PB-173

B.H. Mayoh: Comparative Semantics

TRYK: DAIMI/RECAU

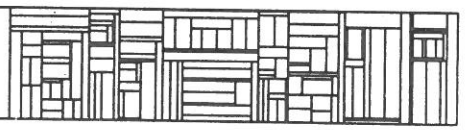
ISSN 0105-8517

COMPARATIVE SEMANTICS OF PROGRAMMING LANGUAGES

Brian H. Mayoh

DAIMI PB-173
April 1984

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



COMPARATIVE SEMANTICS OF PROGRAMMING LANGUAGES

The flood of new programming and specification languages shows no sign of abating, but very few of these languages have a formal definition. The advantages of knowing precisely what is specified in a specification and exactly how a program can behave are obvious, but none of the existing formal definition methods are completely satisfactory. Theoreticians have not been idle, but they have concentrated on problems that are not immediately relavant to language designers (algebraic and categoric structuring of definitions, refined notions of concurrency and the like). In the belief that the answer to some of the language designers' problems is "use different formalisms to define fragments of the languages precisely", we advocate the study of comparative semantics. This paper is a contribution to this study, prompted by the fact that the parallel aspects of ADA seem to require a quite different kind of formal semantics from that used for sequential ADA in [A].

CONTENTS

1. OUTLINE

Most of the important answers to the question "What is the meaning of a program/specification?" are classified in section 2. When one has fixed the meaning of programs/specifications, one has to decide how the semantics should be presented. Several possible ways of presenting meanings are described in section 3, while ways of presenting semantics without revealing meanings are described in section 5. When a language can have several semantics or different fragments of a language have different semantics, one needs semantic transformers like those described in section 6.

Since real programming/specification languages have complex features, their formal semantics may be complicated and language users need conceptual models for programs/specifications like the graphs, nets and diagrams in section 4. Because a wide variety of semantics can be defined from such models, they can form a basis for comparing semantics. In section 7 diagram models are used in the comparison of three very different formal definitions of a semantically challenging mini language. If one only reads section 7, one will have an impression of the current "state of the art" in formal semantics.

2. PROGRAM AND SPECIFICATION MEANINGS

In this section we will describe 23 natural meanings for programs and specifications. These meanings can be divided into relational, functional and set meanings. Relational and functional meanings are natural for programs and specifications in languages with an explicit notion of state, but set meanings are natural for languages which focus on events like messages passing between processes.

We shall not mention specifications hereafter, because the similarity between programs and specifications has been emphasized enough by now.

2.1 Relational Meanings

What is the meaning of a program? What kinds of questions about the behaviour of a program should be answered by a formal semantics? Figure 1 shows a way of thinking about the behaviour of programs in which the only questions a semantics has to answer are of the form "can the program produce result r when started with data d".

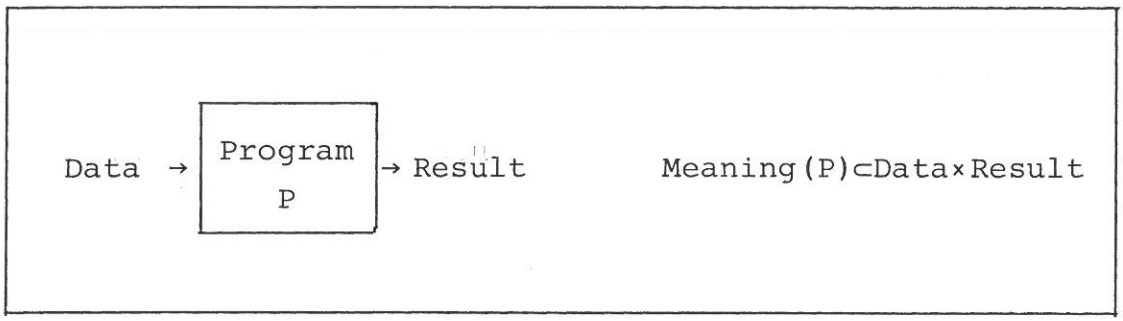


Fig. 2.1 Data-Result Relational Semantics.

The syntax of a programming language determines how a program is built from primitive fragments using combinators like

;loop, both, either, if, ...

If one gives a meaning to every possible fragment, and one specifies a way of combining meanings for every combinator, then every program gets a meaning. For most programming languages it is easy to define a set of internal states, then give the meaning of a primitive fragment F as

$$M_1(F) \subset \text{State} \times \text{State}$$

Example

Once primitive fragments have been given a relation as a meaning, we must specify a way of combining these relations for each combinator in the language. Consider the compositional combinator ";". The usual specification for this is

$$\text{ANGELIC} \quad R_{S1;S2}(x,z) \ .\equiv. \ \exists y[R_{S1}(x,y) \& R_{S2}(y,z)]$$

but another possibility is

$$\text{DEMONIC} \quad R_{S1;S2}(x,z) \ .\equiv. \ \exists y R_{S1}(x,y) \& \forall y[R_{S1}(x,y) \rightarrow R_{S2}(y,z)]$$

□

DEMONIC composition is realistic because it recognises that unfortunate choices can lead to the jamming of computations.

Other kinds of relational composition are natural in a semantics which can answer questions like

- can the program P go into a loop, when started on data d
- can the program P jam, when started on data d
- can the program P terminate without giving a result, when started on data d?

In such a semantics there might be states like divergence, jammed, stopped and the meaning of a primitive fragment F might be

$$M_2(F) \subset \text{State} \times \text{ExtendedState}$$
$$M_3(F) \subset \text{ExtendedState} \times \text{ExtendedState}$$

Example

Consider the program "loop random;exit when (0)." in a language with random assignments, loops and exits. Suppose the meaning of the primitive fragments in the program is

$$M_1(\underline{\text{random}})(s,s') \quad .\equiv. \quad \text{TRUE}$$
$$M_1(\underline{\text{exit when}}(0))(s,s') \quad .\equiv. \quad s' = 0 = s$$

where s and s' range over the set State = (0,1,2,3,...). The angelic combinator would give

$$M_1(\underline{\text{random}};\underline{\text{exit when}}(0))(s,s') \quad .\equiv. \quad s' = 0$$

because the random assignment may give s' = 0. The demonic combinator would give

$$M_1(\underline{\text{random}};\underline{\text{exit when}}(0))(s,s') \quad .\equiv. \quad \text{FALSE}$$

because the random assignment may give s' = 1. To get a loop combinator we can define

$$\text{ExtendedState} = \text{State} + (\bar{0},\bar{1},\bar{2},\ldots)$$
$$M_2(\underline{\text{random}})(s,s') \quad .\equiv. \quad s' \in \text{State}$$
$$M_2(\underline{\text{exit when}}(0))(s,s') \quad .\equiv. \quad (s=s' \,\&\, s'\neq 0) \vee (s=0 \,\&\, s'=\bar{0})$$
$$M_2(\underline{\text{random}};\underline{\text{exit when}}(0))(s,s') \quad .\equiv. \quad s'=\bar{0} \vee s' \in \text{State} - \{0\}$$
$$M_2(\underline{\text{loop random}};\underline{\text{exit when}}(0).)(s,s') \quad .\equiv. \quad s'=\bar{0}$$
$$M_2(\underline{\text{loop}}\ C.)(s,s') \quad .\equiv. \quad M_2(C;\underline{\text{loop}}\ C)(s,s')$$

This semantics is unsatisfactory because

$$M_2(\underline{\text{loop random}};\underline{\text{exit when}}(0)\underline{\text{random}}.)(s,s') \quad .\equiv. \quad \text{FALSE}$$

If we assume a strictness - $\bar{0}\bar{1}\bar{2}\ldots$ are terminal states -

$$M_3(C)(s,s') \quad .\equiv. \quad M_2(C)(s,s') \vee s = s' \;\&\; s \in (\text{ExtendedState}-\text{State})$$

we get a natural semantics.

□

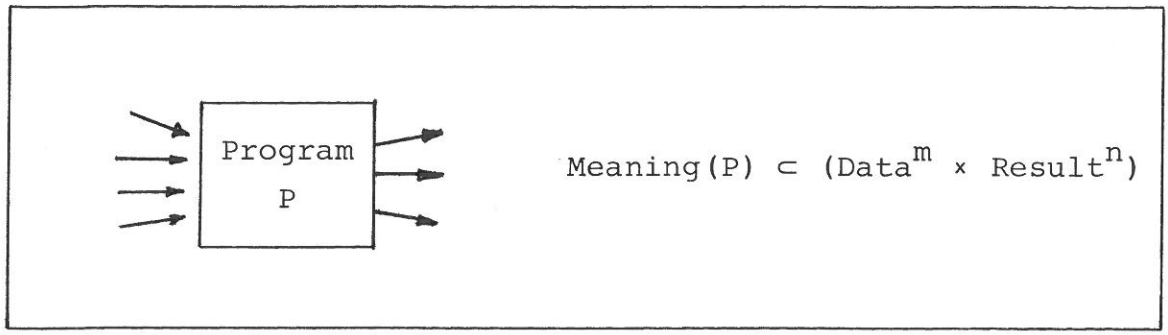


Fig. 2.2 Source Sink Relational Semantics.

Most programming languages have a file mechanism which allows a program to receive data from many sources and transmit results to many sinks. The semantics of such a language should be able to answer questions like "Can the program P give the result $r_1 \ldots r_n$ at the n sinks when started with data $d_1 \ldots d_m$ at the m sources?". In such a semantics the meaning of a primitive fragment might be

$$M_4(F) \subset State^m \times State^n$$
$$M_5(F) \subset State^m \times ExtendedState^n$$
$$M_6(F) \subset ExtendedState^m \times ExtendedState^n$$

In M_4-M_6 we implicitly assume that there is a datum at each fragment source and a result at each fragment sink. Sometimes this assumption is unnatural and the semantics should be able to answer questions like "Can program P give results (r_1,r_3,r_5) at (sink1,sink3,sink5) when started with data (d_2,d_3) at (source2,source3)?". If we define a pattern as "all subsets of (1,2,...,m) for some m", then more general relational semantics are possible:

$$M_7(F) \subset State^\pi \times State^\rho$$
$$M_8(F) \subset State^\pi \times ExtendedState^\rho$$
$$M_9(F) \subset ExtendedState^\pi \times ExtendedState^\rho$$

where π and ρ are patterns.

Example

Consider the program "loop either First or Second;exit when(0)." in a language with alternatives, loops and exits. Suppose the meaning of the primitive fragments in the program is

M_4(First)((s1,s2,s3),(s1',s2',s3')) .≡. s3'=s1 & s2=s2'

M_4(Second)((s1,s2,s3),(s1',s2',s3')) .≡. s3'=s2 & s1=s1'

M_4(exit when(0))((s1,s2,s3),(s1',s2',s3')) .≡. (0=s3=s1'=s2'=s3') ∨ (0≠s2 & s1=s1' & s2=s2' & s3=s3')

The program merges the values of s1 with the values of s2 if we define the combinators

M_4(either C1 or C2.)((s1,s2,s3),(s1',s2',s3'))
.≡. M_4(C1)((s1,s2,s3),(s1',s2',s3')) ∨ M_4(C2)((s1,s2,s3),(s1',s2',s3'))

M_4(C1;C2)((s1,s2,s3),(s1',s2',s3')) .≡. ∃s1",s2",s3"...s1≠0∨s2≠0∨
(M_4(C1)((s1,s2,s3),(s1",s2",s3")) & M_4(C2)((s1",s2",s3"),(s1',s2',s3')))

M_4(loop C.)((s1,s2,s3),(s1',s2',s3'))
.≡. M_4(C;loop C)((s1,s2,s3),(s1',s2',s3'))

This semantics is simpler if we use patterns;

M_7(First)((s1),(s3')) .≡. s3' = s1

M_7(Second((s2),(s3')) .≡. s3' = s2

and we agree that M_4 and M_7 have the same exit-, either-, loop-, and ;-combinators.

□

There are useful programs which give a stream of results when supplied with a stream of data. The semantics of the corresponding programming language should be able to answer questions like "Can the operating system program P give result stream $(r_1,r_2,...)$ when started with the data stream $(d_1,d_2,...)$?".

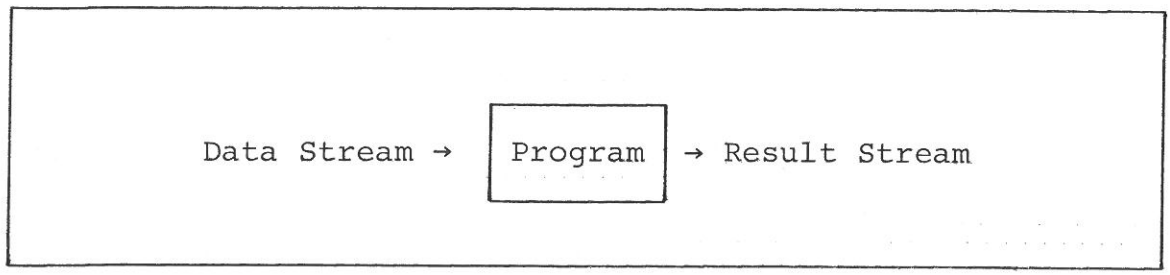


Fig. 2.3 Stream Relational Semantics.

Among the various possibilities for stream semantics of fragments we mention

$$M_{10}(F) \subset S^* \times S^*$$
$$M_{11}(F) \subset S^\infty \times S^\infty$$

where S is State or ExtendedState, S^* is the set of finite S-sequences, and S^∞ is the set of finite or infinite S-sequences (N).

Example

The meaning of a program fragment F which merges two finite streams might be given by

$$((p_1,p_2,\ldots,p_\ell),(q_1,q_2,\ldots,q_m)),(r_1,r_2,\ldots,r_n)) \in M_{10}(F)$$
$$.\equiv. \quad (n=\ell+m) \;\&\; (r_1,r_2,\ldots,r_n) \text{ is an interleaving of } (p_1,p_2,\ldots,p_\ell) \text{ and } (q_1,q_2,\ldots,q_m)$$

□

2.2 Functional Meanings

The analogy between executing a program and evaluating a function is very close. We shall describe various ways of preserving the analogy in the presence of parallelism and non-determinism. The first way is the power domain approach, "The meaning of a program is a function from the initial data to the set of possible results":

$$M_{12}(F) \in \text{State} \to \text{SubsetofState}$$

Another popular way of preserving the analogy between the execution of a program and the evaluation of a function is the predicate transformer approach:

$$M_{13}(F) \in \text{Predicate} \to \text{Predicate}$$

where Predicate = SubsetofState.

In the literature various kinds of predicate transformers have been proposed to deal with the problem of programs that may not always terminate. For the same reason many kinds of power domains have been described, and we made the distinction between State and ExtendedState in section 2.1.

The most popular way of preserving the analogy between programs and functions is the continuation approach:

$$M_{14}(F) \in \text{Continuation} \to \text{Continuation}$$

where Continuation = State → Answer.

Note that there is no essential difference between continuations and predicates when Answer has just two elements. The power of the continuation approach is partly due to the introduction of the set Answer, so one can expect similar power from the introduction of a set Oracle in the oracular approach:

$$M_{15}(F) = \text{Oracle} \times \text{State} \to \text{State}$$

Introducing the set Oracle to take care of non-determinism and parallelism is no different from introducing a set Environment to take care of the "context" of a program fragment in any of relational or functional meanings for program fragments.

Example

Consider a language which allows random assignments. A program fragment "x = ?" could have the relational meaning

$$R(s,s') \ .\equiv. \ \exists n(s' = s[n/x])$$

where s[n/x] is the result of assigning n to x in the state s. In the power domain approach we might have

$$M_{12}(x = ?)s = \{s[0/x],s[1/x],...\}$$

In the predicate transfer approach we might have

$$s \in M_{13}(x = ?)P(s) \ .\equiv. \ \forall n.s[n/x] \in P$$

In the continuation approach we might have

$$M_{14}(x = ?)\theta(s) = \theta(s[0/x]) \cap \theta(s[1/x]) \cap ....$$

whereas the oracular meaning might be

$$M_{15}(x = ?)(n,s) = s[n/x].$$

If our language with random assignment also has block structure, the meaning of the fragment "x = ?" may depend on which block context is given by an environment ρ, the oracular meaning of our fragment might be

$$M_{15}(x = ?)\rho(n,s) = s[n/\rho(x)]$$

result of assigning n to ℓ in state s where ℓ is the location given by ρ to x.

Environments can be introduced in meanings M_1-M_{14} in just the same way.

□

We have described functional meanings as if they could not capture the more elaborate relational meanings in the last section. However there is no reason why the set State in M_{12}-M_{15} could not be replaced by ExtendedState or $State^m$ or $State^\pi$ or State* or $State^\infty$.

Example

We can capture the well known connection between random assignment and fairness by defining

$$M_{15}(F) \in \text{Oracle} \times State^\infty \to State^\infty$$
$$M_{15}(x = ?)((n_1,n_2,\ldots),(s_1,s_2,\ldots)) = (s_1[n_1/x],s_2[n_2/x],\ldots)$$

and requiring that oracles be fair, i.e. all possible values of n occur in any infinite oracle $(n_1,n_2,\ldots)$. In a language with a merge operator one could insist on fair oracles and define

$$M_{15}(\text{merge})((n_1,n_2,\ldots),(d_{11},d_{12},\ldots,d_{21},d_{22},\ldots,d_{31},\ldots))$$
$$= M_{15}(\text{merge})((n_2,\ldots),(d'_{11},d'_{12},\ldots,d'_{21},\ldots,d'_{31},\ldots))$$

where

$$d'_{ij} = \underline{\text{if}}\ i = n_1\ \underline{\text{then}}\ d_{i+1,j}\ \underline{\text{else}}\ d_{ij}$$

□

There is a natural connection between oracles and communication histories; one way of looking at the semantics for Hoare's language for communicating sequential processes CSP [F] is to view communications with each individual sequential process as messages from an oracle to the process. In this view the essential problem in the semantics of CSP is the matching of oracles. The natural connection between oracles and schedulers in operating and multiple processor systems should not be forgotten. An oracular semantics for important new languages like ADA may well suggest a practical way of implementing the languages.

