Procedures and invariants in the refinement calculus

Trevor Vickers* and Carroll Morgan†

May 1994

Abstract

Invariants allow a rigorous treatment of types as sets in the refinement calculus, a method for developing imperative programs. The interaction of procedures and invariants is explored, resulting in a practical formalisation of existing programming practice.

1 Introduction

The notion of *local invariants* [9] was introduced to give rigorous treatment to types in the refinement calculus [7, 6, 8, 10]. Typing is a special kind of invariant. For example, in the scope of the declaration $n : \mathbb{N}$, which introduces a new local variable n of type $\mathbb{N}$ (the natural numbers), the invariant is $n \in \mathbb{N}$, and all commands preserve it.

The exploration [9] of the interaction between invariants and statements of a simple language—Dijkstra's language [2] with extensions—considered only language constructs including assignment, iteration, selection and recursion. We extend that work by examining a more complex language structure: the procedure.

Although this paper deals only with parameterless procedures, the results presented here can easily be applied to procedures with parameters. The formalization of procedures with parameters is presented in [5].

*Department of Computer Science, The Australian National University, Canberra 0200, Australia.
†Programming Research Group, Oxford University, 8–11 Keble Road, Oxford OX1 3QD, UK.

2 Invariant semantics

For completeness we summarise the semantics of invariants for the extension of Dijkstra's language. The meaning of a construct is given relative to an invariant (called the *context*). We write $wp_I(P, \phi)$ for the weakest precondition in context I of a program P with respect to a postcondition ϕ. For most constructs the invariant semantics is equivalent to the usual (non-invariant) semantics if the context is taken as *true*.

The usual language of the refinement calculus was extended by the inclusion of a local invariant block, and appropriate semantics given. In the invariant block $|[\ \mathbf{inv}\ J \bullet P\ ]|$, the new context J is assumed initially, is maintained by every command in P, and therefore is established finally.

3 Environment semantics

We extend the invariant semantics, now giving the meaning of our language relative to an environment. We write $wp_{I,\rho}(P, \phi)$ for the weakest precondition in context I and environment ρ of a program P to establish the postcondition ϕ. The environment will be used to associate a procedure name with the meaning of its declaration. For the language constructs already discussed the environment semantics are the obvious extension of the invariant semantics given, and are presented in Figure 1.

4 Refinement

We extend slightly the definitions of [9] to account for environments. The refinement relation $\sqsubseteq$ holds between programs P and Q if Q satisfies every specification that P does, in every context and environment.

Definition 1 *Refinement.* For programs P and Q, $P \sqsubseteq Q$ iff for all contexts I, environments ρ, and postconditions ϕ,

$$wp_{I,\rho}(P, \phi) \quad \Rrightarrow \quad wp_{I,\rho}(Q, \phi).$$

♡

For program fragments contained within the scope of a local invariant I, we can take advantage of the invariant by exploiting a weaker refinement relation $\sqsubseteq_I$, defined as follows.

Definition 2 *Refinement in context.* For programs P and Q,

$$P \sqsubseteq_I Q \quad \widehat{=} \quad |[\ \mathbf{inv}\ I \bullet P\ ]| \quad \sqsubseteq \quad |[\ \mathbf{inv}\ I \bullet Q\ ]|\ .$$

