Standard, Storeless Semantics for ALGOL-Style Block Structure and Call-by-Name

By

S. Kamal Abdali
Computer Research Lab
Tektronix, Inc.
Beaverton, Oregon 97077

and

David S. Wise
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 186

Standard, Storeless Semantics for
ALGOL-Style Block Structure and Call-by-Name

by

S. Kamal Abdali, Tektronix, Inc.
and David S. Wise, Indiana University

December, 1985

This material is based on work supported (in part) by the National Science Foundation under grant number DCR 84-05241.

Standard, Storeless Semantics for ALGOL-Style Block Structure and Call-by-Name

S. Kamal Abdali
Computer Research Lab
Tektronix, Inc.
Beaverton, Oregon 97077

David S. Wise *
Computer Science Department
Indiana University
Bloomington, Indiana 47405

0. ABSTRACT

This paper presents a formulation for the standard semantics of block structure and `ALGOL 60` style call-by-name. The main features of this formulation are the use of continuations and streams. Continuations are used in such a way that the semantics can be defined without requiring the idea of an explicit store. Thus the concepts of address or L- and R- values are not used, and simple continuations suffice for describing assignments, iterative control statements, compounds, blocks, and functions using call by value. (Side effects are still allowed via assignments to variables global to functions.) Call-by-name is handled by introducing the idea of multiple continuations. Input-output is treated by using streams. In conjunction with continuations, these allow the formulation of program "pipes" exactly like compound functions.

---

* Research supported (in part) by the National Science Foundation under Grant Number DCR84-05241.

1. Introduction

The purpose of denotational semantics, according to Milne and Strachey [8], is to provide an accurate standard by which designers and implementors of programming languages can judge their work. That standard must not be any more specific than is necessary, lest the definition of the language enforce too strong a constraint on its implementation; with new hardware and architectures becoming available one can easily foresee that an overly specific definition will preclude growth of software onto those new machines. "One singularly elegant sort of semantics, 'standard semantics,' is of special significance," because "it reduces to a minimum the amount of substantial information" that is manipulated; "whenever possible precedence should be given to standard semantics rather than store semantics." [8, pp.11–12] The semantics of a programming language should "be specified in the first instance by using standard semantics."

Following the spirit of that dictum, we here demonstrate how to avoid the *store* entirely in providing semantics for the archetype of languages that seem to require it. **ALGOL 60** was designed before development of the tools now familiar to denotational semanticists, but it was designed so carefully [9] that the need for such tools became obvious. By substantially solving the problem of specifying syntax precisely, its designers hastened the development of formal semantics [7].

It was designed, however, with the traditional store in mind. Thus, we believe, much of the early formal semantics also presumed the necessity of that structure. Here we show again [1], but more clearly, how the store might be avoided entirely without changing the (understood) meaning of **ALGOL**.

We are not the only ones doing this; Brookes attacks the same problem elsewhere in this volume [2]. His approach is beautiful, though abstract; ours is effective. To the extent that the lambda calculus is operational, one might say that ours is closer to an implementation; his is clean and elegant. We feel, however, that a simple implementation in the lambda calculus (and, except for the strange machinations of call-by-name, this is simple!) will be sufficiently abstract to generalize to almost any machine [5]. Indeed, we also believe that our semantics is 'fully abstract.'

Previous efforts to define most of the programming features in languages like **ALGOL 60** [e.g., 5, 8] pivot on store semantics, with the store as one domain that maps the so-called L-values (locations) to R-values (contents). This domain has been deemed necessary because even such fundamental features as program variables and assignments seemingly require the concept of an explicit memory. More complex features such as block storage, parameter passing mechanisms, sharing or aliasing of variables introduced by a same variable occurring in different actual parameter positions, etc., seem even more difficult to handle without introducing the store.

Abstractly, the trouble with a store is that the management of the L-values seems to preclude the semantics from being fully abstract in that it may not render the same meanings for the same expressions in different interpretation contexts. Intuitively, it raises problems with storage management: if memory overflow occurs, are we certain that memory was used densely at the penultimate instant? If the storage manager is not to be explicitly provided in the semantics, how are the designers and implementors of a language to understand its properties?

Such questions may be miscast. Like others, we feel that *failures* like these—due to resource exhaustion of time, heap, or stack—are different from *error* signals that might be necessary in the semantics of a programming language. The former restrictions, imposed by the operating system, may well vary from site to site, or with the hourly loading on a shared machine; their cure is as likely to be rescheduling as it is program termination. The latter, however, likely require uniform handling across all implementations and are, therefore, properly included in a formal semantics.

This paper presents an approach to standard, storeless semantics of `ALGOL 60` [9], the key ideas of which are a continuation-based scheme, already used for translating nearly all of `ALGOL 60` to the pure lambda calculus [1], and a clean treatment of input-output using streams [4]. These ideas are, respectively, ten and twenty years old; although he used them differently, streams were invented by Landin [7] to grapple with this very problem! But these ideas were previously used informally and with only intuitive justification. Both of these are newly cast into the rigorous domain theory necessary for denotational semantics. Instead of using a "store", *statically* accessible parameters are explicitly passed (as if a unit) around the system, to the (images of) statements and expressions.

This paper describes three semantic formulations. Section 2, following, introduces the domain equations and describes some salient features of this presentation. The first formulation, introduced in Section 3, covers assignments, blocks, control statements and I/O. The second formulation, in Section 4, provides for procedures with called-by-value parameters. Finally, in Section 5 the third formulation describes the manner in which multiple continuations are used to implement call-by-name. The resulting semantics is a toy language [5] with all the difficult facilities for `ALGOL 60` call-by-name. Section 6 presents an example and conclusions.

2. Salient Features

We make heavy use of the concept of 'continuation' in order to implement the sequential nature of `ALGOL` implicitly through composition of functions. (There is no other way of enforcing a sequential order in an abstract language, like the lambda calculus, that admits alternative orders of evaluation.) Since continuations are likely to be alien to the reader, we have taken pains to force all to be essentially of two kinds: called either "continuation" or "program".

Figure 1 presents all the domains needed here, although not all are used immediately. Most important are $\mathbf{X}$, which is the codomain of the significant semantic functions, and $\mathbf{K}$, the domain of continuations that effect the sequentiality necessary in `ALGOL`.

The domain of continuations, $\mathbf{K}$, is a bunch of functions, each of which maps a pending result (conceptually, the accumulator), environment (current bindings), and input file (its yet unread suffix) into the (yet ungenerated) suffix of the output file. Each may as well be perceived as a function that maps pending result and environment into a function from input files to output files. That last image is the familiar picture of a program, mapping input to output, so we may say that a continuation is merely a running program with the accumulator and current bindings abstracted away.

We do not distinguish between expressions and commands with regard to the nature of their semantics. Both commands and statements are mapped alike into functions, $\xi \in \mathbf{X}$, which at first thought seemed to map only from continuations to continuations. While one might easily overlook the importance of lexical level, one must notice that every phrase in the language resides at a lexical level which becomes essential information at the instant that the binding of a variable must be altered. Therefore, functions in $\mathbf{X}$ necessarily map from lexical levels, in $\mathbf{N}$, as well, carrying the size of a suitable environment.

