Safety and Progress of Recursive Procedures

Hesselink, Wim H.

Published in:
Formal Aspects of Computing

Publication date:
1995

Formal Aspects of Computing (1995) 7: 389–411

Formal Aspects of Computing

Safety and Progress of Recursive Procedures

Wim H. Hesselink

Rijksuniversiteit Groningen, Department of Computing Science, Groningen, The Netherlands

Dedicated to the memory of
Jan van de Snepscheut

Keywords: Predicate transformers; Weakest preconditions; Recursive procedures; Operational semantics; Proof rules; Safety; Progress

Abstract. Temporal weakest precondions are introduced for calculational reasoning about the states encountered during execution of not-necessarily terminating recursive procedures. The formalism can distinguish error from useful nontermination. The precondition functions are constructed in a new and more elegant way. Healthiness laws are discussed briefly. Proof rules are introduced that enable calculational proofs of various safety and progress properties. The construction of the precondition functions is justified in an Appendix that provides the operational semantics.

1. Introduction

The aim of this paper is to present a calculus for the description of the runtime behaviour of (not necessarily terminating) imperative programs during execution. The calculus is an extension of Dijkstra's weakest preconditions with temporal preconditions introduced by Lukkien and Van de Snepscheut [LuS92] and additional precondition functions for safety and absence of errors. Our programming language allows arbitrary mutually recursive procedures. We prove soundness of proof rules and give examples where they are used to discuss the runtime behaviour of nonterminating procedures. The proof rules (formulated for mutually recursive procedures) can easily be specialized to repetitions.

To be more specific, our programs are commands that act on some state space. So, the command starts in some initial state and the command generates a

Correspondence and offprint requests to: Wim H. Hesselink, Rijksuniversiteit Groningen, Department of Computer Science, PO Box 800, 9700 AV Groningen, The Netherlands. email: wim@cs.rug.nl

finite or infinite sequence of subsequent states, in a possibly nondeterminate way. Given predicates p and r on the state space, we are, for example, interested in the question whether there is an intermediate state where p holds or the command terminates in a state where r holds.

Such a question depends on the initial state. We therefore introduce *wev.p.S.r* to stand for the weakest precondition such that execution of command S ever reaches a state where predicate p holds or terminates in a state where predicate r holds. For example, let k be an integer program variable and let S be given by

$$S = \textbf{while } \mathrm{k} > 2 \textbf{ do } \mathrm{k} := \mathrm{k} - 3 \textbf{ od}$$

In this case an easy operational argument shows that

$$\begin{aligned} & \mathit{wev}.(\mathrm{k} = 10).S.(\mathrm{k} = 0) \\ = \quad & (\mathrm{k} \geq 10 \wedge \mathrm{k} \textbf{ mod } 3 = 1) \quad \vee \quad (\mathrm{k} \geq 0 \wedge \mathrm{k} \textbf{ mod } 3 = 0) \end{aligned}$$

Our aim is to provide formal proof rules that can be used effectively to prove such assertions in a calculational style without operational reasoning.

Nonterminating programs are primarily of practical interest when they are reactive, i.e., interact with the environment during their execution. The calculus presented here does not specifically support reactive programs. Yet, by taking input streams and output streams as part of the state space, it is possible to use the calculus to specify reactive programs. Notice that the precondition may be a constraint on the input stream, which need not be known at the time that the program is started. We nevertheless maintain the word precondition, since its logical role has not been changed.

Relation to Previous Work

Temporal preconditions for procedures were introduced first by Morris in the paper [Mor90]. In that paper, all recursion is simple (nonmutual) tail-recursion and, more importantly, the only intermediate states considered are those in which recursive calls occur. In [Luk91] and [LuS92], the runtime semantics of while–programs is characterized by the sequence of all intermediate states, possibly including the final state. This has the effect that many states are treated twice (once as final, once as initial); therefore the formulae are bigger than necessary. In this paper (as in [HeR93]), we generalize the approach of Lukkien and Van de Snepscheut to (mutually) recursive procedures, but we restrict the intermediate states. For us an intermediate state of a computation is one in which a simple command is started.

Novel Contributions

The semantic functions to describe the runtime behaviour are defined as extreme solutions of certain fixpoint equations. In each case the fixpoint equation is easily obtained, but it is hard to decide which solution must be chosen. Prompted by one of the referees we prove our choices in an Appendix that describes the operational semantics.

The definitions of the semantic functions as extreme fixpoints are the starting point of a calculus, which contains healthiness laws and proof rules. The proofs

of these laws and rules are standard variations of the proofs we gave for the postcondition semantics in [Hes92], Chapter 4. So, the main step forward in this paper is the application of the groundwork done before to the runtime semantics of recursive procedures. The present paper is a companion of [HeR93]: there we concentrated on proofs of healthiness laws. Here we give the operational semantics and we provide proof rules. Moreover, we extend the repertoire of [HeR93] by introducing two new semantic functions: *wep* to characterize errorfree computations and *winv* to characterize stability. We give examples in which the new functions are used to specify runtime behaviour and examples in which the proof rules are used to prove such specifications.

Overview of the Paper

In Section 2, we discuss notations and give some preliminary material on functions and orderings. The programming language is presented in Section 3. It is a more elegant version of the language used in [Hes92]. In Section 4, we introduce the runtime semantic functions. In particular, we give the types and the informal meaning, and we present three simple examples. In Section 5, we investigate the behaviour of the semantic functions with respect to sequential composition and nondeterminate choice. Section 6 contains the construction of the functions *wp*, *wlp*, *wep*, *wev*, *winv*, and *wto*, as extreme solutions of certain fixpoint equations.

Section 7 contains a number of healthiness laws. In Section 8, we give three proof rules. Two of them resemble Hoare's Induction Rule for partial correctness of recursive procedures. The third rule is a generalization of the Main Repetition Theorem. We give some examples to show how these rules can be used. Conclusions are drawn in Section 9. We conclude with an Appendix where we present the operational semantics and justify the extreme fixpoint definitions of the semantic functions introduced.

2. Notations and Functions

Function application is denoted by means of the infix operator ".", which binds to the left. In this way, currying is allowed. For example,

$$wto.p.q.c.r = (((wto.p).q).c).r$$

For sets X and Y, the set of functions from X to Y is denoted by $X \to Y$ and also by Y^X. The operator "$\to$" binds to the right, so that

$$X \to Y \to Z \quad = \quad X \to (Y \to Z)$$

If U is a subset of X and $f \in X \to Y$, then $f|U$ is the restriction to U, which is an element of $U \to Y$.

We write $\mathbb{P}$ to denote the set of the predicates on the state space. For $p \in \mathbb{P}$, the assertion $[p]$ means that p holds everywhere on the state space, cf. [DiS90]. Predicate p is said to be *stronger* than q if and only if $[p \Rightarrow q]$. Occasionally, we use the notation

$$p \leq q \quad \equiv \quad [p \Rightarrow q]$$

We write $PT = (\mathbb{P} \to \mathbb{P})$. The elements of PT are called *predicate transformers*.

A function $f \in PT$ is called *monotonic* if and only if

$$(\forall\, p, q \in \mathbb{P} : [p \Rightarrow q] : [f.p \Rightarrow f.q])$$

We write MT to denote the set of the monotonic functions $f \in PT$.

If X is an ordered set and Y is an arbitrary set, the induced order on $Y \to X$ is defined by $f \leq g \equiv (\forall\, y \in Y :: f.y \leq g.y)$. The above order on $\mathbb{P}$ is a special case. The order on $\mathbb{P}$ induces an order on PT and hence on its subset MT, and various sets of the form $MT^Y = (Y \to MT)$.

It is wellknown that $\mathbb{P}$, PT and MT with these orders are complete lattices. The infimum or greatest lower bound ($\sqcap$) of a family of predicates $p.i$ with $i \in I$ is denoted $(\inf i :: p.i)$; it is the universal quantification $(\forall\, i :: p.i)$. Similarly, the supremum or least upper bound $(\sup i :: p.i)$ is the existential quantification $(\exists\, i :: p.i)$. The infimum of a family of predicate transformers $f.i$ with $i \in I$ is the predicate transformer given (argumentwise) by $(\inf i :: f.i).p = (\inf i :: f.i.p)$ for every predicate p. Similarly for the supremum.

Recall that the wellknown theorem of Knaster–Tarski asserts that, for any complete lattice W, a monotonic function $D \in W \to W$ has a least fixpoint and a greatest fixpoint. If D and D' are monotonic functions with least fixpoints x and x' then $D \leq D'$ implies $x \leq x'$ (similarly for greatest fixpoints).

3. Commands

In this section we present the programming language, together with the meanings of the operators and the simple commands, as yet only with respect to the postcondition semantics, i.e., the functions *wp* and *wlp*.

