals.

The naive denotational semantics is defined exclusively for complete partial orders. Therefore in order to present our notation and to provide a basis for our proofs, in Section 2 we present basic features of complete partial orders. We also adopt pipe combinators introduced in [4] which generalize composition to functions delivering many results.

Section 3 presents a skeleton language which contains all control constructs interesting to us and which may be easily extended to a full programming language. The skeleton language does not define declarations, expressions or basic commands; every sensible extension should be possible without changes in our algorithms. Also in this part we introduce special-purpose combinators and show how to combinatorize semantic equations. As a result, we can represent the denotation of a program as a graph whose vertices are combinators and whose leaves are denotations of the basic constructs.

The combinator trees defined in Section 3 are still too complex for our transformations. In Section 4 we introduce *linear trees* which are of a very simple form and provide means for structural induction in the proofs . We show how to linearize combinator trees corresponding to a program.

Finally in Section 5 we describe the main transformation, that is we show how to change all **goto**'s into pointers to appropriate fragments of the code.

The method is illustrated by transformations of an example program.

2 Complete partial orders of partial functions

2.1 Complete partial orders

For the purpose of this paper, all domains may be simply regarded as sets. In a more specific approach, domains form complete partial orders ([2]) or lattices ([5,6]). Properties of complete partial orders are well-known. Here we recall only a few definitions and basic facts.

A partially ordered set $(A, \sqsubseteq_A)$ is called a *complete partial order* (or a *cpo*) if it contains the least element $\bot_A$ (the bottom) and if any increasing countable sequence (a chain) $a_1 \sqsubseteq_A a_2 \sqsubseteq_A \ldots$ in A has a least upper bound $\bigsqcup_A a_n$ in A. We shall omit the index A at the relation symbol, bottom symbol and least upper bound symbol if the set can be understood from the context. Cpo's may be combined together by operations like disjoint union, Cartesian product and a few others. Figure 2 shows two different cpo's of truth values.

A cpo is called a *set-theoretic cpo* if its elements are sets and if they are ordered by inclusion. An example set-theoretic cpo may be the set of all subsets of natural numbers. The least element of this cpo is the empty set. Every element of this cpo is a least upper bound of a chain consisting of finite subsets only. Thus the set-theoretic partial order of all finite subsets of natural numbers forms a *basis* for this cpo. For a set

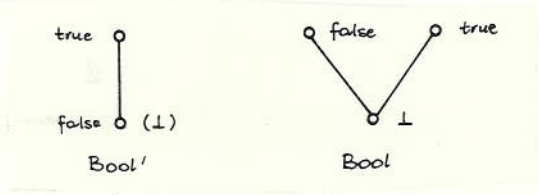


Figure 2. Cpo's of truth values

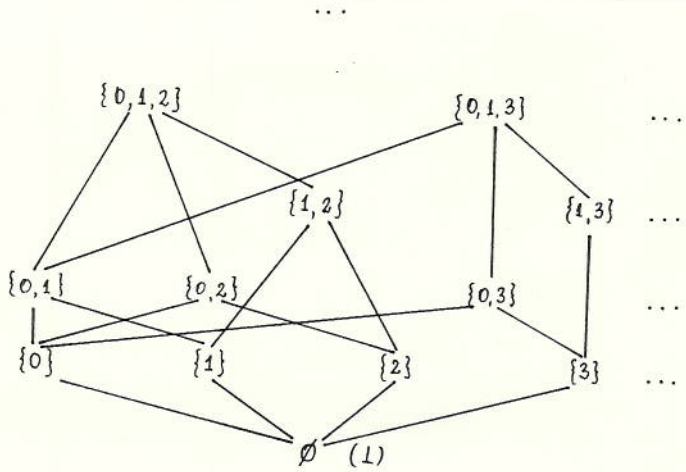


Figure 3. FIN(N) – the partial order of all finite subsets of natural numbers

A we denote the partial order of all its finite subsets by FIN(A). Figure 3 shows a few elements of FIN(N).

2.2 Functions, least fixed points

If A and B are cpo's then a total function $f : A \to B$ is *monotone* iff $a_1 \sqsubseteq_A a_2$ implies $f(a_1) \sqsubseteq_B f(a_2)$. A total function is *continuous* if for any chain $a_1 \sqsubseteq_A a_2 \sqsubseteq_A \ldots$ we have a chain $f(a_1) \sqsubseteq_B f(a_2) \sqsubseteq_B \ldots$ (i.e. f is monotone) and $f(\bigsqcup_A a_n) = \bigsqcup_B f(a_n)$.

If $f : A \to A$ is continuous, then there exists the least a such that $f(a) = a$. It is called the *least fixed point* of f and we denote it by FIX(f). The least fixed point is the limit of approximation sequence $f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq f(f(f(\bot))) \sqsubseteq \ldots$ that is $\mathrm{FIX}(f) = \bigsqcup f^n(\bot)$. This fact (*Kleene's fixed-point theorem*) enables us to find solutions for recursive definitions of functions. In this case, A is a cpo of functions.

Now let us fix our notation. We write $f.x$ for a function application, although we also feel free to write $f(x)$ for the same purpose. The symbol ? is used to denote that the value of some expression is undefined, that is $f.x = ?$ means that the value of f for x is undefined. We write $a \to b, c$ for a conditional expression. Its value is b if a is true and c if a is false. If a is undefined, then the value of the conditional is undefined as well. We write

$$b_1 \to c_1,$$
$$b_2 \to c_2,$$
$$\ldots$$

$$b_n \to c_n,$$
$$\text{OTHERWISE} \to c_{n+1}$$

for $b_1 \to c_1, (b_2 \to c_2, \ldots (b_n \to c_n, c_{n+1}) \ldots)$.

We use the λ-notation to denote functions, for instance $\lambda x.x + a$ denotes a function of x "add a to x". However, more often we define a function by a formula showing the value of $f.x$.

For $b_i \in B$ and $a_i \in A$, $[b_1/a_1, \ldots b_n/a_n]$ denotes a pseudo-mapping from A to B and is defined as:

$$[b_1/a_1, \ldots b_n/a_n].a =$$
$$(a = a_1) \to b_1,$$
$$\ldots$$
$$(a = a_n) \to b_n,$$
$$\text{OTHERWISE} \to ?$$

2.3 Cpo's of partial functions

The set of all partial functions from A to B is a set-theoretic cpo. A function is a set of argument-value pairs. So we can order functions by inclusion; $f \subseteq g$ means that g agrees with f on all arguments where f is defined, g may, however, be defined in some additional points. The least element in this cpo is the totally undefined function (the empty set of pairs). This cpo is of a special interest to us since all semantic functions (program denotations) in our semantics are partial functions. Figure 4 shows a few elements from such a cpo; we denote this set by $A \Rightarrow B$.

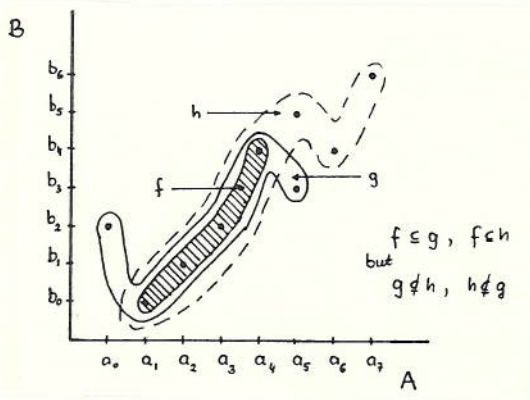


Figure 4. $A \Rightarrow B$ – a cpo of partial functions

Two partial functions $f : A \Rightarrow B$ and $g : B \Rightarrow C$ can be (sequentially) composed together to define a function $f \circ g : A \Rightarrow C$. The meaning of this composition is: $(f \circ g).a = (f.a = ?) \to ?, g.(f.a)$. The composition is associative, that is $f \circ (g \circ h) = (f \circ g) \circ h$. The $\circ$ combinator is total on its arguments. It is monotone and continuous in a cpo of partial functions.

If B is a set (or a cpo) that contains the truth values, $p : A \Rightarrow B$ and $f, g : A \Rightarrow C$, then we define a function (conditional composition) IF p THEN f ELSE g FI : $A \Rightarrow C$ as

(IF p THEN f ELSE g FI).a =

$(p.a = \textbf{true}) \rightarrow f.a,$
$(p.a = \textbf{false}) \rightarrow g.a,$
OTHERWISE $\rightarrow$?

If $A = C$ and g is the identity on A, then we will write IF p THEN f FI. Again, this combinator is total, monotone and continuous in a cpo of partial functions.

Due to the continuity of $\circ$ and IF, every functional from $(A \Rightarrow C) \rightarrow (A \Rightarrow C)$ defined exclusively by means of these combinators is continuous, so it has the least fixed point. Since all our semantic functions are of this form, we guarantee the existence of the least fixed points, so all our recursive definitions have solutions.

2.4 Pipes

Raoult and Sethi in [4] introduced a useful notation that enables us to compose functions delivering many results. They called such a composition a *pipe* because it resembles pipes used in the UNIX operating system. Pipes make it possible to pass some arguments to functions which are not performed immediately. Consider the following example. Let $F : A \rightarrow C \Rightarrow (A \times C)$ and $G : A \rightarrow A$. If we want to peform F for a modified by G and for c, that is if we want to have $F.(G.a)c$, then the traditional definition of composition, i.e. the $\circ$ combinator, is not useful. Here we define a simple version of pipes which make such compositions possible and quite easy.

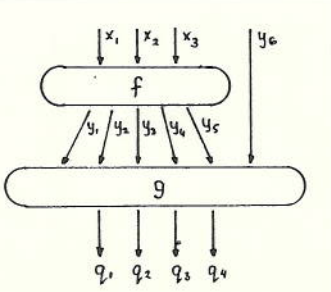


Figure 5. $f \bullet g$ – a pipe of two functions

Let $f : X_1 \rightarrow \cdots X_i \Rightarrow Y_1 \times \cdots \times Y_j$ and $g : Y_1 \rightarrow \cdots Y_j \rightarrow Y_{j+1} \rightarrow \cdots Y_k \Rightarrow Z_1 \times \cdots \times Z_m$. Then the *pipe* of f and g is a function

$$f \bullet g : X_1 \rightarrow \cdots \rightarrow X_i \rightarrow Y_{j+1} \rightarrow Y_k \Rightarrow Z_1 \times \cdots \times Z_m$$

defined as:

$$f \bullet g.x_1 \ldots x_i y_{j+1} \ldots y_k = \begin{cases} ?, & \text{if } f.x_1 \ldots x_i = ? \\ g.y_1 \ldots y_j y_{j+1} \ldots y_k, & \text{if } f.x_1 \ldots x_i = (y_1, \ldots y_j) \end{cases}$$

The function in our example is then $G \bullet F^*$. Figure 5 shows this concept.

The $\bullet$ combinator is monotone and continuous in its arguments (as it is defined by means of composition).

---

*Pipes in [4] are denoted by $G \mid F$.

Property 1 . Right associativity of pipes

Let $f : X_1 \to \cdots \to X_i \Rightarrow Y_1 \times \cdots \times Y_j$,
$g : Y_1 \to \cdots \to Y_j \to Y_{j+1} \to \cdots \to Y_k \Rightarrow Z_1 \times \cdots \times Z_l$,
$h : Z_1 \to \cdots \to Z_l \to Z_{l+1} \cdots \to Z_m \Rightarrow Q_1 \times \cdots \times Q_n$
Then
$(f \bullet g) \bullet h = f \bullet (g \bullet h)$

Due to the right associativity of pipes, we can always linearize them to a standard form of $f_1 \bullet (f_2 \bullet \cdots (f_n \bullet f_{n+1}) \cdots)$. We assume that pipes associate to the right, so this form is simply written as $f_1 \bullet f_2 \bullet \cdots f_n \bullet f_{n+1}$. Notice also that the traditional composition of two functions is a pipe of the functions as well. We can also curry the functions and the pipe is still well-defined.