Any environment represents *only* those bindings that are accessible in the 'current' scope of the program. `ALGOL`, of course, is lexically scoped, so the number of bindings accessible at any point in the execution of a program is proportional to the lexical nesting level at that point in the program; it is independent of how much work 'has already been done' and of the number of pending (unfinished) function invocations. Unlike the number of L-values, therefore, this census of accessible bindings is fixed for any program, and it is likely to be relatively small (because programmers seem to write code structures wider than they are deep.)

The domain of environments, **R**, is structured here as a linear list; each environment is a pair composed of a reference to the first item on the list and a reference to the remainder of the list. The domain equation is isomorphic to that for linear lists. The linear structure is chosen because it is easy to extend (on block entry) and to attenuate on block exit. Indeed, many machines with hardware stacks have been built for `ALGOL`esque languages whose nested structure suggests this arrangement.

Each value to which an identifier might be bound is in **D**, the domain of denoted values. Denoted values are of three varieties, two of which involve functions and arguments. Since this presentation postpones the treatment of functions and procedures until later, for the moment we may perceive these $\delta \in \mathbf{D}$ as simple ground values. Then an environment is a list of such values read left-to-right as bound in the deepest-to-shallowest block.

The domain of streams, **S**, is defined similarly to that of environments, **R**. Streams are also linear lists of ground values. By structuring them as nested pairs, however, we achieve two conveniences for Input/Output semantics [4]. First, the head item on a file is readily accessible from a probe of the stream's left field. Second, the conventional behavior of advancing to the suffix of the file (obtained by removing that head character), is effected by passing its suffix, available from the right field, to a sequent function invocation. This behavior shows up in this semantics only at the interpretation of the expression **read**.

Similarly, 'output' can be effected by returning a pair as a result, whose left element is the ground value put out, and whose right element is a function invocation that is the 'remainder of the program-run,' which likely yields further output pairs. This shows up here only in the semantics for the command **output**.

Input/Output, then, is built on a model of a single input-stream and a single output-stream. Continuations and streams work together to provide a transparent, UNIX^{TM}-like piping of program composition along a single stream [10]. It is straightforward to extend the model to handle multiple I/O streams using functional combination [4], but not enough would be learned to justify the additional functionals necessary.

3. Semantics for a functionless language

Three features are to be observed in the semantic clauses of Figure 2. First is the use of streams, as discussed above for I/O. Second is the unique uniting of allocation and deallocation in the same clause for **begin-end** blocks. Last is the use of the object $\mathcal{A}m$ to effect assignment to the variable $\mathbf{I}_m$, whose binding appears m positions from the right (nested at Position $\nu - m$ in ρ of size ν).

The function that is the image of a program under this semantics deals only with the static environment (lexical scoping) of a program. This is much less confining than what results from premature concern with storage-allocation. A **begin-end** block is mapped as a unit onto a composition of entry, body, and exit functions, the first and the last being interdependent. Upon block entry the environment, ρ, is extended; upon block exit the additional binding is dropped. Thus, the allocation of a local variable is not allowed to become a problem independent of its deallocation [6]. The depth of the program tree, which is the maximum length of that (flattened) environment sequence, ρ, therefore, bounds the accessible storage needed to run the program.

Continuations are used to carry out the evaluation of an expression from the values of its constituents, as well as for propagating the effects of computation from statement to statement. Consequently there is no need for the usual (address-oriented) fetch and store functions. Fetch is provided in the portion of $\mathcal{E}$ dealing with I_m; from lexical level ν it finds position $\nu - m$ in ρ. Store into that same location is provided by $\xi = \mathcal{A}_m$, which reads similarly, but is written separately for reference below.

4. Introducing functions called-by-value

Now we introduce part of the domain of functions, **F**, the second component in **D**. Functions will initially be bound in declarations at block entry, but invocations occur both in ***application*** expressions and in ***call*** commands. Moreover, we also provide a ***result*** command to allow a block (function body) to render a value (for its invocations.) Each $\phi \in \mathbf{F}$ is a triple whose first component is boolean, and will be used (later) to distinguish functions with parameters called-by-name from those with a call-by-value protocol. For now all functions will have single parameters called-by-value, and so this flag will necessarily be `true`. Likewise, the third component of that triple will not be used yet.

Rather than repeating the entire semantics, we show in Figure 3 only the additions necessary to the semantics of the functionless case discussed just above. Introduction of functions and procedures with single arguments called-by-value affects most of the existing semantic clauses. Most importantly, $\mathcal{V}$ now has a non-trivial declaration.

Procedures with parameters called exclusively by value, with recursion, and/or with side-effects via assignments to non-locals do not introduce complexity beyond the domain equations. The images of user-defined functions are allowed as denoted values within the "block" that define them,, but the values on the stack now contain elements from **X**. Their manipulation of the stack, however, is exactly that of **E** or **D**, as already used in Section 3. They look up and store to explicit positions in the environment.

5. Call-by-name Arguments

We now discuss the semantics of call by name in the sense of ALGOL 60. We give the semantic equations in Figure 4, restricting, for simplicity, to procedures with a single argument. Although the semantics presented here provides only one argument called-by-name, we can provide several. Where we discuss a single binding below, however, all of the several simultaneously bound values must be treated. After describing the single argument case in detail, we shall complete the description of that treatment.

The idea for call-by-name semantics is that three continuations (thunks) are passed for each parameter called-by-name. The first, the "assignment program", is only invoked from $\mathcal{C}$ on an assignment to this parameter; it installs a new value as the binding in a calling environment for an identifier called-by-name The second, the "evaluation program", is used to retrieve/compute values in the calling environment, and is only invoked from $\mathcal{E}$ as it discharges an identifier. In both instances the program provides for reconstruction of the (modified) called environment. The third restores the calling context upon final exit from the invocation, permanently abandoning the called environment; it is used in forming the exit continuation as a call-by-name function is invoked.

Now one can better understand the structure of the domain of functions, **F**. The boolean tags the parameter passing mechanism for the single argument; in the case of alternative argument structures, a domain of signatures for the alternatives would replace it. The program, ξ, is to be used later to restore the environment in which the function was defined (closed) before invoking its body. The continuation, κ, is defined according to the meaning ofthe function body, itself. The separation of ξ from κ is useful because ξ may need to be performed repeatedly—after every use of the argument called-by-name—but the body will only be invoked once for each invocation.

The argument, α is passed into the function as part of the "accumulator," in **E**. This convention allows us later to return a value (resulting from use of the "evaluation program") *and* a new α simultaneously. (When multiple arguments are called-by-name, the bindings of all in **A**must be so replaced when any one is used.)

From the preceding description one can anticipate the complications introduced by Call-by-Name: Invocation of such a function in $\mathcal{E}$ is complicated by the need to compute three continuations, and by redirecting the exit continuation through the third. The difference between the lexical level at which the function was declared (closed) and the lexical level at which the invocation takes place determines the amount of environment saving and restoration that must occur upon function entry/exit and upon each reference to its parameter. Closing such a function at its declaration in $\mathcal{V}$, therefore, includes the rudimentary structure of the argument triple based on the lexical level.