2.3 Set Meanings

There have been several voices claiming that the subtleties of languages with true parallelism cannot be captured by program meanings like those we have considered hitherto. The notion of a global state is dubious when a program may consist of many fragments, which can be executed on different computers in places far from one another. Figure 2.4 illustrates an alternative view:

- events correspond to program fragments reading data, writing results, or communicating with one another;
- events are partially ordered by a relation that can be interpreted as "causes" or "earlier in time".

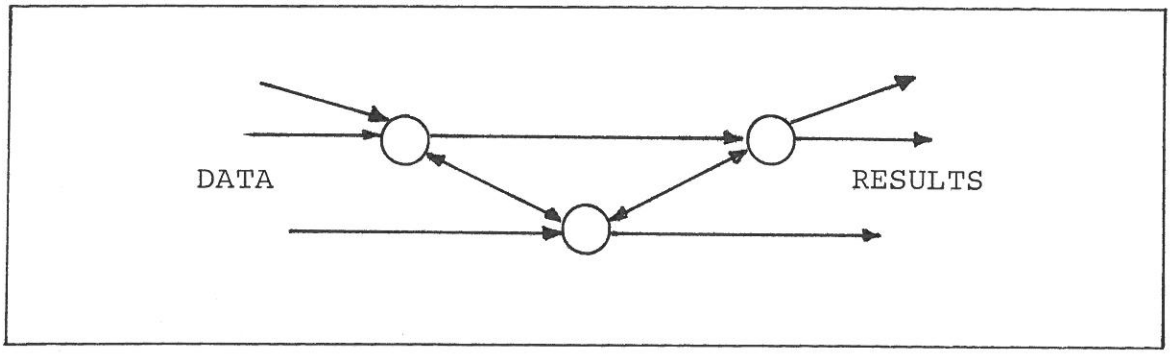


Fig. 2.4 Event Semantics.

Definition

An event sequence is a linearly ordered set of events. An event tree is a set partially ordered by $\leq$ such that $e,e' \leq f \rightarrow e \leq e' \vee e' \leq e$. An event trace is a partially ordered set of events.

The distinction between event sequences, trees and traces prompts various ways of giving meaning to program fragments.

$M_{16}(F) \subset \text{EventSequences}$
$M_{17}(F) \in \text{EventTrees}$
$M_{18}(F) \in \text{EventTraces}$
$M_{19}(F) \subset \text{EventTrees}$
$M_{20}(F) \subset \text{EventTraces}$

In the literature event sequences have been studied under the name computation histories, event trees have been studied under the name behaviours, and event traces have been investigated under the name event structures [W2].

Example
There are languages where the fragments

(F1) either P; Q or P; R.
(F2) P; either Q or R.

have different meanings. The two fragments have the same meaning in a sequence semantics

$$M_{16}(F_1) = M_{16}(F_2) = \{\langle P,Q\rangle,\langle P,R\rangle\}$$

but they have different meanings in tree and trace semantics

$$M_{17}(F_1) = M_{18}(F_1) = \phi \begin{cases} P\text{-}Q \\ P\text{-}R \end{cases}$$

$$M_{17}(F_2) = M_{18}(F_2) = \phi\text{-}P \begin{cases} Q \\ R \end{cases}$$

$$M_{19}(F_1) = M_{20}(F_1) = \{M_{17}(F_1)\} = \{M_{18}(F_1)\}$$

$$M_{19}(F_2) = M_{20}(F_2) = \{M_{17}(F_2)\} = \{M_{18}(F_2)\}$$

The distinction between F_1 and F_2 is similar to that between internal and external choice in CSP [F].

□

The disadvantage of many set semantics is that set composition is more awkward than relation composition and function composition. There are set semantics which do not have this disadvantage.

$M_{21}(F) \subset \text{EventSequence} \times \text{Accept} \times \text{Refuse}$
where Accept = Refuse = SubsetofEvents

$M_{22}(F) \subset \text{EventSequence} \times \text{Future}$
where Future = EventTrees

$M_{23}(F) \subset \text{Resumption} = \text{Subsetof}(\text{Event} \times \text{Resumption})$

Accept-refuse and future semantics were introduced in [R], but resumption semantics are much older.

Example

For the fragments in the last example we have

$$\langle P,(Q),(R)\rangle \in M_{21}(F_1) - M_{21}(F_2)$$
$$\langle P,(Q,R),\phi\rangle \in M_{21}(F_2) - M_{21}(F_1)$$
$$\langle P,Q\rangle \in M_{22}(F_1) - M_{22}(F_2)$$
$$\langle P,\phi \lessdot^{Q}_{R}\rangle \in M_{22}(F_2) - M_{21}(F_1)$$

So both accept-refuse and future semantics distinguish between the two fragments. The natural resumption semantics also makes this distinction:

$M_{23}(F_1) = \{\langle P,r_1\rangle,\langle P,r_2\rangle\}$
where $r_1 = \{\langle Q,\phi\rangle\}$ and $r_2 = \{\langle P,\phi\rangle\}$

$M_{23}(F_2) = \{\langle P,r_3\rangle\}$
where $r_3 = \{\langle Q,\phi\rangle,\langle R,\phi\rangle\}$

□

3. CONCRETE PRESENTATIONS

There are many ways of presenting the semantics of a programming language. In this section we shall describe presentations that are concrete in the sense that they use information about the kinds of meaning the semantics gives to programs.

3.1 Relational Presentations

In the area of relational data bases two ways of presenting relations dominate. In the relational algebra approach one presents a relation as a combination of other relations, using powerful combinators like union, composition, join and projection. In the relational calculus approach one presents a relation as a logical formula, using logical operators like conjunction, disjunction, implication, negation and quantifiers. An interesting variation of the calculus approach is to use a logical programming language instead of logical formulae. If the relational semantics of a language *L* is given in PROLOG, then *L*-programs can be executed by a PROLOG interpreter and the only reason for having *L*-compilers and *L*-interpreters is efficiency.

Perhaps the most elegant way of presenting a relation is by proof rules; some instances of the relation are given by proof rules without premisses, and all other instances can be derived using proof rules with premisses [P1]. Figure 3.1 illustrates our four ways of presenting relations.

algebra	C1;C2 is composition of C1 and C2	either C1 or C2 = C1 ∪ C2
calculus	C1;C2(x,z) .≡. ∃y(C1(x,y)&C2(y,z))	either C1 or C2(x,y) .≡. C1(x,y)∨C2(x,y)
PROLOG	C1;C2(x,z) ← C1(x,y),C2(y,z).	either C1 or C2(x,y)←C1(x,y). either C1 or C2(x,y)←C2(x,y).
proof rules	$\frac{C1,x\to y \ \ldots \ C2,y\to z}{C1;C2.,x\to z}$	$\frac{C1,x\to y}{\text{either C1 or C2.},x\to y}$ $\frac{C2,x\to y}{\text{either C1 or C2.},x\to y}$

Fig. 3.1 Comparison of Relational Presentations.

So far, our relational presentations have been denotational, but each of them becomes operational if one has to take the transitive closure of the presented relation to get the meaning of a program. Just as the usual composition of relations may not be appropriate in a relational semantics, the definition of transitive closure may be unusual.

Comment: Since relations form a complete lattice, presentations can use fixpoints - one can define the relation ⟦loop C.⟧ as the least solution of ⟦loop C.⟧ = ⟦C; loop C.⟧.

3.2 Functional Presentations

The natural way to present a functional semantics of a programming language *L'* is to specify functions in a known language *L*. If *L* is a programming language, then an *L*-compiler can be used to execute *L'*-programs and efficiency is the only reason for writing *L*-compilers and *L*-interpreters. This advantage of choosing a real programming language *L* is usually outweighed by the disadvantage that *L*-functions are obscure because they contain much irrelevant detail. A cleaner presentation is given by identifying *L'* and *L*; the LISP interpreter in LISP is elegant, and the ADA semantics in ADA [A] is not inelegant. However such "circular" presentations have the serious disadvantage of concealing precisely the obscure features of a language that should be revealed by a presentation.

When presenting a functional semantics, it is very tempting to use a special mathematical language for specifying functions. The lambda calculus is often chosen, and the calculus of combinators has also been used. Other possibilities have been unfairly neglected: logic can be used to present predicate transformer semantics: attribute grammars specify functions from inherited attributes to synthesized attributes and they reflect the syntactic structure of programs.

Example

A mathematical presentation of the functional meaning of C1;C2 is

$$[\![C1;C2]\!](s) = [\![C2]\!]([\![C1]\!](s))$$

whereas an attribute grammar presentation might be

$$[\![C1;C2]\!](\downarrow s,\uparrow s') = [\![C1]\!](\downarrow s,\uparrow s'')[\![C2]\!](\downarrow s'',\uparrow s').$$

□

So far all functional presentations have been denotational but each of them becomes operational if one has to take the transitive closure of the presented function to get the meaning of a program. Transitive closures of functions are sometimes very obscure, and occasionally they may not even exist; one must be careful with composition and recursion in presentations of functional semantics; power domain semantics are particularly notorious.

3.3 Set Presentations

The literature on set semantics contains many small examples to illustrate the subtle distinctions made by particular semantics, but says little about presentations. An indirect way of presenting a set semantics is to invent a model, to give a general definition of the meaning of a model, and to describe general ways of building large models from small models. Graphs are natural models of programs, graph grammars can be used to generate large graphs for small graphs, and graphs can be given a set meaning by identifying events (configurations) with edges (vertices). Petri nets are also popular program models, and they give a set meaning if one identifies events with the firing of transitions. Equation schemes are another much studied kind of program model, and with some ingenuity they too can be given set meanings. Figure 3.2 shows graph, net and scheme models for two small programs

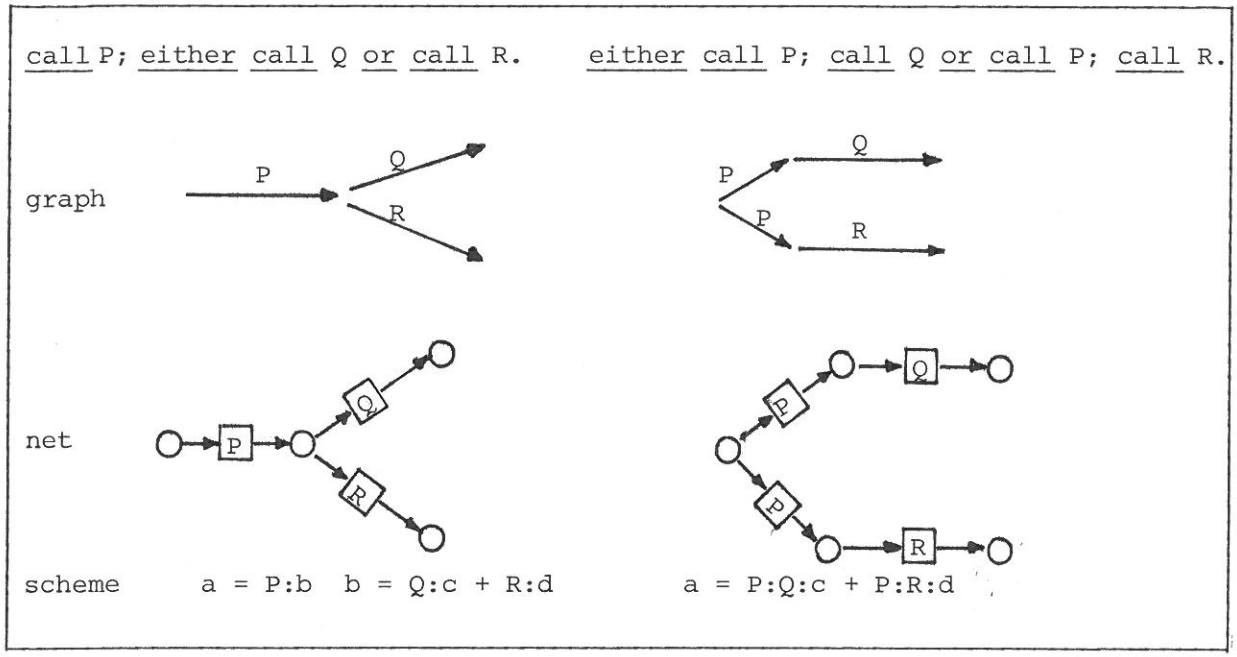


Fig. 3.2 Comparison of Program Models.

In the next section we introduce diagrams as general models of programs. Graphs, equation schemes and most versions of Petri nets are particular kinds of diagrams [M2, M3]. None of these models prejudge the decision between "interleaving", "synchronous", and "true parallelism" semantics; all of them allow moves (events) to be either "single computation steps" or "appearances of a datum at a port".

4. A CANONICAL PRESENTATION

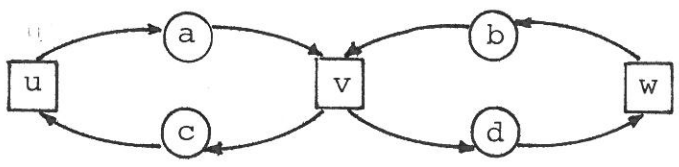
The task of presenting and comparing the semantics of programming languages would be much simpler if everyone could agree on a canonical model for programs. Many program models have been suggested in the literature; the process nets in [P2] seem to be suitable for canonical program models, but we will choose the somewhat similar diagram models.

Definition
A diagram consists of a set C of configurations and a set M of moves. Each move in a diagram (C,M) has a label, a source and a sink; each source and sink is built from C, using + and ×.

Example
In the diagram model of a program configurations correspond to control points, and moves correspond to combinations of program statements that must be executed concurrently. In the three move diagram

$$C = (a,b,c,d) \qquad M = (c \xrightarrow{u} a,\ a\times b \xrightarrow{v} c\times d,\ d \xrightarrow{w} b)$$

the v-move models a program rendezvous in just the same way as the v-transition in the corresponding Petri net.



□

4.1 Canonical Relational Semantics

Diagrams acquire a relational meaning by

- assigning sets to configurations;
- defining + and × on the assigned sets;
- assigning relations to move labels;
- defining composition of relations.

The semantics, M_1 to M_{11}, in section 2.1 differ only in the sets assigned to configurations; they agree in defining + and × as disjoint union and cartesian product of sets.

Example

Consider our three move diagram. Let us assign the integers to each of its configurations. Let us assign unnatural relations to each of its moves.

$$R_u(c,a) \quad .\equiv. \quad (c<a)$$

$$R_v(a,b,c,d) \quad .\equiv. \quad (c = \min(a,b)) \wedge (d = \max(a,b))$$

$$R_w(d,b) \quad .\equiv. \quad (d>b)$$

Let us agree on the usual meanings of +, × and composition. For any sequence of moves in the diagram we can derive a relation on integers; the meaning of the move sequence (v,u,w) is the relation

$$R(a,b,a',b') \quad .\equiv. \quad (\min(a,b) < a') \,\&\, (\max(a,b) > b')$$

□

4.2 Canonical Functional Semantics

Diagrams acquire a functional meaning by

- assigning sets to configurations;
- defining + and × on the assigned sets;
- assigning functions to move labels;
- defining functional composition.

In a power domain semantics M_{12} power sets are assigned to configurations, the functions assigned to move labels are specified in strange ways, and functional composition is unusual. In a predicate transformer semantics M_{13} one also assigns power sets (≡ predicates) to configurations, but the functions assigned to move labels are specified normally. In a continuation semantics M_{14} continuation domains are assigned to configurations, functions assigned to move labels are specified normally, but the

definition of +, × and composition may be problematic. In an oracular semantics M_{15} functions assigned to move labels may have an extra argument domain for the oracle.

Example

Consider the relational semantics for the three move diagram in section 4.1. This becomes an oracular semantics if the moves are assigned the following functions

$$f_u(c,n) = c + n$$

$$f_v(a,b) = \langle \min(a,b),\ \max(a,b) \rangle$$

$$f_w(d,n) = d - n$$

where the oracular argument n ranges over the positive integers. For any sequence of moves in the diagram we have an integer function; the meaning of the move sequence <v,u,w> is the function

$$f(a,b,n_1,n_2) = \langle \min(a,b) + n_1,\ \max(a,b) - n_2 \rangle$$

Suppose now that we assign the family of subsets of the integers to the diagram configurations, not just the integers. We get a forward semantics, if the diagram moves are assigned the functions

$$F_u(P) = \bigcup_{c \in P} (a \mid c<a)$$

$$F_v(P) = \bigcup_{(a,b) \in P} (\langle \min(a,b),\ \max(a,b) \rangle)$$

$$F_w(P) = \bigcup_{d \in P} (b \mid d>b)$$

we get a backward semantics, if the diagram moves are assigned the functions

$$B_u(Q) = \bigcup_{a \in Q} (c \mid c<a)$$

$$B_v(Q) = (<a,b> \mid <\min(a,b), \max(a,b)> \in Q)$$

$$B_w(Q) = \bigcup_{b \in Q} (d \mid d>b)$$

For any sequence of moves in the diagram we have a forwards and a backwards function from integer predicates to integer predicates; the forwards meaning of the move sequence (v,u,w) is the predicate transformer

$$F(P) = \bigcup_{(a,b) \in P} (<c',b'> \mid \min(a,b) < a' \ .\&. \ \max(a,b) > b')$$

the backwards meaning of the move sequence (v,u,w) is the predicate transformer

$$B(Q) = \bigcup_{(a',b') \in Q} (<a,b> \mid \min(a,b) < a' \ .\&. \ \max(a,b) > b')$$

The forward predicate transformer semantics becomes a power domain semantics, if one specifies the move functions by

$$f_u(c) = (a \mid c<a)$$

$$f_v(a,b) = (<\min(a,b), \max(a,b)>)$$

$$f_w(d) = (b \mid d>b)$$

The backward predicate transformer semantics becomes a continuation semantics is one specifies the move functions by

$$f_u\theta(c) = \bigcup(\theta(a) \mid c<a)$$

$$f_v\theta(a,b) = \theta(\min(a,b), \max(a,b))$$

$$f_w\theta(d) = \bigcup(\theta(b) \mid d>b)$$

In these function specifications $\bigcup$ is join in the "answer" domain, not set union, because in the continuation semantics configurations are assigned sets of functions from integers to

"answers". Notice also that the continuation equation for f_v assumes the following definition of $\times$:

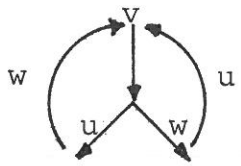
> if c_1 is assigned $S_1 \to A$ and c_2 is assigned $S_2 \to A$,
> then $c_1 \times c_2$ is assigned $S_1 \times S_2 \in A$.