3 An example language and combinatorization

3.1 An example language – a skeleton

We are interested in transforming the **goto** command defined in the naive denotational semantics style ([2]). Therefore our example language should have quite a complex control structure while other constructs, like expressions, basic commands and declarations, should be as general as possible. We do not specify such constructs, so the method should work for almost any extension.

e : *Exp*	– expressions
δ : *Dec*	– declarations
β : *Bas*	– basic commands
l : *Lab*	– labels
c : *Com* = *Bas* |	– commands
Lab : *Com* |	
goto *Lab* |	
Com_1; Com_2 |	
if *Exp* **then** Com_1 **else** Com_2 **fi** |	
Block	
b : *Block* = **block** *Dec* **begin** *Com* **end**	– blocks
p : *Program* = *Block*	

Table 1. Syntax of the language

The syntax of the skeleton language is presented in Table 1. Table 2 shows the semantic domains and functions, while Table 3 defines the semantics of the language. We assume that basic commands, *Bas*, do not change the control flow. So *Bas* may contain assignment statements, input/output operations etc. Expressions in our language are side-effect-free. They are used in conditional commands and therefore the set of expressed values, *Data*, must include the truth values **true** and **false**. *Data* might be *Bool* | *Int* | *Real*....

Domains

d :	$Data$	– expressed values
σ :	$State$	– environments/store (target states)
ι :	$Mode = Lab \mid nil$	– searching/executing modes

Functions

$$\mathcal{E} : Exp \to State \rightharpoonup Data$$
$$\mathcal{P} : Program \to State \rightharpoonup State$$
$$\mathcal{C} : Com \to Mode \to State \rightharpoonup Mode \times State$$
$$\mathcal{K} : Bas \to State \rightharpoonup State$$
$$\mathcal{D} : Dec \to State \rightharpoonup State$$
$$\overline{\mathcal{D}} : Dec \to State \rightharpoonup State$$
$$\mathcal{J} : Com \to \text{FIN}(Lab)$$

Auxiliary functions

$$\text{PASS} = \lambda\iota\sigma.(\iota, \sigma)$$
$$\text{EVAL} = \lambda\iota\sigma.(\iota = nil) \to \sigma, ?$$
$$executing = \lambda\iota.(\iota = nil) \to \textbf{true}, \textbf{false}$$
$$searching = \lambda\iota.(\iota \in Lab) \to \textbf{true}, \textbf{false}$$

Table 2. Domains and functions

The abstract machine defininig the semantics is always in one of two modes: executing or searching for a label. Therefore its state consists of two components: environment/store and mode. The environment/store remains undefined. So it should be possible to apply our method to the common ways of defining them:

- the environment maps identifiers to values, the store corresponds to the state of input and output,
- the environment maps identifiers to locations, while the store maps the latter ones to values.

The function $\overline{\mathcal{D}}$ is to undo declarations, that is to restore the old environment (and possibly change the store by releasing some locations). In usual applications $\mathcal{D}[\![\delta]\!] \circ \overline{\mathcal{D}}[\![\delta]\!]$ is the identity function; we do not require this property, though.

The function $\mathcal{J}$ returns all labels defined and visible within a command. The function $Loop$ in the definition of a block is well-defined since it is built by means of pipes and IF combinators. (Notice that here we regard all total functions as partial functions).

The goal of our transformations is to find a target-machine code equivalent to the original program for a machine which will always be in the executing mode. That is, we would like to find a code which does not require the mode component in order to produce correct answers.

Definition of J

$$
\begin{aligned}
&J[\![l : c]\!] = \{l\} \cup J[\![c]\!] \\
&J[\![c_1 ; c_2]\!] = J[\![c_1]\!] \cup J[\![c_2]\!] \\
&J[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi}]\!] = J[\![c_1]\!] \cup J[\![c_2]\!] \\
&J[\![c]\!] = \emptyset \quad \text{for all other commands}
\end{aligned}
$$

Semantic clauses

$$
\begin{aligned}
&P[\![p]\!] = C[\![p]\!] \bullet \text{EVAL} \,.\, nil \\
&C[\![\beta]\!] = \text{IF } executing \\
&\qquad \text{THEN } \lambda\iota.K[\![\beta]\!] \bullet (\text{PASS} \,.\, \iota) \\
&\qquad \text{ELSE PASS} \\
&\qquad \text{FI} \\
&C[\![l : c]\!] = \lambda\iota.(\iota = l) \rightarrow C[\![c]\!] \,.\, nil, \\
&\qquad\qquad (\iota \in Lab) \rightarrow \text{PASS} \,.\, \iota, \\
&\qquad\qquad \text{OTHERWISE} \rightarrow C[\![c]\!] \,.\, \iota \\
&C[\![\textbf{goto } l]\!] = \lambda\iota.(\iota = nil) \rightarrow \text{PASS} \,.\, l, \text{PASS} \,.\, \iota \\
&C[\![c_1 ; c_2]\!] = C[\![c_1]\!] \bullet C[\![c_2]\!] \\
&C[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi}]\!] = \text{IF } executing \\
&\qquad\qquad \text{THEN} \\
&\qquad\qquad\quad \text{IF EVAL} \bullet \mathcal{E}[\![e]\!] \\
&\qquad\qquad\qquad \text{THEN} \\
&\qquad\qquad\qquad\quad C[\![c_1]\!] \bullet (\text{IF } searching \text{ THEN } C[\![c_2]\!] \text{ ELSE PASS FI}) \\
&\qquad\qquad\qquad \text{ELSE } C[\![c_2]\!] \\
&\qquad\qquad\quad \text{FI} \\
&\qquad\qquad \text{ELSE } C[\![c_1]\!] \bullet (\text{IF } searching \text{ THEN } C[\![c_2]\!] \text{ ELSE PASS FI}) \\
&\qquad\qquad \text{FI} \\
&C[\![\textbf{block } \delta \textbf{ block } c \textbf{ end}]\!] = \text{IF } executing \\
&\qquad\qquad \text{THEN} \\
&\qquad\qquad\quad \text{EVAL} \bullet D[\![\delta]\!] \bullet (\text{PASS} \,.\, nil) \bullet \\
&\qquad\qquad\qquad C[\![c]\!] \bullet Loop \bullet (\lambda\iota.\overline{D}[\![\delta]\!] \bullet (\text{PASS} \,.\, \iota)) \\
&\qquad\qquad \text{ELSE PASS} \\
&\qquad\qquad \text{FI} \\
&\qquad\qquad \text{where} \\
&\qquad\qquad\quad Loop = \text{FIX}(\lambda\theta.\lambda\iota.(\iota \in J[\![c]\!]) \rightarrow C[\![c]\!] \bullet \theta.\iota, \text{PASS} \,.\, \iota)
\end{aligned}
$$

Table 3. Semantic clauses and definition of J

Example

The following program will be used to illustrate the method:

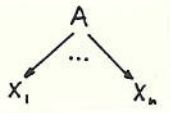
```
block δ1
  begin
    l1: β1;
        β2;
    l2: β3;
      block δ2
        begin
            if e
                then goto l3
                else β4;
                     goto l2
            fi;
          l3: β5
        end;
      goto l1
  end
```

3.2 Combinators and combinator graphs

The first step in our method is to find a more manageable structure of the generated code. We would like to distinguish executable (target) parts of the code and the control structure. Therefore we introduce combinators and auxiliary functions shown in Table 4. The control structure will be some composition of combinators which we represent in a form of a directed graph with

for $f \bullet g$

and



for $A(X_1, \ldots X_n)$

The general form of combinator graphs is not really useful for our transformations. In Section 4 we introduce *linear trees* of a very simple regular form which makes structural induction possible in proving correctness of the method. Intuitively, a graph for a block forms a linear tree if all combinators referencing to labels are located on the rightmost path of the graph. So evaluation in the searching mode is carried out along this path.

Property 2 . Eliminating PASS

$$\text{PASS} \bullet f = f$$

Proof: The property holds since PASS is a total function ∎

$$
\begin{aligned}
&\mathrm{W}(f) = \text{IF } \textit{executing} \text{ THEN } f \text{ ELSE PASS FI}\\
&\overline{\mathrm{W}}(f) = \text{IF } \textit{searching} \text{ THEN } f \text{ ELSE PASS FI}\\
&\mathrm{T}(f,g) = \text{IF } \textit{executing} \text{ THEN } f \text{ ELSE } g \text{ FI}\\
&\text{LABEL } \theta\,(f) = \mathrm{FIX}(\lambda\theta.f)\\
&\mathrm{X} = \lambda xy.(y,x)\\
&\mathrm{I} = \lambda x.x\\
&\mathrm{R}(J)(f,g) = \lambda\iota.(\iota \in J) \rightarrow f.\iota, g.\iota\\
&\mathrm{TIF}(e,f,g) = \text{IF } \textit{executing}\\
&\qquad\qquad \text{THEN}\\
&\qquad\qquad\quad \text{IF EVAL} \bullet e \text{ THEN } f \bullet \overline{\mathrm{W}}(g) \text{ ELSE } g \text{ FI}\\
&\qquad\qquad \text{ELSE } f \bullet \overline{\mathrm{W}}(g)\\
&\qquad\qquad \text{FI}\\
&\mathrm{GOTO}[\![l]\!] = \lambda\iota.(\iota = \textit{nil}) \rightarrow l, \iota\\
&\mathrm{REF}[\![l]\!] = \lambda\iota.(\iota = l) \rightarrow \textit{nil}, \iota\\
&\mathrm{BRA}[\![l]\!](e) = \text{IF } \textit{executing}\\
&\qquad\qquad \text{THEN}\\
&\qquad\qquad\quad \text{IF EVAL} \bullet e \text{ THEN PASS ELSE GOTO}[\![l]\!] \bullet \text{PASS FI}\\
&\qquad\qquad \text{ELSE PASS}\\
&\qquad\qquad \text{FI}\\
&\mathrm{CASE}_n(l_1 \rightarrow f_1, \ldots l_n \rightarrow f_n, g) = \lambda\iota.(\iota = l_1) \rightarrow f_1.\textit{nil},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\iota = l_n) \rightarrow f_n.\textit{nil},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{OTHERWISE} \rightarrow g.\iota\\
&\mathrm{TEST}(e)(f,g) = \text{IF } \textit{executing}\\
&\qquad\qquad\quad \text{THEN}\\
&\qquad\qquad\qquad \text{IF EVAL} \bullet e \text{ THEN } g \text{ ELSE } f \text{ FI}\\
&\qquad\qquad\quad \text{ELSE } g\\
&\qquad\qquad\quad \text{FI}
\end{aligned}
$$

Table 4. Combinators

Property 3. Eliminating X

$$\mathrm{X} \bullet \mathrm{X} \bullet f = f$$

Proof: X is total, so

$$\mathrm{X} \bullet \mathrm{X} \bullet f.x_1x_2x_3\ldots x_k = \mathrm{X} \bullet f.x_2x_1x_3\ldots x_k = f.x_1x_2x_3\ldots x_k \qquad \blacksquare$$

Property 4.