The programming language consists of commands. These are built from elementary commands and the *empty* command ε by means of the operators ";" for sequential composition and "[]" for demonic nondeterminate choice. We use the following syntax for commands. Let A be a set of symbols with $\varepsilon \notin A$. The elements of A are called *elementary commands*. Starting from A, we define the class *Cmd* of command expressions inductively by the clauses

- $A \subseteq Cmd$,
- $\varepsilon \in Cmd$,
- if $c, d \in Cmd$ then $(c;d) \in Cmd$,
- if $(i \in I :: c.i)$ is a nonempty family in *Cmd* then $([]\, i :: c.i) \in Cmd$.

The demonic nondeterminate choice of a family of two commands c and d is denoted by $c \,[]\, d$. We give the infix operator "[]" a lower priority than ";".

The semantics of the commands will be determined by a number of semantic functions. The first two of these functions are the wellknown functions *wp* and *wlp*, which are interpreted as follows. For a command c and a predicate r, condition *wp.c.r* is regarded as the precondition such that command c terminates in a state where r holds. Condition *wlp.c.r* is regarded as the precondition such that command c does not terminate or terminates in a state where r holds. The constant ε and the infix operators ";" and "[]" satisfy

$$\begin{aligned} &w.\varepsilon.r = r, \\ &w.(c;d).r = w.c.(w.d.r) \\ &w.(c \,[]\, d).r = w.c.r \wedge w.d.r \end{aligned} \tag{1}$$

for both $w = wp$ and $w = wlp$. These rules go back to Dijkstra [Dij76].

We assume that the set of commands A is the disjoint union of two sets S and H, which may be infinite. The elements of S are called *simple commands* and the restrictions $wp|S$ and $wlp|S$ are supposed to be given. For definiteness, we provide the following simple commands: assignments $x := e$ for program variable x and expression e, and guards $?b$ and assertions $!b$ for predicate b. The formal semantics of the assignment is given by

$$wp.(x := e).r = r_e^x \wedge def.e$$
$$wlp.(x := e).r = r_e^x \vee \neg def.e$$

where predicate $def.e$ expresses that e is welldefined. If e is not welldefined the operators $\wedge$ and $\vee$ are interpreted in the intuitive way, as has been justified in [Bij90].

The simple commands $?b$ and $!b$ test for the validity of predicate b and do not modify the state. The difference is that $!b$ forces an error if b does not hold, whereas $?b$ is not executed if b does not hold (in that case, $?b$ is said to perform a miracle). The formal semantics is given by

$$\begin{aligned} &wp.(?b).r = \neg b \vee r \\ &wlp.(?b).r = \neg b \vee r \\ &wp.(!b).r = b \wedge r \\ &wlp.(!b).r = \neg b \vee r \end{aligned} \tag{2}$$

As is well known, cf. [Hes92] Section 1.5, it follows that the conditional choice can be expressed by

$$\textbf{if } b \textbf{ then } c \textbf{ else } d \textbf{ fi} \quad = \quad (?b \ ; \ c \ [\!] \ ?\neg b \ ; \ d)$$

The elements of the set H are called *procedures* or *procedure names*. Every procedure $h \in H$ is supposed to be equipped with a body $\textbf{body}.h \in Cmd$. In this way recursion is possible. For example, a repetition

$$L \quad = \quad \textbf{while } b \textbf{ do } c \textbf{ od}$$

is an abbreviation of a recursive procedure $L \in H$ with

$$\textbf{body}.L \quad = \quad (?b ; c ; L \ [\!] \ ?\neg b)$$

We shall construct wp and wlp in such a way that they are functions $w \in Cmd \rightarrow MT$ that satisfy $w.h = w.(\textbf{body}.h)$ for all $h \in H$. This will be expressed by saying that w respects the declaration. The construction is postponed to Section 5 below.

4. Runtime Semantic Functions

In this section, we introduce the new semantic functions. The first one is an easy variation in between wp and wlp. The other three functions describe useful aspects of the runtime behaviour of the commands.

We assume that every simple command only performs one computation step that need not terminate: if it does not terminate, it is said to make an error. Since nontermination is potentially useful behaviour, errors must not be treated as equivalent to nontermination. We therefore treat error as a specific (harmful) form of nontermination and introduce a function wep that stands for the weakest errorfree precondition.

More precisely, *wep.c.r* is regarded as the weakest precondition such that execution of c is errorfree and that c does not terminate or terminates in a state that satisfies r. It follows that $wep.s = wp.s$ for every simple command s, and also that *wp* implies *wep* and that *wep* implies *wlp*. We will see that function *wlp* is not made superfluous by the introduction of *wep*. To summarise, the three functions have the types $wp, wlp, wep \in Cmd \to MT$ and the interpretations

wp.c.r : the precondition of termination in r
wep.c.r : the precondition of termination in r or errorfree nontermination
wlp.c.r : the precondition of termination in r or nontermination

The differences between the three functions *wp*, *wep*, and *wlp* can be illustrated by considering the interpretation of the six extreme cases:

[*wp.c.false*] : c performs a miracle
[*wp.c.true*] : c terminates
[*wep.c.false*] : c is errorfree and does not terminate
[*wep.c.true*] : c is errorfree
[*wlp.c.false*] : c does not terminate
[*wlp.c.true*] : always holds

Example. We use function *wep* to specify the precondition that a reactive program does not generate errors. Let `x` and `y` be integer variables and let `f` and `g` be infinite streams of integers. Let c be the command

```
while true do
      read (f,x) ;  read (g,y)
;     write (x div y)
od
```

Then *wp.c.true* = *false* and *wlp.c.false* = *true* since c never terminates. Since division by zero gives an error, *wep.c.true* is the predicate that all elements of stream `g` are nonzero. Notice that the precondition here concerns the input, which need not be known at the moment the program starts. □

Remark. Of course, as suggested by a referee, the absence of errors can also be discussed by explicit introduction of a program variable `error`. This however has the effect that most of the useful properties of the program only hold under the assumption ¬`error`. This might lead to an uncomfortable number of `error`s in the specification. It does not help to insert guards ?¬`error` in the program, for then the occurrence of an error "establishes" every postcondition, even ¬`error`. □

The other three functions to be introduced have the types

$wev \in \mathbb{P} \to Cmd \to MT$
$winv \in Cmd \to PT$
$wto \in \mathbb{P} \to \mathbb{P} \to Cmd \to MT$

The intended interpretations are as follows. Recall that for us an intermediate state of a computation is one in which a simple command is started. For a command c and predicates p, q, r, we interpret

wev.p.c.r : the precondition of ever p or termination in r
winv.c.r : the precondition such that, if ever r, subsequently always r
wto.p.q.c.r : the precondition such that, if ever p, subsequently ever q or termination in r (3)

Here, "ever p" means that the computation has an intermediate state where p holds. The phrase "subsequently always r" means that r holds in all subsequent states and that no error occurs.

Notice that *wto.p.q.c.r* implies that *every* intermediate state where p holds is followed (after one or more steps) by an intermediate state where q holds or by termination in r; errors are only allowed when *every* occurrence of p has been followed by some occurrence of q. For us, the term "subsequently" excludes simultaneity. So, with respect to *wev* and *wto*, we deviate slightly from the definitions of [Luk91] and [LuS92]. The reason for these deviations is that our choices give drastically simpler expressions for the *wev* and *wto* of simple commands (see also [HeR93]).

We now give two introductory examples. Since they are only intended to sharpen the intuition, we do not give proofs.

A Toy Example with a Practical Flavour

Consider a financial institution with own capital x. Let a be an array variable such that a.i holds the current balance of client i. Bank transfers of values y are repeatedly executed by command h given by

while *true* **do**
  read(i, j, y)
; **if** $\mathtt{y} \geq 0 \ \wedge \ \mathtt{y}+1 \leq \mathtt{a.i}$ **then**
    $\mathtt{b} := \mathit{false}$
  ; $\mathtt{a.i} := \mathtt{a.i} - \mathtt{y} - 1$
  ; $\mathtt{a.j} := \mathtt{a.j} + \mathtt{y}$
  ; $\mathtt{x} := \mathtt{x} + 1$
  ; $\mathtt{b} := \mathit{true}$
  else ε **fi**
od

The bank is interested now in the validity of the following propositions

[*winv.h.*$(\mathtt{x} \geq C)$] : the own capital never decreases
[*winv.h.*$(\mathtt{a.}i \geq 0)$] : client i never becomes a debtor

If we define $p \equiv \mathtt{b} \ \wedge \ \mathtt{x} + \sum_i \mathtt{a.}i = D$, the invariance of the total amount of money can be expressed by

[*wto.p.p.h.false*]

Finally, a negative transfer only occurs when $\neg\mathtt{b} \wedge \mathtt{y} < 0$. So, in order to show that no client can affect a negative transfer, we need to have invariant validity of $r \equiv \mathtt{b} \vee \mathtt{y} \geq 0$. This follows from

$[\mathtt{b} \ \Rightarrow \ r \ \wedge \ \mathit{winv.h.r}]$ □

An Example in Which all Six Functions Occur

Let k be an integer program variable. Consider the repetition

L = **while** $\mathtt{k} \neq 0$ **do** $!(\mathtt{k} \neq 7)$; $\mathtt{k} := \mathtt{k} - 1$ **od**

