Predicative Semantics of Loops

Theodore S. Norvell
Faculty of Engineering
Memorial University of Newfoundland
St. John's NF
A1B 3X5
Canada
theo@engr.mun.ca
www.engr.mun.ca/∼theo/

Abstract

A predicative semantics is a mapping of programs to predicates. These predicates characterize sets of acceptable observations. The presence of time in the observations makes the obvious weakest fixed-point semantics of iterative constructs unacceptable. This paper proposes an alternative. We will see that this alternative semantics is monotone and implementable (feasible). Finally a programming theorem for iterative constructs is proposed, proved, and demonstrated. A novel aspect of this theorem is that it is *not* based on invariants.

Keywords

Predicative semantics, fixedpoint semantics, recursion, loops, refinement calculi.

0 FORMALIZATION

0.0 Specifications and refinement

Define xnat as the set of all natural numbers (nat) joined with an additional object ∞. We will suppose the following properties of ∞: it is larger than any natural number; $\infty + i = \infty - i = \infty$, for all natural numbers i; and $\infty - \infty = 0$.

I will use a 'batch' model for specifications borrowed, in most respects, from (Hehner 1993). Let Σ be any type (a nonempty set). I will call the members of Σ 'states', which is suggestive of imperative programming, but the actual contents of Σ will only be relevant in the examples. *Specifications* are functions of type

$$\Sigma \to \mathsf{xnat} \to \Sigma \to \mathsf{xnat} \to \mathsf{bool} \qquad .$$

For example, $(\lambda\sigma : \Sigma, \tau : \mathsf{xnat}, \sigma' : \Sigma, \tau' : \mathsf{xnat} \cdot \tau' \geq \tau \wedge \sigma' = \sigma)$ is a specification.

The variables $\sigma, \sigma' : \Sigma$ and $\tau, \tau' : \mathsf{xnat}$ will be used in writing specifications follows: for any expression E, I write $\langle E \rangle$ for the abstraction of E with respect to these variables. For example, $\langle \tau' \geq \tau \wedge \sigma' = \sigma \rangle$ is an abbreviation for the specification $(\lambda\sigma : \Sigma, \tau : \mathsf{xnat}, \sigma' : \Sigma, \tau' : \mathsf{xnat} \cdot \tau' \geq \tau \wedge \sigma' = \sigma)$. The variables are used as follows: σ and σ' represent the initial and final states while τ, and τ' represent the initial and final times. Thus the specification $\langle \tau' \geq \tau \wedge \sigma' = \sigma \rangle$ specifies that the final time is no less than the initial time and the final state is the same as the initial state.

All boolean operators ($\top$, $\bot$, $\wedge$, $\vee$, $\neg$, $\equiv$, $\not\equiv$, $\Rightarrow$, $\Leftarrow$)lift to the specification level; for example, if P and Q are specifications, $P \Rightarrow Q$ is the specification

$$\langle P.\sigma.\tau.\sigma'.\tau' \Rightarrow Q.\sigma.\tau.\sigma'.\tau' \rangle \qquad .$$

I use the symbols $\top$ and $\bot$ as the boolean constants true and false.

Refinement is defined as

$$(P \sqsubseteq Q) \equiv (\forall \sigma, \tau, \sigma', \tau' \cdot P.\sigma.\tau.\sigma'.\tau' \Leftarrow Q.\sigma.\tau.\sigma'.\tau') \qquad .$$

Equality of specifications is extensional

$$(P = Q) \equiv (P \sqsubseteq Q) \wedge (Q \sqsubseteq P) \qquad .$$

0.1 Discussion

This formalization applies equally to imperative programming and to functional programming. In imperative programming Σ is a set of states and in functional programming Σ is a set of values.

The use of predicates as specifications follows (Hehner 1984), (Hoare 1985), (Hehner 1993), (Hoare 1994), and (v. Karger and Hoare 1994).

The treatment of time is based on (Hehner 1993) and (Hehner 1994). It is necessary to include the infinity value in the time domain in order to deal with infinite loops. Because a statement may sequentially follow an infinite loop, xnat is used both for the initial and final states. Time is considered a quantity orthogonal to state; this simplifies the writing of specifications, but as we will see, it complicates the theory somewhat. Using the algebra of xnat requires some care, as some laws of nat do not hold unconditionally.

The use of predicates is reminiscent of the Z method (Spivey 1989), but the notion of refinement in Z is quite different and much harder to work with. More relevant are refinement calculi based on predicate transformers (Back and von Wright 1990) and (Morgan 1990). Binary predicates ordered by refinement are order isomorphic to the universally conjunctive predicate transformers (Holt 1991).

0.2 Bounds and classes of specifications

A specification P is said to be *progressive* if

$$\langle \tau' \geq \tau \rangle \sqsubseteq P \qquad ,$$

and to be *implementable* if

$$(\forall \sigma, \tau \cdot (\exists \sigma', \tau' \cdot P.\sigma.\tau.\sigma'.\tau')) \qquad .$$

The term *feasible* is also used in the literature —e.g. (Morgan 1990)— for this property of specifications.

A specification P is called a *condition* if it does not depend on its latter two arguments. An expression B is called a *condition*, if $\langle B \rangle$ is a condition.

A function g of type $\Sigma \rightarrow \mathsf{xnat} \rightarrow \mathsf{xnat}$ is called a *bound* of a specification R if

$$\langle \tau' \leq \tau + g.\sigma.\tau \rangle \sqsubseteq R \qquad .$$

A specification R is said to be *strongly bounded* if it has a natural bound:

$$(\exists g : \Sigma \rightarrow \mathsf{xnat} \rightarrow \mathsf{nat} \cdot \langle \tau' \leq \tau + g.\sigma.\tau \rangle \sqsubseteq R) \qquad .$$

Such a specification guarantees termination after a finite amount of time.

For a given specification R, we can consider the set of all bounds:

$$\{ g : \Sigma \rightarrow \mathsf{xnat} \rightarrow \mathsf{xnat} \mid \langle \tau' \leq \tau + g.\sigma.\tau \rangle \sqsubseteq R \} \qquad .$$

This set may be ordered pointwise: $(g \leq h) \equiv (\forall \sigma, \tau \cdot g.\sigma.\tau \leq h.\sigma.\tau)$. For an implementable and progressive R, this set will have a minimum member m_R characterized by

$$m_R.\sigma.\tau = (\max \sigma', \tau' \mid R.\sigma.\tau.\sigma'.\tau' \cdot \tau' - \tau) \qquad .$$

Henceforth the subscript on m will be omitted if it is the letter R. The key property of m is that a computation, starting in state σ at time τ, could take as long as $m.\sigma.\tau$:

$$(\forall \sigma, \tau \cdot (\exists \sigma', \tau' \cdot R.\sigma.\tau.\sigma'.\tau' \wedge \tau' = m.\sigma.\tau + \tau)) \qquad . \qquad (0)$$

If m does not depend on its second argument, we say that R is *time-insensitively bounded*.

