A Formal Approach to Program Evolution

Lindsay Groves

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

Abstract

We describe a systematic approach to implementing program modifications by composing an existing program with a specification of a desired modification, and then propagating the specification of the modification into the program structure to the place(s) where modifications need to be made. The approach is based on the refinement calculus, augmented with operators for composing specifications in ways required to express modifications, and properties of these operators that show how to modify the existing program to satisfy the new requirements.

1 Introduction

In order to support evolutionary development of provably correct software, we need to consider the following basic problem: Given a specification *Spec*, a program *Prog* that implements *Spec*, and a new specification *Spec*′, which is in some way "similar" to *Spec*, how can we construct a program *Prog*′ which implements *Spec*′?

In this scenario, *Spec*′ may be a modification of *Spec* reflecting a new requirement, a further step in an incremental development, a variation of *Spec* required for an alternative version, or a specification for a larger system in which *Spec* appears as a component. More generally, we may have a repository of specifications, and their implementations, from which *Spec* has been selected as a specifications whose implementation, *Prog*, may be helpful in implementing *Spec*′. In any case, we assume that there will be enough similarity between the required program *Prog*′ (or some component thereof) to make it worthwhile to attempt to reuse *Prog*, possibly in a modified form, in implementing *Spec*′, rather than constructing *Prog*′ from scratch.

One way to do this is to find a syntactic mapping from *Spec* to *Spec*′, and attempt to use this to transform *Prog* into *Prog*′. This kind of approach was pioneered by Dershowitz [DM77, Der83, Der85a, Der85b] as a basis for automatic program modification. Similar approaches have also been taken in program transformation [DF80, ABFP86, Gol90], planning [Mos89], and theorem proving [RS93, Vad95].

This paper presents an alternative approach where the required modification is described using some form of program combinator. The approach is based on the refinement calculus [Mor94, MV94, BvW98], so specifications and programs are expressed using a wide-spectrum language, obtained by adding specification constructs (most notably specification statements) to Dijkstra's guarded commands language [Dij76]. To this we add further program combinators, similar to the specification combinators used in Z [Spi92]. The semantics for the wide-spectrum language is defined in terms of weakest preconditions [Dij76], so we have a uniform basis for reasoning about programs and specifications, and mixtures of the two.

We begin by showing how our program combinators can be used to describe a variety of specification changes (section 2), and then discuss briefly how they can be used to obtain implementations for the resulting specifications (section 3). We conclude by discussing the significance of our results and contrasting to other work (section 4).

2 Specifying Modifications

We show how various kinds of modifications can be expressed by composing the original specification with another specification describing the modification. These are illustrated using a very simple example of a program that finds an integer k in the range 1 to n (for some positive constant n) satisfying some condition p, assuming that at least one such value exists. The specification, written as a specification statement, and our initial implementation, expressed using Dijkstra's guarded commands language, are as follows:

$$k : \left[\begin{array}{c} \exists\, i \in 1 \mathinner{..} n \bullet \\ p(i) \end{array} \middle/ \begin{array}{c} k \in 1 \mathinner{..} n \\ p(k) \end{array} \right] \quad \sqsubseteq \quad \begin{array}{l} k := 1\,; \\ \mathbf{do}\ \neg\, p(k) \rightarrow \\ \qquad k := k + 1 \\ \mathbf{od} \end{array}$$

A *specification statement*, $w : \left[pre \;\middle/\; post \right]$, describes a program that will establish the *postcondition*, *post*, altering only variables in the *frame*, w, provided that the *precondition*, *pre*, holds initially. Its semantics are defined as $wp(w : \left[pre \;\middle/\; post \right], \phi) \mathrel{\widehat{=}} pre \wedge (\forall\, w \bullet post \Rightarrow \phi)[w_0 \setminus w]$, where $E[w_0 \setminus w]$ denotes E with all free occurrences of variables in w replaced by the corresponding variables in w_0. Zero-subscripted variables are used in postconditions to refer to the initial values of variables that may be altered by the program.

We write $Spec \sqsubseteq Prog$ to mean that *Spec is refined by Prog*; i.e. that *Prog* is a correct implementation of *Spec*. Formally, $Spec \sqsubseteq Prog$ iff $wp(Spec\,,\, \phi) \Rightarrow wp(Prog\,,\, \phi)$, for all postconditions ϕ.