Command L never terminates with $\mathtt{k} = 1$:

$$wp.L.(\mathtt{k} = 1) \quad = \quad false$$

Errorfree nontermination occurs if and only if $\mathtt{k} < 0$:

$$wep.L.(\mathtt{k} = 1) \quad = \quad (\mathtt{k} < 0)$$

Nontermination (with error) also occurs if $\mathtt{k} \geq 7$:

$$wlp.L.(\mathtt{k} = 1) \quad = \quad (\mathtt{k} < 0 \quad \vee \quad \mathtt{k} \geq 7)$$

The value 7 is reached if and only if $\mathtt{k} \geq 7$; termination with postcondition *true* is reached if and only if $0 \leq \mathtt{k} < 7$. These assertions are combined in:

$$wev.(\mathtt{k} = 7).L.true \quad = \quad (\mathtt{k} \geq 0)$$

If $\mathtt{k} < 0$, the condition $\mathtt{k} \geq 0$ is never reached; if $0 \leq \mathtt{k} < 7$, condition $\mathtt{k} \geq 0$ remains valid until termination; otherwise an error occurs:

$$winv.L.(\mathtt{k} \geq 0) \quad = \quad (\mathtt{k} < 7)$$

If $\mathtt{k} = -1$ initially, the condition $\mathtt{k} \geq -1$ is broken in the first step; $\mathtt{k} \geq 7$ implies $\mathtt{k} \geq -1$ but leads to an error; in this way we obtain

$$winv.L.(\mathtt{k} \geq -1) \quad = \quad (\mathtt{k} < 7 \quad \wedge \quad \mathtt{k} \neq -1)$$

If $\mathtt{k} \geq 7$, the condition $\mathtt{k} > 2$ holds but $\mathtt{k} = 1$ is not reached since an error occurs; otherwise, either $\mathtt{k} = 1$ is reached or $\mathtt{k} > 2$ is not reached:

$$wto.(\mathtt{k} > 2).(\mathtt{k} = 1).L.false \quad = \quad (\mathtt{k} < 7)$$

The two cases for *winv* show that the predicate transformer *winv.L* is not monotonic. □

5. Properties of ϵ, Composition and Choice

In this section we postulate the runtime properties of sequential composition and demonic choice. The properties are viewed as properties of the functions. Abstraction then leads to the concepts of multiplicative functions, homomorphisms, and accumulators.

The empty command ε is not a simple command and therefore has no intermediate states. It thus satisfies

$$\begin{array}{lll} wep.\varepsilon.r & = & wev.p.\varepsilon.r \quad = \quad r \\ winv.\varepsilon.r & = & wto.p.q.\varepsilon.r \quad = \quad true \end{array} \tag{4}$$

The new semantic functions are supposed to satisfy the following rules for the sequential composition:

$$\begin{array}{lll} wep.(c;d).r & = & wep.c.(wep.d.r) \\ wev.p.(c;d).r & = & wev.p.c.(wev.p.d.r) \\ winv.(c;d).r & = & winv.c.r \quad \wedge \quad wlp.c.(winv.d.r) \\ wto.p.q.(c;d).r & = & wto.p.q.c.(wev.q.d.r) \quad \wedge \quad wlp.c.(wto.p.q.d.r) \end{array} \tag{5}$$

The reader may try and convince himself of the plausability of these rules. The rules for *wev* and *wto* are due to Lukkien [Luk91]. Lukkien proves them in an operational semantics based on sequences of states. It is easy to extend the

arguments to *wep* and *winv*. In the rules for *winv* and *wto*, function *wlp* must not be replaced by *wep*. For *winv*, this can be seen as follows: if, during execution of c, predicate $\neg r$ keeps valid until an error occurs, then $winv.(c;d).r$ is satisfied but $wep.c.(winv.d.r)$ is false. A similar argument holds for *wto*.

We call a function $w \in Cmd \to PT$ *multiplicative* if and only if $w.\varepsilon.r = r$ and $w.(c;d).r = w.c.(w.d.r)$ for all $c, d \in Cmd$ and all $r \in \mathbb{P}$, or equivalently, if

$$w.\varepsilon = id \quad \text{(the identity function)}$$
$$w.(c;d) = w.c \circ w.d \quad \text{for all } c,d \in Cmd \tag{6}$$

In view of (1), (4), and (5), we shall construct *wp*, *wlp*, *wep*, and *wev.p* as multiplicative functions.

A function $w \in Cmd \to PT$ is said to *respect demonic choice* if and only if

$$w.(\,[\!]\, i \in I :: c.i).r = (\forall\, i \in I :: w.(c.i).r)$$

for every nonempty family of commands $(i \in I :: c.i)$ and every predicate r. We shall construct the functions *wp*, *wlp*, *wep*, *wev.p*, *winv* and *wto.p.q* in such a way that they respect demonic choice.

Let us define a function w to be a *homomorphism* if and only if $w \in Cmd \to MT$ and w is multiplicative and respects demonic choice. As a first example, we define the function

$$kd \in Cmd \to MT \text{ given by } kd.c.r = r \tag{7}$$

This function is easily seen to be a homomorphism; here we use that the demonic choice is defined for nonempty families only. Below, we construct *wp*, *wlp*, *wep*, and *wev.p* as homomorphisms.

We now give a unified view of the last two formulae of (5). Let $w \in Cmd \to MT$ be a homomorphism. A function $g \in Cmd \to PT$ is called a *w–accumulator* if and only if it respects demonic choice and satisfies, for all $c, d \in Cmd$ and $r \in \mathbb{P}$,

$$\begin{aligned} g.\varepsilon.r &= true \\ g.(c;d).r &= g.c.(w.d.r) \;\wedge\; wlp.c.(g.d.r) \end{aligned} \tag{8}$$

In view of (4) and (5), we shall construct the functions *winv* and *wto.p.q* in such a way that *winv* is a *kd*–accumulator and that *wto.p.q* is a *wev.q*–accumulator.

6. Construction of the Semantics

In this section we construct the semantic functions *wp*, *wlp*, *wep*, *wev*, *winv*, and *wto*. This construction uses structural induction over the commands. In each case, the first step is to construct the function for the simple commands. The behaviour with respect to composition and choice then yields a definition for all commands, provided we have a definition for the procedure names. As is usual, the idea that a procedure name must be indistinguishable from its body leads to a fixpoint equation. This fixpoint equation has a least and a greatest solution, by the Theorem of Knaster–Tarski. Ultimately the definition is the choice of either the least fixpoint or the greatest one. The justification of these choices requires an operational semantics and is therefore postponed to the Appendix.

Recall that S is the set of simple commands, which contains the assignments, the guards $?b$, and the assertions $!b$. Also recall that *wp.s* and *wlp.s* are given for all simple commands $s \in S$. In order to construct *wp* and *wlp* for all commands,

we apply renaming and define functions $ws_0, ws_1 \in S \to MT$ by $ws_0 = (wp|S)$ and $ws_1 = (wlp|S)$. So, now the semantics of the simple commands are given by ws_0 and ws_1.

Recursive procedures are introduced as follows. Recall that the function **body** determines, for every procedure $h \in H$, the body **body**.$h \in Cmd$. A function $g \in Cmd \to PT$ is said to *respect the declaration* if and only if $g.h = g.(\mathbf{body}.h)$ for every $h \in H$, or equivalently $(g|H) = g \circ \mathbf{body}$.

Remark. It is not difficult to construct a function g that does not respect the declaration. For example, it could be such that $g.h$ is the identity for all $h \in H$. □

We propose to construct the functions *wp*, *wlp*, *wep*, and *wev.p* in such a way that they are homomorphisms that respect the declaration.

By induction over the structure of *Cmd*, one can easily prove that, for every function $v \in A \to MT$, there is precisely one homomorphism $w \in Cmd \to MT$ with restriction $(w|A) = v$. This function is called the homomorphic extension $hom.v = w$ of v. Moreover, the function *hom* is monotonic: if $u \leq v$ then $hom.u \leq hom.v$. In order to prove this, one uses induction over the structure of *Cmd* and monotonicity of the functions $hom.u.c \in MT$. Notice that *hom* is a function $(A \to MT) \to (Cmd \to MT)$.

Let a function $u \in MT^S$ be given, which is to be interpreted as providing the meaning of the simple commands. We now want to extend u to a homomorphism that respects the declaration. For any $x \in MT^H$ we write $u + x$ to denote the function in $A \to MT$ with restrictions u on S and x on H. One easily verifies:

Theorem. A function $w \in Cmd \to MT$ is a homomorphism that extends $u \in S \to MT$ and respects the declaration if and only if $w = hom.(u + (w \circ \mathbf{body}))$. The latter condition is equivalent to $w = D.u.w$ where function D is given by

$$D.u.w = hom.(u + (w \circ \mathbf{body}))$$

Notice that, since $w \in Cmd \to MT$, we have $w \circ \mathbf{body} \in MT^H$ and hence $hom.(u + (w \circ \mathbf{body})) \in Cmd \to MT$.

