

SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-06

Expressing program developments
from the Refinement Calculus in
CARE

Peter Lindsay

March 1994

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

Expressing program developments from the Refinement Calculus in CARE

Peter Lindsay,
Software Verification Research Centre,
University of Queensland,
St Lucia, Queensland 4072, Australia

Abstract

This paper compares Morgan's Refinement Calculus (**RC**) with the CARE approach to program development by illustrating how RC developments map to CARE developments. This paper treats algorithm refinement only, pending further formalization of the CARE approach to data types and data refinement.

A number of RC refinement rules are shown to correspond to CARE templates. RC applicability conditions and CARE proof obligations are compared and shown to be almost identical, except that CARE does not currently check for termination of recursive calls. A proposal is made concerning the addition of "variant functions" to fragments whose bodies contain recursive calls, and the corresponding proof obligations are explored.

Contents

1	Introduction to the Refinement Calculus	3
1.1	RC programs and specification statements	3
1.2	RC constructs	3
1.3	RC refinement laws	4
2	Relationship to CARE	5
2.1	An example RC development	5
2.2	The same development in CARE	6
2.3	The RC applicability conditions	8
2.4	The corresponding CARE proof obligations	9
3	Mapping RC into CARE	11
3.1	Sequential composition	11
3.2	Weaken pre/strengthen post	12
3.3	Assignment	12
3.4	Iteration (single guard)	12
3.5	Alternation (binary split)	13
4	Digression: Termination of Recursion	14
5	Data Refinement	17
6	Other Approaches to Refinement	17

1 Introduction to the Refinement Calculus

The Refinement Calculus (**RC**) was invented and described independently by Ralph Back, Joseph Morris and Carroll Morgan in the 1980's. The version described by Morgan in [3] is fairly indicative of the other two approaches, at least for sequential implementation languages.

1.1 RC programs and specification statements

An **RC (abstract) program** is essentially an imperative program, written in a style and notation very similar to Dijkstra's Guarded Command Language, but with the addition of non-executable **specification statements** of the form

$$w : [pre, post]$$

Here w is a list of program variables—the specification statement's "frame"—and pre and $post$ are predicates, called its "precondition" and "postcondition", respectively.

A specification statement acts as a placeholder for a program fragment which acts on w in such a way that, provided pre is true beforehand, then $post$ will be true afterwards. For example, the specification statement $y : [0 \leq x \leq 9, y^2 = x]$ stands for a program which, provided $0 \leq x \leq 9$, returns the square root of x .

In a specification statement $w : [pre, post]$, the predicate pre can involve variables from w as well as any other program variables which are in scope at the time; $post$ can involve these as well as variables x_0 , where $x \in w$, standing for the value of x *before* the program fragment is executed.¹ Thus, for example, the specification statement $x : [x > 0, 0 \leq x < x_0]$ stands for a program which decrements the value of x .

Note that, in general, an RC program represents a whole set of executable (i.e. GCL) programs—namely, those that can be obtained by replacing specification statements by executable program fragments which satisfy the corresponding specifications.²

1.2 RC constructs

There follows a brief summary of the main programming constructs from RC:

local block: $\llbracket \text{var } x : T \bullet prog \rrbracket$,

sequencing: $\langle prog_1; prog_2 \rangle$

alternation: $\langle \text{if } G_1 \rightarrow prog_1 \rrbracket \dots \rrbracket G_n \rightarrow prog_n \text{ fi} \rangle$

Execution of this program fragment results in execution of one of the $prog_i$'s for which G_i is true, with the choice of i being nondeterministic. The program fragment aborts if no G_i is true.

¹Thus the RC notation is similar to VDM's use of a hooked variable \bar{x} , and contrasts with Z's use of a prime on the variable to stand for its value *after* execution.

²An infeasible specification has an empty set of executable programs.

iteration: ‘do $G_1 \rightarrow prog_1 \parallel \dots \parallel G_n \rightarrow prog_n$ od’

The loop continues as long as some G_i holds, executing the corresponding $prog_i$ (again, chosen nondeterministically when more than one G_i holds). The loop exits when no G_i holds.

assignment ‘ $w := E$ ’

Parallel assignment.