If $f : A \Rightarrow A$ and $g : B \rightarrow A \Rightarrow B \times A$, then

$$\mathrm{X} \bullet f \bullet \mathrm{X} = \lambda\iota.f \bullet (\mathrm{PASS}.\iota)$$
$$\mathrm{W}(\mathrm{EVAL} \bullet f \bullet (\mathrm{PASS}.\textit{nil}) \bullet g) = \mathrm{W}(\mathrm{X} \bullet f \bullet \mathrm{X} \bullet g)$$

Proof:

$$\mathrm{X} \bullet f \bullet \mathrm{X}.\iota\sigma = f \bullet \mathrm{X}.\sigma\iota$$
$$= \begin{cases} ? & \text{if } f.\sigma = ?\\ \mathrm{X}.(f.\sigma)\iota & \text{otherwise} \end{cases}$$

$$= \begin{cases} ? & \text{if } f.\sigma = ? \\ (\text{PASS}.\iota).(f.\sigma) & \text{otherwise} \end{cases}$$
$$= f \bullet (\text{PASS}.\iota).\sigma$$
$$= (\lambda\iota.f \bullet (\text{PASS}.\iota)).\iota\sigma$$

$$\text{EVAL} \bullet f \bullet (\text{PASS}.nil) \bullet g.nil\sigma = f \bullet (\text{PASS}.nil) \bullet g.\sigma$$
$$= \text{X} \bullet f \bullet \text{X} \bullet g.nil\sigma$$

and since the argument of W is evaluated in the executing mode only, the property holds.∎

3.3 Combinatorization and partial linearization

Using combinators defined in the previous section, we can express semantic clauses in a combinatorial form. They are shown in Table 5; we transform a few of them. Denotations of programs form trees; the combinator tree for our example program is shown in Figure 6. Notice that due to the right associativity of pipes (and of the union in the definition of J) some parts of the code are already linearized.

$C[\![\beta]\!] = \text{W}(\text{X} \bullet K[\![\beta]\!] \bullet \text{X})$
$C[\![l : c]\!] = \text{REF}[\![l]\!] \bullet C[\![c]\!]$
$C[\![\textbf{goto } l]\!] = \text{GOTO}[\![l]\!] \bullet \text{PASS}$
$C[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi}]\!] = \text{TIF}(\mathcal{E}[\![e]\!], C[\![c_1]\!], C[\![c_2]\!])$
$C[\![\textbf{block } \delta \textbf{ begin } c \textbf{ end}]\!] = \text{W}(\text{X} \bullet D[\![\delta]\!] \bullet \text{X} \bullet \text{LABEL } \theta(C[\![c]\!] \bullet \text{R}(J[\![c]\!])(\theta, \text{X} \bullet \overline{D}[\![\delta]\!] \bullet \text{X})))$
where θ is a unique variable

Table 5. Combinatorized semantic equations

$C[\![\beta]\!] =$ IF *executing*
THEN $\lambda\iota.K[\![\beta]\!] \bullet (\text{PASS}.\iota)$
ELSE PASS
FI
$=$ IF *executing*
THEN $\text{X} \bullet K[\![\beta]\!] \bullet \text{X}$ by Property 4
ELSE PASS
FI
$= \text{W}(\text{X} \bullet K[\![\beta]\!] \bullet \text{X})$

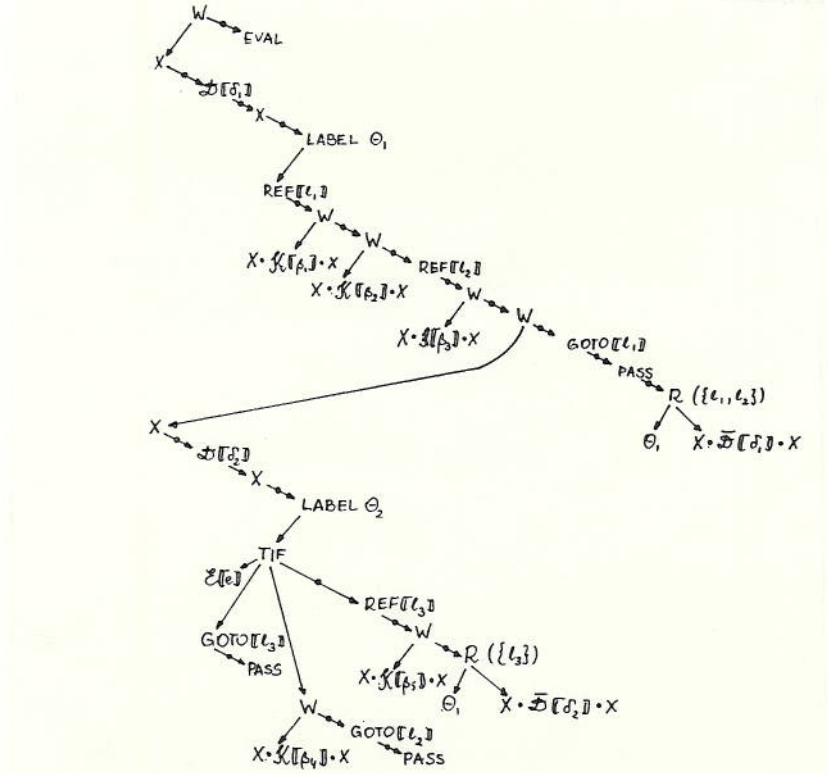


Figure 6. Combinator tree

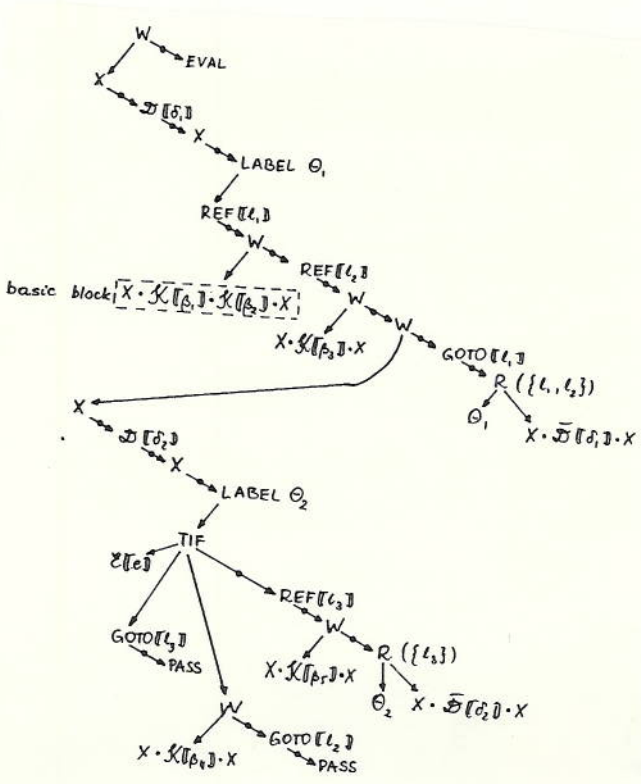


Figure 7. Combinator tree with basic blocks

$$
\begin{aligned}
&C[\![\textbf{block } \delta \textbf{ begin } c \textbf{ end}]\!] \\
&\quad = \text{IF } \textit{executing} \\
&\qquad \text{THEN} \\
&\qquad\quad \text{EVAL} \circ D[\![\delta]\!] \circ (\text{PASS} \,.\, \textit{nil}) \circ C[\![c]\!] \circ \\
&\qquad\qquad\qquad \text{FIX}(\lambda\theta.\lambda\iota.(\iota \in J[\![c]\!]) \rightarrow C[\![c]\!] \circ \theta.\iota, \text{PASS} \,.\, \iota) \circ \\
&\qquad\qquad\qquad (\lambda\iota.\overline{D}[\![\delta]\!] \circ (\text{PASS} \,.\, \iota)) \\
&\qquad \text{ELSE PASS} \\
&\qquad \text{FI} \\
&\quad = \text{IF } \textit{executing} \\
&\qquad \text{THEN} \\
&\qquad\quad \text{X} \circ D[\![\delta]\!] \circ \text{X} \circ C[\![c]\!] \circ \\
&\qquad\qquad\qquad \text{FIX}(\lambda\theta.\lambda\iota.(\iota \in J[\![c]\!]) \rightarrow C[\![c]\!] \circ \theta.\iota, \text{PASS} \,.\, \iota) \circ \\
&\qquad\qquad\qquad \text{X} \circ \overline{D}[\![\delta]\!] \,\text{X} \\
&\qquad \text{ELSE PASS} \\
&\qquad \text{FI} \\
&\quad = \text{W}(\text{X} \circ D[\![\delta]\!] \circ \text{X} \circ C[\![c]\!] \circ \\
&\qquad\qquad\qquad \text{FIX}(\lambda\theta.\lambda\iota.(\iota \in J[\![c]\!]) \rightarrow C[\![c]\!] \circ \theta.\iota, \text{PASS} \,.\, \iota) \circ \\
&\qquad\qquad\qquad \text{X} \circ \overline{D}[\![\delta]\!] \circ \text{X}) \\
&\quad = \text{W}(\text{X} \circ D[\![\delta]\!] \circ \text{X} \circ \\
&\qquad\quad \text{LABEL } \theta(C[\![c]\!] \circ (\lambda\iota.(\iota \in J[\![c]\!]) \rightarrow \theta.\iota, \text{PASS} \,.\, \iota)) \circ \\
&\qquad\quad \text{X} \circ \overline{D}[\![\delta]\!] \circ \text{X}) \\
&\quad = \text{W}(\text{X} \circ D[\![\delta]\!] \circ \text{X} \circ \\
&\qquad\quad \text{LABEL } \theta(C[\![c]\!] \circ \text{R}(J[\![c]\!])(\theta, \text{PASS})) \circ \\
&\qquad\quad \text{X} \circ \overline{D}[\![\delta]\!] \circ \text{X}) \\
&\quad = \text{W}(\text{X} \circ D[\![\delta]\!] \circ \text{X} \circ \text{LABEL } \theta(C[\![c]\!] \circ \text{R}(J[\![c]\!])(\theta, \text{X} \circ \overline{D}[\![\delta]\!] \circ \text{X})))
\end{aligned}
$$

where the last transformations can be proved by unfolding the fixpoint expressions, using for instance Böhm trees ([1,10]).

The combinators have the following property useful to group basic commands in blocks which do not alter the control structure:

Property 5 . Creating basic blocks

$$\text{W}(\text{X} \circ K[\![\beta_1]\!] \circ \text{X}) \circ \text{W}(\text{X} \circ K[\![\beta_2]\!] \circ \text{X}) \circ \cdots \text{W}(\text{X} \circ K[\![\beta_n]\!] \circ \text{X}) = \text{W}(\text{X} \circ K[\![\beta_1]\!] \circ K[\![\beta_2]\!] \circ \cdots K[\![\beta_n]\!] \circ \text{X})$$

We make also use of Property 2 to eliminate some PASS'es introduced by combinatorizing **goto**'s. Both these properties make the tree for next transformations shorter. Figure 7 shows such a tree for our example.

4 Linear trees

4.1 Elements of linear trees

Combinator graphs corresponding to programs are quite complex. When we analyze the control structure of a program, we are not really interested in the detailed structure of blocks of basic commands, expressions or even inner blocks; they only increase the level of difficulties. Therefore we introduce a special class of combinator graphs that reflect only the control structure of a single block. Their form is almost linear, and we

call them *linear trees*. Since they are defined recursively, we can use structural induction to prove their properties.

In linear trees we would like to express the fact that some labels have not been used in a given part of a program. This is easy when we know the entire structure of a combinator graph. If we want to abstract from inner blocks, we must provide a way to guarantee that some labels cannot be produced within those blocks. Therefore we have a combinator:

$$\mathrm{EX}(J) = \lambda\iota\sigma.(\iota \in J) \to ?, (\iota, \sigma)$$

To say that α cannot result in referencing l_1 or l_2, we may write that $\alpha = \alpha \bullet \mathrm{EX}(\{l_1, l_2\})$. In our applications, the set of labels in $\mathrm{EX}(J)$ will be always finite. To shorten the code of basic blocks, let $\mathrm{E}(f) = \mathrm{W}(\mathrm{X} \bullet f \bullet \mathrm{X})$.