Function $D.u \in (Cmd \to MT) \to (Cmd \to MT)$ is easily seen to be monotonic. By the theorem of Knaster–Tarski, the function $D.u$ has a least fixpoint and a greatest fixpoint. We now define $\mu.u$ as the least fixpoint of D and $\nu.u$ as the greatest fixpoint of D. It follows that $\mu.u$ and $\nu.u$ are homomorphisms $Cmd \to MT$, which extend function u and respect the declaration. One can prove that, if $u \leq u'$ in MT^S, then $\mu.u \leq \mu.u'$ and $\nu.u \leq \nu.u'$.

The functions *wp*, *wep* and *wlp* are defined as the homomorphisms

$$wp = \mu.ws_0\,, \quad wep = \nu.ws_0\,, \quad wlp = \nu.ws_1 \tag{9}$$

The choices for *wp* and *wlp* are wellknown, see [DiS90] and [dRo76]. All three choices will be justified in the Appendix.

It follows from these definitions that indeed *wp*, *wep* and *wlp* are homomorphisms that respect the declaration and that $wp|S = wep|S = ws_0$ and $wlp|S = ws_1$.

Remark. In [Hes92] and [HeR93], the restrictions $wp|H$ and $wlp|H$ are constructed as extreme fixpoints and then extended to the set of all commands. The present construction is equivalent and more elegant. It was proposed by R.M. Dijkstra. The same kind of modification has been made in the constructions below of *wev* and *wto*. □

Since *wev.p* is supposed to be a homomorphism that respects the declaration and since *wev.false* should be equal to *wp*, the definition of *wp* suggests to define

$$wev.p = \mu.(wvs.p) \tag{10}$$

where $wvs.p = (wev.p|S)$ is still to be determined. Since *wev.p.s.r* expresses "ever p or termination in r" and since we regard the initial state as the only intermediate state of a simple command, we define

$$wvs.p.s.r \quad = \quad p \vee ws_0.s.r \quad \text{for } p,\ r \in \mathbb{P} \text{ and } s \in S. \tag{11}$$

We give an operational justification of the definitions (10) and (11) in the Appendix. Definition (10) implies that *wev.p* is a homomorphism that respects the declaration.

We now turn to the construction of *winv* and *wto*. Let us first consider the restriction to simple commands. In view of (2), we define $wsi \in S \to PT$ by

$$wsi.s.r \quad = \quad \neg r \vee ws_0.s.r \quad \text{for } s \in S \text{ and } r \in \mathbb{P} \tag{12}$$

and we require $winv|S = wsi$. With respect to *wto*, we recall that, for a simple command, the initial state is the only intermediate state and that there is no subsequent intermediate state. We therefore require $wto.p.q.s.r = \neg p \vee ws_0.s.r$ and hence $wto.p.q|S = wvs.(\neg p)$.

Now let $w \in Cmd \to MT$ be a homomorphism. The construction of *Cmd* from A is such that, for every function $v \in A \to PT$, there is precisely one w–accumulator $g \in Cmd \to PT$ with restriction $(g|A) = v$. Let this extension be denoted $acc.w.v = g$. By induction over the structure of *Cmd*, one can prove that function $acc.w \in (A \to PT) \to (Cmd \to PT)$ is monotonic.

Let the meaning of *winv* or *wto* on the simple commands be given by a function $u \in S \to PT$. For any function $x \in PT^H$, we have a combined function $u + x \in A \to PT$ and hence a w–accumulator $acc.w.(u + x) \in Cmd \to PT$. Again one can verify:

Theorem. A function $g \in Cmd \to PT$ is a w–accumulator that extends $u \in S \to PT$ and respects the declaration if and only if $g = acc.w.(u + (g \circ \mathbf{body}))$. The latter condition is equivalent to $g = E.w.u.g$ where function E is given by

$$E.w.u.g = acc.w.(u + (g \circ \mathbf{body}))$$

The function $E.w.u \in (Cmd \to PT) \to (Cmd \to PT)$ is monotonic. We define $\tau.w.u$ to be the greatest fixpoint of $E.w.u$ in $Cmd \to PT$. By construction, this function $\tau.w.u$ is the greatest w–accumulator that respects the declaration and restricts to $u \in S \to PT$. Now the functions *winv* and *wto* are defined by

$$\begin{aligned} winv \quad &= \quad \tau.kd.wsi \\ wto.p.q \quad &= \quad \tau.(wev.q).(wvs.(\neg p)) \end{aligned} \tag{13}$$

Indeed, in this way, *winv* is a *kd*–accumulator that respects the declaration and *wto.p.q* is a *wev.q*–accumulator that respects the declaration. Moreover, for a simple command $s \in S$, we have

$$\begin{aligned} winv.s.r \quad &= \quad wsi.s.r \quad = \quad \neg r \vee ws_0.s.r \\ wto.p.q.s.r \quad &= \quad \neg p \vee ws_0.s.r \end{aligned}$$

These formulae correspond to the informal description (3). We refer to the Appendix for a proof that we must take the greatest fixpoints. Lukkien gave a proof for the case of *wto* of while–programs in [Luk91], Theorem 62.

One can easily prove that, if $u \in MT^S$, then $\tau.w.u \in Cmd \to MT$. It follows that $wto.p.q.c \in MT$ for every $c \in Cmd$ and $p, q \in \mathbb{P}$.

7. Healthiness Laws and Associativity

In this section we discuss a number of healthiness laws. The most important result is that sequential composition is associative and that ε is its unit element. The proof of these facts requires universal conjunctivity of *wlp*, which is one of the classical healthiness laws. Other healthiness laws are mentioned briefly.

Recall that a function $f \in PT$ is called *universally conjunctive* if and only if

$$f.(\forall\ p \in U :: p) = (\forall\ p \in U :: f.p)$$

for every subset U of $\mathbb{P}$. We need the following postulate concerning the semantics of the simple commands:

Function $ws_1.s$ is universally conjunctive for every $s \in S$. (14)

This postulate is one of the healthiness laws of [DiS90], [Hes92], and [HeR93]. It is used to prove

Theorem. For every command $c \in Cmd$, the function $wlp.c$ is universally conjunctive. In particular, $wlp.c.(p \wedge q) = wlp.c.p \wedge wlp.c.q$ and $[\ wlp.c.true]$. (15)

Proof. See [Hes92] Theorem 4(30) or [HeR93] Theorem (19). □

We now use Theorem (15) to prove the unit property of ε and the associativity of sequential composition, both with respect to the semantic functions under consideration.

Theorem. Let $w \in Cmd \to MT$ be a homomorphism and let $c, d, e \in Cmd$.

(a) $w.(\varepsilon; c) = w.c = w.(c; \varepsilon)$.

(b) $w.(c; (d; e)) = w.((c; d); e)$.

If g is a w–accumulator then

(c) $g.(\varepsilon; c) = g.c = g.(c; \varepsilon)$ and $g.(c; (d; e)) = g.((c; d); e)$. (16)

Proof.

(a) This follows from the fact that w is multiplicative and that the identity function *id* is the unit for function composition.

(b) Composition of functions is associative, so it suffices to observe that both sides reduce to the composition $w.c \circ w.d \circ w.e$.

(c) The equality $g.(c; \varepsilon) = g.c$ is proven in

$$\begin{array}{ll} & g.(c;\varepsilon).r \\ = & \{(8)\} \\ & g.c.(w.\varepsilon.r) \ \wedge \ wlp.c.(g.\varepsilon.r) \\ = & \{(8)\} \\ & g.c.r \ \wedge \ wlp.c.true \\ = & \{(15)\} \\ & g.c.r \end{array}$$

The verification of $g.(\varepsilon; c) = g.c$ is similar and simpler, and is therefore left

to the reader. With respect to the associativity, it suffices to verify that, for every $r \in \mathbb{P}$,

$$
\begin{array}{ll}
 & g.(c;(d;e)).r \\
= & \{(8)\} \\
 & g.c.(w.(d;e).r) \ \wedge \ wlp.c.(g.(d;e).r) \\
= & \{(6) \text{ and } (8)\} \\
 & g.c.(w.d.(w.e.r)) \ \wedge \ wlp.c.(g.d.(w.e.r) \wedge wlp.d.(g.e.r)) \\
= & \{(15); \text{this is the reason to postulate } (14)\} \\
 & g.c.(w.d.(w.e.r)) \ \wedge \ wlp.c.(g.d.(w.e.r)) \ \wedge \ wlp.c.(wlp.d.(g.e.r)) \\
= & \{(8) \text{ and } (6)\} \\
 & g.(c;d).(w.e.r) \ \wedge \ wlp.(c;d).(g.e.r) \\
= & \{(8)\} \\
 & g.((c;d);e).r \qquad \square
\end{array}
$$

This proves that the six semantic functions respect the semantic equality $c;(d;e) \cong (c;d);e$. In [Hes92], we enforced associativity of the sequential composition syntactically. For our present purposes that choice is less convenient.

