A Formal Approach to Program Modification

Lindsay Groves
School of Mathematical and Computing Sciences
Victoria University of Wellington, New Zealand
lindsay@mcs.vuw.ac.nz

Abstract

This paper presents a systematic approach to implementing certain kinds of program modifications, in which (conceptually at least) the modification is implemented as a separate program and then integrated with the original program using semantically based transformations. This approach allows us to ensure that the required modification is implemented correctly and also allows us to explore different ways of implementing a given modification. The approach is illustrated informally using an example where the modification can be implemented in two distinct ways, and then formalised within the refinement calculus by defining a program conjunction operator whose properties justify the transformations required in the example.

1 Introduction

Computer programs undergo change for a variety of reasons, for example, because the requirements change, an incremental development strategy is followed, or modified versions are required for reuse. Research in program evolution has examined the management of change within the software life-cycle, and the development of tools and techniques to assist a programmer in understanding and restructuring a program prior to modifying it. But there has been little work on determining just how a program should be changed in order to implement a particular modification.

This paper presents a systematic approach to implementing certain kinds of program modifications, in which (conceptually at least) the modification is implemented as a separate program and then integrated with the original program using semantically based transformations. As well as allowing us to ensure that the required modification is implemented correctly, this approach also allows us to explore different ways of implementing a given modification and to explain how different implementations of the modification are obtained.

We begin, in Section 2, by considering an example from [11] in which a given modification can be implemented in two distinct ways, and illustrate informally how the two implementations can be obtained using our approach. In Section 3, we show how the required combination of two programs can be defined formally in terms of a program conjunction operator, similar to schema conjunction in Z, and give some of its properties — including the ones required to justify the transformations performed informally in Section 2. The formalisation is in terms of the refinement calculus [8]; details of the refinement calculus are presented as needed. In Section 4, we summarise our approach, and discuss the limitations of the approach and indicate how some of them can be addressed.

2 An Example

We illustrate our approach informally using an example which has been used by Siddiqi and Roast to illustrate their idea of viscosity (a measure of how resistant a program is to change) [11] and in earlier behavioural studies of how programmers modify programs [10]. The example illustrates the very common form of modification which involves adding an extra output to a program, and is interesting because the modification can be implemented in two quite distinct ways.

2.1 The initial program

The specification for the initial program is as follows:

> Given a list of digits, 1 and 2, terminated with 0. It is required to return: (*a*) the number of 1's in the list, and (*b*) the number of 2's in the list.

A typical implementation of this specification, using Dijkstra's guarded commands language [2], is shown in Figure 1. This program has essentially the same structure as the one given in [11], although they only give a pseudocode outline.

$$
\begin{array}{l}
[\![\ \textbf{var}\ a, b, x \bullet \\
\quad a, b := 0, 0\ \textbf{;} \\
\quad \textbf{read}\ x\ \textbf{;} \\
\quad \textbf{do}\ x \neq 0 \rightarrow \\
\quad\quad \textbf{if}\ x = 1 \rightarrow a := a{+}1 \\
\quad\quad [\!]\ x = 2 \rightarrow b := b{+}1 \\
\quad\quad \textbf{fi}\ \textbf{;} \\
\quad\quad \textbf{read}\ x \\
\quad \textbf{od}\ \textbf{;} \\
\quad \textbf{write}\ a, b \\
]\!]
\end{array}
$$

Figure 1. The initial program

2.2 The modification

We are now required to make the following modification:

> Return the additional output: (c) the length of the longest sequence of 2's.

In order to make such a modification, we need to consider how we could write code to implement the new requirement and how this code could be combined with the existing code. In considering how to implement the new requirement, we may look to the existing program for guidance; and we may modify the existing program in order to facilitate its integration with the new code. In order to make these steps clear, we will implement the new requirement in a separate program and then attempt to combine the two programs.

It is well known that the kind of problem embodied in this modification can be solved in two basic ways: one using one loop, and one using nested loops. Typical programs based on the two approaches are shown in Figure 2, where *d* is used to record the length of the current sequence of 2's.

The one loop version just treats the input as a flat sequence of digits, and requires a careful analysis of the possible sequences of actions to ensure that it is indeed correct.

The version with nested loops is obtained by taking a more abstract view of the input, and considering each sequence of 2's as a single entity. Although this version is textually longer than the previous one, it may be regarded as being conceptually simpler.

