VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Adapting Formal Derivations

Lindsay Groves

Technical Report CS-TR-95/9
April 1995

Abstract

The refinement calculus is usually presented as a formalisation of stepwise refinement, so there is an (often tacit) assumption that programs are going to be constructed from scratch, in a strictly top-down fashion. Most of the rules in the refinement calculus are concerned with decomposing specifications into more procedural forms, and existing refinement tools primarily provide support for this kind of construction.

We argue that in order to support more realistic styles of program construction, we need to consider ways in which an existing program derivation can be adapted to reflect a modification in the initial specification, or perhaps to produce a different algorithm for the same specification. This will allow us to support more flexible styles of program construction, such as incremental enhancement, and also to provide support for program maintenance.

We discuss some of the mechanisms required to support adaptation of formal derivations in the refinement calculus, and outline the architecture of a proposed tool to support this style style, based on the idea of an editable derivation script. We then show how this approach can be used to adapt formal derivations to obtain programs to satisfy different specifications.

Publishing Information

Appeared in *The Fourth Australian Refinement Workshop*, University of New South Wales, Sydney, April 20-21 1995, *pp85*–99.

Adapting Formal Derivations

Lindsay Groves

Department of Computer Science
Victoria University of Wellington
Wellington, New Zealand
Email: lindsay@comp.vuw.ac.nz

Abstract

The refinement calculus is usually presented as a formalisation of stepwise refinement, so there is an (often tacit) assumption that programs are going to be constructed from scratch, in a strictly top-down fashion. Most of the rules in the refinement calculus are concerned with decomposing specifications into more procedural forms, and existing refinement tools primarily provide support for this kind of construction.

We argue that in order to support more realistic styles of program construction, we need to consider ways in which an existing program derivation can be adapted to reflect a modification in the initial specification, or perhaps to produce a different algorithm for the same specification. This will allow us to support more flexible styles of program construction, such as incremental enhancement, and also to provide support for program maintenance.

We discuss some of the mechanisms required to support adaptation of formal derivations in the refinement calculus, and outline the architecture of a proposed tool to support this style style, based on the idea of an editable derivation script. We then show how this approach can be used to adapt formal derivations to obtain programs to satisfy different specifications.

1 Introduction

The search for better ways of developing software by refinement is largely a search for mathematical power and elegance. In this search for elegant solutions, we should not forget the less elegant processes which precede them. Our tools and techniques should support those inelegant, opportunistic processes as much as they support the elegant results.
Robert Worden [Worden, 1991]

Most work on formal program construction is based on the (often tacit) assumption that programs will be constructed from scratch, in a strictly top-down fashion. The refinement calculus [Morgan, 1990] [Morgan, Robinson and Gardiner, 1988], for example, is generally presented as a formalisation of step-wise refinement [Wirth, 1971], and current refinement tools best support this style of development [Carrington, *et al*]. Top-down design, however, is at best an idealisation, not a realistic model of program development.

In constructing programs, we don't always make decisions in a strictly top-down fashion. In order to make a good decision, we often need to be able to look ahead several steps to predict how later steps will be handled. We frequently don't manage to accurately predict how later steps are going to work out; we may thus make false starts and have to back up and change earlier decisions.

Moreover, we seldom construct programs completely from scratch in the manner that the top-down model suggests. We routinely modify or extend existing programs, either to meet new requirements or to improve performance. If we are constructing a new program, we might initially construct a simpler program than the one we want, by ignoring some of the requirements, and then modify or extend that program to address the additional requirements. We may also use previously written fragments of code, embodying useful programming techniques (clichés), adapting them to the current context as necessary.

These observations hold just as much for formal programming as for informal. The fact that a program derivation is described formally as a sequence of refinement steps does not mean that it will necessarily be constructed in that way. If the refinement calculus is to be used as a practical tool for constructing programs, rather than as a way of describing idealised developments, we need to base our calculus and our support tools on a more realistic model of software development.

The formal methods literature often suggests that recording the steps in a formal development might be helpful for later maintenance, but this does not appear to have been studied in any detail, and current refinement tools provide only limited support for anything other than strict top-down construction. Although facilities may be provided for editing derivations, these are rather *ad hoc*, and are not based on a systematic analysis of the kinds of modifications that may be required in order to support any particular style of program development.

In this paper, we attempt to identify some of the ways in which we would need to be able to adapt formal derivations in order to support program construction by incremental enhancement, and program maintenance and reuse. We begin by outlining a model of the programming process we envisage, then describe the architecture of a proposed refinement tool intended to support this model, based on the idea of an editable derivation script. We then describe our notation for writing derivation scripts in more detail, and present a sequence of examples illustrating this notation and some ways in which program derivations can be obtained by adapting existing derivation scripts.

2 Modelling the refinement process

In designing a refinement tool, we should begin with a model of the kinds of programming activity we intend to support. This will then serve as a basis for deciding which activities should be performed by the refinement tool, and which should be left to the programmer (this is known as "task allocation").

Most current refinement tools [Carrington, *et al*] are based on a fairly basic (and often implicit) model of program construction by top-down refinement. It is acknowledged that programmers may make mistakes, and so basic facilities are provided for undoing and modifying earlier steps, but these are very limited. Naftalin [Naftalin, 1992] observes that programmers typically work at several levels of abstraction simultaneously and that insights at one level are often applied to other levels in an opportunistic manner, and outlines the design of a refinement tool to support this.

Our aim is to understand ways in which adaptation and reuse of existing programs can be accommodated within a formal framework. We view this as important for a number of reasons.