Specifications that make no demands about the final state, when the initial time is ∞ are called *reasonable*. We want to allow reasonable specifications to be progressive. Define

$$Z = \langle \tau = \infty = \tau' \rangle$$

a specification R is said to be reasonable iff

$$R \; = Z \vee R \qquad .$$

0.3 Semantics

The semantics of programming constructs is given by defining the simple constructs as specifications and compound constructs as functions from (one or more) specifications to specifications. A program can then be defined as a specification built using only the programing constructs.

Some straight-line programming constructs are defined by:

$$skip = \langle \sigma' = \sigma \wedge \tau' = \tau \rangle$$

$$tick = \langle \sigma' = \sigma \wedge \tau' = \tau + 1 \rangle$$

$$P; Q = \langle \exists \hat{\sigma}, \hat{\tau} \cdot P.\sigma.\tau.\hat{\sigma}.\hat{\tau} \wedge Q.\hat{\sigma}.\hat{\tau}.\sigma'.\tau' \rangle$$

$$\mathbf{if}\, B\, \mathbf{then}\, P\, \mathbf{else}\, Q = (\langle B \rangle \wedge P) \vee (\neg \langle B \rangle \wedge Q) \qquad .$$

In the if-statement, it will be assumed that B is a condition. The *tick* construct is not really a programming construct, but is useful in defining the **while** loop.

It should be noted that these definitions hold for P and Q being any specifications, not only those formed from programming constructs.

For condition B and specification P define a function $w_{B,P}$ by

$$w_{B,P}.Q = \mathbf{if}\, B\, \mathbf{then} (P; tick; Q)\, \mathbf{else}\, skip \tag{1}$$

Let $W_{B,P}$ stand for **while** B **do** P. Henceforth the subscripts on w and W will be omitted where they are the letters B, P. As in (Norvell 1993) and (Norvell 1994) I define the while loop by three axioms

Progression: $\langle \tau' \geq \tau \rangle \sqsubseteq W$
Post-fixed-point: $w.W \sqsubseteq W$

Induction: $(\langle \tau' \geq \tau \rangle \sqsubseteq Q) \wedge (w.Q \sqsubseteq Q) \Rightarrow (W \sqsubseteq Q) \qquad$, for all Q.

This definition of the while-loop is called the *weakest progressive post-fixed-point* definition. The lattice of progressive specifications is complete and thus we may apply the Knaster-Tarski (Tarski 1955) theorem to tell us that W is well defined, that it is a fixed-point:

$$w.W = W \qquad , \tag{2}$$

and that it is the weakest of the fixed-points:

$$(\langle \tau' \geq \tau \rangle \sqsubseteq Q) \wedge (w.Q = Q) \Rightarrow (W \sqsubseteq Q) \qquad \text{, for all } Q.$$

As an alternative, but equivalent, definition, it is possible to replace the post-fixed-point and the induction axioms with these last two formula.

0.4 Discussion

It should be noted that the only action among the statements presented so far that is considered to take any time at all is the backward jump of the while loop. The "time" calculated for programs under this model is not proportional to the actual time that an implementation would take. However it does give the correct order for time complexities if all primitive operations are $O(1)$. This is not the only alternative, it is possible to use a semantics that keeps more careful track of time. Further discussion can be found in (Hehner 1994).

The importance of the progression axiom and the corresponding antecedent in the post-fixed-point axiom is that simply taking the weakest fixed-point of the equation

$$W = \textbf{if}\, B\, \textbf{then}(P; \textit{tick}; W)\, \textbf{else}\, \textit{skip}$$

would not result in a construct that is closed under progressiveness. For example, with the current definition we have

$$\langle \tau' = \infty \rangle \sqsubseteq \textbf{while}\, \top\, \textbf{do}\, \textit{skip} \qquad .$$

However, with the weakest fixed-point semantics, such a loop would not even be progressive. The difference between these approaches is manifested for (potentially) infinite loops, and one may wonder if these are worth the trouble. With the limited notion of observations used in this paper (initial and final states only), the point is arguable. However, for communicating programs, which can be modeled using a slightly richer notion of observations, (potentially) infinite loops are of great importance.

It might seem that it would be easier to simply work within the lattice of progressive specifications or even the semilattice of progressive and implementable specifications. From a semantic point of view, this may be true. But from the point of view of specifying, it is simplest if specifications are simply predicates with no restrictions. Also the progressive specifications are not closed under negation, so this would restrict the ways in which specifications are composed.

1 ESSENTIAL THEOREMS

Compound programming constructs are generally monotonic with respect to refinement and preserve implementability and progressiveness. Both these properties hold for the **while** loop as defined above.

We start by showing monotonicity.

Theorem 0 *For any condition B, and specifications P and Q, if $P \sqsubseteq Q$, then*

$$\mathbf{while}\, B\, \mathbf{do}\, P \sqsubseteq \mathbf{while}\, B\, \mathbf{do}\, Q \qquad .$$

Monotonicity is not hard to prove and illustrates all three axioms at work.

$$\begin{array}{ll}
 & W_{B,P} \sqsubseteq W_{B,Q} \\
\Leftarrow & \qquad \text{“ Induction axiom with } Q \text{ instantiated by } W_{B,Q}\text{. ”} \\
 & (\langle \tau' \geq \tau \rangle \sqsubseteq W_{B,Q}) \wedge (w_{B,P}.W_{B,Q} \sqsubseteq W_{B,Q}) \\
\equiv & \qquad \text{“ Progression axiom } (\tau' \geq \tau \sqsubseteq W_{B,Q}) \text{ and prop. calc. ”} \\
 & w_{B,P}.W_{B,Q} \sqsubseteq W_{B,Q} \\
\Leftarrow & \qquad \text{“ Post-fixpoint axiom } (w_{B,Q}.W_{B,Q} \sqsubseteq W_{B,Q}) \text{ and} \\
 & \qquad \text{transitivity of } \sqsubseteq \text{.”} \\
 & w_{B,P}.W_{B,Q} \sqsubseteq w_{B,Q}.W_{B,Q} \\
\equiv & \qquad \text{“ Definition of } w \text{ (1). ”} \\
 & \mathbf{if}\, B\, \mathbf{then}(P; \mathit{tick}; W_{B,Q}) \sqsubseteq \mathbf{if}\, B\, \mathbf{then}(Q; \mathit{tick}; W_{B,Q}) \\
\Leftarrow & \qquad \text{“ Monotonicity of } \mathbf{if} \text{ and ; . ”} \\
 & P \sqsubseteq Q
\end{array}$$

Next we show that progressiveness and implementability are jointly preserved.

Theorem 1 *For any condition B, and specification P, if*

P is progressive and
P is implementable,

then $\mathbf{while}\, B\, \mathbf{do}\, P$ *is progressive and implementable.*

The proof is given in appendix 0. This proof also shows that progressiveness is preserved on its own.

