# Refinement Semantics and Loop Rules

Eric C.R. Hehner[1], Andrew M. Gravell[2]

[1] Dep't Computer Science, University of Toronto,
Toronto ON M5S 3G4, Canada
hehner@cs.toronto.edu
[2] Electronics and Computer Science Dep't, University of Southampton,
Southampton SO9 5NH UK
amg@ecs.soton.ac.uk

**Abstract.** Refinement semantics is an alternative to least fixpoint semantics that is more useful for programming. From it we derive a variety of rules for **while**-loops, **for**-loops, and loops with intermediate and deep exits. We illustrate the use of these rules with examples.

## 1 Introduction

A specification is a boolean expression whose variables represent quantities of interest. By "boolean expression" we mean an expression of type boolean; we do not mean to restrict the types of variables and subexpressions, nor the operators, within a specification. Quantifiers, functions, terms from the application domain, and terms invented for one particular specification are all welcome. Here is an example specification using $x$ and $n$ as the initial values of two integer variables, $x'$ and $n'$ as their final values, $t$ as the time when execution starts, and $t'$ as the time when execution finishes.

$$n{\geq}0 \;\Rightarrow\; x' = x{\times}2^n \;\wedge\; t' \leq t{+}n$$

A specification is implemented on a computer when, for any initial values of the variables, the computer generates (computes) final values to satisfy the specification. A specification is implementable if, for any initial values of the variables, there are final values to satisfy the specification with nondecreasing time. In our example variables, a specification $S$ is implementable if

$$\forall x, n, t{\cdot}\; \exists x', n', t'{\cdot}\;\; S \wedge t'{\geq}t$$

A program is a specification that has been implemented, so that a computer can execute it. The program notations we use include: $ok$ (the empty program), $x{:=}\,e$ (assignment), $P;Q$ (sequential composition), and **if** $b$ **then** $P$ **else** $Q$ (conditional composition). In variables $x$, $n$, and $t$, they are defined as

$$ok \;=\; x'{=}x \wedge n'{=}n \wedge t'{=}t$$

$$x{:=}\,e \;=\; x'{=}e \wedge n'{=}n \wedge t'{=}t$$

$$P;Q \;=\; \exists x'', n'', t''{\cdot}\quad (\text{for } x', n', t' \text{ substitute } x'', n'', t'' \text{ in } P)$$
$$\wedge\; (\text{for } x, n, t \text{ substitute } x'', n'', t'' \text{ in } Q)$$

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q \;=\; b{\wedge}P \vee \neg b{\wedge}Q$$
$$=\; (b{\Rightarrow}P) \wedge (\neg b{\Rightarrow}Q)$$

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1497-1510, 1999.

There are many useful laws that save us from having to use the definitions directly; for a list of laws see [4]. One such law is the Substitution Law

$x{:=}\,e;\ P \quad = \quad (\text{for } x \text{ substitute } e \text{ in } P)$

which can be proven from the equations just given defining assignment and sequential composition.

Suppose we are given specification $S$. If $S$ is a program, we can execute it. If not, we have some programming to do. That means building a program $P$ such that $S \Leftarrow P$ is a theorem; this is called refinement. Since $S$ is implied by $P$, all computer behavior satisfying $P$ also satisfies $S$. We might refine in steps, finding specifications $R$, $Q$, ... such that $S \Leftarrow R \Leftarrow Q \Leftarrow \ldots \Leftarrow P$.

If $S$ is a specification and $P$ is a program, and $S \Leftarrow P$, then we have an implementation for $S$: to execute $S$ just execute $P$. So we can consider $S$ to be a program also. If $S$ is implementable, and $P$ would be a program but for occurrences of $S$, we still have an implementation of $S$: when $S$ reoccurs (recursively), just reexecute $P$ (recursively). So we can still consider $S$ to be a program. For example,

$n{\geq}0 \ \Rightarrow\ x' = x{\times}2^n \ \wedge\ t' \leq t{+}n$

$\Leftarrow$ **if** $n{\neq}0$ **then** $(x{:=}\,x{\times}2;\ n{:=}\,n{-}1;\ t{:=}\,t{+}1;\ n{\geq}0 \Rightarrow x'{=}x{\times}2^n \wedge t'{\leq}t{+}n)$ **else** $ok$

In this context, we may pronounce $\Leftarrow$ as any of "is implied by", "is refined by", or "is implemented by". The occurrence of $t{:=}\,t{+}1$ is not executed in the sense of having a value computed and stored, but only in the sense that it accounts for the time required to execute other instructions. We could have chosen different time increments and placed them differently; this choice simply counts iterations. Inside the brackets we use the Substitution Law three times (from right to left), and replace **if** and $ok$ by their definitions, to obtain

$n{\geq}0 \ \Rightarrow\ x'{=}x{\times}2^n \wedge t'{\leq}t{+}n$

$\Leftarrow$ $(n{>}0 \ \Rightarrow\ x'{=}x{\times}2^n \wedge t'{\leq}t{+}n\ ) \ \wedge\ (n{=}0 \ \Rightarrow\ x'{=}x \wedge n'{=}n \wedge t'{=}t)$

which is clearly a theorem.