4.3 Canonical Set Semantics

Diagrams acquire a set meaning when one interprets the occurrence of a move as an event. In the simplest interpretation M_{16} the meaning of a diagram is the set of move sequences that can occur. In other interpretations M_{17}-M_{23} the set of move sequences that can occur is structured in various ways.

Example

The set of move sequences that can occur in our three move diagram D is given by the paths in the graph



This set of move sequences can be structured as an event tree

$$M_{17}(D) = \ldots\ v_1 \begin{cases} u_1 - w_1 - v_2 \begin{cases} \ldots \\ \ldots \end{cases} \\ w_1 - u_1 - v_2 \begin{cases} \ldots \\ \ldots \end{cases} \end{cases}$$

$$M_{19}(D) = (M_{17}(D)).$$

The set of move sequences can be structured as an event trace in two ways

$M_{18}(D)$ = ... v_1 ⟨ $u_1 — w_1$ / $w_1 — u_1$ ⟩ v_2 ⟨ ... / ... ⟩

$M_{20}(D)$ = (... v_1 ⟨ u_1 / w_1 ⟩ v_2 ⟨ ... / ... ⟩)

where the second way corresponds to the scenario approach [B1].

One can divide the set of move sequences in four parts and give a future semantics $M_{22}(D)$

past event sequence	future event trees
ends in v	$u_1 — w_1 — v_2$⟨ ⟩ $w_1 — u_1 — v_2$⟨ ⟩
ends in v,u	$w_1 — v_2$⟨ ... / ... ⟩
ends in v,w	$u_1 — v_2$⟨ ... / ... ⟩
ends in u,w or w,u	v_1⟨ $u_1 — w_1 — v_2$ / $w_1 — u_1 — v_2$ ⟩

or an accept-refuse semantics $M_{21}(D)$

past event sequence	accept	refuse
ends in v	u,w	v
ends in v,u	w	u,v
ends in v,w	u	v,w
ends in u,w or w,u	v	u,w

These four kinds of past event sequence become four resumptions in the resumption semantics $M_{23}(D)$:

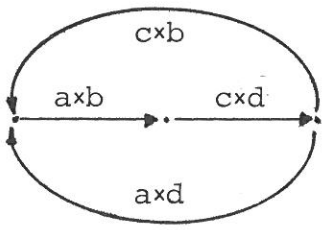
$$r_1 = (\langle u_1, r_2\rangle, \langle w, r_3\rangle)$$
$$r_2 = (\langle w, r_4\rangle)$$
$$r_3 = (\langle u, r_4\rangle)$$
$$r_4 = (\langle v, r_1\rangle)$$

□

Diagrams also acquire a set meaning, when one interprets "coming to a configuration" as an event.

Example

The set of configuration sequences that can occur in our three move diagram D is given by the paths in the graph



This set of configuration sequences can be structured as an event tree

$$M_{17}(D) = \ldots\ a\times b \text{ — } c\times d \begin{cases} c\times b \text{ — } a\times b \text{ — } c\times d \begin{cases} \ldots \\ \ldots \end{cases} \\ a\times d \text{ — } a\times b \text{ — } c\times d \begin{cases} \ldots \\ \ldots \end{cases} \end{cases}$$

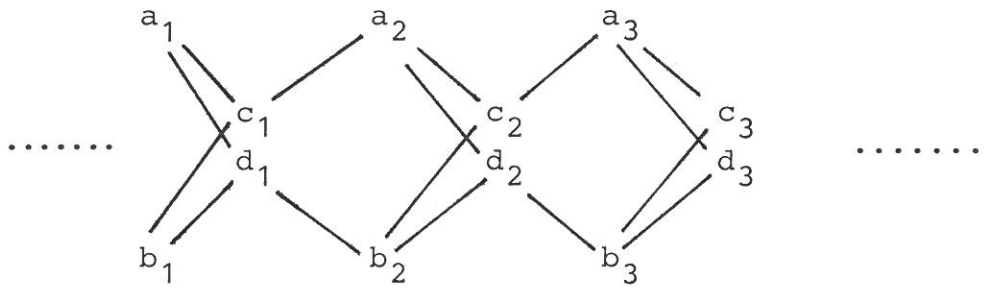
$$M_{19}(D) = (M_{17}(D))$$

The set of configuration sequences can be structured as an event trace

$$M_{18}(D) = \ldots\ a_1\times b_1 \text{ — } c_1\times d_1 \begin{cases} c_1\times b_2 \\ a\times d \end{cases} a\times b \text{ — } c\times d \begin{cases} \ldots \\ \ldots \end{cases}$$

$$M_{20}(D) = (M_{18}(D))$$

One can partition the set of configuration sequences and give an accept-refuse semantics, a past-future semantics and a resumption semantics. Instead of giving these semantics, we give another trace semantics



This trace semantics is natural if one thinks of moves as processes and configurations as ports.

□

5. INDIRECT PRESENTATIONS

Some semantics of programming languages are presented so indirectly that the meaning of programs is never revealed. In algebraic presentations one specifies a relation = between program fragments by equations, and one agrees that fragment p has the same meaning as fragment q when one can derive the equation p=q. In term rewriting presentations one specifies a relation → between program fragments, and one agrees that fragment p has the same meaning as fragment q when both p and q can be rewritten as the same fragment r. If the rewriting system does not have the Church-Rosser property, this definition of "has the same meaning as" does not give an equivalence relation, and there is much confusion.

Example

Consider a programming language with syntax

<term> ::= 0 | S <term> | (<term>+<term>)

The natural semantics has the algebraic presentation

$$(t + 0) = t$$
$$(t + Su) = S(t + u)$$

As an example of programs with the same meaning we can give

(SS0 + SSS0), (SSSS0 + S0), SSSS0

but it is not clear whether the common program meaning is the whole equivalence class, the reduced representative SSSS0, or the abstract number 5.

If we replace = in our two equations by →, we get a term replacement presentation with the Church-Rosser property. The resulting notion of "having the same meaning" is the same as that for the algebraic presentation. But what if we had chosen the term rewriting presentation

$$t \rightarrow (t + 0)$$
$$(t + Su) \rightarrow S(t + u)$$

so we have the rewriting sequences

(1) $(SS0+SSS0) \rightarrow S(SS0+SS0) \rightarrow SS(SS0+S0) \rightarrow SSS(SS0+0)$
(2) $(SSSS0+S0) \rightarrow S(SSSS0+0)$
(3) $SSSSS0 \rightarrow SSS(SS0+0)$
(4) $SSSSS0 \rightarrow S(SSSS0+0)$

Sequences (1) and (3) give "(SS0+SSS0) and SSSSS0 have the same meaning"; sequences (2) and (4) give "(SSSS0+S0) and SSSSS0 have the same meaning"; yet (SS0+SS0) and (SSSS0+S0) do not have the same meaning, because they have no common descendant.

□

The most subtle form of indirect presentation is the axiomatic approach. This presupposes a language L for expressing properties of program fragments, but the logicians have given us the predicate calculus and many other suitable languages, while the computer scientists have suggested algorithmic logic, dynamic logic, temporal logic and much else. Let $\phi(p)$ denote a formula in a language L that expresses a property of a program fragment p. If the language L has a proof theory, then we can use $T \vdash \phi(p)$ for "$\phi(p)$ can be proved from assumptions T". Now we can define a partial order $\leqq$ on program fragments by

$$p \leqq q \ .\equiv. \text{ if } T \vdash \phi(p) \text{ then } T \vdash \phi(q)$$

and we can agree on $p \leqq q \ .\&. \ q \leqq p$ as the criterion for: fragment p has the same meaning as q. If the language L has a model theory, then we can use $M \models \phi(p)$ for "$\phi(p)$ is satisfied in structure M". Now we can define a partial order $\lessdot$ on program fragments by

$$p \lessdot q \ .\equiv. \text{ if } M \models \phi(p) \text{ then } M \models \phi(q)$$

and we can agree on $p \lessdot q \ .\&. \ q \lessdot p$ as the criterion for: fragment p has the same meaning as q. Most languages have both proof and model

theories and no disagreement between the fragment orders $\leqq$ and $\lessdot$ would be welcome. We want the language L to be both <u>sound</u> - $p \leqq q$ implies $p \lessdot q$ - and <u>complete</u> - $p \lessdot q$ implies $p \leqq q$. This use of the traditional terms, sound and complete, is natural because in most languages with a material implication "$\supset$" we have

$$p \leqq q \quad .\equiv. \vdash \phi(p) \supset \phi(q) \quad \text{for all } \phi$$
$$p \lessdot q \quad .\equiv. \models \phi(p) \supset \phi(q) \quad \text{for all } \phi$$

Note that one usually has limits of $\leqq$-chains and $\lessdot$-chains because theories T and structures M form complete lattices. Note also that many semantics M give a partial order on program fragments

$$p \{ q \quad .\equiv. \; M(p) \text{ approximates } M(q)$$

so an axiomatic presentation of the semantics may be available and informative.

<u>Example</u>
Consider the language L with syntax

```
    <term> ::= 0 | S <term> | (<term> + <term>)
<formula> ::= p = <term>
```

A structure of the language consists of a particular natural number M. The formula p=<term> is satisfied in the structure M if and only if M is the value of term when one interprets 0, S and + by the number zero, the successor function and addition respectively. Note that each formula is satisfied in exactly one structure, and we have

$$\langle term_1 \rangle \lessdot \langle term_2 \rangle \; .\equiv. \; p=\langle term_1 \rangle \text{ is satisfied in the same structure as } p=\langle term_2 \rangle$$

Programs have the same meaning in this axiomatic presentation if and only if they have the same meaning in the algebraic presentation.

The language *L* has a proof theory if we agree on the rules

$$\frac{p=(t+0)}{p=t} \qquad \frac{p=t}{p=(t+0)} \qquad \frac{p=(t+Su)}{p=S(t+u)} \qquad \frac{p=S(t+u)}{p=(t+Su)}$$

The formula p=<term> can be proved from the assumptions T if and only if there is a sequence of formulae $\phi_1\ \phi_2\ \ldots\ \phi_n$ such that

$$\phi_1 \in T\ ,\ \phi_n \text{ is } p=\langle term\rangle$$

$$\frac{\phi_i}{\phi_{i+1}} \text{ is a proof rule for } i=1,2,\ldots,n-1$$

One can easily check that the partial order

$$\langle term_1\rangle \leqq \langle term_2\rangle \ .\equiv.\ T \vdash p=\langle term_1\rangle \text{ if and only if } T \vdash p=\langle term_2\rangle$$

is the same as $\langle term_1\rangle \lessdot \langle term_2\rangle$ so the language is sound and complete.

Suppose we add material implication to the language *L* by changing the definition of <formula>

$$\langle formula\rangle ::= p=\langle term\rangle \mid \langle formula\rangle \supset \langle formula\rangle$$

The satisfaction rule for the new formulae is:

$$M \models \phi_1 \supset \phi_2 \ .\equiv.\ M \models \phi_2 \text{ or not } M \models \phi_1$$

and the provability rule for the new formulae is:

$$T \vdash \phi_1 \supset \phi_2 \ .\equiv.\ \{\phi_1\} \vdash \phi_2$$

The partial orders $\lessdot$ and $\leqq$ have not changed because we have the equivalence chains

$$\begin{aligned}
\vdash \phi(p) \supset \phi(q) \quad &.\equiv. \quad M \models \phi(q) \ \underline{\text{or}} \ \underline{\text{not}} \ M \models \phi(p) \\
&.\equiv. \quad \underline{\text{if}} \ M \models \phi(p) \ \underline{\text{then}} \ M \models \phi(q) \\
\vdash \phi(p) \supset \phi(q) \quad &.\equiv. \quad \{\phi(p)\} \vdash \phi(q) \\
&.\equiv. \quad \underline{\text{if}} \ \{\phi(p)\} \vdash \phi(p) \ \underline{\text{then}} \ \{\phi(p)\} \vdash \phi(q) \\
&.\equiv. \quad \underline{\text{if}} \ T \vdash \phi(p) \ \underline{\text{then}} \ T \vdash \phi(q)
\end{aligned}$$

The extended language is still sound and complete, it has not changed the meanings of programs, but it can now express more. The true and provable formula

$$p = (SS0 + SSS0) \supset p = (SSSS0 + S0)$$

expresses the fact that (SS0+SSS0) and (SSSS0+S0) are programs with the same meaning.

□

6. COMPARISON OF SEMANTICS

How can we compare two semantics, two ways of giving meaning to programs? One way of comparing a semantics $S_1 : P \to M_1$ with another semantics $S_2 : P \to M_2$ is to see whether they agree on when two programs have the same meaning - to check whether

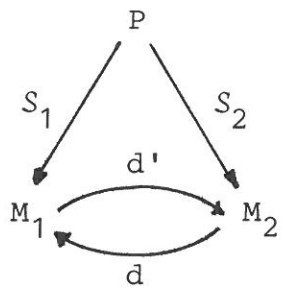
$$p \sim_1 p' \quad .\equiv. \quad S_1(p) = S_1(p')$$

$$p \sim_2 p' \quad .\equiv. \quad S_2(p) = S_2(p')$$

are the same equivalence relation. If these equivalence relations are the same, we say that S_1 and S_2 are <u>transforms</u> of one another. Semantics that are transforms of one another convey the same information about programs because we have

<u>Lemma</u>

Semantics, $S_1 : P \to M_1$ and $S_2 : P \to M_2$, are transforms of one another if and only if we have functions, $d' : M_1 \to M_2$ and $d : M_2 \to M_1$, such that $S_1 = d \circ S_2$ and $S_2 = d' \circ S_1$.



<u>Proof</u>

($\leftarrow$) We have the chain of implications

$$S_1(p) = S_1(p') \ .\supset. \ d' \circ S_1(p) = d' \circ S_1(p') \ .\supset. \ S_2(p) = S_2(p')$$

$$S_2(p) = S_2(p') \ .\supset. \ d \circ S_2(p) = d \circ S_2(p') \ .\supset. \ S_1(p) = S_2(p')$$

(→) Choose any m_2 in M_1 and define $d:M_2 \to M_1$ by

$$d(m_2) = \underline{\text{if}}\ m_2 = S_2(p)\ \underline{\text{then}}\ S_1(p)\ \underline{\text{else}}\ m_1$$

This function is well defined because $S_2(p) = S_2(p')$ implies $S_1(p) = S_1(p')$. Clearly we have $S_1 = d \circ S_2$ and a similar argument gives $S_2 = d \circ S_1$ for a well defined $d':M_1 \to M_2$.

□

Definition

A set M_1 can be embedded in a set M_2, and we write $M_1 \text{—(} M_2$ if and only if there are functions $d':M_1 \to M_2$ and $d:M_2 \to M_1$ such that $d \circ d'(m_1) = m_1$ for all $m \in M_1$.

□

Embedding Theorem

If a set M_1 can be embedded in a set M_2, then any semantics $S_1:P \to M_1$ has a transform $S_2:P \to M_2$.

Proof

Let $d':M_1 \to M_2$ and $d:M_2 \to M_1$ be such that $d \circ d'$ is the identity on M_1. Define $S_2:P \to M_2$ by $S_2 = d' \circ S_1$. Because $d \circ S_2 = d \circ d' \circ S_1 = S_1$, the lemma ensures that S_1 and S_2 are transforms of one another.