2 A PROGRAMMING THEOREM

Among the most useful of the theorems in a refinement calculus are those of the form "if ... then $R \sqsubseteq c.P_0.P_1.\cdots.P_{n-1}$" where c is a program constructor. For example, in the theory used in this paper $R \sqsubseteq \mathbf{if}\, B\, \mathbf{then}(\langle B\rangle \Rightarrow R)\, \mathbf{else}(\neg\langle B\rangle \Rightarrow R)$ is a useful theorem.

In order to derive programs involving while-loops, we would like to have theorems that conclude with $R \sqsubseteq \mathbf{while}\, B\, \mathbf{do}\, P$. In most refinement calculi, such theorems are usually based on invariants. In (Hehner 1979) the idea of "recursive refinement" instead of invariants is suggested. In this section we propose a theorem based on recursive refinement.

A specification R is said to be *recursively refined* if

$$R \sqsubseteq \mathbf{if}\, B\, \mathbf{then}(P; \mathit{tick}; R)\, \mathbf{else}\, \mathit{skip} \qquad . \tag{3}$$

It is often true that when (3) is true, it is also true that

$$R \sqsubseteq \mathbf{while}\, B\, \mathbf{do}\, P \qquad . \tag{4}$$

So the question arises of under what conditions (3) implies (4). The next theorem presents one possible set of conditions.

Theorem 2 *For any condition B, and specifications P and R, if*

P is progressive;
R is implementable, strongly bounded, time insensitively bounded, and reasonable; and
R is recursively refined:

$$R \sqsubseteq \mathbf{if}\, B\, \mathbf{then}(P; \mathit{tick}; R)\, \mathbf{else}\, \mathit{skip} \tag{5}$$

then $R \sqsubseteq \mathbf{while}\, B\, \mathbf{do}\, P$.

The proof of this is by induction and is given in appendix 1.

2.0 Discussion

It is informative to look at why additional conditions are required beyond (5) are required.

Here is a simple counter example showing that (5) does not imply $R \sqsubseteq \mathbf{while}\, B\, \mathbf{do}\, P$. Take P to be *skip* and B to be $\top$. We have

$$\bot \sqsubseteq \mathbf{if}\ \top\ \mathbf{then}(\mathit{skip}; \mathit{tick}; \bot)\ \mathbf{else}\ \mathit{skip} \qquad .$$

But it is certainly not the case that

$$\bot \sqsubseteq \mathbf{while}\ \top\ \mathbf{do}\ \mathit{skip}$$

as this would mean that the while loop is unimplementable. Yet we know from the Theorem 1 and the fact that *skip* is progressive and implementable that the while loop is also implementable.

In (Hehner 1993) it is suggested that recursive refinement is only a valid programming method for implementable specifications. What if we restrict R to be an implementable specification? Again we are disappointed. Consider the following ‘monster’. It is true that

$$\langle \sigma' = 0 \rangle \sqsubseteq \mathbf{if}\ \top\ \mathbf{then}(\mathit{skip}; \mathit{tick}; \langle \sigma' = 0 \rangle)\ \mathbf{else}\ \mathit{skip}$$

and that

$$\langle \sigma' = 1 \rangle \sqsubseteq \mathbf{if}\ \top\ \mathbf{then}(\mathit{skip}; \mathit{tick}; \langle \sigma' = 1 \rangle)\ \mathbf{else}\ \mathit{skip} \qquad .$$

Yet, if it were true both that

$$\langle \sigma' = 0 \rangle \sqsubseteq \mathbf{while}\ \top\ \mathbf{do}\ \mathit{skip}$$

and

$$\langle \sigma' = 1 \rangle \sqsubseteq \mathbf{while}\ \top\ \mathbf{do}\ \mathit{skip} \qquad ,$$

we would have to conclude that

$$\bot \sqsubseteq \langle \sigma' = 0 \wedge \sigma' = 1 \rangle \sqsubseteq \mathbf{while}\ \top\ \mathbf{do}\ \mathit{skip} \qquad .$$

Again this contradicts the implementability of **while** loops.

The problem illustrated in this example occurs basically because the loop is infinite. By adding the requirement of strong bounding, infinite loops are eliminated. The requirements that P be progressive and that the minimum bound is not sensitive to the starting time ensure that the induction goes through.

The requirement that R be reasonable is the most troubling, as it requires one to write specifications in a way that one otherwise would not. This requirement is necessary because the induction does not work when the initial time is ∞. One solution to this problem is to use the ordinal numbers as the time domain rather than xnat. The problem with this 'solution' is that the proof of Theorem 1 no longer goes through. Other solutions might involve changing the definitions of refinement or of the programming constructs, or removing the orthogonality of time and state (i.e. treat specifications as binary relations on $(\Sigma \times \mathsf{nat}) \cup \{\infty\}$). Such changes are rather undesirable as the simplicity of these definitions is an important attribute of the predicative programming approach.

A somewhat similar theorem appears in (Sekerinski 1993). However the definition of the programming connectives (e.g. sequential composition) is different and the theorem can only be used to show that a specification is a fixed-point. From a programming point of view, this is less satisfactory as one is not so much interested in whether a while loop equals a given specification, but rather whether it implements the specification at hand.

3 A PROGRAMMING EXAMPLE

In imperative programming, Σ consists of states, which may be considered functions from program variable names to values. This function can be extended to expressions. For any program variable name x, and expression E the assignment statement can be defined as

$$x := E \quad = \quad \langle \sigma'.x = \sigma.E \wedge (\forall y \in \mathrm{dom}\,.\sigma \mid x \neq y \cdot \sigma'.y = \sigma.y) \wedge \tau' = \tau \rangle$$

where $\mathrm{dom}\,.\sigma$ is the set of variable names.

In this section, for any program variable name such as x, I will use the convention of writing x in specifications rather than $\sigma.x$ and of writing x' rather than $\sigma'.x$. With this convention the assignment is simply

$$x := E \quad = \quad \langle x' = E \wedge y' = y \wedge \cdots \wedge \tau' = \tau \rangle \qquad .$$

where the $\cdots$ depends on what variables are in the domain of the states. We have the following useful substitution law

$$x := E; \langle P \rangle \quad = \quad \langle P^x_E \rangle \qquad , \tag{6}$$

provided P is written with the convention. There is an analogous law for the time variable

$$tick; \langle P \rangle \quad = \quad \left\langle P^{\tau}_{\tau+1} \right\rangle \qquad . \tag{7}$$

I will solve a trivial programming example in two ways, illustrating the relationship of recursive refinement to the invariant method. The problem is that of finding the product of the elements in an array A of size N. The specification is

$$S = \langle s' = (\Pi i \in \{0, ..N\} \cdot A.i) \wedge \tau' \leq \tau + N \rangle \vee Z$$

I will write $\{i, ..k\}$ for the set of all integers j, such that $i \leq j < k$ and $\{i, .., k\}$ for the set of all integers j, such that $i \leq j \leq k$. The "$\vee Z$" part of the specification is required to satisfy the condition of 'reasonableness'. The predicate Z was defined in Section 0.2.