2.3 Combining the original and the modification

We now consider how to integrate each of the programs implementing the modification with the original program. Since what the original program and the modification do is almost independent, we can think of the resulting program as executing the two programs in parallel — the only complication is that they both read the same input. If we assume that the input is only to be traversed once, the two programs need to read at the same time. (Otherwise, we could store the input, and run the two programs in sequence.)

Rather than just forming a syntactic combination of the two programs without regard for their semantics, we will do this in a semantic way, using a program conjunction operator, $\curlywedge$. The program we want will be described as $P \curlywedge Q$, where P and Q are the two programs being combined — this can be thought of as describing a program that does what P does and what Q does. Since the program conjunction operator is not in our programming language, and cannot be implemented efficiently, we will proceed to remove it using a sequence of correctness preserving transformations. These transformations effectively combine programs with similar structure, pushing conjunctions into their components, until they can be easily removed, usually because they apply to atomic programs.

For now, we will rely upon an intuitive understanding of what program conjunction means to motivate the required transformations; later we will give a formal definition of $\curlywedge$ and give the properties that justify these transformations.

2.4 The one loop version

Since the one loop version of the modification has a similar structure to the original program, we anticipate that it will be relatively easy to combine with the original program, and indeed this is the case. The programs being combined (we'll call the original P and the modification Q) are both variable declaration blocks, so we need to merge them somehow. We might expect that merging two declaration blocks will give a program declaring all the variables of the two, and whose body is obtained by combining the bodies of the two programs, so it appears that we should be able to make the replacement:

$$
\begin{array}{l}
\quad [\![\textbf{var}\ a, b, x \bullet P_1\ ]\!] \curlywedge [\![\textbf{var}\ c, d, x \bullet Q_1\ ]\!] \\
\Longrightarrow \\
\quad [\![\textbf{var}\ a, b, c, d, x \bullet P_1 \curlywedge Q_1\ ]\!]
\end{array}
\tag{1}
$$

where P_1 and Q_1 are the bodies of the two declaration blocks.

Now, P_1 and Q_1 are both sequential compositions:

$$a, b := 0, 0\ \textbf{;}\ \textbf{read}\ x\ \textbf{;}\ P_2\ \textbf{;}\ \textbf{write}\ a, b$$

and

$$c, d := 0, 0\ \textbf{;}\ \textbf{read}\ x\ \textbf{;}\ Q_2\ \textbf{;}\ \textbf{write}\ c$$

where P_2 and Q_2 are the loops in each case.

$$
\begin{array}{l}
[\![\ \textbf{var}\ c, d, x\ \bullet \\
\quad c, d := 0, 0\ \textbf{;} \\
\quad \textbf{read}\ x\ \textbf{;} \\
\quad \textbf{do}\ x \neq 0 \rightarrow \\
\qquad \textbf{if}\ x = 1 \rightarrow d := 0 \\
\qquad [\!]\ x = 2 \rightarrow d := d{+}1\ \textbf{;}\ c := c\ \mathsf{max}\ d \\
\qquad \textbf{fi}\ \textbf{;} \\
\qquad \textbf{read}\ x \\
\quad \textbf{od}\ \textbf{;} \\
\quad \textbf{write}\ c \\
]\!]
\end{array}
\qquad
\begin{array}{l}
[\![\ \textbf{var}\ c, d, x\ \bullet \\
\quad c := 0\ \textbf{;} \\
\quad \textbf{read}\ x\ \textbf{;} \\
\quad \textbf{do}\ x \neq 0 \rightarrow \\
\qquad \textbf{if}\ x = 1 \rightarrow \textbf{read}\ x \\
\qquad [\!]\ x = 2 \rightarrow \\
\qquad\quad d := 0\ \textbf{;} \\
\qquad\quad \textbf{do}\ x = 2 \rightarrow \\
\qquad\qquad d := d{+}1\ \textbf{;} \\
\qquad\qquad \textbf{read}\ x \\
\qquad\quad \textbf{od}\ \textbf{;} \\
\qquad\quad c := c\ \mathsf{max}\ d \\
\qquad \textbf{fi} \\
\quad \textbf{od}\ \textbf{;} \\
\quad \textbf{write}\ c \\
]\!]
\end{array}
$$