♡

$$
\begin{array}{rcl}
wp_{I,\rho}(\mathbf{skip}, \phi) & \widehat{=} & I \wedge \phi \\
wp_{I,\rho}(\mathbf{abort}, \phi) & \widehat{=} & \mathit{false} \\
wp_{I,\rho}(x := E, \phi) & \widehat{=} & I \wedge (I \Rightarrow \phi)[x \backslash E] \\
wp_{I,\rho}(P;\ Q, \phi) & \widehat{=} & wp_{I,\rho}(P, wp_{I,\rho}(Q, \phi)) \\
wp_{I,\rho}(\mathbf{if}\ (\,[\!]\, i \bullet G_i \rightarrow P_i)\ \mathbf{fi}, \phi) & \widehat{=} & (\vee i \bullet G_i) \wedge \\
 & & (\wedge i \bullet G_i \Rightarrow wp_{I,\rho}(P_i, \phi)) \\
wp_{I,\rho}(w : [post], \phi) & \widehat{=} & I \wedge (\forall\, w \bullet I \wedge post \Rightarrow \phi) \\
wp_{I,\rho}(\{pre\}\ , \phi) & \widehat{=} & pre \wedge I \wedge \phi \\
wp_{I,\rho}(|[\ \mathbf{var}\ x \bullet P\ ]|, \phi) & \widehat{=} & (\forall\, x \bullet wp_{I,\rho}(P, \phi))\ \ \dagger \\
wp_{I,\rho}(|[\ \mathbf{con}\ x \bullet P\ ]|, \phi) & \widehat{=} & (\exists\, x \bullet wp_{I,\rho}(P, \phi))\ \ \dagger \\
wp_{I,\rho}(|[\ \mathbf{inv}\ J \bullet P\ ]|, \phi) & \widehat{=} & wp_{I \wedge J}(P, \phi) \\
wp_{I,\rho}(\mathbf{mu}\ p \bullet \mathcal{C}(p)\ \mathbf{um}, \phi) & \widehat{=} & \mathit{fix}\ \mathcal{C}\ I\ p\ \phi\ \ \ddagger
\end{array}
$$

The substitution $[x \backslash E]$ replaces all free occurrences of x by E.
Iteration $\mathbf{do} \ldots \mathbf{od}$ is a special case of recursion ($\ddagger$: see [9], Section 8).

$\dagger$: x not free in I, ρ or ϕ.

Figure 1: Environment semantics for Dijkstra's language with extensions

Of more practical use is the following lemma.

Lemma 1 *Refinement in context.* For programs P and Q, we have $P \sqsubseteq_I Q$ iff for all postconditions ϕ and environments ρ, and stronger contexts J with $J \Rightarrow I$,

$$wp_{J,\rho}(P, \phi) \quad \Rrightarrow \quad wp_{J,\rho}(Q, \phi).$$

Proof: From Definition 1, Definition 2 and the semantics in Figure 1, $P \sqsubseteq_I Q$ iff for all postconditions ϕ, contexts K and environments ρ,

$$wp_{I \wedge K,\rho}(P, \phi) \quad \Rrightarrow \quad wp_{I \wedge K,\rho}(Q, \phi).$$

But the set of contexts "$I \wedge K$ for all K" is exactly the set of contexts "J with $J \Rightarrow K$", as required.
♡

An immediate consequence of Lemma 1 is that strengthening the context cannot invalidate a refinement.

Lemma 2 *Strengthen context.* If $P \sqsubseteq_I Q$ and $J \Rightarrow I$, then also $P \sqsubseteq_J Q$.

Proof: Trivial, since Lemma 1 treats all stronger J than I.
♡

5 Procedures

Procedures are simply a way of naming program text so that it may be referenced later. Like other language constructs in a context J, the truth of J is assumed beforehand, and must be established on completion. But there is a further issue with procedures since the program state is 'visible' during the call: must the context hold at every step of the procedure? We choose not, as the following definitions show. Section 5.7 motivates our choice and discusses alternatives.

5.1 Semantics

Like all declared objects, a procedure is introduced in a block. We write

$$|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|$$

to bind the name P to the program B; the scope of the binding is the program Q.

The semantics capture the behaviour we wish procedure declarations and their calls to exhibit. That is, the context of a procedure call may be assumed and must be established by the call, but need not be maintained during the call.

We use the symbol $\oplus$ for functional overriding, and $N \mapsto P$ to bind N to P. In the definition of procedure declaration, note that the declaring invariant is used when the procedure is called.

Definition 3 *Procedure declaration.*

$$wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|, \phi) \quad \widehat{=} \quad wp_{I,\rho'}(Q, \phi)$$

$$\text{where } \rho' = \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(B, \phi)\}$$

$\heartsuit$

The call of a procedure named P is written P. For a procedure call to establish ϕ in a context I requires that I hold initially and that the procedure body establish ϕ when I holds finally.