3.0 A solution that does not use an invariant

We can refine S using the substitution law (supposing x is of type $\{0, .., N\}$):

$$S \sqsubseteq s, x := 1, 0; R0 \qquad ,$$

where

$$R0 = \langle s' = s \times (\Pi i \in \{x, ..N\} \cdot A.i) \wedge \tau' \leq \tau + N - x \rangle \vee Z \qquad .$$

This is refined by cases

$$R0 \sqsubseteq \textbf{if } x < N \textbf{ then} (\langle x < N \rangle \Rightarrow R0) \textbf{ else } skip \qquad .$$

Now we can refine the remaining specification:

$$\begin{array}{ll} & \langle x < N \rangle \Rightarrow R0 \\ \sqsubseteq & \qquad \text{“ Splitting the product. ”} \\ & \left\langle \begin{array}{ll} & s' = s \times A.x \times (\Pi i \in \{x+1, ..N\} \cdot A.i) \\ \wedge & \tau' \leq \tau + 1 + N - (x+1) \end{array} \right\rangle \vee Z \\ = & \qquad \text{“ Substitution (6) and (7). ”} \\ & s, x := s \times A.x, x+1; tick; R0 \qquad . \end{array}$$

Putting these results together (by the monotonicity of **if**) we get

$$R0 \sqsubseteq \mathbf{if}\, x < N\, \mathbf{then} (s, x := s \times A.x, x + 1; \mathit{tick}; R0)\, \mathbf{else}\, \mathit{skip} \qquad .$$

Since all the conditions of Theorem 2 are met, we may conclude

$$R0 \sqsubseteq \mathbf{while}\, x < N\, \mathbf{do}\, s, x := s \times A.x, x + 1 \qquad .$$

3.1 A solution using an invariant

It should be noted that nowhere in the above development, nor even in the thinking behind it, did the formula

$$s = (\Pi i \in \{0, ..x\} \cdot A.i)$$

appear. This is the invariant that would be used if the invariant method were used. Recursive refinement does not exclude the use of invariants and, in the development of many loops, it is the simplest method.

Using substitution, and some simplification, we could also refine S by

$$S \sqsubseteq s, x := 1, 0; R1$$

where $R1$ is

$$\langle s = (\Pi i \in \{0, ..x\} \cdot A.i) \Rightarrow s' = (\Pi i \in \{0, ..N\} \cdot A.i) \wedge \tau' \leq \tau + N - x \rangle \vee Z.$$

As with $R0$ we can derive that

$$R1 \sqsubseteq \mathbf{if}\, x < N\, \mathbf{then} (s, x := s \times A.x, x + 1; \mathit{tick}; R1)\, \mathbf{else}\, \mathit{skip} \qquad ,$$

although the reasoning is a little more involved. Again the Theorem 2 can be applied.

We can recognize that the $R1$ has the form of a precondition $s = (\Pi i \in \{0, ..x\} \cdot A.i)$ and a postrelation $s' = (\Pi i \in \{0, ..N\} \cdot A.i) \wedge \tau' \leq \tau + N - x$, and that the precondition is a loop invariant, but there is no real need to think in these terms.

3.2 Views and refinement by parts

It may seem unpleasant to carry the specification of the time bound around while deriving a recursive refinement. However, it is not necessary to consider all parts of a specification while deriving a refinement for it. If f is a monotone function of specifications, then

$$(T \wedge U \sqsubseteq f.(V \wedge W)) \Leftarrow (T \sqsubseteq f.V) \wedge (U \sqsubseteq f.W) \qquad .$$

In particular it may be useful to *derive* a recursive refinement for only the part of the specification that does not deal with time and then *check* that the same recursive refinement applies to the remainder of the specification.

For example $R0$ can be split into two 'views': $R0 = T \wedge U$

$$\begin{array}{lcl} T & = & \langle s' = s \times (\Pi i \in \{x, ..N\} \cdot A.i\rangle \vee Z \\ U & = & \langle \tau' \leq \tau + N - x\rangle \vee Z \end{array}$$

We can then derive that

$$T \sqsubseteq \textbf{if}\, x < N \,\textbf{then}(s, x := A.x, x+1; \mathit{tick}; T)\, \textbf{else}\, \mathit{skip}$$

and check that

$$U \sqsubseteq \textbf{if}\, x < N \,\textbf{then}(s, x := A.x, x+1; \mathit{tick}; U)\, \textbf{else}\, \mathit{skip} \qquad .$$

This gives us that

$$R0 \sqsubseteq \textbf{if}\, x < N \,\textbf{then}(s, x := A.x, x+1; \mathit{tick}; R0)\, \textbf{else}\, \mathit{skip} \qquad .$$

4 CALCULATING THE LOOPS.

Weakest prespecification (Hoare and He 1987) is defined as

$$S \swarrow U = \langle \forall \hat{\sigma}, \hat{\tau} \cdot U.\sigma'.\tau'.\hat{\sigma}.\hat{\tau} \Rightarrow S.\sigma.\tau.\hat{\sigma}.\hat{\tau}\rangle \qquad . \tag{8}$$

The key property of the weakest prespecification is this Galois connection:

$$(S \swarrow U \sqsubseteq T) \equiv (S \sqsubseteq T; U) \qquad . \tag{9}$$

Suppose one has an R that is implementable, strongly bounded, time insensitively bounded, and reasonable. The remaining condition on R, that of recursive refinement, is equivalent to the conjunction

$$\begin{array}{ll} & (R \sqsubseteq \langle \neg B\rangle \wedge \mathit{skip}) \qquad (10) \\ \wedge & ((\langle B\rangle \Rightarrow R) \swarrow (\mathit{tick}; R) \sqsubseteq P) \qquad (11) \end{array}$$

So one can find a suitable loop implementation for R by first finding a B such that (10) (of course, the stronger this B is, the better) and then using as a loop body $P = \langle \tau' \geq \tau \rangle \wedge (\langle B \rangle \Rightarrow R) \swarrow (\mathit{tick}; R)$, which is the weakest, progressive specification satisfying (11). Any refinement method can then be attempted to refine P by a program.

The first step, that of searching for a suitable B, can also be made more calculational. One can start with $\neg R.\sigma.\tau.\sigma.\tau$, which is the strongest solution of (10), and then weaken until a condition is found that is easily implemented.

5 CONCLUSION AND FUTURE WORK.

I have presented a semantics for iteration within a particular version of the refinement calculus. This semantics has been shown to enjoy the properties one would expect and also to give interesting results for infinite loops that suggest applications for communicating processes. From the semantics, I have proved a theorem that allows it to be used in the derivation of programs.

The proofs are done in a calculational and point-free style.

Although the notation used is suggestive of imperative programming, the results are equally applicable to functional programming — program variables are only introduced in the examples.