□

6.1 Comparison of Relational Semantics

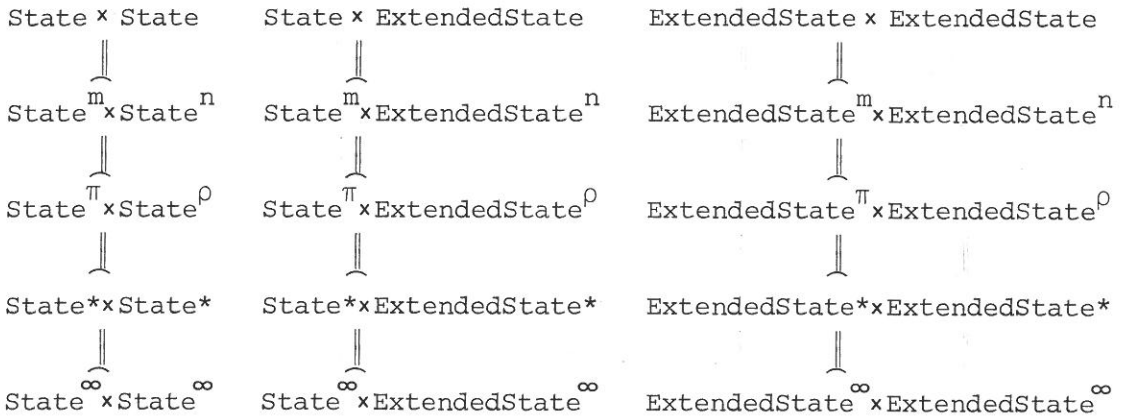
In this section the embedding theorem is used to give information about transformations between the relational semantics in section 2.1. The embedding "State —(ExtendedState" gives

$$\text{State} \times \text{State} \text{ ==(} \text{State} \times \text{ExtendedState} \text{ ==(} \text{ExtendedState} \times \text{ExtendedState}$$

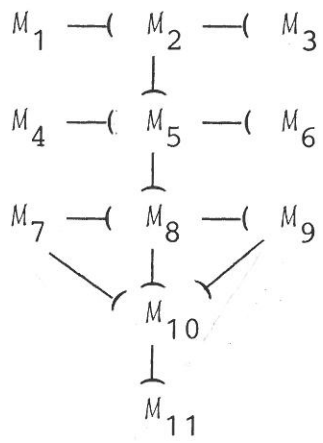
$$\text{State}^m \times \text{State}^n \text{ ==(} \text{State}^m \times \text{ExtendedState}^n \text{ ==(} \text{ExtendedState}^m \times \text{ExtendedState}^n$$

$$\text{State}^\pi \times \text{State}^\rho \text{ ==(} \text{State}^\pi \times \text{ExtendedState}^\rho \text{ ==(} \text{ExtendedState}^\pi \times \text{ExtendedState}^\rho$$

where S ==(S' abbreviates: Subsets of S —(Subsets of S'.
The embeddings "S —(S^m —(S^π —(S^* —(S^∞" give



For the semantics M_1-M_{11} in section 2.1 we have



These transformations of relational semantics can be useful when different semantics are appropriate for different parts of a programming language - M_1 might be appropriate for simple language features, while M_{10} might be suitable for more complicated features.

6.2 Comparison of Functional Semantics

In this section the embedding theorem is used to give transformations between functional semantics. First we note that power domain semantics and relational semantics M_1 can be transformed into one another because

$$s' \in f(s) \ .\equiv. \ R(s,s')$$

gives the mutual embedding

$$S \to \text{Subset of } S \ )\!\!-\!\!(\ \text{Subset of } S\times S.$$

Next we note that a relational semantics M_1 can be transformed into a forward and a backward predicate transformer semantics because

$$R(s,s') \ .\equiv. \ s'\in F(\{s\}) \qquad s'\in F(P) \ .\equiv. \ (\exists s\in P)R(s,s')$$

$$R(s,s') \ .\equiv. \ s\in B(\{s'\}) \qquad s \in B(Q) \ .\equiv. \ (\exists s'\in Q)R(s,s')$$

both give embeddings for

$$\text{Subset of } S\times S \ —\!(\ \text{Subset of } S \to \text{Subset of } S$$

Warning
This transformation to a backward predicate transformer semantics should be used with care, because a more natural semantics is given by $s\in B(Q) \ .\equiv. \ (\forall s')(R(s,s') \supset s'\in Q)$.

Predicate transformer semantics can be transformed into continuation semantics because taking A = (true,false) gives an embedding

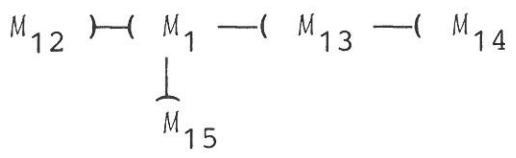
$$\text{Subsets of } S \ —\!(\ S \to A$$

To get a result about oracular semantics, we need an "undefined" element s_0 in order to define the embeddings

$$f(s,n) = \underline{\text{if}} \ \{s' \mid r(s,s')\} \text{ has fewer than n elements}$$
$$\underline{\text{then}} \ s_0 \ \underline{\text{else}} \ \text{n-th element of Q}$$

for Subset of $S\times S$ —($S\times$Oracle $\to S$.

Let us summarise our results for the functional semantics in section 2.2 in a schema



6.3 Comparison of Set Semantics

In this section the embedding theorem is used to give information about transformations between the set semantics in section 2.3. The embedding "Event Sequence —(Event Tree —(Event Trace" gives

Event Sequence =(Event Tree =(Event Trace.

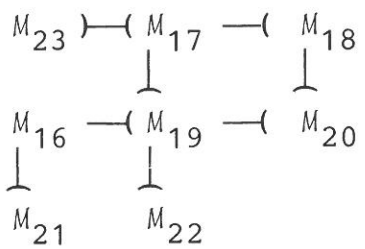
We have the trivial embeddings

Event Sequence —(Event Sequence × Accept × Refuse
Event Tree —(Past × Future

and [B2] gives a transformation between most Accept-Refuse semantics and Event Trace semantics. A mutual embedding for Event Trees and Resumption semantics is given by identifying resumptions with tree vertices and agreeing on:

(e,r') ∈ r .≡. edge labelled e from r to r'

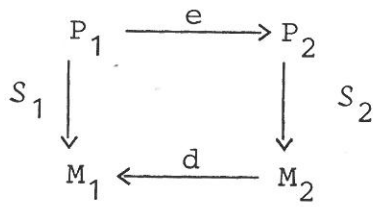
For the semantics M_{16}-M_{23} in section 2.3 we have



From [W1] one can infer that most event trace semantics can be transformed into event tree semantics.

6.4 Translations and Implementations of Semantics

It is natural to say that a semantics $S_2:P_2\to M_2$ <u>implements</u> a semantics $S_1:P_1\to M_1$ when S_1 can be recovered from S_2.



From the diagram we see that a suitable criterion is: $S_1 = d \bullet S_2 \bullet e$ for some $e:P_1\to P_2$ and $d:M_2\to M_1$. This definition is more general than most definitions of implementation of data types, but slightly less general than that in [M1]. But why have we introduced the idea of one semantics implementing another? The answer is that the natural semantics for part of a complex programming language may be very different from the natural semantics for another part of the language; implementing one semantics by another is a useful way of merging various semantics.

<u>Example</u>

The natural definition of the expression part of most languages is given by a semantics $S_1:E\to(S\to V)$, but this semantics can be implemented by a semantics $S_2:E\to(S\to V\times S)$ to allow for expressions with side effects.

□

<u>Example</u>

The formal definition of sequential ADA [ ] is given by a continuation semantics $S_1:P\to(C\to C)$. Because $C = S\to A$ and the domain A is never specified, the continuation semantics can be implemented by a semantics $S_2:P\to(S\to S)$. It is much easier to define the parallel features of ADA by extending S_2, rather than S_1.

□

Suppose semantics, S_1 and S_2, are transforms of one another. By the lemma in section 6.1, S_1 implements S_2 and S_2 implements S_1. Another particular case of semantic implementation is given by the canonical presentations in section 4: a semantics S_1:P→M is defined as the composition of a translation e:P→Diagrams and a semantics S_2:Diagrams→M.

This report is just a small beginning in the study of comparative semantics as further investigation of transforms, translations and implementations should illuminate practical problems just as much as the study of abstract data types has illuminated the practical problems of modularity in programming and specification languages.

7. THE MINILANGUAGE BED

In this section we introduce a minilanguage BED as an aid in comparing the many ways of giving meanings to programs. Real programming languages are not suitable for comparing semantics, because the advantages and disadvantages of the various semantic methods are obscured by much irrelevant detail. The published semantics of more compact languages like CSP give a better basis for comparing various ways of giving meaning to programs, but every existing compact minilanguage omits some important but semantically challenging language feature. In our minilanguage (test-) BED we focus on such semantically challenging features and ignore well understood features like block structure, scoping, expressions and their side effects.

The syntax of the minilanguage BED is given in figure 7.1, while the rest of this section is devoted to the description of the formal semantics in figures 7.2, 7.3, 7.9.

```
<command>::=<identifier>:=<identifier>|pause|delay<command>.
                          |loop <command>.
                          |exit when <command>.
                          |<command>; <command>
                          |either <command> or <command>.
                          |both <command> and <command>.
                          |call <identifier>
                          |accept <identifier> do <command>.
```

Fig. 7.1 BED syntax

In our informal description of the intended meaning of BED commands we assume an unlimited supply of identifiers and a map STORE which takes

- each identifier into a value and an action
- the reserved identifiers, external and internal, into the improper value and improper action.

The assignment command "ℓ:=r" has the usual meaning when ℓ and r are not reserved identifiers; the value of ℓ is changed to the value of r. The assignment "ℓ:=external" inputs some value to ℓ from the context; the assignment "external:=r" outputs the value of r to the context; the assignment "ℓ:=internal" gives a random value to ℓ; the assignment "internal:=ℓ" gives the value of ℓ to all unreserved identifiers.

The execution of a BED command C begins with a control point before the command; during the execution there may be zero, one or many control points before subcommands of C; if the execution of C terminated, it ends with a control point after C. Every execution of an assignment command consists of one step in which a control point moves over the command. The execution of a pause command may have several steps in which a control point rests before the command. Every execution of the command "delay C." is like an execution of C but it may have several steps in which a control point rests before a subcommand of C.

Sometimes the execution of a BED command is futile in that it makes no change in the values and actions associated with identifiers (pause is always futile, and "ℓ:=r" is futile when ℓ and r have the same value). This possibility allows us to avoid conditions in BED; we can define the meaning of exit when C. as:

leave the enclosing loop
if and only if executing C is futile.

The command "loop C end." has the usual (ADA) meaning: repeat the execution of C until exit is possible.

Let us turn to the BED distinction between sequencing, nondeterminism and parallelism. The executions of "$C_1;C_2$" are the executions of C_1 concatenated with the executions of C_2. The executions of "either C_1 or C_2." are the executions of C_1, together with the executions of C_2. The executions of "both C_1 and C_2." are the executions of C_1 synchronized with the execu-

tions of C_2 by a matching rule. The BED matching rule is like the CSP and ADA rendezvous rules: executions can be interleaved except that an entry call must be synchronized with an accept command for that entry.

The accept command is the BED way of declaring an entry or procedure; the meaning of "accept p do C end" is: assign the command C as the action of identifier p. The call command is the BED way of activating an entry or procedure; the meaning of "call p" is: obey the action associated with the identifier p. The BED distinction between procedures and entries is syntactic; the command "call p" is a procedure call if it is nested within a block containing an accept p command, otherwise it is an entry call. The command "both C_1 and C_2 end" in some block B introduces C_1 and C_2 as subblocks nested within B.

Examples

In the appendix we give a relational, function and set meaning to the following BED commands

(1) delay ℓ:=r.

(2) both ℓ:=r and pause.

(3) either ℓ:=r or ℓ:=ℓ.

(4) exit when either ℓ:=r or ℓ:=ℓ..

(5) loop either ℓ:=r or ℓ:=ℓ..

(6) loop exit when either ℓ:=r or ℓ:=ℓ...

(7) accept Q do ℓ:=r.; call Q

(8) either accept Q do ℓ:=r. or accept Q do ℓ:=ℓ..; call Q

(9) both accept Q do ℓ:=r. and accept Q do ℓ:=ℓ ..; call Q

(10) both accept Q do ℓ:=r. and call Q.

(11) both either accept Q do ℓ:=r. or accept Q do ℓ:=ℓ.. and call Q.

(12) both both accept Q do ℓ:=r. and accept Q do ℓ:=ℓ.. and call Q.

(13) accept P do pause.;
both loop call Q; exit when call P..
and accept Q do external:=internal..

Commands (4)-(6) illustrate exit and loop, commands (7)-(8) illustrate procedure calls, and commands (10)-(12) illustrate entry calls. Because call P in command (13) is accepted by an outer block command, it is a procedure call; because call Q can only be accepted by a parallel inner block, it is an entry call.

Do not worry if the informal semantics seems vague and imprecise, in spite of the fact that it has been formulated with great care. The main aim of formal semantics is to replace vagueness with precision. This paper will not have been in vain if it convinces the reader that there are many equivalent formal ways of presenting the semantics of a programming language, once the vagueness of an informal semantics has been resolved in the mind of the language designer.

7.1 Relational Semantics for BED

In this section we give an M_1-semantics to BED. Since we want the meaning of a BED command to be a relation on states, the first step is to define the set of states. The natural choice is the set of functions from identifiers to value-action pairs:

$$\text{States} = \text{Identifiers} \to \text{Values} \times \text{Actions}$$

Having made this choice, we have the functions

$$\begin{aligned}
&\text{Fetch Value}(r,s) = \text{first component of } s(r)\\
&\text{Fetch Action}(r,s) = \text{second component of } s(r)\\
&\text{Store Value}(n,\ell,s) = \lambda i.\ \underline{\text{if}}\ i \neq \ell\ \underline{\text{then}}\ s(i)\\
&\qquad\qquad\qquad\qquad \underline{\text{else}}\ \langle n, \text{Fetch Action}(\ell,s)\rangle\\
&\text{Store Action}(c,\ell,s) = \lambda i.\ \underline{\text{if}}\ i \neq \ell\ \underline{\text{then}}\ s(i)\\
&\qquad\qquad\qquad\qquad \underline{\text{else}}\ \langle \text{Fetch Value}(\ell,s), c\rangle
\end{aligned}$$

These functions can be converted to relations in a way that captures the special role of the reserved identifiers

$r,s \xrightarrow{FV} n \quad .\equiv.$ (Fetch Value$(r,s)=n) \vee (r=\underline{external}) \vee (r=\underline{internal})$

$r,s \xrightarrow{FA} C \quad .\equiv.$ (Fetch Action$(r,s)=C$)

$n,\ell,s \xrightarrow{SV} s' \quad .\equiv.$ $(\ell=\underline{external}$ & $s=s')$
$\vee(\ell=\underline{internal}$ & Spread$(n,s)=s')$
$\vee(\ell\neq\underline{external}$ & $\ell\neq\underline{internal}$ & Store Value$(n,\ell,s)=s')$

$C,\ell,s \xrightarrow{SA} s' \quad .\equiv.$ (Store Action$(C,\ell,s)=s')$

where Spread$(n,s) = \lambda i.$ $\underline{if}(i=\underline{external} \vee i=\underline{internal})$ $\underline{then}$ $s(i)$
$\underline{else}$ $\langle n,$Fetch Action$(i,s)\rangle$

Now we can give the meaning of an assignment command by the proof rule

$$\frac{r,s \xrightarrow{FV} n \ \ldots\ n,\ell,s \xrightarrow{SV} s'}{\ell := r,s \longrightarrow s'}$$

where ... separates the premisses of the rule.

There are no premisses in the two proof rules for the BED pause command:

$$\frac{}{\underline{pause},s \longrightarrow s} \qquad \frac{}{\underline{pause},s \longrightarrow \underline{pause},s}$$

There are three proof rules for the delay command

$$\frac{C,s\rightarrow s'}{\underline{delay}\ C.,s\rightarrow s'} \qquad \frac{C,s\rightarrow C',s'}{\underline{delay}\ C.,s\rightarrow \underline{delay}\ C'.,s'} \qquad \frac{}{\underline{delay}\ C.,s\rightarrow \underline{delay}\ C.,s}$$

but a useful convention allows the first two rules to be combined

$$\frac{C,s\rightarrow[C',]s'}{\underline{delay}\ C.,s\rightarrow[\underline{delay}\ C'.,]s'}$$

This convention can be used to combine two of the proof rules for the BED loop command:

$$\frac{C,s\rightarrow[C',]s'}{\underline{\text{loop}}\ C.,s\rightarrow[C';]\ \underline{\text{loop}}\ C.,s'}$$

The other two proof rules for loop commands

$$\frac{}{\underline{\text{failure}};\ [C';]\ \underline{\text{loop}}\ C.,s\rightarrow s}$$

use a pseudo command "failure" that is introduced by the proof rules for exit commands

$$\frac{s \xrightarrow{C} s' \ \ldots\ s=s'}{\underline{\text{exit}}\ \underline{\text{when}}\ C.,s\rightarrow\underline{\text{failure}},s} \qquad \frac{s \xrightarrow{C} s' \ldots s\neq s'}{\underline{\text{exit}}\ \underline{\text{when}}\ C.,s\rightarrow s}$$

These two rules use the state relation $\xrightarrow{C}$ that is the meaning of a command C; the proof rules we are gathering define this relation recursively.