While the points on function closure and application are complicated, the semantics for interpreting a call-by-name parameter end up being quite direct. $\mathcal{C}$ or $\mathcal{E}$ need only invoke the appropriate piece of the bound triple. Thus, the complicated situation where one identifier called-by-need is bound to another sorts itself out quite nicely.

Upon a context switch for call-by-name, the variable sequence (stack) shrinks and then re-expands according to the declarations within the intervening blocks jumped by the closing of the function. This restoration is necessary to provide for side-effects to a non-local environment, particularly in the case where call-by-name identifiers are cascaded: bound to-one another in an arrangement wherein the use of one paramenter (called-by-name) causes multiple side effects in several others at different lexical levels.

It is really remarkable how much formalism is necessary to provide for this one "intuitive feature." As we set aside L-values, the introduction of call-by-name increases the bulk of semantic equations by 50% , approximating the burden it causes the implementor!

ALGOL 60's call-by-name is a particularly complex programming feature and was abandoned in its descendents. It is interesting to note, however, that the concept of closure in Scheme [11] and the newer Lisps resembles classic call-by-name. Moreover, the style of using continuations there resembles the method used in this paper to implement "store". But these vistiges do not include the difficulties of assignement to an identifier passed-by-name.

What about space limitations when arguments are called-by-name? The size of the lexical environment may always be bounded *a priori* according to the syntactic depth of the program. In the absence of call-by-name, we note that environments grow and shrink predictably according to static scoping rules. Call-by-value parameters require some environment saving and restoration via exit continuations, but only for simple bindings and only for the full duration of a function invocation. Thus, even storage within continuations can be anticipated before run-time, except for the effects of recursion.

The picture turns out little different for call-by-name (as long as upward functions are prohibited.) However, continuations and the storage necessary for state associated with them may be more expensive. A single argument called-by-name requires the meaning of our triple of continuations, but the three likely share the state information for restoring the calling environment. The allowed context switches may occur repeatedly, but only one context (at a time) need be remembered. In fact, though structured differently, the restoration information is exactly that kept for the exit continuation for call-by-name, though it is held in a manner to be used in any of the three ways. As before, recursion muddies the picture, but static binding still is a very good first bound on the space require for stacks and continuations.

This situation should be constrasted with that for a store, with L-values. There is no treatment here for bindings not either in the lexical environment or (implicitly) in a continuation. While this is a weaker space measure when recursive functions are considered, it is far simpler than those requiring explicit space release upon block exit and concerns about garbage collection and exhaustion of space [6]. While we do not give explicit semantics for space exhaustion, neither do standard machine-independent languages.

Finally, let us reconsider the effect of multiple arguments called-by-name. We have modified this semantics to run examples with n such arguments. All that is necessary is to augment each continuation in the triple to purge and later to restore all n bindings upon the use of any one. At first this seems like a great complication, but it isn't. The only change occurs in the meaning for $\mathcal{V}$ where each triple is established, and in the function application line of $\mathcal{E}$ where initial triples are provided. In fact, the third part of each triple (as described here) need only be provided as if it were the only argument, because function exit, provided by it, is parameter-independent.

6. Examples and Conclusions

The semantics described above has been verified by translating a number of programs and evaluating their $\mathcal{P}$-meanings applied to relevant inputs. A program has been written in Scheme to actually carry out this process automatically. Two sample programs used in the test are shown in Figure 5. Program P_1 illustrates both call by value and call by name used in one-argument procedures. The call by name part, though simple, involves the nontrivial operations of evaluating and altering the argument *in the environment of the procedure call.* This program consumes a single item from its input stream, and appends three items to its output stream. Thus, as a particular execution of this program, if $\mathcal{P}[\![\mathrm{P}_1]\!]$is applied to the input stream represented by the tuple $\langle 3,1,4,1,5\rangle$, the result is the tuple $\langle 3,6,7,eof,\langle 1,4,1,5\rangle\rangle$. —

Program P_2 provides a more interesting illustration of call by name. It highlights the use of global variables as called by name arguments whose evaluation and alteration requires crossing several block levels. It also shows nested procedure calls and rather complex side-effecting. We have used an straightforward syntactic convention for multi-argument procedures and functions in which *all* arguments are called alike, either by value or by name. The semantic equations for obtaining the $\mathcal{P}$-meaning of such programs are not given in Figure 4, but we have outlined the method of their translation in the previous section. This program does not read anything from its input, but writes two items on its output. As a particular execution of this program, if the expression $(\mathcal{P}[\![\mathrm{P}_1]\!]\langle 3,1,4\rangle)$ representing the execution of the program with inputs cosisiting of 3, 1, and 4 is evaluated, the result is found to be the tuple $\langle 24,18,eof,\langle 3,1,4\rangle\rangle$.

We have not provided for recursive definitions here. While we know different ways to include it, none offer domain equations quite as elegant as appear here. The most tractable solution is to redefine the codomain of $\mathcal{V}$ to be reflexive—something other than X. That would suffice for recursive function definitions, but it would, for example, preclude initializing variables from input (using `read`). A complete solution confounds the domain definitions and, we feel, would detract from the Call-by-Name semantics which we want to highlight.

If an implementor would like to perceive a ***stack*** or ***display*** in this behavior, then she is likely to invent an efficient-on-current-hardware implementation of `ALGOL 60`, as correct as her extension of this semantics. But if she sees some other, efficient-on-future-hardware structure and faithfully uses it to extend this semantics, , then she will effect an implementation that is both correct and efficient on hardware yet unknown.

This last point is very important, because one of the virtues of formally describing languages like ALGOL 60 is the ability to express meaning of existing programs on architectures unforeseen by the language designers. If one wanted to implement, say, ALGOL 60 on a machine that had no easy way of providing L-values, would it be necessary to provide them? We have clearly answered that question in the negative, showing how an iterative, lexically scoped, but richly side-effecting language might be implemented on some pure Lambda-calculus machine. The implication is that applicative architectures will be able to run ALGOL 60 programs, perhaps with little loss of efficiency (because they will only run into complicated code for the lesser-used pathologies of call-by-name.)

ALGOL 60 was not designed with the semantic rigor established by Denotational Semantics, but its definition set a very high standard in its day. Indeed, one can argue that the difficulties that arose in defining ALGOL 60 led directly to the development of the denotational approach. That input/output and call-by-value is expressed so cleanly here is a testament to the relative simplicity of these features. That call-by-name (in the context of side effects) is so burdensome should not be surprising—its implications were not all that well understood even to the ALGOL 60 committee.

REFERENCES

[1] S.K. Abdali, A lambda-calculus model of programming languages. *J. Computer Languages* **1** (1976), 287-301 + 303-320.

[2] S.D. Brookes. A fully abstract semantics and a proof system for an ALGOL-like language with sharing. (Included in this volume.)