As mentioned above, one of the prime motivations for the weakest progressive post-fixed-point definition of while loops is to accommodate communicating processes. In this case observations would consist not only of an initial and final state, but also a history of communications. Thus generalizing these results to this more general setting is important.

Once processes may interact, potentially infinite loops become of greater interest and the restriction to strong bounding is too severe. The search for such programming theorems will be the subject of future research.

The work so far has concentrated on while loops, but the results should be extendible to any set of mutually recursive subroutines.

The relationship to (v. Karger and Hoare 1994) is intriguing. In that paper a very abstract calculus of specifications is presented. The principal difference between their calculus and relational calculus is the replacement of the converse operator by a relative converse operator. This prevents the formation of specifications that "undo". This is the same motivation for restricting our attention to the lattice of progressive specifications and forming fixed-points within that lattice.

6 ACKNOWLEDGMENTS

Thanks are due to Eric Hehner for many discussions and comments, to the referees for helpful comments, to NSERC for funding, and to the Faculty of

Engineering, Memorial University of Newfoundland, for equipment, location, and funding.

APPENDIX 0 PROOF OF THE IMPLEMENTABILITY THEOREM 1

(This appendix represents joint work with Eric Hehner.)

Given an implementable and progressive P we must show that

$$W = \textbf{while}\, B\, \textbf{do}\, P$$

too is implementable and progressive.

Throughout this appendix specification P will be assumed to be implementable and progressive. Rather than work with P, I will work with $P0 = P; \mathit{tick}$. $P0$ is also implementable and progressive.

By the progression axiom, we know that $W = \textbf{while}\, B\, \textbf{do}\, P$ is progressive. The question remains whether it is implementable.

Suppose we can find an implementable Q that is also progressive and such that

$$\textbf{if}\, B\, \textbf{then}(P0; Q)\, \textbf{else}\, \mathit{skip} \sqsubseteq Q \qquad .$$

The induction axiom tells us that $W \sqsubseteq Q$. Since any specification that is refined by an implementable specification must itself be implementable, this would imply that W is implementable. Thus the goal becomes: find a Q that is implementable, progressive, and a post-fixed-point.

There must be at least one progressive fixed-point. We know this because W is one example. In the following let S be any progressive fixed-point:

$$\tau' \geq \tau \quad \sqsubseteq \quad S \tag{12}$$

$$S \quad = \quad \textbf{if}\, B\, \textbf{then}(P0; S)\, \textbf{else}\, \mathit{skip} \qquad . \tag{13}$$

S may or may not be implementable. We define a condition D to identify the initial states for which S accepts no final state:

$$D \equiv \neg(\exists \sigma', \tau' \cdot S.\sigma.\tau.\sigma'.\tau') \qquad . \tag{14}$$

Now define Q as

$$Q = S \vee (\langle D \wedge \tau' = \infty \rangle) \qquad . \tag{15}$$

It is clear that Q is implementable and progressive. We need only prove that it is a post-fixed-point.

To do this I will use two lemmata to be proved later:

$$\begin{aligned} \langle B \rangle &\sqsubseteq \langle D \rangle & (16) \\ \langle D \rangle \wedge P0; X &= \langle D \rangle \wedge P0; (\langle D \rangle \wedge X) \qquad , & (17) \end{aligned}$$

for any X. Both follow from the fact that S is a fixed-point and will be proved later.

We prove that Q is a post-fixed-point by considering separately D true and D false. First for D true:

$$\begin{array}{ll} & \langle D \rangle \wedge (\mathbf{if}\, B\, \mathbf{then} (P0; Q)\, \mathbf{else}\, \mathit{skip}) \\ = & \qquad \text{“ (16) } \langle B \rangle \sqsubseteq \langle D \rangle \text{ . ”} \\ & \langle D \rangle \wedge (P0; Q) \\ = & \qquad \text{“ Defn of } Q \text{. ”} \\ & \langle D \rangle \wedge (P0; (S \vee \langle D \wedge \tau' = \infty \rangle)) \\ = & \qquad \text{“ Distribute ; over } \vee \text{ . ”} \\ & \langle D \rangle \wedge ((P0; S) \vee (P0; \langle D \wedge \tau' = \infty \rangle)) \\ = & \qquad \text{“ Lemma (16) } \langle B \rangle \sqsubseteq \langle D \rangle \text{ ”} \\ & \langle D \rangle \wedge ((\mathbf{if}\, B\, \mathbf{then} (P0; S)\, \mathbf{else}\, \mathit{skip}) \vee (P0; \langle D \wedge \tau' = \infty \rangle)) \\ = & \qquad \text{“ } S \text{ is a fixed-point. ”} \\ & \langle D \rangle \wedge (S \vee (P0; \langle D \wedge \tau' = \infty \rangle)) \\ = & \qquad \text{“ Lemma (17). ”} \\ & \langle D \rangle \wedge (S \vee (P0; \langle \tau' = \infty \rangle)) \\ \sqsubseteq & \qquad \text{“ } P0 \text{ is implementable. ”} \\ & \langle D \rangle \wedge (S \vee \langle \tau' = \infty \rangle) \\ = & \qquad \text{“ Propositional calculus. ”} \\ & \langle D \rangle \wedge (S \vee \langle D \wedge \tau' = \infty \rangle) \\ = & \qquad \text{“ Defn of } Q \text{. ”} \\ & \langle D \rangle \wedge Q \end{array}$$

Now for D false:

$$\begin{array}{ll} & \neg \langle D \rangle \wedge (\mathbf{if}\, B\, \mathbf{then} (P0; Q)\, \mathbf{else}\, \mathit{skip}) \\ = & \qquad \text{“ Defn of } Q \text{. ”} \\ & \neg \langle D \rangle \wedge (\mathbf{if}\, B\, \mathbf{then} (P0; (S \vee \langle D \wedge \tau' = \infty \rangle))\, \mathbf{else}\, \mathit{skip}) \\ \sqsubseteq & \qquad \text{“ Monotonicity. ”} \\ & \neg \langle D \rangle \wedge (\mathbf{if}\, B\, \mathbf{then} (P0; S)\, \mathbf{else}\, \mathit{skip}) \\ = & \qquad \text{“ } S \text{ is a fixed-point. ”} \end{array}$$

$$\begin{array}{ll} & \neg\,\langle D\rangle \wedge S \\ = & \quad \text{“ Propositional calculus .”} \\ & \neg\,\langle D\rangle \wedge (S \vee \langle D \wedge \tau' = \infty\rangle) \\ = & \quad \text{“ Defn of } Q\text{. ”} \\ & \neg\,\langle D\rangle \wedge Q \end{array}$$