For example, a program $factorial(n)$ for calculating factorials (and storing the result in r) might be written as follows:

$$\begin{aligned} & \parallel \text{ var } i, r : \mathbb{N} \bullet \\ & \quad i, r := 0, 1; \\ & \quad \text{do } i \neq n \rightarrow i, r := i + 1, i * r \text{ od} \\ & \parallel \end{aligned}$$

1.3 RC refinement laws

RC program P is said to **refine** Q (written ‘ $Q \sqsubseteq P$ ’) if the set of executable programs corresponding to P is a *subset* (in a sense made precise in [3]) of those corresponding to Q . Top-down program design, or **program refinement** as it is usually known, is concerned with making design decisions which progressively narrow the set of possible implementations down to a single program.

The Refinement Calculus consists of laws about refinement of RC programs, such as the following:

Sequential composition ([3] p.39)

$$w : [pre, post] \sqsubseteq w : [pre, mid]; w : [mid, post]$$

In other words, one way of refining an RC program $w : [pre, post]$ is to split it into a sequential composition of two programs $P_1; P_2$ such that P_1 satisfies $w : [pre, mid]$ and P_2 satisfies $w : [mid, post]$, for some intermediate assertion mid .

Strengthen postcondition ([3] p.54)

$$\text{If } \forall w \bullet pre[w_0/w] \wedge post_2 \Rightarrow post_1 \text{ }^3 \text{ then } w : [pre, post_1] \sqsubseteq w : [pre, post_2].$$

In other words, strengthening the postcondition narrows the set of possible implementations.

Assignment ([3] p.54)

$$\text{If } \forall w, r, o \bullet w = w_0 \wedge pre \Rightarrow post[E/w] \text{ }^4 \text{ then } w, o : [pre, post] \sqsubseteq w := E.$$

For example, $x : [x > 0, 0 \leq x < x_0] \sqsubseteq x := x - 1$.

³Morgan’s notation for entailment ($A \Rightarrow B$ means that in every state, if A is true then B is true) has been expanded out as universal quantification over state variables in this paper.

⁴Here, w is the set of variables being assigned to, r stands for the read-only variables, and o stands for the other variables in the frame.

Iteration ([3] p.62)

$w : [I, I \wedge \neg (\bigvee_i G_i)] \sqsubseteq \mathbf{do} \llbracket i \bullet G_i \rightarrow w : [I \wedge G_i, I \wedge (0 \leq V < V_0)] \mathbf{od}$.
 In other words, given loop invariant I , variant function V , and programs P_i satisfying $w : [I \wedge G_i, I \wedge (0 \leq V < V_0)]$, then

$\mathbf{do} G_1 \rightarrow P_1 \llbracket \dots \llbracket G_n \rightarrow P_n \mathbf{od}$

satisfies $w : [I, I \wedge \neg (\bigvee_i G_i)]$.

For example, $\mathbf{do} x \neq 0 \rightarrow x := x - 1 \mathbf{od}$ satisfies $x : [x \geq 0, x = 0]$.

Note that some RC laws have associated **applicability conditions**: that is, conditions which must be established in order to justify a refinement step.

2 Relationship to CARE

We show how a “typical” RC program refinement might be paraphrased in the CARE style [1], and informally compare the proof obligations resulting from the two approaches.

2.1 An example RC development

Consider the following derivation in RC of a program for finding the *integer square root* of a natural number s (cf. [3] pp.70-72):

The program has specification

$$\begin{aligned} & \llbracket \mathbf{var} s, r : \mathbb{N} \bullet \\ & \quad r : [\text{true}, r^2 \leq s < (r+1)^2] \\ & \rrbracket \end{aligned} \tag{1}$$

where s is the input value and r is the result to be returned. The first step in a development of a program to satisfy this specification statement might be to initialize r to 0, introduce a new local variable q initialized to $s+1$, and then—keeping $r^2 \leq s < q^2$ invariant—to bring q and r progressively closer together until finally $q = r + 1$.