Let us turn to the semantics of sequencing nondeterminism and parallelism in BED. The proof rules for the sequencing command are easy:

$$\frac{C1,s\rightarrow[C',]s'}{C1;C2,s\rightarrow[C';]C2,s'}$$

The proof rules for the choice command are also straightforward:

$$\frac{C1,s\rightarrow[C',]s'}{\underline{\text{either}}\ C1\ \underline{\text{or}}\ C2.,s\rightarrow[C',]s'} \qquad \frac{C2,s\rightarrow[C',]s'}{\underline{\text{either}}\ C1\ \underline{\text{or}}\ C2.,s\rightarrow[C',]s'}$$

Parallelism is more tricky, and we need syntactic distinctions like our earlier distinction between procedures and entries. One can define a syntactic predicate, Free(C), for "the first subcommand of C is not an entry call or accept" and introduce the proof rules

$$\frac{\text{C1,s}\rightarrow\text{[C',]s' ... Free(C1)}}{\underline{\text{both}}\ \text{C1}\ \underline{\text{and}}\ \text{C2.,s}\rightarrow\text{[}\underline{\text{both}}\ \text{C'}\ \underline{\text{and}}\text{]C2[.],s'}}$$

$$\frac{\text{C2,s}\rightarrow\text{[C',]s' ... Free(C2)}}{\underline{\text{both}}\ \text{C1}\ \underline{\text{and}}\ \text{C2.,s}\rightarrow\text{[}\underline{\text{both}}\text{]}\ \text{C1}\ \text{[}\underline{\text{and}}\ \text{C'.],s'}}$$

One can also define a syntactic predicate, Match(C_1,C_2) for "the first subcommand of C_1 is an entry call and the first subcommand of C_2 is an accept of the same entry". This predicate is used in the rules

$$\frac{\text{C1,s}\rightarrow\text{[C',]s' ... C2,s'}\rightarrow\text{C",s" ... Match(C2,C1)}}{\underline{\text{both}}\ \text{C1}\ \underline{\text{and}}\ \text{C2.,s}\rightarrow\text{[}\underline{\text{both}}\ \text{C'}\ \underline{\text{and}}\text{]C"[ ],s"}}$$

$$\frac{\text{C1,s}\rightarrow\text{[C',]s' ... C2,s'}\rightarrow\text{s" ... Match(C2,C1)}}{\underline{\text{both}}\ \text{C1}\ \underline{\text{and}}\ \text{C2.,s}\rightarrow\text{[C',]s"}}$$

$$\frac{\text{C1,s'}\rightarrow\text{C",s" ... C2,s}\rightarrow\text{[C',]s' ... Match(C1,C2)}}{\underline{\text{both}}\ \text{C1}\ \underline{\text{and}}\ \text{C2.,s}\rightarrow\text{[}\underline{\text{both}}\ \text{C"}\ \underline{\text{and}}\ \text{C'.],s"}}$$

$$\frac{\text{C1,s'}\rightarrow\text{s" ... C2,s}\rightarrow\text{[C',]s' ... Match(C1,C2)}}{\underline{\text{both}}\ \text{C1}\ \underline{\text{and}}\ \text{C2.,s}\rightarrow\text{[C',]s"}}$$

Note how the rules force entries to be accepted before calls, while also enforcing the call and the acceptance to take place in one single rendezvous step. Because of this subtlety the proof rules for the remaining BED commands are surprisingly simple

$$\frac{\text{p,s} \xrightarrow{\text{FA}} \text{C ... s} \xrightarrow{\text{C}} \text{s'}}{\text{call p,s}\rightarrow\text{s'}} \qquad \frac{\text{C,p,s} \xrightarrow{\text{SA}} \text{s'}}{\underline{\text{accept}}\ \text{p}\ \underline{\text{do}}\ \text{C.,s}\rightarrow\text{s'}}$$

Example

Consider the three block BED command (13) in the last section. Assume s_0,s_1,s_2,s_3 satisfy:

(1) $\underline{\text{pause}},P,s_0 \xrightarrow{SA} s_1$

(2) $\underline{\text{external}} := \underline{\text{internal}},Q,s_1 \xrightarrow{SA} s_2$

(3) $\underline{\text{external}} := \underline{\text{internal}},s_2 \longrightarrow s_3$

(4) $\underline{\text{pause}},s_3 \longrightarrow s_3$

Appropriate proof rules give

(5) $\underline{\text{accept}}\ P\ \underline{\text{do}}\ \underline{\text{pause}}.,s_0 \longrightarrow s_1$ -- from (1)

(6) $\underline{\text{accept}}\ Q\ \underline{\text{do}}\ \underline{\text{external}} := \underline{\text{internal}}.,s_1 \longrightarrow s_2$ -- from (2)

(7) $\underline{\text{call}}\ Q;\ s_2 \longrightarrow s_3$ -- from (3)

(8) $\underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.,s_3 \longrightarrow \underline{\text{failure}},s_3$ -- from (4)

(9) $\underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.,s_2 \longrightarrow \underline{\text{exit}}\ \underline{\text{when}}\ P.,s_3$ -- from (7)

(10) $\underline{\text{loop}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.,s_2$

$\rightarrow \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.;\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..,s_3$

-- from (9)

(11) $\underline{\text{both}}\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..$

$\underline{\text{and}}\ \underline{\text{accept}}\ Q\ \underline{\text{do}}\ \underline{\text{external}} := \underline{\text{internal}}\ ..,s_1$

$\rightarrow \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.;\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..,s_3$

-- from (6,10)

$\rightarrow \underline{\text{failure}};\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..,s_3$ -- from (8)

$\rightarrow s_3$

Corresponding computation has four steps - (5) and the three steps in (11) - and the proof shows that s_0 and s_3 are in the relation that is the meaning of the BED program.

Presumably one is only interested in the input-output behaviour of the program, but this can be extracted from the assignment steps which use the reserved identifier external. The only such assignment in our computation is explicit in (3) and implicit in (7,9,10,11), so our program has no input and outputs a random value.

□

The formal relational semantics for all of BED is given in Figure 7.2. Extracting the input-output behaviour of a program from its relational meaning is an obscure process. There is another problem, which we have glossed over - the syntactic predicates, Free and Match, use the notion of "first subcommand", but what is the first subcommand of "both C_1 and C_2." and "either C_1 or C_2."?

$$\frac{r,s \xrightarrow{FV} n \ldots n,\ell,s \xrightarrow{SV} s'}{\ell := r,s \to s'}$$

$$\frac{}{\text{pause},s \to [\text{pause},]s}$$

$$\frac{C,s \to [C',]s'}{\text{delay } C.,s \to [\text{delay } C'.,]s'} \qquad \frac{}{\text{delay } C.,s \to \text{delay } C.,s}$$

$$\frac{C,s \to [C',]s'}{\text{loop } C.,s \to [C';]\text{loop } C.,s'} \qquad \frac{}{\text{failure};[C;]\text{loop } C.,s \to s}$$

$$\frac{s \xrightarrow{C} s' \ldots s=s'}{\text{exit when } C.,s \to \text{failure},s} \qquad \frac{s \xrightarrow{C} s' \ldots s \neq s'}{\text{exit when } C.,s \to s'}$$

$$\frac{C1,s \to [C',]s'}{C1;C2,s \to [C';]C2,s'}$$

$$\frac{C1,s \to [C',]s'}{\text{either } C1 \text{ or } C2.,s \to [C',]s'} \qquad \frac{C2.s \to [C',]s'}{\text{either } C1 \text{ or } C2.,s \to [C',]s'}$$

Rules for "both C1 and C2 end" are in the text.

$$\frac{p,s \xrightarrow{FA} c \ldots s \xrightarrow{C} s'}{\text{call } p,s \to s'}$$

$$\frac{C,p,s \longrightarrow s'}{\text{accept } p \text{ do } C.,s \to s'}$$

Fig. 7.2 Relational Semantics for BED.

7.2 Functional Semantics for BED

In this section we shall describe an M_{15}-semantics that gives an oracular function as the meaning of every BED command. These function meanings belong to

$$\text{Oracle} \times \text{State} \to \text{Oracle} \times \text{State}$$

where Oracle = (T,F)* and State was defined in section 7.1. The oracular function for the BED assignment command could be presented as

$$[\![\ell := r]\!](w,s) = (w,s') \text{ where } r,s \xrightarrow{FV} n \text{ and } n,\ell,s \xrightarrow{SV} s'$$

but we prefer a neater way of presenting complicated functions [S]. The basic idea is that functions operate on argument streams and complicated functions can be built from simpler functions using combinators like

$$f \mid g(y_n \ldots y_{\ell+1}, x_k \ldots x_1) = (.. z_n \ldots z_1) \text{ where } f(x_k \ldots x_1) = (y_\ell \ldots y_1) \text{ and } g(y_n \ldots y_1) = (z_n \ldots z_1)$$

The functional meaning of the assignment command can now be presented by

$$[\![\ell := r]\!] = FV\ r \mid SV\ \ell$$

where

$$FV\ r\ (w\ s) = \underline{\text{if}}\ (r=\underline{\text{external}} \vee r=\underline{\text{internal}})\ \underline{\text{then}}\ \underline{\text{let}}\ w = T^m\ F\ w'\ \underline{\text{in}}\ (w',s,m)\ \underline{\text{else}}\ \underline{\text{let}}\ m = \text{unique } n \text{ such that } r,s \xrightarrow{FV} n\ \underline{\text{in}}\ (w,s,m)$$

and SV ℓ (s,n) = unique s' such that $n,\ell,s \xrightarrow{SV} s'$.

The non-determinism in the BED pause command is resolved by consulting the oracle:

$$[\![\underline{\text{pause}}]\!] = \text{PEEK}(\text{ID},[\![\underline{\text{pause}}]\!])$$

where

$$\text{PEEK}(f,g)(Tw,s) = f(w,s)$$
$$\text{PEEK}(f,g)(Fw,s) = g(w,s)$$

and ID(w,s) = (w,s). The combinator PEEK is also used in the function for the loop command

$$[\![\underline{\text{loop}}\ C.]\!] = \text{ENTER} \mid [\![c]\!] \mid \text{LEAVE} \mid \text{PEEK}(\text{ID},[\![\underline{\text{loop}}\ C\ \underline{\text{end}}]\!]$$

where

$$\text{LEAVE}(T,s,w,s') = (Tw,s)$$
$$\text{LEAVE}(F,s,w,s') = (Fw,s')$$

and $$\text{ENTER}(w,s) = (F,s,w,s).$$

The meaning of the loop command is complicated by the fact that exit commands cannot force exit from the loop. The functional meaning of an exit command is

$$[\![\underline{\text{exit}}\ \underline{\text{when}}\ C.]\!] = \text{ENTER} \mid [\![c]\!] \mid \text{FUTILE}$$

where

$$\text{FUTILE}(b,s,F,s',w,s'') = \underline{\text{if}}\ s' \neq s''\ \underline{\text{then}}\ (b,s,w,s'')$$
$$\underline{\text{else}}\ \underline{\text{if}}\ b=F\ \underline{\text{then}}\ (T,s'',w,s'')$$
$$\underline{\text{else}}\ (T,s,w,s'')$$

Note how close our functional presentations are to the code a BED compiler might generate.

Let us now turn to sequencing, nondeterminism and parallelism in BED. Sequencing is easy

$$[\![C1;C2]\!] = [\![C1]\!] \; [\![C2]\!]$$

and choice is also straightforward

$$[\![\underline{\text{either}}\ C1\ \underline{\text{or}}\ C2.]\!] = \text{PEEK}([\![C1]\!],[\![C2]\!])$$

Parallelism is much more tricky, because commands have to be divided into the "first subcommand" and the "rest".

Consider the command "both C1 and C2." when C1 satisfies the syntactic predicate Free. In this situation the appropriate function is

$$f1 = \underline{\text{if}}\ \text{Free}(c1)\ \underline{\text{then}}\ [\![\text{First}(C1)]\!]\,|\,[\![\text{Rest1}(\underline{\text{both}}\ C1\ \underline{\text{and}}\ C2.)]\!]\ \underline{\text{else}}\ \text{UNDEF}$$

where UNDEF is the never defined function. The function

$$f2 = \underline{\text{if}}\ \text{Free}(C2)\ \underline{\text{then}}\ [\![\text{First}(C2)]\!]\,|\,[\![\text{Rest2}(\underline{\text{both}}\ C1\ \underline{\text{and}}\ C2.)]\!]\ \underline{\text{else}}\ \text{UNDEF}$$

is appropriate for the situation when C2 satisfies the syntactic predicate Free. If C1 and C2 satisfy the syntactic predicate Match, then the function

$$\begin{aligned} g = {} & \underline{\text{if}}\ \text{Match}(C2,C1)\ \underline{\text{then}}\ [\![\text{First}(C1)]\!]\,|\,[\![\text{First}(C2)]\!]\,|\,[\![\text{Rest}(\underline{\text{both}}\ C1\ \underline{\text{and}}\ C2.)]\!] \\ & \underline{\text{else if}}\ \text{MATCH}(C1,C2)\ \underline{\text{then}} \\ & \qquad [\![\text{First}(C2)]\!]\,|\,[\![\text{First}(C1)]\!]\,|\,[\![\text{Rest}(\underline{\text{both}}\ C1\ \underline{\text{and}}\ C2.)]\!] \\ & \underline{\text{else}}\ \text{UNDEF} \end{aligned}$$

is part of the functional meaning of "both C1 and C2.". If we consult the oracle to choose between our three functions we get

$$[\![\underline{\text{both}}\ C1\ \underline{\text{and}}\ C2.]\!] = \text{PEEK}(g,\text{PEEK}(f1,f2))$$

This technique gives the functional meaning of the BED delay command as

$$[\![\underline{\text{delay}}\ C.]\!] = \text{PEEK}([\![\text{First}(C)]\!]\,|\,[\![\text{Rest}(\underline{\text{delay}}\ C.)]\!],[\![(\underline{\text{delay}}\ C.]\!])$$

The definition of the syntactic predicates - First, Rest1, Rest2, Rest - is just as problematic as the definition of the predicates Free and Match.

The functional meaning of the BED call and accept commands is given by

$$[\![\underline{\text{call}}\ P]\!] = FA\ P$$

$$[\![\underline{\text{accept}}\ P\ \underline{\text{do}}\ C.]\!] = SA\ P\ C$$

where

$$FA\ p(w,s) = \underline{\text{let}}\ C' = \text{unique } C \text{ such that } p,s \xrightarrow{FA} C\ \underline{\text{in}}\ [\![C']\!](w,s)$$

$$SA\ p\ C(w,s) = \underline{\text{let}}\ s' = \text{unique } S'' \text{ such that } C,p,s \xrightarrow{SA} s''\ \underline{\text{in}}\ (w,s')$$

Example

Consider the three block BED command (13) in section 7. For the first command we have the function

$$[\![\underline{\text{accept}}\ p\ \underline{\text{do}}\ \underline{\text{pause}}]\!] = SA\ p\ \underline{\text{pause}}$$

For the first inner block we have the function

$$f = [\![\underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.]\!]$$

$$= ENTER\ |\ [\![\underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.]\!]\ |\ LEAVE\ |\ PEEK(ID,f)$$

$$= ENTER\ |\ [\![\underline{\text{call}}\ Q]\!]\ |\ [\![\underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.]\!]\ |\ LEAVE\ |\ PEEK(ID,f)$$

$$= ENTER\ |\ FA\ Q\ |\ ENTER\ |\ [\![\underline{\text{call}}\ P]\!]\ |\ FUTILE\ |\ LEAVE\ |\ PEEK(ID,f)$$

$$= ENTER\ |\ FA\ Q\ |\ ENTER\ |\ FA\ P\ |\ FUTILE\ |\ LEAVE\ |\ PEEK(ID,f)$$

For the second inner block we have the function

$$g = [\![\underline{\text{accept}}\ Q\ \underline{\text{do}}\ \underline{\text{external}} := \underline{\text{internal}}.]\!]$$

$$= SA\ Q\ \underline{\text{external}} := \underline{\text{internal}}$$

Because the inner blocks are executed in parallel, complications arise. We need the functions

$$f_1 = \text{ENTER} \mid \text{SA Q external := internal} \mid \text{FA Q}$$

$$f_2 = \text{ENTER} \mid \text{FA P} \mid \text{FUTILE} \mid \text{LEAVE} \mid \text{PEEK(ID,f)}$$

in order to express the function f_3 for the parallel command

$$f_3 = \text{PEEK}(f_1 \mid f_2, \text{PEEK(UNDEF,UNDEF)})$$

The meaning of the whole program is the function

$$\text{SA p } \underline{\text{pause}} \mid f_3$$

□

The formal functional semantics for all of BED is given in figure 7.3. It has the same disadvantages as the relational semantics in figure 6:

- intricate syntactic predicates and functions are left undefined
- extracting the input-output behaviour of a program from its functional meaning is an obscure process.

```
⟦ℓ:=r⟧ = FV r | SV ℓ
⟦pause⟧ = PEEK(ID,⟦pause⟧)
⟦delay C.⟧ = PEEK(⟦First(C)⟧|⟦Rest(delay C.)⟧,⟦delay C.⟧)
⟦loop C.⟧ = ENTER|⟦C⟧|LEAVE | PEEK(ID,⟦loop C end⟧)
⟦exit when C.⟧ = ENTER|⟦C⟧ | FUTILE
⟦C1;C2⟧ = ⟦C1⟧ | ⟦C2⟧
⟦either C1 or C2.⟧ = PEEK(⟦C1⟧,⟦C2⟧)
⟦both C1 and C2.⟧ = function in text
⟦call p⟧ = FA p
⟦accept p do C.⟧ = SA p C
```

Fig. 7.3 BED Functional Semantics.

7.3 Set Semantics for BED

In this section we shall describe an M_{18}-semantics that gives a labelled event system as the meaning of every BED command. A labelled event system consists of a set E, a function $\ell: E \to L$ and a family F of subsets of E. The subsets of E in F are called configurations, and they are partially ordered by set inclusion. We shall follow [W1] and present labelled event systems by covering diagrams with an edge from configuration γ to γ', if and only if

$$\gamma'' \in F \;\&\; \gamma \subset \gamma'' \subset \gamma' . \to . \; \gamma = \gamma'' \vee \gamma'' = \gamma'$$

The labelled event systems for the BED commands "ℓ:=r", "pause", "call P", and "accept P do C" are given in figure 7.4. We see that the configurations of the system are given by labelling the empty configuration with ϕ and the covering edge from γ to γ' by the labels of the events in the set $\gamma' - \gamma$.

Events	Labels	Configurations	Covering diagram
1	ℓ:=r	ϕ, {1}	$\phi \xrightarrow{\ell:=r} \cdot$
1,2,3	τ	ϕ, {1},{2}	$\phi \xrightarrow{\tau} \xrightarrow{\tau} \cdot \ldots$
1	call P	ϕ, {1}	$\phi \xrightarrow{\text{call P}} \cdot$
1	accept P do C	ϕ, {1}	$\phi \xrightarrow{\text{accept P do C}} \cdot$

Fig. 7.4 Event System for four BED Commands.