From

$$\langle D\rangle \wedge (\mathbf{if}\, B\, \mathbf{then}(P0; Q)\, \mathbf{else}\, \mathit{skip} \sqsubseteq \langle D\rangle \wedge Q$$

and

$$\neg\,\langle D\rangle \wedge (\mathbf{if}\, B\, \mathbf{then}(P0; Q)\, \mathbf{else}\, \mathit{skip} \sqsubseteq \neg\,\langle D\rangle \wedge Q$$

we can conclude

$$\mathbf{if}\, B\, \mathbf{then}(P0; Q)\, \mathbf{else}\, \mathit{skip} \sqsubseteq Q \qquad .$$

It remains to prove the two lemmata. We start with (16)

$$\begin{array}{ll} & \langle B\rangle \sqsubseteq \langle D\rangle \\ = & \quad \text{“ Contrapositive. ”} \\ & \neg\,\langle D\rangle \sqsubseteq \neg\,\langle B\rangle \\ = & \quad \text{“ Definition of } D\text{. ”} \\ & \langle \exists \sigma', \tau' \cdot S.\sigma.\tau.\sigma'.\tau'\rangle \sqsubseteq \neg\,\langle B\rangle \\ = & \quad \text{“ } S \text{ is a fixed-point. ”} \\ & \langle \exists \sigma', \tau' \cdot (\mathbf{if}\, B\, \mathbf{then}(P0; S)\, \mathbf{else}\, \mathit{skip}).\sigma.\tau.\sigma'.\tau'\rangle \sqsubseteq \neg\,\langle B\rangle \\ = & \quad \text{“ Definition of } \mathbf{if}\text{ . ”} \\ & \langle \exists \sigma', \tau' \cdot \mathit{skip}.\sigma.\tau.\sigma'.\tau'\rangle \sqsubseteq \neg\,\langle B\rangle \\ = & \quad \text{“ } \mathit{skip} \text{ is implementable. ”} \\ & \top \end{array}$$

It now remains only to prove (17). This follows easily from

$$\langle D'\rangle \sqsubseteq P0 \wedge \langle D\rangle$$

(D' is D with σ and τ replaced by σ' and τ') which is proved by contradiction. Start with the negation:

$$\exists \sigma, \tau, \sigma', \tau' \cdot D \wedge P0.\sigma.\tau.\sigma'.\tau' \wedge \neg D'$$

$\equiv$ “ Rename variables. ”

$\exists \sigma, \tau, \sigma'', \tau'' \cdot D \wedge P0.\sigma.\tau.\sigma''.\tau'' \wedge \neg D''$

$\equiv$ “ Defn D. ”

$\exists \sigma, \tau, \sigma'', \tau'' \cdot D \wedge P0.\sigma.\tau.\sigma''.\tau'' \wedge (\exists \sigma', \tau' \cdot S.\sigma''.\tau''.\sigma'.\tau')$

$\equiv$ “ Predicate calculus: $\exists$ over $\wedge$. ”

$\exists \sigma, \tau, \sigma', \tau', \sigma'', \tau'' \cdot D \wedge P0.\sigma.\tau.\sigma''.\tau'' \wedge S.\sigma''.\tau''.\sigma'.\tau'$

$\equiv$ “ Predicate calculus: $\exists$ over $\wedge$. ”

$\exists \sigma, \tau, \sigma', \tau' \cdot D \wedge (\exists \sigma'', \tau'' \cdot P0.\sigma.\tau.\sigma''.\tau'' \wedge S.\sigma''.\tau''.\sigma'.\tau')$

$\equiv$ “ Definition of ; . ”

$\exists \sigma, \tau, \sigma', \tau' \cdot D \wedge (P0; S).\sigma.\tau.\sigma'.\tau'$

$\equiv$ “ Lemma (16). ”

$\exists \sigma, \tau, \sigma', \tau' \cdot D \wedge (\mathbf{if}\, B\, \mathbf{then}(P0; S)\, \mathbf{else}\, \mathit{skip}).\sigma.\tau.\sigma'.\tau'$

$\equiv$ “ S is a fixed-point. ”

$\exists \sigma, \tau, \sigma', \tau' \cdot D \wedge S.\sigma.\tau.\sigma'.\tau'$

$\equiv$ “ Defn D. ”

$\exists \sigma, \tau, \sigma', \tau' \cdot \neg(\exists \sigma', \tau' \cdot S.\sigma.\tau.\sigma'.\tau') \wedge S.\sigma.\tau.\sigma'.\tau'$

$\equiv$ “ Predicate calculus: $\exists$ over $\wedge$. ”

$\exists \sigma, \tau \cdot \neg(\exists \sigma', \tau' \cdot S.\sigma.\tau.\sigma'.\tau') \wedge (\exists \sigma', \tau' \cdot S.\sigma.\tau.\sigma'.\tau')$

$\equiv$ “ Contradiction. ”

$\perp$

APPENDIX 1 PROOF OF THE WHILE-LOOP THEOREM 2

We will assume

P is progressive;
R is implementable, strongly bounded, time insensitively bounded, and reasonable; and
R is recursively refined:

$$R \sqsubseteq \mathbf{if}\, B\, \mathbf{then}(P; \mathit{tick}; R)\, \mathbf{else}\, \mathit{skip} \qquad .$$

It must be shown that $R \sqsubseteq \mathbf{while}\, B\, \mathbf{do}\, P$.

Throughout this appendix condition B and specification P will be assumed to have the properties stated in Theorem 2.

APPENDIX 1.0 $\langle \tau = \infty \rangle$ or $\langle \tau \in \mathsf{nat} \rangle$

Ultimately I want to prove $R \sqsubseteq W$. I will prove this by cases on $\langle \tau = \infty \rangle$; i.e. by proving $R \sqsubseteq \langle \tau = \infty \rangle \wedge W$ and $R \sqsubseteq \langle \tau \in \mathsf{nat} \rangle \wedge W$. I start with the first

$$\begin{array}{ll}
 & \langle \tau = \infty \rangle \wedge W \\
\sqsupseteq & \qquad \text{“ Progression axiom ”} \\
 & \langle \tau = \infty \rangle \wedge \langle \tau' \geq \tau \rangle \\
= & \qquad \text{“ } \mathsf{xnat} \text{ arithmetic”} \\
 & \langle \tau = \infty = \tau' \rangle \\
\sqsupseteq & \qquad \text{“ Definition of } Z \text{ and generalization”} \\
 & Z \vee R \\
= & \qquad \text{“ } R \text{ is reasonable ”} \\
 & R
\end{array}$$

APPENDIX 1.1 B or $\neg B$

It remains to prove $R \sqsubseteq \langle \tau \in \mathsf{nat} \rangle \wedge W$. I will prove this by cases on $\langle B \rangle$. The easy case is when B is initially false. We do not need to use the fact that τ is finite.