2.1 Sequential Composition

We can describe various kinds of modifications using sequential composition; for example, modifications that just involve changing the form of the input or output can be described by composing the existing specification with a specification describing the required transformation. We can also describe modifications that place additional constraints on the inputs or outputs by composing the original specification with a statement that expresses the additional constraints.

2.1.1 Assertions

An *assertion*, $\left\{ A \right\}$, where A is a state predicate, is a statement that does nothing (i.e. behaves like **skip**, defined as $wp(\mathbf{skip}\,,\, \phi) \mathrel{\widehat{=}} \phi$) if A is true in its initial state, and can do anything, including not terminate[1] (i.e. behaves like **abort**, defined as $wp(\mathbf{skip}\,,\, \phi) \mathrel{\widehat{=}} \mathit{false}$), otherwise. Its semantics are defined as $wp(\left\{ A \right\}, \phi) \mathrel{\widehat{=}} A \wedge \phi$.

Assertions can be used to describe modifications that impose additional constraints on the inputs (initial state) of a computation. For example, consider the search program described above, and suppose we are told that p never holds for even numbers. We can express this modification by sequentially composing the original specification with an assertion:

$$\left\{ \forall\, i \in 1 \mathinner{..} n \bullet even(i) \Rightarrow \neg\, p(i) \right\} ;\, k : \left[\exists\, i \in 1 \mathinner{..} n \bullet p(i) \;\middle/\; k \in 1 \mathinner{..} n \wedge p(k) \right]$$

It is easy to show that, for any assertion $\left\{ A \right\}$ and specification statement $w : \left[P \;\middle/\; R \right]$, we have $\left\{ A \right\} ;\, w : \left[P \;\middle/\; R \right] = w : \left[A \wedge P \;\middle/\; R \right]$, so the above composition is, as desired, equivalent to:

[1] In weakest precondition semantics, we cannot (and don't need to) distinguish between a program that never terminates and one that sometimes fails to terminate

$$k : \left[\begin{array}{c} \forall i \in 1 \,..\, n \bullet even(i) \Rightarrow \neg\, p(i) \\ \exists i \in 1 \,..\, n \bullet p(i) \end{array} \;\middle/\; k \in 1 \,..\, n \wedge p(k) \right]$$

2.1.2 Coercions

A *coercion*, $\left[A \right]$, where A is a state predicate, is a statement that does nothing (i.e. behaves like **skip**) if A is true in its initial state, and is infeasible (i.e. behaves like **magic**, defined as $wp(\mathbf{skip}\,,\, \phi) \mathrel{\widehat=} true$) otherwise. Its semantics are defined as $wp(\left[A \right], \phi) \mathrel{\widehat=} A \Rightarrow \phi$.

Coercions can be used to describe modifications that impose additional constraints on the outputs (final state) of a computation, which is only a sensible form of modification if the original specification is nondeterministic. For example, consider the search program described above, and suppose we are now required to find the *smallest* integer in $1 \,..\, n$ satisfying p. We can express this modification by sequentially composing the original specification with a coercion:

$$k : \left[\exists i \in 1 \,..\, n \bullet p(i) \;\middle/\; k \in 1 \,..\, n \wedge p(k) \right] ; \left[\forall i \in 1 \,..\, k-1 \bullet \neg\, p(i) \right]$$

It is easy to show that, for any coercion $\left[A \right]$ and specification statement $w : \left[P \;/\; R \right]$, we have $w : \left[P \;/\; R \right] ; \left[A \right] = w : \left[P \;/\; A \wedge R \right]$, so the above composition is, as desired, equivalent to:

$$k : \left[\exists i \in 1 \,..\, n \bullet p(i) \;\middle/\; \begin{array}{c} k \in 1 \,..\, n \wedge p(k) \\ \forall i \in 1 \,..\, k \bullet \neg\, p(i) \end{array} \right]$$