Remark. The above theorem is not as innocent as it looks. In fact, several authors ([BvW90], [Mrg90], [MoG90], [Mor87], see also [Hes94]), have proposed an operator for angelic choice, say "$\diamond$", with the property that

$$wp.(c \diamond d).p = wp.c.p \vee wp.d.p$$

The introduction of this operator in the theory almost inevitably leads to violation of Theorem (15). Consequently, the proof of part (c) of Theorem (16) would fail and, presumably, part (c) would not be valid. We have therefore refrained from introducing this operator here. $\square$

There is a vast number of other healthiness laws and not all of them are equally important. We mention the ones we have come across, but we do not prove them here. Most of them have been proven in [HeR93]. All of them can easily be verified in the operational semantics of the Appendix. We assume that the simple commands satisfy what might be called the termination postulate:

$$ws_0.s.r = ws_0.s.true \wedge ws_1.s.r \quad \text{for all } s \in S \text{ and } r \in \mathbb{P}$$

With this postulate it is possible to prove the two termination laws:

$$
\begin{array}{l}
wp.c.r = wp.c.true \wedge wlp.c.r \\
wep.c.r = wep.c.true \wedge wlp.c.r \quad \text{for all } c \in Cmd \text{ and } r \in \mathbb{P}
\end{array}
$$

Immediately from the definitions we get

$$
\begin{array}{l}
[\, wp.c.r \Rightarrow wep.c.r \,] \\
wp = wev.false
\end{array}
$$

It is easy to prove the monotonicity rules:

$$
\begin{array}{l}
[\, p \Rightarrow p' \,] \ \Rightarrow \ [\, wev.p.c.r \Rightarrow wev.p'.c.r \,] \\
[\, p' \Rightarrow p \,] \ \wedge \ [\, q \Rightarrow q' \,] \ \Rightarrow \ [\, wto.p.q.c.r \Rightarrow wto.p'.q'.c.r \,]
\end{array}
$$

More interesting are the transitivity rules:

$$
\begin{array}{l}
[\, wev.(p \vee q).c.r \wedge wto.p.q.c.r \ \Rightarrow \ wev.q.c.r \,] \\
[\, wto.p.(q \vee r).c.m \wedge wto.q.r.c.m \ \Rightarrow \ wto.p.r.c.m \,]
\end{array}
$$

The proofs of the last two formulae are highly nontrivial, see [HeR93].

The paper [HeR93] also contains healthiness laws concerning semantic functions *wlev* and *wlto*. These functions are liberal versions of *wev* and *wto*, defined by taking ws_1 and v instead of ws_0 and μ. These liberal versions are of theoretical interest, but they are often too liberal for actual specifications.

8. Proof Rules

The proof rules presented in this section serve to prove specifications of recursive procedures $h.i$ by means of preconditions $p.i$ and postconditions $q.i$. The rules are induction principles: in order to prove the validity of the specification, it suffices to prove that the procedure bodies satisfy the specification under assumption that the recursive calls satisfy the specification.

The first rule, (16), is a generalization of Hoare's Induction Rule: in this case the proof obligation must be met for an abstraction of the semantic function under consideration. The second rule, (22), is a generalization of the Main Repetion Theorem. So, a variant functions is needed to force termination. Rule (24) again is a variation of Hoare's Rule. The rules may be compared with the rules for the postcondition semantics, as discussed in [Hes92] Chapter 2 and [Hes93].

All three rules are formulated in such a way that arbitrary families of Hoare triples $\langle p.i, h.i, q.i \rangle$ can be dealt with. This is useful if parameters or specification values occur, see [Hes93] and [Hes92] Chapter 2. In the examples here, we do not use this additional power.

As announced above, the first rule is a generalization and rephrasing of Hoare's classical induction rule for partial correctness, cf. [Hoa71]. It applies to an arbitrary greatest fixpoint homomorphism $v.u$.

Theorem. Let $(i \in I :: h.i)$ be a family of procedure names and let $(i \in I :: p.i)$ and $(i \in I :: q.i)$ be families of predicates. Let $u \in S \to MT$ be a function such that, for every homomorphism w with $(w|S) = u$,

$$(\forall\, i :: [\, p.i \;\Rightarrow\; w.(h.i).(q.i)\,]\,)$$
$$\Rightarrow\quad (\forall\, i :: [\, p.i \;\Rightarrow\; w.(\mathbf{body}.(h.i)).(q.i)\,]\,)$$

Then $[\, p.i \;\Rightarrow\; v.u.(h.i).(q.i)\,]$ for every $i \in I$. (17)

Proof. This is proven in the same way as [Hes92] Theorem 4(44). □

Remark. In Theorem (17), the goal is an assertion about $v.u$, but the proof obligation is an implication concerning an abstraction w of $v.u$. The antecedent of this implication is usually called the induction hypothesis. It is not sufficient to prove the proof obligation for the special case $w = v.u$. For, in that special case, the proof obligation holds trivially if the goal is false. □

As an application of this rule, we present a simple example. Let `t` be an integer program variable and let procedure h be declared by

$$\mathbf{body}.h \;=\; (\,!\,\mathtt{t} > 0 \;;\; \mathtt{t} := \mathtt{t} + 1 \;[\!]\; h \;;\; \mathtt{t} := \mathtt{t} - 1 \;;\; h\,)$$

We claim that $\mathtt{t} > 0$ implies that procedure h is errorfree and does not terminate

or terminates with $t > 1$, i.e.,

$$[\, t > 0 \;\Rightarrow\; wep.h.(t > 1)\,] \tag{18}$$

Since $wep = v.ws_0$, Theorem (17) can be applied with $u = ws_0$. All families are singletons, the precondition is $p = (t > 0)$ and the postcondition is $q = (t > 1)$. Now Theorem (17) implies that it suffices to prove that every homomorphism w with $(w|S) = ws_0$ satisfies

$$[\, t > 0 \;\Rightarrow\; w.h.(t > 1)\,] \;\Rightarrow\; [\, t > 0 \;\Rightarrow\; w.(\mathbf{body}.h).(t > 1)\,] \tag{19}$$

We regard the antecedent of (19) as the induction hypothesis and we prove the consequent of (19) in the following calculation:

$$\begin{array}{ll} & w.(\mathbf{body}.h)(t > 1) \\ = & \quad \{\text{declaration of } h,\ w \text{ is a homomorphism}\} \\ & w.(\,!\,t > 0).(w.(t := t + 1).(t > 1)) \\ & \wedge\ \ w.h.(w.(t := t - 1).(w.h.(t > 1))) \\ \Leftarrow & \quad \{(w|S) = ws_0, \text{ i.e., } wp|S;\ \text{induction hypothesis, monotonicity}\} \\ & wp.(\,!\,t > 0).(wp.(t := t + 1).(t > 1)) \\ & \wedge\ \ w.h.(wp.(t := t - 1).(t > 0)) \\ = & \quad \{(1) \text{ and } wp \text{ of assignment}\} \\ & t > 0 \ \wedge\ \ t + 1 > 1 \ \wedge\ \ w.h.(t - 1 > 0) \\ \Leftarrow & \quad \{\text{calculus and induction hypothesis}\} \\ & t > 0 \end{array}$$

This proves claim (18).

For the discussion of the other proof rules, we use a slightly more difficult example. Let t be an integer program variable and let procedure h be declared by

$$\begin{array}{l} \mathbf{body}.h \;= \\ (\ ?(t \le 0)\ ;\ t := t + b \\ \ [\!]\ ?(t > 0)\ ;\ t := t - c\ ;\ h\ ;\ h) \end{array} \tag{20}$$

where b and c are integer constants with $0 < c \le b$. One can argue operationally about this procedure, but such arguments are tricky and error prone. Let us only say that the operational intuition suggests the following claims.

If $t > 0$ initially, then procedure h does not terminate:

$$[\, t > 0 \;\Rightarrow\; wlp.h.false\,]$$

Every execution of h reaches some state where $t \le 0$ holds:

$$[\, wev.(t \le 0).h.false\,] \tag{21}$$

If during execution, ever $t \le b$ holds, then $t \le b$ remains valid:

$$[\, winv.h.(t \le b)\,] \tag{22}$$

The first claim can again be proven by means of Theorem (17), now with $u = ws_1$. We therefore leave this as an exercise to the reader. We proceed by providing proof rules that allow calculational proofs of the claims (21) and (22). The rule for homomorphisms like $wev.p$ is as follows (see also [Hes93]).