It has often been observed that specifications often change as implementation proceeds, for example, because of changes in the the client's expectations or the programmer's understanding of the problem. This has lead to considerable interest in the use of prototypes. In particular we would like to support program construction by incremental enhancement in a formal setting, so a programmer may begin by developing a program for a simplified specification and later extend it to address additional requirements. For example, in constructing a compiler, we might begin by developing a simple recogniser for the language, then extend this to produce a parse tree or translation of the program recognised, and still later add error reporting and recovery. In each case, we need to be able to take an existing derivation and adapt it to a modified specification.

Some refinement tools support the reuse of sequences of refinement steps, by allowing them to be packaged as derived rules [Vickers, 1990] or as tactics [Nickson, 1994]. While these facilities may simplify the task of constructing new programs, they do not address the problem of adapting existing derivations to address modified specifications.

It is not our aim here to describe a model of the refinement process we envisage in detail. Rather, we outline some essential aspects of such a model, and indicate some of the options they open in the design of a refinement tool.

The following are some of the basic steps we expect the user to be able to perform:

- Construct a program (and its derivation), by beginning with a specification and applying rules one at a time. This will involve selecting the rule to be applied, the position in the current program to which it is to be applied, and providing values for parameters of the rule (metavariables in the rule that do not appear in the subject pattern). This can be done in a number of ways, for example using a point and click interface.

- Generate the proof obligations associated with rule applications, and either provide support for performing the necessary proofs, or at least giving the user the opportunity to judge whether they are valid.

- Edit the current derivation, for example, by undoing one or more steps or changing parameters in an earlier rule application. A simple way to do this is by chronologically undoing preceding steps. A more ambitious (and useful) approach is to allow the user to undo an arbitrary step, and for the tool to work out which subsequent steps are no longer valid.

- Review a derivation, stepping through the refinements (both forwards and backwards), and observing the effects of each step.

These facilities are supported by a number of current refinement tools [Carrington, *et al*], notably Red [Vickers, 1990] and Ray Nickson's refinement tool [Nickson, 1994], which we will refer to here as RRT. Our model of refinement (and thus our refinement tools) might also incorporate data refinement, and the introduction of procedures and modules, but we will not discuss these further here.

In order to support more flexible styles of program development, such as incremental enhancement and reuse, we also wish to be able to adapt an existing derivation to suit a new situation. In particular, we want to be able to take an existing derivation and apply it to a new specification, adapting it as necessary in order to suit the new specification.

3 An architecture for adapting derivations

Our proposed approach is based on the idea of an editable derivation script. A derivation is described as a sequence of steps, typically involving the application of refinement rules (or perhaps some other form of transformation rule). If we construct a program in a top-down manner, as described above, the refinement tool can automatically construct a script describing that derivation. We can then browse the derivation by stepping backwards and forwards through this script, undoing and redoing the refinement steps. We can reuse derivations by inserting a saved script into the current script, adapting it to address modified requirements as necessary.

In order to support this style of operation, we envisage a refinement tool intended which allows derivation scripts to be constructed, saved, copied and edited in various ways. The refinement tool will use two main windows, one displaying the current program, and one displaying a derivation script. The program window is similar to the main window in RRT or in Red. The user can select components of the program to be transformed by clicking or dragging with the mouse.

The script window is divided into three parts. The centre part displays the refinement step that is currently being assembled or waiting to be applied. The part above the current step contains a history of steps that have been applied so far: the current program is always the result of applying the steps in the history to the initial specification. The part below the current step contains steps that are available to be applied in the future. These steps can be applied one at a time, edited, etc., and provide the basic capabilities for adapting derivations.

At each step, the user may perform a number of operations on the script window, for example:

- Insert a new refinement step. A standard top-down development can be done by starting with an empty script and inserting new steps in much the same manner as RRT.

- Step backwards through previous steps, or jump to any earlier step. The program window is updated to reflect the program as at that step, and the steps that are undone move into the lower part of the script window, and can be reapplied as described below.

- Apply the next step from the script, provided that it is applicable. The subject of the current step is highlighted in the program window; the user can ask to see the proof obligation, and either prove it or vouch for it before proceeding to apply that step.

- Modify the next step of the script. This may be used to modify a step that was previously inapplicable, or to alter a step which was applicable so that it gives a different result (e.g. to get a different algorithm or improve performance). Such modifications may also be propagated through later steps in the script (this is rather like the notion of analogy discussed in [Vadera, 1994]).

- Delete the next step (or any number of steps) from the script window.

- Insert an existing script into the script window. This will allow us to insert an existing derivation for a subproblem (or variant thereof) and adapt it if necessary to suit the current situation.

- Apply remaining steps from the script window automatically, until a step fails, all steps are used, or the user interrupts. In this way, we can run an existing script until it needs adaptation, e.g. till it tries to apply a rule that doesn't match its subject or generates a proof obligation that can't be discharged. At that stage, we can revert to manual adaptation.

4 Describing refinements

We treat a program derivation as a sequence of steps, each of which involves applying some transformation to part of the program resulting from the preceding steps. A program derivation can thus be described by a script, comprising a sequence of descriptions of such steps.

For this discussion, the steps in a derivation will typically involve applying a refinement rule, from a collection of named rules. An application of a refinement rule can be described as follows:

$$Subject \ \lhd \ RuleName(parameters)$$

Subject is a name identifying the component of the current program to be transformed, *RuleName* is the name of the rule to be applied, and *parameters* is a list of parameters to be used in instantiating the rule.