Definition 4 *Procedure call.* For all procedure names P,

$$wp_{I,\rho}(P, \phi) \quad \widehat{=} \quad I \wedge \rho(P)(I \Rightarrow \phi).$$

$\heartsuit$

5.2 A small example

We can gain some intuition for the above declarations from the following small example. We define our example program:

$$\begin{aligned} Ex \quad \widehat{=} \quad & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} S;\ T \bullet \\ & \qquad |[\ \textbf{inv}\ J \bullet P\ ]| \\ &]|\ . \end{aligned} \tag{1}$$

Using the above definitions, and the environment semantics of the constructs used,

$$wp_{I,\rho}(Ex, \phi) \quad \equiv \quad J \wedge wp_{I,\rho}(S, wp_{I,\rho}(T, J \Rightarrow \phi)).$$

If J holds initially, the procedure body establishes that ϕ holds when J holds finally. Note that neither S nor T maintain the context J.

5.3 Consequences of Definitions 3 and 4

An unused binding in the environment can be ignored.

Lemma 3 For all programs S, procedure names P, environments ρ, postconditions ϕ, and environment terms δ, if P is not used in program S, then

$$wp_{I,\rho\oplus\{P\mapsto\delta\}}(S,\phi) \quad \equiv \quad wp_{I,\rho}(S,\phi).$$

Proof: Structural induction over S.
♡

Definition 5 $\rho \triangleright \mu$. If ρ and μ are environments binding the same procedure names, then $\rho \triangleright \mu$ iff for all postconditions ϕ and procedure names P for which each environment holds a binding,

$$\rho(P)\ \phi \quad \Rrightarrow \quad \mu(P)\ \phi.$$

♡

Lemma 4 *Monotonicity of environments.* If ρ and μ are environments and $\rho \triangleright \mu$ then for all programs Q, contexts I and postconditions ϕ,

$$wp_{I,\rho}(Q,\phi) \quad \Rrightarrow \quad wp_{I,\mu}(Q,\phi).$$

Proof: Structural induction over Q.
♡

The crucial lemmas from the refinement calculus with invariants are upgraded to the refinement calculus with environments. They justify our choice of semantics in Figure 1 and Section 5.1.

Lemma 5 *Assume invariant.* No program P, in context I and environment ρ, is guaranteed to terminate unless I holds initially:

$$wp_{I,\rho}(P, true) \quad \Rrightarrow \quad I.$$

Proof: Structural induction over P.
♡

Lemma 6 *Establish invariant.* Any program P, in context I and environment ρ, establishes I iff it establishes anything:

$$wp_{I,\rho}(P,\phi) \quad \equiv \quad wp_{I,\rho}(P, I \wedge \phi).$$

Proof: Structural induction over P.
♡

The *stepwise refinement* of a program in context is justified by the following theorem.

Theorem 1 *Monotonicity.* Let $\mathcal{C}$ be a program scheme containing the program name p, and let $\mathcal{C}(X)$ be the result of replacing all occurrences of p in $\mathcal{C}$ by the program X. Then for programs P and Q, if $P \sqsubseteq_I Q$, then also $\mathcal{C}(P) \sqsubseteq_I \mathcal{C}(Q)$.

Proof: Structural induction over $\mathcal{C}$. The novel cases are procedure bodies and scopes, which we treat together. Let $\mathcal{D}$ and $\mathcal{S}$ be program schemes as described above.

Assume $P \sqsubseteq_I Q$ and $J \Rrightarrow I$.

$wp_{J,\rho}(|[\ \mathbf{procedure}\ N \mathrel{\widehat{=}} \mathcal{D}(P) \bullet \mathcal{S}(P)\ ]|, \phi)$

$\equiv$ $wp_{J,\rho \oplus \{N \mapsto \lambda\, \phi \bullet wp_{I,\rho}(\mathcal{D}(P), \phi)\}}(\mathcal{S}(P), \phi)$

$\Rrightarrow$ "Induction Hypothesis twice; Lemma 4"

$wp_{J,\rho \oplus \{N \mapsto \lambda\, \phi \bullet wp_{I,\rho}(\mathcal{D}(Q), \phi)\}}(\mathcal{S}(Q), \phi)$

$\equiv$ $wp_{J,\rho}(|[\ \mathbf{procedure}\ N \mathrel{\widehat{=}} \mathcal{D}(Q) \bullet \mathcal{S}(Q)\ ]|, \phi)$

$\heartsuit$

Theorem 2 *Use local invariant.* With $\mathcal{C}$ as before, if $P \sqsubseteq_{I \wedge J} Q$, then

$$|[\ \mathbf{inv}\ J \bullet \mathcal{C}(P)\ ]| \ \sqsubseteq_I \ |[\ \mathbf{inv}\ J \bullet \mathcal{C}(Q)\ ]|\ .$$

Proof: From Theorem 1, $\mathcal{C}(P) \sqsubseteq_{I \wedge J} \mathcal{C}(Q)$. The result follows from Definition 2 and the semantics in Figure 1.