Figure 2. Two implementations for the modification

We might expect to be able to combine these by merging the corresponding components, with the replacement:

$$
\begin{array}{ll}
 & (a, b := 0, 0\ \textbf{; read}\ x\ \textbf{;}\ P_2\ \textbf{; write}\ a, b) \curlywedge \\
 & (c, d := 0, 0\ \textbf{; read}\ x\ \textbf{;}\ Q_2\ \textbf{; write}\ c) \\
\Longrightarrow & \\
 & (a, b := 0, 0 \curlywedge c, d := 0, 0)\ \textbf{;} \\
 & (\textbf{read}\ x \curlywedge \textbf{read}\ x)\ \textbf{;} \\
 & (P_2 \curlywedge Q_2)\ \textbf{;} \\
 & (\textbf{write}\ a, b \curlywedge \textbf{write}\ c)
\end{array}
\tag{2}
$$

Combining two assignments should reduce to a single assignment:

$$
\begin{array}{ll}
 & a, b := 0, 0 \curlywedge c, d := 0, 0 \\
\Longrightarrow & \\
 & a, b, c, d := 0, 0, 0, 0
\end{array}
\tag{3}
$$

We want the two **read**s to read the same input, so we want to make the replacement:

$$
(\textbf{read}\ x \curlywedge \textbf{read}\ x) \Longrightarrow \textbf{read}\ x \tag{4}
$$

And we want the conjunction of two **write**s to write all of the results of both:

$$
(\textbf{write}\ a, b \curlywedge \textbf{write}\ c) \Longrightarrow \textbf{write}\ a, b, c \tag{5}
$$

We now consider how to combine the loops. Fortuitously, the loops in this example have the same guard, so we should be able to combine them into a single loop with that guard, whose body is the conjunction of the two loop bodies. Thus, writing P_3 and Q_3 for the two **if** statements, we make the replacement:

$$
\begin{array}{ll}
 & (\textbf{do}\ x \neq 0 \rightarrow P_3\ \textbf{; read}\ x\ \textbf{od}) \curlywedge \\
 & (\textbf{do}\ x \neq 0 \rightarrow Q_3\ \textbf{; read}\ x\ \textbf{od}) \\
\Longrightarrow & \\
 & \textbf{do}\ x \neq 0 \rightarrow (P_3\ \textbf{; read}\ x) \curlywedge (Q_3\ \textbf{; read}\ x)\ \textbf{od}
\end{array}
\tag{6}
$$

Again, we expect to combine the two sequences by merging corresponding elements, and merging the two **read**s produces a single **read**.

$$
\begin{array}{ll}
 & (P_3\ \textbf{; read}\ x) \curlywedge (Q_3\ \textbf{; read}\ x) \\
\Longrightarrow & \\
 & (P_3 \curlywedge Q_3)\ \textbf{; read}\ x
\end{array}
\tag{7}
$$

We now have to combine P_3 and Q_3, which are both **if** statements:

$$
\begin{array}{l}
\textbf{if}\ x = 1 \rightarrow a := a{+}1 \\
[\!]\ x = 2 \rightarrow b := b{+}1 \\
\textbf{fi}
\end{array}
$$

and

$$
\begin{array}{l}
\textbf{if}\ x = 1 \rightarrow d := 0 \\
[\!]\ x = 2 \rightarrow d := d{+}1\ \textbf{;}\ c := c\ \mathsf{max}\ d \\
\textbf{fi}
\end{array}
$$

Since they have the same set of guards, and the cases are mutually exclusive, it seems reasonable to expect that we can combine them to give a single **if** statement with the same set of guards, whose bodies are obtained by combining the relevant bodies from the two **if** statements. Those bodies are assignments, which we have seen reduce to a single assignment, provided we first turn $d := d+1$ **;** $c := c \max d$ into a multiple assignment.

