ON DENOTATIONAL VERSUS PREDICATIVE SEMANTICS

Manfred Broy* and
Christian Lengauer

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-29 July 1987

Abstract

Two definitions of a language of communicating programs are offered: one by denotational semantics, and one by predicative specifications. The equivalence of both definitions is established. Both partial and total correctness semantics are considered. Nondeterminism and its interaction with recursion is studied. The main contribution is a comparative study of the descriptive and the prescriptive viewpoint of program semantics.

This paper is also available as tech. report MIP-8713 of the
Fakultät für Mathematik und Informatik, Universität Passau.

*Fakultät für Mathematik und Informatik, Universität Passau, Postfach 2540, D-8390, Passau, F.R.G.

Table of Contents

1. Introduction 1
2. Syntax 1
3. Program States 2
4. Deterministic Programs 4
    4.1. Denotational Semantics 5
    4.2. Predicative Semantics 7
    4.3. Connection Between the Two Semantics 10
    4.4. Partial Correctness and Predicative Semantics 11
    4.5. Robust Correctness and Predicative Specifications 11
5. Nondeterministic Programs 12
    5.1. Denotational Semantics 12
    5.2. Predicative Semantics 13
    5.3. Predicative Specifications as Sets of Functions 13
6. Correctness of Nondeterministic Programs 15
    6.1. Partial Correctness and Predicative Semantics 15
        6.1.1. Considering only Finite States 15
        6.1.2. Connection Between the Two Semantics 16
    6.2. Robust Correctness and Predicative Specifications 17
        6.2.1. Considering only Total States 18
7. On Nontermination 19
8. Conclusions 19
References 20

1. Introduction

The purpose of this paper is to clarify the relationship between two distinct styles of formally defining the semantics of a programming language: the denotational style and the predicative style. Of the two, the denotational style has been around longer [9] and is better understood; the predicative style is more recent [5, 6]. We shall compare the two styles on a specific language that is very similar to Hehner's language of communicating processes [5].

Our treatment is composed of three parts:

(i) basic concepts and definitions (Sects. 2 and 3),

(ii) deterministic programs (Sect. 4), and

(iii) nondeterministic programs (Sect. 5).

The correspondence between the denotational and predicative semantics of deterministic programs will be quite straight-forward. Essentially, the denotational definition will be based on the λ-calculus, while the predicative definition will be based in first-order predicate logic. For the description of repetitive constructs, we will offer two alternative induction schemes that will entail alternative correctness proof techniques. The two schemes are based on computational induction and fixed point induction, respectively.

In the presence of nondeterminism, the choice of induction scheme will be crucial and will depend on the notion of correctness that we are interested in. We will distinguish three notions of correctness [3]: partial correctness, robust correctness, and total correctness. *Partial correctness* specifies a superset of the program's possible input-output behaviors; it considers the question whether a program produces just correct answers, if any. *Robust correctness* (which is traditionally called "total correctness") describes a subset[1] of the program's possible input-output behaviors; it considers the question what outcomes of the program are guaranteed. Finally, (what we call) *total correctness* describes the precise set of the program's possible input-output behaviors.

2. Syntax

We propose a simple language of communicating programs. The BNF-style syntax of our language is defined as follows:

<stat>	::=	<pvi> := <exp>	(assignment)
	|	**if** <exp> **then** <stat> **else** <stat> **fi**	(alternation)
	|	<ici> ? <pvi>	(input)
	|	<oci> ! <exp>	(output)

[1] Actually, one usually takes the upward closure of this subset [8].

```
|   <stat> ; <stat>                  (sequential composition)
|   <stat> || <stat>                 (parallel composition)
|   <stat> □ <stat>                  (nondeterministic choice)
|   chan <ici> ← <oci> : <stat>      (channel connection)
|   <prog-id>                        (refinement call)
|   <prog-id> :: <stat>              (refinement definition)
```

Nonterminals are taken from the following disjoint sets of identifiers:

- PVI is the set of identifiers <pvi> representing program variables,
- ICI is the set of identifiers <ici> for input channels,
- OCI is the set of identifiers <oci> for output channels, and
- PROG-ID is the set of identifiers <prog-id> for programs.

For convenience, we introduce the set ID of identifiers:

$$\text{ID} \quad = \quad \text{PVI} \cup \text{ICI} \cup \text{OCI} \ .$$

The language of expressions, EXP (the sentences derived from nonterminal <exp>), is assumed as given. STAT refers to the set of sentences derived from the start symbol of the grammar, <stat>, i.e., to the entire programming language. Parallel composition must obey Hehner's context condition [5]: the identifiers from PVI that denote assigned variables in the two statements of the composition (i.e., that appear on the left side of assignments or in input statements) and the set of channels, i.e., identifiers from ICI $\cup$ OCI must be mutually disjoint.

3. Program States

States are mappings from identifiers to values. We assume a given set, D, of values that does not contain $\perp$. Value $\perp$ denotes undefinedness. We write $D^\perp$ for set D with the additional element $\perp$. For elements d1, d2 $\in D^\perp$, the partial ordering $\sqsubseteq$ is defined by:

$$\text{d1} \sqsubseteq \text{d2} \quad \Leftrightarrow \quad \text{d1} = \text{d2} \ \vee \ \text{d1} = \perp$$

The history of communications on a channel is modelled by a finite or infinite sequence of messages called a *stream*. We write D^* for the set of finite sequences of D elements, and D^∞ for the set of infinite sequences of D elements. The set of streams is defined as:

$$\text{STREAM(D)} \quad = \quad (D^* \times \{\perp\}) \cup D^* \cup D^\infty \ .$$

A (finite) stream in D^* represents a history in which the sender terminates. A (finite) stream in $D^* \times \{\perp\}$ represents a history in which the sender diverges after having sent a finite number of messages (and where it is not known whether the communication will be continued). A (infinite) stream in D^∞ represents a history in which the sender does not terminate but generates an infinite number of messages.

Our language distinguishes input and output channels. An input channel is broken if its value is stream $\langle\perp\rangle$. An output channel is broken if its value is a stream that ends with $\perp$. Streams that end with $\perp$ are called *partial*. All other streams are called *total*. For streams s1, s2 $\in$ STREAM(D), the partial ordering $\sqsubseteq$ is defined by:

$$s1 \sqsubseteq s2 \quad \Leftrightarrow \quad s1 = s2 \lor (\exists\, s3 \in D^*, s4 \in STREAM(D){:}\ (s1 = s3\frown<\perp>) \land (s2 = s3\frown s4))$$

Here, $\frown$ denotes concatenation, and <a> denotes the one-element sequence consisting of just the value a. We use the following operations on streams:

$$\begin{array}{ll} \&: & D^{\perp} \times STREAM(D) \rightarrow STREAM(D) \\ first: & STREAM(D) \rightarrow D^{\perp} \\ rest: & STREAM(D) \rightarrow STREAM(D) \end{array}$$

They are defined by the equations:

$$a \,\&\, s = \begin{cases} <a>\frown s & \text{if } a \neq \perp \\ <\perp> & \text{if } a = \perp \end{cases}$$

$$first.\epsilon = \perp$$

$$first.(a \,\&\, s) = a$$

$$rest.\epsilon = <\perp>$$

$$rest.(a \,\&\, s) = \begin{cases} s & \text{if } a \neq \perp \\ <\perp> & \text{if } a = \perp \end{cases}$$

We write s_i for $first.rest^i.s$, where $rest^0.s = s$, and $rest^{i+1}.s = rest.rest^i.s$. In addition, we define the special left-strict concatenation operation on streams:

$$@{:}\quad STREAM(D) \times STREAM(D) \rightarrow STREAM(D)$$

by the following equations:

$$\begin{array}{lll} \epsilon \;@\; s2 & = & s2 \\ (a \,\&\, s1) \;@\; s2 & = & a \,\&\, (s1 \;@\; s2) \quad \text{for } a \in DATA^{\perp} \\ <\perp> \;@\; s2 & = & <\perp> \end{array}$$

That is, if s1 is a partial or infinite stream then, for all streams s2, s1 @ s2 = s1 and, if s1 is a finite total stream, then s1 @ s2 = s1$\frown$s2.

We distinguish between terminating and nonterminating programs. *Terminating* programs produce finite total streams as output. *Nonterminating* programs produce infinite or partial streams as output. If all output streams of a nonterminating program are partial, we speak of *divergence*.

Our domain of program states is tailored for a strict semantics - either all or none of the output streams are finite and total:

$$\text{STATE} = \{\sigma : \text{ID} \rightarrow D^{\perp} \cup \text{STREAM(D)}:$$
$$[\ (\forall x \in \text{PVI} \cup \text{ICI}: \sigma.x = \perp)\ \wedge \quad \text{(nontermination)}$$
$$(\forall x \in \text{OCI}: \sigma.x \in (D^* \times \{\perp\}) \cup D^{\infty})\ ]\ \vee$$
$$[\ (\forall x \in \text{PVI}: \sigma.x \in D)\ \wedge \quad \text{(termination)}$$
$$(\forall x \in \text{ICI}: \sigma.x \in (D^* \times \{\perp\}) \cup D^{\infty})\ \wedge$$
$$(\forall x \in \text{OCI}: \sigma.x \in D^*)\ ]\}$$

In the case of nontermination, the "final" states of all program variables and input streams are undefined and all output streams are either infinite or partial. Since our language does not provide a means of testing whether an input channel is empty, i.e., whether no input will be supplied ever more, there is no point in distinguishing partial and total input streams. We require input streams to be either partial or infinite. A state indicates termination if all program variables are defined, all output streams are total, and input streams are either infinite or partial.

