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

Deriving Programs by Combining and Adapting Refinement Scripts

Lindsay Groves*

Technical Report CS-TR-95/13
June 1995

Abstract

Although program refinement is usually presented as a top-down process, real programs are usually constructed by extending, adapting and combining existing programs. We show how this kind of program development can be performed using editable refinement scripts, which can be extended, adapted and combined in this way. Our approach is illustrated by a sequence of example, beginning with a list insertion algorithm and culminating in a stable sorting algorithm.

Publishing Information

Submitted to 1995 Asia-Pacific Software Engineering Conference.

*Email: `lindsay@comp.vuw.ac.nz`.

Deriving Programs by Combining and Adapting Refinement Scripts

Lindsay Groves

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

Abstract

Although program refinement is usually presented as a top-down process, real programs are usually constructed by extending, adapting and combining existing programs. We show how this kind of program development can be performed using editable refinement scripts, which can be extended, adapted and combined in this way. Our approach is illustrated by a sequence of example, beginning with a list insertion algorithm and culminating in a stable sorting algorithm.

1 Introduction

There has been considerable interest in recent years in constructing verified software by deriving programs from formal specifications by correctness preserving transformations, rather than constructing programs by conventional, informal methods and then attempting to verify them as a separate activity. The refinement calculus ([Back, 1981], [Back, 1988], [Morgan, 1990], [Morris, 1987], [Morgan, Robinson and Gardiner, 1988]) provides a sound basis for this kind of formal development. Practical application of refinement techniques, however, requires tool support to manage derivations and perform various formula manipulation tasks which are required.

Program refinement is usually presented as a top-down process in which programs are constructed from scratch, in a strictly top-down fashion, and existing refinement tools best support top-down development ([Carrington, *et al*]). While top-down development is an elegant way to present program derivations, real programs are not normally constructed in this way. We often begin by constructing a program that does part of what we want (a prototype) and then extend it. We also reuse and adapt existing programs or program fragments where possible, to avoid repeating work we have already done.

It seems reasonable to expect similar development patterns to occur in formal software development, and to develop suitable tools to support them. We are thus investigating techniques that will allow us to formally derive a program by first deriving part of the required program and then extending this to obtain the final program, perhaps via several steps, making use of existing derivations where possible. Our proposed approach is based on the use of editable refinement scripts which provide descriptions of refinement steps in a form that can be readily extended, adapted and combined.

In this paper, we discuss the use of refinement scripts in constructing formal derivations of programs. We begin with a brief introduction to the refinement calculus (Section 2), which we illustrate by deriving an algorithm to insert an element into an ordered list (Section 3). We then introduce our script notation (Section 4), and show how this can be used to describe the list insertion derivation and also a generic derivation for a counting-up loop (Section 5). We then show how these scripts can be combined, after minor adjustments, to produce a derivation for an insertion sort algorithm (Section 6) and how this can then be modified to produce a stable sort which sorts on keys rather than whole items (Section 7). Finally, we present our conclusions (Section 8).

2 The Refinement Calculus

The refinement calculus is a formalism for deriving imperative programs from logical specifications. Programs are derived in a wide-spectrum language based on Dijkstra's guarded commands language [Dijkstra, 1976], augmented with constructs for expressing specifications, which are are (in general) non-executable. Most notably, we can write a specification statement

$$w\colon \left[\, Pre \;\middle/\; Post \,\right]$$

which specifies a program that will make the postcondition *Post* true, altering only variables in w (known as the *frame*), provided that the precondition *Pre* is true initially.

For example, the following statement specifies a program that will decrease the value of x by either one or two, provided that x is initially greater than zero:

$$x\colon \left[\, 0 < x \;\middle/\; x_0 < x \leq x_0 + 2 \,\right]$$

Note that zero-subscripted variables, such as x_0, are used to refer to the initial values of variables within the postcondition.

Programs are constructed in a number of steps, by performing correctness preserving transformations. Each transformation is required to preserve the refinement relation between programs, which is defined so that S is refined by P, written $S \sqsubseteq P$, if every implementation of P is a correct implementation of S. This is defined formally in terms of weakest preconditions, as:

$$S \sqsubseteq P \text{ iff for every postcondition } R,\ wp(S,R) \Rightarrow wp(P,R)$$

This relation is *transitive*, i.e. if $S \sqsubseteq S'$ and $S' \sqsubseteq S''$, then $S \sqsubseteq S''$. This means that we can construct a program via a sequence of refinement steps, and be assured that the final program is a refinement of the initial one.