In RC, such a design could be expressed in two refinement steps as follows:

$$\begin{aligned} (1) \sqsubseteq & \llbracket \mathbf{var} q : \mathbb{N} \bullet \\ & \quad r, q := 0, s + 1; \\ & \quad q, r : [I, I \wedge q = r + 1] \\ & \rrbracket \end{aligned} \tag{2}$$

where $I == r^2 \leq s < q^2$.

$$\begin{aligned}
(2) \quad &\sqsubseteq \text{invariant } I; \text{ variant } q - r \\
&\text{do } r + 1 < q \rightarrow \\
&\quad q, r: [I \wedge r + 1 < q, I \wedge 0 \leq q - r < q_0 - r_0] \\
&\text{od}
\end{aligned} \tag{3}$$

The fact that these are valid refinement steps is established by combining appropriate instances of various RC laws (see Section 2.3 below).

The next step in the development might be to refine the body of the loop by choosing a point p somewhere between r and q and, by comparing the value of p^2 with s , adjusting the value of r or q to equal p appropriately. This could be expressed as follows:

$$\begin{aligned}
(3) \quad &\sqsubseteq \llbracket \text{var } p : \mathbb{N} \bullet \\
&\quad p: [r + 1 < q, r < p < q]; \\
&\quad q, r: [I \wedge r < p < q, I \wedge 0 \leq q - r < q_0 - r_0] \\
&\rrbracket
\end{aligned} \tag{4}$$

$$(5) \quad \sqsubseteq \text{if } s < p^2 \rightarrow q := p \parallel s \geq p^2 \rightarrow r := p \text{ fi}$$

The final step in the development is to choose a value for p such that $r < p < q$. Let us simply take the midpoint of r and q :

$$(4) \quad \sqsubseteq p := (q + r) \text{ div } 2$$

Note how the Refinement Calculus has been used above to refine an abstract program to a concrete program, delaying design decisions (such as the choice of a value for p) until late in the development.

2.2 The same development in CARE

Now let's look at how the same development might be done in the CARE approach:

The first fragment to be implemented would have header

$$\text{sqrt} (s : \mathbb{N}) \langle\langle \text{true} \rangle\rangle :: r : \mathbb{N}; \quad \langle\langle r^2 \leq s < (r + 1)^2 \rangle\rangle$$

It would be implemented as follows:

$$\begin{aligned}
&\text{sqrt} (s : \mathbb{N}) \langle\langle \text{true} \rangle\rangle :: r : \mathbb{N}; \quad \langle\langle r^2 \leq s < (r + 1)^2 \rangle\rangle \\
&\quad ::= \text{zero}, \text{incr}(s) :: r, q : \mathbb{N}; \\
&\quad \quad \text{loop}(s, q, r) :: q, r; \\
&\quad \quad r .
\end{aligned}$$

where

`zero () <<true>> 0`

`incr (n : ℕ) <<true>> n + 1`

are fragments one could expect to find in the library and

`loop (s, q, r : ℕ) <<I>> :: q', r' : ℕ; <<I' ∧ q' = r' + 1>>`

is a new fragment to be implemented, where $I' == (r')^2 \leq s < (q')^2$. (In other words, we introduce a looping fragment which preserves the invariant, and which eventually makes $q = r + 1$.)

The `loop` fragment would in turn be implemented as follows:

```
loop (s, q, r : ℕ) <<I>> :: q', r' : ℕ; <<I' ∧ q' = r' + 1>>
  := lessthan(incr(r), q) ? closeGap(s, q, r) :: q, r;
    loop(s, q, r)
    ! q, r .
```

where

`lessthan (x, y : ℕ) <<true>> ? <<x < y>> !`

is a library (branching) fragment and

`closeGap (s, q, r : ℕ) <<I ∧ r + 1 < q>> :: q', r' : ℕ; <<I' ∧ 0 ≤ q' - r' < q - r>>`

is a new fragment to be implemented. (As an aside, note how the variant function has been used in the postcondition of `closeGap`: see Section 3.4 below for more details.)

Next, `closeGap` would be implemented by

```
closeGap (s, q, r : ℕ) <<I ∧ r + 1 < q>> :: q', r' : ℕ; <<I' ∧ 0 ≤ q' - r' < q - r>>
  := chooseIntermed(q, r) :: p : ℕ;
    adjustBnds(s, p, q, r) .
```

where

`chooseIntermed (q, r : ℕ) <<r + 1 < q>> :: p : ℕ; <<r < p < q>>`

`adjustBnds (s, p, q, r : ℕ) <<I ∧ r < p < q>> :: q', r' : ℕ; <<I' ∧ 0 ≤ q' - r' < q - r>>`