$\heartsuit$

Lemma 7 For all contexts I, environments ρ, programs B, and postconditions ϕ,

$$wp_{I,\rho}(B, \phi) \quad \equiv \quad I \wedge wp_{I,\rho}(B, I \Rightarrow \phi).$$

Proof:

$wp_{I,\rho}(B, \phi)$

$\equiv$ "Lemma 6"

$wp_{I,\rho}(B, I \wedge \phi)$

$\equiv$ $wp_{I,\rho}(B, I \wedge (I \Rightarrow \phi))$

$\equiv$ "Lemma 6"

$wp_{I,\rho}(B, I \Rightarrow \phi)$

$\equiv$ "Lemma 5, Predicate Calculus"

$I \wedge wp_{I,\rho}(B, I \Rightarrow \phi)$

$\heartsuit$

5.4 Development method

As is standard practice when using the refinement calculus we define as our "programming" language all the constructions in Figure 1 (an extension of Dijkstra's language) and those of Section 5.1. That includes abstract programs, like $x : [x > y]$, and it of course admits "ordinary" programs too, like $x := y + 1$ (which refines the abstract example).

The aim of development is to refine a given program into a subset of the language, called *code*, which can be executed automatically by computer. For that we use refinement laws; we need not use *wp* directly—it is used only to prove the refinement laws. The following table distinguishes those constructs which are code (on the left) from those which are not (on the right).

$\mathbf{skip}$	$w : [post]$
$\mathbf{abort}$	$\{pre\}$
$x := E$	$\|[\ \mathbf{con}\ x \bullet S\ ]\|$
$S;\ T$	$\|[\ \mathbf{inv}\ J \bullet S\ ]\|$
$\mathbf{if}\ ..\ \mathbf{fi}$	
$\mathbf{mu}\ ..\ \mathbf{um}$	
$\|[\ \mathbf{var}\ x \bullet S\ ]\|$	
$\|[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]\|$	
P	

The following are standard examples of refinement laws. The third, though less common, is used in later proofs.

Law 1 *Strengthen postcondition.* If $(I \wedge post') \Rrightarrow post$,

$$w : [post] \sqsubseteq_I w : [post'].$$

♡

Law 2 *Assignment.* If $(I \wedge pre) \Rrightarrow post[w \backslash E]$,

$$\{pre\}\ \ w, x : [post] \sqsubseteq_I \{pre\}\ \ w := E.$$

♡

Law 3 **con** *introduction.* If c does not occur in S,

$$S \sqsubseteq\ |[\ \mathbf{con}\ c \bullet S\ ]|\ .$$

♡

5.4.1 Introduction and use

Procedures may be introduced in one of two ways. In the first, one simply defines a procedure to use it later.

Law 4 *Procedure introduction.* If P is not free in S, then

$$S \;=\; |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet S\ ]|\ .$$

♡

A procedure also may be introduced to name existing text, in which case the text is replaced by a call on the procedure.

Law 5 *Procedure introduction.*

$$B \;=\; |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet P\ ]|$$

♡

Note that an occurrence of a call to a procedure P in B above refers to an already existing procedure. That reference is not changed by the introduction of the local procedure P. A recursive procedure (or more precisely a procedure with a recursive body) is defined using **mu** ... **um** [9].

In the scope of a procedure declaration, occurrences of the procedure body may be replaced by a call on the procedure.