[3] W.D. Clinger. *Foundations of Actor Semantics*, Ph.D. dissertation, Artificial Intelligence Technical Report 633, Massachusetts Institute of Technology (1981), 86.

[4] D.P. Friedman and D.S. Wise. Applicative Programming for file systems. *Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices* **12**, 3 (March 1977), 41-55.

[5] M.J. Gordon. *The Denotational Description of Programming Languages, An Introduction*, Springer, New York (1979).

[6] J.V. Halpern, A.R. Meyer, and B.A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free. *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages*, ISBN 0-89791-125-3 (1983), 245-257.

[7] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda notation, Part I. *Comm. ACM* **8**, 2 (February 1965), 89-101.

[8] R. Milne and C. Strachey. *A theory of programming language semantics*, London, Chapman and Hall (1976).

[9] P. Naur (ed.) et al. Revised report on the algorithmic language ALGOL 60. *Comm. ACM* **6**, 1 (January 1963), 1-17.

[10] J.-C. Raoult and R. Sethi. Properties of a notation for combining functions. *J. ACM* **30**, 3 (July 1983), 595-611.

[11] G.L. Steele and G.J. Sussman. Scheme: an interpreter for extended lambda-calculus. Artificial Intelligence Lab Memo 349, Massachusetts Institute of Technology (December 1975).

Figure 1. Domain specifications

Syntactic Domains

I $\in$ Ide	Identifiers
B $\in$ Bas	Basic constants
O $\in$ Opr	Operators
P $\in$ Pro	Programs
E $\in$ Exp	Expressions
C $\in$ Com	Commands
V $\in$ Val	Values declarable

Syntactic Clauses

I : Ide	::=	{ identifiers}	Identifiers
B : Bas	::=	**true** | **false** | 0 | 1 | −1 | 2 | −2 | ...	Basic Constants
O : Ope	::=	+ | − | × | ÷ | < | ≤ | = | > | ≥ | ...	Operators
P : Pro	::=	**program C**	Programs
E : Exp	::=	**B** | **true** | **false** | **read** | **I** |	Expressions
		if E_0 **then** E_1 **else** E_2 | $E_1 O E_2$ | $E_0 E_1$	
V : Val	::=	**E** |	Values declarable
		function I value C | **procedure I value C**	
		function I name C | **procedure I name C**	
C : Com	::=	**I:= E** | **output E** | $(C_1; C_2)$ |	Commands
		if E_0 **then** C_1 **else** C_2 | **while E do C** |	
		begin var I:= E; C end | **call** $E_0 E_1$ | **result E**	

Semantic Domains

$\nu \in \mathbf{N} = \{0, 1, 2, 3, ...\}$	integers
$\mathbf{B} = \{TRUE, FALSE\}$	booleans
$\mathbf{G} = \mathbf{N} + \{eof\}$	ground values
$\epsilon \in \mathbf{E} = (\mathbf{B} + \mathbf{G} + \mathbf{F}) \times \mathbf{E}$	expressed values (accumulator)
$\mathbf{D} = \mathbf{E} + \mathbf{A}$	denoted values
$\sigma \in \mathbf{S} = \mathbf{G} \times \mathbf{S}$	streams
$\kappa \in \mathbf{K} = \mathbf{E} \to \mathbf{R} \to \mathbf{S} \to \mathbf{S}$	continuations
$\pi \in \mathbf{P} = \mathbf{K} \to \mathbf{K} = \mathbf{K} \to \mathbf{E} \to \mathbf{R} \to \mathbf{S} \to \mathbf{S}$	pure code
$\xi \in \mathbf{X} = \mathbf{N} \to \mathbf{P} = \mathbf{N} \to \mathbf{K} \to \mathbf{E} \to \mathbf{R} \to \mathbf{S} \to \mathbf{S}$	code
$\rho \in \mathbf{R} = \mathbf{D} \times \mathbf{R}$	environments
$\phi \in \mathbf{F} = \mathbf{B} \times \mathbf{X} \times \mathbf{K}$	functions
$\alpha \in \mathbf{A} = \mathbf{P} \times \mathbf{P} \times \mathbf{K}$	arguments called by name

Semantic Functions

$\mathcal{P}$: Pro$\to\mathbf{S}\to\mathbf{S}$	meaning of program
$\mathcal{E}$: Exp$\to\mathbf{X}$	meaning of expression
$\mathcal{C}$: Com$\to\mathbf{X}$	meaning of command
$\mathcal{V}$: Val$\to\mathbf{X}$	meaning of decalarable values
$\mathcal{B}$: Bas$\to\mathbf{E}$	meaning of base values
$\mathcal{O}$: $\mathbf{G}\times\mathbf{G}\to\mathbf{G}$	meaning of arithmetic opearators
$\mathcal{A}$: $\mathbf{N}\to\mathbf{X}$	alteration to lexical position ν

Figure 2. Semantics for a functionless language

Semantic Clauses

$$\mathcal{P}\llbracket \mathtt{program}\ \ \mathtt{C}\rrbracket = \mathcal{C}\llbracket \mathtt{C}\rrbracket 0\ (\lambda\epsilon\rho\sigma.\ \langle eof,\sigma\rangle)\ \bot_{\mathbf{E}}\bot_{\mathbf{R}}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{B}\rrbracket &= \lambda\nu\kappa\epsilon.\kappa(\mathcal{B}\llbracket \mathtt{B}\rrbracket)\\
\mathcal{E}\llbracket \mathtt{true}\ \rrbracket &= \lambda\nu\kappa\epsilon.\kappa(\mathit{TRUE}\ \mathbf{in}\ \ \mathbf{E})\\
\mathcal{E}\llbracket \mathtt{false}\ \rrbracket &= \lambda\nu\kappa\epsilon.\kappa(\mathit{FALSE}\ \mathbf{in}\ \ \mathbf{E})\\
\mathcal{E}\llbracket \mathtt{read}\ \rrbracket &= \lambda\nu\kappa\epsilon\rho\sigma.\ \kappa(\sigma{\downarrow}1\ \ \mathbf{in}\ \ \mathbf{E})\ \rho(\sigma{\downarrow}2)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{I}_m\rrbracket = {}& \mathbf{fix}\ \ \lambda\xi.\ \ (\lambda\nu\kappa\epsilon_1\rho_1.\\
& (\nu{=}m \rightarrow \kappa(\rho_1{\downarrow}1 \mid \mathbf{E})\rho_1,\\
& (\nu{>}m \rightarrow \xi(\nu{-}1)\ (\lambda\epsilon_2\rho_2.\ \kappa\epsilon_2\langle\rho_1 \downarrow 1, \rho_2\rangle)\epsilon_1(\rho_1{\downarrow}2),\\
& \qquad\qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\ )))
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{if}\ \ \mathtt{E}_0\ \mathtt{then}\ \ \mathtt{E}_1\ \mathtt{else}\ \ \mathtt{E}_2\rrbracket = {}& \lambda\nu\kappa.\ \mathcal{E}\llbracket \mathtt{E}_0\rrbracket\nu(\lambda\epsilon.\\
& (\epsilon|\mathbf{B} = \mathit{TRUE} \rightarrow \mathcal{E}\llbracket \mathtt{E}_1\rrbracket,\\
& (\epsilon|\mathbf{B} = \mathit{FALSE} \rightarrow \mathcal{E}\llbracket \mathtt{E}_2\rrbracket,\\
& \qquad\qquad\qquad \bot_{\mathbf{X}}))\nu\kappa\epsilon)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{E}_1 \mathtt{O}\ \mathtt{E}_2\rrbracket = {}& \lambda\nu\kappa.\ \mathcal{E}\llbracket \mathtt{E}_1\rrbracket\nu(\lambda\epsilon_1.\\
& \mathcal{E}\llbracket \mathtt{E}_2\rrbracket\nu(\lambda\epsilon_2.\ \kappa((\mathcal{O}\llbracket \mathtt{O}\rrbracket(\epsilon_1|\mathbf{G})\ (\epsilon_2|\mathbf{G}))\ \ \mathbf{in}\ \ \mathbf{E}))\ \epsilon_1)
\end{aligned}$$