The `adjustBnds` fragment would be implemented as follows:

```
adjustBnds (s, p, q, r : ℕ) <<I ∧ r < p < q>> :: q', r' : ℕ; <<I' ∧ 0 ≤ q' - r' < q - r>>
  := lessthan(s, square(p)) ? p, r
    ! q, p .
```

where

`square` $(n : \mathbb{N}) \langle\langle \text{true} \rangle\rangle n^2$

Finally `chooseIntermed` would be implemented by:

`chooseIntermed` $(q, r : \mathbb{N}) \langle\langle r + 1 < q \rangle\rangle :: p : \mathbb{N}; \langle\langle r < p < q \rangle\rangle$
`:=` `div2`(`add`(`q`,`r`)) .

where

`add` $(x, y : \mathbb{N}) \langle\langle \text{true} \rangle\rangle x + y$

`div2` $(n : \mathbb{N}) \langle\langle \text{true} \rangle\rangle n \text{ div } 2$

Note how closely the CARE development parallels the RC refinement: this correspondence will be made more precise below.

2.3 The RC applicability conditions

This section discusses some of the applicability conditions which would be associated with the RC refinement given above, and the following section looks at the corresponding CARE proof obligations.

Step (1) in the RC development can be justified by a combination of instances of the RC laws given in Section 1.3 and the following law:

Introduce local block ([3] p.32) If w and x are disjoint, and pre and $post$ do not involve x , then $w : [pre, post] \sqsubseteq \llbracket \mathbf{var} \ x : T \bullet w, x : [pre, post] \rrbracket$.

The individual steps are as follows:

$$r : [\text{true}, r^2 \leq s < (r + 1)^2] \tag{1}$$

$$(1) \sqsubseteq \text{intro local block} \\ \llbracket \mathbf{var} \ q : \mathbb{N} \bullet \\ \quad q, r : [\text{true}, r^2 \leq s < (r + 1)^2] \\ \rrbracket \tag{a}$$

$$(a) \sqsubseteq \text{strengthen post} \\ \{ \text{since } \forall q, r : \mathbb{N} \bullet I \wedge q = r + 1 \Rightarrow r^2 \leq s < (r + 1)^2 \} \\ q, r : [\text{true}, I \wedge q = r + 1]$$

$$\begin{aligned}
&\sqsubseteq \text{ sequential composition} \\
& q, r : \left[\text{true} , I \right]; \tag{b} \\
& q, r : \left[I , I \wedge q = r + 1 \right] \tag{2}
\end{aligned}$$

$$\begin{aligned}
\text{(b)} \quad &\sqsubseteq \text{ assignment} \\
& \{ \text{since } \forall q, r : \mathbb{N} \bullet \text{true} \Rightarrow I[0, s + 1/r, q] \} \\
& r, q := 0, s + 1
\end{aligned}$$

(Note that $I[0, s + 1/r, q] = 0^2 \leq s < (s + 1)^2$.)

Step (2) is established by using ‘Strengthen post’ and ‘Iteration’ as follows:

$$\begin{aligned}
\text{(2)} \quad &\sqsubseteq q, r : \left[I , I \wedge q = r + 1 \right] \\
&\sqsubseteq \text{strengthen post} \\
& \{ \text{since } \forall q, r : \mathbb{N} \bullet I \wedge \neg (r + 1 < q) \Rightarrow I \wedge q = r + 1 \} \\
& q, r : \left[I , I \wedge \neg (r + 1 < q) \right] \\
&\sqsubseteq \text{iteration: invariant } I; \text{ variant } q - r \\
& \text{do } r + 1 < q \rightarrow q, r : \left[I \wedge r + 1 < q , I \wedge 0 \leq q - r < q_0 - r_0 \right]
\end{aligned}$$

Step (3) follows from a combination of ‘Intro local block’, ‘Seq composition’, ‘Strengthen post’, ‘Contract frame’ ([3] p.55) and ‘Remove invariant’ ([3] p.68).⁵

Step (5) uses the following law:

Alternation ([3] p.48) If $\forall w \bullet \text{pre} \Rightarrow \bigvee_i G_i$ then

$$w : \left[\text{pre} , \text{post} \right] \sqsubseteq \text{if } \parallel \quad i \bullet G_i \rightarrow w : \left[G_i \wedge \text{pre} , \text{post} \right] \text{ fi}$$