Also, the constructs in the wide-spectrum language are *monotonic* with respect to refinement, i.e. if $S \sqsubseteq S'$, then $\mathcal{F}[S] \sqsubseteq \mathcal{F}[S']$, where $\mathcal{F}[S]$ is a program with S as a component and $\mathcal{F}[S']$ is the result of replacing the designated occurrence of S in $\mathcal{F}[S]$ by S'. This means that we can refine components of a program independently, and be assured that the result will also be a refinement of the whole program.

To avoid having to reason about weakest preconditions directly, we normally justify refinement steps using *refinement rules* which embody laws showing that certain refinements are always valid, perhaps subject to certain applicability conditions or proof obligations.

A small selection of refinement rules is shown below. These are the only rules used in this paper, and several are in simplified/specialised form. A more comprehensive collection of rules (and more general forms of some of these rules) can be found in [Morgan, 1990]. We write refinement rules as inference rules with the proof obligations (if any) above the line, and the inferred refinement below. Some of these conditions are purely syntactic (e.g. x does not occur in w) and can be checked mechanically; others are logical conditions requiring proof.

Introduce Local Variable:

$$\frac{x \text{ does not occur in } w}{w\colon \left[P \;\middle/\; R \right] \sqsubseteq [\![\;\mathbf{var}\;\; x \bullet x, w\colon \left[P \;\middle/\; R \right]]\!]}$$

Split Specification:

$$w\colon \left[P \;\middle/\; R \right] \sqsubseteq w\colon \left[P \;\middle/\; Q \right] ;\; w\colon \left[Q \;\middle/\; R \right]$$

Introduce Assignment:

$$\frac{P \Rightarrow R[x \leftarrow e]}{x, w\colon \left[P \;\middle/\; R \right] \sqsubseteq x := e}$$

where $R[x \leftarrow e]$ is the formula obtained by replacing all free occurrences of x in R by e.

Leading Assignment:

$$\frac{x \text{ does not occur in } e \text{ or } P}{x, w\colon \left[P \;\middle/\; R \right] \sqsubseteq x := e \;;\; w\colon \left[P \wedge x = e \;\middle/\; R \right]}$$

Following Assignment:

$$x, w\colon \left[P \;\middle/\; R \right] \sqsubseteq w\colon \left[P \;\middle/\; R[x \leftarrow e] \right] ;\; x := e$$

Introduce Do:

$$\frac{\begin{array}{c} P \Rightarrow I \\ I \wedge \neg B \Rightarrow R \end{array}}{w\colon \left[P \;\middle/\; R \right] \sqsubseteq \mathbf{do}\;\; B \rightarrow w\colon \left[I \wedge B \;\middle/\; I \wedge 0 \leq t < t_0 \right] \;\mathbf{od}}$$

where t_0 is obtained from t by replacing each variable x in w by x_0.

These rules are actually schemas, describing (usually infinite) sets of theorems about refinements. When we apply a refinement rule whose bottom line is *before* $\sqsubseteq$ *after* to a selected program component *prog*, we must instantiate the schema variables in the rule so that the resulting instance of *before* matches *prog*, and prove the corresponding instances of the applicability conditions. The schema variables which do not occur in *before* are called *parameters* of the rule, and may be chosen freely, so long as the applicability conditions hold. In *Introduce Do*, I is the loop invariant, B is the loop guard and t is the variant function. In *Introduce Assignment*, *Leading Assignment* and *Following Assignment*, we take the whole assignment ($x := e$) as a single parameter, rather than treating x and e as separate parameters.

3 Example: Insertion in an Ordered List

We illustrate the use of the refinement calculus by deriving an algorithm to insert an element into an ordered list, stored as an array segment. This derivation is also used as the basis for later examples in Sections 4, 6 and 7.

In presenting derivations we explicitly label every specification statement (usually $S_1, S_2, \cdots$) and use these labels to indicate the statement being refined. Also for every step, we give the name of the refinement rule and the parameters used — this makes the correspondence with steps described in our script notation clear (see Section 4).

We wish to insert an element x into an ordered list, in such a way that the ordering is preserved; x and the list elements will be assumed to be integers, though they could equally be of any type with an ordering $\leq$ defined. The initial list is stored as an array segment $a(1 \mathinner{..} k)$, which is assumed to be arranged in (non-strictly) ascending order, and k is assumed to be less than the size of a (written $\#a$). The resulting list, which will be stored in $a(1 \mathinner{..} k+1)$, must contain all of the elements of the initial list along with x, and be arranged in ascending order.