2.2 Program Disjunction

We define a program disjunction operator, $\curlyvee$, similar to Z's schema disjunction, so that $S \curlyvee T$ behaves like S for inputs in the precondition of S and like T for inputs in the precondition of T. We don't care what happens for inputs that are not in the precondition of either S or T; and for inputs in the preconditions of both S and T, $S \curlyvee T$ behaves nondeterministically like either S or T. When S is chosen, $S \curlyvee T$ should only modify variables that S can modify, and when T is chosen, $S \curlyvee T$ should only modify variables that T can modify — unlike schema disjunction in Z, where the disjunction of two schemas can modify any variables that either of the schemas can modify. We define program disjunction in terms of specification statements, as follows.[2]

$$w : \left[P \;/\; R \right] \curlyvee w : \left[P' \;/\; R' \right] \;=\; w : \left[P \vee P' \;\middle/\; \begin{array}{c} P[w \setminus w_0] \wedge R \\ \vee \\ P'[w \setminus w_0] \wedge R' \end{array} \right]$$

We can obtain a version for specification statements with different frames by first applying Morgan's *expand frame* law to unify the frames of the two components ([Mor94]).

Program disjunction can be used to describe modifications that involve relaxing the precondition, and specifying the outputs required for inputs that were outside the old precondition, which is only a sensible form of modification if the original specification is not total. For example, consider the search program described above, and suppose we are no longer able to assume that there is an integer in $1 \,..\, n$ satisfying p, and that the program is now required to return an integer outside the range $1 \,..\, n$ if there is no value in $1 \,..\, n$ satisfying p. This new behaviour can be described by the specification statement:

[2] To avoid a few technical complications, we assume that all programs are *conjunctive* (i.e. $wp(P\,,\, \phi \wedge \psi) = wp(P\,,\, \phi) \wedge wp(P\,,\, \psi)$, for all programs P and predicates ϕ and ψ). With this restriction, which rules out angelic nondeterminism, all programs can be expressed as equivalent specification statements.

$$k\colon\left[\forall\, i \in 1\,..\,n \bullet \neg\, p(i) \;\middle/\; k \notin 1\,..\,n\right]$$

Thus, the new specification can be obtained by using program disjunction to compose the specification of the new behaviour with the old specification:

$$k\colon\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; k \in 1\,..\,n \wedge p(k)\right] \curlyvee k\colon\left[\forall\, i \in 1\,..\,n \bullet \neg\, p(i) \;\middle/\; k \notin 1\,..\,n\right]$$

Expanding the above definition, we see that this is, as desired, equivalent to:

$$k\colon\left[\mathit{true} \;\middle/\; \begin{array}{c}(\exists\, i \in 1\,..\,n \bullet p(i)) \Rightarrow k \in 1\,..\,n \wedge p(k)\\ (\forall\, i \in 1\,..\,n \bullet \neg\, p(i)) \Rightarrow k \notin 1\,..\,n\end{array}\right]$$

2.3 Program Conjunction

We define a program conjunction operator, $\curlywedge$, similar to Z's schema conjunction, so that $S \curlywedge T$ behaves like S *and* like T, for inputs in the preconditions of both S and T. We don't care what happens for inputs that are not in the preconditions of both S and T, and since $S \curlywedge T$ is required to behave like S *and* like T, it must be able to modify any variables that either S or T can modify. Thus, $S \curlywedge T$ can change any variables that S or T can change, can assume everything that S and T can assume, and must achieve everything that S achieves, and everything that T achieves. We define program conjunction in terms of specification statements, as follows.

$$u\colon\left[P \;\middle/\; R\right] \curlywedge v\colon\left[P' \;\middle/\; R'\right] \;\widehat{=}\; u, v\colon\left[P \wedge P' \;\middle/\; R \wedge R'\right]$$

We can show that $S \curlywedge T$ is the least refined conjunctive program that refines both $\left\{\mathsf{pre}\ T\right\}\ ;\ S$ and $\left\{\mathsf{pre}\ S\right\}\ ;\ T$, provided both are first relaxed to allow variables in the other's frame to be modified [Gro00a].