## 2 Notation

Here are all the notations used in this paper, arranged by precedence level.

| | | |
|---|---|---|
| 0. | $0\ 1\ 2\ \infty\ \top\ \bot\ x\ y\ (\ )\ [\ ]$ | numbers, booleans, variables, brackets |
| 1. | $fx$ | application, indexing |
| 2. | $2^x$ | exponentiation |
| 3. | $\times$ | multiplication |
| 4. | $+\ -$ | addition, subtraction |
| 5. | $=\ \neq\ <\ >\ \leq\ \geq$ | comparisons |
| 6. | $\neg$ | negation |
| 7. | $\wedge$ | conjunction |
| 8. | $\vee$ | disjunction |
| 9. | $\Rightarrow\ \Leftarrow$ | implications |
| 10. | $:=$ **if then else** **while do** **for do** | assignment, conditional, loops |
| 11. | $\forall\cdot\ \exists\cdot\ ;$ | quantifications, sequential composition |
| 12. | $=\ \Rightarrow\ \Leftarrow$ | equation, implications |

Superscripting serves to bracket all operations within it. The infix operator – associates from left to right. The infix operators $\times$ + $\wedge$ $\vee$ ; are associative (they associate in both directions). On levels 5, 9, and 12 the operators are continuing; for example, $a{=}b{=}c$ neither associates to the left nor associates to the right, but means $a{=}b \wedge b{=}c$ . On any one of these levels, a mixture of continuing operators can be used. For example, $a{\leq}b{<}c$ means $a{\leq}b \wedge b{<}c$ . The operators $=$ $\Longrightarrow$ $\Longleftarrow$ are identical to $=$ $\Rightarrow$ $\Leftarrow$ except for precedence. Square brackets [ ] universally quantify over all state variables (initial and final values), including time.

We use $\sigma$ to stand for all the unprimed variables, $\sigma'$ for primed versions of the same variables, and $\sigma''$ for double-primed versions. If $e$ is an expression in unprimed variables, then $e'$ is the same expression as $e$ but with primes on all the variables, and $e''$ has double-primes on all the variables. If $P$ is a specification (a boolean expression in unprimed and primed variables), then $P\sigma_{\sigma''}$ is the same as $P$ but with all its unprimed variables replaced with the corresponding double-primed variables.

## 3 Least Fixpoints

Least fixpoints are a standard way to define the semantics of loop constructs. "Least" means least refined, so in the context of this paper it means least strong; to avoid confusion about the ordering, we shall say "weakest". The **while**-loop can be defined by the following two axioms.

**while** $b$ **do** $S$ $=$ **if** $b$ **then** ($S$; **while** $b$ **do** $S$) **else** $ok$

[$W$ $=$ **if** $b$ **then** ($S$; $W$) **else** $ok$] $\Rightarrow$ [$W \Rightarrow$ **while** $b$ **do** $S$]

The first axiom says that **while** $b$ **do** $S$ is a fixpoint of the function (in variable $W$ )

**if** $b$ **then** ($S$; $W$) **else** $ok$

The second axiom says that **while** $b$ **do** $S$ is weaker than or equal to any fixpoint of that function. Together, they say that **while** $b$ **do** $S$ is the weakest fixpoint.

In place of fixpoints, we can use prefixpoints to define the semantics of loop constructs. The **while**-loop can be defined by the following two axioms.

**while** $b$ **do** $S$ $\Longrightarrow$ **if** $b$ **then** ($S$; **while** $b$ **do** $S$) **else** $ok$

[$W$ $\Rightarrow$ **if** $b$ **then** ($S$; $W$) **else** $ok$] $\Rightarrow$ [$W \Rightarrow$ **while** $b$ **do** $S$]

The weakest fixpoint and weakest prefixpoint definitions are equivalent, but the latter may be preferred because, from it, the former is easily proven (algebraically), but from the former the proof of the latter is harder (topological).

When we include time among the observable properties of a computation, we can strengthen our loop semantics by using the weakest progressive prefixpoint [9]. This time we define the **while**-loop as follows.

**while** $b$ **do** $S$ $\Longrightarrow$ $t'{\geq}t$

**while** $b$ **do** $S$ $\Longrightarrow$ **if** $b$ **then** ($S$; $t{:=}t{+}1$; **while** $b$ **do** $S$) **else** $ok$

[$W$ $\Rightarrow$ $t'{\geq}t$] $\wedge$ [$W \Rightarrow$ **if** $b$ **then** ($S$; $t{:=}t{+}1$; $W$) **else** $ok$]

$\Rightarrow$ [$W \Rightarrow$ **while** $b$ **do** $S$]

This definition is not equivalent to the previous two. With it we can prove

**while** $\top$ **do** $ok$ $=$ $t'{=}\infty$

which says sensibly that the loop takes infinite time, whereas the previous two say

$$\textbf{while } \top \textbf{ do } ok \;=\; \top$$

which tells us nothing useful. The only disadvantage of the weakest progressive prefixpoint is that it is tied to the particular measure of time that counts iterations, whereas the others can be used with a real-valued time variable that measures the real execution time.

## 4 Refinement Semantics

All three of the least fixpoint semantics (weakest fixpoint, weakest prefixpoint, weakest progressive prefixpoint) say what a loop is by saying how it can be implemented (or refined). For example, the axiom

$$[W \;\Rightarrow\; \textbf{if } b \textbf{ then } (S;\, W) \textbf{ else } ok] \;\Rightarrow\; [W \Rightarrow \textbf{while } b \textbf{ do } S]$$

says that $\textbf{while } b \textbf{ do } S$ can be implemented (refined) by $W$ if

$$W \;\Rightarrow\; \textbf{if } b \textbf{ then } (S;\, W) \textbf{ else } ok$$

Refinement semantics says what a loop is by saying what it implements (or refines). Whereas a least fixpoint semantics tells the implementers what they want to know, refinement semantics tells programmers what they want to know in order to use loops as programming notations.