$$
\begin{array}{l}
\quad \textbf{if}\, x = 1 \rightarrow a := a+1 \\
\quad [\!]\; x = 2 \rightarrow b := b+1 \\
\quad \textbf{fi} \\
\quad\; \curlywedge \\
\quad \textbf{if}\, x = 1 \rightarrow d := 0 \\
\quad [\!]\; x = 2 \rightarrow d := d+1 \,\textbf{;}\, c := c \max d \\
\quad \textbf{fi} \\
\Longrightarrow \hfill (8) \\
\quad \textbf{if}\, x = 1 \rightarrow (a := a+1 \curlywedge d := 0) \\
\quad [\!]\; x = 2 \rightarrow (b := b+1 \curlywedge d := d+1 \,\textbf{;}\, c := c \max d) \\
\quad \textbf{fi} \\
\Longrightarrow \hfill (9) \\
\quad \textbf{if}\, x = 1 \rightarrow a, d := a+1, 0 \\
\quad [\!]\; x = 2 \rightarrow b, c, d := b+1, c \max d+1, d+1 \\
\quad \textbf{fi}
\end{array}
$$

We can now assemble the resulting program; the result is shown in Figure 3 and is the same as the first style of solution given in [11].

$$
\begin{array}{l}
[\![\; \textbf{var}\; a, b, x \bullet \\
\quad a, b, c, d := 0, 0, 0, 0 \,\textbf{;} \\
\quad \textbf{read}\; x \,\textbf{;} \\
\quad \textbf{do}\; x \neq 0 \rightarrow \\
\qquad \textbf{if}\, x = 1 \rightarrow a, d := a+1, 0 \\
\qquad [\!]\; x = 2 \rightarrow b, c, d := b+1, c \max d+1, d+1 \\
\qquad \textbf{fi} \,\textbf{;} \\
\qquad \textbf{read}\; x \\
\quad \textbf{od} \,\textbf{;} \\
\quad \textbf{write}\; a, b \\
]\!]
\end{array}
$$

Figure 3. Result of using the one loop version of the modification

2.5 The nested loops version

Integrating the nested loops version of the modification is not so straightforward, because it can read several digits in each iteration of the outer loop, while the original program only reads one. In order to arrange that the programs read the input "in step", we need to modify the original program so that it has the same structure as the program implementing the modification; the result is shown on the left of Figure 4.

After restructuring the original program, it is now easy to integrate the nested loops version of the modification. There are only two transformations we haven't seen before, because the corresponding branches of the **if** statements do not have the same structure.

In the first branch, we have to sequence of two statements with a single statement:

$$
\begin{array}{l}
\quad (a := a+1 \,\textbf{; read}\; x) \curlywedge \textbf{read}\; x \\
\Longrightarrow \hfill (8) \\
\quad a := a+1 \,\textbf{; read}\; x
\end{array}
$$

and in the second branch, we have to combine a sequence of three statements with a single statement:

$$
\begin{array}{l}
\quad P' \curlywedge (d := 0 \,\textbf{;}\, Q' \,\textbf{;}\, c := c \max d) \\
\Longrightarrow \hfill (9) \\
\quad d := 0 \,\textbf{;}\, (P' \curlywedge Q') \,\textbf{;}\, c := c \max d
\end{array}
$$

where P' and Q' are both loops.

The resulting program is shown on the right of Figure 4, and is the same as the second style of solution given in [11].

3 Let's Get Formal

The sequence of steps performed above has been based largely on wishful thinking — which is unfortunately too common in the way that most programmers write their programs, and even more so in the way that they modify them. In fact, by comparison, the above is a very systematic process, since it allows us to clearly identify the properties of the conjunction operator upon which the validity of each step relies.

We will now show how to define a suitable program conjunction operator, and show that it has the properties required in the above derivation. We will also see what semantic relationships the $\Longrightarrow$ arrows represent.

First, we need to be able to formalise our specifications, which we do using *specification statements* [7]. A specification statement has the general form $w \,\textbf{:}\, \left[pre \;\middle/\; post \right]$. This specifies a program that will establish *post* (the *postcondition*), assuming that *pre* (the *precondition*) holds initially, changing only variables in *w* (the *frame*).

```
[[ var a, b, x •
   a, b := 0, 0 ;
   read x ;
   do x ≠ 0 →
      if x = 1 → a := a+1 ; read x
      [] x = 2 →
         do x = 2 →
            b := b+1 ;
            read x
         od
      fi
   od ;
   write a, b
]]
```

```
[[ var a, b, c, d, x •
   a, b, c := 0, 0, 0 ;
   read x ;
   do x ≠ 0 →
      if x = 1 → a := a+1 ; read x
      [] x = 2 →
         d := 0 ;
         do x = 2 →
            b, d := b+1, d+1 ;
            read x
         od ;
         c := c max d
      fi
   od ;
   write a, b, c
]]
```