Linear trees are built of EX,E, combinators defined in Section 2, labels from *Lab* and variables of the following sorts:

- θ, θ', θ_0, θ_1,... ranging over $Mode \to State \Rightarrow Mode \times State$,
- ϵ, ϵ', ϵ_0, ϵ_1,... ranging over $Mode \to State \Rightarrow Data$; they replace expressions $\mathrm{EVAL} \bullet \mathcal{E}[\![e]\!]$,
- η, η', η_0, η_1,... ranging over $State \Rightarrow State$; they are used as substitutions for $\mathrm{W}(\mathrm{X} \bullet \mathcal{K}[\![\beta_1]\!] \bullet \cdots \mathcal{K}[\![\beta_n]\!] \bullet \mathrm{X})$ to the form of $\mathrm{E}(\eta)$,
- κ, κ', κ_0, κ_1,... ranging over $Mode \to State \Rightarrow Mode \times State$ which are used in place of inner blocks.

To end this introductory part, let us define graphs free of some combinators. For any combinator Z a combinator graph is Z-*free* (or *free of* Z) if it does not contain Z. So we talk about $\mathrm{REF}[\![l]\!]$-free, GOTO-free graphs and so on.

4.2 Linear trees

Internal *nodes* of linear tress are of the following forms:

- $\mathrm{E}(\eta)$,
- $\mathrm{W}(\kappa)$, $\mathrm{W}(\kappa \bullet \mathrm{EX}(J))$ where J is a finite subset of *Lab*; we call them *blocks* since they correspond to inner blocks,
- $\mathrm{GOTO}[\![l]\!]$,
- $\mathrm{REF}[\![l]\!]$,
- $\mathrm{BRA}[\![l]\!](\epsilon)$.

Linear trees are defined recursively. Let α' denote a linear tree in the recursive part of the definition. A finite combinator tree α is a *linear tree* if it is of one of the following forms:

- $\alpha = \mathrm{PASS}$,
- $\alpha = \beta \bullet \alpha'$ where β is a node,
- $\alpha = \mathrm{LABEL}\ \theta(\alpha')$.

As for other acyclic graphs, we can define the *rightmost path* of a linear tree. We use $\alpha \mapsto Z$ to denote a combinator graph (not necessarily a linear tree) which is formed

by extending the rightmost path of a linear tree α by a (rooted) graph Z, that is when the rightmost node of α is "piped" with the root (distinguished node) of Z. Since the rightmost node is PASS, Z replaces it rather than follows it. As a convention we assume that $\alpha \rightarrowtail \theta_0$ means that θ_0 does not occur in α at all, unless otherwise stated. Figure 8 shows such a composition.

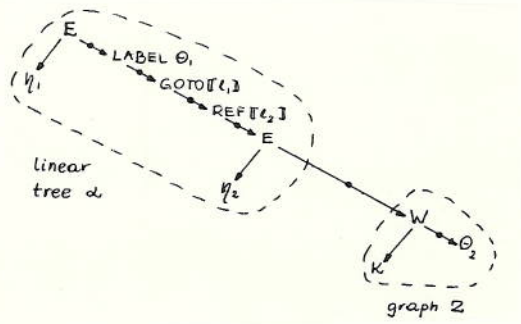


Figure 8. Composition $\alpha \rightarrowtail Z$

A variable θ is called a *free variable* in a combinator graph if it is not bound by any LABEL combinator in this graph. To make our proofs easier, we assume that if a linear tree contains a subtree LABEL $\theta(\alpha)$ then θ should not occur outside this subtree. In particular, LABEL combinators should bind distinct variables. We call such trees *well-formed* and from now on we are interested in well-formed trees only.

A linear tree is *without* l iff:

- every block is of the form of $W(\kappa \bullet EX(J))$ and $l \in J$,
- the tree is free of REF⟦l⟧, GOTO⟦l⟧, BRA⟦l⟧.

The following properties can be easily proved by structural induction on α.

Property 6 . Evaluation

If α is a linear tree, then for any ι, σ:

$$\alpha \rightarrowtail \theta_0.\iota\sigma = ? \qquad \text{or} \qquad \alpha \rightarrowtail \theta_0.\iota\sigma = \theta_0.\iota'\sigma'$$

Property 7 . Evaluation of search

Let α be a REF⟦l⟧-free linear tree. Then for any state σ:

$$\alpha \rightarrowtail \theta_0.l\sigma = \theta_0.l\sigma$$

This property says that if REF⟦l⟧ does not appear in a linear tree, then the search for l is carried out along the righmost path and terminates in the rightmost node.

Property 8 . Unobtainable labels

Let α be a linear tree without l. For every $\iota \neq l, \sigma$ if $\alpha \rightarrowtail \theta_0.\iota\sigma = \theta_0.\iota'\sigma'$ then $\iota' \neq l$.

This property says that a tree without l cannot generate jumps to l.

Property 9 . Composition

If all θ variables in a linear tree α are free in a rooted combinator graph β and all bound variables in β do not occur in α, then $\alpha \rightarrowtail \beta$ is well-formed.

If α and β satisfy the assumptions, we say that α is *composable* with β. If β is a linear tree, then $\alpha \rightarrowtail \beta$ is also a linear tree.

4.3 Linearization of conditionals

Our combinator trees are still far from being linear, mainly because of TIF combinators used in conditional commands. Since our source language contains jumps, we can use them to linearize both branches of a TIF combinator and replace this combinator by a BRA combinator with two unique labels.

In the next three properties we assume that:

- α, β are two linear trees without l_1, l_2,
- α is composable with β.

Property 10 .

For any $\iota \neq l_1$

$$\alpha \rightarrowtail \mathrm{W}(\beta).\iota = \alpha \rightarrowtail \mathrm{GOTO}[\![l_2]\!] \bullet \mathrm{REF}[\![l_1]\!] \bullet \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.\iota$$

Proof: The only interesting case is when $\alpha \rightarrowtail \theta_0.\iota\sigma = \theta_0.\iota'\sigma'$. Since α is without l_1, l_2, then $\iota' \neq l_1$ and $\iota' \neq l_2$. Let us consider two cases: $\iota' = nil$ and $\iota' \neq nil$.

Case 1. The lefthand side evaluates to (nil, σ'). The evaluation of the righthand side is:

$$\begin{aligned}
&\mathrm{GOTO}[\![l_2]\!] \bullet \mathrm{REF}[\![l_1]\!] \bullet \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.nil\sigma' = \\
&\quad \mathrm{REF}[\![l_1]\!] \bullet \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.l_2\sigma' = \\
&\quad \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.l_2\sigma' = \\
&\quad \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.l_2\sigma' = \\
&\quad \mathrm{PASS}.nil\sigma' = (nil, \sigma')
\end{aligned}$$

Case 2. $\iota' \neq nil$

The lefthand side evaluates to $\beta.\iota'\sigma'$. And the righthand side:

$$\begin{aligned}
&\mathrm{GOTO}[\![l_2]\!] \bullet \mathrm{REF}[\![l_1]\!] \bullet \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.\iota'\sigma' = \\
&\quad \mathrm{REF}[\![l_1]\!] \bullet \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.\iota'\sigma' = \\
&\quad \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.\iota'\sigma' = \\
&\quad \beta.\iota'\sigma'
\end{aligned}$$

since β is without l_2 and $\iota' \neq l_2$. ∎

Property 11 .

$$\beta.nil = \mathrm{GOTO}[\![l_1]\!] \bullet \alpha \rightarrowtail \mathrm{GOTO}[\![l_2]\!] \bullet \mathrm{REF}[\![l_1]\!] \bullet \beta \rightarrowtail \mathrm{REF}[\![l_2]\!] \bullet \mathrm{PASS}.nil$$

Proof: Proof is similar to Fact 10. ∎

Corollary 12 .

For unique labels l_1, l_2

$$\text{TIF}(\epsilon, \alpha, \beta) = \text{BRA}[\![l_1]\!](\epsilon) \bullet \alpha \;\mapsto\; \text{GOTO}[\![l_2]\!] \bullet \text{REF}[\![l_1]\!] \bullet \beta \;\mapsto\; \text{REF}[\![l_2]\!] \bullet \text{PASS}$$

and the righthand side is a linear tree.

Property 13 .

Let α be any linear tree, β be any combinator graph and α be composable with β. Then

$$(\text{BRA}[\![l]\!](\epsilon) \bullet \alpha) \bullet \beta = \text{BRA}[\![l]\!](\epsilon) \bullet \alpha \;\mapsto\; \beta$$

and by Property 2 we can eliminate the rightmost PASS in α.

Figure 9 shows linearization of a TIF.

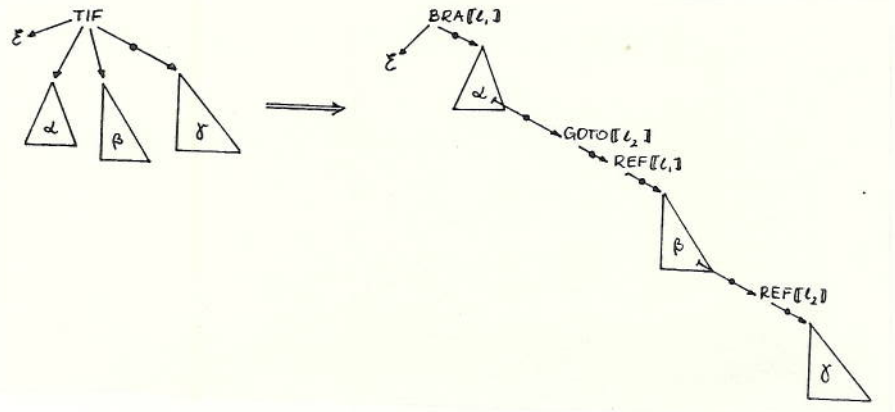


Figure 9. Linearization of TIF

To avoid the problem of generating unused labels, we can introduce a new auxiliary set of internal labels disjoint with the set of labels available to programs. The set *Lab* should then be the union of internal labels and user labels. We assume that we can use a function that generates a new unused element from the set of internal labels.

By Fact 13, we can define a function s_1 to linearize the code. This function should have the following property:

Property 14 . Linearization of conditionals

Let l_1 and l_2 be unique (unused) internal labels for $s_1(f), s_1(g)$ and $s_1(h)$. Then

$$s_1(\text{TIF}(e, f, g) \bullet h) = \text{BRA}[\![l_1]\!](e) \bullet s_1(f) \;\mapsto\; \text{GOTO}[\![l_2]\!] \bullet \text{REF}[\![l_1]\!] \bullet s_1(g) \;\mapsto\; \text{REF}[\![l_2]\!] \bullet s_1(h)$$

Proof: After applying f, g and h to s_1, the results are trees without l_1 and l_2. We construct linear trees for $s_1(f), s_1(g)$ in the following way:

– all EVAL $\bullet \mathcal{E}[\![e]\!]$ expressions are replaced by (unique) ϵ variables,

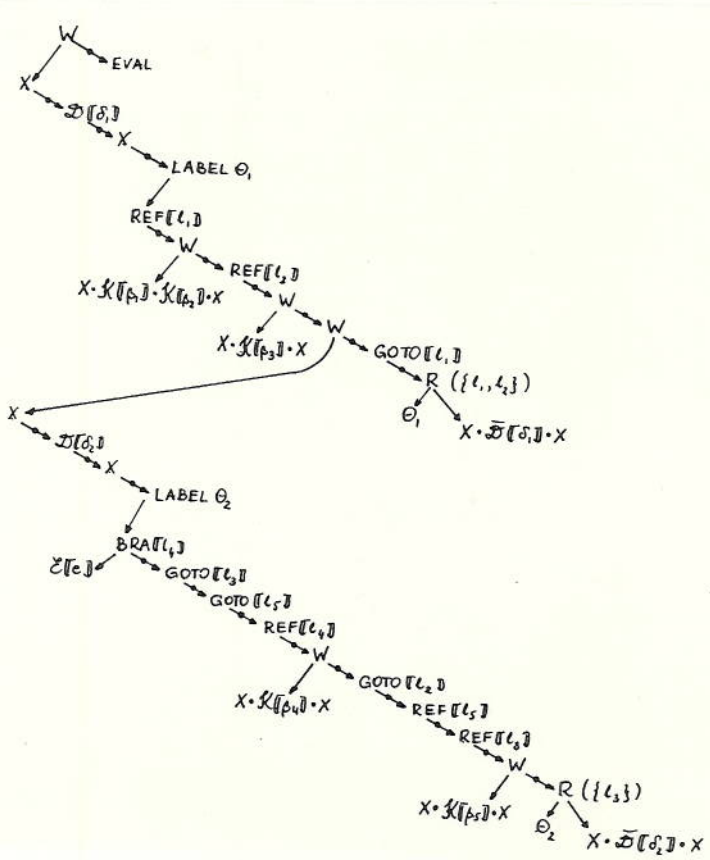


Figure 10. Linear code

- all blocks of basic commands are replaced by $\mathrm{E}(\eta)$ expressions,
- all inner blocks are replaced by $\mathrm{W}(\kappa \bullet \mathrm{EX}(\{l_1, l_2\}))$.