Theorem. Let w be a homomorphism that respects the declaration. Let $(i \in I :: h.i)$ be a family of procedure names and let $(i \in I :: p.i)$ and $(i \in I :: q.i)$

be families of predicates. Let $(i \in I :: vf.i)$ be a family of integer valued state functions such that for every integer m:

$$(\forall\, i :: [\, p.i \wedge vf.i < m \wedge m \geq 0 \;\Rightarrow\; w.(h.i).(q.i)\,])$$
$$\Rightarrow\quad (\forall\, i :: [\, p.i \wedge vf.i = m \;\Rightarrow\; w.(\mathbf{body}.(h.i)).(q.i)\,])$$

Then $[\, p.i \;\Rightarrow\; w.(h.i).(q.i)\,]$ for every $i \in I$. (23)

Proof. The proof is an immediate generalization of [Hes92] Theorem 2(16). □

Remark. As suggested by a referee, Theorem (23) can be illustrated by showing that it is a generalization of the main repetition theorem. This has been done in Section 2.8 of [Hes92]. □

We here use Theorem (23) to prove formula (21) in the example of procedure h declared in (20). So we apply (23) with $w = wev.(\mathtt{t} \leq 0)$. The families are singletons with $p = true$ and $q = false$. We use the state function $vf = \mathtt{t}$. According to (23), it suffices to prove, for every integer m,

$$\begin{array}{l} \quad [\, \mathtt{t} < m \wedge m \geq 0 \;\Rightarrow\; w.h.false\,] \\ \Rightarrow\quad [\, \mathtt{t} = m \;\Rightarrow\; w.(\mathbf{body}.h).false\,] \end{array} \qquad (24)$$

We take the antecedent of (24) as an induction hypothesis and prove the consequent. First observe that, since $w = wev.(\mathtt{t} \leq 0)$, it follows from (2), (10) and (11) that

$$\begin{array}{ll} & w.(?(\mathtt{t} \leq 0)).r \\ = & \mathtt{t} \leq 0 \;\vee\; (\mathtt{t} > 0 \;\vee\; r) \\ = & true \end{array}$$

and

$$\begin{array}{ll} & w.(?(\mathtt{t} > 0)\,;\, \mathtt{t} := \mathtt{t} - c).r \\ = & \mathtt{t} \leq 0 \;\vee\; (\mathtt{t} \leq 0 \;\vee\; w.(\mathtt{t} := \mathtt{t} - c).r) \\ = & \mathtt{t} \leq 0 \;\vee\; ws_0.(\mathtt{t} := \mathtt{t} - c).r \end{array}$$

Now it remains to verify the consequent of (24) in

$$\begin{array}{ll} & w.(\mathbf{body}.h).false \\ = & \{\text{declaration } h \text{ in (20); first branch is } true\} \\ & \mathtt{t} \leq 0 \;\vee\; ws_0.(\mathtt{t} := \mathtt{t} - c).(w.(h;h).false) \\ \Leftarrow & \{\, [w.h.false \Leftarrow false] \text{ and } w \text{ homomorphism}\} \\ & \mathtt{t} \leq 0 \;\vee\; ws_0.(\mathtt{t} := \mathtt{t} - c).(w.h.false) \\ \Leftarrow & \{\text{induction hypothesis}\} \\ & \mathtt{t} \leq 0 \;\vee\; ws_0.(\mathtt{t} := \mathtt{t} - c).(\mathtt{t} < m \wedge m \geq 0) \\ = & \{\text{calculus}\} \\ & \mathtt{t} \leq 0 \;\vee\; (\mathtt{t} - c < m \wedge m \geq 0) \\ \Leftarrow & \{c > 0\} \\ & \mathtt{t} = m \end{array}$$

This concludes the proof of (21).

For the proof of (22), we need a proof rule for accumulators of the form $\tau.w.u$. This is a variation of Hoare's Induction Rule.

Theorem. Let w be a homomorphism. Let $(i \in I :: h.i)$ be a family of procedure names and let $(i \in I :: p.i)$ and $(i \in I :: q.i)$ be families of predicates. Let

$u \in S \to PT$ be a function such that, for every w–accumulator g with $(g|S) = u$,

$$(\forall\, i :: [\, p.i \;\Rightarrow\; g.(h.i).(q.i)\,])$$
$$\Rightarrow\quad (\forall\, i :: [\, p.i \;\Rightarrow\; g.(\mathbf{body}.(h.i)).(q.i)\,]).$$

Then $[\, p.i \;\Rightarrow\; \tau.w.u.(h.i).(q.i)\,]$ for every $i \in I$. (25)

Proof. Since $\tau.w.u$ is constructed by means of a greatest fixpoint, this assertion can be proven in the same way as [Hes92] Theorem 4(44). □

In the example of procedure h of declaration (20), Theorem (25) is applied to the function $winv = \tau.kd.wsi$, see (13). We use the rule to prove (22) in the form

$$[\, true \;\Rightarrow\; winv.h.(\mathtt{t} \le b)\,]$$

So we use Theorem (25) with singleton families and $p = true$ and $q = (\mathtt{t} \le b)$. It suffices to prove that, for every kd–accumulator g with $(g|S) = wsi$

$$[\, g.h.(\mathtt{t} \le b)\,] \;\Rightarrow\; [\, g.(\mathbf{body}.h).(\mathtt{t} \le b)\,]. \tag{26}$$

Let g be a kd–accumulator with $(g|S) = wsi$. One easily verifies with (12) that $wsi.(?p).r = true$ for all predicates p and r. It follows with (2), (7), and (8) that $g.(?p;c).r = (p \;\Rightarrow\; g.c.r)$ for every predicate p. Assuming the antecedent of (26), the consequent is proven in

$$\begin{array}{ll}
 & g.(\mathbf{body}.h).(\mathtt{t} \le b) \\
= & \quad \{(20)\} \\
 & g.(\ ?(\mathtt{t} \le 0)\ ;\ \mathtt{t} := \mathtt{t} + b \\
 & \quad [\!]\ ?(\mathtt{t} > 0)\ ;\ \mathtt{t} := \mathtt{t} - c\ ;\ h\ ;\ h).(\mathtt{t} \le b) \\
= & \quad \{\text{rule obtained above and calculus}\} \\
 & (\mathtt{t} \le 0 \;\Rightarrow\; g.(\mathtt{t} := \mathtt{t} + b).(\mathtt{t} \le b)) \\
\wedge & \ (\mathtt{t} > 0 \;\Rightarrow\; g.(\mathtt{t} := \mathtt{t} - c\ ;\ h\ ;\ h).(\mathtt{t} \le b)) \\
= & \quad \{\text{first conjunct is true, from } (g|S) = wsi \text{ and } (12)\} \\
 & \mathtt{t} > 0 \;\Rightarrow\; g.(\mathtt{t} := \mathtt{t} - c\ ;\ h\ ;\ h).(\mathtt{t} \le b) \\
= & \quad \{g \text{ is a } kd\text{–accumulator, } (7),\ (8)\} \\
 & \quad \mathtt{t} > 0 \;\Rightarrow \\
 & \quad g.(\mathtt{t} := \mathtt{t} - c).(\mathtt{t} \le b) \\
 & \wedge\ \ wlp.(\mathtt{t} := \mathtt{t} - c).(g.h.(\mathtt{t} \le b) \wedge wlp.h.(g.h.(\mathtt{t} \le b))) \\
= & \quad \{(g|S) = wsi,\ (12),\ \text{calculus and } c \ge 0\} \\
 & \mathtt{t} > 0 \;\Rightarrow \\
 & \quad wlp.(\mathtt{t} := \mathtt{t} - c).(g.h.(\mathtt{t} \le b) \wedge wlp.h.(g.h.(\mathtt{t} \le b))) \\
= & \quad \{\text{antecedent of (26), twice}\} \\
 & \mathtt{t} > 0 \;\Rightarrow\; wlp.(\mathtt{t} := \mathtt{t} - c).(wlp.h.true) \\
= & \quad \{(15),\ \text{calculus}\} \\
 & true\ .
\end{array}$$

This concludes the proof of (22).

9. Conclusions

We have shown that the theory of predicate transformation semantics, originally designed for the usual postcondition semantics, can also be used in an effective and elegant way to construct predicate transformation functions to describe the runtime semantics of (not necessarily terminating) recursive procedures.

These functions are constructed as extreme solutions of fixpoint equations.

This made it possible to use earlier work and thus to obtain proof rules that enable calculational proofs of temporal properties of (not necessarily terminating) recursive procedures. Even in simple cases, however, the calculations are quite long. It seems therefore that mechanical support will be indispensable for real applications.

Experience will have to show which predicate transformation functions are the most useful for specification of runtime behaviour. In [HeR93], we used a slightly different set of functions. Other functions could also be suggested.

References

[BvW90] Back, R. J. R. and Wright, J. von.: Refinement calculus, Part I: Sequential Nondeterministic Programs. In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds) Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430. Springer, Berlin, 1990, pp. 42–66.

[Bij90] Bijlsma, A.: Semantics of quasi-boolean expressions. In: W.H.J. Feijen et al. (eds.): Beauty is our business, a birthday salute to Edsger W. Dijkstra. Springer. 1990, pp. 27–35.

[Dij76] Dijkstra, E. W.: A discipline of programming. Prentice-Hall 1976.