As a first effort at refinement semantics, we might try

$$\textbf{while } b \textbf{ do } S \;\Leftarrow\; \textbf{if } b \textbf{ then } (S;\, \textbf{while } b \textbf{ do } S) \textbf{ else } ok$$

$$[W \;\Leftarrow\; \textbf{if } b \textbf{ then } (S;\, W) \textbf{ else } ok] \;\Rightarrow\; [W \Leftarrow \textbf{while } b \textbf{ do } S]$$

making $\textbf{while } b \textbf{ do } S$ the greatest (strongest) postfixpoint. The second of these axioms says that $\textbf{while } b \textbf{ do } S$ implements (refines) $W$ if

$$W \;\Leftarrow\; \textbf{if } b \textbf{ then } (S;\, W) \textbf{ else } ok$$

With this axiom alone, $\textbf{while } b \textbf{ do } S$ might just be $\bot$, but according to the first axiom it must be implemented (refined) by its first unrolling. Unfortunately, that definition sometimes makes $\textbf{while } b \textbf{ do } S$ unimplementable (even when $S$ is implementable). Restricting $W$ to be implementable and insisting that $\textbf{while } b \textbf{ do } S$ be implementable is unfortunately inconsistent. Dropping the first axiom and restricting $W$ to be implementable in the second, we can still prove

$$x'{=}2 \wedge t'{=}\infty \;\Leftarrow\; \textbf{while } \top \textbf{ do } t := t{+}1$$

By itself, this is not a problem. Although it may be strange to say that an infinite loop results in a final value of 2 for variable $x$, this final value is promised only at time $\infty$ when no-one can observe the contrary. But we can equally well prove

$$x'{=}3 \wedge t'{=}\infty \;\Leftarrow\; \textbf{while } \top \textbf{ do } t := t{+}1$$

and hence, by boolean algebra,

$$\bot \;\Leftarrow\; \textbf{while } \top \textbf{ do } t := t{+}1$$

and so, by transitivity,

$$x'{=}2 \wedge t'{=}t \;\Leftarrow\; \textbf{while } \top \textbf{ do } t := t{+}1$$

which promises a final value of 2 for variable $x$ at the present time, when we can easily observe the contrary. Greatest fixpoints just don't work.

To avoid all these problems and still provide a semantics oriented toward programming rather than implementation, we define the refinement semantics of **while** as follows. Let

$W \Leftarrow$ **while** $b$ **do** $S$

be an abbreviation (syntactic sugar) for the refinement

$W \Leftarrow$ **if** $b$ **then** $(S; W)$ **else** $ok$

Refinement semantics does not ascribe any meaning to the **while**-loop by itself, but only to the refinement.

As an example, we previously proved

$n{\geq}0 \;\Rightarrow\; x' = x{\times}2^n \;\wedge\; t' \leq t{+}n$

$\Leftarrow$ **if** $n{\neq}0$ **then** $(x{:=}\, x{\times}2;\; n{:=}\, n{-}1;\;\; t{:=}\, t{+}1;\;\; n{\geq}0 \Rightarrow x'{=}x{\times}2^n \wedge t'{\leq}t{+}n)$ **else** $ok$

hence refinement semantics says

$n{\geq}0 \;\Rightarrow\; x' = x{\times}2^n \;\wedge\; t' \leq t{+}n$

$\Leftarrow$ **while** $n{\neq}0$ **do** $(x{:=}\, x{\times}2;\;\; n{:=}\, n{-}1;\;\; t{:=}\, t{+}1)$

Programming constructs are required to be monotonic, which means for the **while**-loop

$[P{\Rightarrow}Q] \Rightarrow [$**while** $b$ **do** $P \implies$ **while** $b$ **do** $Q]$

Since refinement semantics does not give a meaning to the **while**-loop, we cannot prove monotonicity in this form. Instead we can prove monotonicity in the form

$[W \Leftarrow$ **while** $b$ **do** $P] \;\wedge\; [P \;\Leftarrow\; Q] \;\Rightarrow\; [W \Leftarrow$ **while** $b$ **do** $Q]$

which is exactly the Law of Stepwise Refinement used by programmers to refine a specification in a sequence of steps. Similarly we can prove the Law of Partwise Refinement

$[W \Leftarrow$ **while** $b$ **do** $P] \wedge [X \Leftarrow$ **while** $b$ **do** $Q]$

$\Rightarrow\; [W{\wedge}X \Leftarrow$ **while** $b$ **do** $P{\wedge}Q]$

which allows programmers to write a specification in parts, refine the parts separately (with the same structure), and then combine the refinements to get a solution to the combined specification.

## 5 Comparison of Least Fixpoint and Refinement Semantics

If the body of a loop does not decrease variable $x$, then the loop does not decrease $x$. The refinement

$x'{\geq}x \;\Leftarrow$ **while** $b$ **do** $x{:=}\, x{+}1$

is an easy theorem by refinement semantics, but not a theorem at all by any of the least fixpoint semantics. The problem is that the loop condition $b$ might be $\top$ and the loop execution is infinite. It may seem reasonable to refrain from concluding anything about final values after an infinite computation, but

$t'{\geq}t \;\Leftarrow$ **while** $b$ **do** $t{:=}\, t{+}1$

is reasonable even if the computation is infinite. It is easily provable by refinement semantics. It is an axiom in weakest progressive prefixpoint semantics. It is not provable by weakest fixpoint semantics, nor (of course) by weakest prefixpoint semantics.

The next example

$x{<}0 \Rightarrow t'{=}\infty \;\Leftarrow$ **while** $x{\neq}0$ **do** $(x{:=}\, x{-}1;\; t{:=}\, t{+}1)$

informs us that for negative initial $x$, the computation is infinite. This too is easily provable by refinement semantics, provable with difficulty by weakest progressive prefixpoint semantics, but not provable by weakest fixpoint semantics, nor (of

course) by weakest prefixpoint semantics.

The final example for the purpose of comparison

$t'{=}3 \Leftarrow$ **while** $\top$ **do** $t{:=}\,t{+}1$

says, unreasonably, that this computation will end at time 3. To their credit, it is not provable by any of the least fixpoint semantics. To its discredit, it is provable by refinement semantics. However, refinement semantics says that this is just an abbreviation for

$t'{=}3 \Leftarrow$ **if** $\top$ **then** $(t{:=}\,t{+}1;\ t'{=}3)$ **else** $ok$

and as stated earlier, to consider that $t'{=}3$ is implemented by this recursion, it must first be implementable. Since it is not, it is excluded from consideration.

As a practical matter, it is convenient to be able to prove invariance (safety) properties without having to prove termination (or liveness) first. Refinement semantics allows this separation of concerns; the various least fixpoint semantics do not. With the addition of communication (input and output, not covered in this paper, see [4]), nonterminating executions can perform useful computation, so a semantics that does not insist on termination is useful.