Program conjunction can be used to describe modifications that involve adding assumptions about inputs (like assertion) and adding constraints on outputs (like coercions). It can also be used to describe modifications that introduce additional outputs. For example, consider the search program described above, and suppose we are required to also find the number of values in $1\,..\,n$ satisfying p. Again, we can specify the new behaviour using a separate specification statement:

$$c\colon\left[\mathit{true} \;\middle/\; c = \#\{i \in 1\,..\,n \bullet p(i)\}\right]$$

Thus, the new specification can be obtained by using program conjunction to compose the specification of the new behaviour with the old specification:

$$k\colon\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; k \in 1\,..\,n \wedge p(k)\right] \curlywedge c\colon\left[\mathit{true} \;\middle/\; c = \#\{i \in 1\,..\,n \bullet p(i)\}\right]$$

Expanding the definition above, we see that this is, as desired, equivalent to:

$$k, c\colon\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; \begin{array}{c}k \in 1\,..\,n \wedge p(k)\\ c = \#\{i \in 1\,..\,n \bullet p(i)\}\end{array}\right]$$

3 Implementing Modifications

We now consider how to implement specifications constructed using the combinators described above. Suppose the new specification has been written as $\mathit{Spec}' = \mathcal{F}(\mathit{Spec}, \mathit{Spec}'')$, where $\mathcal{F}$ is one of the specification constructors described in section 2 and Spec'' is a specification embodying the new requirements. More generally, Spec' may involve more than one combinator, but the case of one combinator will simply be repeated.

If the combinator used is *monotonic* with respect to refinement,[3] we can replace *Spec* by *Prog*, to get $\mathit{Prog}' = \mathcal{F}(\mathit{Prog}, \mathit{Spec}'')$, and have $\mathit{Spec} \sqsubseteq \mathit{Prog}'$. We can then proceed to transform Prog' into the required program, without having to repeat the derivation steps that lead to *Prog*. Of the combinators described above, only program disjunction is not monotonic; in that case, we can expand the disjunction into an equivalent **if** statement before making the replacement, or use weak monotonicity laws that allow component replacement in a disjunction provided that the refinement to one component does not introduce behaviours that conflict with the other component [Gro00a].

To transform Prog' into the required program, we use a variety of laws for propagating the program combinators described in section 2 into the structure of a program and for turning these combinators into executable programs. Some of these laws require information, such as loop invariants, that is not explicit in the program; this can either be obtained from a program derivation, if that is available, or by analysis of the program. In some cases, we have to modify the structure of the exisiting program before we can apply these laws.

We illustate this approach by describing briefly the changes that would result for the various modifications of the search algorithm discussed in section 2.

3.1 Strengthening a precondition

Strengthening a precondition, as described in section 2.1.1, does not *require* any modification to the program, since if $S \sqsubseteq P$ holds, $\{A\}\ ;\ S \sqsubseteq P$ will also hold. Sometimes, however, we can use the additional information provided by a strengthened precondition to make the program more efficient.

We can use a variety of laws to propagate as assertion into the structure of a program, possibly also collecting other information about the effect of the program, till it (or an assertion derived from it) reaches a point where some change is required. We can then use other laws that use the assertion as context to make the required modification(s).

In the modification described in section 2.1.1, the added assumption means that the program only needs to test p for even numbers. In making this modification, we propagate the assertion into the program structure till we reach the loop. The additional information can then be used to modify the implementation of the loop body, noting that the loop invariant $(k \in 1 \mathinner{..} n \land \exists\, i \in k \mathinner{..} n \bullet p(i))$ can be maintained while incrementing k by 2. The resulting program is:

$$
\begin{array}{l}
k := 1\ \mathbf{;} \\
\mathbf{do}\ \neg\, p(k) \rightarrow \\
\quad k := k + 2 \\
\mathbf{od}
\end{array}
$$

3.2 Strengthening a postcondition

Strengthening a postcondition, as described in section 2.1.2, may require no modification to the program, if the existing program happens to resolve nondeterminism in a way that is consistent with the added

[3] An N-ary program combinator, $\mathcal{F}$, is *monotonic* with respect to refinement iff $S_i \sqsubseteq S_i'$ implies $\mathcal{F}(\ldots, S_i, \ldots) \sqsubseteq \mathcal{F}(\ldots, S_i', \ldots)$, for $i = 1, \ldots, N$.

requirements; may require some minor modification if the program is still nondeterministic; may require more major modification if the existing program resolves nondeterminism in a way that is inconsistent with the added requirements; or may be impossible if the added requirements are inconsistent with the original specification.

We can use a variety of laws to propagate a coercion backwards through the structure of a program, till it (or a coercion derived from it) reaches a point where some change is required. We can then use other laws that use the coercion as context to make the required modification(s).