$$\mathcal{V} = \mathcal{E}$$

$$\mathcal{C}\llbracket \mathtt{I}_m\mathtt{:=}\ \mathtt{E}\rrbracket = \lambda\nu\kappa.\ \mathcal{E}\llbracket \mathtt{E}\rrbracket\nu(\mathcal{A}m\ \nu\kappa)$$

$$\mathcal{C}\llbracket \mathtt{output}\ \mathtt{E}\rrbracket = \lambda\nu\kappa.\ \mathcal{E}\llbracket \mathtt{E}\rrbracket\nu(\lambda\epsilon\rho\sigma.\ \langle\epsilon|\mathbf{G},\ \kappa\epsilon\rho\sigma\rangle)$$

$$\begin{aligned}
\mathcal{C}\llbracket \mathtt{if}\ \ \mathtt{E}_0\ \mathtt{then}\ \ \mathtt{C}_1\ \mathtt{else}\ \ \mathtt{C}_2\rrbracket = {}& \lambda\nu\kappa.\ \mathcal{E}\llbracket \mathtt{E}_0\rrbracket\nu(\lambda\epsilon.\\
& (\epsilon|\mathbf{B}{=}\ \mathit{TRUE} \rightarrow \mathcal{C}\llbracket \mathtt{C}_1\rrbracket,\\
& (\epsilon|\mathbf{B}{=}\ \mathit{FALSE} \rightarrow \mathcal{C}\llbracket \mathtt{C}_2\rrbracket,\\
& \qquad\qquad\qquad \bot_{\mathbf{X}}))\nu\kappa\epsilon)\\
\mathcal{C}\llbracket \mathtt{while}\ \ \mathtt{E}\ \mathtt{do}\ \ \mathtt{C}\rrbracket = {}& \lambda\nu.\ (\ \mathbf{fix}\ \ \lambda\pi.\ (\lambda\kappa.\ \mathcal{E}\llbracket \mathtt{E}\rrbracket\nu(\lambda\epsilon.\\
& (\epsilon|\mathbf{B}{=}\ \mathit{TRUE} \rightarrow \mathcal{C}\llbracket \mathtt{C}\rrbracket\nu(\pi\kappa),\\
& (\epsilon|\mathbf{B}{=}\ \mathit{FALSE} \rightarrow \kappa,\\
& \qquad\qquad\qquad \bot_{\mathbf{K}}))\ \epsilon)))\\
\mathcal{C}\llbracket (\mathtt{C}_1;\ \mathtt{C}_2)\ \rrbracket = {}& \lambda\nu\kappa.\ \mathcal{C}\llbracket \mathtt{C}_1\rrbracket\nu(\mathcal{C}\llbracket \mathtt{C}_2\rrbracket\nu\kappa)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}\llbracket \mathtt{begin}\ \ \mathtt{var}\ \ \mathtt{I}\mathtt{:=}\ \mathtt{E};\ \mathtt{C}\ \mathtt{end}\ \rrbracket = {}& \lambda\nu\kappa.\ \mathcal{V}\llbracket \mathtt{E}\rrbracket\nu(\lambda\epsilon_1\rho_1.\\
& \mathcal{C}\llbracket \mathtt{C}\rrbracket(\nu{+}1)\ (\lambda\epsilon_2\rho_2.\kappa\epsilon_2(\rho_2 \downarrow 2))\ \epsilon_1\langle\epsilon_1\ \ \mathbf{in}\ \ \mathbf{D}, \rho_1\rangle)
\end{aligned}$$

$$\begin{aligned}
\mathcal{A} = {}& \lambda m\ .\ (\ \mathbf{fix}\ \ \lambda\xi.\ \ (\lambda\nu\kappa\epsilon_1\rho_1\ .\\
& (\nu{=}m \rightarrow \kappa\epsilon_1\langle\epsilon_1\ \mathbf{in}\ \mathbf{D}, \rho_1 \downarrow 2\rangle\ ,\\
& (\nu{>}m \rightarrow \xi(\nu{-}\ 1)\ (\lambda\epsilon_2\rho_2\ .\ \kappa\epsilon_2\ \langle\rho_1 \downarrow 1, \rho_2\rangle)\epsilon_1\ (\rho_1{\downarrow}2),\\
& \qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\ )))
\end{aligned}$$

Figure 3. Semantics for a language with call by value.

Syntactic Clauses.

$\mathtt{I} : \mathrm{Ide}$	$::=$	$\{\ \text{identifiers}\}$	Identifiers
$\mathtt{B} : \mathrm{Bas}$	$::=$	$\mathtt{true} \mid \mathtt{false} \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \ldots$	Basic Constants
$\mathtt{O} : \mathrm{Ope}$	$::=$	$+ \mid - \mid \times \mid \div \mid < \mid \leq \mid = \mid > \mid \geq \mid \ldots$	Operators
$\mathtt{P} : \mathrm{Pro}$	$::=$	$\mathtt{program}\ \ \mathtt{C}$	Programs
$\mathtt{E} : \mathrm{Exp}$	$::=$	$\mathtt{B} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{read} \mid \mathtt{I} \mid$ $\mathtt{if}\ \ \mathtt{E}_0\ \ \mathtt{then}\ \ \mathtt{E}_1\ \ \mathtt{else}\ \ \mathtt{E}_2 \mid \mathtt{E}_1\mathtt{O}\,\mathtt{E}_2 \mid \mathtt{E}_0\mathtt{E}_1$	Expressions
$\mathtt{V} : \mathrm{Val}$	$::=$	$\mathtt{E} \mid$ $\mathtt{function}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C} \mid \mathtt{procedure}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C}$	Values declarable
$\mathtt{C} : \mathrm{Com}$	$::=$	$\mathtt{I} := \mathtt{E} \mid \mathtt{output}\ \mathtt{E} \mid \mathtt{if}\ \ \mathtt{E}_0\ \ \mathtt{then}\ \ \mathtt{C}_1\ \ \mathtt{else}\ \ \mathtt{C}_2 \mid$ $\mathtt{while}\ \ \mathtt{E}\ \mathtt{do}\ \ \mathtt{C} \mid \mathtt{C}_1; \mathtt{C}_2 \mid \mathtt{begin}\ \ \mathtt{var}\ \ \mathtt{I} := \mathtt{E}; \mathtt{C}\ \mathtt{end} \mid$ $\mathtt{call}\ \ \mathtt{E}_0\mathtt{E}_1 \mid \mathtt{result}\ \ \mathtt{E}$	Commands