The meaning of the command "either C1 or C2." is the sum of the labelled event systems for C1 and C2. The event system $\langle E_1, F_1, \ell_1 \rangle + \langle E_2, F_2, \ell_2 \rangle$ has $E_1 \cup E_2$ as its set of events, $F_1 \cup F_2$ as its set of configurations, and $\ell(e)$ = if e in E_1 then $\ell_1(e)$ else $\ell_2(e)$ as its labelling function. Note that no C1-event is a C2-event, so this definition of + is adequate and we avoid the minor complication of having to separate E_1 and E_2 when they are disjoint.

The meaning of the command "both C1 and C2." is the product of the labelled event systems for C1 and C2. Suppose the event system $\langle E_1, F_1, \ell_1 \rangle \odot \langle E_2, F_2, \ell_2 \rangle$ has $E_1 \cup E_2$ as its set of events, $\{\gamma_1 \cup \gamma_2 \mid \gamma_1 \in F_1, \gamma_2 \in F_2\}$ as its set of configurations, and $\ell(e)$ = if e in E_1 then $\ell_1(e)$ else $\ell_2(e)$ as its labelling function. If we accept the doctrine "events of concurrent processes can be interleaved freely", the combinator $\odot$ can be used for the product of event systems. In figure 7.5 we show how the combinator o gives the meaning of the BED command "delay C" as the product of the event systems for C and pause. The reason why "delay C" and "both C and pause." have the same meaning is that there is no possibility of a rendezvous between C and pause. Suppose we define

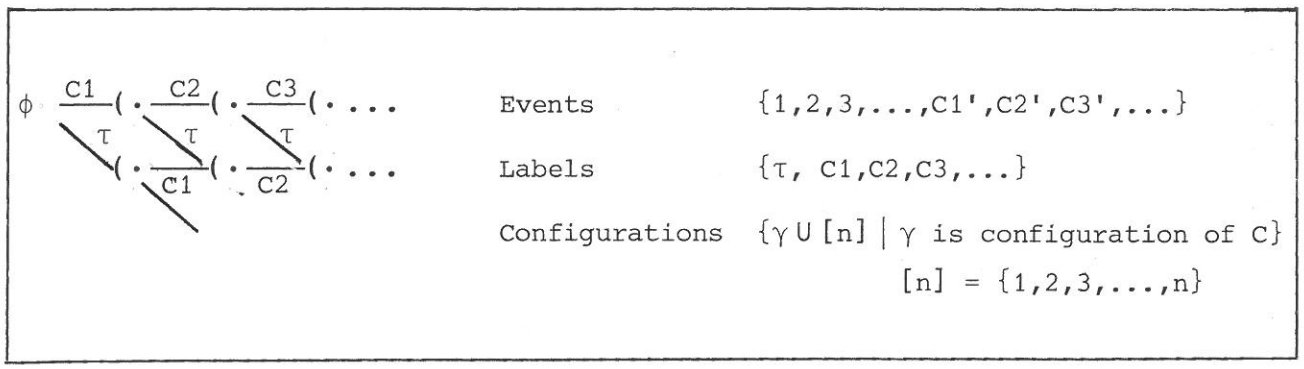


Fig. 7.5 Event System for "delay C".

$\langle E,F,\ell\rangle = \langle E_1,F_1,\ell_1\rangle \otimes \langle E_2,F_2,\ell_2\rangle$ by

(1) if $e \in E_1 \cup E_2$ does not have a rendezvous label, then $e \in E$;

(2) if $e_1 \in E_1$ has the rendezvous label call p and $e_2 \in E_2$ has the rendezvous label accept p do C., then $\langle e_1,e_2\rangle \in E$ has the label rendezvous C.;

(3) if $e_1 \in E_1$ has the rendezvous label accept p do C. and $e_2 \in E_2$ has the rendezvous label call p, then $\langle e_1,e_2\rangle \in E$ has the label rendezvous C.;

(4) the events in E are given by (1), (2), (3);

(5) a subset γ of E is in F if and only if $\pi_1(E) \in F_1$ & $\pi_2(E) \in F_2$ where

$$e_1 \in \pi_1(\gamma) \;.\equiv.\; e_1 \in E_1 \;\&\; (e_1 \in \gamma \vee \exists e_2.\langle e_1,e_2\rangle \in \gamma)$$

$$e_2 \in \pi_2(\gamma) \;.\equiv.\; e_2 \in E_2 \;\&\; (e_2 \in \gamma \vee \exists e_1.\langle e_1,e_2\rangle \in \gamma)$$

How can we remove events with label rendezvous C. from $\langle E,F,\ell\rangle$ to get the product event system $\langle E_1,F_1,\ell_1\rangle \times \langle E_2,F_2,\ell_2\rangle$? The meaning of the BED command C is an event system $\langle E_0,F_0,\ell_0\rangle$ and we can replace the events $\langle e_1,e_2\rangle$ by $\langle e_0,e_1,e_2\rangle$ where $e_0 \in E_0$. For a subset γ of E to be a configuration of the product we must also have

$$\{e_0 \mid \langle e_0,e_1,e_2\rangle \in \gamma\} \in F_0$$

for every $\langle e_1,e_2\rangle$ in γ.

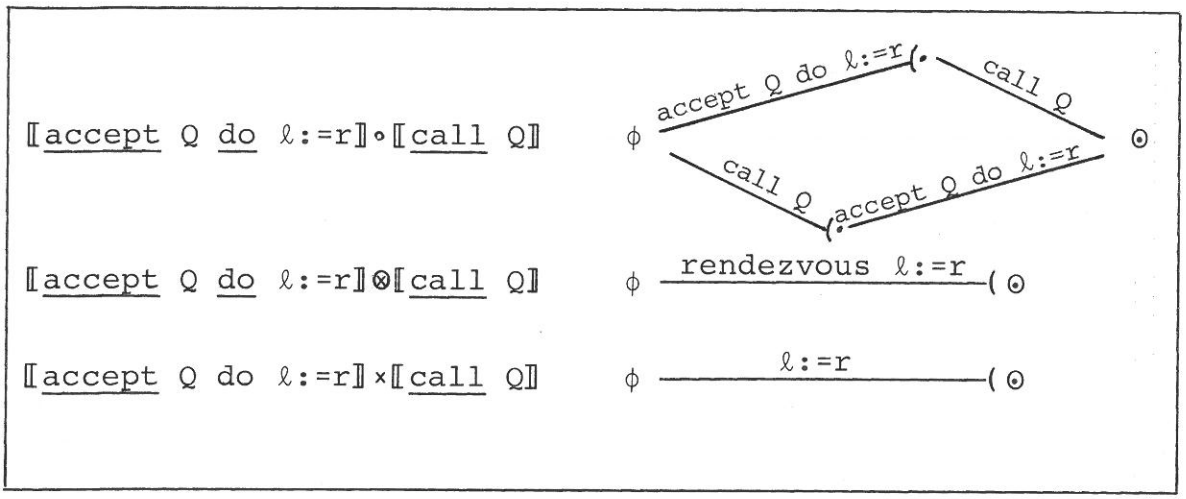


Fig. 7.6 Three Event System Combinators.

The meaning of the command "C1;C2" is the composition of the labelled event systems for C1 and C2. The event system $\langle E_1, F_1, \ell_1 \rangle \circledcirc \langle E_2, F_2, \ell_2 \rangle$ is the same as $\langle E_1, F_1, \ell_1 \rangle \circ \langle E_2, F_2, \ell_2 \rangle$ except that configurations incorporate sequencing

- a subset γ of $E_1 \cup E_2$ is a configuration if and only if $\gamma \in F_1$ or $\gamma = \gamma_1 \cup \gamma_2$ for some $\gamma_2 \in F_2$ and some terminal $\gamma_1 \in F_1$

How can we remove procedure calls from $\langle E_1, F_1, \ell_1 \rangle \circledcirc \langle E_2, F_2, \ell_2 \rangle$ to get the event system $\langle E_1, F_1, \ell_1 \rangle ; \langle E_2, F_2, \ell_2 \rangle$? Suppose $e_2 \in E_2$ with the label call Q occurs in a configuration $\gamma = \gamma_1 \cup \gamma_2$. If there is no event labelled accept Q do C. in γ_1, e_2 cannot be relabelled; if there is a last event labelled accept Q do C. then the event e_2 can be replaced by the event system for C, just as events with label rendezvous C were replaced; if there are several last events labelled accept Q do C. then the event e_2 can be replaced by the sum of the event systems for the C in these last events labelled accept Q do C.

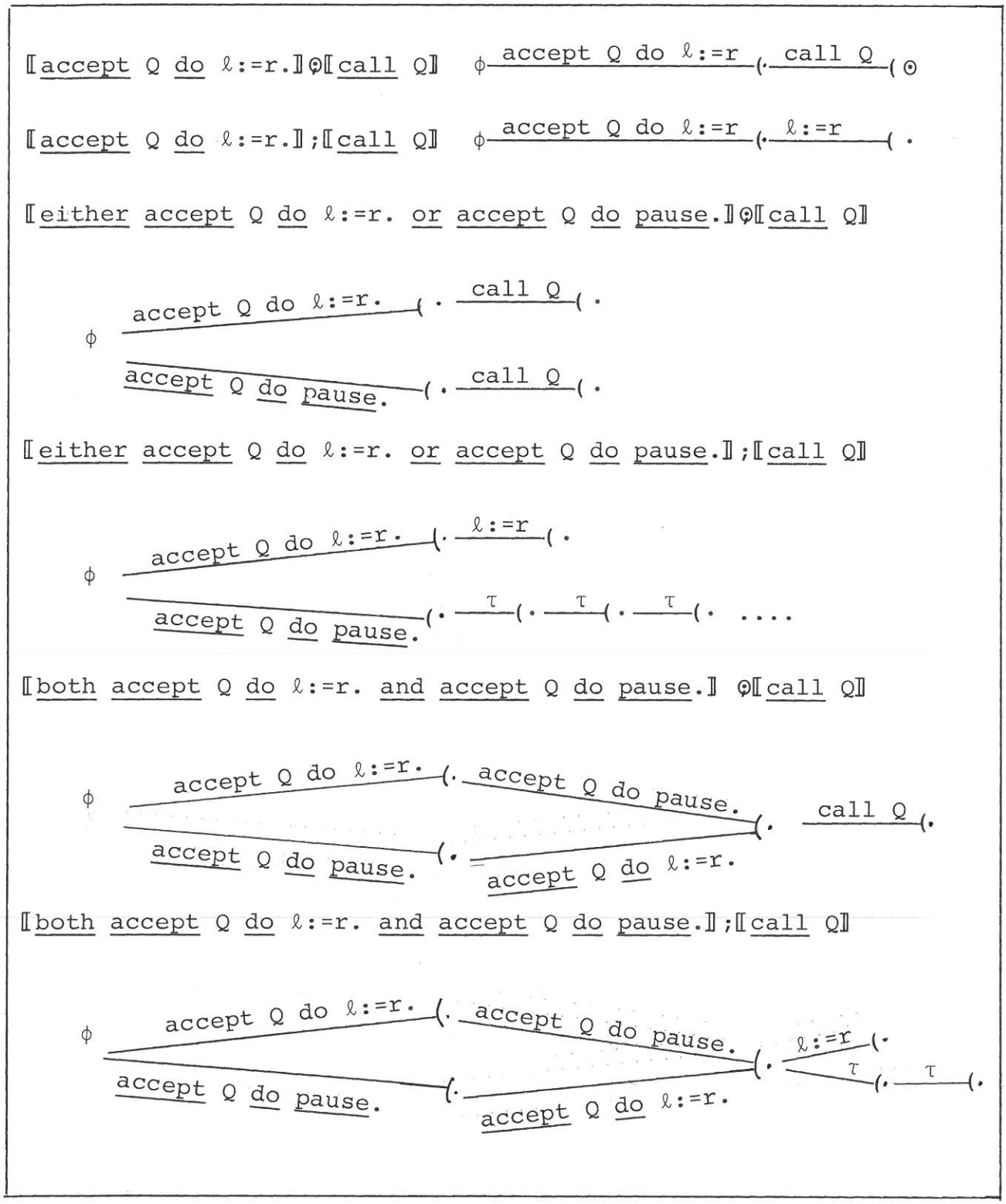


Fig. 7.7 Event Systems for Sequencing Commands.

We have used the notion of a terminal configuration in a labelled event system without defining it. We want terminal configurations to be different from maximal configurations because

- any configuration of the event system for pause should be terminal;
- any configuration of the event system for delay C. should be terminal, if it contains all the events for C;
- maximal configuration in the event system for exit when C. should only be terminal when the execution of C is not futile.

When we define the meanings for BED commands in Fig. 7.9 the terminal configurations will be specified precisely. The event system for the command exit when C is the same as that for C but some of the terminal configurations of ⟦C⟧ may no longer be terminal. This distinction between the event systems for C and exit when C is crucial to the meaning of loop C' end. The equation

$$⟦\text{loop } C'.⟧ = C'; ⟦\text{loop } C'.⟧$$

usually gives an infinite event system for loop C' end, but we are only interested in the maximal configurations given by the nonterminal configurations of C'.

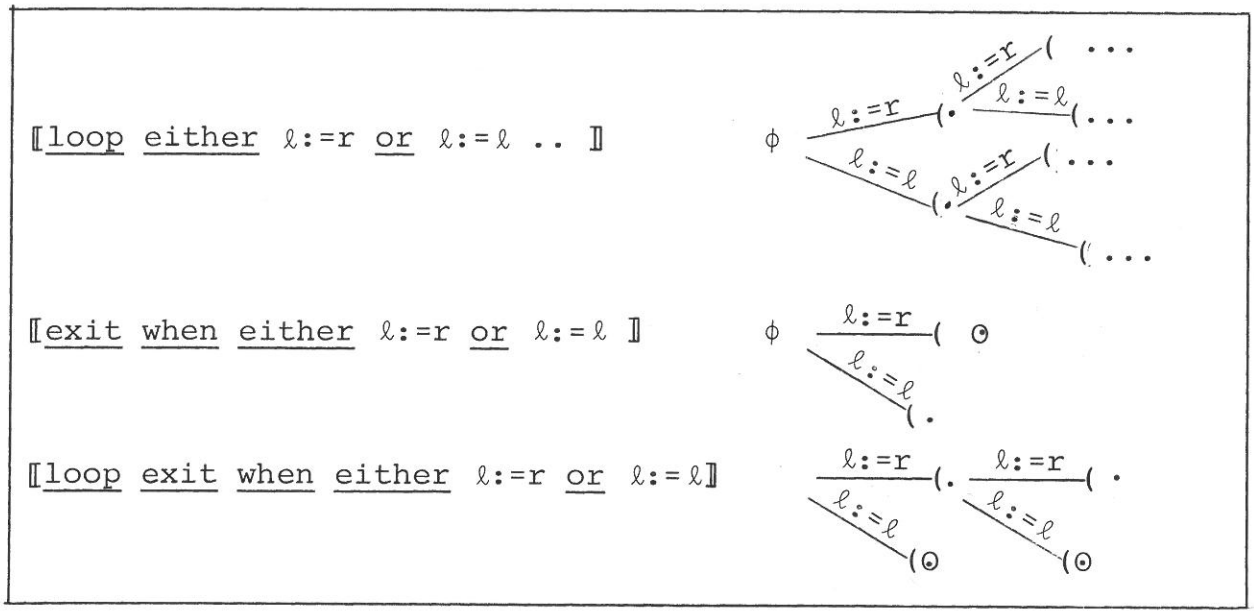


Fig. 7.8. Event Systems for Loop and Exit Commands.

$[\![\ell := r]\!] = \xrightarrow{\ell := r} (\odot$	-- $\odot$ indicates terminal configuration
$[\![\text{pause}]\!] = \odot \xrightarrow{\tau} (\odot \xrightarrow{\tau} (\odot \ldots$	-- all configurations terminal
$[\![\text{delay C.}]\!] = [\![C]\!] \times [\![\text{pause}]\!]$	-- configuration is terminal iff projection on $[\![C]\!]$ is terminal
$[\![\text{loop C.}]\!] = [\![C]\!] \ ; \ [\![\text{loop C}]\!]$	-- maximal configurations are terminal
$[\![\text{exit when C.}]\!] = [\![C]\!]$	-- configuration is terminal iff its events change the state
$[\![C1;C2]\!] = [\![C1]\!] \ ; \ [\![C2]\!]$	-- configuration is terminal iff projections on $[\![C1]\!]$ and $[\![C2]\!]$ are
$[\![\text{either C1 or C2.}]\!] = [\![C1]\!] + [\![C2]\!]$	-- configuration is terminal iff it is terminal in C1 or C2
$[\![\text{both C1 and C2.}]\!] = [\![C1]\!] \times [\![C2]\!]$	-- configuration is terminal iff projections on $[\![C1]\!]$ and $[\![C2]\!]$ are
$[\![\text{call p}]\!] = \cdot \xrightarrow{\text{call p}} (\odot$	
$[\![\text{accept P do C.}]\!] = \cdot \xrightarrow{\text{accept P do C}} (\odot$	

Fig. 7.9 Set Semantics for BED Commands.

7.4 Diagrams for BED Commands

The relation, function and set semantics for BED are interconnected. In this section we specify a diagram for each BED command and describe how the diagram is reflected in the three semantics. The diagrams for the four primitive BED commands are given in Figure 7.10. Unravelling these diagrams gives the event systems in Figure 7.4; assigning relations to the move labels gives the relational meaning of the four commands; assigning functions to the labels gives their functional meaning.