When modelling a program's execution, we shall model nontermination by "smashing" the program state. The postfix operator $\downarrow$ indicates a smashed state:

$$\sigma\!\downarrow\!.x = \begin{cases} \sigma.x \ @ <\perp> & \text{if } x \in \text{OCI} \\ \perp & \text{if } x \in \text{PVI} \cup \text{ICI} \end{cases}$$

In smashing, only the output produced so far is kept. All other information is destroyed. Smashing is idempotent, i.e., $\sigma\!\downarrow\downarrow = \sigma\!\downarrow$. A predicate NT (for NonTermination) recognizes a smashed state:

$$\text{NT}.\sigma = (\forall x \in \text{PVI} \cup \text{ICI}: \sigma.x = \perp) \wedge (\forall x \in \text{OCI}: x \notin D^*)$$

We denote a pointwise state change as usual: given a state σ, a value d, and an identifier x, $\sigma[d/x]$ represents the state that maps x on d and every other identifier y on $\sigma.y$. Formally, for $x \in \text{ID}$:

$$\sigma[d/x].y = \begin{cases} \sigma.y & \text{if } x \neq y \ \wedge\ (x \in \text{PVI} \Rightarrow d \neq \perp)\ \wedge\ (x \in \text{OCI} \Rightarrow d \in D^*) \\ d & \text{if } x = y \\ \sigma\!\downarrow\!.y & \text{otherwise} \end{cases}$$

By this definition, $\sigma[d/x]$ is a smashed state whenever d is $\perp$ or a partial stream.

For states $\sigma 1$, $\sigma 2$, the partial ordering $\sqsubseteq$ is defined elementwise:

$$\sigma 1 \sqsubseteq \sigma 2 \Leftrightarrow (\forall x \in \text{ID}: \sigma 1.x \sqsubseteq \sigma 2.x)$$

4. Deterministic Programs

Initially, we describe only deterministic programs, i.e., we do not consider programs that contain the choice operator $\Box$. Following [5], one distinctive property of our language is that channel communication, per se, does not induce nondeterminism.

4.1. Denotational Semantics

Our denotational semantics represents a deterministic program by a function from states to states. $[X \rightarrow X]$ denotes the set of continuous functions from X to X, and N denotes the set of natural numbers:

$$\text{PROG-FUNC} \;=\; \{f \in [\text{STATE} \rightarrow \text{STATE}]: (\forall\, \sigma \in \text{STATE}: (\forall\, c \in \text{ICI}: (\exists\, i \in N \cup \{\infty\}: \text{rest}^i.\sigma.c = f.\sigma.c) \;\wedge\; (\forall\, d \in \text{OCI}: (\exists\, s \in \text{STREAM}(D): \sigma.d \;@\; s = f.\sigma.d))\}$$

where the superscript on rest is, again, functional iteration and $\text{rest}^{\infty}.s = \perp$. If the program terminates then, for input channel c, the stream f.σ.c, which is a postfix of σ.c, denotes the input values not consumed by the program and, for output channel d, the stream f.σ.d, for which σ.d is a prefix, contains the values added by the program to d. All elements of PROG-FUNC only take from the input streams and add to the output streams.

For functions f1, f2 ∈ PROG-FUNC, the partial ordering $\sqsubseteq$ is defined elementwise:

$$f1 \sqsubseteq f2 \quad \Leftrightarrow \quad (\forall\, \sigma \in \text{STATE}: f1.\sigma \sqsubseteq f2.\sigma)$$

Functions that represent programs are monotonic in the sense that an extension of the input streams in σ implies an extension of the output streams in f.σ.

An environment associates program functions with program identifiers. As we shall explain later (in Sect. 5.2), nondeterministic programs will be modelled by sets of functions, not by relations. In view of that, the environment maps to the (non-empty) powerset of PROG-FUNC:

$$\text{ENV} \;=\; [\text{PROG-ID} \rightarrow (\mathcal{P}(\text{PROG-FUNC}) - \{\emptyset\})] \;.$$

We assume as given a function, V, that gives meaning to expressions:

$$V\!: \;\; \text{EXP} \rightarrow \text{STATE} \rightarrow D^{\perp} \;.$$

The strictness of V translates into the requirement: $V[\![E]\!].\sigma = \perp$ if NT.σ. We define the semantics of statements in our language by the meaning function:

$$B\!: \;\; \text{STAT} \rightarrow \text{ENV} \rightarrow (\mathcal{P}(\text{PROG-FUNC}) - \{\emptyset\}) \;.$$

$B_\delta[\![S]\!]$ is defined by induction on the structure of S. Since $B_\delta[\![S]\!]$ is a set of functions, the definition of $B_\delta[\![S]\!]$ is of the form:

$$B_\delta[\![S]\!] \;=\; \{\lambda\, \sigma\!: h.f.\sigma\!: f \in B_\delta[\![S0]\!]\}$$

where S0 represents the subcomponents S is composed of and h represents the way they are composed. We shall instead write more concisely:

$$B_\delta[\![S]\!] \;=\; (\lambda\, \sigma\!: h.B_\delta[\![S0]\!].\sigma) \;.$$

If S is deterministic, δ is a one-element set, and so is $B_\delta[\![S]\!]$. We will now give define the meaning of deterministic programs. Let x be a program variable, c an input channel variable, and d an output channel variable. For δ ∈ ENV:

$B_\delta[\![x := E]\!] \quad = \quad (\lambda\ \sigma\colon \sigma[V[\![E]\!].\sigma/x])$

$B_\delta[\![\textbf{if}\ E\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{fi}]\!] \quad = \quad (\lambda\ \sigma\colon \mathrm{IF}.(V[\![E]\!].\sigma,\ B_\delta[\![S1]\!].\sigma,\ B_\delta[\![S2]\!].\sigma))$

$$\text{where} \qquad \mathrm{IF}.(b.\sigma,\ f.\sigma,\ g.\sigma) = \begin{cases} f.\sigma & \text{if } b.\sigma \\ g.\sigma & \text{if } \neg b.\sigma \\ \sigma\!\downarrow & \text{if } b.\sigma = \bot \end{cases}$$

$B_\delta[\![c\ ?\ x]\!] \quad = \quad (\lambda\ \sigma\colon \sigma[\mathrm{first}.\sigma.c/x,\ \mathrm{rest}.\sigma.c/c])$

$B_\delta[\![c\ !\ E]\!] \quad = \quad (\lambda\ \sigma\colon \sigma[(\sigma.d\ @\ \langle V[\![E]\!].\sigma\rangle)/d])$

$B_\delta[\![S1\ ;\ S2]\!] \quad = \quad (\lambda\ \sigma\colon B_\delta[\![S2]\!].B_\delta[\![S1]\!].\sigma)$

$B_\delta[\![S1\ ||\ S2]\!] \quad = \quad (\lambda\ \sigma\colon \mathrm{par}.(B_\delta[\![S1]\!],\ B_\delta[\![S2]\!],\ \sigma))$

$$\text{with} \qquad \mathrm{par}.(f1,\ f2,\ \sigma) = \begin{cases} f2.f1.\sigma & \text{if } \neg NT.f1.\sigma \\ f1.f2.\sigma & \text{if } \neg NT.f2.\sigma \\ \mathrm{join}.(f1.\sigma,\ f2.\sigma) & \text{if } NT.f1.\sigma \wedge NT.f2.\sigma \end{cases}$$

where join is defined, on smashed states,

$$\text{by} \qquad \mathrm{join}.(\sigma1,\ \sigma2).x = \begin{cases} \sigma1.x & \text{if } \sigma2.x \sqsubseteq \sigma1.x \\ \sigma2.x & \text{if } \sigma1.x \sqsubseteq \sigma2.x \end{cases}$$

If one operand of the parallel composition terminates but the other does not, we may derive the output state of the terminating computation first and apply to it the nonterminating computation. If both operands do not terminate, the more defined operand determines the final state of the parallel composition. When both operands terminate, their order of application is irrelevant. This is guaranteed by the context condition that we imposed on parallel composition (Sect. 2). Thus, when several cases apply simultaneously, the alternative definitions coincide. Remember that S1 and S2 are required to have no update conflicts. Therefore we know, in the case of termination, i.e., when $\neg NT.B_\delta[\![S1]\!].\sigma$ and $\neg NT.B_\delta[\![S2]\!].\sigma$, that $B_\delta[\![S1]\!].B_\delta[\![S2]\!].\sigma = B_\delta[\![S2]\!].B_\delta[\![S1]\!].\sigma$.

$B_\delta[\![\textbf{chan}\ c \leftarrow d\colon S]\!] \quad = \quad (\lambda\ \sigma\colon \mathrm{feedback}_{c,d}.B_\delta[\![S]\!].\sigma)$

$$\text{where} \qquad \mathrm{feedback}_{c,d}.f.\sigma = \begin{cases} \sigma2[\sigma.c/c,\ \sigma.d/d] & \text{if } \neg NT.\sigma2 \\ \sigma2[(\sigma.d\ @\ \langle\bot\rangle)/d] & \text{if } NT.\sigma2 \end{cases}$$

$$\text{with} \qquad \sigma2 \quad = \quad (\mu\ \sigma1 \in \mathrm{STATE}\colon f.\sigma[\sigma1.d/c,\ \epsilon/d])$$

Here, $(\mu\ f \in M\colon G.f)$ denotes the least fixed point of function G in set M. Since all our language constructs are monotonic, the fixed point exists and is unique. Channel connection feeds output channel d of programs S into input channel c. That is, in the input state of program S, c coincides with the output produced by S on d. This "feedback" does not affect the values of c and d.

Here is why channel connection must be a fixed point and why $\sigma2$ is the right one. Say $B_\delta[\![S]\!]$ is the one-element function set $\{f\}$. The following sequence of states models communications along channel connection $c \leftarrow d$:

$$\sigma_0 \quad = \quad \sigma[\langle\bot\rangle/c,\ \epsilon/d]$$

$$\sigma_{i+1} = f.\sigma_i[(f.\sigma_i.d @ <\bot>)/c, \epsilon/d]$$

State σ_i models the i-fold feedback along $c \leftarrow d$. In particular, the state σ_0 is the result of applying f to σ with stream $<\bot>$ for input channel c and the empty stream ϵ for output channel d. State σ_1 is obtained by applying f to σ_0 with stream $\sigma_0.d$ for c and ϵ for d, and so on. A fixed point is reached by this sequence of applications of f if the output stream $\sigma_i.d$ is no longer increased, even not by substituting $\sigma_i.d$ for c. By continuity of f (guaranteed by our choice of language constructs), the fixed point, the limit of this application sequence, exists and is $\sigma 2$.

$$B_\delta[\![p :: S]\!] = (\mu\ f \in \text{PROG-FUNC}: B_{\delta[f/p]}[\![S]\!])$$

$$B_\delta[\![p]\!] = \delta.p$$

With the context condition for parallel composition, all programs in our language are deterministic.