In the modification described in section 2.1.2, the program requires no change because the given program already does return the smallest number in $1 \mathrel{..} n$ satisfying p — but we first need to prove that this is the case. We can do this by propagating the coercion backwards into the program until we reach a point where its condition can be shown to hold. In this case, we need to show that the loop invariant can be strengthened to include the additional condition $\forall\, i \in 1 \mathrel{..} k-1 \bullet \neg\, p(i)$.

If the modification instead required that we find the *largest* number in $1 \mathrel{..} n$ satisfying p, we would need to add the condition $\forall\, i \in k+1 \mathrel{..} n \bullet \neg\, p(i)$ to the loop invariant. Since this condition is not preserved by the loop in the existing program, we would need to change the implementation. We might, for example, use a counting down loop (as in the program on the left), or introduce an extra variable and update k each time a value satisfying p is found (as in the program on the right):

$$
\begin{array}{l}
k := n\ \textbf{;} \\
\textbf{do}\ \neg\, p(k) \rightarrow \\
\qquad k := k - 1 \\
\textbf{od}
\end{array}
$$

$$
\begin{array}{l}
[\![\ \textbf{var}\ j \bullet \\
\quad j := 1\ \textbf{;} \\
\quad \textbf{do}\ j \leq n \rightarrow \\
\qquad \textbf{if}\ \ p(j) \rightarrow k := j\ [\!]\ \neg\, p(j) \rightarrow \textbf{skip fi}\ \textbf{;} \\
\qquad j := j + 1 \\
\quad \textbf{od}\ \textbf{;} \\
]\!]
\end{array}
$$

Since this example is very small, this results in the whole program being changed, but this would not generally be the case in a larger program.

3.3 Weakening a precondition

Weakening a precondition, as described in section 2.2, usually requires the introduction of a case analysis to allow the new inputs to be distinguished from the old ones. Sometimes this can be done by wrapping the existing program in an **if** statement, but in many cases this will not give an acceptable implementation because the existing program already does much of the work required to distinguish the cases. In such cases, the case analysis needs to be introduced deeper in the structure of the program, and may require other changes, such as modifying loop guards.

We can effect this kind of modification by pushing the disjunction into the structure of the program, using various distribution laws, until it reaches the place(s) where the modification needs to be made. At each step, we turn the specification of the modification into a program with a "similar" structure to the existing program, so that the two can be merged — where by "similar" we mean a form that allows one of our distribution laws to be applied. We can think of this process as constructing an implementation for the required new behaviour that is structurally consistent with the existing program and then merging them. In some cases, we may need to modify the structure of the original program in order to obtain the required similarity.

For the modification described in section 2.2, we need to modify the program so that it will work for cases where there is no number in $1 \mathrel{..} n$ satisfying p. We can't simply turn the new specification into an **if** statement, because to determine that there is no number in $1 \mathrel{..} n$ satisfying p requires a loop similar to the one in the existing program. Instead, we push the required change into the structure of the program until we find the place(s) where changes are required. As outlined above, this effectively means creating an implementation for the new behaviour that is structurally consistent with the existing program, which in this case is:

$$
\begin{array}{l}
k := 1\ \mathbf{;} \\
\mathbf{do}\ k \le n \rightarrow \\
\quad \left\{\neg\ p(k)\right\}\ \mathbf{;} \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}
$$

The assertion allows the program to abort if a number satisfying p is found, so this program is only guaranteed to terminate if no number in $1 \mathinner{..} n$ satisfies p. In that case it will return $k = n + 1$, which satisfies the requirement of not being in the range $1 \mathinner{..} n$.

The disjunction of two loops can be thought of as a program that executes the two loops in parallel until one of them terminates. If their bodies can be refined to the same statement (thereby removing the parallelism), we can merge the two loops to produce a loop whose guard is the conjunction of their guards. In our example, the guard of the original program ensures that $\neg\ p(k)$ holds in the loop body, so the assertion $\left\{\neg\ p(k)\right\}$ in the above program will always behave like **skip** and can thus be removed, so both loop bodies end up being the same. Thus, we obtain the expected program:

$$
\begin{array}{l}
k := 1\ \mathbf{;} \\
\mathbf{do}\ \neg p(k) \wedge k \le n \rightarrow \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}
$$

In practice, we do not need to construct the program for the new behaviour separately like this, but this is a convenient and compact way to describe what is happening.