Now by Property 13 we can produce a linear tree equivalent to TIF combining these trees. By a reverse substitution, we get the result. ∎

Figure 10 shows the linear code (linear trees for each block) for our example program. Of course, during linearization we make use of the right associativity of pipes.

4.4 Introduction of new labels

Having the code with linearized conditionals, we are able to carry out the first transformation that leads us closer to the target code. Our goal is to eliminate the searching mode, so, in effect, to remove the fixed-point definition of a block. Some recursive equations will still be necessary since jumps backwards introduce loops in our programs. We can use the following property of λ-calculus:

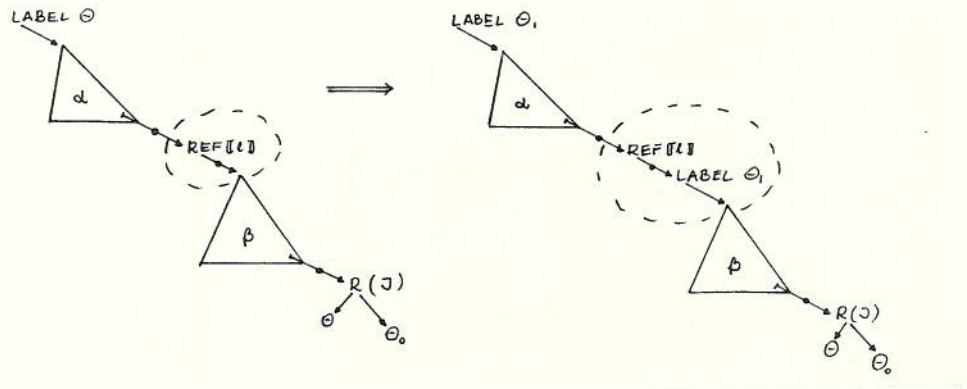


Figure 11. Introduction of new labels

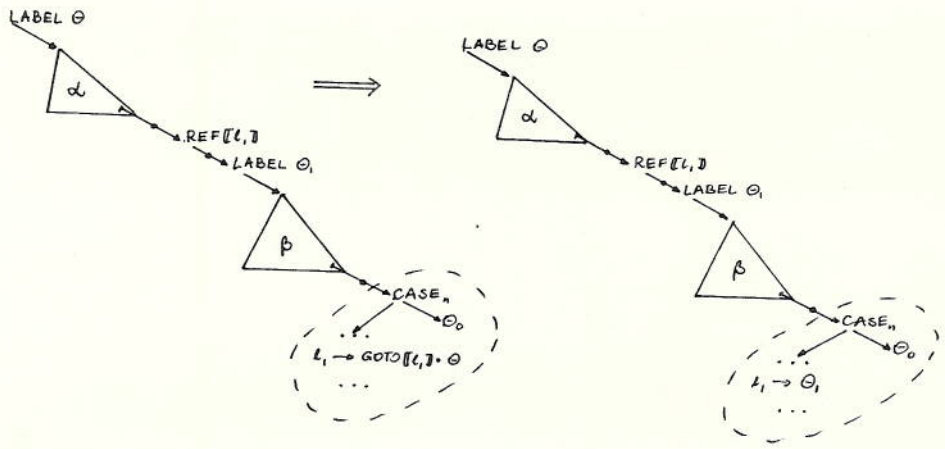


Figure 12. Preparation of jumps

Property 15 .

If θ is not a free variable in α, then $\text{FIX}(\lambda\theta.\alpha) = \alpha$.

So we can define a function s_2 which introduces new LABEL combinators in front of every REF combinator. That is s_2 satisfies:

Property 16 .

$$s_2(\text{REF}[\![l]\!] \bullet \alpha) = \text{REF}[\![l]\!] \bullet \text{LABEL}\ \theta(s_2(\alpha))$$

where θ is a unique variable; s_2 does not change any other constructs.

Now we can combine all the information about jumps within a block. We change $\text{R}(J)(\theta, \alpha)$ using a CASE combinator first and then evaluate the jumps in CASE.

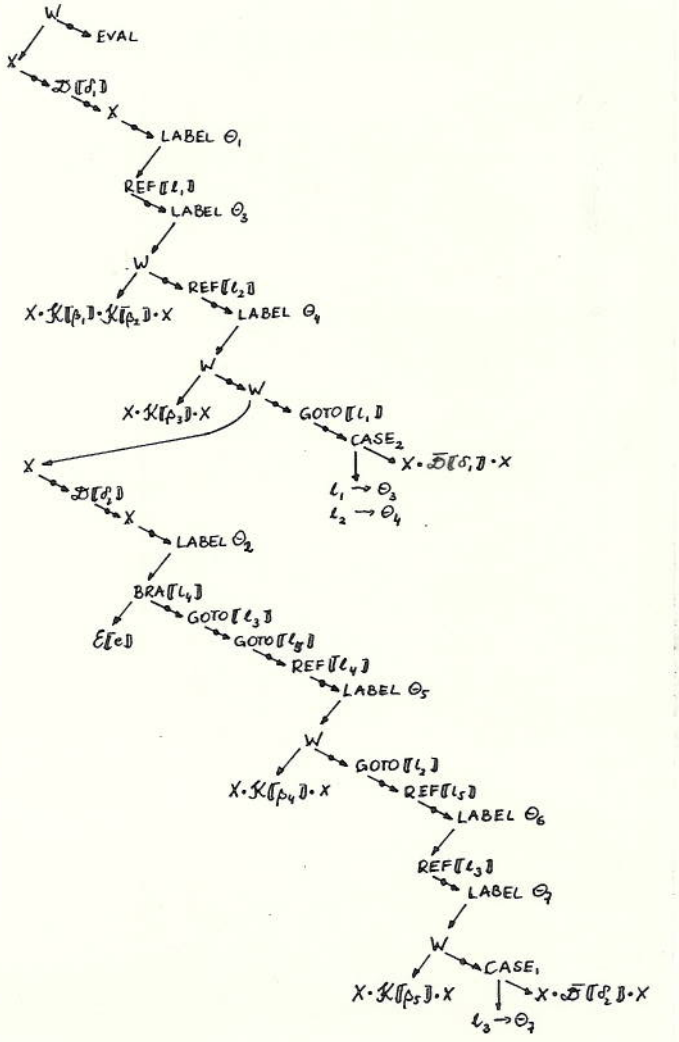


Figure 13. Linear code with conditional exits

Property 17 .

Let $J = \{l_1, \ldots l_n\}$. Then

$$\mathrm{R}(J)(\theta, \alpha) = \mathrm{CASE}_n(l_1 \rightarrow \mathrm{GOTO}[\![l_1]\!] \bullet \theta, \ldots \mathrm{GOTO}[\![l_n]\!] \bullet \theta, \alpha)$$

Property 18 .

Let α be a linear tree free of $\mathrm{REF}[\![l]\!]$, β be a linear tree without θ_1. Then

$$\mathrm{LABEL}\ \theta(\alpha \rightarrowtail \mathrm{REF}[\![l]\!] \bullet \mathrm{LABEL}\ \theta_1(\beta \rightarrowtail \mathrm{CASE}_n(\ldots l \rightarrow \mathrm{GOTO}[\![l]\!] \bullet \theta \ldots, \theta_0))) =$$
$$\mathrm{LABEL}\ \theta(\alpha \rightarrowtail \mathrm{REF}[\![l]\!] \bullet \mathrm{LABEL}\ \theta_1(\beta \rightarrowtail \mathrm{CASE}_n(\ldots l \rightarrow \theta_1 \ldots, \theta_0)))$$

Proof: Since α is REF$[\![l]\!]$-free then GOTO$[\![l]\!] \bullet \alpha \rightarrowtail \theta_0 =$ GOTO$[\![l]\!] \bullet \theta_0$. And we have:

$$\begin{aligned}
&\text{LABEL } \theta(\alpha \rightarrowtail \text{REF}[\![l]\!] \bullet \text{LABEL } \theta_1(\beta \rightarrowtail \text{CASE}_n(\ldots l \rightarrow \text{GOTO}[\![l]\!] \bullet \theta \ldots, \theta_0))) = \\
&\quad \text{LABEL } \theta(\alpha \rightarrowtail \text{REF}[\![l]\!] \bullet \text{LABEL } \theta_1(\beta \rightarrowtail \\
&\qquad \text{CASE}_n(\ldots l \rightarrow \text{GOTO}[\![l]\!] \bullet \text{REF}[\![l]\!] \bullet \text{LABEL } \theta_1(\beta \rightarrowtail \ldots) \ldots, \theta_0))) = \\
&\quad \text{LABEL } \theta(\alpha \rightarrowtail \text{REF}[\![l]\!] \bullet (\beta \rightarrowtail \text{CASE}_n(\ldots l \rightarrow (\beta \rightarrowtail \text{CASE}_n(\ldots, \theta_0)) \ldots, \theta_0)))
\end{aligned}$$

The last transformation holds since θ_1 is not free in β. Now using Böhm trees, we can show that this expression defines the same function as the righthand side in the formulation of this property. (We can also prove that λ and FIX are continuous, and then use this fact for a proof based on Kleene's fixed-point theorem). ∎

Figures 11 and 12 show this transformation and Figure 13 presents the example code without R combinators.

5 Transformations of jumps

5.1 Distribution of CASE

The transformations discussed so far enable us to consider a linear tree for a block as a set of mutually recursive functions:

$$\begin{aligned}
\theta &= \text{LABEL } \theta(\alpha \rightarrowtail \text{REF}[\![l_1]\!] \bullet \theta_1) \\
\theta_1 &= \text{LABEL } \theta_1(\alpha_1 \rightarrowtail \text{REF}[\![l_2]\!] \bullet \theta_2) \\
&\ldots \\
\theta_k &= \text{LABEL } \theta_k(\alpha_k \rightarrowtail \text{CASE}_k(l_1 \rightarrow \theta_1, \ldots l_k \rightarrow \theta_k, \theta_0))
\end{aligned}$$

if the block does not contain any conditionals. In the other case, the CASE combinator does not contain labels introduced during linearization of conditionals.

The linear structure of the block means that all of these definitions appear on the rightmost path. If we use the same θ variable to denote a function defined by LABEL $\theta(f)$, as we have done above, then the use of θ outside f means a reference to the defined functions (compare with [7]). It does not lead to any problems since we assumed that all bound variables in the entire graph are distinct. Therefore the scope of any θ variable is global.

We can distribute the information stored in the CASE combinator up the rightmost path of the code. In the next section we show how to move all visible labels (from the outside blocks) to the rightmost CASE in a block, so let us assume that the CASE combinator contains all the necessary information needed to eliminate GOTO combinators.