The denotational semantics given above reflects the following design decisions:

(1) Smashed states remain smashed. I.e., for a state σ with $NT.\sigma$, it is ensured that $B[\![S]\!].\sigma = \sigma$. This is a simple consequence of the strictness of V and the definition of $\sigma[d/x]$ for partial streams d.

(2) All language constructs are strict - also the parallel composition S1||S2. If S1 does not terminate for an input state $\sigma 1$, then $\sigma 2 = B_\delta[\![S1]\!].\sigma 1$ is a smashed state and $B_\delta[\![S2]\!].\sigma 2 = \sigma 2$. If one of the operands S1 and S2 in the parallel composition S1||S2 diverges, so does S1||S2.

Following Hehner [5], and contrary to other approaches, parallel composition and channel connection are independent concepts and are represented by distinct operators.

4.2. Predicative Semantics

The second definition of our programming language is in terms of predicates, i.e., relations between states, not functions from states to states. A *predicative relation* on the states σ‘, σ’ is an element of

$$\begin{aligned} \text{PRED} = \{p \subseteq \text{STATE} \times \text{STATE}: (\forall\ \sigma‘, \sigma’ \in \text{STATE}: p.(\sigma‘, \sigma’) \Rightarrow \\ (\forall\ c \in \text{ICI}: (\exists\ i \in \mathbb{N}\cup\{\infty\}: \text{rest}^i.\sigma‘.c = \sigma’.c)) \quad \wedge \\ (\forall\ d \in \text{OCI}: (\exists\ s \in \text{STREAM}(D): \sigma‘.d\ @\ s = \sigma’.d))\} \end{aligned}$$

A state-transition function can be viewed as a special case of a predicative relation. A *predicative specification* is a first-order formula that represents a predicative relation. Technically, a predicative specification is a formula equivalent to the proposition p.(σ‘, σ’) where the relation, p, is defined as: We call σ‘ (read: "sigma in") the *input state* and σ’ (read: "sigma out") the *output state*. Following Hehner [5], we equip references to variables in the predicative specification with the according apostrophe, depending whether they refer to the variable's value in σ‘ or σ’. For variable x, we write x‘ for σ‘.x and x’ for σ’.x. For expression E, we write E‘ for $V[\![E]\!].\sigma$‘ and E’ for $V[\![E]\!].\sigma$’. We also write def.E‘ for $V[\![E]\!].\sigma$‘ $\neq \bot$ and NT’ for NT.σ’. #c denotes the length of the stream for channel c, i.e., with $a \in D$:

$$\#\bot = \#\epsilon = 0,$$
$$\#(a\ \&\ r) = 1 + \#r,$$

$\#r = \infty \qquad$ for $r \in D^{\infty}$.

Let LOG-ID be a set of logical identifiers disjoint from all other sets of identifiers. The following function maps programs to predicates on the input state σ‘ and the output state σ’:

PS: STAT → PRED

Before we define PS, let us, again, talk about smashed states. Let P be a predicative specification with free identifiers σ‘ and σ’. Then P↓ is the predicative specification:

$$P{\downarrow} \;=\; NT' \wedge (\forall\, d \in OCI, \sigma \in STATE: P[\sigma/\sigma'] \Rightarrow (d' = \sigma.d \;@\; <\perp>)) .$$

The definition of PS follows Hehner's predicative specifications [5]:

$$PS[\![x := E]\!] \;=\; (\neg def.E‘ \wedge \sigma' = \sigma‘{\downarrow}) \vee (def.E‘ \wedge \sigma' = \sigma‘[E‘/x])$$

$$PS[\![\textbf{if } E \textbf{ then } S1 \textbf{ else } S2 \textbf{ fi}]\!] \;=\; (\neg def.E‘ \wedge \sigma' = \sigma‘{\downarrow}) \vee (E‘ \wedge PS[\![S1]\!]) \vee (\neg E‘ \wedge PS[\![S2]\!])$$

$$PS[\![c\ ?\ x]\!] \;=\; (\#c‘ = 0 \wedge \sigma' = \sigma‘{\downarrow}) \vee (\#c‘ > 0 \wedge \sigma' = \sigma‘[first.c‘/x,\ rest.c‘/c])$$

$$PS[\![d\ !\ E]\!] \;=\; (\neg def.E‘ \wedge \sigma' = \sigma‘{\downarrow}) \vee (def.E‘ \wedge \sigma' = \sigma‘[(d \;@\; <E‘>)/d])$$

$$PS[\![S1\ ;\ S2]\!] \;=\; (NT' \wedge PS[\![S1]\!]) \vee (\exists\, \sigma \in STATE: (PS[\![S1]\!] \wedge \neg NT')[\sigma/\sigma'] \wedge PS[\![S2]\!][\sigma/\sigma‘])$$

$$PS[\![S1\ ||\ S2]\!] \;=\; \begin{array}{l} (PS[\![S1]\!] \wedge PS[\![S2]\!] \wedge \neg NT') \vee \\ (PS[\![S1]\!] \wedge PS[\![S2]\!]{\downarrow} \wedge NT') \vee \\ (PS[\![S2]\!] \wedge PS[\![S1]\!]{\downarrow} \wedge NT') \end{array}$$

Channel connection and refinement definition contain recursion. In denotational semantics, recursive definitions are phrased as fixed point equations. As long as the functionals that define the meaning of the language are continuous, we can interpret a recursive definition in two semantically equivalent ways:

- as the least upper bound of functional iteration, i.e., by *computational induction*, and
- by the least fixed point of the respective functional, i.e., by *fixed point induction*.

Both techniques can be used as a basis for a predicative semantics, but they lead to rather distinct formulas.

Channel Connection by Computational Induction:

Let a, b, e ∈ LOG-ID be pairwise distinct:

$$S0 \;=\; (\exists\, a: PS[\![S]\!][\epsilon/d‘,\ b/d',\ a/c'])$$

$$P_0 \;=\; S0[<\perp>/c‘]$$

$$P_{i+1} \;=\; (\exists\, e: (\exists\, \sigma: P_i[e/b][\sigma/\sigma'] \wedge PS[\![S]\!][e/c‘]))$$

P_i is a predicate that depends on σ‘, σ’, and on the logical identifier b. It indicates what result b is available on d’ after i functional iterations (i.e., i times identifying output stream d’ with input stream c‘). Based on P_i, we define:

$$PS[\![\textbf{chan } c \leftarrow d\!: S]\!] \;=\; \begin{array}{l} (\neg NT' \wedge (\exists\, i \in \mathbf{N}: (\exists\, b: P_i \wedge PS[\![S]\!][b/c‘]))) \vee \\ (NT' \wedge (\forall\, y \in OCI,\ j \in \mathbf{N}: \\ \qquad (\forall\, i \in \mathbf{N}: (\exists\, \sigma, b: P_i[\sigma/\sigma'] \wedge \sigma.y_j \sqsubseteq y_j')) \wedge \\ \qquad (\exists\, i \in \mathbf{N}: (\exists\, \sigma, b: P_i[\sigma/\sigma'] \wedge \sigma.y_j = y_j')))) \end{array}$$

The two disjuncts cover the cases of termination and nontermination. In the case of termination (the first disjunct), the approximation sequence $(P_i)_{i \in N}$ becomes constant at some point. In the case of nontermination, the approximation sequence may never assume its least upper bound.

Channel Connection by Fixed Point Induction:

Let x0, y0, x1, y1 ∈ LOG-ID be pairwise distinct and not free in PS⟦S⟧:

$$PS[\![\textbf{chan } c \leftarrow d\text{: } S]\!] \;=\; (\exists\, x0, y0\text{: } Q.(x0, y0) \wedge (\forall\, x1, y1\text{: } Q.(x1, y1) \Rightarrow x0 \sqsubseteq x1 \wedge y0 \sqsubseteq y1))$$

$$\text{where} \quad Q.(x, y) \;\Leftrightarrow\; PS[\![S]\!][y/c`, \epsilon/d`, x/c', y/d']\ .$$

The first conjunct expresses that PS⟦**chan** c ← d: S⟧ is a fixed point; the second conjunct expresses that it is the least fixed point.

The definition of channel connection in [7] is essentially based on fixed point properties:

$$PS[\![\textbf{chan } c \leftarrow d\text{: } S]\!] \;=\; (\exists\, c', d'\text{: } PS[\![S]\!][d'/c`, \epsilon/d`])$$

The right side of the equation corresponds to the formula:

$$(\exists\, x, y\text{: } PS[\![S]\!][y/c`, \epsilon/d`, x/c', y/d'])$$

which is equivalent to the equational formula:

$$(\exists\, x0, x1, y0, y1\text{: } ((c` = d') \wedge (d` = \epsilon) \wedge PS[\![S]\!])[y0/c`, x0/d`, y1/c', x1/d'])$$

Note the similarity with the left conjunct of our fixed point semantics for **chan** c ← d: S. However, the right conjunct of our definition, i.e., the least fixed point property is not expressed here.

The predicative specification of refinement definition can, again, be based on either computational induction or fixed point induction. Again, the definitions proceed by the previous case analysis.