This together with two applications of the ‘Assignment’ law yields the following applicability conditions for step (5):

$$\begin{aligned}
&\forall p, q, r : \mathbb{N} \bullet I \wedge r < p < q \Rightarrow s < p^2 \vee s \geq p^2 \\
&\forall p, q, r : \mathbb{N} \bullet I \wedge r < p < q \wedge s < p^2 \Rightarrow r^2 \leq s < p^2 \wedge 0 \leq p - r < q - r \\
&\forall p, q, r : \mathbb{N} \bullet I \wedge r < p < q \wedge s \geq p^2 \Rightarrow p^2 \leq s < q^2 \wedge 0 \leq q - p < q - r
\end{aligned}$$

Finally, step (4) follows from ‘Assignment’ since

$$\forall p, q, r : \mathbb{N} \bullet r + 1 < q \Rightarrow r < (q + r) \text{ div } 2 < q$$

2.4 The corresponding CARE proof obligations

The CARE process of proof obligation generation is outlined in [1] and described in detail in [2]. The CARE proof obligation corresponding to the `sqrt` fragment is

⁵The applicability of ‘Remove invariant’ follows from the fact that p is not free in I .

$$\begin{aligned} \forall q, r, s : \mathbb{N} \bullet r = 0 \wedge q = s + 1 \Rightarrow \\ I \wedge \forall u, v : \mathbb{N} \bullet v^2 \leq s < u^2 \wedge u = v + 1 \Rightarrow v^2 \leq s < (v + 1)^2 \end{aligned}$$

Note how similar in content this is to the RC applicability condition for step (1). The correspondence will be made more precise below.

Ignoring problems with recursion for the moment, the CARE proof obligation for the `loop` fragment has two parts, corresponding to the two paths through its body. The first, corresponding to the recursive call, is long and complicated to state, but almost trivial to prove:

$$\begin{aligned} \forall q, r, s : \mathbb{N} \bullet I \wedge r + 1 < q \Rightarrow (I \wedge r + 1 < q) \\ \wedge \forall u, v : \mathbb{N} \bullet v^2 \leq s < u^2 \wedge 0 \leq u - v < q - r \Rightarrow v^2 \leq s < u^2 \\ \wedge \forall x, y : \mathbb{N} \bullet y^2 \leq s < x^2 \wedge x = y + 1 \Rightarrow y^2 \leq s < x^2 \wedge x = y + 1 \end{aligned}$$

The second path generates the following proof obligation:

$$\forall q, r, s : \mathbb{N} \bullet I \wedge \neg (r + 1 < q) \Rightarrow I \wedge q = r + 1$$

Again, these are very similar to the RC applicability conditions for the corresponding refinement step. Note however that RC ensures termination of the recursion: see below for a discussion of how CARE could be extended to cover termination proof obligations.

The proof obligation for `closeGap` is straightforward (left as an exercise).

The proof obligations for `adjustBnds` are

$$\begin{aligned} \forall p, q, r, s : \mathbb{N} \bullet I \wedge r < p < q \wedge s < p^2 \Rightarrow r^2 \leq s < p^2 \wedge 0 \leq p - r < q - r \\ \forall p, q, r, s : \mathbb{N} \bullet I \wedge r < p < q \wedge \neg (s < p^2) \Rightarrow p^2 \leq s < q^2 \wedge 0 \leq q - p < q - r \end{aligned}$$

which again are very similar to the RC applicability conditions for the corresponding refinement step (although the RC alternation construct is a little more general than CARE's use of branching).

Finally, the proof obligation for `chooseIntermed` is

$$\forall q, r : \mathbb{N} \bullet r + 1 < q \Rightarrow r < (q + r) \text{ div } 2 < q$$

3 Mapping RC into CARE

A CARE **template** (or partial development) consists of one or more complete (but possibly parameterized) fragments together with a collection of fragment headers. In a CARE refinement using a template, the header of one of the complete fragments gets matched against (or unified with) the fragment header to be implemented and, after instantiating the template parameters appropriately, the incomplete fragments become the new fragments to be implemented. This process continues until all fragments have implementations.

It should be clear by now that for many—if not most—RC refinement laws it is possible to give a CARE template which expresses the same refinement and for which the proof obligations are very similar. Some examples are given below.