Law 6 *Use of procedure.* Let $\mathcal{C}$ be a program scheme containing the program name p, and let $\mathcal{C}(X)$ be the result of replacing all occurrences of p in $\mathcal{C}$ by the program X. If P does not occur free in B, then

$$|[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet \mathcal{C}(B)\ ]| \;=\; |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet \mathcal{C}(P)\ ]|\ .$$

Proof: Structural induction over $\mathcal{C}$.
♡

5.4.2 Distribution

One may wish a procedure to be available over a larger scope. In increasing the scope of a procedure one cannot simply 'move the procedure outwards' without regard to declarations already present. It is crucial, for example, that variables remain correctly bound. It is for this reason that we deal with increasing scope in two ways. In the first, the scope is increased over constructs other than program blocks. In the second, the scope is increased over each of the block constructs, with appropriate care to retain correct bindings.

Law 7 *Increase scope.* Let $\mathcal{C}$ be a program scheme containing the program name p, and let $\mathcal{C}(X)$ be the result of replacing all occurrences of p in $\mathcal{C}$ by the program X. If $\mathcal{C}$ does not contain blocks (**var**, **con**, **inv**, **procedure**) in whose scope p lies, then

$$\mathcal{C}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|) \quad = \quad |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet \mathcal{C}(Q)\ ]|\ .$$

Proof: Structural induction over $\mathcal{C}$.
♡

Part of the process of increasing a procedure's scope may be to pass the procedure declaration outwards through other blocks: **var**, **con**, **inv**, and **procedure**. The passage through each is exactly that required for the complementary distribution through a procedure declaration.

Law 8 **inv** *distribution.*

$$\begin{array}{lll} & |[\ \textbf{inv}\ J \bullet\ |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]| \\ = & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}}\ |[\ \textbf{inv}\ J \bullet B\ ]|\ \bullet \\ & \qquad\qquad |[\ \textbf{inv}\ J \bullet Q\ ]| \\ &]| \end{array}$$

Proof:

$$\begin{array}{ll} & wp_{I,\rho}(|[\ \textbf{inv}\ J \bullet\ |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]|, \phi) \\ \equiv & wp_{I \wedge J, \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I \wedge J, \rho}(B, \phi)\}}(Q, \phi) \\ \equiv & wp_{I, \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}([\![\ \textbf{inv}\ J \bullet B\ ]\!], \phi)\}}(|[\ \textbf{inv}\ J \bullet Q\ ]|, \phi) \\ \equiv & wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}}\ |[\ \textbf{inv}\ J \bullet B\ ]|\ \bullet\ |[\ \textbf{inv}\ J \bullet Q\ ]|, \phi) \end{array}$$
♡

Law 9 **var** *distribution.*

$$\begin{array}{lll} & |[\ \textbf{var}\ x \bullet\ |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]| \\ \sqsubseteq & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}}\ |[\ \textbf{con}\ x \bullet B\ ]|\ \bullet \\ & \qquad\qquad |[\ \textbf{var}\ x \bullet Q\ ]| \\ &]| \end{array}$$

Proof:

$$\begin{array}{ll} & wp_{I,\rho}(|[\ \textbf{var}\ x \bullet\ |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]|, \phi) \\ \equiv & \big(\forall\, x \bullet wp_{I, \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(B, \phi)\}}(Q, \phi)\big) \\ \Rrightarrow & \text{“Lemma 4, since } B \sqsubseteq\ |[\ \textbf{con}\ x \bullet B\ ]| \text{ (Law 3)”} \\ & \big(\forall\, x \bullet wp_{I, \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}([\![\ \textbf{con}\ x \bullet B\ ]\!], \phi)\}}(Q, \phi)\big) \\ \equiv & \text{“}x \text{ not free in } |[\ \textbf{con}\ x \bullet B\ ]|\text{”} \\ & wp_{I, \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}([\![\ \textbf{con}\ x \bullet B\ ]\!], \phi)\}}(|[\ \textbf{var}\ x \bullet Q\ ]|, \phi) \\ \equiv & wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}}\ |[\ \textbf{con}\ x \bullet B\ ]|\ \bullet\ |[\ \textbf{var}\ x \bullet Q\ ]|, \phi) \end{array}$$
♡

Note that the straightforward distribution of **var** through **procedure** would have the problem of scope: if the variable x occurs in the body B it will no longer be bound correctly after distribution. Of course if x does not occur free in B that problem is not present, but under that condition $|[\ \mathbf{con}\ x \bullet B\ ]| \sqsubseteq B$.