Refinement Definition by Computational Induction:

With auxiliary definitions:

$$p_0 \;=\; (\sigma' = \sigma`\!\downarrow)$$

$$p_{i+1} \;=\; PS[\![S]\!][p_i/PS[\![p]\!]]$$

we define:

$$\begin{aligned} PS[\![p :: S]\!] \;=\; & (\exists\, i \in \mathbf{N}\text{: } p_i \wedge \neg NT') \vee \\ & (NT' \wedge (\forall\, y \in OCI, j \in \mathbf{N}\text{:} \\ & \quad (\forall\, i \in \mathbf{N}\text{: } (\exists\, \sigma, b\text{: } p_i[\sigma/\sigma'] \wedge \sigma.y_j \sqsubseteq y_j')) \wedge \\ & \quad (\exists\, i \in \mathbf{N}\text{: } (\exists\, \sigma, b\text{: } p_i[\sigma/\sigma'] \wedge \sigma.y_j = y_j'))\ )) \end{aligned}$$

Refinement Definition by Fixed Point Induction:

For the fixed point semantics, we introduce identifiers for relations on states:

$$\begin{aligned} PS[\![p :: S]\!] \;=\; & (\exists\, q \in PRED\text{: } q.(\sigma`, \sigma') \wedge Q.q \wedge \\ & \quad (\forall\, q1 \in PRED\text{: } Q.q1 \Rightarrow \\ & \quad\quad (\forall\, \sigma1, \sigma2 \in STATE\text{: } q1.(\sigma1, \sigma2) \Rightarrow \\ & \quad\quad\quad (\exists\, \sigma3 \in STATE\text{: } (\sigma3 \sqsubseteq \sigma2 \wedge q.(\sigma1, \sigma3)))) \end{aligned}$$

where Q.q is specified by

$(\forall\ \sigma`, \sigma'$: q.$(\sigma`, \sigma') \Leftrightarrow$ PS⟦S⟧[q.$(\sigma`, \sigma')$/PS⟦p⟧]) .

Again, the first conjunct expresses that PS⟦p :: S⟧ is a fixed point; the second conjunct expresses that it is the least fixed point. Compare this definition of p :: S with the denotational one. Substitution is expressed in the λ-calculus much more gracefully than in the predicate calculus.

We do not explicitly state the predicative semantics of the refinement call. For a program identifier p, the predicative semantics PS⟦p⟧ can be understood as an identifier for a predicative relation.

4.3. Connection Between the Two Semantics

The following theorem states the consistency of our denotational and predicative semantics.

Theorem: (Consistency Theorem)

For all states $\sigma`$ and σ', and for all programs S in our programming language,

$$\sigma' = B_\delta[\![S]\!].\sigma` \quad \Leftrightarrow \quad PS[\![S]\!] \ .$$

The proof is deferred to an appendix. It proceeds by structural induction on S.

A predicative specification is a logical formula representing a predicate on states, namely, the input state $\sigma`$ and the output state $\sigma'$. Let the predicative formula p.$(\sigma`, \sigma')$, where we write x` for $\sigma`$.x and x' for σ'.x, stand for a predicative specification, and let S be a program; we define a satisfaction relation, **sat**, on programs and specifications:

S **sat** p.$(\sigma`, \sigma')$ $=$ $(\forall\ \sigma \in$ STATE: p.$(\sigma, B_\delta[\![S]\!].\sigma))$

It follows immediately from the Consistency Theorem that a program satisfies its own predicative specification, i.e., S **sat** PS⟦S⟧. With our translation of programs to predicative specifications, the formula S **sat** p.$(\sigma`, \sigma')$ is equivalent to:

PS⟦S⟧ $\Rightarrow$ p.$(\sigma`, \sigma')$.

We say then: program S is *correct* with respect to (is a *correct implementation* of) predicative specification p.$(\sigma`, \sigma')$. Trivially, **sat** can be extended to a relation between specifications:

p0.$(\sigma`, \sigma')$ **sat** p1.$(\sigma`, \sigma')$ $=$ p0.$(\sigma`, \sigma') \Rightarrow$ p1.$(\sigma`, \sigma')$.

Often, one is only interested in particular satisfaction relations such as those of partial, robust, and total correctness. Brief definitions of the notions of partial, robust, and total correctness have been provided in the introduction (Sect. 1). For more precise definitions, see [3]. As an example, consider the program

S :: x:=1 $\square$ **abort** .

The statement **abort** denotes nonterminating programs (without any free output channels). For example, **abort** might stand for the program p :: p, or the program **chan** c $\leftarrow$ d: c?x. Partial correctness gives program S the behavior of x:=1, robust correctness that of **abort**, and total correctness either of the two.

Examples:

Let the set ID contain program variables x and y, input channel c, and output channel d.

(1) The following program may execute infinitely without producing any output.

p :: **if** odd(x) **then** d!x **else** p **fi** **sat**
$(odd(x') \Rightarrow (\neg NT' \wedge (x' = x') \wedge (y' = y') \wedge (c' = c') \wedge (d' = d' \mathbin{@} \langle x' \rangle))) \wedge$
$(even(x') \Rightarrow (NT' \wedge (d' = d' \mathbin{@} \langle \bot \rangle)))$

(2) The following program, when executing infinitely, will produce an infinite sequence of outputs.

p :: **if** odd(x) **then** d!x **else** d!x ; p **fi** **sat**
$(odd(x') \Rightarrow (\neg NT' \wedge (x' = x') \wedge (y' = y') \wedge (c' = c') \wedge (d' = d' \mathbin{@} \langle x' \rangle))) \wedge$
$(even(x') \Rightarrow (NT' \wedge (\exists\, t \in STREAM(D){:}\ (t = x' \,\&\, t) \wedge (d' = d' \mathbin{@} t))))$

Predicative specifications equate a program with a logical formula. While the program is concise and easy to read, the logical formula is precise and hard to read. The program notation is, more implicit, for human consumption; the logical notation is, more explicit, for formal reasoning.

4.4. Partial Correctness and Predicative Semantics

For the partial correctness of deterministic programs, given the predicative specification p.(σ', σ'), we need only consider the weaker predicate $(\exists\ \sigma{:}\ \sigma' \sqsubseteq \sigma \wedge p.(\sigma', \sigma))$. This defines, trivially, a predicative semantics suitable for partial correctness proofs. Since we add with every state σ such that p.(σ',σ) all states σ' such that $\sigma' \sqsubseteq \sigma$, we speak of a "downward closure" of the set of final states.

Partial correctness expresses safety properties: a program is partially correct with respect to a specification if it does not produce output that is incorrect with respect to the specification. However, the program may not be guaranteed to produce output (other than $\bot$) at all. We define accordingly: a deterministic program S is *partially correct* with respect to a predicative specification p.(σ', σ') if

$$(\forall\ \sigma', \sigma' \in STATE{:}\ (\sigma' = B_\delta[\![S]\!].\sigma') \Rightarrow (\exists\ \sigma \in STATE{:}\ \sigma' \sqsubseteq \sigma \wedge p.(\sigma', \sigma)))\ ,$$

or, equivalently,

$$(\forall\ \sigma' \in STATE{:}\ (\exists\ \sigma \in STATE{:}\ B_\delta[\![S]\!].\sigma' \sqsubseteq \sigma \wedge p.(\sigma', \sigma)))\ .$$

We suggest the shorthand: S $\mathbf{sat}_{pc}$ p.(σ', σ').

4.5. Robust Correctness and Predicative Specifications

A program S is called *robustly correct* with respect to a predicative specification p.(σ', σ') if

$$(\forall\ \sigma', \sigma' \in STATE{:}\ (\sigma' = B_\delta[\![S]\!].\sigma') \Rightarrow (\exists\ \sigma \in STATE{:}\ \sigma \sqsubseteq \sigma' \wedge p.(\sigma', \sigma)))\ ,$$

or, equivalently,

$$(\forall\ \sigma' \in STATE{:}\ (\exists\ \sigma \in STATE{:}\ \sigma \sqsubseteq B_\delta[\![S]\!].\sigma' \wedge p.(\sigma', \sigma)))\ .$$

We suggest the shorthand: S $\mathbf{sat}_{rc}$ p.(σ', σ').

5. Nondeterministic Programs

Before we extend our semantic definition to incorporate nondeterministic choice, let us point out a severe problem that arises. Consider the three programs:

S1 :: a!1 ; a!1 ; **abort**

S2 :: a!1 ; **abort**

S3 :: **abort**

Our deterministic predicative semantics distinguishes these three programs in all three cases: partial, robust, and total correctness.

Let us now consider nondeterministic programs:

S4 :: S1 □ S3

S5 :: S1 □ S2 □ S3

S4 either produces no output and diverges, or it produces two 1's on output channel a and diverges then. S5 may, in addition, diverge after output of only one 1 on channel a. We want to explore our options of predicative semantics that distinguish S4 and S5 from each other and from S1, S2, and S3.

We shall propose several distinct semantic models for our language that aim at different issues and lead to different ways of distinguishing S4 and S5.

5.1. Denotational Semantics

If S is a nondeterministic program, environment δ is a multi-element set of functions, and so is $B_\delta[\![S]\!]$. Let us first explain, why we must represent nondeterministic programs by sets of functions and not by relations (i.e., set-valued functions). Consider the following simple situation:

C :: c?x
A :: d!1 ; a!1
B :: d!1 ; b!1

P1 :: A ; C
P2 :: B ; C

Q :: (**chan** c ← d: P1 □ P2)

For compositionality, it is highly desirable that choice distribute over recursion:

(**chan** c ← d: P1 □ P2) = (**chan** c ← d: P1) □ (**chan** c ← d: P2)

This precludes behaviors that mix steps of both recursions like, in our example, behaviors that exhibit communications on both a and b. However, a definition by relations or, equivalently, by set-valued functions permits such behaviors. Set-valued functions carry less information than sets of functions. The use of sets of functions instead of set-valued functions will avoid the combination of nondeterministic alternatives that must belong to distinct computations.

We proceed with the definition of meaning function B in the face of nondeterminism. For all language constructs but one, B remains defined as before. Only the meaning of refinement definition must be revised to the least fixed point of the following equation, as described in [2]:

$$B_\delta[\![p :: S]\!] \quad = \quad \{f \in F: F = B_{\delta[F/p]}[\![S]\!]\}$$

The sense in which fixed point F is "least" is not easily formulated. This reflects the inherent problems in the connection of nondeterminism with recursion.

We also add the choice construct:

$$B_\delta[\![S1 \;\Box\; S2]\!] \quad = \quad B_\delta[\![S1]\!] \cup B_\delta[\![S2]\!]$$

5.2. Predicative Semantics

The first definition of nondeterministic choice that comes to mind is:

$$PS[\![S1 \;\Box\; S2]\!] \quad = \quad PS[\![S1]\!] \vee PS[\![S2]\!] \;.$$

Unfortunately this definition creates problems in the presence of recursion. The reason is that our predicative definition specifies a set-valued function from STATE to $\mathcal{P}$(STATE), not a set of functions from STATE to STATE (see previous section).