To avoid including it in all later conditions, we assume $0 \leq k < \#a$ to hold globally, just as we make assumptions about the types of a and x, and assume various properties of numbers, arrays, etc. We thus write our specification as:

$$a\colon \left[\begin{array}{l} \mathbf{asc}\, a(1 \mathinner{..} k) \\ a(1 \mathinner{..} k) = b \end{array} \Big/ \begin{array}{c} \mathbf{asc}\, a(1 \mathinner{..} k+1) \\ a(1 \mathinner{..} k+1) \approx b \frown \langle x \rangle \end{array} \right] \tag{S_1}$$

where $\mathbf{asc}\,\alpha$ means that the elements of α are in ascending order, $\alpha \approx \beta$ means that α is a permutation of β, $b \frown \langle x \rangle$ is the array formed by concatenating x onto the end of b, and b is a logical constant used to refer to the initial value of $a(1 \mathinner{..} k)$ in the postcondition.

In order to establish $a(1 \mathinner{..} k+1) \approx b \frown \langle x \rangle$, given $b = a(1 \mathinner{..} k)$ initially, we must clearly store x in some element of a, possibly rearranging the original contents of a. To also establish $\mathbf{asc}\, a(1 \mathinner{..} k+1)$, this must be a location, i say, such that all preceding elements are less than or equal to x (written $a(1 \mathinner{..} i-1) \leq x$) and all subsequent elements are greater than or equal to x (written $x \leq a(i+1 \mathinner{..} k+1)$). We first introduce a local variable i which will be used to locate the required position for x.

$$(S_1) \quad \sqsubseteq \quad (\,\textit{Introduce Local Variable}\colon i\,)$$

$$\begin{array}{l} \mathbf{var}\ i \bullet \\ i, a\colon \left[\begin{array}{l} \mathbf{asc}\, a(1 \mathinner{..} k) \\ a(1 \mathinner{..} k) = b \end{array} \Big/ \begin{array}{c} \mathbf{asc}\, a(1 \mathinner{..} k+1) \\ a(1 \mathinner{..} k+1) \approx b \frown \langle x \rangle \end{array} \right] \end{array} \tag{S_2}$$

We now split S_2 into two parts: one to find the required position for x and create a space for it, and one to insert x in the space thus created. The first part will need to establish $a(1 \mathinner{..} i-1) \leq x \leq a(i+1 \mathinner{..} k+1)$, where $1 \leq i \leq k+1$, while ensuring that the array segment $a(1 \mathinner{..} k+1)$ with the ith element deleted, which we write as $a(1 \mathinner{..} k+1 \backslash i)$, remains in ascending order and contains the same elements as b (i.e. the original contents of $a(1 \mathinner{..} k)$).

$$(S_2) \quad \sqsubseteq \quad \left(\textit{Split Spec}\colon \begin{array}{c} 1 \le i \le k+1 \wedge \mathbf{asc}\, a(1\,..\,k+1\backslash i) \wedge \\ a(1\,..\,k+1\backslash i) \approx b \wedge \\ a(1\,..\,i-1) \le x \le a(i+1\,..\,k+1) \end{array} \right)$$

$$i, a\colon \left[\begin{array}{l} \mathbf{asc}\, a(1\,..\,k) \\ a(1\,..\,k) = b \end{array} \Big/ \begin{array}{c} 1 \le i \le k+1 \\ \mathbf{asc}\, a(1\,..\,k+1\backslash i) \\ a(1\,..\,k+1\backslash i) \approx b \\ a(1\,..\,i-1) \le x \le a(i+1\,..\,k+1) \end{array} \right]; \qquad (S_3)$$

$$i, a\colon \left[\begin{array}{c} 1 \le i \le k+1 \\ \mathbf{asc}\, a(1\,..\,k+1\backslash i) \\ a(1\,..\,k+1\backslash i) \approx b \\ a(1\,..\,i-1) \le x \le a(i+1\,..\,k+1) \end{array} \Big/ \begin{array}{c} \mathbf{asc}\, a(1\,..\,k+1) \\ a(1\,..\,k+1) \approx b \frown \langle x \rangle \end{array} \right] \qquad (S_4)$$

We can establish all of the postcondition of S_3, except for $a(1\,..\,i-1) \le x$, by setting i to $k+1$. This suggests introducing a loop which maintains $\mathbf{asc}\, a(1\,..\,k+1\backslash i) \wedge a(1\,..\,k+1\backslash i) \approx b \wedge x \le a(i+1\,..\,k+1)$, while rearranging elements of a so that $a(1\,..\,i-1) \le x$ also holds.