Semantic Clauses. (Additions to Fig. 2)

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{E}_0\mathtt{E}_1 \rrbracket = {} & \lambda\nu_1\kappa_1.\ \mathcal{E}\llbracket \mathtt{E}_0 \rrbracket \nu_1(\lambda\epsilon_0. \\
& (\phi{\downarrow}1 = \mathit{TRUE} \rightarrow \mathcal{E}\llbracket \mathtt{E}_1 \rrbracket \nu_1((\phi{\downarrow}2)\ \nu_1\kappa_1)\epsilon_0, \\
& \qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\,))
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}\llbracket \mathtt{function}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C} \rrbracket = {} & \lambda\nu_1\kappa_1\epsilon_1.\ \kappa_1\langle\ \mathit{TRUE},\ \xi,\ \bot_{\mathbf{K}}\rangle \\
& \mathbf{where} \\
& \xi\ =\ \mathbf{fix}\ \ \lambda\xi.\ (\lambda\nu_2\kappa_2\epsilon_2\rho_2. \\
& \quad (\nu_2{=}\nu_1 \rightarrow \mathcal{C}\llbracket \mathtt{C} \rrbracket(\nu_1{+}1)(\lambda\epsilon_3\rho_3\ .\ \kappa_2\epsilon_3(\rho_3 \downarrow 2)\epsilon_2\langle\epsilon_2, \rho_2\rangle, \\
& \quad (\nu_2 > \nu_1 \rightarrow \xi(\nu_2{-}1)\ (\lambda\epsilon_3\rho_3\ .\ \kappa_2\epsilon_3\langle\rho_2 \downarrow 1, \rho_3\rangle)\epsilon_2(\rho_2{\downarrow}2\ )\ , \\
& \qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\,)))
\end{aligned}$$

$$\mathcal{V}\llbracket \mathtt{procedure}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C} \rrbracket = \mathcal{V}\llbracket \mathtt{function}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C} \rrbracket$$

$$\mathcal{V}\llbracket \mathtt{E} \rrbracket = \mathcal{E}\llbracket \mathtt{E} \rrbracket$$

$$\begin{aligned}
\mathcal{C}\llbracket \mathtt{call}\ \ \mathtt{E}_0\mathtt{E}_1 \rrbracket &= \mathcal{E}\llbracket \mathtt{E}_0\mathtt{E}_1 \rrbracket \\
\mathcal{C}\llbracket \mathtt{result}\ \ \mathtt{E} \rrbracket &= \mathcal{E}\llbracket \mathtt{E} \rrbracket
\end{aligned}$$

$$\begin{aligned}
\mathcal{A} = {} & \lambda m.\ (\ \mathbf{fix}\ \ \lambda\xi.\ \ (\overline{\lambda\nu}\kappa\epsilon_1\rho_1\ . \\
& (\nu{=}m \rightarrow\ \ (\\
& \qquad ((\rho_1{\downarrow}1) \in \mathbf{E} \rightarrow \kappa\epsilon_1\langle\epsilon_1\ \ \mathrm{in}\ \ \mathbf{D},\ \rho_1 \downarrow 2\rangle\ , \\
& \qquad\qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\,)), \\
& (\nu > m \rightarrow \xi(\nu - 1)\ (\lambda\epsilon_2\rho_2\ .\ \kappa\epsilon_2\ \langle\rho_1 \downarrow 1, \rho_2\rangle)\epsilon_1\ (\rho_1{\downarrow}2), \\
& \qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\,))
\end{aligned}$$

Figure 4. Semantics for a language with call by name

Semantic Clauses

$$\mathcal{P}\llbracket \mathtt{program}\ \ \mathtt{C}\rrbracket = \mathcal{C}\llbracket \mathtt{C}\rrbracket 0\ (\lambda\epsilon\rho\sigma.\ \langle \mathit{eof},\sigma\rangle)\ \bot_{\mathbf{E}}\bot_{\mathbf{R}}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{B}\rrbracket &= \lambda\nu\kappa\epsilon.\kappa(\mathcal{B}\llbracket \mathtt{B}\rrbracket)\\
\mathcal{E}\llbracket \mathtt{true}\ \rrbracket &= \lambda\nu\kappa\epsilon.\kappa(\mathit{TRUE}\ \ \mathbf{in}\ \ \mathbf{E})\\
\mathcal{E}\llbracket \mathtt{false}\ \rrbracket &= \lambda\nu\kappa\epsilon.\kappa(\mathit{FALSE}\ \ \mathbf{in}\ \ \mathbf{E})\\
\mathcal{E}\llbracket \mathtt{read}\ \rrbracket &= \lambda\nu\kappa\epsilon\rho\sigma.\ \kappa(\sigma{\downarrow}1\ \ \mathbf{in}\ \ \mathbf{E})\ \rho(\sigma{\downarrow}2)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{I}_m\rrbracket = {} & \mathbf{fix}\ \ \lambda\xi.\ \ (\lambda\nu\kappa\epsilon_1\rho_1.\\
& (\nu{=}m \rightarrow\ \ ((\rho_1{\downarrow}1)\in\mathbf{A}\rightarrow(\rho_1{\downarrow}1|\mathbf{A}{\downarrow}1)\ \nu\kappa\epsilon_1(\rho_1{\downarrow}2),\\
& \qquad\qquad\qquad\qquad \kappa(\rho_1{\downarrow}1\mid\mathbf{E})\ \rho_1),\\
& (\nu{>}m \rightarrow\xi(\nu-1)\ (\lambda\epsilon_2\rho_2.\ \kappa\epsilon_2\langle\rho_1{\downarrow}1,\ \rho_2\rangle)\epsilon_1(\rho_1{\downarrow}2),\\
& \qquad\quad \bot_{\mathbf{S}\rightarrow\mathbf{S}})))
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{if}\ \ \mathtt{E}_0\ \mathtt{then}\ \ \mathtt{E}_1\ \mathtt{else}\ \ \mathtt{E}_2\rrbracket = {} & \lambda\nu\kappa.\ \mathcal{E}\llbracket \mathtt{E}_0\rrbracket\nu(\lambda\epsilon.\\
& (\epsilon|\mathbf{B} = \mathit{TRUE}\rightarrow\mathcal{E}\llbracket \mathtt{E}_1\rrbracket,\\
& (\epsilon|\mathbf{B} = \mathit{FALSE}\rightarrow\mathcal{E}\llbracket \mathtt{E}_2\rrbracket,\\
& \qquad\qquad \bot_{\mathbf{X}}))\nu\kappa\epsilon)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{E}_1\mathtt{O}\ \mathtt{E}_2\rrbracket = {} & \lambda\nu\kappa.\ \mathcal{E}\llbracket \mathtt{E}_1\rrbracket\nu(\lambda\epsilon_1.\\
& \mathcal{E}\llbracket \mathtt{E}_2\rrbracket\nu(\lambda\epsilon_2.\ \kappa((\mathcal{O}\llbracket \mathtt{O}\rrbracket(\epsilon_1|\mathrm{G})\ (\epsilon_2|\mathrm{G}))\ \ \mathbf{in}\ \ \mathbf{E}))\ \epsilon_1)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\llbracket \mathtt{E}_0\mathtt{E}_1\rrbracket = {} & \lambda\nu_1\kappa_1.\ \mathcal{E}\llbracket \mathtt{E}_0\rrbracket\nu_1(\lambda\epsilon_0.\\
& (\phi{\downarrow}1 = \mathit{TRUE}\rightarrow\mathcal{E}\llbracket \mathtt{E}_1\rrbracket\nu_1((\phi{\downarrow}2)\ \nu_1\kappa_1)\epsilon_0,\\
& (\phi{\downarrow}1 = \mathit{FALSE}\rightarrow(\phi{\downarrow}2\ )\ \nu_1(\phi{\downarrow}3)\ (\langle\ \epsilon_0\ \mathbf{in}\ \ \mathbf{D},\alpha\rangle\ \ \mathbf{in}\ \ \mathbf{E})\ ,\\
& \qquad\qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\ )))
\end{aligned}$$