In order to associate names with program components, we also allow the description of a rule application to include a pattern which is matched with the result of the rule:

$$Subject \ \lhd \ RuleName(parameters) \ \rhd \ Pattern$$

The syntax for patterns is the same syntax as for programs, except that any component may be replaced by a meta-variable to be associated with that component, much like patterns used in refinement rules. We also allow unnamed variables (denoted by underscore, as in Prolog) to be used where a component does not need to be referred to later.

The first line of a script is of the form $Name = Pattern$, which associates names with the initial specification ($Name$) and its components, and allows the form of the initial specification to be restricted.

This notation is based on the tactic notation used in RRT [Nickson, 1994]. Although not as compact as the notation used in Red [Vickers and Gardiner, 1994], this appears to be more suitable as a basis for the kinds of adaptations we wish to perform. It should also be easy to transform a script into that notation.

For practical use, this script language will need to be extended to include, for example, notation for declaring and applying derived rules and tactics, defining new syntax and other abbreviations, and importing theories. It may also include records of proofs, or at least hints to a proof checker. The notation described above is, however, all we need for this paper.

5 Example: Minimum of two numbers

Consider the following derivation (ignoring proof obligations) for a program to find the minimum, m, of two numbers, x and y:

$$m\colon \left[\, true \, \middle/ \, m = min(x, y) \right] \tag{S_1}$$

$$
\begin{array}{lll}
(S_1) & \sqsubseteq & (\mathit{Introduce\ IF}) \\
& & \textbf{if}\quad x \le y \rightarrow m\colon \left[x \le y \;\middle/\; m = \mathit{min}(x, y) \right] \qquad (S_2) \\
& & [\!]\quad x \ge y \rightarrow m\colon \left[x \ge y \;\middle/\; m = \mathit{min}(x, y) \right] \qquad (S_3) \\
& & \textbf{fi}
\end{array}
$$

$$
\begin{array}{lll}
(S_2) & \sqsubseteq & (\mathit{Introduce\ Assignment}) \\
& & m := x
\end{array}
$$

$$
\begin{array}{lll}
(S_3) & \sqsubseteq & (\mathit{Introduce\ Assignment}) \\
& & m := y
\end{array}
$$

The resulting program is:

$$
\begin{array}{ll}
\textbf{if} & x \le y \rightarrow m := x \\
[\!] & x \ge y \rightarrow m := y \\
\textbf{fi} &
\end{array}
$$

This derivation can be described using the following script:

$$
\begin{array}{l}
S_1 = m\colon \left[\mathit{true} \;\middle/\; m = \mathit{min}(x, y) \right] \\
S_1 \;\lhd\; \mathit{IntroIF}(x \le y, x \ge y) \;\rhd\; \textbf{if}\ \_ \rightarrow S_2 \ [\!]\ \_ \rightarrow S_3\ \textbf{fi} \\
S_2 \;\lhd\; \mathit{IntroAssign}(m := x) \\
S_3 \;\lhd\; \mathit{IntroAssign}(m := y)
\end{array}
$$

The pattern in the first line indicates the form of specification that this script can be applied to, and matching this pattern with a specification binds names in the script to those in the specification. Thus, to find the minimum of u and v and store it in z, we just replace m, x and y by z, u and v, respectively. Also note how the result pattern in the *IntroIF* step provides names for the components to be refined by the following steps.

6 Example: Sum of an array

As a second example, consider the following derivation (again ignoring proof obligations) for a program to sum to elements of an array segment $A[1 \mathinner{..} n]$:

$$
s\colon \left[\mathit{true} \;\middle/\; s = \mathit{sum}(A[1 \mathinner{..} n]) \right] \qquad (S_1)
$$

$$
\begin{array}{lll}
(S_1) & \sqsubseteq & (\mathit{Introduce\ Local\ Variable}) \\
& & \textbf{var}\ k\ \bullet \\
& & s, k\colon \left[\mathit{true} \;\middle/\; s = \mathit{sum}(A[1 \mathinner{..} n]) \right] \qquad (S_2)
\end{array}
$$

$$(S_2) \quad \sqsubseteq \quad (\textit{Split Specification})$$

$$s, k\colon \left[\, \textit{true} \;\Big/\; 0 \leq k \leq n \wedge s = \textit{sum}(A[1 \mathinner{..} k]) \,\right]; \qquad (S_3)$$

$$s, k\colon \left[\, 0 \leq k \leq n \wedge s = \textit{sum}(A[1 \mathinner{..} k]) \;\Big/\; s = \textit{sum}(A[1 \mathinner{..} n]) \,\right] \qquad (S_4)$$

$$(S_3) \quad \sqsubseteq \quad (\textit{Introduce Assignment})$$

$$s, k := 0, 0$$

$$(S_4) \quad \sqsubseteq \quad (\textit{Introduce DO})$$

$$\begin{array}{l} \mathbf{do}\ k \neq n \rightarrow \\ \qquad s, k\colon \left[\begin{array}{c} 0 \leq k < n \\ s = \textit{sum}(A[1 \mathinner{..} k]) \end{array} \;\Bigg/\; \begin{array}{c} 0 \leq k \leq n \\ s = \textit{sum}(A[1 \mathinner{..} k]) \\ k_0 < k \end{array} \right] \qquad (S_5) \\ \mathbf{od} \end{array}$$

$$(S_5) \quad \sqsubseteq \quad (\textit{Following Assignment})$$

$$s\colon \left[\, 0 \leq k < n \wedge s = \textit{sum}(A[1 \mathinner{..} k]) \;\Big/\; s = \textit{sum}(A[1 \mathinner{..} k+1]) \,\right]; \qquad (S_6)$$

$$k := k + 1$$

Note that the conjunct $k \leq n$ has been removed from the postcondition, because k is no longer in the frame[1].