Property 19 . Distribution to basic blocks

$$\text{E}(\eta) \bullet \text{CASE}_n(\ldots, \alpha) = \text{CASE}_n(\ldots, \text{E}(\eta) \bullet \alpha)$$

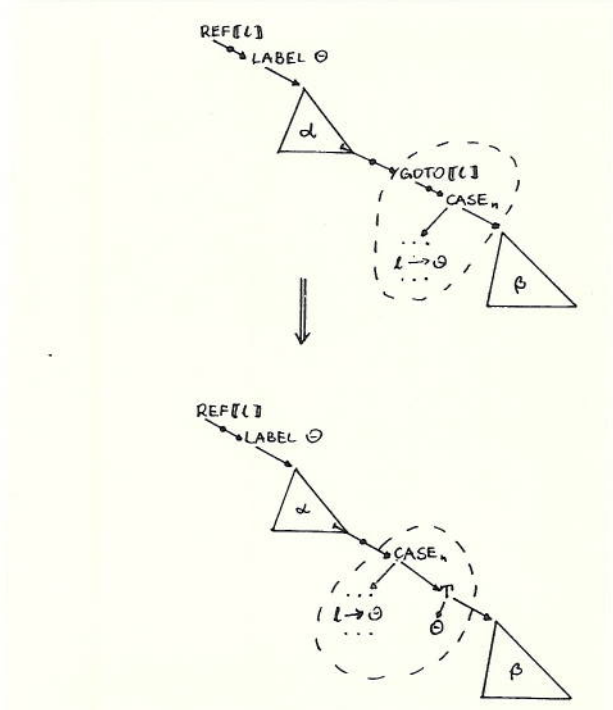


Figure 14. Evaluation of GOTO

Property 20 . Evaluation of REF

$$\text{REF}[\![l]\!] \bullet \text{CASE}_n(\ldots l \to \chi \ldots, \alpha) = \text{CASE}_n(\ldots l \to \alpha \ldots, \alpha)$$

If l does not appear in CASE, then

$$\text{REF}[\![l]\!] \bullet \text{CASE}_n(\ldots, \alpha) = \text{CASE}_{n+1}(\ldots, l \to \alpha, \alpha)$$

Property 21 . Evaluation of GOTO

$$\text{GOTO}[\![l]\!] \bullet \text{CASE}_n(\ldots l \to \chi \ldots, \alpha) = \text{CASE}_n(\ldots l \to \chi \ldots, \text{T}(\chi, \alpha))$$

Figure 14 shows this transformation.

Property 22 . Evaluation of BRA

$$\text{BRA}[\![l]\!](\epsilon) \bullet \text{CASE}_n(\ldots l \to \chi \ldots, \alpha) = \text{CASE}_n(\ldots l \to \chi \ldots, \text{TEST}(\epsilon)(\chi, \alpha))$$

Properties 21 and 22 give the only way to use θ variables in the code. Notice that the only occurence of θ variables is LABEL $\theta(\alpha)$, T(θ, α) or TEST$(\epsilon)(\theta, \alpha)$. So θ variables are always given *nil* and σ as parameters. Therefore the following property holds:

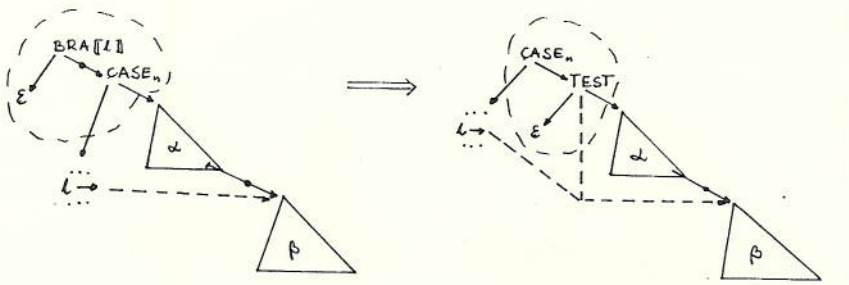


Figure 15. Evaluation of BRA

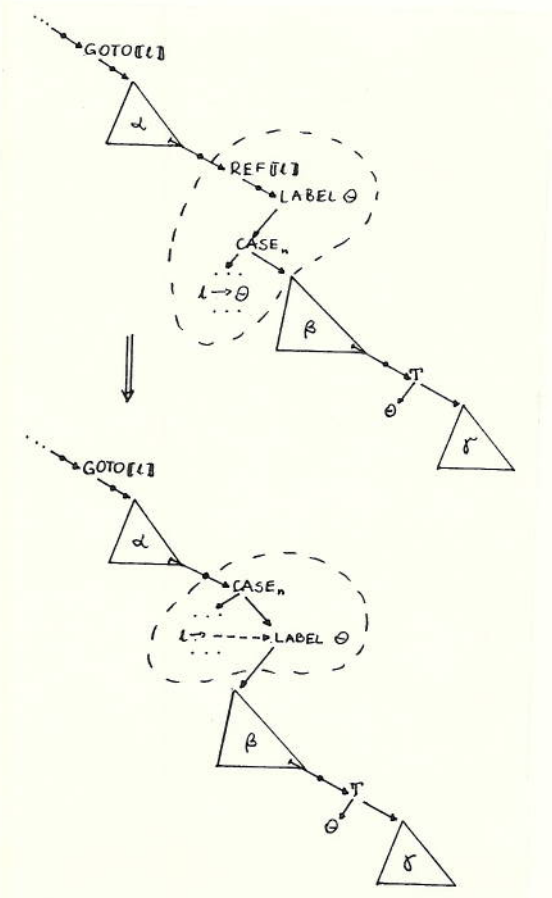


Figure 16. Evaluation of LABEL

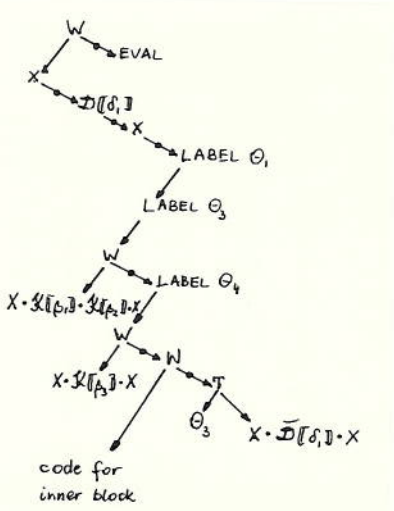


Figure 17. Partial distribution of CASE

Property 23 . Evaluation of LABEL

$$\text{LABEL } \theta(\text{CASE}_n(\ldots l \to \theta \ldots, \alpha)) = \text{CASE}_n(\ldots l \to \text{LABEL } \theta(\alpha) \ldots, \text{LABEL } \theta(\alpha))$$

If θ does not appear in CASE then

$$\text{LABEL } \theta(\text{CASE}_n(\ldots, \alpha)) = \text{CASE}_n(\ldots, \text{LABEL } \theta(\alpha))$$

Figure 15 shows a transformation of BRA and Figure 16 shows evaluation of LABEL and REF.

Let us assume for a while that the CASE combinator is somehow distributed to inner blocks and passed along the rightmost path unchanged. Figure 17 shows the transformations of the outer block from our example program. The distribution finishes by:

Property 24 .

$$\text{W}(\text{X} \bullet \mathcal{D}[\![\delta]\!] \bullet \text{X} \bullet \text{CASE}_n(\ldots, \alpha)) = \text{W}(\text{X} \bullet \mathcal{D}[\![\delta]\!] \bullet \text{X} \bullet \alpha)$$

5.2 Distribution to blocks

The transformations from the previous section are sufficient to remove GOTO and REF combinators within a block. The partially compiled code would still contain GOTO's to the outside blocks. We can eliminate these jumps if we move the CASE combinator from the outside block to the inner block. This is possible due to the following properties:

Property 25 . Composition as extension

Let α be a linear tree composable with graph β, i.e. β does not contain variables bound in α. Then

$$\alpha \bullet \beta = \alpha \rightarrowtail \beta$$

Proof: By structural induction on α. ∎

So we have:

$$(\mathrm{X} \bullet \mathcal{D}[\![\delta]\!] \bullet \mathrm{X} \bullet \alpha \rightarrowtail \mathrm{X} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \mathrm{X}) \bullet \mathrm{CASE}_n(\ldots, \beta) =$$
$$\mathrm{X} \bullet \mathcal{D}[\![\delta]\!] \bullet \mathrm{X} \bullet \alpha \rightarrowtail \mathrm{X} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \mathrm{X} \bullet \mathrm{CASE}_n(\ldots, \beta)$$

Property 26 .

$$\mathrm{X} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \mathrm{X} \bullet \mathrm{CASE}_n(l_1 \rightarrow \chi_1, \ldots l_n \rightarrow \chi_n, \beta) =$$
$$\mathrm{CASE}_n(l_1 \rightarrow \mathrm{X} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \mathrm{X} \bullet \chi_1, \ldots l_n \rightarrow \mathrm{X} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \mathrm{X} \bullet \chi_n, \mathrm{X} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \mathrm{X} \bullet \beta)$$

So we can move the CASE combinator inside the inner block. And we can combine this combinator with the CASE already existing in the block:

Property 27 .

Let $L = \{l_1, \ldots l_n\}$ and $L' = \{l_1', \ldots l_k'\}$. Let $L' - L = \{l_{i_1}', \ldots l_{i_j}'\}$. Then

$$\mathrm{CASE}_n(l_1 \rightarrow \chi_1, \ldots l_n \rightarrow \chi_n, \mathrm{CASE}_k(l_1' \rightarrow \chi_1', \ldots l_k' \rightarrow \chi_k', \alpha)) =$$
$$\mathrm{CASE}_{n+j}(l_1 \rightarrow \chi_1, \ldots l_n \rightarrow \chi_n, l_{i_1}' \rightarrow \chi_{i_1}', \ldots l_{i_j}' \rightarrow \chi_{i_j}', \alpha)$$

By these properties we can distribute the labels defined in the outer blocks to the body of an inner block.

5.3 The target code

During distribution of CASE we can also produce the target code. Notice that before distribution of labels in the outmost block, the code of the entire program is:

$$\mathrm{W}(\kappa) \bullet \mathrm{EVAL} = \mathrm{W}(\kappa) \bullet \mathrm{CASE}_0(\mathrm{EVAL})$$

And because programs are always started in the executing mode, this is equivalent to

$$\mathrm{W}(\kappa) \bullet \mathrm{EVAL} = \kappa \bullet \mathrm{CASE}_0(\mathrm{EVAL})$$

We show that during distribution CASE is of the form of $\mathrm{CASE}_n(\ldots, \mathrm{EVAL} \bullet \alpha)$. First let us introduce additional combinators:

$$\mathrm{L} = \lambda\sigma.(nil, \sigma)$$

and

$$\text{ERROR} = \lambda\sigma.?$$

Notice that $\text{L} \bullet \text{EVAL} \bullet \alpha = \alpha$. To preserve the fact that all important combinators are on the rightmost path, we introduce:

$$\text{GLUE}(f, g) = f$$

and

$$\overline{\text{IF}}(e, f, g) = \text{IF } e \text{ THEN } g \text{ ELSE } f \text{ FI}$$

Let us consider the code produced by distribution of CASE.

Property 28 . Undefined labels

If l does not appear in CASE, then

$$\text{GOTO}[\![l]\!] \bullet \text{CASE}_n(\ldots, \text{EVAL} \bullet \alpha) = \text{CASE}_n(\ldots, \text{T}(\text{EVAL} \bullet \text{ERROR}, \text{EVAL} \bullet \alpha))$$

Property 29 . Evaluation of basic blocks

$$\text{E}(\eta) \bullet \text{EVAL} \bullet \alpha = \text{EVAL} \bullet (\eta \bullet \alpha)$$

and by the right associativity of pipes, we can linearize the basic commands with α.