where

$$\phi = \epsilon_0|\ \mathbf{F}$$

and

$$\begin{aligned}
\alpha\ = {} & \mathbf{fix}\ \ \lambda\alpha.\ \langle\ \lambda\kappa_2.\ \mathcal{E}\llbracket \mathtt{E}_1\rrbracket\nu_1(\lambda\epsilon_2.\ (\phi{\downarrow}2)\nu_1\kappa_2.\ \langle\epsilon_2,\ \alpha\rangle\ ,\\
& \lambda\kappa_2.\ (\mathtt{E}_1{=}\mathtt{I}_m\rightarrow\mathcal{A}m,\ \bot_{\mathbf{X}})\ \nu_1(\lambda\epsilon_2.\\
& ((\phi{\downarrow}2)\nu_1\kappa_2(\langle\epsilon_2,\ \alpha\rangle\ \ \mathbf{in}\ \ \mathbf{E})\ ))\ ,\\
& \kappa_1\rangle
\end{aligned}$$

$$\mathcal{V}\llbracket \mathtt{function}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C}\ \rrbracket = \lambda\nu_1\kappa_1\epsilon_1.\ \kappa_1\langle\ \mathit{TRUE},\ \xi,\ \bot_{\mathbf{K}}\rangle$$

where

$$\begin{aligned}
\xi\ = {} & \mathbf{fix}\ \ \lambda\xi.\ (\lambda\nu_2\kappa_2\epsilon_2\rho_2.\\
& (\nu_2{=}\nu_1\rightarrow\mathcal{C}\llbracket \mathtt{C}\rrbracket(\nu_1+1)(\lambda\epsilon_3\rho_3\ .\ \kappa_2\epsilon_3(\rho_3\downarrow 2)\epsilon_2\langle\epsilon_2,\rho_2\rangle,\\
& (\nu_2{>}\ \nu_1\rightarrow\xi(\nu_2-1)\ (\lambda\epsilon_3\rho_3\ .\ \kappa_2\epsilon_3\langle\rho_2\downarrow 1,\rho_3\rangle)\epsilon_2(\rho_2{\downarrow}2\ )\ ,\\
& \qquad\qquad \bot_{\mathbf{S}\rightarrow\mathbf{S}}\ )))
\end{aligned}$$

$$\mathcal{V}\llbracket \mathtt{procedure}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C}\ \rrbracket = \mathcal{V}\llbracket \mathtt{function}\ \ \mathtt{I}\ \mathtt{value}\ \ \mathtt{C}\ \rrbracket$$

$$\begin{array}{l}
\mathcal{V}\llbracket \mathtt{function}\ \ \mathtt{I\ name}\ \ \mathtt{C}\rrbracket = \lambda\nu_0\kappa_0\epsilon_0.\ \kappa_0\langle\ \mathit{FALSE},\ \xi_{\mathrm{restore}}, \kappa_{\mathrm{start}}\rangle \\
\qquad\qquad\qquad\textbf{where} \\
\qquad\qquad\qquad\kappa_{\mathrm{start}} = \mathcal{C}\llbracket\mathtt{C}\rrbracket(\ \nu_0+1)(\lambda\epsilon_1\rho_1.\ (\rho_1{\downarrow}1|\mathbf{A}{\downarrow}3)\epsilon_1(\rho_1{\downarrow}2)\ ) \\
\qquad\qquad\qquad\textbf{and} \\
\qquad\qquad\qquad\xi_{\mathrm{restore}} = \mathbf{fix}\ \ \lambda\xi.\ (\ \lambda\nu_1\kappa_1\epsilon_1\rho_1. \\
\qquad\qquad\qquad\qquad(\nu_1{=}\nu_0 \rightarrow \kappa_1(\epsilon_1{\downarrow}1)\ \langle\epsilon_1\downarrow 2\ \mathbf{in}\ \mathbf{D}, \rho_1\rangle, \\
\qquad\qquad\qquad\qquad(\nu_1{>}\nu_0 \rightarrow \xi(\nu_1{-}1)\kappa_1(\langle\epsilon_1\downarrow 1, \alpha\rangle\ \ \mathbf{in}\ \ \mathbf{E})\ (\rho_1{\downarrow}2)\ , \\
\qquad\qquad\qquad\qquad\qquad\perp_{\mathbf{S}\to\mathbf{S}}\ ))) \\
\qquad\textbf{where in turn} \\
\qquad\qquad\qquad\qquad\alpha = \langle\ \mathbf{fix}\ \ \lambda\pi_1\ .\ (\ \lambda\kappa_2\epsilon_2\rho_2.\ (\epsilon_1{\downarrow}2\ {\downarrow}1)\kappa_2\epsilon_2\langle\rho_1{\downarrow}1\ ,\ \rho_2\rangle), \\
\qquad\qquad\qquad\qquad\qquad\mathbf{fix}\ \ \lambda\pi_2\ .\ (\ \lambda\kappa_2\epsilon_2\rho_2.\ (\epsilon_1{\downarrow}2\ {\downarrow}2)\ \kappa_2\epsilon_2\langle\rho_1{\downarrow}1\ ,\ \rho_2\rangle), \\
\qquad\qquad\qquad\qquad\qquad\mathbf{fix}\ \ \lambda\kappa_3\ .\ (\ \lambda\epsilon_2\rho_2.\ (\epsilon_1{\downarrow}2\ {\downarrow}3)\ \epsilon_2\langle\rho_1{\downarrow}1\ ,\ \rho_2\rangle)\ \rangle
\end{array}$$