[DiS90] Dijkstra, E. W. and Scholten, C. S.: Predicate calculus and program semantics. Springer. 1990.

[Hes88] Hesselink, W. H.: Interpretations of recursion under unbounded nondeterminacy. *Theoretical Computer Science* **59** (1988) 211–234.

[Hes92] Hesselink, W. H.: Programs, Recursion and Unbounded Choice, predicate transformation semantics and transformation rules. Cambridge University Press 1992.

[Hes93] Hesselink, W. H.: Proof rules for recursive procedures. *Formal Aspects of Computing* 5 (1993) 554–570.

[Hes94] Hesselink, W. H.: Nondeterminacy and recursion via stacks and games. *Theoretical Computer Science* 124 (1994) 273–295.

[HeR93] Hesselink, W. H. and Reinds, R.: Temporal preconditions of recursive procedures. In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): Semantics: Foundations and Applications. Proceedings of REX Workshop Beekbergen 1992. Springer Verlag 1993 (LNCS 666), pp. 236–260.

[Hoa71] Hoare, C. A. R.: Procedures and parameters: an axiomatic approach. In: Symposium on Semantics of Algorithmic Languages. (ed. E. Engeler), Springer Verlag (Lecture Notes in Math. 188) 1971, pp. 102–116.

[Luk91] Lukkien, J. J.: Parallel Program Design and Generalized Weakest Preconditions. Thesis, Groningen, 1991.

[LuS92] Lukkien, J. J. and van de Snepscheut, J. L. A.: Weakest preconditions for progress. *Formal Aspects of Computing* 4 (1992) 195–236.

[Mrg90] Morgan, C.: Programming from Specifications. Prentice Hall, 1990.

[MoG90] Morgan, C. and Gardiner, P. H. B.: Data refinement by calculation. *Acta Informatica* 27 (1990) 481–503.

[Mor87] Morris, J. M.: A theoretical basis for stepwise refinement and the programming calculus. *Science of Comp. Programming* 9 (1987) 287–306.

[Mor90] Morris, J. M.: Temporal predicate transformers and fair termination. *Acta Informatica* 27 (1990) 287–313.

[dRo76] de Roever, W. P.: Dijkstra's predicate transformer, non-determinism, recursion, and termination. In: Mathematical Foundations of Computer Science 1976 (Lect. Notes Comp. Sci., vol. 45) Berlin, Heidelberg, New York: Springer Verlag 1976, pp. 472–481.

Appendix: The Formal Operational Semantics

In order to justify the definitions in Section 6 of *wep*, *wev*, *winv*, *wto* as extreme fixpoints of certain operators, we describe the operational semantics. The main idea is to formalize the concept of computation by describing it in terms of pairs

(x, c) where x is the current state and c is the command that is still to be executed. Such a pair will be called a configuration. Every computation step corresponds to a transition from one configuration to another. The configurations will therefore form a directed graph. Every computation is a path in this graph.

This idea is formalized as follows. With respect to the commands, we treat ε as the unit for sequential composition and thus identify $\varepsilon; c = c = c; \varepsilon$ for every $c \in \mathit{Cmd}$. We also treat sequential composition as associative and therefore omit parentheses in $(c; d); e$ and $c; (d; e)$. These identifications are justified by Theorem (16).

We write X to denote the state space. Let $\mathbb{B}$ be the set of the truth values. So we have $\mathbb{P} = \mathbb{B}^X$, the set of the boolean functions on X. The value of a predicate $r \in \mathbb{P}$ in a state $x \in X$ is the boolean $r.x \in \mathbb{B}$.

The semantics of a simple command $s \in S$ are supposed to be given by an input–output relation $[\![s]\!] \subseteq X \times X$ and an error set $\mathit{Err}.s \subseteq X$: here $(x, y) \in [\![s]\!]$ means that s executed in state x may have result state y, and $x \in \mathit{Err}.s$ means that s executed in state x may make an error. Then we have

$$\begin{array}{lcl} \mathit{ws}_1.s.p.x & \equiv & (\forall\ y : (x, y) \in [\![s]\!] : p.y) \\ \mathit{ws}_0.s.p.x & \equiv & \mathit{ws}_1.s.p.x \ \wedge \ x \notin \mathit{Err}.s \end{array}$$

We now introduce the set of configurations. A *configuration* is either the error configuration $\perp$ or a pair (x, c) with $x \in X$ and $c \in \mathit{Cmd}$. In the latter case, we interpret c as a command still to be executed and to be started in state x. A configuration is called *final* if it is of the form (x, ε) with $x \in X$.

The set of the configurations is made into a directed graph by defining a transition relation "$\rightarrow$" between configurations. There are four types of transitions, namely

$$\begin{array}{lcll} (x,\ h; c) & \rightarrow & (x,\ \mathbf{body}.h; c) & \\ (x,\ ([\!]i \in I :: c.i); d) & \rightarrow & (x,\ c.j; d) & \text{for } j \in I \\ (x,\ s; c) & \rightarrow & (y, c) & \text{if } (x, y) \in [\![s]\!] \\ (x,\ s; c) & \rightarrow & \perp & \text{if } x \in \mathit{Err}.s \end{array}$$

for $x, y \in X$, and $h \in H$, and $c, d, e \in \mathit{Cmd}$, and all $c.i \in \mathit{Cmd}$, and $s \in S$.

A path in the configuration graph is called a *computation* if it is either infinite, or ends in the error configuration $\perp$, or ends in a final configuration. Only in the third case we say that the computation *terminates* (in the second case, it is *finite*, but does not terminate).

Remark. A configuration is called *failing* if it has no outgoing transitions and yet is not a final configuration and differs from the error configuration $\perp$. A configuration is failing if and if it is of the form $(x,\ s; c)$ with $s \in S$ and $(\forall\ y :: (x, y) \notin [\![s]\!])$ and $x \notin \mathit{Err}.s$. A path in the configuration graph that cannot be extended is either a computation or ends in a failing configuration. An executing mechanism that enters a failing configuration will have to backtrack. □

A configuration is said to *satisfy* predicate p if it is not the error configuration and its state component satisfies p. A configuration is called *simple* if it is of the form $(x,\ s; c)$ with $s \in S$ and $c \in \mathit{Cmd}$. A computation is called a *p–computation* if p holds in every simple configuration that occurs in it.

In order to justify the fixpoint definitions of the semantic functions, we abolish these definitions and replace them by definitions in terms of the operational semantics. We then prove that, in each case, the operationally defined function

is indeed the extreme solution of the fixpoint equation as announced. So, we now give the operational definitions of the semantic functions *wp*, *wlp*, *wep*, *wev*, *winv*, and *wto*. Compare the informal descriptions of (3).

- *wp.c.r.x* means that every computation starting in configuration (x, c) terminates in a final configuration where r holds.
- *wlp.c.r.x* means that every terminating computation starting in configuration (x, c) terminates in a final configuration where r holds.
- *wep.c.r.x* means that every finite computation starting in configuration (x, c) terminates in a final configuration where r holds.
- *wev.p.c.r.x* means that every $(\neg p)$–computation starting in configuration (x, c) terminates in a final configuration where r holds.
- *winv.c.r.x* means that, in every computation starting in configuration (x, c), if r holds in some configuration it holds in all subsequent configurations.
- *wto.p.q.c.r.x* means that, in every computation that starts in configuration (x, c), every simple configuration where p holds is followed (after one or more transitions) by a simple configuration where q holds or by termination in a final configuration where r holds.

The postulates of Section 5 are justified by the following result, the proof of which is surprisingly complicated.

Theorem. The functions *wp*, *wep*, *wlp*, *wev.p* are homomorphisms $Cmd \to MT$ that respect the declaration and that satisfy

$$wp|S = ws_0 \quad , \quad wlp|S = ws_1$$
$$wep|S = ws_0 \quad , \quad wev.p|S = wvs.p$$

Function *winv* is a *kd*–accumulator $Cmd \to PT$ that respects the declaration and satisfies $winv|S = wsi$. Function *wto.p.q* is a *wev.p*–accumulator $Cmd \to MT$ that respects the declaration and satisfies $wto.p.q|S = wvs.(\neg p)$.

The reader who wants to is invited to give a proof of this result. The ideas are not new, see for instance [Hes88], Section 2. We shall concentrate on the remaining part: the characterizations as extreme fixpoints.

Let a configuration be called *rewritable* if it is of the form $(x, d; e)$ where d is a procedure name or a demonic choice. In that case, the configuration has one or more transitions to configuration(s) with the same state component x (these transitions will be called *rewritings*). By inspection of the transition relation and the other relevant definitions, one can easily obtain the following two lemmas.

Lemma. Let (x, c) be a rewritable configuration. Let $r \in \mathbb{P}$ and let w be a homomorphism that respects the declaration.

(a) $w.c.r.x$ holds if and only if $w.d.r.x$ holds for every transition $(x, c) \to (x, d)$.