Property 30 .

$$\text{T}(\chi, \text{EVAL} \bullet \alpha) = \text{EVAL} \bullet \text{GLUE}(\text{L} \bullet \chi, \alpha)$$

Property 31 .

$$\text{TEST}(\epsilon)(\chi, \text{EVAL} \bullet \alpha) = \text{EVAL} \bullet \overline{\text{IF}}(\epsilon, \text{L} \bullet \chi, \alpha)$$

Property 32 .

$$\text{LABEL } \theta(\text{EVAL} \bullet \alpha) = \text{EVAL} \bullet \text{LABEL } \theta(\alpha[\text{EVAL} \bullet \theta/\theta])$$

Notice that all occurrences of θ are inside α. They have been changed (Property 30 and 31) to $\text{L} \bullet \theta$, so we have $\text{L} \bullet \text{EVAL} \bullet \theta = \theta$.

Property 33 .

$$\text{X} \bullet \mathcal{D}[\![\delta]\!] \bullet \text{X} \bullet \text{EVAL} \bullet \alpha = \text{EVAL} \bullet \mathcal{D}[\![\delta]\!] \bullet \alpha$$
$$\text{X} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \text{X} \bullet \text{EVAL} \bullet \alpha = \text{EVAL} \bullet \overline{\mathcal{D}}[\![\delta]\!] \bullet \alpha$$

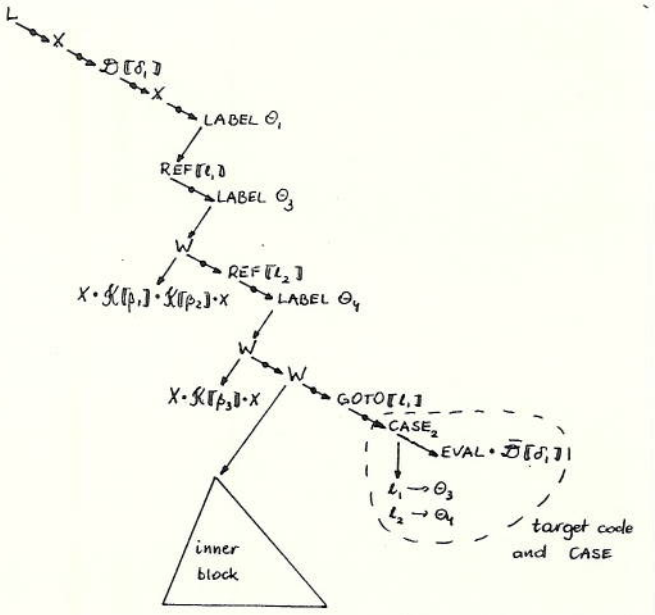


Figure 18. Distribution in the outer block – 1

Property 34 .

$$\mathrm{W}(\kappa) \bullet \mathrm{CASE}_n(\ldots, \mathrm{EVAL} \bullet \alpha) = \mathrm{CASE}_n(\ldots, \mathrm{EVAL} \bullet \mathrm{L} \bullet \kappa \bullet \mathrm{CASE}_n(\ldots, \mathrm{EVAL} \bullet \alpha))$$

Property 35 .

$$\mathrm{L} \bullet \mathrm{X} \bullet \overline{D}[\![\delta]\!] \bullet \mathrm{X} \bullet \alpha = \overline{D}[\![\delta]\!] \bullet \mathrm{L} \bullet \alpha$$

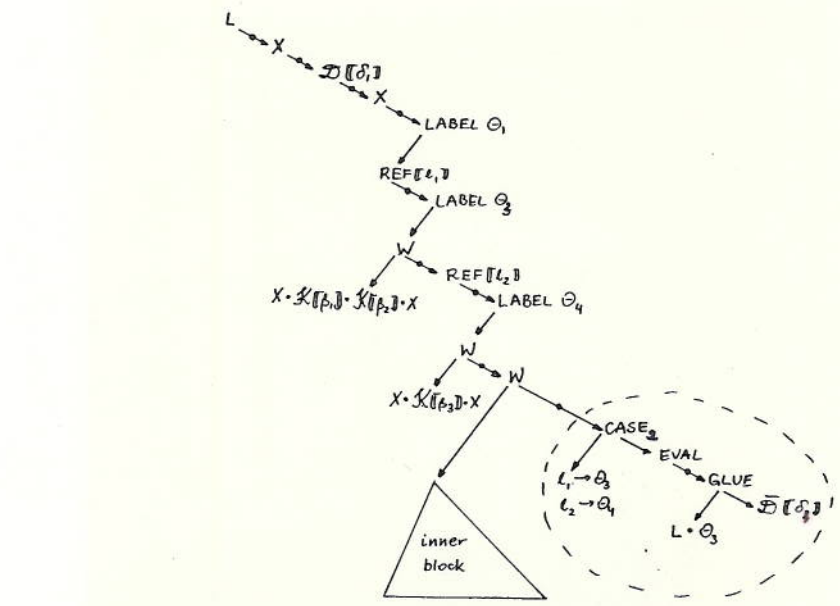
Figures 18–23 show the steps of the entire distribution to the example program. The distribution in the inner block is carried out after the distribution to the outer block.

Notice that we can reduce the number of GLUE combinators by the following property:

Property 36 .

$$\mathrm{GLUE}(\alpha, \mathrm{GLUE}(\beta, \gamma)) = \mathrm{GLUE}(\alpha, \gamma)$$
$$\mathrm{GLUE}(\alpha, \beta) = \alpha \qquad \text{if } \beta \text{ is a meaning function}$$

Figure 23 presents the entire code.



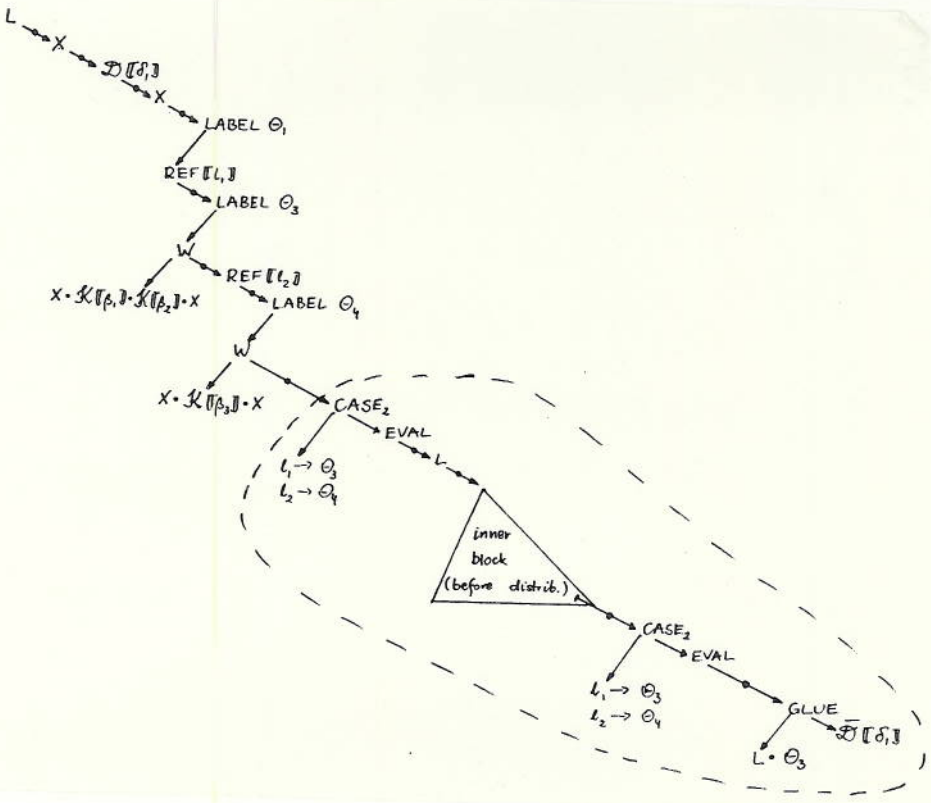


Figure 19. **Distribution in the outer block – 2**

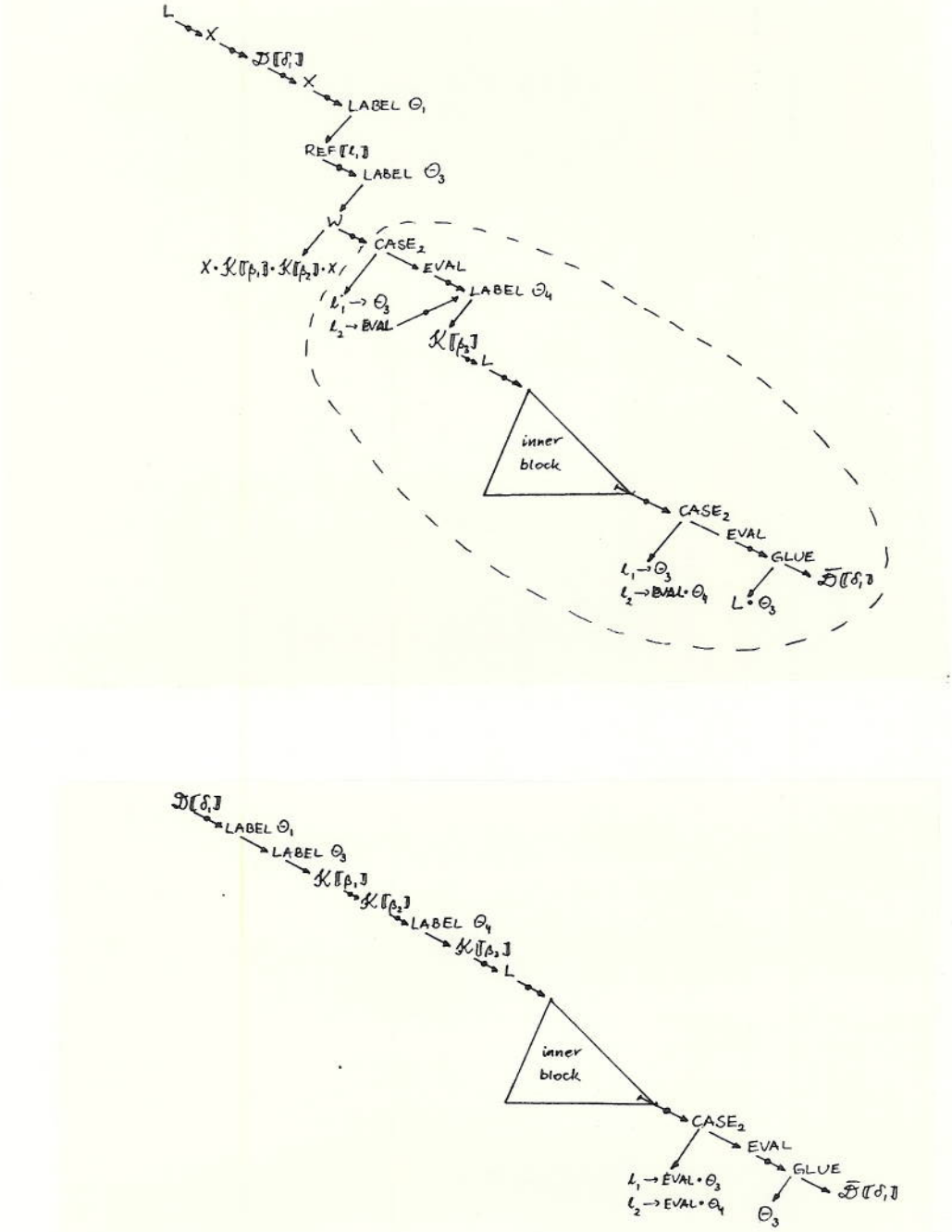
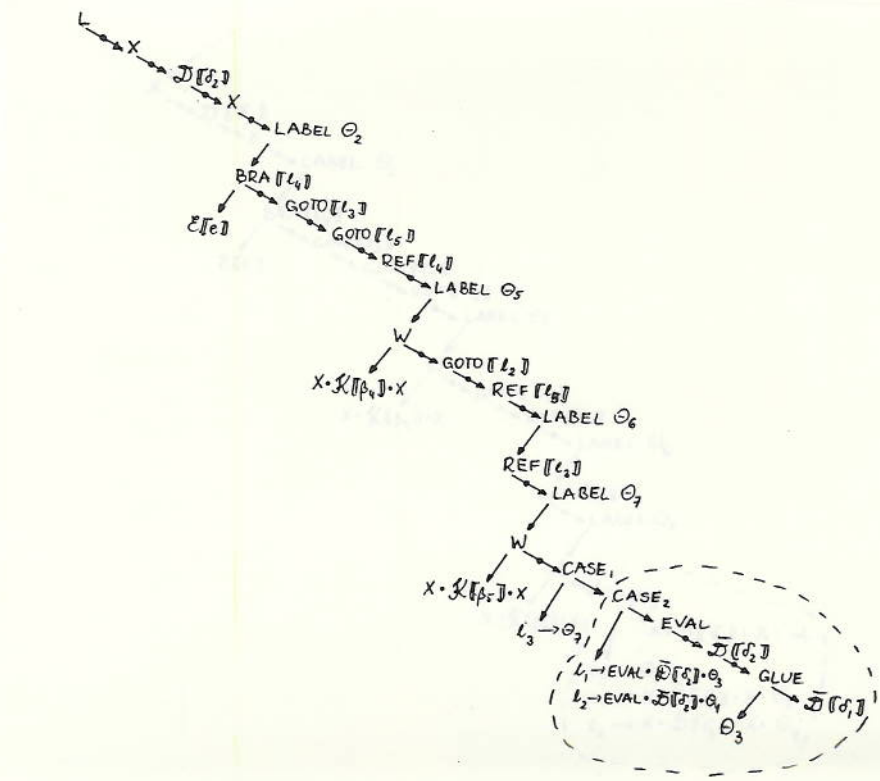