3.4 Adding an output

Adding an output, as described in section 2.3, requires that we add extra code to compute the values for the additional output. Sometimes we can turn a conjunction into a sequence, and do the computations for the additional output separately; but often it is better to combine the new computations along with the existing ones (for example, we often prefer to just traverse an input sequence once).

We can effect such modifications by pushing the conjunction into the structure of the program until we find the place(s) where changes are required, in a way similar to that described in section 3.3. At each step, we again turn the specification of the modification into a program with a "similar" structure to the existing program, so that the two can be merged. We can think of this process as constructing an implementation for the required new behaviour that is structurally consistent with the existing program and then merging them. Again, in some cases, we may need to modify the structure of the original program in order to obtain the required similarity (see [Gro00b]).

For the modification described in section 2.3, we need to augment the program with code to find the number of values in $1 \mathinner{..} n$ satisfying p. We can write a program for this with a "similar" structure to the original:

$$
\begin{array}{l}
k, c := 1, 0\ \mathbf{;} \\
\mathbf{do}\ k \le n \rightarrow \\
\quad \mathbf{if}\ \ p(k) \rightarrow c := c + 1\ [\!]\ \neg\ p(k) \rightarrow \mathbf{skip}\ \mathbf{fi}\ \mathbf{;} \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}
$$

The conjunction of two loops can be though of as a program that executes the two loops in parallel until both of them terminate. Since, in our example, the loop of the original program will terminate before the loop in the modification, we need to allow the latter to continue after the former has finished. So that the two loops do not interfere with each other, we also need to introduce an additional variable to use as a loop counter. The resulting program is:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ j \bullet \\
\quad j, k, c := 1, 1, 0\ ; \\
\quad \mathbf{do}\ \neg\, p(j) \wedge j \le n \rightarrow \\
\quad\quad \mathbf{if}\ \ p(k) \rightarrow c := c + 1\ [\!]\ \neg\, p(k) \rightarrow \mathbf{skip}\ \mathbf{fi}\ ; \\
\quad\quad j, k := j + 1, k + 1 \\
\quad \mathbf{od}\ ; \\
\quad \mathbf{do}\ j \le n \rightarrow \\
\quad\quad \mathbf{if}\ \ p(k) \rightarrow c := c + 1\ [\!]\ \neg\, p(k) \rightarrow \mathbf{skip}\ \mathbf{fi}\ ; \\
\quad\quad j := j + 1 \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

Although this may not be the simplest program one could write to implement the modified specification, it was obtained by incorporating the additional requirements in a systematic and reliable way, driven by the specification of the required modification.

Again, in practice we do not need to construct the program for the new behaviour separately like this, but this is a convenient and compact way to describe what is happening.

4 Discussion

We have shown that a range of typical program modifications can be specified by composing the original specification with a specification describing the required change. Some modifications can be described using existing features of the refinement calculus (notable, sequential composition with assertions and coercions). To describe others we have introduced program combinators similar to Z's schema conjunction and disjunction, but adapted to behave in a way that is better suited to program development.

We have also shown that a modification specified in this way can often be implemented by propagating the required modification through the existing program to the place(s) where changes need to be made (of course, we do not expect all possible modifications to be effected this easily!). These transformations are justified using laws based on the semantics for the wide-spectrum language, providing both guidance as to how to proceed and assurance that the resulting program is correct.

This approach is similar to that of Frappier *et al.* [FMD98], but uses a different programming language and different combinators, and thus ends up with different laws and a different style of presentation. The approach can be contrasted to the earlier syntactic approaches, like Dershowitz [DM77, Der83, Der85a, Der85b], where there is no semantic relationship between the original program and the new one.

We have only been able to provide a brief outline of the approach here, using very simple examples; further details of the laws used and more extensive examples can be found elsewhere: [Gro98] and [Gro00b] deal with conjunction, [Gro00c] deals with disjunction, and [Gro01] illustrates the use of assertions in program specialisation. [Gro00a] provides detailed proofs and examples of the laws for all of the combinators used here, in a more general setting that includes angelic nondeterminism, and also explores the use of an *override* operation, which alters the behaviour of an existing program for specified inputs, and a *join* operation, which is a kind of cross between disjunction and conjunction (the join of S and T

behaves like S whenever S terminates and like T whenever T terminates, so it behaves like $S \curlywedge T$ when they both terminate [AW94]).