$$(S_6) \quad \sqsubseteq \quad (\textit{Introduce Assignment})$$

$$s := s + A[k+1]$$

This completes the derivation. The resulting program is:

$$\begin{array}{l} [\![\ \mathbf{var}\ k \bullet \\ \quad s, k := 0, 0; \\ \quad \mathbf{do}\ k \neq n \rightarrow \\ \qquad s := s + A[k+1]; \\ \qquad k := k + 1 \\ \quad \mathbf{od} \\ ]\!] \end{array}$$

This derivation can be described by the following script:

[1] Our *Following Assignment* rule removes the variables assignment by the assignment from the frame of the leading specification.

$S_1 = s\colon \left[true \;\middle/\; s = sum(A[1\,..\,n]) \right]$

$S_1 \;\lhd\; IntroVar(k) \;\rhd\; [\![\;\mathbf{var}\; k \bullet S_2 \;]\!]$

$S_2 \;\lhd\; SplitSpec(0 \le k \le n \land s = sum(A[1\,..\,k])) \;\rhd\; S_3\;;\; S_4$

$S_3 \;\lhd\; IntroAssign(s, k := 0, 0)$

$S_4 \;\lhd\; IntroDO(0 \le k \le n \land s = sum(A[1\,..\,k]), k \ne n, n - k) \;\rhd\; \mathbf{do}\; k \ne n \to S_5 \;\mathbf{od}$

$S_5 \;\lhd\; FollAssign(k := k + 1) \;\rhd\; S_6\;;\; \_$

$S_6 \;\lhd\; IntroAssign(s := s + A[k + 1])$

7 Example: Minimum of an array

Suppose we wish to construct an algorithm to find the minimum element in an array. We already know how to construct algorithms to find the minimum of two numbers and to find the sum of an array, so we should be able to use this knowledge to construct the new algorithm.

We will explain how this would be done by modifying the previously constructed scripts using our proposed refinement tool. We show the result of each step applied as a refinement, rather than displaying the entire resulting program as it would appear in the program window.

We begin with the new specification in the program window:

$$lo\colon \left[n \ge 1 \;\middle/\; lo = min(A[1\,..\,n]) \right] \qquad (S_1)$$

We load the array summation script into the script window, and attempt to apply its steps to the new specification.

- $S_1 = s\colon \left[true \;\middle/\; s = sum(A[1\,..\,n]) \right]$

  The pattern in the first line doesn't match the current specification. To obtain a match, we replace s by lo, $true$ by $n \ge 1$ and sum by min. We make these changes systematically throughout the rest of the script, using a query-replace dialogue. This allows the user to ensure that only appropriate occurrences of s and sum are replaced — though in this case all occurrences should be replaced.

- $S_1 \;\lhd\; IntroVar(k) \;\rhd\; [\![\;\mathbf{var}\; k \bullet S_2 \;]\!]$

  This step can be applied unchanged, and corresponds to the following refinement:

$$\begin{array}{lll} (S_1) & \sqsubseteq & (\textit{Introduce Local Variable}) \\ & & \mathbf{var}\; k \bullet \\ & & lo, k\colon \left[n \ge 1 \;\middle/\; lo = min(A[1\,..\,n]) \right] \end{array} \qquad (S_2)$$

- $S_2 \;\lhd\; SplitSpec(0 \le k \le n \land lo = min(A[1\,..\,k])) \;\rhd\; S_3\;;\; S_4$

  We could apply this step as is, but looking ahead, we realise that the intermediate assertion should be strengthened to use 1 as the lower bound for k. Again, this change is propagated through the rest of the script, using a query-replace dialogue to ensure that only the appropriate occurrences of 0 are replaced — in this case, not all occurrences should be replaced. Note that s and min have already been replaced by lo and min.

$$(S_2) \quad \sqsubseteq \quad (\textit{Split Specification})$$

$$lo, k\colon \left[n \geq 1 \;\middle/\; 1 \leq k \leq n \wedge lo = min(A[1\mathinner{..}k]) \right] ; \qquad (S_3)$$

$$lo, k\colon \left[1 \leq k \leq n \wedge lo = min(A[1\mathinner{..}k]) \;\middle/\; lo = min(A[1\mathinner{..}n]) \right] \qquad (S_4)$$

- $S_3 \;\lhd\; \textit{IntroAssign}(lo, k := 0, 1)$

  If we apply this rule as is, we end up with $0 = min(A[1\mathinner{..}1])$ as a proof obligation, which is not valid. Since $min(A[1\mathinner{..}1])$ simplifies to $A[1]$, we modify the initialisation to $lo, k := A[1], 1$. Note that the 0 assigned to k was changed to 1 in the preceding step, but the 0 assigned to lo was not.

$$(S_3) \quad \sqsubseteq \quad (\textit{Introduce Assignment})$$

$$lo, k := A[1], 1$$

- $S_4 \;\lhd\; \textit{IntroDO}(1 \leq k \leq n \wedge lo = min(A[1\mathinner{..}k]), k \neq n, n - k) \;\;\rhd\;\; \mathbf{do}\ k \neq n \rightarrow S_5\ \mathbf{od}$

  This rule can be applied as is. The earlier modifications have made the necessary adjustments to the loop invariant, and the same variant can be used.