Figure 4. Integrating the nested loops version of the modification

We can thus write the original specification (ignoring the requirement to print the results) as:

$$a, b \colon \left[in \in (1\,|\,2)^{*}0 \;\Big/\; a = 1\#in \wedge b = 2\#in \right]$$

where *in* denotes the value of the input, $in \in (1\,|\,2)^{*}0$ means that *in* is in the language defined by the regular expression $(1\,|\,2)^{*}0$, and $x\#in$ denotes the number of occurrences of x in *in*.

Similarly, we can express the required modification as:[1]

$$c \colon \left[in \in (1\,|\,2)^{*}0 \;\Big/\; c = maxseqlen(in, 2) \right]$$

where $maxseqlen(s, u) \;\widehat{=}\; \mathsf{max}\, \{i, j : \mathrm{dom}\, in \mid (\forall\, k \mid i \leq k \leq j \bullet s[k] = u) \bullet j - i + 1\}$.

Note that the specification does not say what should happen if the input does not conform to the precondition. By including the assumption $in \in (1\,|\,2)^{*}0$ in the specification, we are explicitly saying that we don't care what happens if this assumption does not hold. In fact, all of the programs shown above will fail to terminate if the given input contains anything other than 0, 1 and 2, or if there is no 0 in the input — they may loop indefinitely, or may terminate abnormally with an error message, according what happens when the program tries to read past the end of its input.

[1] Strictly speaking, we should include *in* in the frames of both specifications, since the input is modified as a side-effect of reading. For simplicity, we will omit it.

Next, we need to say what it means for a program to correctly implement a specification. For this we use the refinement calculus, in which specifications and programs are both expressed in a single, wide spectrum language. The semantics are defined in terms of weakest preconditions and correctness is formalised in terms of a refinement relation ($\sqsubseteq$) between arbitrary members of this language — thus, refinement can be used to describe transformations on specifications and on programs, as well as the traditional implementation relation between a specification and a program.

An important feature of the refinement calculus is the inclusion of programs **abort**, which does not establish any postcondition (i.e. is never guaranteed to terminate), and **magic**, which will establish any postcondition, even *false*, but can never be implemented (i.e. is infeasible). **abort** and **magic** are the bottom and top, respectively, of the refinement lattice, so every program is a refinement of **abort** and every program is refined by **magic**. For a gentle introduction to the refinement calculus, see [8]; for more detailed treatments, see [9, 1].

3.1 Defining program conjunction

We now consider how to define a program conjunction operator with the meaning and properties suggested above. For the purposes of this paper, we assume that all programs are conjunctive (i.e. $wp\,.\,S\,.\,(\phi \wedge \psi) \equiv wp\,.\,S\,.\,\phi \wedge wp\,.\,S\,.\,\psi$, for all predicates ϕ and ψ). This is a reasonable assumption, since the specifications and implementations we consider are both conjunctive, and simplifies the definition of conjunction.

We want to define $S \curlywedge T$ so that it behaves like S *and* like T, so long as both terminate. Thus, for S and T with the same frame, we want:

$$\{\mathsf{pre}\ T\}\ ;\ S \sqsubseteq S \curlywedge T \quad \text{and} \quad \{\mathsf{pre}\ S\}\ ;\ T \sqsubseteq S \curlywedge T$$

where $\{\mathsf{pre}\ S\}$ is the same as **skip** when S terminates and the same as **abort** otherwise.

We also want $S \curlywedge T$ to be able to modify any variables that S or T can modify. We achieve this by expanding the frame of each component to allow arbitrary assignments (written $x := ?$) to variables that are only in the frame of the other component. So we generalise the requirements above to:

$$\{\mathsf{pre}\ T\}\ ;\ S_{\Uparrow v-u} \sqsubseteq S \curlywedge T \quad \text{and}$$
$$\{\mathsf{pre}\ S\}\ ;\ T_{\Uparrow u-v} \sqsubseteq S \curlywedge T$$

where, $u = \mathit{frame}(S)$, $v = \mathit{frame}(T)$, and for $x \notin \mathit{frame}(U)$, $U_{\Uparrow x} = U\ ;\ x := ?$.