$$\mathcal{V}\llbracket \mathtt{procedure}\ \ \mathtt{I\ name}\ \ \mathtt{C}\ \rrbracket = \mathcal{V}\llbracket \mathtt{function}\ \ \mathtt{I\ name}\ \ \mathtt{C}\ \rrbracket$$

$$\mathcal{V}\llbracket\mathtt{E}\rrbracket = \mathcal{E}\llbracket\mathtt{E}\rrbracket$$

$$\mathcal{C}\llbracket\mathtt{I}_m\mathtt{:=}\ \mathtt{E}\rrbracket = \lambda\nu\kappa.\ \mathcal{E}\llbracket\mathtt{E}\rrbracket\nu(\mathcal{A}m\nu\kappa)$$

$$\mathcal{C}\llbracket\mathtt{output}\ \ \mathtt{E}\rrbracket = \lambda\nu\kappa.\ \mathcal{E}\llbracket\mathtt{E}\rrbracket\nu(\lambda\epsilon\rho\sigma.\ \langle\epsilon|\ \mathbf{G},\ \kappa\epsilon\rho\sigma\rangle)$$

$$\begin{array}{l}
\mathcal{C}\llbracket\mathtt{if}\ \ \mathtt{E}_0\ \mathtt{then}\ \ \mathtt{C}_1\ \mathtt{else}\ \ \mathtt{C}_2\rrbracket = \lambda\nu\kappa.\ \mathcal{E}\llbracket\mathtt{E}_0\rrbracket\nu(\lambda\epsilon. \\
\qquad\qquad\qquad\qquad(\epsilon|\mathbf{B}{=}\ \mathit{TRUE} \rightarrow \mathcal{C}\llbracket\mathtt{C}_1\rrbracket, \\
\qquad\qquad\qquad\qquad(\epsilon|\mathbf{B}{=}\ \mathit{FALSE} \rightarrow \mathcal{C}\llbracket\mathtt{C}_2\rrbracket, \\
\qquad\qquad\qquad\qquad\qquad\qquad\perp_{\mathbf{X}}))\nu\kappa\epsilon)
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\llbracket\mathtt{while}\ \ \mathtt{E}\ \mathtt{do}\ \ \mathtt{C}\rrbracket = \lambda\nu.\ (\ \mathbf{fix}\ \ \lambda\pi.\ (\lambda\kappa.\ \mathcal{E}\llbracket\mathtt{E}\rrbracket\nu(\lambda\epsilon. \\
\qquad\qquad\qquad\qquad(\epsilon|\mathbf{B}{=}\ \mathit{TRUE} \rightarrow \mathcal{C}\llbracket\mathtt{C}\rrbracket\nu(\pi\kappa), \\
\qquad\qquad\qquad\qquad(\epsilon|\mathbf{B}{=}\ \mathit{FALSE} \rightarrow \kappa, \\
\qquad\qquad\qquad\qquad\qquad\qquad\perp_{\mathbf{K}}))\ \epsilon)))
\end{array}$$

$$\mathcal{C}\llbracket\mathtt{C}_1;\ \mathtt{C}_2\rrbracket = \lambda\nu\kappa.\ \mathcal{C}\llbracket\mathtt{C}_1\rrbracket\nu(\mathcal{C}\llbracket\mathtt{C}_2\rrbracket\nu\kappa)$$

$$\begin{array}{l}
\mathcal{C}\llbracket\mathtt{begin}\ \ \mathtt{var}\ \ \mathtt{I}{=}\ \mathtt{E};\ \mathtt{C}\ \mathtt{end}\ \rrbracket = \lambda\nu\kappa.\ \mathcal{V}\llbracket\mathtt{E}\rrbracket\nu(\lambda\epsilon_1\rho_1.\ \mathcal{C}\llbracket\mathtt{C}\rrbracket(\nu{+}1)\ (\lambda\epsilon_2\rho_2\ . \\
\qquad\qquad\qquad\qquad\qquad\kappa\epsilon_2(\rho_2\downarrow 2))\ \epsilon_1\langle\epsilon_1\ \ \mathbf{in}\ \ \mathbf{D},\rho_1\rangle)
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\llbracket\mathtt{call}\ \ \mathtt{E}_0\mathtt{E}_1\rrbracket = \mathcal{E}\llbracket\mathtt{E}_0\mathtt{E}_1\rrbracket \\
\mathcal{C}\llbracket\mathtt{result}\ \ \mathtt{E}\rrbracket = \mathcal{E}\llbracket\mathtt{E}\rrbracket
\end{array}$$

$$\begin{array}{l}
\mathcal{A} = \lambda m.\ (\ \mathbf{fix}\ \ \lambda\xi.\ \ (\lambda\nu\kappa\epsilon_1\rho_1\ . \\
\quad(\nu{=}m \rightarrow\ \ ((\rho_1{\downarrow}1){\in}\mathbf{A}\rightarrow(\rho_1{\downarrow}1|\mathbf{A}{\downarrow}2)\nu\kappa\epsilon_1(\rho_1{\downarrow}2), \\
\qquad\qquad\quad((\rho_1{\downarrow}1){\in}\mathbf{E}\rightarrow\kappa\epsilon_1\langle\epsilon_1\ \ \mathbf{in}\ \ \mathbf{D},\ \rho_1\downarrow 2\rangle\ , \\
\qquad\qquad\qquad\qquad\perp_{\mathbf{S}\to\mathbf{S}}\ )), \\
\quad(\nu{>}m \rightarrow \xi(\nu{-}\ 1)\ (\lambda\epsilon_2\rho_2\ .\ \kappa\epsilon_2\ \langle\rho_1\downarrow 1, \rho_2\rangle)\epsilon_1\ (\rho_1{\downarrow}2), \\
\qquad\quad\perp_{\mathbf{S}\to\mathbf{S}}\ ))
\end{array}$$

Figure 5. Examples of programs using the call by name feature

```
P₁ ≡ program
       begin var dbl := function i value i := 2 × i;
         begin var inc := procedure i name i := i + 1;
           begin var x := read;
             ((((output x;
             x := dblx); output x);
             call inc x); output x)
           end
         end
       end
```

```
P₂ ≡ program
       begin var magic := function(n, i, a, b) name4
                   (i := 0; while i < n do ((i := i + 1; a := b); result 999));
         begin var prod := function(i, j, p, q) value4 result i × j;
           begin var p := 999;
             begin var q := 999;
               begin var dummy := 999;
                 begin var a := 1;
                   (((dummy := magic(4, p, a, a × p); output a); a := 0);
                   dummy := magic(2, p, dummy,
                        magic(3, q, a, (a + prod(p, q, dummy, dummy)))));
                   output a)
                 end
               end
             end
           end
         end
       end
```

Note that the second call to *magic* is effectively equivalent to the statements:

```
(p := 0;
while p < 2 do
  ((p := p + 1; q := 0);
  while q < 3 do
    (q := q + 1; a := a + p × q)))
```