## 6 Variant

A variant $v$ is an expression in unprimed variables, together with an ordering $<$ satisfying the well-founded induction axiom:

$[(v'{<}v;\ \neg P) \vee P] \Rightarrow [P]$

or, more verbosely [2],

(0) $[(\forall \sigma'' \cdot\ v''{<}v \Rightarrow P\sigma_{\sigma''}) \Rightarrow P] \Rightarrow [P]$

When specialized to the natural numbers,

$(\forall n \cdot (\forall m \cdot m{<}n \Rightarrow Pm) \Rightarrow Pn) \quad \Rightarrow \quad (\forall n \cdot Pn)$

it is sometimes called "course-of-values induction" or "Noetherian induction".

When the body of a loop decreases a variant, refinement semantics is a consequence of least fixpoint semantics. All we need is the prefixpoint axiom

(1) **while** $b$ **do** $v'{<}v \quad \Longrightarrow \quad$ **if** $b$ **then** $(v'{<}v;$ **while** $b$ **do** $v'{<}v)$ **else** $ok$

Now suppose

(2) $S \quad \Leftarrow \quad$ **if** $b$ **then** $(v'{<}v;\ S)$ **else** $ok$

From (0), (1), and (2) we can prove

(3) $S \quad \Leftarrow \quad$ **while** $b$ **do** $v'{<}v$

Proof: We start with what we want to prove.

$[S \quad \Leftarrow \quad$ **while** $b$ **do** $v'{<}v]$ use (0) with (3) as $P$

$\Leftarrow \quad [(\forall \sigma'' \cdot\ v''{<}v \Rightarrow (S \quad \Leftarrow \quad$ **while** $b$ **do** $v'{<}v)\sigma_{\sigma''}) \Rightarrow (S \quad \Leftarrow \quad$ **while** $b$ **do** $v'{<}v)]$

To prove this, we prove the final implication, making use of its context (the other information on the same line) when necessary.

$S$ use (2)

$\Leftarrow$ **if** $b$ **then** $(v'{<}v;\ S)$ **else** $ok$ expand the ;

$=$ **if** $b$ **then** $(\exists \sigma'' \cdot\ v''{<}v \ \wedge\ S\sigma_{\sigma''})$ **else** $ok$ strengthen $S\sigma_{\sigma''}$ using context

$\Leftarrow$ **if** $b$ **then** $(\exists \sigma'' \cdot\ v''{<}v \ \wedge$ (**while** $b''$ **do** $v'{<}v''))$ **else** $ok$ contract to ;

$=$ **if** $b$ **then** $(v'{<}v;$ **while** $b$ **do** $v'{<}v)$ **else** $ok$ use (1)

$\Leftarrow$ **while** $b$ **do** $v'{<}v$

Thus, in the presence of a variant, refinement semantics is sound relative to fixpoint semantics. In fact, in the presence of a variant, there is exactly one fixpoint, and all postfixpoints are weaker than or equal to the fixpoint. Although the loop body $v'<v$ appears to do nothing but decrease the variant, the result generalizes to loops whose bodies do other work while decreasing the variant (the variant $v$ and its relation $<$ can be defined so that $v'<v$ includes useful work). Although the result has been stated and proven for **while**-loops, it generalizes to any recursion in which each recursive call occurs in a monotonic context and the variant is decreased before the call.

## 7 Rule of Invariants and Variants

Since the least fixpoint semantics is oriented to implementation rather than programming, programmers are not able to use it directly. Instead, they have used rules that can be derived from it. The best-known rule for the use of **while**-loops is the Rule of Invariants and Variants. The version in [8] is as follows: Let $I$ (the invariant) be a boolean expression in unprimed variables, and let $v$ (the variant) be an integer expression in unprimed variables. Then

$$I \Rightarrow I' \wedge \neg b' \quad \Leftarrow \quad \textbf{while } b \textbf{ do } I \wedge b \Rightarrow I' \wedge 0 \leq v' < v$$

If the body of the loop maintains the invariant and decreases the variant but not below 0 , then the loop maintains the invariant and negates the condition.

The Rule of Invariants and Variants is a special case of the refinement semantics. It is easy to prove

$$I \Rightarrow I' \wedge \neg b' \quad \Leftarrow \quad \textbf{if } b \textbf{ then } (I \wedge b \Rightarrow I' \wedge 0 \leq v' < v;\ I \Rightarrow I' \wedge \neg b') \textbf{ else } ok$$

but that doesn't prove termination. To use refinement semantics to prove that the variant gives termination, we augment the specification with $0 \leq v \Rightarrow t' \leq t+v$ , and add $t := t+1$ to the loop body. We prove

$$I \wedge 0 \leq v \Rightarrow I' \wedge \neg b' \wedge t' \leq t+v$$
$$\Leftarrow \quad \textbf{while } b \textbf{ do } (I \wedge 0 \leq v \wedge b \Rightarrow I' \wedge 0 \leq v' < v;\ t := t+1)$$

by proving

$$I \wedge 0 \leq v \Rightarrow I' \wedge \neg b' \wedge t' \leq t+v$$
$$\Leftarrow \quad \textbf{if } b$$
$$\textbf{then } (I \wedge 0 \leq v \wedge b \Rightarrow I' \wedge 0 \leq v' < v;\ t := t+1;\ I \wedge 0 \leq v \Rightarrow I' \wedge \neg b' \wedge t' \leq t+v)$$
$$\textbf{else } ok$$

The proof is easy and is omitted.

It is well-known that the Rule of Invariants and Variants is incomplete; for example, it cannot be used as it stands to prove

$$x' = x \quad \Leftarrow \quad \textbf{while } \bot \textbf{ do } \top$$

because $x' = x$ cannot be rewritten in the required form. The standard work-around is to allow a slightly different form of the rule, using so-called “logical constants”. Instead of the preceding, we prove

$$x' = x \quad \Leftarrow \quad \forall X \cdot\ x = X \Rightarrow x' = X$$
$$\forall X \cdot\ (x = X \Rightarrow x' = X \quad \Leftarrow \quad \textbf{while } \bot \textbf{ do } \top)$$

Here is an example of the use of the Rule of Invariants and Variants.