We thus split S_3 into two parts: one to initialise i, the other will become the loop. The intermediate assertion is the loop invariant, which is the condition mentioned above along with the constraint on i.

$$(S_3) \quad \sqsubseteq \quad \left(\textit{Split Spec}\colon\ \textit{Inv} \mathrel{\widehat{=}} \begin{array}{l} 1 \le i \le k+1 \wedge \mathbf{asc}\, a(1\,..\,k+1\backslash i) \wedge \\ a(1\,..\,k+1\backslash i) \approx b \wedge x \le a(i+1\,..\,k+1) \end{array} \right)$$

$$i, a\colon \left[\begin{array}{l} \mathbf{asc}\, a(1\,..\,k) \\ a(1\,..\,k) = b \end{array} \Big/ \textit{Inv} \right]; \qquad (S_5)$$

$$i, a\colon \left[\textit{Inv} \Big/ \begin{array}{c} 1 \le i \le k+1 \\ \mathbf{asc}\, a(1\,..\,k+1\backslash i) \\ a(1\,..\,k+1\backslash i) \approx b \\ a(1\,..\,i-1) \le x \le a(i+1\,..\,k+1) \end{array} \right] \qquad (S_6)$$

The loop invariant is named to avoid writing it out in full each time it is mentioned; we treat this as an abbreviation in the meta-language.

We have already decided that S_5 should set i to $k+1$:

$$(S_5) \quad \sqsubseteq \quad (\ \textit{Introduce Assignment}\colon\ i := k+1\ )$$
$$i := k+1$$

This step has the following proof obligation, which we already know to hold:

$$\begin{array}{l} \mathbf{asc}\, a(1\,..\,k) \wedge a(1\,..\,k) = b \\ \Rightarrow\ 1 \le k+1 \le k+1 \wedge \mathbf{asc}\, a(1\,..\,k+1\backslash k+1) \wedge \\ \quad a(1\,..\,k+1\backslash k+1) \approx b \wedge x \le a(k+1+1\,..\,k+1) \end{array}$$

We now introduce the loop, giving the loop invariant ($\textit{Inv}$), guard and variant as parameters. The loop has to stop when $a(1\,..\,i-1) \le x$. Since we already know that $\mathbf{asc}\, a(1\,..\,i-1)$ holds, $a(1\,..\,i-1) \le x$ will also hold when $i \le 1$ or $a(i-1) < x$, which is easier to test. Thus, we take the negation of this, $i > 1 \wedge x \leq a(i-1)$, as the guard.[1] Anticipating that i will be decreased in the loop, we take i as the variant.

$$
\begin{array}{lll}
(S_6) & \sqsubseteq & (\ \textit{Introduce DO}: \textit{Inv}, i > 1 \wedge x \leq a(i-1), i\ ) \\
 & & \textbf{do}\ i > 1 \wedge x \leq a(i-1) \rightarrow \\
 & & \quad i, a: \left[\, \textit{Inv} \wedge i > 1 \wedge x \leq a(i-1) \;\middle/\; \textit{Inv} \wedge 0 \leq i < t_0 \,\right] \qquad (S_7) \\
 & & \textbf{od}
\end{array}
$$

This step has the following proof obligations:

$$
\textit{Inv} \Rightarrow \textit{Inv}
$$

$$
\begin{array}{l}
\textit{Inv} \wedge \neg(i > 1 \wedge x \leq a(i-1)) \\
\Rightarrow\ 1 \leq i \leq k+1 \wedge \textbf{asc}\, a(1\mathinner{..}k+1 \backslash i) \wedge a(1\mathinner{..}k+1 \backslash i) \approx b \wedge \\
\quad\ a(1\mathinner{..}i-1) \leq x \leq a(i+1\mathinner{..}k+1)
\end{array}
$$

The latter relies of the fact that $\textbf{asc}\, a(1\mathinner{..}k+1\backslash i) \wedge x \leq a(i+1\mathinner{..}k+1) \wedge \neg(i > 1 \wedge x \leq a(i-1))$ implies $\textbf{asc}\, a(1\mathinner{..}k+1)$. Again, the loop guard and invariant were chosen so that these conditions would hold and are easily verified.