$$(S_4) \quad \sqsubseteq \quad (\textit{Introduce DO})$$

$$\mathbf{do}\ k \neq n \rightarrow$$

$$lo, k\colon \left[\begin{array}{c} 1 \leq k < n \\ lo = min(A[1\mathinner{..}k]) \end{array} \;\middle/\; \begin{array}{c} 1 \leq k \leq n \\ lo = min(A[1\mathinner{..}k]) \\ k_0 < k \end{array} \right] \qquad (S_5)$$

$$\mathbf{od}$$

- $S_5 \;\lhd\; \textit{FollAssign}(k := k + 1) \;\;\rhd\;\; S_6 \; ; \; \_$

  This rule can also be applied unchanged.

$$(S_5) \quad \sqsubseteq \quad (\textit{Following Assignment})$$

$$lo\colon \left[1 \leq k < n \wedge lo = min(A[1\mathinner{..}k]) \;\middle/\; lo = min(A[1\mathinner{..}k+1]) \right] ; \qquad (S_6)$$

$$k := k + 1$$

- $S_6 \;\lhd\; \textit{IntroAssign}(lo := lo + A[k + 1])$

  This step needs to be modified, because a the assignment doesn't establish the postcondition $lo = min(A[1\mathinner{..}k{+}1])$. We observe that $min(A[1\mathinner{..}k{+}1])$ is equal to $min(min(A[1\mathinner{..}k]), A[k{+}1])$, which is the same as $min(lo, A[k + 1])$ since we know that $lo = min(A[1\mathinner{..}k])$. We should thus be able to reuse our earlier derivation for finding the minimum of two numbers. We delete the above step from the script window, load the script for finding the minimum of two numbers in its place, and proceed to consider its steps. The labels used will be automatically adjusted so as not to conflict with ones in preceding steps — we will rename them as T_1, etc.

- $T_1 = m\colon \left[\mathit{true} \;\middle/\; m = min(x, y) \right]$

  We want to apply the steps in this script to S_6, so we replace T_1 by S_6. We also need to replace true by $1 \leq k < n \wedge lo = min(A[1\mathinner{..}k])$ in the precondition, m by lo, $min(x, y)$ by $min(A[1\mathinner{..}k + 1])$, x by lo and y by $A[k + 1]$. Thus, we make the following replacements throughout the script:

$$
\begin{array}{lcl}
T_1 & \longrightarrow & S_6 \\
\mathit{true} & \longrightarrow & 1 \le k < n \wedge lo = \mathit{min}(A[1\,..\,k]) \\
m & \longrightarrow & lo \\
\mathit{min}(x, y) & \longrightarrow & \mathit{min}(A[1\,..\,k+1]) \\
x & \longrightarrow & lo \\
y & \longrightarrow & A[k+1]
\end{array}
$$

- $S_6 \ \lhd \ \mathit{IntroIF}(lo \le A[k+1], lo \ge A[k+1]) \ \rhd$ **if** $\_ \rightarrow T_2 \ [\!] \ \_ \rightarrow T_3$ **fi**

  The necessary adjustments have been made, so this step can now be applied as is.

$$
\begin{array}{lll}
(S_6) & \sqsubseteq & (\mathit{Introduce\ IF}) \\
& & \textbf{if} \ \ lo \le A[k+1] \rightarrow \\
& & \quad lo\colon \left[\begin{array}{c} 1 \le k < n \\ lo = \mathit{min}(A[1\,..\,k]) \\ lo \le A[k+1] \end{array} \Bigg/ \ lo = \mathit{min}(A[1\,..\,k+1]) \right] \qquad (T_2) \\
& & [\!] \ \ lo \ge A[k+1] \rightarrow \\
& & \quad lo\colon \left[\begin{array}{c} 1 \le k < n \\ lo = \mathit{min}(A[1\,..\,k]) \\ lo \ge A[k+1] \end{array} \Bigg/ \ lo = \mathit{min}(A[1\,..\,k+1]) \right] \qquad (T_3) \\
& & \textbf{fi}
\end{array}
$$

- $T_2 \ \lhd \ \mathit{IntroAssign}(lo := lo)$

  This step can be applied as is, but since the assignment is not necessary, we replace it by $T_2 \ \lhd \ \mathit{IntroSkip}$.

$$
\begin{array}{lll}
(T_2) & \sqsubseteq & (\mathit{Introduce\ Skip}) \\
& & \textbf{skip}
\end{array}
$$

- $T_3 \ \lhd \ \mathit{IntroAssign}(lo := A[k+1])$

  This step can be applied as is.

$$
\begin{array}{lll}
(T_3) & \sqsubseteq & (\mathit{Introduce\ Assignment}) \\
& & lo := A[k+1]
\end{array}
$$

This completes the derivation. The resulting program is:

$$
\begin{array}{l}
[\![\ \textbf{var} \ k \bullet \\
\quad lo, k := A[1], 1; \\
\quad \textbf{do} \ k \ne n \rightarrow \\
\qquad \textbf{if} \ \ lo \le A[k+1] \rightarrow \textbf{skip} \\
\qquad [\!] \ \ lo \ge A[k+1] \rightarrow lo := A[k+1] \\
\qquad \textbf{fi}; \\
\qquad k := k+1 \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