$n{\geq}0 \Rightarrow x'{=}2^n \quad \Leftarrow \quad x{:=}\,1;\ n{\geq}0 \Rightarrow x' = x{\times}2^n$

$n{\geq}0 \Rightarrow x' = x{\times}2^n \quad \Leftarrow \quad \textbf{while}\ n{\neq}0\ \textbf{do}\ (x{:=}\,x{\times}2;\ n{:=}\,n{-}1)$

To put the specification $n{\geq}0 \Rightarrow x' = x{\times}2^n$ in the proper form to use the rule, we need to find an invariant and a variant. The variant is obvious: $n$. For the invariant, we need a "logical constant" $C$; the invariant is then $0{\leq}n \wedge x{\times}2^n{=}C$.

$n{\geq}0 \Rightarrow x'{=}2^n \quad \Leftarrow \quad x{:=}\,1;\ \forall C{\cdot}\ 0{\leq}n \wedge x{\times}2^n{=}C \ \Rightarrow\ 0{\leq}n' \wedge x'{\times}2^{n'}{=}C \wedge n'{=}0$

$\forall C{\cdot}\ (\quad 0{\leq}n \wedge x{\times}2^n{=}C \ \Rightarrow\ 0{\leq}n' \wedge x'{\times}2^{n'}{=}C \wedge n'{=}0$

$\quad \Leftarrow\ \textbf{while}\ n{\neq}0\ \textbf{do} \quad 0{\leq}n \wedge x{\times}2^n{=}C \wedge n{\neq}0$

$\quad\quad\quad\quad\quad\quad \Rightarrow\ 0{\leq}n' \wedge x'{\times}2^{n'}{=}C \wedge 0{\leq}n'{<}n)$

$0{\leq}n \wedge x{\times}2^n{=}C \wedge n{\neq}0 \Rightarrow 0{\leq}n' \wedge x'{\times}2^{n'}{=}C \wedge 0{\leq}n'{<}n \quad \Leftarrow \quad x{:=}\,x{\times}2;\ n{:=}\,n{-}1$

For this very ordinary example, the Rule of Invariants and Variants has made the proof considerably harder than the refinement semantics proof.

There is a hidden subtlety in the Rule of Invariants and Variants: the body is unimplementable unless $I{\wedge}b \ \Rightarrow\ 0{<}v$. The rule is still sound without this constraint, but then the loop body cannot be implemented.

For further special cases of this rule worth mentioning see [3].

## 8 Terminating While-Loop Rule

Early work [5,1] presented semantics and proof rules by a pair of boolean expressions (then called "predicates"). One expression of the pair characterized initial states, and the other characterized final states. It was soon realized that most often the final state depends on the initial state, and "logical constants" were needed to relate the two states. All current work (VDM, Z, B, TLA, refinement calculus) uses two related sets of variables (undecorated and decorated) in the same boolean expression, making "logical constants" unnecessary. An invariant is a boolean expression about one state; it is a remnant of the early work. The Rule of Invariants and Variants is leftover from the days when the initial and final states had to be described separately and then related by "logical constants". There is no longer any need to do so. We now present a new rule, the terminating **while**-loop rule, which is simpler, more convenient, and more general.

At the same time as we get rid of invariants, independently we take the opportunity to relabel the variant as an upper bound on the remaining execution time, measured as a count of iterations. The Rule of Invariants and Variants uses this time bound (the variant) to imply termination, then throws it away; it does not appear in the loop specification $I \Rightarrow I' \wedge \neg b'$. But a time bound is interesting information in its own right, so we won't throw it away.

Let $f$ be a nonnegative real-valued function of the state $\sigma$ and let $\delta$ be a positive real constant. Then

$W \wedge t'{\leq}t{+}f\sigma \ \Leftarrow\ \textbf{while}\ b\ \textbf{do}\ S$

if

$W \wedge t'{\leq}t{+}f\sigma \ \Leftarrow\ \textbf{if}\ b\ \textbf{then}\ (S;\ W \wedge t'{\leq}t{+}f\sigma{+}\delta)\ \textbf{else}\ ok$

To use this rule on our example problem we must restrict $n$ to be a natural variable. Then

$x' = x \times 2^n \ \wedge \ t' \leq t+n \quad \Leftarrow \quad$ **while** $n \neq 0$ **do** $(x := x \times 2; \ n := n-1)$

because

$x' = x \times 2^n \ \wedge \ t' \leq t+n$

$\Leftarrow$ **if** $n \neq 0$ **then** $(x := x \times 2; \ n := n-1; \ x' = x \times 2^n \ \wedge \ t' \leq t+n+1)$ **else** $ok$

The terminating **while**-loop rule can be proven both by refinement semantics (trivially) and by least fixpoint semantics (harder).

## 9 Loops with Exits

Loops with intermediate or deep exits are awkward to define by least fixpoint semantics, but quite straightforward by refinement semantics. For example, to prove

```
L ⇐ loop
        P;
        exit 1 when b;                      exit one level of loop
        Q;
        loop
            exit 2 when c;                  exit two levels of loop
            R;
            exit 1 when d                   exit one level of loop
        end
    end
```

find a specification $M$ for the inner loop and prove

$L \ \Leftarrow \ P; \ \textbf{if } b \textbf{ then } ok \textbf{ else } (Q; \ M; \ L)$

$M \ \Leftarrow \ \textbf{if } c \textbf{ then } ok \textbf{ else } (R; \ \textbf{if } d \textbf{ then } L \textbf{ else } M)$

Refinement semantics requires a specification for every loop, which is recommended programming practice anyway.