Further experience is necessary to determine which of these combinators are most useful in practice, and to determine what other combinators may be required. Although our aim is primarily to provide a logical basis for evolution of provably correct programs, we need tool support in order to be able to experiment with application of this approach to larger scale examples. We thus need to consider whether refinement tools such as PRT [CHN+98] and The Refinement Calculator [BGL+97] can be extended to provide support for this kind of evolutionary development. Our experience to date suggests that navigation is a more acute problem in program modification than in program refinement, and that new ways of visualising the state of a development will be needed.

References

[ABFP86] Guillermo Arango, Ira Baxter, Peter Freeman, and Christopher Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, May 1986.

[AW94] Mike Ainsworth and Peter Wallis. Co-refinement. In David Till, editor, *6th Refinement Workshop*, pages 151–166. Springer-Verlag, 1994.

[BGL+97] Michael Butler, Jim Grundy, Thomas Långbacka, Rimvydas Ruksenas, and Joakim von Wright. The refinement calculator: Proof support for program refinement. In Lindsay Groves and Steve Reeves, editors, *Formal Methods Pacific '97*, Discrete Mathematics and Theoretical Computer Science Series, pages 40–61. Springer-Verlag, July 1997.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[CHN+98] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.

[Der83] Nachum Dershowitz. *The Evolution of Programs*. Birkhäuser, 1983.

[Der85a] Nachum Dershowitz. Program abstraction and instantiation. *ACM Transactions on Programming Languages and Systems*, 7(3):446–477, July 1985.

[Der85b] Nachum Dershowitz. Synthetic programming. *Artificial Intelligence*, 25:323–373, March 1985.

[DF80] John Darlington and Martin Feather. A transformational approach to modification. Technical Report 80/3, Department of Computing and Control, Imperial College, London, 1980.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Academic Press, 1976.

[DM77] Nachum Dershowitz and Zohar Manna. The evolution of programs: Automatic program modification. *IEEE Transactions on Software Engineering*, SE-3(6):377–385, November 1977.

[FMD98] M. Frappier, A. Mili, and J. Desharnais. Unifying program construction and modification. *Logic Journal of the IGPL*, 6(2):317–340, 1998.

[Gol90] Allen Goldberg. Reusing software developments. In *Proc. ACM SIGSOFT 4th Symposium on Software Development Environments*, number 15 in Software Engineering Notes, pages 107–119, December 1990.

[Gro98] Lindsay Groves. Adapting program derivations using program conjunction. In Jim Grundy, Martin Schwenke, and Trevor Vickers, editors, *Proc. International Refinement Workshop and Formal Methods Pacific 1998*, pages 145–164. Springer Verlag, Singapore, 1998.

[Gro00a] Lindsay Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[Gro00b] Lindsay Groves. A formal approach to program modification. In *Proc. of the Seventh Asia-Pacific Software Engineering Conference*, pages 274–281, Singapore, 5-8 December 2000. IEEE Computer Society.

[Gro00c] Lindsay Groves. Modifying program derivations using program disjunction. In P. Krishnan, editor, *Proceedings of the Sixth New Zealand Formal Program Development Colloquium*, pages 35–46, Christchurch, 23 August 2000. University of Canterbury.

[Gro01] Lindsay Groves. Program specialisation in the refinement calculus. In He Jifeng, Li Yiping, and G. Lowe, editors, *Proc. of the Eigth Asia-Pacific Software Engineering Conference*, pages 29–36, Macau, 4-7 December 2001. IEEE Computer Society.

[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[Mos89] Jack Mostow. Design by derivational analogy: Issues in the automated replay of design plans. *Artificial Intelligence*, 40:119–184, 1989.

[MV94] Carroll Morgan and Trevor Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1994.

[RS93] Wolfgang Reif and Kurt Stenzel. Reuse of proofs in software verification. In *Proc. 13th Conf. on Foundations of Software Technology and Theoretical Computer Science*, number 761 in Lecture Notes in Computer Science, pages 284–293. Springer-Verlag, 1993.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual (second edition)*. Prentice Hall International, 1992.

[Vad95] Sunil Vadera. Proof by analogy in mural. *Formal Aspects of Computing*, 7(2):183–206, 1995.