Finally, we want $S \curlywedge T$ to be the least refined program which satisfies the above constraints. Since we are assuming that all programs are conjunctive, we are guaranteed that for any S and T, there is a unique program statisfying this definition (for a more detailed account of the definition of program conjunction in the case where programs may not be conjunctive, see [6, 5]).

For specification statements, this definition gives:

$$\begin{array}{ll} & u\!:\!\left[P \,\middle/\, R\right] \curlywedge v\!:\!\left[P' \,\middle/\, R'\right] \\ = & u,v\!:\!\left[P \wedge P' \,\middle/\, R \wedge R'\right] \end{array}$$

Thus for our example, we have, as expected:

$$\begin{array}{ll} & a,b\!:\!\left[\mathit{in} \in (1|2)^*0 \,\middle/\, a = 1\#\mathit{in} \wedge b = 2\#\mathit{in}\right] \curlywedge \\ & c\!:\!\left[\mathit{in} \in (1|2)^*0 \,\middle/\, c = \mathit{maxseqlen}(\mathit{in}, 2)\right] \\ = & a,b,c\!:\!\left[\mathit{in} \in (1|2)^*0 \,\middle/\, \begin{array}{c} a = 1\#\mathit{in} \wedge b = 2\#\mathit{in} \\ c = \mathit{maxseqlen}(\mathit{in}, 2) \end{array}\right] \end{array}$$

3.2 Properties of program conjunction

Program conjunction has a number of properties that can be used to justify the kinds of transformations required in the above example. Here we state a few of the important properties, including the ones required above; proofs of these, and many others, can be found in [3]. Unless otherwise stated, these laws are for programs with identical frames. Programs with different frames can be combined by first exanding their frames as shown above.

A vital property is the fact that conjunction is monotonic with respect to refinement. This is important because it means that we can refine the components of a conjunction independently, which we have done throughout our examples, and obtain a refinement of the whole conjunction.

Law 1 $S \curlywedge T \sqsubseteq S' \curlywedge T'$ if $S \sqsubseteq S'$ and $T \sqsubseteq T'$

If S and T can't be jointly satisfied, the result is **magic**, which means that the required program cannot be implemented.

Program conjunction is idempotent, commutative and associative, and has **abort** as its zero. The arbitrary assignment $x := ?$ is the identity for programs with frame x. Idempotence is used in steps 4 and 7 in Section 2.4 to combine two **read** statements.

Law 2 $S \curlywedge S = S$

Law 3 $S \curlywedge T = T \curlywedge S$

Law 4 $(S \curlywedge T) \curlywedge U = S \curlywedge (T \curlywedge U)$

Law 5 $S \curlywedge$ **abort** $=$ **abort**

Law 6 $S \curlywedge w := ? = S$, where S has frame w

A conjunction of two assignments to different variables is the same as a multiple assignment (steps 3 and 9).

Law 7 $x := e \curlywedge y := f = x, y := e, f$, for distinct x and y

A conjunction of two assignments to the same variable is the same as a single assignment if the two variables assign the same value, and **magic** otherwise.

Law 8 $x := e \curlywedge x := f = \begin{cases} x\!:\!\left[e = f \,\middle/\, x = e\right] \\ x := e \text{ if } e = f \text{ for all states} \\ \textbf{magic} \text{ if } e \neq f \text{ for all states} \end{cases}$

A conjunction can be turned into a sequential composition, provided this introduces no interference — the easiest way to guarantee this is to require that the frame of the first program and the signature of the second (the set of variables it depends on or can modify) are disjoint.

Law 9 $S \curlywedge T \sqsubseteq S\ ;\ T$, if $\mathit{frame}(S) \cap \mathit{sig}(T) = \emptyset$

Program conjunction enjoys some very nice distribution properties, which do the work of pushing conjunctions inside structures.

We can move a conjunction into the body of a declaration block, provided the variables declared do not occur in the other component.

Law 10 $[\![\textbf{var}\ x \bullet S]\!] \curlywedge T \sqsubseteq [\![\textbf{var}\ x \bullet S \curlywedge T]\!]$, provided x doesn't occur in T.

We can also combine two declaration blocks to give a block declaring all of the variables declared by the components (step 1). If the way the components use a common variable is inconsistent, the resulting conjunction will be infeasible, in which case it will be necessary to rename one of the variables.