Figure 20. **Distribution in the outer block – 3**



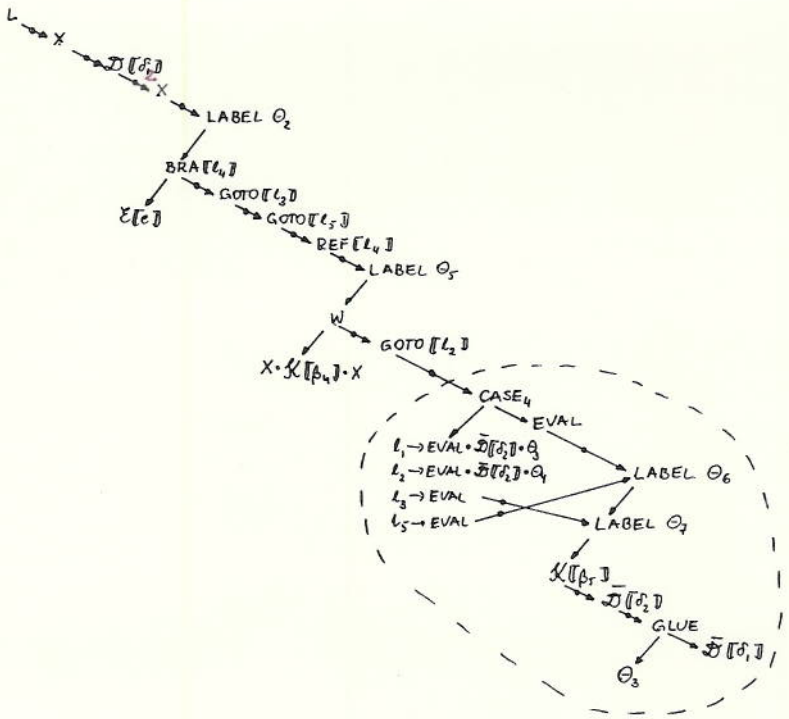


Figure 21. **Distribution in the inner block – 1**

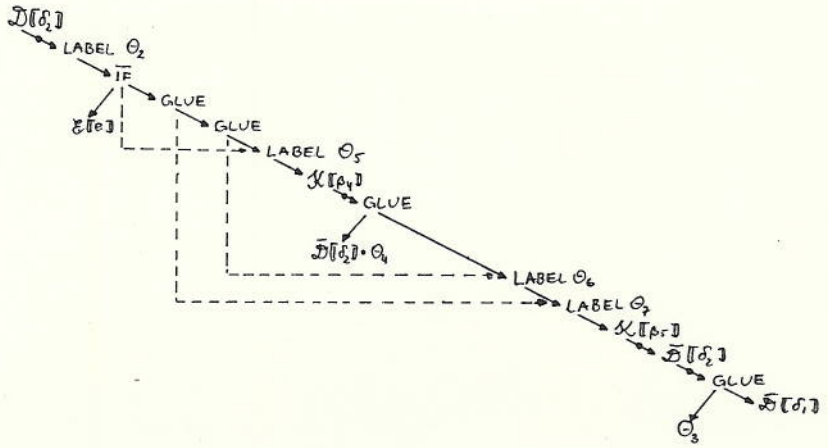


Figure 22. **Distribution in the inner block – 2**

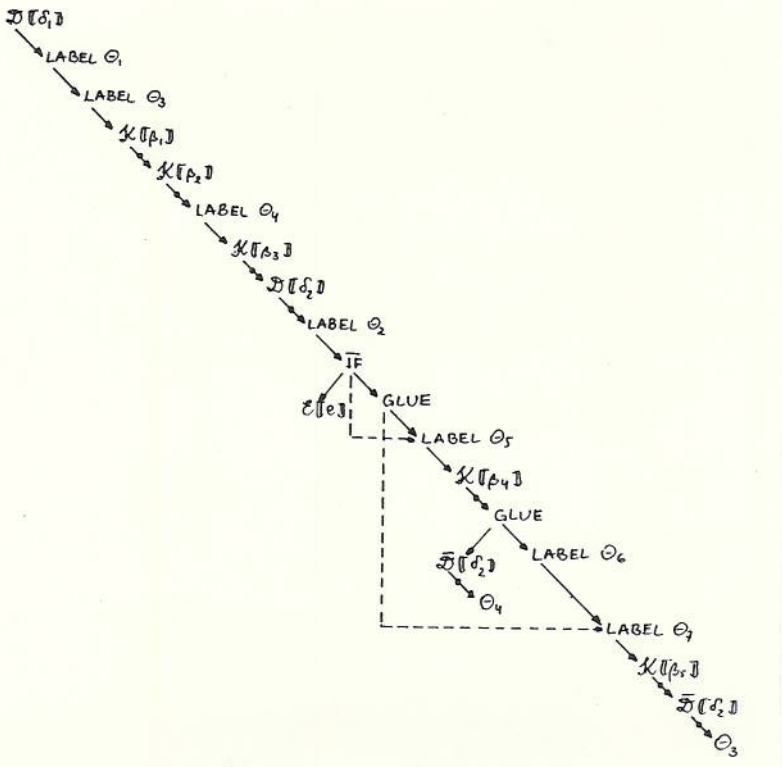


Figure 23. **Target code**

5.4 Elimination of LABEL's

Due to the introduction of GLUE combinators, the target code satisfies the rules for the scope of θ variables. That is a θ variable is used only inside a subgraph LABEL $\theta(\alpha)$. Therefore we can perform a transformation similar to the elimination of LABEL combinators in [10]. We can eliminate θ variables and LABEL combinators by introducing pointers, so by introducing circular expressions ([7]). This transformation is shown in Figure 24. The graph is not acyclic anymore. After this transformation we can eliminate all GLUE's and get a target code of the form from Figure 25.

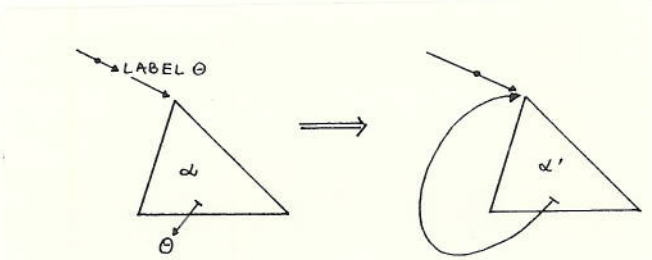


Figure 24. Elimination of LABEL

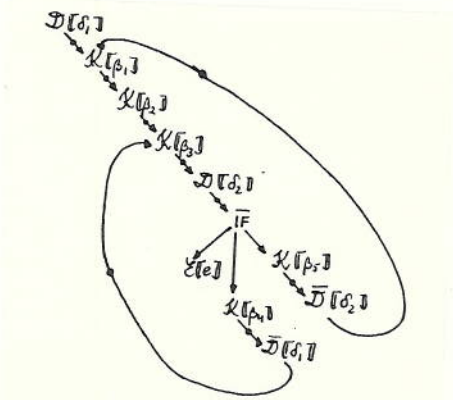


Figure 25. Target code without LABEL's and variables

6 Summary

The method presented in this paper can form a basis for a compiler. The code obtained is of the desired form; it does not contain **goto**'s or labels. Since the transformations preserve the meaning of the program, the compiler is in fact correct. Some implementation issues concern the introduction of internal labels. That is how we

can stop moving these very local labels inside inner blocks, so how we can decrease the size of the distributing CASE combinator. One of the possible solutions can use the EX combinator. We can compose every inner block with this combinator in the code (program graph). Then during linearization we can propagate the information about local labels to all inner blocks accordingly to $\text{EX}(J) \bullet \text{EX}(J') = \text{EX}(J \cup J')$. Another solution can involve an opposite combinator (another version of CASE) like $\text{ONLY}(J) = \lambda\iota.(\iota \in J) \rightarrow \iota, ?$ which means that only labels from J can be produced inside a given block and its inner blocks. In both cases, before moving the distributing CASE combinator into an inner block, the combinator can be "trimmed" accordingly to EX or ONLY.

The intermediate graphs (trees) formed by this method enable us to perform additional transformations. Due to the fact that basic blocks are distinguished from the rest of the code, we can linearize them and compile to a target form in a way presented by Wand ([8,9,10]) for a continuational semantics and by Raoult and Sethi ([4]) for a direct semantics. We can also optimize this code. After linearization of blocks, we can distribute the information stored in compiler's lookup tables and produce a code for a display machine. This issue is presented in [3] for a block-structured language defined in a continuational style. The same graph may be used for linearizing the $\overline{\text{IF}}$ combinator, that is for producing a linear code to evaluate the expression and then select the appropriate branch. This linearization is also discussed in [9] and [3]. Of course, introduction of while-loops should not be difficult at all.

To conclude these comments, we think that simplicity of the method and easy way to extend the compiler to perform additional actions, make the method an excellent frame for building combinator-based compilers and target machines. The transition from denotational semantics to operational one is almost transparent and correct without need for additional proofs.

References

[1] Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981

[2] Blikle, A., Tarlecki, A. "Naive Denotational Semantics", Institute of Computer Science, Polish Academy of Sciences, Warsaw, March, 1983

[3] Lao, M.J. "A Case Study of a Combinator-Based Compiler for a Language with Blocks and Recursive Functions", Indiana University Computer Science Department Technical Report No. 150, October, 1983

[4] Raoult, J.-C., Sethi, R. "Properties of a Notation for Combining Functions", *Journal of ACM*, vol.30, No. 3, July, 1983, pp. 595-611

[5] Scott, D. "Data Types as Lattices", *SIAM Journal on Computing*, vol. 5, 1976, pp.522-585

[6] Scott, D. "Domain Equations", *Proc. 6th Symp. on Mathematical Foundations of Computer Sciences*, IBM Japan, 1981

[7] Sethi, R. "Circular Expressions: Elimination of Static Environments", *Automata, Languages and Programming. Proceedings 1981 (ICALP '81)*, Lecture Notes in Computer Science, vol. 115, Springer-Verlag, Berlin, 1981, pp. 378-392

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

[9] Wand, M. "Deriving Target Code as a Representation of Continuation Semantics" *ACM Trans. on Prog. Lang. and Systems*, vol. 4, No. 3, July, 1982, pp. 496-517

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