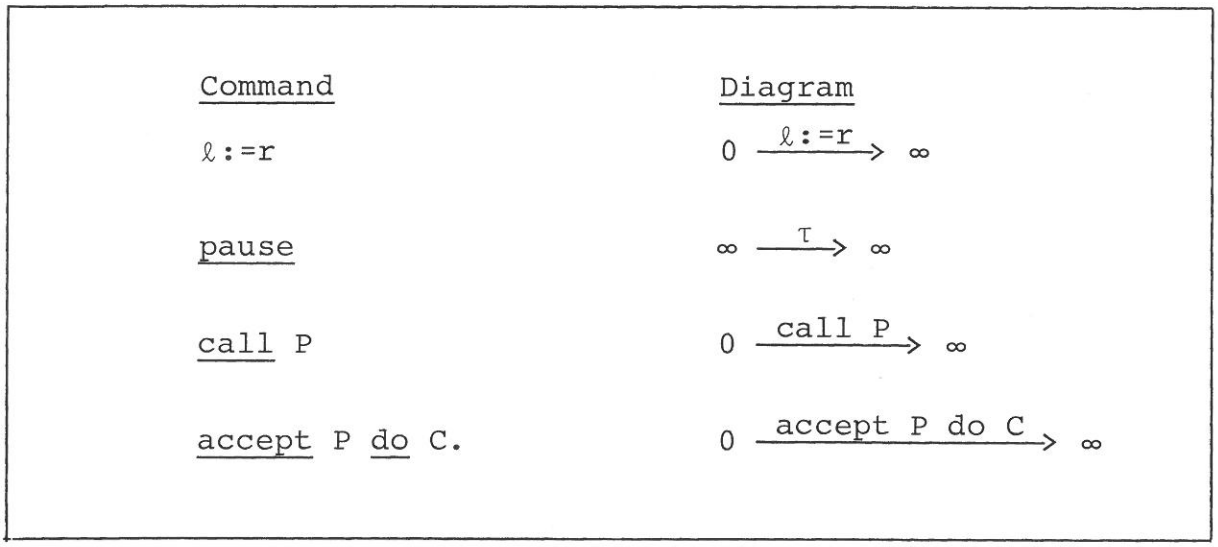


Fig. 7.10 Diagrams for four BED commands.

The diagram for the command "either C1 or C2." is the sum of the diagrams for C1 and C2. If the diagram $\mathcal{D}_1$ has the moves M1 and configurations S1, and the diagram $\mathcal{D}_2$ has the moves M2 and configurations S2, then the diagram $\mathcal{D}_1+\mathcal{D}_2$ has the moves M1 ∪ M2, and configurations S1 ∪ S2 - provided that the start configuration in S1 is identified with the start configuration in S2. Each of the diagrams for a BED command has a unique start configuration 0, reflecting the fact that the event system for each BED command has a unique start configuration ϕ (see figures in last section). Unravelling the sum of the diagrams for C1 and C2 gives the event system for either C1 or C2. Figure 7.11 shows how the relational and functional meanings of either C1 or C2. come from the command's diagram.

Moves in diagram : (1) $i \xrightarrow{\alpha} j$ in C1

(2) $k \xrightarrow{\beta} l$ in C2

Relational meaning: (1) $$\frac{\text{C1,s} \xrightarrow{\alpha} \text{[C',]s'}}{\underline{\text{either}}\ \text{C1}\ \underline{\text{or}}\ \text{C2,.s} \rightarrow \text{[C',]s'}}$$

(2) $$\frac{\text{C2,s} \xrightarrow{\beta} \text{[C',]s'}}{\underline{\text{either}}\ \text{C1}\ \underline{\text{or}}\ \text{C2.,s} \rightarrow \text{[C',]s'}}$$

Functional meaning: (1) $[\![\underline{\text{either}}\ \text{C1}\ \underline{\text{or}}\ \text{C2.}]\!](\text{Tw,s}) \rightarrow [\![\text{C1}]\!](\text{w,s})$

(2) $[\![\underline{\text{either}}\ \text{C1}\ \underline{\text{or}}\ \text{C2.}]\!](\text{Fw, }) \rightarrow [\![\text{C2}]\!](\text{w,s})$

Fig. 7.11 Meanings of either C1 or C2.

The diagram of the command "both C1 and C2." is the product of the diagrams for C1 and C2. If the diagram $\mathcal{D}_1$ has moves M1 and configurations S1, and the diagram $\mathcal{D}_2$ has moves M2 and configurations S2, then the diagram $\mathcal{D}_1 \times \mathcal{D}_2$ has the configurations S1×S2 and the moves given by

- $i_1 \times i_2 \xrightarrow{\alpha} j_1 \times i_2$ when $i_1 \xrightarrow{\alpha} j_1$ is a move of $\mathcal{D}_1$ and α is neither call P nor accept P do C.;
- $i_1 \times i_2 \xrightarrow{\beta} i_1 \times j_2$ when $i_2 \xrightarrow{\beta} j_2$ is a move of $\mathcal{D}_2$ and α is neither call P nor accept P do C.;
- $i_1 \times i_2 \xrightarrow{\text{rendezvous C}} j_1 \times j_2$ when $i_1 \xrightarrow{\text{call P}} j_1$ is a move of $\mathcal{D}_1$ and $i_2 \xrightarrow{\text{accept P do C.}} j_2$ is a move of $\mathcal{D}_2$;
- $i_1 \times i_2 \xrightarrow{\text{rendezvous C}} j_1 \times j_2$ when $i_1 \xrightarrow{\text{accept P do C.}} j_1$ is a move of $\mathcal{D}_1$ and $i_2 \xrightarrow{\text{call P}} j_2$ is a move of $\mathcal{D}_2$.

Fig. 7.12 shows how the relational and functional meanings of both C1 and C2 come from the command's diagram.

Unravelling the diagram of both C1 and C2. gives the labelled event system $[\![C1]\!] \otimes [\![C2]\!]$ and substituting the event system of C for the events with label rendezvous C gives the event system for both C1 and C2.

Moves in diagram :

(1) $i_1 \times i_2 \xrightarrow{\alpha} j_1 \times j_2$ from $i_1 \xrightarrow{\alpha} j_1$ in $[\![C1]\!]$

(2) $i_1 \times i_2 \xrightarrow{\beta} j_1 \times j_2$ from $i_2 \xrightarrow{\beta} j_2$ in $[\![C2]\!]$

(3) $i_1 \times i_2 \xrightarrow{\text{rendezvous C}} j_1 \times j_2$

Relational meaning:

(1) $$\frac{\text{C1,s} \rightarrow \text{[C1',]s'... Free(C1)}}{\text{both C1 and C2.,s} \rightarrow \text{[both C' and]C2[.],s'}}$$

(2) $$\frac{\text{C2,s} \rightarrow \text{[C',]s' ... Free(C1)}}{\text{both C1 and C2.,s} \rightarrow \text{[both]C1[and C'.],s'}}$$

(3) four rules in section 7.1.

Functional meaning:

(1) $[\![\text{both C1 and C2.}]\!](FTw,s) = f_1(w,s)$

(2) $[\![\text{both C1 and C2.}]\!](FFw,s) = f_2(w,s)$

(3) $[\![\text{both C1 and C2.}]\!](Tw,s) = g(w,s)$

with the functions f_1, f_2, g from section 7.2.

Fig. 7.13 Meanings of both C1 and C2.

The diagram for the command "C1;C2" is the composition of the diagrams for C1 and C2. If the diagram $\mathcal{D}_1$ has moves M1 and configurations S1, and the diagram $\mathcal{D}_2$ has moves M2 and configurations S2, then the diagram $\mathcal{D}_1;\mathcal{D}_2$ has moves M1∪M2 and configurations S1∪S2 - provided that each terminal configuration in S1 is identified with the start configuration of S2. Figure 7.14 shows how the relational and functional meanings of "C1;C2" come

from the command's diagram. Unravelling the diagram of "C1;C2" gives the labelled event system ⟦C1⟧ ⊙ ⟦C2⟧ and substituting the event system of C for calls of procedures with body C gives the event system for C1;C2 - as we saw in section 7.3 the details of this substitution are tricky.

Moves in diagram :	(1)	$i \xrightarrow{\alpha} j$ in C1
	(2)	$k \xrightarrow{\beta} l$ in C2
Relational meaning:	(1)	$\dfrac{C1,s \to [C',]s'}{C1;C2,s \to [C',]C2,s}$
Functional meaning:	(1)	$⟦C1;C2⟧(w,s) = ⟦C2⟧(⟦C1⟧(w,s))$

Fig. 7.14 Meanings of C1;C2

We have used the notion of a terminal configuration of a diagram without defining it. If we defined a diagram configuration j to be terminal when there was no move with source j, then we would have problems with exit when C., pause and delay C.

If we introduce the convention that terminal configurations are labelled ∞, figure 7.10 gives the diagram for pause. The diagram for delay C. is defined to be the same as the diagram for both C and pause. Unravelling this diagram gives the event system for delay C and figure 7.15 shows how the relational and functional meanings of delay C come from the diagram.

Moves in diagram : (1) $i\times 0 \xrightarrow{\alpha} j\times 0$ from $i \xrightarrow{\alpha} j$ in $[\![C]\!]$

(2) $i\times 0 \xrightarrow{\tau} i\times 0$

where $j\times 0$ is ∞ when j is terminal in $[\![C]\!]$

Relational meaning: (1) $\dfrac{C,S \to [C',]S'}{\underline{\text{delay}}\ C.,S \to [\underline{\text{delay}}\ C'.]S'}$

(2) $\dfrac{}{\underline{\text{delay}}\ C.,S \to \underline{\text{delay}}\ C.,S}$

Functional meaning: (1) $[\![\underline{\text{delay}}\ C.]\!](Tw,s) = f(w,s)$

(2) $[\![\underline{\text{delay}}\ C.]\!](Fw,s) = [\![\underline{\text{delay}}\ C.]\!](w,s)$

where $f = [\![\text{First}(C)]\!] \,|\, [\![\text{Rest}(\underline{\text{delay}}\ C.)]\!]$

Fig. 7.15 Meanings of delay C.

The diagram for exit when C is the same as the diagram for C but their terminal configurations differ, and this difference is crucial to the diagram for loop C. If we identify the terminal configurations of C with the start configuration we get the diagram for loop C. when we agree that the terminal configurations are given by: no move has this configuration as source. If one compares the diagrams for loop either ℓ:=r or ℓ:=ℓ.., exit when either ℓ:=r or ℓ:=ℓ.., and loop exit when either ℓ:=r or ℓ:=ℓ.. . in figure 7.16 with the event systems in figure 7.8, one sees how diagrams for exit and loop commands unravel to the appropriate event systems.

command	moves in diagram
loop either ℓ:=r or ℓ:=ℓ..	$0 \xrightarrow{\ell:=r} 0$ $0 \xrightarrow{\ell:=\ell} 0$
exit when either ℓ:=r or ℓ:=ℓ..	$0 \xrightarrow{\ell:=r} \infty$ $0 \xrightarrow{\ell:=\ell} 1$
loop exit when either ℓ:=r or ℓ:=ℓ..	$0 \xrightarrow{\ell:=r} 0$ $0 \xrightarrow{\ell:=\ell} \infty$

Fig. 7.16 Diagrams for loop and exit commands.

One should not expect a close connection between the diagrams for exit/loop commands and functional/relational semantics because of the difference between static program text and its dynamic execution. In sections 7.1 and 7.2 we captured this difference by ad hoc devices, but we could have used fixed points, continuations and other clean techniques in the literature.

The diagram, relational, functional and set semantics of BED are rather intricate, but the language has complex features and there are more examples in the appendix.

REFERENCES

[A] "Formal definition of ADA", CII Honeywell Bull, 1981, Paris.

[B1] J.D. Brock & W.B. Ackerman: "Scenarios: a model of non-deterministic computation" in "Formalisation of programming concepts", ed. J. Diaz & I. Ramos, LNCS 107 (1981), 252-259.

[B2] S.D. Brookes: "On the relationship between CCS and CSP", ICALP 83 proc. LNCS 154 (1983), 83-96.

[F] N. Franchez, C.A.R. Hoare, D.J. Lehmann, W.P. de Roever: "Semantics on nondeterminacy, concurrency and communication", J. Comp. Sys. Sci. 19 (1979), 290-308.

[M1] B.H. Mayoh: "Datatypes as functions", MFCS 78, proc. LNCS 64, (1978), 56-70.

[M2] B.H. Mayoh: "A computational model for ADA and other concurrent languages", DAIMI PB-162, Aarhus University 1983.

[M3] B.H. Mayoh: "Program models, meaning and proof", DAIMI PB-157, Aarhus University, 1983.

[N] M. Nivat: "Infinitary relations", CAAP 81 proc., LNCS 112 (1981), 46-75.

[P1] G. Plotkin: "A structural view of operational semantics", DAIMI FN-19, Aarhus University 1981.

[P2] V.R. Pratt: "On the composition of processes", POPL 9 proc., ACM (1982), 213-223.

[R] W.C. Rounds & S.D. Brookes: "Possible futures, acceptances, refusals and communication processes", FOCS 22 proc., IEEE (1981), 140-149.

[S] R. Sethi: "Circular expressions: elimination of static environments", Science of Computer Programming 1 (1982), 203-222.

[W1] G. Winskel: "Event structure for CCS and related languages", ICALP 82 proc. LNCS 140 (1982), 561-576.

[W2] G. Winskel: "Synchronization trees", ICALP 83 proc. LNCS 154 (1983), 695-711.

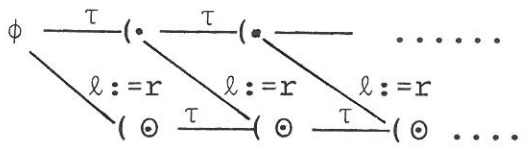
LNCS abbreviates "Springer Lecture Notes in Computer Science". Because the literature on semantics is so large, many relevant and important papers have not been referenced.

APPENDIX

For thirteen BED commands we will give a diagram and the relational, functional and set meaning of each command. The command (1) delay ℓ:=r. has the diagram with the moves:

$$0 \xrightarrow{\ell:=r} \infty \ , \ 0 \xrightarrow{\tau} 0 \ , \ \infty \xrightarrow{\tau} \infty$$

The set meaning of (1) is the event system with covering diagram



The relational meaning of (1) is given by the proof rules

$$\frac{}{\text{delay } \ell:=r.,s \rightarrow \text{delay } \ell:=r.,s} \qquad \frac{r,s \xrightarrow{FV} n \ \ldots \ n,\ell,s \xrightarrow{SV} s'}{\text{delay } \ell:=r.,s \rightarrow s'}$$

If neither ℓ nor r is a reserved identifier,

$$r,s \xrightarrow{FV} n \ \& \ n,\ell,s \xrightarrow{SV} s' \text{ implies } s' = \text{StoreValue}(\text{FetchValue}(r,s),\ell,s)$$

so the result of executing (1) from s is uniquely determined. The functional meaning of (1) is

$$[\![\text{delay } \ell:=r.]\!] = \text{PEEK}([\![\ell:=r]\!],[\![\text{delay } \ell:=r.]\!])$$

$$[\![\text{delay } \ell:=r.]\!](Tw,s) = [\![\ell:=r]\!](w,s) = FV\ r\ SV\ \ell(w,s)$$

$$= SV\ \ell(FV\ r\ (w,s))$$

$$[\![\text{delay } \ell:=r.]\!](Fw,s) = [\![\text{delay } \ell:=r.]\!](w,s)$$

where SV ℓ and FV r are the oracular functions defined in section 7.2. Note that the diagram and the set meanings of delay C allows for "waiting after C" but the relational and functional meanings do not.

The command (2) both ℓ:=r and pause. has the same diagram and event system as delay ℓ:=r. The relational meaning of (2) is given by the proof rules

$$\frac{}{\text{both } \ell:=r \text{ and pause.},s \to \text{both } \ell:=r \text{ and pause.},s}$$

$$\frac{r,s \xrightarrow{FV} n \ldots n,\ell,s \xrightarrow{SV} s'}{\text{both } \ell:=r \text{ and pause.},s \to \text{pause},s'}$$

so (2) is different from (1) because it can "wait after ℓ:=r". The functional meaning of (2) is

$$[\![\text{both } \ell:=r \text{ and pause.}]\!] = PEEK(UNDEF,PEEK([\![\ell:=r]\!],[\![\text{pause}]\!]))$$

$$[\![\text{both } \ell:=r \text{ and pause.}]\!](FTw,s) = [\![\ell:=r]\!](w,s)$$

$$[\![\text{both } \ell:=r \text{ and pause.}]\!](FFTw,s) = [\![\ell:=r]\!](w,s)$$

$$[\![\text{both } \ell:=r \text{ and pause.}]\!](FFFw,s) = [\![\text{both } \ell:=r \text{ and pause.}]\!](w,s)$$

$$[\![\text{both } \ell:=r \text{ and pause.}]\!](Tw,s) = \text{undefined}$$

so it cannot wait after ℓ:=r.

The command (3) either ℓ:=r or ℓ:=ℓ. has the diagram

$$0 \xrightarrow{\ell:=r} \infty \quad , \quad 0 \xrightarrow{\ell:=\ell} \infty$$

and the set meaning

$$\phi \xrightarrow{\ell:=r} (\odot$$
$$\phi \xrightarrow{\ell:=\ell} (\odot$$

The relational meaning of (3) is given by the proof rules

$$\frac{r,s \xrightarrow{FV} n \;..\; n,\ell,s \xrightarrow{SV} s'}{\text{either } \ell:=r \text{ or } \ell:=\ell.,s \to s'} \qquad \frac{\ell,s \xrightarrow{FV} n \;..\; n,\ell,s \xrightarrow{SV} s'}{\text{either } \ell:=r \text{ or } \ell:=\ell.,s \to s'}$$

and the functional meaning of (2) is

$$[\![\underline{\text{either}}\ \ell:=r\ \underline{\text{or}}\ \ell:=\ell.]\!] = \text{PEEK}([\![\ell:=r]\!],[\![\ell:=\ell]\!])$$

$$[\![\underline{\text{either}}\ \ell:=r\ \underline{\text{or}}\ \ell:=\ell.]\!](Tw,s) = [\![\ell:=r]\!](w,s)$$

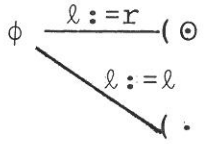
$$[\![\underline{\text{either}}\ \ell:=r\ \underline{\text{or}}\ \ell:=\ell.]\!](Fw,s) = [\![\ell:=\ell]\!](w,s)$$

Note how PEEK uses the oracle to choose between alternatives.