The resulting script, which can now be saved, is:

$$
\begin{array}{l}
S_1 = lo\colon \left[\, n \geq 1 \;\middle/\; lo = min(A[1\mathinner{..}n]) \,\right] \\
S_1 \;\lhd\; \mathit{IntroVar}(k) \;\;\rhd\; [\![\ \mathbf{var}\ k \bullet S_2\ ]\!] \\
S_2 \;\lhd\; \mathit{SplitSpec}(1 \leq k \leq n \wedge lo = min(A[1\mathinner{..}k])) \;\;\rhd\; S_3\ ;\ S_4 \\
S_3 \;\lhd\; \mathit{IntroAssign}(lo, k := A[1], 1) \\
S_4 \;\lhd\; \mathit{IntroDO}(1 \leq k \leq n \wedge lo = min(A[1\mathinner{..}k]), k \neq n, n-k) \;\;\rhd\; \mathbf{do}\ k \neq n \rightarrow S_5\ \mathbf{od} \\
S_5 \;\lhd\; \mathit{FollAssign}(k := k+1) \;\;\rhd\; S_6\ ;\ \_ \\
S_6 \;\lhd\; \mathit{IntroIF}(lo \leq A[k+1], lo \geq A[k+1]) \;\;\rhd\; \mathbf{if}\ \ \_ \rightarrow T_2\ [\!]\ \ \_ \rightarrow T_3\ \mathbf{fi} \\
T_2 \;\lhd\; \mathit{IntroSkip} \\
T_3 \;\lhd\; \mathit{IntroAssign}(lo := A[k+1])
\end{array}
$$

We have been able to derive a program to find the minimum of an array by reusing two existing derivations and making fairly minor changes to them. The largest changes required occur when we start applying an existing script, since we have to make replacements to make the initial pattern of the script match the current subject and propagate these changes through the rest of the script. The additional proof obligations were quite simple, and often provided useful guidance for determining what modifications were required.

8 Example: Maximum and minimum of an array

Now, suppose we wish to find the maximum as well as the minimum element of $A[1\mathinner{..}n]$. We can achieve this by adapting the above derivation for finding the minimum of an array.

We again begin with a modified specification in the program window:

$$lo, hi\colon \left[\, n \geq 1 \;\middle/\; lo = min(A[1\mathinner{..}n]) \wedge hi = max(A[1\mathinner{..}n]) \,\right] \qquad (S_1)$$

We load the array minimum script into the script window, and attempt to apply its steps to the new specification.

- $S_1 = lo\colon \left[\, n \geq 1 \;\middle/\; lo = min(A[1\mathinner{..}n]) \,\right]$

  Again, the initial pattern does not match the modified specification. To obtain a match, we replace lo by lo, hi where it occurs in a frame and $lo = min(A[1\mathinner{..}n])$ by $lo = min(A[1\mathinner{..}n]) \wedge hi = max(A[1\mathinner{..}n])$.

- $S_1 \;\lhd\; \mathit{IntroVar}(k) \;\;\rhd\; [\![\ \mathbf{var}\ k \bullet S_2\ ]\!]$

  This step can be applied as is, since we will still need just one local variable.

$$
\begin{array}{lll}
(S_1) & \sqsubseteq & (\mathit{Introduce\ Local\ Variables}) \\
 & & \mathbf{var}\ k \bullet \\
 & & lo, hi, k\colon \left[\, n \geq 1 \;\middle/\; lo = min(A[1\mathinner{..}n]) \wedge hi = max(A[1\mathinner{..}n]) \,\right] \qquad (S_2)
\end{array}
$$

- $S_2 \ \lhd \ \mathit{SplitSpec}(1 \leq k \leq n \wedge lo = min(A[1 \mathinner{..} k]))) \ \rhd \ S_3 \ ; \ S_4$

  This step could be applied as is, but we would end up with an algorithm that computed the minimum in one loop and the maximum in another. In order to compute minimum and maximum in the same loop, we need to add the conjunct $hi = max(A[1 \mathinner{..} k])$ to the intermediate assertion[2].

$$
\begin{array}{lll}
(S_2) & \sqsubseteq & (\mathit{Split\ Specification}) \\
& & lo, hi, k \colon \left[n \geq 1 \Big/ \begin{array}{c} 1 \leq k \leq n \\ lo = min(A[1 \mathinner{..} k]) \\ hi = max(A[1 \mathinner{..} k]) \end{array} \right]; \qquad (S_3) \\
& & lo, hi, k \colon \left[\begin{array}{c} 1 \leq k \leq n \\ lo = min(A[1 \mathinner{..} k]) \\ hi = max(A[1 \mathinner{..} k]) \end{array} \Big/ \begin{array}{c} lo = min(A[1 \mathinner{..} n]) \\ hi = max(A[1 \mathinner{..} n]) \end{array} \right] \qquad (S_4)
\end{array}
$$

- $S_3 \ \lhd \ \mathit{IntroAssign}(lo, k := A[1], 1)$

  This step will now not succeed, since the resulting statement will not establish the conjunct $hi = max(A[1 \mathinner{..} k])$. Since $max(A[1 \mathinner{..} k])$ reduces to $A[1]$ when $k = 1$, we modify the step to also assign $A[1]$ to hi.

$$
\begin{array}{lll}
(S_3) & \sqsubseteq & (\mathit{Introduce\ Assignment}) \\
& & lo, hi, k := A[1], A[1], 1
\end{array}
$$

- $S_4 \lhd \mathit{IntroDO}(1 \leq k \leq n \wedge lo = min(A[1 \mathinner{..} k]) \wedge hi = max(A[1 \mathinner{..} k]), k \neq n, n-k) \ \rhd \ \mathbf{do}\ k \neq n \rightarrow S_5\ \mathbf{od}$