Law 11 $[\![\textbf{var}\ x \bullet S]\!] \curlywedge [\![\textbf{var}\ y \bullet T]\!] \sqsubseteq [\![\textbf{var}\ z \bullet S \curlywedge T]\!]$, where z contains all of the variables in x and in y.

A conjunction of two sequential compositions can be refined by forcing them to synchronise at their midpoint (steps 2 and 7).

Law 12 $(S_1 \ ; \ S_2) \curlywedge (T_1 \ ; \ T_2) \ \sqsubseteq \ (S_1 \curlywedge T_1) \ ; \ (S_2 \curlywedge T_2)$

In applying this law in step 2, we have to first unify the frames of the component statements. This means rewriting $a,b := 0,0$ and $c,d := 0,0$ as $a,b,x,c,d := 0,0,x,?,?$ and $a,b,x,c,d :=?,?,x,0,0$, which are combined in step 3 to give $a,b,x,c,d := 0,0,x,0,0$ and then simplified to omit the x. It also means treating the **read** statements as having frame a,b,x,c,d, but only being able to change x, and similarly with the **write** statements. Finally, we expand the frames of the statements in the two loops in a similar way, so they already have identical frames when we come to combine the corresponding statements in step 7.

A conjunction can be moved into the branches of an **if** statement.

Law 13 **if** $[\!]_i \ G_i \rightarrow S_i$ **fi** $\curlywedge T \ \sqsubseteq$ **if** $[\!]_i \ G_i \rightarrow S_i \curlywedge T$ **fi**

Two **if** statements can be combined by considering all possible combinations of guards that might be true. The result can be simplified if the sets of guards are the same or if they are mutually exclusive, or both (step 8).

Law 14 **if** $[\!]_i \ G_i \rightarrow S_i$ **fi** $\curlywedge$ **if** $[\!]_j \ H_j \rightarrow T_j$ **fi** $\sqsubseteq$
if $[\!]_{i,j} \ G_i \wedge H_j \rightarrow S_i \curlywedge T_j$ **fi**

A conjunction of two loops with the same guard is refined by a loop with the same guard whose body is the conjunction of the bodies of the components (step 6). If the way the two loops work is not compatible, the result will be infeasible.

Law 15 **do** $G \rightarrow S$ **od** $\curlywedge$ **do** $G \rightarrow T$ **od** $\sqsubseteq$
do $G \rightarrow S \curlywedge T$ **od**

We can also combine loops with different guards and different frames. The result is a program that effectively executes both loops until one of them terminates, and then continues to execute the one that hasn't terminated — when the frames are disjoint, this will not disrupt the effects of the first loop to terminate.

Law 16 **do** $G \rightarrow S$ **od** $\curlywedge$ **do** $H \rightarrow T$ **od** $\sqsubseteq$
do $G \wedge H \rightarrow (S \curlywedge T)$ **od** **;**
if $G \rightarrow$ **do** $G \rightarrow S$ **od**
$[\!]$ $H \rightarrow$ **do** $H \rightarrow T$ **od**
$[\!]$ $\neg G \wedge \neg H \rightarrow$ **skip**
fi

provided $frame(S) \cap frame(T) = \emptyset$.

To handle the two new steps in the nested loops version (Section 2.5), we need to be aware of the implicit frames of the programs being combined. We will only indicate briefly here how this is done; a more precise treatment is given in [3].

To combine $a := a+1$ **; read** x and **read** x (step 8), we first rewrite **read** x as **skip ; read** x, and use law 12 to transform $a := a+1$ **; read** $x \curlywedge$ **skip ; read** x into $a := a+1$ **; read** x. This involves unifying the frames as described above for step 2, and simplifying the result.

In combining P' and $d := 0$ **;** Q' **;** $c := c \max d$ (step 9), we decided that P' should be combined with Q', since they are both loops. So we rewrite P' as **skip ;** P' **; skip**, combine this with $d := 0$ **;** Q' **;** $c := c \max d$ using law 12, to give (**skip** $\curlywedge d := 0$) **;** $(P' \curlywedge Q')$ **;** (**skip** $\curlywedge c := c \max d$), and then remove the **skip**s again as shown above.