Brock and Ackerman [1] drew attention to this problem. The following program replicates their by now famous "anomaly" (see Fig. 1) in our programming language:

```
D ::  c?a ; eo!a ; eo!a ; D
Merge ::  (ei?x ; zo!x) □ (vi?x ; zo!x) ; Merge
P1 ::  zi?y1 ; DO
    where  DO ::  zi?y2 ; wo!y1 ; d!y1 ; y1:=y2 ; DO
P2 ::  zi?y ; wo!y ; d!y ; P2
Plus1 ::  wi?z ; z:=z+1 ; vo!z ; Plus1
```

Now define for k = 1,2:

Rk :: (**chan** ei ← eo: (**chan** zi ← zo: D || Merge || Pk))

Qk :: (**chan** wi ← wo: (**chan** vi ← vo: Rk || Plus1))

With the previous definition of □, PS⟦R1⟧ = PS⟦R2⟧, but PS⟦Q1⟧ ≠ PS⟦Q2⟧: if we assume, for instance, c‘ = 1&<⊥>, then d’ = 1&2&... is a behavior of Q2 but not of Q1.

5.3. Predicative Specifications as Sets of Functions

Hehner's predicative notation is very elegant: he expresses relations between input states, σ‘, and output states, σ’, nicely as predicates in the program variables - x‘ stands for σ‘.x and x’ stands for σ’.x. The predicative description of sets of functions between states seems much more difficult. We make the following suggestion.

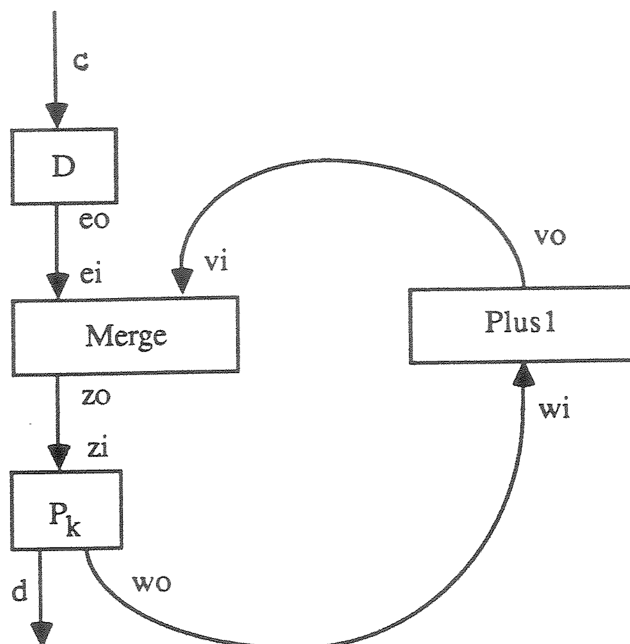

Figure 1. The Brock-Ackerman Anomaly

A set of functions is specified by a predicate on the continuous functions from states to states (the set PROG-FUNC). Employing a similar trick as before, we use in the predicate a special free identifier f. This free identifier is a place holder for continuous state-to-state functions that satisfy the predicate. More precisely, we use in our predicative definitions predicates Φ with the particular free identifiers f, σ‘, and σ’, where we always assume the relationship f.σ‘ = σ’. The predicate Φ.(f, σ‘, σ’) specifies the set of functions

$$\{f \in \text{PROG-FUNC}: (\forall\, \sigma‘, \sigma’ \in \text{STATE}: f.\sigma‘ = \sigma’ \;\Rightarrow\; \Phi.(f, \sigma‘, \sigma’))\}$$

Then, we define the predicative specification of the nondeterministic choice operator as follows:

$$\text{PS}[\![\text{S1} \,\square\, \text{S2}]\!] \;=\; ((\forall\, \sigma‘, \sigma’: f.\sigma‘ = \sigma’ \;\Rightarrow\; \text{PS}[\![\text{S1}]\!]) \vee (\forall\, \sigma‘, \sigma’: f.\sigma‘ = \sigma’ \;\Rightarrow\; \text{PS}[\![\text{S2}]\!]))$$

Note that we do not assume that nondeterministic choice distributes over recursive program definitions. In this sense, we have not given a complete predicative specification so far. Obviously, a complete predicative specification requires a rather complicated predicative formalism. However, we can greatly simplify the predicative specification if we restrict ourselves to partial or robust correctness.

6. Correctness of Nondeterministic Programs

6.1. Partial Correctness and Predicative Semantics

6.1.1. Considering only Finite States

For partial correctness, we may consider only finite states, i.e., states without infinite streams. This avoids infinite elements altogether. Every infinite state $\sigma 2$ is uniquely characterized by the set of finite states $\sigma 1$ such that $\sigma 1 \sqsubseteq \sigma 2$. Let $STATE_{fin}$ denote the set of finite states, i.e., states that contain only finite streams. We introduce a new meaning function specifically for partial correctness:

$$F^{pc}: STAT \rightarrow ENV_{pc} \rightarrow PROG\text{-}FUNC_{pc}$$

where

$$\begin{aligned} PROG\text{-}FUNC_{pc} \; = \; & \{f \in \mathcal{P}([STATE_{fin} \rightarrow STATE_{fin}]): \\ & \quad (\forall \sigma \in STATE: \\ & \quad\quad (\forall c \in ICI: (\exists i \in N: rest^{i}.\sigma.c = f.\sigma.c)) \quad \wedge \\ & \quad\quad (\forall d \in OCI: (\exists s \in D^{*} \cup (D^{*} \times \{\bot\}): \sigma.d \mathbin{@} s = f.\sigma.d))\} \end{aligned}$$

$$ENV_{pc} \; = \; [PROG\text{-}ID \rightarrow PROG\text{-}FUNC_{pc}]$$

F^{pc} is defined as:

$$F^{pc}_{\delta}[\![S]\!] \; = \; \{f \in PROG\text{-}FUNC_{pc}: (\exists f1 \in F_{\delta}[\![S]\!]: f1 \sqsubseteq f)\}$$

For partial correctness, the consideration of finite states is sufficient because only the absence of incorrect output is required. Since every infinite stream is uniquely determined by its set of finite approximations, output that is incorrect with respect to some specification will always show up in a finite state.

Let LOG-ID be a set of logical identifiers disjoint from all other sets of identifiers. We now propose the predicative specification PS_{pc} for partial correctness. $PS_{pc}[\![S]\!]$ is a predicate $\Phi.(f, \sigma', \sigma')$ on a function f and on input and output states σ' and σ'. We shall also employ a kind of "closure" on Φ, namely the predicate $CLOSE.\Phi.(f, \sigma', \sigma')$, with only free identifier f, defined by:

$$CLOSE.\Phi.(f, \sigma', \sigma') \; = \; (\forall \sigma', \sigma' \in STATE: f.\sigma' = \sigma' \Rightarrow \Phi.(f, \sigma', \sigma'))$$

CLOSE is a predicate transformer that takes a predicate in three free variables - a function and two states - and binds the two state variables.[2] Note that $\Phi.(f, \sigma', \sigma') = CLOSE.\Phi.(f, \sigma', \sigma')$ if σ' and σ' do not occur freely in $\Phi.(f, \sigma', \sigma')$.

$$PS_{pc}[\![x := E]\!] \; = \; \sigma' \sqsubseteq \sigma'[E'/x]$$

$$PS_{pc}[\![\textbf{if } E \textbf{ then } S1 \textbf{ else } S2 \textbf{ fi}]\!]$$
$$= \; (\neg def.E' \wedge (\sigma' \sqsubseteq \sigma'\!\downarrow)) \vee (E' \wedge PS_{pc}[\![S1]\!]) \vee (\neg E' \wedge PS_{pc}[\![S2]\!])$$

$$PS_{pc}[\![c \; ? \; x]\!] \; = \; \sigma' \sqsubseteq \sigma'[first.c'/x, rest.c'/c]$$

[2] CLOSE is very similar to Dijkstra's square brackets [4].

$$PS_{pc}[\![d \,!\, E]\!] \;=\; \sigma' \sqsubseteq \sigma`[(d \;@\; \langle E`\rangle)/x]$$

$$PS_{pc}[\![S1 \,;\, S2]\!] \;=\; (NT' \wedge PS_{pc}[\![S1]\!]) \vee (\exists\, f1, f2 \in \text{PROG-FUNC}_{pc}: \text{CLOSE}.PS_{pc}[\![S1]\!][f1/f] \wedge \text{CLOSE}.PS_{pc}[\![S2]\!][f2/f] \wedge (\sigma' = f2.f1.\sigma`)$$

$$PS_{pc}[\![S1 \,||\, S2]\!] \;=\; PS_{pc}[\![S1]\!] \wedge PS_{pc}[\![S2]\!]$$

$$PS_{pc}[\![S1 \,\Box\, S2]\!] \;=\; \text{CLOSE}.PS_{pc}[\![S1]\!] \vee \text{CLOSE}.PS_{pc}[\![S2]\!]$$

Channel Connection by Computational Induction:

$$PS_{pc}[\![\textbf{chan}\; c \leftarrow d\!: S]\!] \;=\; (\exists\, g \in \text{PROG-FUNC}_{pc}: \text{CLOSE}.PS_{pc}[\![S]\!][g/f] \wedge (\forall\, \sigma`\!: (\forall\, i \in \mathbb{N}\!: f.\sigma` \sqsubseteq \sigma_i[\sigma`.c/c,\, \sigma`.d/d]))$$

where
$$\sigma_0 = \sigma`[\langle\bot\rangle/c,\, \epsilon/d]$$
$$\sigma_{i+1} = g(\sigma`[\sigma_i.d/c,\, \epsilon/d])$$

Channel Connection by Fixed Point Induction:

Let x, y $\in$ LOG-ID be pairwise distinct and not free in $PS_{pc}[\![S]\!]$:

$$PS_{pc}[\![\textbf{chan}\; c \leftarrow d\!: S]\!] \;=\; (\exists\, g \in \text{PROG-FUNC}_{pc}: \text{CLOSE}.PS_{pc}[\![S]\!][g/f] \wedge (\forall\, \sigma\!: (\exists\, x, y\!: \sigma[y/c,\, x/d] = g(\sigma`[x/c,\, \epsilon/d])) \Rightarrow f.\sigma \sqsubseteq \sigma)))$$

Refinement Definition by Computational Induction:

With auxiliary definitions:

$$p_0 = \text{CLOSE}.(\sigma' = \sigma`\!\downarrow)$$
$$p_{i+1} = \text{CLOSE}.PS_{pc}[\![S]\!][p_i/PS_{pc}[\![p]\!]]$$

we define:

$$PS_{pc}[\![p :: S]\!] \;=\; (\forall\, i \in \mathbb{N}\!: p_i)$$

Refinement Definition by Fixed Point Induction:

$$(\forall\, \Phi\!: ((\forall\, f, g\!: \text{CLOSE}.\Phi \wedge g \sqsubseteq f \Rightarrow \text{CLOSE}.\Phi[g/f]) \wedge (\forall\, f\!: \text{CLOSE}.PS_{pc}[\![S]\!] = \text{CLOSE}.\Phi)) \Rightarrow \text{CLOSE}.f)$$

Here, we express again a least fixed point property. For partial correctness, if several fixed points exist, then non-least fixed points may produce incorrect results. However, this does not affect the partial correctness of the program, if the least fixed point produces only correct, not incorrect results.