  This step succeeds as is. The additional proof obligations are again trivial.

$$
\begin{array}{lll}
(S_4) & \sqsubseteq & (\mathit{Introduce\ DO}) \\
& & \mathbf{do}\ k \neq n \rightarrow \\
& & \quad lo, hi, k \colon \left[\begin{array}{c} 1 \leq k < n \\ lo = min(A[1 \mathinner{..} k]) \\ hi = max(A[1 \mathinner{..} k] \end{array} \Big/ \begin{array}{c} 1 \leq k \leq n \\ lo = min(A[1 \mathinner{..} k]) \\ hi = max(A[1 \mathinner{..} k]) \\ k_0 < k \end{array} \right] \qquad (S_5) \\
& & \mathbf{od}
\end{array}
$$

- $S_5 \ \lhd \ \mathit{FollAssign}(k := k + 1) \ \rhd \ S_6 \ ; \ \_$

  This step also succeeds as is.

$$
\begin{array}{lll}
(S_5) & \sqsubseteq & (\mathit{Following\ Assignment}) \\
& & lo, hi \colon \left[\begin{array}{c} 1 \leq k < n \\ lo = min(A[1 \mathinner{..} k]) \\ hi = max(A[1 \mathinner{..} k]) \end{array} \Big/ \begin{array}{c} lo = min(A[1 \mathinner{..} k+1]) \\ hi = max(A[1 \mathinner{..} k+1]) \end{array} \right]; \qquad (S_6) \\
& & k := k + 1
\end{array}
$$

[2] A more flexible matching and substitution facility that we're assuming might have already performed this replacement.

- $S_6 \lhd \mathit{IntroIF}(lo \leq A[k+1], lo \geq A[k+1]) \ \rhd$ **if** $\_ \rightarrow T_2 \ [\!] \ \_ \rightarrow T_3$ **fi**

  This step could be applied as is, but would result in an algorithm where the code to update hi was repeated in each branch of the **if** statement. To avoid this, we need to modify this step in some way. We could split S_6 into two specification statements, one of establish $lo = min(A[1\mathinner{..}k])$ and one top establish $hi = max(A[1\mathinner{..}k])$, which would both be refined to **if** statements. Alternatively, we could use an **if** statement with four branches, corresponding to the four possible combinations of truth values of $lo \geq A[k+1]$ and $hi \leq A[k+1]$. Since $lo \leq hi$ always holds, we can reduce this to three branches with simpler guards: $lo \leq A[k+1] \leq hi$, $lo \geq A[k+1]$ and $hi \leq A[k+1]$. The resulting step is:

  $S_6 \lhd \mathit{IntroIF}(lo \leq A[k+1] \leq hi, lo \geq A[k+1], hi \leq A[k+1]) \ \rhd$ **if** $\_ \rightarrow T_2 \ [\!] \ \_ \rightarrow T_3 \ [\!] \ \_ \rightarrow T_4$ **fi**

$$
\begin{array}{lll}
(S_6) & \sqsubseteq & (\mathit{Introduce\ IF}) \\
& & \textbf{if} \ \ lo \leq A[k+1] \leq hi \rightarrow \\
& & \qquad lo, hi\colon \left[\begin{array}{c} 1 \leq k < n \\ lo = min(A[1\mathinner{..}k]) \\ hi = max(A[1\mathinner{..}k]) \\ lo \leq A[k+1] \wedge A[k+1] \leq hi \end{array} \middle/ \begin{array}{c} lo = min(A[1\mathinner{..}k+1]) \\ hi = max(A[1\mathinner{..}k+1]) \end{array} \right] \qquad (T_2) \\
& & [\!] \ \ lo \geq A[k+1] \rightarrow \\
& & \qquad lo, hi\colon \left[\begin{array}{c} 1 \leq k < n \\ lo = min(A[1\mathinner{..}k]) \\ hi = max(A[1\mathinner{..}k]) \\ lo \geq A[k+1] \end{array} \middle/ \begin{array}{c} lo = min(A[1\mathinner{..}k+1]) \\ hi = max(A[1\mathinner{..}k+1]) \end{array} \right] \qquad (T_3) \\
& & [\!] \ \ hi \leq A[k+1] \rightarrow \\
& & \qquad lo, hi\colon \left[\begin{array}{c} 1 \leq k < n \\ lo = min(A[1\mathinner{..}k]) \\ hi = max(A[1\mathinner{..}k]) \\ hi \leq A[k+1] \end{array} \middle/ \begin{array}{c} lo = min(A[1\mathinner{..}k+1]) \\ hi = max(A[1\mathinner{..}k+1]) \end{array} \right] \qquad (T_4) \\
& & \textbf{fi}
\end{array}
$$

- $T_2 \lhd \mathit{IntroSkip}$

  This step succeeds, though there is a more complex proof obligation.

$$
\begin{array}{lll}
(T_2) & \sqsubseteq & (\mathit{Introduce\ Skip}) \\
& & \textbf{skip}
\end{array}
$$

- $T_3 \lhd \mathit{IntroAssign}(lo := A[k+1])$

  This step also succeeds, again with a more complex proof obligation.

$$
\begin{array}{lll}
(T_3) & \sqsubseteq & (\mathit{Introduce\ Assignment}) \\
& & lo := A[k+1]
\end{array}
$$

- There are no further steps available in the script window, so we introduce a new step to implement the remaining specification — perhaps noting the similarity with the previous step.