## 10 For-Loops

The **for**-loop has usually been treated as a syntactic sugar for a **while**-loop, given neither a semantics of its own nor rules for its use. We now offer four rules for the use of **for**-loops; one of them is taken from [4], and is similar to [6]; the other three are new. Any of the three new rules can serve as the refinement semantics of the **for**-loop.

We shall use the syntax

**for** $i := m,..n$ **do** $Si$

for controlled iteration, where $i$ is a fresh identifier, not assignable within the loop body, $m$ and $n$ are integer expressions evaluated once, $m \leq n$, and $Si$ is a specification indexed by $i$. The asymmetric notation $m,..n$ indicates that $m$ is included and $n$ excluded, so there are $n-m$ iterations. This asymmetry simplifies the rules for the use of **for**-loops.

**Rule I (Invariant).** Our first **for**-loop rule is taken from [4]. Let $Ii$ be a boolean expression in unprimed variables indexed by $i$. Then

$$m{\le}n \wedge Im \Rightarrow I'n \quad \Leftarrow \quad \textbf{for } i{:=}\, m,..n \textbf{ do } m{\le}i{<}n \wedge Ii \;\Rightarrow\; I'(i{+}1)$$

Here is an example of the use of Rule I. Let $Ii = x{=}2^i$.

$$n{\ge}0 \Rightarrow x'{=}2^n \quad \Leftarrow \quad x{:=}\,1;\; 0{\le}n \wedge x{=}2^0 \Rightarrow x'{=}2^n$$
$$0{\le}n \wedge x{=}2^0 \Rightarrow x'{=}2^n \quad \Leftarrow \quad \textbf{for } i{:=}\,0,..n \textbf{ do } 0{\le}i{<}n \wedge x{=}2^i \Rightarrow x'{=}2^{i+1}$$
$$0{\le}i{<}n \wedge x{=}2^i \Rightarrow x'{=}2^{i+1} \quad \Leftarrow \quad x{:=}\,x{\times}2$$

Like the **while**-loop Rule of Invariants and Variants, Rule I is incomplete; for example, it cannot be used as it stands to prove

$$x'{=}x \quad \Leftarrow \quad \textbf{for } i{:=}\,0,..0 \textbf{ do } \top$$

because $x'{=}x$ cannot be rewritten in the required form. However, Rule I becomes complete if we allow the use of "logical constants". Instead of the preceding, we prove

$$x'{=}x \quad \Leftarrow \quad \forall X{\cdot}\; x{=}X \Rightarrow x'{=}X$$
$$\forall X{\cdot}\; (x{=}X \Rightarrow x'{=}X \quad \Leftarrow \quad \textbf{for } i{:=}\,0,..0 \textbf{ do } \top)$$

As in the Rule of Invariants and Variants, the invariant is a vestige of earlier programming methods, and is completely superseded by the following three rules.

**Rule F (Forward).** Let $Fi$ be a specification indexed by $i$. Then

$$m{\le}n \Rightarrow Fm \quad \Leftarrow \quad \textbf{for } i{:=}\, m,..n \textbf{ do } m{\le}i{<}n \Rightarrow Si$$

if

$$\forall i{:}\; m,..n{\cdot}\; (Si;\, F(i{+}1)) \Rightarrow Fi$$
$$ok \;\Rightarrow\; Fn$$

Specification $Fi$ describes what has yet to be done at iteration $i$. At the beginning, everything ( $Fm$ ) has yet to be done. At iteration $i$, $Fi$ will be done by doing $Si$ and then $F(i{+}1)$. At the end, $Fn$ will be done by doing nothing more ( $ok$ ).

Here is an example of the use of Rule F. Define $Fi = x'{=}x{\times}2^{n-i}$. Then

$$n{\ge}0 \Rightarrow x'{=}2^n \quad \Leftarrow \quad x{:=}\,1;\; 0{\le}n \Rightarrow x'{=}x{\times}2^n$$
$$0{\le}n \Rightarrow x'{=}x{\times}2^n \quad \Leftarrow \quad \textbf{for } i{:=}\,0,..n \textbf{ do } x{:=}\,x{\times}2$$

because

$$\forall i{:}\; 0,..n{\cdot}\; (x{:=}\,x{\times}2;\; x'{=}x{\times}2^{n-(i+1)}) \;\Rightarrow\; x'{=}x{\times}2^{n-i}$$
$$ok \;\Rightarrow\; x'{=}x{\times}2^{n-n}$$

The soundness of Rule F can be demonstrated by correspondence with the following computation.

$$Fm \text{ where}$$
$$Fi \;\Leftarrow\; \textbf{if } i{=}n \textbf{ then } ok \textbf{ else } (Si;\, F(i{+}1))$$

which says: execute procedure $F$ with argument $m$, where procedure $F$ with parameter $i$ is implemented as **if** $i{=}n$ **then** $ok$ **else** $(Si;\, F(i{+}1))$. This is the standard **while**-loop definition of a **for**-loop. If we accept that this execution is what we intended, then Rule F is sound.

To show the completeness of Rule F, let $Fi = Si;\, S(i{+}1);\, ...;\, S(n{-}1)$. Then $Fm$ specifies the **for**-loop exactly.

**Rule B (Backward).** Let $Bi$ be a specification indexed by $i$. Then

$m{\le}n \Rightarrow Bn \Leftarrow$ **for** $i{:=}\, m,..n$ **do** $m{\le}i{<}n \Rightarrow Si$

if

$ok \Rightarrow Bm$

$\forall i{:}\ m,..n{\cdot}\ (Bi;\ Si) \Rightarrow B(i{+}1)$

Specification $Bi$ describes what has been done up to iteration $i$. At the beginning, when we have done nothing ( $ok$ ), we have done $Bm$. When we have done $Bi$ and then we do $Si$, then we have done $B(i{+}1)$. At the end we have done everything ( $Bn$ ).