Law 10 con *distribution.*

$$\begin{array}{rl} & |[\ \mathbf{con}\ c \bullet\ |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]| \\ \sqsubseteq & |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}}\ |[\ \mathbf{con}\ c \bullet B\ ]|\ \bullet\ |[\ \mathbf{con}\ c \bullet Q\ ]|\ ]| \end{array}$$

Proof:

$wp_{I,\rho}(|[\ \mathbf{con}\ c \bullet\ |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]|, \phi)$

$\equiv$ $\big(\exists\, c \bullet wp_{I,\rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(B,\phi)\}}(Q,\phi)\big)$

$\Rrightarrow$ "Lemma 4, since $B \sqsubseteq\ |[\ \mathbf{con}\ c \bullet B\ ]|$ (Law 3)"

$\big(\exists\, c \bullet wp_{I,\rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}([\![\ \mathbf{con}\ c \bullet B\ ]\!],\phi)\}}(Q,\phi)\big)$

$\equiv$ $wp_{I,\rho}(|[\ \mathbf{procedure}\ P \mathrel{\widehat{=}}\ |[\ \mathbf{con}\ c \bullet B\ ]|\ \bullet\ |[\ \mathbf{con}\ c \bullet Q\ ]|, \phi)$

$\heartsuit$

Law 11 procedure *distribution.* If P and N are distinct procedure names, not used in programs B and D respectively, then

$$\begin{array}{rl} & |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet\ |[\ \mathbf{procedure}\ N \mathrel{\widehat{=}} D \bullet Q\ ]|\ ]| \\ = & |[\ \mathbf{procedure}\ N \mathrel{\widehat{=}} D \bullet\ |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]|\ . \end{array}$$

Proof:

$wp_{I,\rho}(|[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet\ |[\ \mathbf{procedure}\ N \mathrel{\widehat{=}} D \bullet Q\ ]|\ ]|, \phi)$

$\equiv$ "Let $\rho' = \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(B,\phi)\}$

and $\rho'' = \{N \mapsto \lambda\, \phi \bullet wp_{I,\rho'}(D,\phi)\}$"

$wp_{I,\rho \oplus \rho' \oplus \rho''}(Q,\phi)$

$\equiv$ "property of $\oplus$, P and N distinct"

$wp_{I,\rho \oplus \rho'' \oplus \rho'}(Q,\phi)$

$\equiv$ "Lemma 3 twice, P unused in D, N unused in B;

Let $\dot\rho = \{N \mapsto \lambda\, \phi \bullet wp_{I,\rho}(D,\phi)\}$

and $\ddot\rho = \{P \mapsto \lambda\, \phi \bullet wp_{I,\dot\rho}(B,\phi)\}$"

$wp_{I,\rho \oplus \ddot\rho \oplus \dot\rho}(Q,\phi)$

$\equiv$ $wp_{I,\rho}(|[\ \mathbf{procedure}\ N \mathrel{\widehat{=}} D \bullet\ |[\ \mathbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|\ ]|, \phi)$

$\heartsuit$

5.5 Invariant elimination

Explicit invariants must be removed from a program before it can be executed by computer. The following law complements those given in [9] by allowing **inv** to be eliminated when it surrounds a procedure call.

Law 12 *Invariant elimination from calls.* Let $\mathcal{C}$ be a program scheme containing the program name p, and let $\mathcal{C}(X)$ be the result of replacing all occurrences of p by the program X such that the only occurrences of X in $\mathcal{C}(X)$ are those where p occurred in $\mathcal{C}$. If p is not within the scope of other **inv** blocks or declarations of procedure P, and x is the list of identifiers declared in **con** and **var** blocks in whose scope p lies, then

$$\begin{array}{ll} & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet \mathcal{C}(|[\ \textbf{inv}\ J \bullet P\ ]|)\ ]| \\ \sqsubseteq & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} \{(\exists\, x \bullet J)\}\ |[\ \textbf{con}\ x \bullet B\ ]|\ [(\forall\, x \bullet J)] \bullet \mathcal{C}(P)\ ]|\ . \end{array}$$