(b) If g is a w–accumulator that respects the declaration then $g.c.r.x$ holds if and only if $g.d.r.x$ holds for every transition $(x, c) \to (x, d)$. (27)

Lemma. Consider a simple configuration $(x, s; c)$ with $s \in S$ and $c \in Cmd$. Let w be a homomorphism and let $r \in \mathbb{P}$.

(a) Assume $w|S = ws_0$. Then $w.(s; c).r.x$ holds if and only if every transition from $(x, s; c)$ goes to a configuration (y, c) such that $w.c.r.y$ holds.

(b) Assume $w|S = ws_1$. Then $w.(s;c).r.x$ holds if and only if every transition from $(x,\ s;c)$ goes to $\perp$ or to a configuration (y,c) such that $w.c.r.y$ holds. (28)

Theorem. Let w be a homomorphism that respects the declaration. Let $c \in Cmd$ and $r \in \mathbb{P}$.

(a) If $w|S = ws_0$ then $[w.c.r \Rightarrow wep.c.r]$.
(b) If $w|S = ws_1$ then $[w.c.r \Rightarrow wlp.c.r]$. (29)

Proof.

(a) According to the definition of *wep*, it suffices to show that $w.c.r.x$ implies that every finite computation starting in (x,c) terminates in a final configuration where r holds. This is proven as follows. By induction, the lemmas (27) and (28) imply that all configurations of the computation differ from $\perp$ and are of the form (y,d) with $w.d.r.y$. Since it is a finite computation, there is a last configuration (y,d). The definition of computation implies that the last configuration (y,d) is a final one with $d = \varepsilon$. Now $w.d.r.y$ implies $r.y$.
(b) Here one uses a largely similar argument in which "finite computation" has been replaced by "terminating computation". □

Since *wep* is itself a homomorphism w that respects the declaration and satisfies $w|S = ws_0$, Theorem (29)(a) implies that it is the greatest one. This proves that $wep = v.ws_0$. Similarly, Theorem (29)(b) implies that $wlp = v.ws_1$. Therefore, the present definitions of *wep* and *wlp* coincide with the definitions (9).

We need the following lemma for the treatment of function *wev*.

Lemma. Let $p \in \mathbb{P}$. Let w be a homomorphism such that $w|S = wvs.p$. Let a simple configuration $(x,\ s;c)$ and a predicate r be given. Then $w.(s;c).r.x$ holds if and only if $p.x$ holds or every transition from $(x,\ s;c)$ goes to a configuration (y,c) where $w.c.r.y$ holds. (30)

Proof. We first compute

$$
\begin{aligned}
& w.(s;c).r.x \\
= \quad & \{w \text{ is a homomorphism and } s \in S\} \\
& wvs.p.s.(w.c.r).x \\
= \quad & \{\text{definition } wvs \text{ in (10)}\} \\
& (p \ \vee \ ws_0.s.(w.c.r)).x \\
= \quad & \{\text{calculus}\} \\
& p.x \ \vee \ ws_0.s.(w.c.r).x
\end{aligned}
$$

It remains to observe that $ws_0.s.q.x$ holds if and only if every transition from $(x,\ s;c)$ goes to a configuration (y,c) where $q.y$ holds. □

Theorem. Let $p \in \mathbb{P}$. Let w be a homomorphism that respects the declaration and satisfies $w|S = wvs.p$. Then $wev.p$ implies w. (31)

Proof. We give a proof by contradiction in order to avoid a case distinction. Let r be a predicate. It suffices to prove that the function h given by

$$h.(x,c) \quad \equiv \quad wev.p.c.r.x \ \wedge \ \neg w.c.r.x$$

is everywhere false. So, assume that $h.(x,c)$ holds. Since $wev.p$ and w both satisfy the conditions of the lemmas (27), (28), and (30), we have:

- if configuration (x, c) is rewritable there is a transition $(x, c) \rightarrow (x, d)$ such that $h.(x, d)$ holds.
- if configuration (x, c) is simple, then $\neg p.x$ holds and there is a transition $(x, c) \rightarrow (y, d)$ such that $h.(y, d)$ holds.

Since $h.(y, \varepsilon) = \mathit{false}$ by convention, this shows that there is an infinite $(\neg p)$–computation starting in (x, c). This implies $\neg \mathit{wev.p.c.r.x}$ and hence contradicts the assumption. □

Since $\mathit{wev.p}$ is itself a homomorphism w that respects the declaration and satisfies $w|S = \mathit{wvs.p}$, Theorem (31) shows that it is the least one. This proves that $\mathit{wev.p} = \mu.(\mathit{wvs.p})$. By specialization to the case $p = \mathit{false}$, we get $\mathit{wp} = \mu.\mathit{ws}_0$. Therefore, the present definitions of *wev* and *wp* coincide with the definitions in (10) and (9).

Lemma. Let w be a homomorphisme and let g be a w–accumulator that respects the declaration. If the configuration graph has a path from (x, c) to (y, d) then $g.c.r.x$ implies $g.d.r.y$. (32)

Proof. This is proven by induction in the length of the path. So it suffices to consider a single transition. If the transition is a rewriting, the assertion follows from Lemma (27)(b). In the case of a simple transition, it follows from the implication

$$[\, g.(s; d).r \;\Rightarrow\; \mathit{ws}_1.s.(g.d.r)\,]$$

which follows from (8). □

Theorem. Let g be a *kd*–accumulator that respects the declaration and satisfies $g|S = \mathit{wsi}$. Then g implies *winv*. (33)

Proof. We prove that $g.c.r.x$ implies $\mathit{winv.c.r.x}$ for every predicate r, every command c, and every state x. Assume $g.c.r.x$. Consider a computation that starts in (x, c), and let (y, d) be a configuration in this computation where r holds. We have to prove that r holds in all subsequent configurations. By induction it suffices to show that r holds in the next configuration. Since r remains valid in every rewriting, we may assume that (y, d) is a simple configuration, say $d = s; e$ with $s \in S$, $e \in \mathit{Cmd}$. Lemma (31) implies $g.d.r.y$. We observe

$$\begin{array}{ll} & r.y \;\wedge\; g.d.r.y \\ \Rightarrow & \quad \{d = s; e \text{ and } g \text{ is a } kd\text{–accumulator, (7) and (8)}\} \\ & r.y \;\wedge\; g.s.r.y \\ = & \quad \{g|S = \mathit{wsi} \text{ and definition of } \mathit{wsi} \text{ in (12)}\} \\ & r.y \;\wedge\; (\neg r \vee \mathit{ws}_0.s.r).y \\ \Rightarrow & \quad \{\text{calculus}\} \\ & \mathit{ws}_0.s.r.y\ . \end{array}$$

This implies that every transition from (y, d) goes to a configuration where r holds. In particular, r holds in the next configuration of the computation considered. □

Since *winv* is itself a *kd*–accumulator that respects the declaration and satisfies $g|S = \mathit{wsi}$, Theorem (33) shows that it is the greatest one. This proves that $\mathit{winv} = \tau.\mathit{kd.wsi}$. Therefore, the present definition of *winv* coincides with the one in (13).

Theorem. Let $p, q \in \mathbb{P}$. Let g be a *wev.q*–accumulator that respects the declaration and satisfies $g|S = wvs.(\neg p)$. Then g implies *wto.p.q*. (34)

Proof. We prove that $g.c.r.x$ implies $wto.p.q.c.r.x$ for every predicate r, every command c, and every state x. Assume $g.c.r.x$. Consider a computation that starts in (x, c), and let (y, d) be a simple configuration in this computation where p holds. Lemma (32) implies $g.d.r.y$. Since configuration (y, d) is simple, we can write $d = s; e$ with $s \in S$. We have

$$
\begin{array}{ll}
 & p.y \ \wedge \ g.d.r.y \\
\Rightarrow & \{d = s; e \text{ and } g \text{ is a } wev.q\text{–accumulator}\} \\
 & p.y \ \wedge \ g.s.(wev.q.e.r).y \\
= & \{g|S = wvs.(\neg p) \text{ and definition } wvs \text{ in } (11)\} \\
 & p.y \ \wedge \ (\neg p \ \vee \ ws_0.s.(wev.q.e.r)).y \\
\Rightarrow & \{\text{calculus}\} \\
 & ws_0.s.(wev.q.e.r).y
\end{array}
$$

Therefore, every transition from (y, d) goes to a configuration (z, e) such that $wev.q.e.r.z$ holds. This implies that the configuration (y, d) is followed after one or more transitions by a simple configuration where q holds or by termination in a configuration where r holds. This proves $wto.p.q.c.r.x$. □

Since *wto.p.q* is itself a *wev.q*–accumulator that respects the declaration and satisfies $g|S = wvs.(\neg p)$, Theorem (34) shows that it is the greatest one. This proves that

$$wto.p.q = \tau.(wev.q).(wvs(\neg p)).$$

Therefore, the present definition coincides with the one in (13).

Received September 1993
Accepted in revised form November 1994 by C. B. Jones