Here is an example of the use of Rule B. Define $Bi = x'{=}x{\times}2^i$. Then

$n{\ge}0 \Rightarrow x'{=}2^n \Leftarrow x{:=}\,1;\ 0{\le}n \Rightarrow x'{=}x{\times}2^n$

$0{\le}n \Rightarrow x'{=}x{\times}2^n \Leftarrow$ **for** $i{:=}\,0,..n$ **do** $x{:=}\,x{\times}2$

because

$ok \Rightarrow x'{=}x{\times}2^0$

$\forall i{:}\ 0,..n{\cdot}\ (x'{=}x{\times}2^i;\ x{:=}\,x{\times}2) \Rightarrow x'{=}x{\times}2^{i+1}$

The soundness of Rule B can be demonstrated by correspondence with the following computation.

$Bn$ where

$Bi \Leftarrow$ **if** $i{=}m$ **then** $ok$ **else** $(B(i{-}1);\ S(i{-}1))$

This computation dives into its recursions from $n$ down to $m$, executing the $Si$ on the way back up. If we accept this as an execution of the **for**-loop, then Rule B is sound.

To show the completeness of Rule B, let $Bi = Sm;\ S(m{+}1);\ ...;\ S(i{-}1)$. Then $Bn$ specifies the **for**-loop exactly.

**Rule G (General).** Let $Gik$ be a specification indexed by $i$ and $k$. Then

$m{\le}n \Rightarrow Gmn \Leftarrow$ **for** $j{:=}\, m,..n$ **do** $m{\le}j{<}n \Rightarrow Gj(j{+}1)$

if

$m{=}n \wedge ok \Rightarrow Gmn$

$\forall i, j, k{\cdot}\ m{\le}i{<}j{<}k{\le}n \wedge (Gij;\ Gjk) \Rightarrow Gik$

Here is an example of the use of Rule G. Define $Gik = x'{=}x{\times}2^{k-i}$. Then

$n{\ge}0 \Rightarrow x'{=}2^n \Leftarrow x{:=}\,1;\ 0{\le}n \Rightarrow x'{=}x{\times}2^n$

$0{\le}n \Rightarrow x'{=}x{\times}2^n \Leftarrow$ **for** $j{:=}\,0,..n$ **do** $0{\le}j{<}n \Rightarrow x'{=}x{\times}2^{(j+1)-j}$

$0{\le}j{<}n \Rightarrow x'{=}x{\times}2^{(j+1)-j} \Leftarrow x{:=}\,x{\times}2$

because

$m{=}n \wedge x'{=}x \Rightarrow x'{=}x{\times}2^{n-m}$

$\forall i, j, k{\cdot}\ 0{\le}i{<}j{<}k{\le}n \wedge (x'{=}x{\times}2^{j-i};\ x'{=}x{\times}2^{k-j}) \Rightarrow x'{=}x{\times}2^{k-i}$

The soundness of Rule G can be demonstrated by correspondence with the following computation.

**if** $m{=}n$ **then** $ok$ **else** $Gmn$ where

$Gik \Leftarrow$ **if** $i{+}1{=}k$ **then** $Si$ **else** $(i{<}j'{<}k;\ Gij;\ Gjk)$

If we accept this as an execution of the **for**-loop, then Rule G is sound.

To show the completeness of Rule G, let $Gik = Si;\ S(i{+}1);\ ...;\ S(k{-}1)$. Then $Gmn$ specifies the **for**-loop exactly.

## 11 Comparison of the For-Loop Rules

Each rule asks us to think about the computation in a different way.

Rule I: what is true between iterations?

Rule F: what is true of a final segment of the iterations?

Rule B: what is true of an initial segment of the iterations?

Rule G: what is true of an arbitrary segment of the iterations?

Rules F and B require us to choose a direction; rules I and G are directionless. Rules F, B, and G are like the definition of lists: we may construct lists by appending items, prepending items, or catenation of lists.

Each of the rules F, B, and G is a special case of each of the other two, so all three of them are sound and complete if one of them is. In one respect, Rule G seems to demand more than necessary: it asks us to prove $(Gij; Gjk) \Rightarrow Gik$ for all $j$ between $i$ and $k$, when one such $j$ is enough. Rules B and F are the special cases of Rule G when $j$ is chosen to be either $i{+}1$ or $k{-}1$. But we have to specify the effect of the **for**-loop from $m$ to $n$ anyway, and so it may be easy to generalize the specification to an arbitrary segment.

Rules F and B ask us to specify a single step ( $Si$ ) in addition to a segment ( $Fi$ or $Bi$ ); Rules I and G do not, since $Ii \Rightarrow I(i{+}1)$ and $Gi(i{+}1)$ are single steps. We can rewrite Rule F so that it does not require us to specify $Si$, as follows.

$$m \le n \Rightarrow Fm \Leftarrow \textbf{for } i := m,..n \textbf{ do } m \le i < n \Rightarrow \neg(\neg Fi;\ F(i{+}1)^{\cup})$$

where $^{\cup}$ is transposition (put primes on all unprimed variables and simultaneously remove primes from all primed variables). The expression $\neg(\neg Fi;\ F(i{+}1)^{\cup})$ is known as the weakest prespecification of $Fi$ and $F(i{+}1)$ [7]. We can similarly rewrite Rule B so that it does not require us to specify $Si$, as follows.

$$m \le n \Rightarrow Bm \Leftarrow \textbf{for } i := m,..n \textbf{ do } m \le i < n \Rightarrow \neg(Bi^{\cup};\ \neg B(i{+}1))$$

The expression $\neg(Bi^{\cup};\ \neg B(i{+}1))$ is the weakest postspecification of $Bi$ and $B(i{+}1)$. We did not do so, judging that the specification of $Si$ was the lesser evil.

For the record, the rules remain valid when $n{=}\infty$. Also for the record, the **for**-loop rules could be stated more simply as follows:

Rule I: $Im \Rightarrow I'n \Leftarrow \textbf{for } i := m,..n \textbf{ do } Ii \Rightarrow I'(i{+}1)$

Rule F: $Fm \Leftarrow \textbf{for } i := m,..n \textbf{ do } Si$

Rule B: $Bn \Leftarrow \textbf{for } i := m,..n \textbf{ do } Si$

Rule G: $Gmn \Leftarrow \textbf{for } i := m,..n \textbf{ do } Gi(i{+}1)$

The missing parts can be incorporated into the remaining parts. The way we have stated the rules is longer but more convenient for use.