6.1.2. Connection Between the Two Semantics

Let S be a program; then:

$$(\exists\, g \in B_\delta[\![S]\!]\!: f \sqsubseteq g \Leftrightarrow \text{CLOSE}.PS_{pc}[\![S]\!])$$

The boolean expression $PS_{pc}[\![S]\!]$ that contains x` and x' as identifiers for data objects (where x $\in$ PVI) and c` and c' as identifiers for streams (where c $\in$ ICI $\cup$ OCI) is a shorthand for the predicate that we obtain from $PS[\![S]\!]$ by replacing x` by $\sigma`.x$, x' by $\sigma'.x$, c` by $\sigma`.c$, and c' by $\sigma'.c$. Furthermore, $PS_{pc}[\![S]\!]$ may include free occurrences of the function identifier f. CLOSE.$PS_{pc}[\![S]\!]$ contains only the identifier f freely (but no longer $\sigma`$ and σ'), and therefore is the specification of a set of state-to-state functions.

6.2. Robust Correctness and Predicative Specifications

Again, predicative specifications of robustly correct programs require only slight modifications. It is even possible to use simpler fixed point definitions, because it is no longer necessary to capture the properties of least fixed points exactly. (Every fixed point is $\sqsubseteq$-greater than a least fixed point. Therefore the output states σ' of any fixed point are $\sqsubseteq$-greater than the output states of the least fixed point.) Thus, it is also not necessary to work with sets of functions. The meaning function for robust correctness is:

$$B^{rc}: \text{STAT} \rightarrow \text{ENV} \rightarrow \mathcal{P}(\text{PROG-FUNC})$$

B^{rc} is defined as:

$$B^{rc}_{\delta}[\![S]\!] \quad = \quad \{f \in \text{PROG-FUNC}: (\forall\, \sigma \in \text{STATE}: B_{\delta}[\![S]\!].\sigma \sqsubseteq f.\sigma)\}$$

PS_{rc} is obtained from PS by eliminating most of the references to NT in basic (i.e., not composed) statements and weakening a conjunction in the rule of composition to an implication:

$$PS_{rc}[\![x := E]\!] \quad = \quad \sigma`[E`/x] \sqsubseteq \sigma'$$

$$PS_{rc}[\![\textbf{if } E \textbf{ then } S1 \textbf{ else } S2 \textbf{ fi}]\!]$$
$$= \quad (\neg\text{def}.E` \wedge \sigma`\!\downarrow\; \sqsubseteq \sigma') \vee (E` \wedge PS_{rc}[\![S1]\!]) \vee (\neg E` \wedge PS_{rc}[\![S2]\!])$$

$$PS_{rc}[\![c\ ?\ x]\!] \quad = \quad \sigma`[\text{first}.c`/x,\ \text{rest}.c`/c] \sqsubseteq \sigma'$$

$$PS_{rc}[\![d\ !\ E]\!] \quad = \quad \sigma`[(d\ @\ \langle E`\rangle)/d] \sqsubseteq \sigma'$$

$$PS_{rc}[\![S1\ ;\ S2]\!] \quad = \quad (\exists\, \sigma \in \text{STATE}: (PS_{rc}[\![S1]\!][\sigma/\sigma'] \wedge (\sigma \sqsubseteq \sigma' \wedge \text{NT}.\sigma \vee (\neg\text{NT}.\sigma \wedge PS_{rc}[\![S2]\!][\sigma/\sigma`]))))$$

$$PS_{rc}[\![S1\ ||\ S2]\!] \quad = \quad \begin{array}{l} (PS_{rc}[\![S1]\!] \wedge PS_{rc}[\![S2]\!] \wedge \neg\text{NT}') \vee \\ (PS_{rc}[\![S1]\!] \wedge PS_{rc}[\![S2]\!]\!\downarrow \wedge\, \text{NT}') \vee \\ (PS_{rc}[\![S2]\!] \wedge PS_{rc}[\![S1]\!]\!\downarrow \wedge\, \text{NT}') \end{array}$$

$$PS_{rc}[\![S1 \,\square\, S2]\!] \quad = \quad PS_{rc}[\![S1]\!] \vee PS_{rc}[\![S2]\!]$$

Channel Connection by Computational Induction:

Let a, b, e ∈ LOG-ID be pairwise distinct:

$$S0 \quad = \quad (\exists\, a: PS_{rc}[\![S]\!][\epsilon/d`,\ b/d',\ a/c'])$$

$$P_0 \quad = \quad S0[\langle\perp\rangle/c`]$$

$$P_{i+1} \quad = \quad (\exists\, e: (\exists\, \sigma: P_i[e/b][\sigma/\sigma'] \wedge S[e/c`]))$$

These definitions correspond exactly to the ones for PS in Sect 4.2.

$$PS_{rc}[\![\textbf{chan } c \leftarrow d: S]\!] \quad = \quad (\forall\, i \in \mathbb{N}: (\exists\, \sigma,\ b: P_i[\sigma/\sigma'] \wedge \sigma \sqsubseteq \sigma')$$

Channel Connection by Fixed Point Induction:

Let x, y ∈ LOG-ID be pairwise distinct and not free in $PS_{rc}[\![S]\!]$:

$$PS_{rc}[\![\textbf{chan } c \leftarrow d: S]\!] \quad = \quad (\exists\, x,\ y: PS_{rc}[\![S]\!][y/c`,\ \epsilon/d`,\ x/c',\ y/d'])$$

This formula expresses the fixed point property that c` = d' must follow from $PS_{rc}[\![S]\!][\epsilon/d`]$. is a fixed point.

Refinement Definition by Computational Induction:

With auxiliary definitions:

$$p_0 \quad = \quad \text{true}$$

$$p_{i+1} \quad = \quad PS_{rc}[\![S]\!][p_i/PS_{rc}[\![p]\!]]$$

we define:

$$PS_{rc}[\![p :: S]\!] \quad = \quad (\forall\, i \in N: (\exists\, \sigma: P_i[\sigma/\sigma'] \wedge \sigma \sqsubseteq \sigma'))$$

Refinement Definition by Fixed Point Induction:

For the fixed point semantics, we introduce identifiers for relations on states:

$$PS_{rc}[\![p :: S]\!] \quad = \quad (\exists\, q \in \text{PRED-REL}: q.(\sigma`, \sigma') \wedge (\forall\, \sigma`, \sigma' \in \text{STATE}: q.(\sigma 1, \sigma 2) \Leftrightarrow PS_{rc}[\![S]\!][q.(\sigma`, \sigma')/PS_{rc}[\![p]\!]]))$$

This formula expresses that $PS[\![p :: S]\!]$ is a fixed point.

For robust correctness, the Brock-Ackerman merge anomaly does not arise. The Brock-Ackerman merge anomaly reflects the fact that, for set-valued functions (i.e., for relations) it is not always possible to distinguish least fixed points from other fixed points. For robust correctness such separations are not required. Robust correctness considers all functions that are approximated by the least fixed points. But these functions coincide with the set of all functions that are approximated by some arbitrary fixed point, since every fixed point is approximated by a least fixed point.

6.2.1. Considering only Total States

If we are interested only in robust correctness, we may disregard partial states altogether and consider only total states. Note that, for every partial state $\sigma 1$, there exists a total state $\sigma 2$ such that $\sigma 1 \sqsubseteq \sigma 2$. Let $STATE_{tot}$ denote the set of total states. We introduce a new meaning function:

$$B^r: \text{STAT} \rightarrow \text{ENV}_r \rightarrow \text{PROG-FUNC}_r$$

where

$$\text{PROG-FUNC}^r \quad = \quad [\text{STATE}_{tot} \rightarrow \mathcal{P}(\text{STATE}_{tot})]$$

$$\text{ENV}_r \quad = \quad [\text{PROG-ID} \rightarrow \text{PROG-FUNC}_r]$$

B^r is defined as:

$$B^r_\delta[\![S]\!].\sigma` \quad = \quad \{\sigma \in \text{STATE}_{tot}: (\exists\, \sigma' \in \text{STATE}: \sigma' \sqsubseteq \sigma \wedge (\sigma' \in B_\delta[\![S]\!].\sigma`))\}$$

Nontermination is modelled by associating with an input state the set of total states that are approximated by the partial output state. Nondeterminism leads to pathologies in this approach. For example, the strictness of sequential composition is not easily specified. Tricks that circumvent this problem lead to unpleasant effects (such as the nonassociativity of sequential composition [5]). Let us consider, as a simple example, a program with only one boolean variable x and, therefore, with a finite state space:

(1) (x:=true □ x:=false) ; **if** x **then** x:=x **else** x:=true **fi**

(2) p ; **if** x **then** x:=x **else** x:=true **fi** where p :: (x:=true □ x:=false □ p)

Especially from the viewpoint of robust correctness, the two programs should be distinguished: Program 1 always terminates, but Program 2 may not terminate. However, without the special element $\perp$ for divergence the programs cannot be distinguished.