We do not provide a law for combining **write** statements, since that raises issues about the order of output which could invalidate our commutativity law (law 3). For the examples discussed here, it is sufficient to just combine the parts of the program computing the required outputs.

4 Discussion

We have defined a program conjunction operator and shown how it can be used to provide a systematic way of implementing program modifications. Our approach, as presented here, is to describe the required program as a conjunction of an existing program and a program implementing the additional requirement, and then combine the two programs so as to eliminate the conjunction. This is done using properties of program conjunction which have been proved to preserve correctness. The inclusion of **magic** in the language allows us to define a conjunction operator which is monotonic with respect to refinement (unlike schema conjunction in Z), and leads to considerable simplification in the rules. We usually don't need to check whether the two programs can be combined properly — if they can't, the result is **magic**, never an incorrect program!

There are two apparent drawbacks to the method we have described: the range of modifications that can be handled is rather limited, and explicitly constructing a separate program to implement the additional requirement before integrating it with the original program may be considered too much effort to be worthwhile. We will now consider these issues briefly.

The conjunction operator we have defined allows us to describe a wide range of modifications that involve adding extra outputs or adding extra constraints on inputs and/or outputs. While such modifications appear to be quite common, there are also other kinds of modifications we may wish to make. We can apply the same approach to a wider range of modifications by defining other operators with the appropriate meanings. For example, we can define an operator that will combine programs S and T to obtain a program that behaves like S for inputs satisfying its precondition and like T for inputs satisfying its precondition. In fact we can define two such operators according to what happens for inputs satisfying both preconditions: one that acts, nondeterministically, like either S or T in this case, and one that acts like both S and T (i.e. $S \curlywedge T$). What all of these operators have in common is that they allow us to describe a desired modification in terms of a semantically meaningful combination of two progams (or specifications). While not all possible modifications can be described in this way, the approach allows a systematic and dependable approach to implementing modifications that can be so described.

The approach presented here may be seen as a way of thinking about program modifications, rather than as a way of actually implementing those modifications. In practice, it is probably better to interleave the steps of constructing the program for the additional requirements and integrating it with the existing program. In fact, one might argue that this is something like what programmers actually do when modifying programs. We can support a systematic approach to implementing modifications based on these ideas by depicting both progams as structure diagrams (e.g. as in [4]) — we can then construct the structure diagram for the modification along side that of the original program, merging the two as the required structures are constructed. Alternatively, if the original program has been developed formally, via a sequence of transformations from a formal specification, we can interleave the construction of a derivation for the modification with integration of this derivation with the steps of the original derivation. The interleaving approach also allows us to identify places where the original program needs to be restructured in order to allow the modification to be implemented.

The approach as presented here, however, provides a clear way of understanding what is involved in modifying programs, which is easier to explain and understand than the interleaving version. It also provides a way of assessing the impact of different modifications, and different ways of implementing a given modification. For example, it illustrates clearly why a programmer might choose to implement the modification in the example using the one loop version, rather than the nested loops version, as observed in [10]. Even though the latter might be seen as a superior solution, it requires a substantial restructuring of the original program before it can be implemented.

Acknowledgements

Part of this work was completed while I was on study leave at Oxford University. I would like to thank the Oxford University Computing Laboratory, Kellogg College, and my host, Jim Woodcock, for their hospitality and support during this period. Thanks also to Ray Nickson for many fruitful discussions on this work, and helpful comments on this paper.

References

[1] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.

[2] E. W. Dijkstra. *A Discipline of Programming*. Academic Press, 1976.

[3] Lindsay Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[4] Michael Jackson. *Principles of Program Design*. Academic Press, 1975.

[5] K. Rustan M. Leino and Rajit Manohar. Joining specification statements. *Theoretical Computer Science*, 216:375–394, 1999.

[6] Brendan Mahony. The least conjunctive refinement and promotion in the refinement calculus. *Formal Aspects of Computing*, 11:75–105, 1999.

[7] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[8] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[9] Carroll Morgan and Trevor Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1993.

[10] J. I. Siddiqi and C. R. Roast. An empirical investigation into problem decomposition strategies used in program design. *International Journal of Man-Machine Studies*, 22:77–90, 1985.

[11] J. I. Siddiqi and C. R. Roast. Viscosity as a metaphor for measuring modifiability. *IEE Proceedings on Software Engineering*, 144(4):215–223, August 1997.