In what follows, suppose the program fragment to be refined has specification statement $w: [pre, post]$ and that r stands for the other variables in scope in the program in which the fragment appears. In CARE notation, this would correspond to being required to find an “implementation” for the following fragment header:

```
frag (w, r) <<pre>> :: w'; <<post'>>
```

where, for a predicate P , P' stands for P with w replaced by w' and w_0 replaced by w .

3.1 Sequential composition

RC law: ([3] p.39)

$$w: [pre, post] \sqsubseteq w: [pre, mid]; w: [mid, post]$$

The corresponding CARE template is:

```
frag (w, r) <<pre>> :: w'; <<post'>>
  ::= frag1::w:N;
     frag2(w, r) .
```

```
frag1 (w, r) <<pre>> :: w'; <<mid'>>
```

```
frag2 (w, r) <<mid>> :: w'; <<post'>>
```

The CARE proof obligation for `frag` is

$$\forall w, r \bullet (pre \Rightarrow pre \wedge \forall w' \bullet (mid' \Rightarrow mid' \wedge \forall w'' \bullet (post'' \Rightarrow post'')))$$

which simplifies to true.

3.2 Weaken pre/strengthen post

RC law: ([3] pp.8,54)

If $\forall w \bullet (pre1 \Rightarrow pre2) \wedge (pre1[w_0/w] \wedge post2 \Rightarrow post1)$ then

$$w : [pre1, post1] \sqsubseteq w : [pre2, post2]$$

The corresponding CARE template is

```
frag1 (w, r)  <<pre1>>  :: w';  <<post1'>>
  ::= frag2(w, r) .
```

```
frag2 (w, r)  <<pre2>>  :: w';  <<post2'>>
```

The CARE proof obligation is:

$$\forall w, r \bullet pre1 \Rightarrow (pre2 \wedge \forall w' \bullet post2' \Rightarrow post1')$$

which is equivalent to the RC applicability condition.

3.3 Assignment

RC law: ([3] p.54)

If $\forall w, r \bullet w = w_0 \wedge pre \Rightarrow post[E/w]$ then

$$w : [pre, post] \sqsubseteq w := E$$

The corresponding CARE template is simply

```
frag (w, r)  <<pre>>  :: w';  <<post'>>
  ::= E .
```

Its proof obligation is

$$\forall w, r \bullet pre \Rightarrow post'[E/w']$$

which is logically equivalent to the RC applicability condition.

3.4 Iteration (single guard)

RC law: ([3] p.208)

$$w : [I, I \wedge \neg G] \sqsubseteq \mathbf{do} G \rightarrow w : [I \wedge G, I \wedge 0 \leq V < V_0] \mathbf{od}$$

The corresponding CARE template is

```

loop (w, r)  ⟨⟨I⟩⟩  ::= w';  ⟨⟨I' ∧ ¬ G'⟩⟩
  ::= test(w, r) ? body(w, r) :: w;
                    loop(w, r)
                    ! w .

```

```

test (w, r) ⟨⟨I⟩⟩ ? ⟨⟨G⟩⟩ !

```

```

body (w, r) ⟨⟨I ∧ G⟩⟩ ::= w';  ⟨⟨I' ∧ 0 ≤ V' < V⟩⟩

```

The CARE proof obligation for path 1 in `loop` is

$$\begin{aligned}
& \forall w, r \bullet I \Rightarrow I \wedge \\
& \quad (G \Rightarrow I \wedge G \wedge \\
& \quad \quad \forall w' \bullet I' \wedge 0 \leq V' < V \Rightarrow I' \wedge \\
& \quad \quad \forall w'' \bullet I'' \wedge \neg G'' \Rightarrow I'' \wedge \neg G'')
\end{aligned}$$

which simplifies to true. The proof obligation for path 2 is

$$\forall w, r \bullet I \Rightarrow I \wedge (\neg G \Rightarrow I \wedge \neg G)$$

which again simplifies to true. The postcondition of body ensures termination of the recursion in `loop`: see below for more discussion.