$$\begin{array}{ll}
 & \neg \langle B \rangle \wedge W \qquad\qquad\qquad\qquad (18) \\
= & \qquad \text{“ } W \text{ is a fixpoint of } w \text{ (2). ”} \\
 & \neg \langle B \rangle \wedge w.W \\
= & \qquad \text{“ Definition of } w \text{ (1). ”} \\
 & \neg \langle B \rangle \wedge \textbf{if } B \textbf{ then} (P; \mathit{tick}; W) \textbf{ else } \mathit{skip} \\
= & \qquad \text{“ Definition of } \textbf{if.} \text{ ”} \\
 & \neg \langle B \rangle \wedge \mathit{skip} \\
= & \qquad \text{“ Definition of } \textbf{if.} \text{ ”} \\
 & \neg \langle B \rangle \wedge \textbf{if } B \textbf{ then} (P; \mathit{tick}; R) \textbf{ else } \mathit{skip} \\
= & \qquad \text{“ Definition of } w \text{ (1). ”} \\
 & \neg \langle B \rangle \wedge w.R \\
\sqsupseteq & \qquad \text{“ } R \text{ is recusively refined (5) and monotonicity of } \wedge \text{ . ”} \\
 & \neg \langle B \rangle \wedge R \\
\sqsupseteq & \qquad \text{“ Specialization. ”} \\
 & R \quad .
\end{array}$$

On the other hand what about the case when B is initially true?

$$
\begin{array}{ll}
 & \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge W \qquad (19)\\
= & \quad \text{“ } W \text{ is a fixpoint of } w \text{ (2). ”}\\
 & \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge w.W\\
= & \quad \text{“ Definition of } w \text{ (1). ”}\\
 & \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge \mathbf{if}\, B\, \mathbf{then}(P; \mathit{tick}; W)\, \mathbf{else}\, \mathit{skip}\\
= & \quad \text{“ Definition of } \mathbf{if.} \text{ ”}\\
 & \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge (P; \mathit{tick}; W)\\
= & \quad \text{“ } B \wedge \langle \tau \in \mathsf{nat}\rangle \text{ is a condition. ”}\\
 & (\langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge P); \mathit{tick}; W \qquad .
\end{array}
$$

At this point we are a little stuck. It is time to bring out the heavy artillery...

APPENDIX 1.2 The repetend decreases the bound more than it spends time.

Here and below, I will write M for the expression $m.\sigma.\tau$, and M' for $m.\sigma'.\tau'$. As R is strongly bounded, M is a natural number (i.e. not ∞); as R is time-insensitively bounded, it does not depend on τ.

We now use our knowledge of R to conclude that P decreases the bound more than it spends time.

$$
\begin{array}{ll}
 & \quad \text{“ Recursive refinement (5) ”}\\
 & R \sqsubseteq w.R\\
= & \quad \text{“ Definition of } w \text{ (1). ”}\\
 & R \sqsubseteq \mathbf{if}\, B\, \mathbf{then}(P; \mathit{tick}; R)\, \mathbf{else}\, \mathit{skip}\\
\Rightarrow & \quad \text{“ Definition of } \mathbf{if.} \text{ ”}\\
 & R \sqsubseteq \langle B\rangle \wedge (P; \mathit{tick}; R)\\
\equiv & \quad \text{“ } B \text{ is a condition. ”}\\
 & R \sqsubseteq (\langle B\rangle \wedge P); \mathit{tick}; R\\
\Rightarrow & \quad \text{“ As } M \text{ is a bound, } \langle \tau' \le \tau + M\rangle \sqsubseteq R\text{; transitivity of } \sqsubseteq \text{. ”}\\
 & \langle \tau' \le \tau + M\rangle \sqsubseteq (\langle B\rangle \wedge P); \mathit{tick}; R\\
\Rightarrow & \quad \text{“ } R \text{ could take as much time as } M \text{ . ”} \qquad (20)\\
 & \langle \tau' + M' \le \tau + M\rangle \sqsubseteq (\langle B\rangle \wedge P); \mathit{tick}\\
\equiv & \quad \text{“ } M \text{ does not depend on } \tau \text{ ”}\\
 & \langle \tau' + M' + 1 \le \tau + M\rangle \sqsubseteq \langle B\rangle \wedge P \qquad . \qquad (21)
\end{array}
$$

The hint at step (20) is an appeal to computational intuition, but can be fleshed out. To do so, we use the weakest prespecification (8).

It will be helpful to restate (0) using different variable names:

$$(\forall \sigma', \tau' \cdot (\exists \dot{\sigma}, \dot{\tau} \cdot R.\sigma'.\tau'.\dot{\sigma}.\dot{\tau} \wedge \dot{\tau} = \tau' + M')) \qquad . \tag{22}$$

We can now calculate:

$$\begin{array}{ll}
 & \langle \tau' \leq \tau + M \rangle \swarrow R \qquad\qquad (23) \\
= & \quad \text{“ Definition of weakest prespecification. ”} \\
 & \langle \forall \hat{\sigma}, \hat{\tau} \cdot R.\sigma'.\tau'.\hat{\sigma}.\hat{\tau} \Rightarrow \hat{\tau} \leq \tau + M \rangle \\
\sqsupseteq & \quad \text{“ Specialize to any } \dot{\sigma} \text{ and } \dot{\tau} \text{ that (22) says exist.} \\
 & \quad \text{(These variables are dependant on } \sigma' \text{ and } \tau' \text{.) ”} \\
 & \langle R.\sigma'.\tau'.\dot{\sigma}.\dot{\tau} \Rightarrow \dot{\tau} \leq \tau + M \rangle \\
= & \quad \text{“ From (22) we know } R.\sigma'.\tau'.\dot{\sigma}.\dot{\tau}. \text{ ”} \\
 & \langle \dot{\tau} \leq \tau + M \rangle \\
= & \quad \text{“ From (22) we know } \dot{\tau} = \tau' + M'. \text{ ”} \\
 & \langle \tau' + M' \leq \tau + M \rangle \qquad .
\end{array}$$

Now we can flesh out step (20):

$$\begin{array}{ll}
 & \langle \tau' \leq \tau + M \rangle \sqsubseteq (\langle B \rangle \wedge P); \mathit{tick}; R \\
\equiv & \quad \text{“ Galois connection (9). ”} \\
 & \langle \tau' \leq \tau + M \rangle \swarrow R \sqsubseteq (\langle B \rangle \wedge P); \mathit{tick} \\
\Rightarrow & \quad \text{“ Calculation (23) and transitivity. ”} \\
 & \langle \tau' + M' \leq \tau + M \rangle \sqsubseteq (\langle B \rangle \wedge P); \mathit{tick} \qquad .
\end{array}$$