Proof: Structural induction on $\mathcal{C}$. We present the base case only.

$$\begin{array}{ll} & wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet\ |[\ \textbf{inv}\ J \bullet P\ ]|\ ]|, \phi) \\ \equiv & wp_{I \wedge J, \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(B,\phi)\}}(P, \phi) \\ \equiv & I \wedge J \wedge wp_{I,\rho}(B, I \wedge J \Rightarrow \phi) \\ \equiv & \text{“Lemma 6”} \\ & I \wedge J \wedge wp_{I,\rho}(B, I \wedge (I \wedge J \Rightarrow \phi)) \\ \equiv & I \wedge wp_{I,\rho}(\{J\}\ B\ [J], I \Rightarrow \phi) \\ \equiv & wp_{I,\rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(\{J\}\ B\ [J], \phi)\}}(P, \phi) \\ \equiv & wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} \{J\}\ B\ [J] \bullet P\ ]|, \phi) \\ \heartsuit & \end{array}$$

Note that the invariant J can be broken by the procedure body, and that the invariant must be eliminated from all calls on P.

5.6 Copy rule

A common ability in programming languages is the replacement of a procedure call by the procedure's body. The following law defines the analogous operation in our development method.

Law 13 *Call expansion.* Let $\mathcal{C}$ be a program scheme containing the program name p, and let $\mathcal{C}(X)$ be the result of replacing all occurrences of p in $\mathcal{C}$ by the program X. If $\mathcal{C}$ does not contain **inv** blocks in whose scope p lies, and if P is not used in B, then

$$\begin{array}{ll} & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet \mathcal{C}(|[\ \textbf{inv}\ J \bullet P\ ]|)\ ]| \\ = & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet \mathcal{C}(\{J\}\ B\ [J])\ ]|\ . \end{array}$$

Proof: Structural induction on $\mathcal{C}$. We present the base case only.

$wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet\ |[\ \textbf{inv}\ J \bullet P\ ]|\ ]|, \phi)$

$\equiv$ "Let $\rho' = \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(B, \phi)\}$"

$wp_{I \wedge J, \rho'}(P, \phi)$

$\equiv$ $I \wedge J \wedge wp_{I,\rho}(B, I \wedge J \Rightarrow \phi)$

$\equiv$ "Lemma 3, Lemma 6"

$I \wedge J \wedge wp_{I,\rho'}(B, I \wedge (I \wedge J \Rightarrow \phi))$

$\equiv$ $I \wedge wp_{I,\rho'}(\{J\}B[J], I \Rightarrow \phi)$

$\equiv$ "Lemma 7"

$wp_{I,\rho'}(\{J\}B[J], \phi)$

$\equiv$ $wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet \{J\}\ B\ [J]\ ]|, \phi)$

$\heartsuit$

5.7 Alternatives

The preceding semantics describe a procedure whose call may assume and must establish its context, but which is free to break that context during the call. One alternative approach is where the context must also be maintained during the call. A second alternative is where the context is neither maintained nor established by the call. We discuss these alternatives now, in comparison to the preceding presentation, which we refer to as Alternative 0.

5.7.1 First alternative

If the context of the call is maintained by the procedure body, the following semantics for procedure declaration and call are used:

Definition 6 *Procedure declaration.*

$$wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|, \phi) \quad \widehat{=} \quad wp_{I,\rho'}(Q, \phi)$$

where $\rho' = \rho \oplus \{P \mapsto \lambda\, I \bullet\ \lambda\, \phi \bullet wp_{I,\rho}(B, \phi)\}$

$\heartsuit$

Definition 7 *Procedure call.*

$$wp_{I,\rho}(P, \phi) \quad \widehat{=} \quad \rho(P)\ I\ \phi$$

$\heartsuit$

This is perhaps the expected approach for procedures in local invariants, as the context is maintained within the procedure call as well as before and after it. Results which hold for Alternative 0 also hold for this alternative, and a similar method of development can be constructed, with slight differences in the laws for **inv** distribution, invariant elimination from calls, and call expansion.