3.5 Alternation (binary split)

RC law:

$$\begin{aligned}
& w : [pre, post] \\
& \sqsubseteq \text{if } G \rightarrow w : [pre \wedge G, post] \parallel \neg G \rightarrow w : [pre \wedge \neg G, post] \text{ fi}
\end{aligned}$$

The corresponding CARE template is

```

frag (w, r)  ⟨⟨pre⟩⟩  ::= w';  ⟨⟨post'⟩⟩
  ::= test(w, r) ? frag1(w, r)
                    ! frag2(w, r) .

```

```

test (w, r) ⟨⟨pre⟩⟩ ? ⟨⟨G⟩⟩ !

```

```

frag1 (w, r) ⟨⟨pre ∧ G⟩⟩ ::= w';  ⟨⟨post'⟩⟩

```

```

frag2 (w, r) ⟨⟨pre ∧ ¬ G⟩⟩ ::= w';  ⟨⟨post'⟩⟩

```

4 Digression: Termination of Recursion

This section proposes an extension of the CARE language and proof obligations to support proof of termination of fragments involving recursive calls to themselves. (Mutual recursion will be ignored for now.)

The proposal has two parts:

1. Add a **variant function** (a Z term in the input variables) to each fragment whose body involves a recursive call. There will be a proof obligation to show that the variant represents a partial function from input variables to natural numbers whose domain corresponds to the fragment's precondition.
2. Add a clause to the appropriate place in the proof obligation for each path which involves a recursive call, to show that the variant is strictly decreasing on recursive calls. Since it is bounded below by zero, this will ensure the recursion terminates.

Some examples are given below. In the examples, the variant is written between the header and the body of the fragment, using the keyword “variant”:

Example 1

```
factorial (n : ℕ)  ⟨⟨true⟩⟩ fact(n) variant: n
  := isZero(n) ? 1
                ! factorial(decr(n)) : m;
                times(m, n) .
```

where *fact* is the factorial function and

```
decr (n : ℕ)  ⟨⟨n ≠ 0⟩⟩ n - 1
```

```
times (x, y : ℕ)  ⟨⟨true⟩⟩ x * y
```

```
isZero (n : ℕ)  ⟨⟨true⟩⟩ ? ⟨⟨n = 0⟩⟩ !
```

There is a trivial p.o. to show that the variant is well-formed: $\forall n : \mathbb{N} \bullet n \in \mathbb{N}$. The p.o. for path 1 is also easy to discharge: $\forall n : \mathbb{N} \bullet n = 0 \Rightarrow 1 = \text{fact}(n)$. The revised p.o. for path 2 in `factorial` would be:

$$\forall n : \mathbb{N} \bullet \neg (n = 0) \Rightarrow \overbrace{n - 1 < n}^{new} \wedge \text{fact}(n - 1) * n = \text{fact}(n)$$

Example 2

```

concat (s1, s2 : seq X)  ⟨⟨true⟩⟩ s1 ∩ s2  variant: #s1
  ::= snoc(s1) ? s2
      ! ::h,t;
      concat(t,s2)::u;
      cons(h,u) .

```

where

```

cons (x : X, s : seq X)  ⟨⟨true⟩⟩ ⟨x⟩ ∩ s

```

```

snoc (s : seq X)  ⟨⟨true⟩⟩
  ?⟨⟨s = ⟨⟩⟩⟩
  ! s ::h : X, t : seq X; ⟨h⟩ ∩ t

```

The p.o. for the variant is to show that $\forall s1 : \text{seq } X \bullet \#s1 \in \mathbb{N}$. The p.o. for path 1 is $\forall s1 : \text{seq } X \bullet s1 = \langle \rangle \Rightarrow s2 = s1 \cap s2$. The revised p.o. for path 2 is

$$\begin{aligned}
& \forall s1 : \text{seq } X \bullet \neg (s1 = \langle \rangle) \Rightarrow \\
& \quad \forall h : X, t : \text{seq } X \bullet s1 = \langle h \rangle \cap t \Rightarrow \\
& \quad \underbrace{\#t < \#s1}_{\text{new}} \wedge \langle h \rangle \cap (t \cap s2) = s1 \cap s2
\end{aligned}$$

Example 3

Returning to the development of `sqrt` above, the revised version of `loop` would be:

```

loop (s, q, r : ℕ)  ⟨⟨I⟩⟩  ::q',r' : ℕ; ⟨⟨I' ∧ q' = r' + 1⟩⟩  variant: q - r
  ::= lessthan(incr(r), q) ? closeGap(s,q,r)::q,r;
      loop(s,q,r)
      ! q,r .