7. On Nontermination

There are two distinct ways in which communicating programs may refrain from terminating:

(1) *Infinite Output:* a program may define a computer behavior with infinitely many observable actions, e.g., Hehner's ONES :: d!1 ; ONES.

(2) *Divergence:* a program may define an infinite computer behavior without any further observable action like output, e.g., p :: p.

Nontermination can never be observed in finite time. But we can conclude it from an inspection of the (infinite) set of all possible finite observations.

It is debatable whether programs that fail to terminate for certain input are useful - in some sense, they are if they are used in a "safe" environment - and whether the explicit specification of nontermination is relevant. In our language, the predicative specification of program ONES contains conjunct NT' to make nontermination explicit. Hehner deals only with total states. He replaces partial states by sets of total states that approximate them. This leads to a number of irritating little problems with sequential composition and programs over finite state spaces (as Hehner points out in [5]).

8. Conclusions

The appropriate choice of a predicative semantics depends closely on the concept of correctness and combination of features in the programming language. While certain combinations lead to elegant predicative specifications, others lead to a number of technical problems. We found that arbitrary fixed points suffice for the robust correctness, while least fixed points are required for the partial correctness of recursion. Fixed point induction is best suited for robust correctness, while computational induction is best suited for partial correctness.

In the presence of recursion, nondeterministic choice has to be defined carefully - by sets of functions rather than by set-valued functions, in the case of partial or total correctness.

We have shown that denotational and predicative semantics can be chosen to be isomorphic. One may ask why the two different definitions should then be given at all. The answer is: because they emphasize different aspects of a programming language - much like two programming languages which are Turing-equivalent emphasize different aspects of an algorithm.

A denotational semantics makes certain mathematical properties such as monotonicity, continuity, fixed point properties, and the existence of an output state for every input state more explicit. Moreover, a denotational semantics translates programs into a functional calculus (the λ-calculus), the formulas of which can again be understood as representations of algorithms. That is, denotational semantics aims at implementations. A predicative semantics translates programs into logical formulas which can be reasoned about conveniently. That is, a predicative semantics aims at proofs.

References

1. Brock, J. D., and Ackerman, W. B. Scenarios: A Model of Nondeterminate Computation. In *Formalization of Programming Concepts*, J. Diaz and I. Ramos, Eds., Lecture Notes in Computer Science 107, Springer-Verlag, 1981, pp. 252-259.

2. Broy, M. Fixed Point Theory for Communication and Concurrency. IFIP TC2 Working Conference on Formal Description of Programming Concepts II, 1983, pp. 125-147.

3. Broy, M. Extensional Behaviour of Concurrent, Nondeterministic, Communicating Programs. In *Control Flow and Data Flow: Concepts of Distributed Programming*, M. Broy, Ed., NATO ASI Series F: Computer and Systems Sciences, Vol. 14, Springer-Verlag, 1985. (2nd printing: Springer Study Edition)..

4. van Gasteren, A. J. M., and Dijkstra, E. W. Remarks on Notation. AvG19/EWD-815, Department of Mathematics and Computing Science, Eindhoven University of Technology, Mar., 1982.

5. Hehner, E. C. R. "Predicative Programming (Part I and II)". *Comm. ACM 27*, 2 (Feb. 1984), 134-151.

6. Hoare, C. A. R. Programs are Predicates. In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. S. Sheperdson, Eds., Series in Computer Science, Prentice-Hall Int., 1985, pp. 141-155.

7. Lengauer, C. "Technical Correspondence: Predicative Programming". *Comm. ACM 27*, 5 (May 1985), 537-538.

8. Smyth, M. "Power Domains". *JCSS 16* (1978), 23-26.

9. Stoy, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

Appendix: Consistency Theorem

Theorem:

For all states σ‘ and σ’, and for all deterministic programs S in our programming language:

$$(*) \qquad \sigma' = B_\delta[\![S]\!].\sigma` \quad \Leftrightarrow \quad PS[\![S]\!]$$

provided for identifiers p for programs occuring in S we can assume the same predicates

$$\sigma' = B_\delta[\![p]\!].\sigma` \Leftrightarrow PS[\![p]\!] \ .$$

(If S is deterministic, $B_\delta[\![S]\!]$ has exactly one element f which we denote by $B_\delta[\![S]\!]$.)

Proof: (by structural induction on S)

Assume (*) holds for proper substatements.

(1) **Assignment**: Let $S \equiv x := E$; then we have $X = \{x\}$. According to the definitions we obtain for (*):

$$(V[\![E]\!].\sigma` \neq \bot \Rightarrow \sigma'.x = \sigma`[V[\![E]\!].\sigma`/x].x) \wedge (V[\![E]\!].\sigma` = \bot \Rightarrow \sigma'.x = \sigma`\downarrow.x)$$

$$\Leftrightarrow (\neg def.E` \wedge \sigma' = \sigma`\downarrow) \vee (def.E` \wedge \sigma' = \sigma`[E`/x])$$

This is equivalent to

$$(\neg V[\![E]\!].\sigma` \neq \bot \vee \sigma' = \sigma`[V[\![E]\!].\sigma`]) \wedge (\neg V[\![E]\!].\sigma` = \bot \vee \sigma' = \sigma`\downarrow)$$

$$\Leftrightarrow (\neg V[\![E]\!].\sigma` \neq \bot \wedge \sigma' = \sigma`\downarrow) \vee (V[E].\sigma` \neq \bot \wedge \sigma' = \sigma`[V[\![E]\!].\sigma`/x])$$

which trivially is true, of course.

(2) **Conditional**: Let S $\equiv$ **if** E **then** S1 **else** S2 **fi**; then we obtain for (*):

$$(V[\![E]\!].\sigma` \Rightarrow \sigma' = B_\delta[\![S1]\!].\sigma`) \wedge (\neg V[\![E]\!].\sigma` \Rightarrow \sigma' = B_\delta[\![S2]\!].\sigma`) \wedge (V[\![E]\!].\sigma` = \bot \Rightarrow \sigma' = \sigma`\downarrow)$$

$$\Leftrightarrow (\neg def.E` \wedge \sigma' = \sigma`\downarrow) \vee (E` \wedge PS[\![S1]\!]) \vee (\neg E` \wedge PS[\![S2]\!])$$

We have to consider three cases:

(a) $V[\![E]\!].\sigma`$ holds, then we have to prove:

$$\sigma' = B_\delta[\![S1]\!].\sigma` \Leftrightarrow (E` \wedge PS[\![S1]\!])$$

which is equivalent to the induction hypothesis (since $E` \equiv V[\![E]\!].\sigma`$):

$$\sigma' = B_\delta[\![S1]\!].\sigma` \Leftrightarrow PS[\![S1]\!]$$

(b) $\neg V[\![E]\!]$ holds, then we have to prove:

$$\sigma' = B_\delta[\![S2]\!].\sigma` \Leftrightarrow (\neg E` \wedge PS[\![S2]\!])$$

which is equivalent to the induction hypothesis:

$$\sigma' = B_\delta[\![S2]\!].\sigma` \Leftrightarrow PS[\![S2]\!]$$

(c) $V[\![E]\!] = \bot$ holds, then we have to prove:

$$\sigma' = \sigma`\downarrow \Leftrightarrow \sigma' = \sigma`\downarrow$$

which is trivially true.

(3) **Receive statement**: Let S $\equiv$ c ? x; then we obtain for (*)

$$(\text{first}.\sigma.\text{c} \neq \bot \Rightarrow \sigma' = \sigma`[\text{first}.\sigma`.\text{c}/\text{x}, \text{rest}.\sigma`.\text{c}/\text{c}]) \wedge (\text{first}.\sigma.\text{c} = \bot \Rightarrow \sigma' = \sigma`\downarrow)$$

$$\Leftrightarrow ((\#\text{c}` = 0 \wedge \sigma' = \sigma`\downarrow) \vee (\#\text{c}`>0 \wedge \sigma' = \sigma`[\text{first.c}`/\text{x}, \text{rest.c}`/\text{c}]))$$

We have to consider two cases:

(a) first.σ.c $\neq \bot$; then we obtain (by #c`>0) for (*)

$$\sigma' = \sigma`[\text{first}.\sigma`.\text{c}/\text{x}, \text{rest}.\sigma`.\text{c}/\text{c}] \Leftrightarrow \sigma' = \sigma`[\text{first.c}`/\text{x}, \text{rest.c}`/\text{c}]$$

which is true, of course.

(b) first.σ.c $= \bot$; then we obtain (by #c` = 0) for (*)

$$\sigma' = \sigma`\downarrow \Leftrightarrow \sigma' = \sigma`\downarrow$$

which is trivial.

(4) **Send statement**: Let S $\equiv$ c ! E; then we obtain for (*)

$$(\text{V}[\![\text{E}]\!].\sigma` \neq \bot \Rightarrow \sigma' = \sigma`[(\sigma(\text{c}) \text{ @ } <\text{V}[\![\text{E}]\!].\sigma`>)/\text{c}]) \wedge (\text{V}[\![\text{E}]\!].\sigma` = \bot \Rightarrow \sigma' = \sigma`\downarrow)$$

$$\Leftrightarrow (\neg\text{def.E}` \wedge \sigma' = \sigma`\downarrow) \vee (\text{def.E}` \wedge \sigma' = \sigma`[(\text{d}` \text{ @ } <\text{E}`>)/\text{d}])$$

We consider two cases:

(a) V[[E]].σ` $\neq$ 1; we obtain for (*)

$$\sigma' = \sigma`[(\sigma(\text{c}) \text{ @ } <\text{V}[\![\text{E}]\!].\sigma`>)/\text{c}] \Rightarrow \sigma' = \sigma`[\text{d}` \text{ @ } <\text{E}`>/\text{d}]$$

which is trivially true.

(b) V[[E]].σ` $= \bot$; we obtain for (*)

$$\sigma' = \sigma`\downarrow \Leftrightarrow \sigma' = \sigma`\downarrow$$

which is trivially true.