## 12 Examples

In practice, the differences among the rules may be small. The most common use of a **for**-loop is to do something to every item (element) of a list (array). As an example, let's just add 1 to every item of list $L$. Formally,

$$\#L' = \#L \wedge (\forall j:\ 0,..\#L \cdot\ L'j = Lj + 1)$$

For Rule I we have to introduce "logical constant" $M$ to be the initial value of $L$. The four rules require us to invent the following four specifications.

$$
\begin{array}{lcl}
Ii & = & \#L{=}\#M \wedge (\forall j{:}\ 0,..i{\cdot}\ Lj = Mj + 1) \wedge (\forall j{:}\ i,..\#L{\cdot}\ Lj = Mj) \\
Fi & = & \#L'{=}\#L \wedge (\forall j{:}\ 0,..i{\cdot}\ L'j = Lj) \wedge (\forall j{:}\ i,..\#L{\cdot}\ L'j = Lj + 1) \\
Bi & = & \#L'{=}\#L \wedge (\forall j{:}\ 0,..i{\cdot}\ L'j = Lj + 1) \wedge (\forall j{:}\ i,..\#L{\cdot}\ L'j = Lj) \\
Gik & = & \#L'{=}\#L \wedge (\forall j{:}\ 0,..i{\cdot}\ L'j = Lj) \wedge (\forall j{:}\ i,..k{\cdot}\ L'j = Lj + 1) \\
 & & \wedge\ (\forall j{:}\ k,..\#L{\cdot}\ L'j = Lj)
\end{array}
$$

Our next example is cubing by addition.

$$
\begin{array}{ll}
 & x'{=}n^3 \\
\Leftarrow & x{:=}\ 0;\ \ y{:=}\ 1;\ \ z{:=}\ 6;\ \ \textbf{for}\ i{:=}\ 0;..n\ \textbf{do}\ (x{:=}\ x{+}y;\ \ y{:=}\ y{+}z;\ \ z{:=}\ z{+}6)
\end{array}
$$

The four rules require us to invent the following four specifications.

$$
\begin{array}{lcl}
Ii & = & x{=}i^3 \wedge y = 3i^2{+}3i{+}1 \wedge z = 6i{+}6 \\
Fi & = & x{=}i^3 \wedge y = 3i^2{+}3i{+}1 \wedge z = 6i{+}6 \Rightarrow x'{=}n^3 \wedge y' = 3n^2{+}3n{+}1 \wedge z' = 6n{+}6 \\
Bi & = & x{=}0 \wedge y{=}1 \wedge z{=}6 \Rightarrow x' = i^3 \wedge y' = 3i^2{+}3i{+}1 \wedge z' = 6i{+}6 \\
Gik & = & x' = x{+}k^3{-}i^3 \wedge y' = y{+}3(k^2{-}i^2){+}3(k{-}i) \wedge z' = z{+}6(k{-}i)
\end{array}
$$

In those two examples at least, there is little to help us decide which rule is best.

## 13 Conclusions

Refinement semantics is an alternative to least fixpoint semantics that is more useful for programming. From it we derived a variety of rules for **while**-loops, **for**-loops, and loops with intermediate and deep exits. We illustrated the use of these rules with examples.

The difficulty of finding invariants is one of the deterrents to wider adoption of formal methods. Invariants are a vestige of the earliest work on loop rules, which used two one-state expressions. The invariant rules are entirely superseded by simpler, more general, easier-to-use rules.

The variant, used to prove loop termination, is entirely superseded by the more general, easier-to-use time variable. A variant is equivalent to the special case of a time variable that counts loop iterations. With a time variable, we can measure time any way we want, including real time, and no special rule is required to prove time bounds.

Least fixpoint semantics quantifies over specifications, and so it is second order. Refinement semantics is absolutely first order. It achieves this by treating loop constructs as second-class citizens; they are merely a "syntactic sugar" for a recursive refinement. Whether by least fixpoint or refinement semantics, loop constructs are given meaning by translation to a recursive form. If we use formal methods for programming, it is easier to refine to the recursive form than to the loop constructs; a compiler can then compile the recursive form to an efficient machine code with branching. It is therefore appropriate to treat loop constructs as second-class: they are neither necessary nor convenient.

## Acknowledgments

We thank Victor Kwan, Emil Sekerinski, and Michael Butler for substantive contributions to this paper. The first author thanks IFIP Working Groups 2.1 and 2.3 for being his research fora, and the University of Southampton for support and hospitality during the writing of this paper.

## References

1. E.W.Dijkstra: *a Discipline of Programming*, Prentice-Hall, New Jersey, 1976
2. E.W.Dijkstra, A.J.M.vanGasteren: "a Simple Fixpoint Argument without the Restriction to Continuity", *Acta Informatica* v.13 p.1-7, 1986
3. A.M.Gravell: "Simpler Laws for the Introduction of Loops", ECS, University of Southampton, 1996
4. E.C.R.Hehner: *a Practical Theory of Programming*, Springer-Verlag, New York, 1993
5. C.A.R.Hoare: "an Axiomatic Basis for Computer Programming", *CACM* 12(10), 1969
6. C.A.R.Hoare: "a Note on the **for** statement", *BIT* v.12 n.3 p.334-341, 1972
7. C.A.R.Hoare, J.He: "the Weakest Prespecification", *Fundamenta Informaticae* v.9 p.51-84, 217-252, 1986
8. C.C.Morgan: *Programming from Specifications*, second edition, Prentice-Hall, London, 1994
9. T.S.Norvell: "Predicative Semantics of Loops", *Algorithmic Languages and Calculi*, Chapman-Hall, 1997