The command (4) exit when either $\ell:=r$ or $\ell:=\ell$.. has the diagram

$$0 \xrightarrow{\ell:=r} \infty \ , \ 0 \xrightarrow{\ell:=\ell} 1$$

and the set meaning



The corresponding relational meaning of (4) is given by the proof rules

$$\frac{r,s \xrightarrow{FV} n \ .. \ n,\ell,s \xrightarrow{SV} s'}{\underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{either}}\ \ell:=r\ \underline{\text{or}}\ \ell:=\ell..,s\rightarrow s'}$$

$$\frac{}{\underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{either}}\ \ell:=r\ \underline{\text{or}}\ \ell:=\ell..,s\rightarrow \underline{\text{failure}},s}$$

The functional meaning of (4) is

$$[\![\underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{either}}\ \ell:=r\ \underline{\text{or}}\ \ell:=\ell]\!]$$
$$= \text{ENTER} \mid \text{PEEK}([\![\ell:=r]\!],[\![\ell:=\ell]\!] \mid \text{FUTILE}$$

and the relational meaning of (4) is given by the rules:

$$\frac{r,s \xrightarrow{FV} n\ ..\ n,\ell,s \xrightarrow{SV} s}{(4),s \to \underline{failure},s} \qquad \frac{r,s \xrightarrow{FV} n\ ..\ n,\ell,s \xrightarrow{SV} s'..s \neq s'}{(4),s \to s'}$$

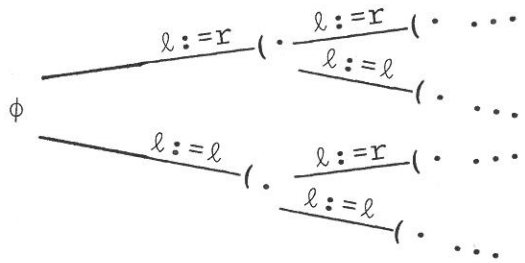
$$\frac{\ell,s \xrightarrow{FV} n\ ..\ n,\ell,s \xrightarrow{SV} s}{(4),s \to \underline{failure},s} \qquad \frac{\ell,s \xrightarrow{FV} n\ ..\ n,\ell,s \xrightarrow{SV} s'..s \neq s'}{(4),s \to s'}$$

Note how these two meanings capture the remote possibility that ℓ:=r might be futile or ℓ:=ℓ might not be futile. The diagram and event system approaches can be modified to handle such possibilities, but we have not done so because futility is just an artificial way of avoiding the tests and conditions in real programming languages.

The command (5) loop either ℓ:=r or ℓ:=ℓ.. has the diagram

$$0 \xrightarrow{\ell:=r} 0, \quad 0 \xrightarrow{\ell:=\ell} 0$$

and the event system



The relational meaning of (5) is given by the proof rules

$$\frac{[\![\ell:=r]\!],s \to s'}{(5),s \to (5),s'} \qquad \frac{[\![\ell:=\ell]\!],s \to s'}{(5),s \to (5),s'} \qquad \frac{}{\underline{failure};(5),s \to s}$$

and the functional meaning of (5) is

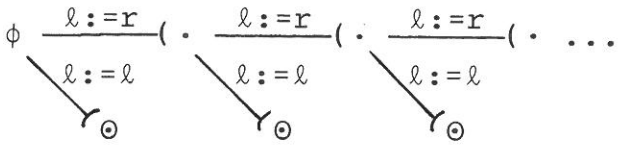
$$[\![\underline{loop}\ \underline{either}\ \ell:=r\ \underline{or}\ \ell:=\ell..]\!]$$
$$= \text{ENTER} \mid \text{PEEK}([\![\ell:=r]\!],[\![\ell:=\ell]\!]) \mid \text{LEAVE} \mid \text{PEEK}(\text{ID},$$
$$[\![\underline{loop}\ \underline{either}\ \ell:=r\ \underline{or}\ \ell:=\ell..]\!])$$

Note how these two meanings do not reveal the fact that loops without exit commands are eternal.

The command (6) loop exit when either ℓ:=r or ℓ:=ℓ.. has the diagram

$$0 \xrightarrow{\ell:=r} \ , \ 0 \xrightarrow{\ell:=\ell} \infty$$

and the event system



The relational meaning of (6) is given by the proof rules

$$\frac{[\![\ell:=r]\!],s \to s}{(6),s \to \underline{failure};(6),s} \qquad \frac{[\![\ell:=\ell]\!],s \to s}{(6),s \to \underline{failure};(6),s} \qquad \frac{}{\underline{failure};(6),s \to s}$$

$$\frac{[\![\ell:=r]\!],s \to s' \ .. \ s \neq s'}{(6),s \to (6),s'} \qquad \frac{[\![\ell:=\ell]\!],s \to s'}{(6),s \to (6),s'}$$

and the functional meaning of (6) is

$$[\![\underline{loop}\ \underline{exit}\ \underline{when}\ \underline{either}\ \ell:=r\ \underline{or}\ \ell:=\ell..]\!]$$
$$= \text{ENTER}\,|\,\text{ENTER}\,|\,\text{PEEK}([\![\ell:=r]\!],[\![\ell:=\ell]\!])\,|\,\text{FUTILE}\,|\,\text{LEAVE}\,|\,\text{PEAK}(\text{ID},(6))$$

Note how the relational meaning introduces an extra "recognition of failure" step in computations.

The command (7) accept Q do ℓ:=r.; call Q has the diagram

$$0 \xrightarrow{\text{accept Q do } \ell:=r.} 1, \ 1 \xrightarrow{\text{call Q}} \infty$$

and the event system

$$\phi \xrightarrow{\text{accept Q do } \ell:=r} \cdot \xrightarrow{\ell:=r} \odot$$

The relational meaning of (7) is given by the proof rules

$$\frac{\ell:=r,Q,s \xrightarrow{SA} s'}{(7),s \to \underline{call}\ Q,s'} \qquad \frac{Q,s' \xrightarrow{FA} \ell:=r\ ..\ \ell:=r,s' \to s''}{\underline{call}\ Q,s' \to s''}$$

and the functional meaning is

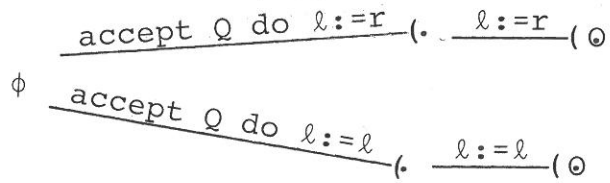
$$[\![\underline{accept}\ Q\ \underline{do}\ \ell:=r.;\ \underline{call}\ Q]\!] = SA\ Q\ \ell:=r \mid FA\ Q$$

Note how close the functional meaning is to the code a compiler might generate from (7).

The command (8) $\underline{either}$ $\underline{accept}$ Q $\underline{do}$ ℓ:=r. $\underline{or}$ $\underline{accept}$ Q $\underline{do}$ ℓ:=ℓ..; $\underline{call}$ Q has the diagram

$$0 \xrightarrow{\underline{accept}\ Q\ \underline{do}\ \ell:=r.} 1,\ 0 \xrightarrow{\underline{accept}\ Q\ \underline{do}\ \ell:=\ell.} 1,\ 1 \xrightarrow{\underline{call}\ Q} \infty$$

and the event system



The relational meaning of (8) is given by the proof rules

$$\frac{\ell:=r,Q,s \xrightarrow{SA} s'}{(8),s \to \underline{call}\ Q,s'} \qquad \frac{Q,s' \longrightarrow \ell:=r..\ell:=r,s' \to s''}{\underline{call}\ Q,s' \to s''}$$

$$\frac{\ell:=\ell,Q,s \xrightarrow{SA} s'}{(8),s \to \underline{call}\ Q,s'} \qquad \frac{Q,s' \xrightarrow{FA} \ell:=\ell..\ell:=\ell,s' \to s''}{\underline{call}\ Q,s' \to s''}$$

and the functional meaning is

$$[\![\underline{either}\ \underline{accept}\ Q\ \underline{do}\ \ell:=r.\ \underline{or}\ \underline{accept}\ Q\ \underline{do}\ \ell:=\ell..;\ \underline{call}\ Q]\!]$$
$$= PEEK(SA\ Q\ \ell:=r,\ SA\ Q\ \ell:=\ell) \mid FA\ Q$$

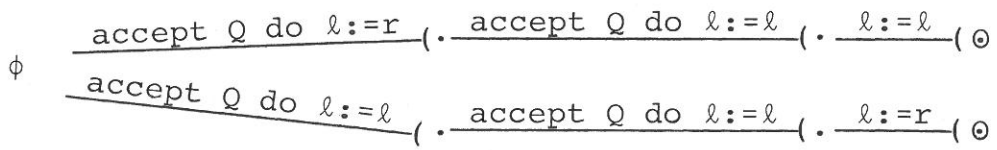
Note how the functional meaning reflects the diagram, not the event system of (8).

The command (9) both accept Q do ℓ:=r. and accept Q do ℓ:=ℓ..; call Q has the diagram

$$0 \xrightarrow{\text{accept Q do } \ell:=r} 1,\ 1 \xrightarrow{\text{accept Q do } \ell:=\ell} 2,\ 2 \xrightarrow{\text{call Q}} \infty$$

$$0 \xrightarrow{\text{accept Q do } \ell:=\ell} 3,\ 3 \xrightarrow{\text{accept Q do } \ell:=r} 2$$

and the event system



The relational meaning of (9) is given by the proof rules

$$\frac{\ell:=r,Q,s \xrightarrow{SA} s'''}{(9),s \to \text{accept Q do } \ell:=\ell.;\text{call Q},s'''} \qquad \frac{\ell:=\ell,Q,s''' \xrightarrow{SA} s'}{\text{accept Q do } \ell:=\ell.;\text{call Q},s''' \to \text{call Q},s'}$$

$$\frac{\ell:=\ell,Q,s \xrightarrow{SA} s'''}{(9),s \to \text{accept Q do } \ell:=r.,;\text{call Q},s'''} \qquad \frac{\ell:=r,Q,s''' \xrightarrow{FA} s'}{\text{accept Q do } \ell:=\ell.;\text{call Q},s''' \to \text{call Q},s'}$$

$$\frac{Q,s' \xrightarrow{FA} \ell:=\ell \ \ldots \ \ell:=\ell,s' \to s''}{\text{call Q},s' \to s''} \qquad \frac{Q,s' \xrightarrow{FA} \ell:=r \ \ldots \ \ell:=r,s' \to s''}{\text{call Q},s' \to s''}$$

and the functional meaning is

$$[\![\text{both accept Q do } \ell:=r. \text{ and accept Q do } \ell:=\ell..; \text{ call Q}]\!]$$
$$= \text{PEEK(UNDEF,PEEK(SA Q } \ell:=r \mid \text{SA Q } \ell:=\ell,\text{SA Q } \ell:=\ell \mid \text{SA Q } \ell:=\ell) \mid \text{FA Q}$$

Note that none of these meanings reflects the fact that (8) and (9) have the same behaviour.

The command(10) both accept Q do ℓ:=r. and call Q has the diagram

$$0 \xrightarrow{\text{rendezvous } \ell:=r} \infty$$

and the event system

$$\phi \xrightarrow{\ell:=r} (\odot$$

The relational meaning of (10) is given by the proof rules

$$\frac{\ell:=r,Q,s \longrightarrow s'}{\text{accept } Q \text{ do } \ell:=r.,s\rightarrow s'} \qquad \frac{Q,s' \longrightarrow \ell:=r \ldots \ell:=r,s'\rightarrow s"}{\text{call } Q,s'\rightarrow s"}$$

$$\frac{\text{accept } Q \text{ do } \ell:=r,s\rightarrow s'..\text{call } Q,s'\rightarrow s" \ldots \text{Match(call } Q,\text{accept } Q \text{ do } \ell:=r)}{\text{both accept } Q \text{ do } \ell:=r. \text{ and call } Q,s\rightarrow s"}$$

and the functional meaning is

$$[\![\text{both accept } Q \text{ do } \ell:=r. \text{ and call } Q.]\!] = \text{PEEK(SA } Q\ \ell:=r \mid \text{FA } Q,\text{UNDEF})$$

where UNDEF = PEEK(UNDEF,UNDEF) is the always undefined function.

The command (11) both either accept Q do ℓ:=r. or accept Q do ℓ:=ℓ.. and call Q. has the diagram

$$0 \xrightarrow{\text{rendezvous } \ell:=r} \infty,\ 0 \xrightarrow{\text{rendezvous } \ell:=\ell} \infty$$

and the event system

$$\phi \begin{array}{l} \xrightarrow{\ell:=r} (\odot \\ \xrightarrow{\ell:=\ell} (\odot \end{array}$$

The relational meaning of (11) is given by the proof rules

$$\frac{\ell:=r,Q,s \longrightarrow\!\!\!\!\!\rightarrow s'...Q,s' \longrightarrow \ell:=r\ ..\ \ell:=r,s'\rightarrow s"\ ..\ \text{Match(call Q,accept Q do } \ell:=r)}{(11),s \rightarrow s"}$$

$$\frac{\ell:=\ell,Q,s \longrightarrow s'..\ Q,s' \longrightarrow \ell:=\ell\ ..\ \ell:=\ell,s'\rightarrow s"\ ..\ \text{Match(call Q,accept Q do } \ell:=\ell)}{(11),s \rightarrow s'}$$

and the functional meaning is

both either accept Q do ℓ:=r. or accept Q do ℓ:=ℓ..and call Q.

= PEEK(PEEK SA Q ℓ:=r|FA Q,SA Q ℓ:=ℓ|FA Q),UNDEF)

One would expect the meaning of (11) to be similar to the meaning of (12) both both accept Q do ℓ:=r, and accept Q do ℓ:=ℓ.. and call Q. because the meaning of (8) was so close to the meaning of (9). This expectation is frustrated because both accept Q do ℓ:=r. and accept Q do ℓ:=ℓ.. deadlocks when Q is an entry. For this reason (12) "jams". The diagram has no move, the event system has ϕ as its only configuration, no proof rules apply, and UNDEF is the functional meaning. A both command jams when either of its components jams, but an either command only jams when both its components jam.

To get a true impression of the advantages and disadvantages of the various BED semantics, let us look at the larger command (13)

```
accept P do pause.;
both loop call Q; exit when call P ..
and accept Q do external := internal ..
```

This command has the diagram

$$0 \xrightarrow{\text{accept P do pause}} 1,1 \xrightarrow{\text{rendezvous external:=internal}} \infty,\infty \xrightarrow{\tau} \infty$$

because pause is always futile and exit when call P terminates the loop. The set meaning of (13) is the event system

$$\phi \xrightarrow{\text{accept P do pause}} \cdot \xrightarrow{\text{external:=internal}} \odot \xrightarrow{\tau} \odot \xrightarrow{\tau} \odot \ \ldots$$

The relational meaning of (13) was analyzed in the example in section 7.1; it is given by the proof rules

$$\frac{\text{pause},P,s0 \xrightarrow{SA} s1}{(13),s0\ \underline{\text{both}}\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..\ \underline{\text{and}}\ \underline{\text{accept}}\ Q\ \underline{\text{do}}\ \underline{\text{external}}:=\underline{\text{internal}}..,s1}$$

$$\frac{\text{external} := \text{internal},s1 \xrightarrow{SA} s2\ ..\ \text{external}:=\text{internal},s2\rightarrow s3}{\begin{array}{l}\underline{\text{both}}\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..\ \underline{\text{and}}\ \underline{\text{accept}}\ Q\ \underline{\text{do}}\ \underline{\text{external}}:=\underline{\text{internal}}..,s1 \\ \qquad\qquad \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.;\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..,s3\end{array}}$$

$$\frac{\text{pause},s3 \rightarrow s3}{\begin{array}{l}\underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P.;\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..,s3 \\ \qquad\qquad \underline{\text{failure}};\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..,s3\end{array}}$$

$$\frac{}{\underline{\text{failure}};\ \underline{\text{loop}}\ \underline{\text{call}}\ Q;\ \underline{\text{exit}}\ \underline{\text{when}}\ \underline{\text{call}}\ P..,\ s3 \rightarrow s3}$$

The functional meaning of (13) was analyzed in the example in section 7.2; it was

$$[\![13]\!] = \text{SA P pause} \mid \text{PEEK (ENTER}\mid\text{SA Q external:=internal} \mid \text{FA Q} \mid\text{ENTER}\mid\text{FA P}\mid\text{FUTILE}\mid\text{LEAVE}\mid\text{PEEK(ID,f)},\text{PEEK(UNDEF,UNDEF))}$$

where f = ENTER|FA Q|ENTER|FA P|FUTILE|LEAVE|PEEK(ID,f).

From the examples in this appendix we might conclude

- set semantics handles parallelism best, but it must be completed by an assignment of a relational or functional meaning to events;
- relational and functional semantics require clumsy syntactic predicates;
- functional semantics is closer to code than relational semantics, but it is somewhat more complicated.