We can ensure that the loop makes progress (i.e. the variant is reduced) by decrementing i. We use *Following Assignment* to introduce an assignment statement to decrement i and calculate a specification statement to ensure that the loop invariant is preserved.

$$
\begin{array}{lll}
(S_7) & \sqsubseteq & (\ \textit{Following Assignment}:\ i := i-1\ ) \\
 & & a: \left[\, \textit{Inv} \wedge i > 1 \wedge x \leq a(i-1) \;\middle/\; \textit{Inv}[i \leftarrow i-1] \wedge 0 \leq i-1 < i_0 \,\right]; \qquad (S_8) \\
 & & i := i-1
\end{array}
$$

Our version of *Following Assignment* removes the variable assigned from the frame of the preceding specification, so i does not appear in the frame of S_8. Since S_8 cannot modify i, we can remove $0 \leq i-1 < i_0$ from its postcondition, as this is implied by the precondition.

We can implement S_8 by moving $a(i-1)$ to $a(i)$.

$$
\begin{array}{lll}
(S_8) & \sqsubseteq & (\ \textit{Introduce Assignment}:\ a(i) := a(i-1)\ ) \\
 & & a(i) := a(i-1)
\end{array}
$$

Semantically, the assignment $a(i) := a(i-1)$ is treated as $a := a[i := a(i-1)]$, where $a[i := a(i-1)]$ is the array obtained from a by replacing the ith element by $a(i-1)$.

This step has the following proof obligation:

$$
\textit{Inv} \wedge i > 1 \wedge x \leq a(i-1) \Rightarrow \textit{Inv}[i \leftarrow i-1][a \leftarrow a[i := a(i-1)]]
$$

The significant parts of this, which are easily verified, are:

[1]The alert reader might notice that we could have used $x < a(i-1)$ here instead of $x \leq a(i-1)$; we will revisit this decision in Section 7.

$$a(1\mathinner{..}k+1\backslash i) \approx b \wedge i > 1 \Rightarrow a[i := a(i-1)](1\mathinner{..}k+1\backslash i-1) \approx b$$

$$\begin{array}{l}\mathbf{asc}\, a(1\mathinner{..}k+1\backslash i) \wedge x \leq a(i+1\mathinner{..}k+1) \wedge i > 1 \wedge x \leq a(i-1) \\ \Rightarrow \mathbf{asc}\, a[i := a(i-1)](1\mathinner{..}k+1\backslash i-1)\end{array}$$

$$\begin{array}{l}x \leq a(i+1\mathinner{..}k+1) \wedge i > 1 \wedge x \leq a(i-1) \\ \Rightarrow x \leq a[i := a(i-1)](i\mathinner{..}k+1)\end{array}$$

Finally, we implement S_4 by setting $a(i)$ to x.

$$\begin{array}{lll}(S_4) & \sqsubseteq & (\ \textit{Introduce Assignment}\text{: } a(i) := x\ ) \\ & & a(i) := x\end{array}$$

This step has the following proof obligation, taking $a(i) := x$ as $a := a[i := x]$, which is also easily verified:

$$\begin{array}{l}\mathbf{asc}\, a(1\mathinner{..}k+1\backslash i) \wedge a(1\mathinner{..}k+1\backslash i) \approx b \wedge a(1\mathinner{..}i-1) \leq x \leq a(i+1\mathinner{..}k+1) \\ \Rightarrow \mathbf{asc}\, a[i := x](1\mathinner{..}k+1) \wedge a[i := x](1\mathinner{..}k+1) \approx b \frown \langle x \rangle\end{array}$$

This completes the derivation; the resulting program is shown in Figure 1. Throughout the derivation, our choices have been guided by the specifications, and refinement rules and their parameters have been chosen so that the resulting proof obligations will hold.

$$\begin{array}{l}[\![\ \mathbf{var}\ i\ \bullet \\ \quad i := k+1; \\ \quad \mathbf{do}\ i > 1 \wedge x \leq a(i-1) \rightarrow \\ \qquad a(i) := a(i-1); \\ \qquad i := i-1 \\ \quad \mathbf{od}; \\ \quad a(i) := x; \\ ]\!];\end{array}$$

Figure 1: List insertion program

4 Describing Refinements

In order to support the kind of program development style outlined earlier, we need to be able to adapt and combine program derivations. For example, we should be able to use the above list insertion derivation in constructing an insertion sort algorithm or in implementing set operations using an ordered list representation for sets. To facilitate this, we introduce a notation for describing formal derivations which allows them to be adapted and combined in this way. The ability to package sequences of refinement steps so that they can be reused is also provided by derived rules in Red [Vickers, 1990] and by tactics in Ray Nickson's refinement tool, which we refer to as RRT [Nickson, 1994]. Our approach is more flexible in some ways, and should be regarded as complementary.