It is the invariant elimination from calls which is the obstacle under this alternative. There, a local invariant surrounding a procedure call is eliminated by moving it to surround the procedure body. For example,

$$\begin{array}{ll} & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet\ |[\ \textbf{inv}\ J \bullet P\ ]|\ ]| \\ \sqsubseteq & |[\ \textbf{procedure}\ P \mathrel{\widehat{=}}\ |[\ \textbf{inv}\ J \bullet B\ ]|\ \bullet P\ ]|\ . \end{array}$$

A difficulty of this approach, however, is evident when one wishes to call a procedure from within two different contexts. The procedure body must then maintain both contexts.

5.7.2 Second Alternative

The second alternative is the opposite of the first. The context of the procedure call is neither maintained nor established by the procedure body. The following semantics for procedure declaration and call are used:

Definition 8 *Procedure declaration.*

$$wp_{I,\rho}(|[\ \textbf{procedure}\ P \mathrel{\widehat{=}} B \bullet Q\ ]|, \phi) \quad \widehat{=} \quad wp_{I,\rho'}(Q, \phi)$$

where $\rho' = \rho \oplus \{P \mapsto \lambda\, \phi \bullet wp_{I,\rho}(B, \phi)\}$

$\heartsuit$

Definition 9 *Procedure call.*

$$wp_{I,\rho}(P, \phi) \quad \widehat{=} \quad \rho(P)\ \phi$$

$\heartsuit$

Under these semantics the example program (1) has the following meaning:

$$wp_{I,\rho}(Ex, \phi) \quad \equiv \quad wp_{I,\rho}(P, wp_{I,\rho}(Q, \phi)).$$

The context J is neither assumed nor established by the procedure body.

While most results of Alternative 0 hold for this alternative, Lemma 5 (*Assume invariant*) and Lemma 6 (*Establish invariant*) do not, making this approach difficult to justify.

6 Related work

The refinement calculus was first proposed by Back [1], and invented again by Morris [11] and by Morgan [6]. The latter work, at Oxford, was motivated by the search for a rigorous and practical means of developing programs from Z specifications [3, 13].

The roots of local invariants can be traced to the use of invariants in a Z schema. There, the invariant frequently carries type information, and is maintained automatically.

The work of Lamport and Schneider [4] was compared in our earlier paper [9] describing local invariants for simpler language constructs. Lamport and Schneider use Hoare's partial correctness semantics expressed within temporal logic, and do not allow specifications as program constructs. They touch on procedures only briefly, concentrating only on parameter passing mechanisms, avoiding the interaction of their constraints on the procedure construct.

Reynolds [12] introduces *general invariants* which are true within their scope, and which may be temporarily falsified. But his *specification logic* does not give them a meaning and, like Lamport and Schneider, he uses partial correctness.

7 Conclusions

The introduction of local invariants [9] gave the refinement calculus a mechanism to describe predicates which are maintained within a block. It gave programmer's the ability to factor details like feasibility, and hence type-checking: essential in a practical method. But it did not allow programmers to use one of the most powerful of all programming tools: the procedure.

This paper remedies that lack.

The interaction of local invariants with procedures involved several choices. We have chosen to result in a development method which best fits current programming practice.

Perhaps the most significant choice was whether the invariant should be maintained within the body of a procedure whose call lay within the invariant's scope. We chose not. Section 5.7.1 explains why.

Acknowledgement

We thank Paul Gardiner for his help on aspects of this work.

References

[1] R.-J.R. Back. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[2] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[3] I.J. Hayes. *Specification Case Studies*. Prentice-Hall, London, 1987.

[4] L. Lamport and F.B. Schneider. Constraints: a uniform approach to aliasing and typing. In *12th Symposium on Principles of Programming Languages*, pages 205–216, New Orleans, January 1985. ACM.

[5] C.C. Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17–28, 1988. Reprinted in [10].

[6] C.C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [10].

[7] C.C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[8] C.C. Morgan and K.A. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5), September 1987. Reprinted in [10].

[9] C.C. Morgan and T.N. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990. Reprinted in [10].

[10] C.C. Morgan and T.N. Vickers, editors. *On the Refinement Calculus*. Springer, 1994.

[11] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[12] J.C. Reynolds. *The Craft of Programming*. Prentice-Hall, London, 1981.

[13] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, London, 1989.