This completes the proof of (21). As a corollary we have:

$$\begin{array}{ll}
 & \quad \text{“ } P \text{ decreases the bound more than it consumes time (21). ”} \\
 & \langle \tau' + M' + 1 \leq \tau + M \rangle \sqsubseteq \langle B \rangle \wedge P \\
= & \quad \text{“ } P \text{ is progressive. ”} \\
 & \langle \tau' + M' + 1 \leq \tau + M \wedge \tau \leq \tau' \rangle \sqsubseteq \langle B \rangle \wedge P \\
\Rightarrow & \quad \text{“ Monotonicity of } \wedge \text{ ”} \\
 & \langle \tau' + M' + 1 \leq \tau + M \wedge \tau \leq \tau' \wedge \tau \in \mathsf{nat} \rangle \sqsubseteq \langle B \rangle \wedge \langle \tau \in \mathsf{nat} \rangle \wedge P \\
= & \quad \text{“ Both } M \text{ and } M' \text{ are naturals ”} \\
 & \langle \tau' + M' + 1 \leq \tau + M \wedge \tau \leq \tau' \wedge \tau, \tau' \in \mathsf{nat} \rangle \sqsubseteq \langle B \rangle \wedge \langle \tau \in \mathsf{nat} \rangle \wedge P \\
\Rightarrow & \quad \text{“ Transitivity of } \leq . \text{ ”}
\end{array}$$

$$\begin{array}{ll}
 & \langle \tau' + M' + 1 \le \tau' + M \wedge \tau \le \tau' \wedge \tau, \tau' \in \mathsf{nat}\rangle \sqsubseteq \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge P \\
\Rightarrow & \quad \text{“ } \mathsf{nat} \text{ arithmetic. ”} \\
 & \langle M' + 1 \le M \wedge \tau \le \tau' \wedge \tau, \tau' \in \mathsf{nat}\rangle \sqsubseteq \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge P \\
\Rightarrow & \quad \text{“ Weakening the LHS of the refinement. ”} \\
 & \langle M' + 1 \le M\rangle \sqsubseteq \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge P \\
\Rightarrow & \quad \text{“ } M \text{ is natural. ”} \\
 & \langle M' < M\rangle \sqsubseteq \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge P \qquad . \qquad (24)
\end{array}$$

APPENDIX 1.3 The induction

With these results in hand, we will prove that $R \sqsubseteq W$. As we already have $R \sqsubseteq \neg\langle B\rangle \wedge W$, and $R \sqsubseteq \langle B\rangle \wedge \langle \tau = \infty\rangle \wedge W$ it remains to prove $R \sqsubseteq \langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge W$. The proof is by complete induction on M, which, because of strong bounding, is a natural expression. Specifically we will prove, for any natural i,

$$R \sqsubseteq \langle M = i\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge \langle B\rangle \wedge W$$

follows from the induction hypothesis $R \sqsubseteq \langle M < i\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge \langle B\rangle \wedge W$. In light of calculation (18), we can see that the induction hypothesis implies

$$R \sqsubseteq \langle M < i\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge W \qquad . \qquad (25)$$

Calculate

$$\begin{array}{ll}
 & \langle M = i\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge \langle B\rangle \wedge W \\
= & \quad \text{“ Calculation (19). ”} \\
 & \langle M = i\rangle \wedge (\langle B\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge P); \mathit{tick}; W \\
\sqsupseteq & \quad \text{“ (24). ”} \\
 & (\langle B\rangle \wedge P); (\langle M < i\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge \mathit{tick}); W \\
= & \quad \text{“ Time insensitivity. ”} \\
 & (\langle B\rangle \wedge P); \mathit{tick}; (\langle M < i\rangle \wedge \langle \tau \in \mathsf{nat}\rangle \wedge W) \\
\sqsupseteq & \quad \text{“ Induction hypothesis (25). ”} \\
 & (\langle B\rangle \wedge P); \mathit{tick}; R \\
= & \quad \text{“ } B \text{ is a condition. ”} \\
 & \langle B\rangle \wedge (P; \mathit{tick}; R) \\
\sqsupseteq & \quad \text{“ Definition of } \mathbf{if}\text{. ”} \\
 & \mathbf{if}\ B\ \mathbf{then}(P; \mathit{tick}; R)\ \mathbf{else}\ \mathit{skip}
\end{array}$$

$=$ “ Definition of w (1). ”

$w.R$

$\sqsupseteq$ “ R is recursively refined (5). ”

R .

It is interesting that there is no need to break the proof into zero and non-zero cases. The reason is that one falls into the $\neg B$ case before hitting zero.

REFERENCES

Back, R. J. R. and von Wright, J., (1990). Refinement calculus, part 1: Sequential nondeterministic programs. In de Backer, J. W., de Roever, W.-P., and Rosenberg, G., editors, *Stepwise Refinement of Distributed Systems: Models Formalisms, Correctness*, number 430 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

Hehner, Eric C. R., (1979). **do** considered **od**: A contribution to the programming calculus. *Acta Informatica*, 11:287–304, 1979.

Hehner, Eric C. R., (1984). Predicative programming. *Communications of the ACM*, 27(2):134–151, 1984.

Hehner, Eric C. R., (1993). *A Practical Theory of Programming*. Springer-Verlag, 1993.

Hehner, Eric C. R., (1994). Abstractions of time. In Roscoe, A. W., editor, *A Classical Mind*, chapter 12, pages 195–214. Prentice-Hall International, 1994.

Hoare, C. A. R and He, JiFeng, , (1987). The weakest prespecification. *Information Processing Letters*, 24:127–132, 1987.

Hoare, C. A. R., (1985). Programs are predicates. In Hoare, C. A. R. and Shepherdson, J. C., editors, *Mathematical Logic and Programming Languages*, pages 141–155. Prentice Hall, 1985.

Hoare, C. A. R., (1994). Mathematical models for computer science. Technical report, Oxford University Computing Laboratory, Oxford University, August 1994.

Holt, Richard C., (1991). Healthiness versus realizability in predicate transformers. Technical Report CSRI-240, Computer Systems Research Institute, University of Toronto, 1991.

Morgan, Carroll, (1990). *Programming from Specifications*. Prentice Hall International, 1990.

Norvell, Theodore S., (1993). *A Predicative Theory of Machine Languages and its Application to Compiler Correctness*. PhD thesis, University of Toronto, December 1993.

Norvell, Theodore S., (1994). Machine code programs are predicates too. In Till, David, editor, *Sixth Refinement Workshop*, Workshops in Computing, pages 188–204. Springer Verlag, 1994.

Sekerinski, Emil, (1993). A calculus for predicative programming. In *Mathematics of Program Construction 1992*, number 669 in LNCS, pages 302–322. Springer-Verlag, 1993.

Spivey, J. M., (1989). *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

Tarski, Alfred, (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

v. Karger, Burghard and Hoare, C. A. R, (1994). Sequential calculus. *Information Processing Letters*, 53:123–131, 1994.

BIOGRAPHY

A native of Halifax, Nova Scotia, Theodore Norvell obtained Master's and Ph.D. degrees in Computing Science from the University of Toronto. He has worked in the software development industry and has done postdoctoral work both at the Programming Research Group of Oxford University and at the Software Engineering Research Group of McMaster University. He is currently a professor of engineering at Memorial University of Newfoundland.

His research interests include formal specification and derivation of programs, programming language design, and formal aspects of software and hardware engineering.