```

The p.o. for the variant is $\forall s, q, r : \mathbb{N} \bullet I \Rightarrow q - r \in \mathbb{N}$. The p.o. for path 1 is:

$$\begin{aligned}
& \forall q, r, s : \mathbb{N} \bullet I \wedge r + 1 < q \Rightarrow (I \wedge r + 1 < q) \\
& \quad \wedge \forall u, v : \mathbb{N} \bullet v^2 \leq s < u^2 \wedge 0 \leq u - v < q - r \Rightarrow \\
& \quad \underbrace{u - v < q - r}_{\text{new}} \wedge v^2 \leq s < u^2 \\
& \quad \wedge \forall x, y : \mathbb{N} \bullet y^2 \leq s < x^2 \wedge x = y + 1 \Rightarrow y^2 \leq s < x^2 \wedge x = y + 1
\end{aligned}$$

Example 4

Consider fragment header

```
bddDecr (m : ℕ) ⟨⟨m > 5⟩⟩ :: n : ℕ;   ⟨⟨0 < n < m⟩⟩
```

- One possible implementation is

```
bddDecr (m : ℕ) ⟨⟨m > 5⟩⟩ :: n : ℕ;   ⟨⟨0 < n < m⟩⟩
  ::= three .
```

The p.o. is $\forall m : \mathbb{N} \bullet m > 5 \Rightarrow 0 < 3 < m$.

- A possible recursive implementation is

```
bddDecr (m : ℕ) ⟨⟨m > 5⟩⟩ :: n : ℕ;   ⟨⟨0 < n < m⟩⟩
  ::= equal(m,six) ? four
                    ! bddDecr(decr(m)) .
```

with variant m .

The p.o. for the variant is $\forall m : \mathbb{N} \bullet m > 5 \Rightarrow m \in \mathbb{N}$. The p.o. for path 1 is $\forall m : \mathbb{N} \bullet m > 5 \wedge m = 6 \Rightarrow 0 < 4 < m$. The p.o. for path 2 is

$$\forall m : \mathbb{N} \bullet m > 5 \wedge \neg (m = 6) \Rightarrow \\ m - 1 < m \wedge \forall n : \mathbb{N} \bullet 0 < n < m - 1 \Rightarrow 0 < n < m$$

- The following “implementation” satisfies the p.o.’s from the original CARE approach (i.e. without the use of a variant function):

```
bddDecr (m : ℕ) ⟨⟨m > 5⟩⟩ :: n : ℕ;   ⟨⟨0 < n < m⟩⟩
  ::= bddDecr(incr(m)) : : k;
      lessthan(one,k) ? decr(k)
                        ! one .
```

but it is not possible to define a variant function $f(m)$ so that the following p.o.s are true:

$$\forall m : \mathbb{N} \bullet m > 5 \Rightarrow f(m) \in \mathbb{N}$$

$$\forall m : \mathbb{N} \bullet m > 5 \Rightarrow f(m+1) < f(m) \wedge \\ \forall k : \mathbb{N} \bullet 0 < k < m+1 \Rightarrow \\ (1 < k \Rightarrow 0 < k-1 < m) \wedge (\neg(1 < k) \Rightarrow 0 < 1 < m)$$

5 Data Refinement

The Refinement Calculus also has laws concerning data refinement which we expect to be able to emulate easily in CARE using user-defined types, however treatment of this topic must wait until the CARE approach has been defined more precisely.

6 Other Approaches to Refinement

VDM's notions of data reification and operation decomposition have been around for even longer than—but have now been largely superseded by—the Refinement Calculus. Refinement in Z is sometimes done (if at all) by mapping Z specifications into the Refinement Calculus.

Thus these preliminary results seem to indicate that the CARE approach covers most of the major elements of refinement of (sequential) imperative programs. A later paper will relate the CARE approach back to the Refinement Calculus.

References

- [1] K. Harwood, P. Lindsay, and R. Matthews. An approach to constructing verified software. In *Seventeenth Annual Computer Science Conference*, pages 777–786, University of Canterbury, Christchurch, January 1994.
- [2] D. Hemer and P. Lindsay. Top-level Specification for Proof Obligation Generation. October 1993.
- [3] C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.