(5) **Sequential composition**: Let S $\equiv$ (S1 ; S2); we obtain for (*)

$$\sigma' = \text{B}_\delta[\![\text{S2}]\!].\text{B}_\delta[\![\text{S1}]\!].\sigma`$$

$$\Leftrightarrow (\text{NT}' \wedge \text{PS}[\![\text{S1}]\!]) \vee (\exists\, \sigma \in \text{STATE} : (\text{PS}[\![\text{S1}]\!] \wedge \neg\, \text{NT}')[\sigma/\sigma'] \wedge \text{PS}[\![\text{S2}]\!][\sigma/\sigma`])$$

We consider two cases:

(a) NT.B_δ[[S1]].σ`; then we obtain for (*)

$$\sigma' = \text{B}_\delta[\![\text{S1}]\!].\sigma`\downarrow \Leftrightarrow \text{NT}' \wedge \text{PS}[\![\text{S1}]\!]$$

which is equivalent to the induction hypothesis

$$\sigma' = \text{B}_\delta[\![\text{S1}]\!].\sigma` \Leftrightarrow \text{PS}[\![\text{S1}]\!]$$

(b) ¬NT.B_δ[[S1]].σ`; then we obtain for (*)

$$\sigma' = \text{B}_\delta[\![\text{S2}]\!].\text{B}_\delta[\![\text{S1}]\!].\sigma`$$

$$\Leftrightarrow (\exists\, \sigma \in \text{STATE} : (\text{PS}[\![\text{S1}]\!] \wedge \neg\text{NT}')[\sigma/\sigma'] \wedge \text{PS}[\![\text{S2}]\!][\sigma/\sigma`]$$

Now set $\sigma = \text{B}_\delta[\![\text{S1}]\!].\sigma`$; then we obtain for (*)

$$\sigma' = B_\delta[\![S2]\!].\sigma \Leftrightarrow ((PS[\![S1]\!] \wedge \neg NT)[\sigma/\sigma'] \wedge PS[\![S2]\!][\sigma/\sigma'])$$

which can be proved from

$$(\sigma' = B_\delta[\![S2]\!].\sigma \Leftrightarrow PS[\![S2]\!][\sigma/\sigma']) \wedge (\sigma = B_\delta[\![S1]\!].\sigma' \Leftrightarrow PS[\![S1]\!][\sigma/\sigma'])$$

which is equivalent to the induction hypothesis.

(6) **Parallel composition**: Let S $\equiv$ (S1 || S2); we obtain for (*)

$$(\neg NT.B_\delta[\![S1]\!].\sigma' \Rightarrow \sigma' = B_\delta[\![S2]\!].B_\delta[\![S1]\!].\sigma') \wedge$$
$$(\neg NT.B_\delta[\![S2]\!].\sigma' \Rightarrow \sigma' = B_\delta[\![S1]\!].B_\delta[\![S2]\!].\sigma') \wedge$$
$$(NT.B_\delta[\![S1]\!].\sigma' \wedge NT.B_\delta[\![S2]\!].\sigma') \Rightarrow \sigma' = join.(B_\delta[\![S1]\!].\sigma', B_\delta[\![S2]\!].\sigma')$$

$$\Leftrightarrow (PS[\![S1]\!] \wedge PS[\![S2]\!] \wedge \neg NT') \vee$$
$$(PS[\![S1]\!] \wedge NT' \wedge PS[\![S2]\!]\downarrow) \vee$$
$$(PS[\![S2]\!] \wedge NT' \wedge PS[\![S1]\!]\downarrow)$$

We consider three cases:

(a) $\neg NT.B_\delta[\![S1]\!].\sigma'$; then (*) can be proved from

$$\sigma' = B_\delta[\![S2]\!].B_\delta[\![S1]\!].\sigma'$$
$$\Leftrightarrow (PS[\![S2]\!] \wedge NT' \wedge PS[\![S1]\!]\downarrow) \vee (PS[\![S1]\!] \wedge PS[\![S2]\!] \wedge \neg NT')$$

We consider two subcases:

(i) $\neg NT.B_\delta[\![S2]\!].\sigma'$; then (*) reads

$$\sigma' = B_\delta[\![S2]\!].B_\delta[\![S1]\!].\sigma' \Leftrightarrow (PS[\![S1]\!] \wedge PS[\![S2]\!] \wedge \neg NT')$$

which can be concluded from the induction hypothesis due to the independence of S1 and S2.

(ii) $NT.B_\delta[\![S2]\!].\sigma'$; then (*) reduces to

$$\sigma' = B_\delta[\![S2]\!].B_\delta[\![S1]\!].\sigma' \Leftrightarrow (PS[\![S2]\!] \wedge NT' \wedge PS[\![S1]\!]\downarrow)$$

which can be concluded from the induction hypothesis and the independence of S1 and S2.

(b) $\neg NT.B_\delta[\![S2]\!].\sigma'$; then (*) can be proved in analogy to (a).

(c) $NT.B_\delta[\![S1]\!].\sigma' \wedge NT.B_\delta[\![S2]\!].\sigma'$; then (*) can be proved from

$$\sigma' = join.(B_\delta[\![S1]\!].\sigma', B_\delta[\![S2]\!].\sigma') \Leftrightarrow (PS[\![S1]\!] \wedge NT' \wedge PS[\![S2]\!])$$

which follows from the induction hypothesis, the definition of join and the independence of S1 and S2.

(7) **Channel connection**: Let S $\equiv$ [**chan** c $\rightarrow$ d: S1]; then we define $\sigma 2$ as the least fixed point of the state-to-state function

$$f = (\lambda\ \sigma : B_\delta[\![S1]\!].\sigma'[\sigma(d)/c, \in/d])$$

f is continuous; thus, the following equation holds for $\sigma 2$ as defined in the denotational semantics of channel connect:

$$\sigma 2 = \sup\{f^i(\omega) : i \in N\}$$

where

$f^0.\omega = \omega$
$f^{i+1}.\omega = f^i.\omega$

and

$\omega.x = \bot$ for all $x \in \mathbb{D}$

By induction hypothesis we have

$\sigma' = B_\delta[\![S1]\!].\sigma` \Leftrightarrow PS[\![S1]\!]$

We prove by induction on i

(**) $(\sigma' = f^i.\omega) \Leftrightarrow P_i[d'/b]$

(a) if $i = 0$ we obtain

$$\sigma' = B_\delta[\![S1]\!].\sigma`[<\bot>/c, \epsilon/d] \Leftrightarrow (\exists\ a: PS[\![S1]\!][\epsilon/d`, b/d', a/c'])[<\bot>/c`][d'/b]$$

which simplifies to

$$\sigma' = B_\delta[\![S1]\!].\sigma`<\bot>/c, \epsilon/d] \Leftrightarrow (\exists\ a: PS[\![S1]\!][\epsilon/d`, <\bot>/c`])$$

which is a simple consequence of the induction hypothesis.

(b) Assume (**) holds for i; then

$$(\sigma' = f^{i+1}.\omega) \Rightarrow (\exists\ \sigma: \sigma = f^i.\omega \wedge \sigma' = f.\sigma)$$

$$\Leftrightarrow (\exists\ \sigma: P_i[\sigma/\sigma'][d'/b, c'/a] \wedge \sigma' = B_\delta[\![S1]\!].\sigma`[\sigma.d/c, \epsilon/d]) \Rightarrow$$
$$(\exists\ e: (\exists\ \sigma : P_i[e/b][\sigma/\sigma'] \wedge S1[e/c`]))[d'/b]$$

Either $\neg NT.sup\{f^i.\omega: i \in N\}$ or not. In the first case there exists some i such that $\neg NT.f_i.\omega$ which implies the first line of the predicative specification (together with (**)). The second case implies the second half of the predicative specification (together with (**)).

The fixed point definition of the channel connection is a simple consequence of the least fixed point property of $(\mu\ \sigma: f.\sigma)$. If σ is a fixed point of f, then

$\sigma = B_\delta[\![S1]\!].\sigma`[\sigma.d/c, \epsilon/d]$

If $\neg NT.\sigma$ then

$\sigma' = B_\delta[\![\textbf{chan}\ c \rightarrow d: S1]\!].\sigma`$

which is equivalent to

$\sigma' = (B_\delta[\![S1]\!].\sigma`[\sigma.d/c, \epsilon/d])[\sigma`(c)/c, \sigma`.d/d]$

which is (with $y = \sigma.d$, $x = \sigma.c$) equivalent to

$(\exists\ x, y: \sigma'[x/c, y/d] = B_\delta[\![S1]\!].\sigma`[y/c, \epsilon/d])$

which is (by induction hypothesis) equivalent to

$(\exists\ x, y: PS[\![S1]\!][y/c`, \epsilon/d`, x/c', y/d'])$

This proves the first half of the predicative specification from $\sigma' = B_\delta[\![\textbf{chan}\ c \rightarrow d: S1]\!]$. The second half is obtained the same way by the least fixed point property of σ.

(8) **Recursive definition**: Let $S = B_\delta[\![p :: S0]\!]$; according to our definitions $B_\delta[\![p :: S0]\!]$ is the least fixed point of the continuous function

$\tau = (\lambda\ f\colon B_{\delta[f/p]}[\![S0]\!])$

With the auxiliary definitions used for specifying PS$[\![p :: S0]\!]$ by computational induction we obtain

$$p_i \Leftrightarrow \sigma' = \tau^j.\Omega.\sigma'$$

where Ω is defined by

$$\Omega.\sigma = \sigma\downarrow$$

and τ^j is defined by

$$\tau^0 = \Omega$$
$$\tau^{j+1}.f = \tau.f$$

The equivalence (*) follows from

$$\sup\{\tau^j.\Omega\colon i \in \mathbf{N}\} = B_\delta[\![p :: S0]\!]$$

For the fixed point definition of PS$[\![p :: S0]\!]$ the given formula is a simple consequence of the least fixed point property of $B_\delta[\![p :: S0]\!]$ with respect to τ.

(9) **Refinement Call:** Let $S = p$, where p is a program identifier. The statement of the theorem follows by assumption.

(End of Proof)