  $T_4 \lhd \mathit{IntroAssign}(hi := A[k+1])$

This completes this adaptation of the derivation. The resulting program is:

$$
\begin{array}{l}
[\![\ \textbf{var}\ k \bullet \\
\quad lo, hi, k := A[1], A[1], 1; \\
\quad \textbf{do}\ k \neq n \rightarrow \\
\qquad \textbf{if} \quad lo \leq A[k+1] \leq hi \rightarrow \textbf{skip} \\
\qquad [\!] \quad lo \geq A[k+1] \rightarrow lo := A[k+1] \\
\qquad [\!] \quad hi \leq A[k+1] \rightarrow hi := A[k+1] \\
\qquad \textbf{fi}; \\
\qquad k := k+1 \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

The resulting derivation script is now:

$$
\begin{array}{l}
S_1 = lo, hi\colon \left[n \geq 1 \ \middle/ \ lo = min(A[1\,..\,n]) \wedge hi = max(A[1\,..\,n]) \right] \\
S_1 \ \lhd \ IntroVar(k) \ \rhd \ [\![\ \textbf{var}\ k \bullet S_2 \ ]\!] \\
S_2 \ \lhd \ SplitSpec(1 \leq k \leq n \wedge lo = min(A[1\,..\,k]) \wedge hi = max(A[1\,..\,k])) \ \rhd \ S_3 \ ; \ S_4 \\
S_3 \ \lhd \ IntroAssign(lo, hi, k := A[1], A[1], 1) \\
S_4 \ \lhd \ IntroDO(1 \leq k \leq n \wedge lo = min(A[1\,..\,k]) \wedge hi = max(A[1\,..\,n]), k \neq n, n-k) \ \rhd \\
\qquad \textbf{do}\ k \neq n \rightarrow S_5\ \textbf{od} \\
S_5 \ \lhd \ FollAssign(k := k+1) \ \rhd \ S_6 \ ; \ \_ \\
S_6 \ \lhd \ IntroIF(lo \leq A[k+1] \leq hi, lo \geq A[k+1], hi \leq A[k+1]) \ \rhd \\
\qquad \textbf{if} \ \ \_ \rightarrow T_2 \ [\!] \ \ \_ \rightarrow T_3 \ [\!] \ \ \_ \rightarrow T_4 \ \textbf{fi} \\
T_2 \ \lhd \ IntroSkip \\
T_3 \ \lhd \ IntroAssign(lo := A[k+1]) \\
T_4 \ \lhd \ IntroAssign(hi := A[k+1])
\end{array}
$$

Again, we have been able to adapt an existing derivation, this time to address an additional requirement, with little effort. Again, the additional proof obligations have been quite simple, and have provided useful guidance in adapting the derivation.

9 Discussion

In this paper, we have attempted to show how a refinement tool might be designed to support more flexible styles of program development than the strict top-down refinement — in particular program construction by incremental enhancement, and program reuse. We have described an architecture for such a tool, based on the idea of an editable derivation script, and explained how it would be used in some example developments. Although conceptually quite simple, this architecture appears to provide a promising basis for adapting formal derivations.

We have implemented a simple tool to apply derivation scripts to specifications, but have not yet implemented most of the adaptation facilities described. We have also begun a formal specification of the proposed tool.

Our main concern here has been to understand the kinds of modifications that are required when adapting a derivation. There is considerable scope for automating parts of this process, but we must understand clearly want we want to be able to do before we can sensibly decide what parts can most usefully be automated. We are thus content to begin with very basic, well understood, facilities, and gain experience with those before contemplating more ambitious designs.

Performing transformations on derivation scripts also provides alternative ways of looking at a number of other programming activities and raises a number of additional issues. We could, for example, perform functional data refinements by replacing abstract variables by their concrete representations in the derivation script. We could also treat the introduction of procedures as transformations on the derivation. This then raises the issue of whether these transformations should be considered as part of the development history of the program, or whether they are treated as incidental steps that should be discarded.

References

[Carrington, *et al*] D. Carrington, I. Hayes, R. Nickson, G. Watson and J. Welsh. *A Review of Existing Refinement Tools.* Technical Report 94-8, Software Verification Research Centre, The University of Queensland, 1994.

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

[Morgan, Robinson and Gardiner, 1988] Carroll Morgan, Ken Robinson and Paul Gardiner. *On the Refinement Calculus.* Technical Monograph PRG-70, Oxford University, 1988.

[Naftalin, 1992] Maurice Naftalin. "A Model of the Refinement Process". *Proc. 5th Refinement Workshop*, C. B. Jones, R. C. Shaw and T. Denver (Eds.), Springer Verlag, 1992, pp 211–229.

[Nickson, 1994] Ray Nickson. *Tool Support for the Refinement Calculus.* PhD thesis, Victoria University of Wellington, 1994.

[Vadera, 1994] Sunil Vadera. "Proof by Analogy in mural". *Formal Aspects of Computing* **3** (1994), pp 1–25.

[Vickers, 1990] Trevor Vickers. "An overview of a refinement editor". *Proc. Fifth Australian Software Engineering Conference*, 1990.

[Vickers and Gardiner, 1994] Trevor Vickers and Paul Gardiner. *A Language of Refinements.* Technical Report TR-CS-95-04, Australian National University, May 1994.

[Wirth, 1971] N. Wirth. "Program development by stepwise refinement". *CACM* **14** (April 1971), pp 221–227.

[Worden, 1991] Robert Worden. "The Process of Refinement". *Proc. 4th Refinement Workshop*, J. M. Morris and R.C. Shaw (Eds.), Springer Verlag, 1991, pp 1–5.