A program derivation is described by a *refinement script*, comprising a sequence of steps. Each step may apply a refinement rule (from a collection of named rules) to part of the program resulting from the preceding steps, or perform some other action such as introducing definitions, extracting and manipulating components of programs, etc. The main kinds of steps used in scripts are as follows:

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

  Describes a refinement rule application. *Subject* is a name identifying the program component to be transformed, *RuleName* is the name (in abbreviated form) of the rule to be applied, and *parameters* is a list of parameters to be used in instantiating the rule. Instantiation of parameters can be deferred by giving metavariables (starting with a capital letter) as the parameters; these can the be bound later.

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

  Describes a refinement rule application, as above, but includes a pattern to be matched with the result of the rule application. This allows names to be associated with (parts of) program components created by refinement rules. The syntax for patterns is the same as for programs, except that any component may be replaced by a metavariable to be associated with that component, much like the patterns used in refinement rules. Unnamed variables (denoted by underscore) may be used where a component is not referred to later.

- $Name == Pattern$

  Bind *Name* to *Pattern* if it is not already bound; otherwise match *Pattern* with the value of *Name*, binding variables within *Pattern* to components of the structure bound to *Name* as necessary to obtain a match. This is used to introduce definitions, such as the initial specification in a script or to name intermediate assertions and loop invariants. It can also be used to extract program components by pattern matching, and to bind metavariables given in place of parameter values.

This notation is based on the tactic notation used in RRT [Nickson, 1994]. As in RRT, this notation is embedded in Prolog, so it is possible to call Prolog goals within a script in order to perform various computations such as textual replacements. Although not as compact as the notation used in Red [Vickers and Gardiner, 1994], this linear form appears to be more suitable as a basis for the kinds of processing 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. It may also include records of proofs, or at least hints to a prover. The notation described above is, however, all we need for this paper.

A script for the insertion algorithm derivation in Section 3 is shown in Figure 2. Definitions are used to give the initial specification (S_1) and to define *Inv* as an abbreviation for the loop invariant. The other steps are all refinement rule applications. Note how the result patterns in the *IntroVar*, *SplitSpec*, *IntroDo* and *FollowingAssign* steps provide names for the components that need to be refined by following steps. For convenience, we define *Inv* earlier than it was introduced in the derivation in Section 3, and use it in the first application of *SplitSpec*.

$$S_1 == a\colon \left[\mathbf{asc}\, a(1\mathinner{..}k) \land a(1\mathinner{..}k) = b \;\middle/\; \mathbf{asc}\, a(1\mathinner{..}k+1) \land a(1\mathinner{..}k+1) \approx b \frown \langle x \rangle \right]$$
$$S_1 \lhd \mathit{IntroVar}(i) \rhd [\![\_ \bullet S_2]\!]$$
$$\mathit{Inv} == 1 \le i \le k+1 \land \mathbf{asc}\, a(1\mathinner{..}k+1 \backslash i) \land a(1\mathinner{..}k+1 \backslash i) \approx b \land x \le a(i+1\mathinner{..}k+1)$$
$$S_2 \lhd \mathit{SplitSpec}(\mathit{Inv} \land a(1\mathinner{..}i-1) \le x) \rhd S_3 \,;\, S_4$$
$$S_3 \lhd \mathit{SplitSpec}(\mathit{Inv}) \rhd S_5 \,;\, S_6$$
$$S_5 \lhd \mathit{IntroAssign}(i := k+1)$$
$$S_6 \lhd \mathit{IntroDo}(\mathit{Inv}, i > 1 \land x \le a(i-1), i) \rhd \mathbf{do}\ \_ \to S_7\ \mathbf{od}$$
$$S_7 \lhd \mathit{FollowingAssign}(i := i-1) \rhd S_8 \,;\, \_$$
$$S_8 \lhd \mathit{IntroAssign}(a(i) := a(i-1))$$
$$S_4 \lhd \mathit{IntroAssign}(a(i) := x)$$

Figure 2: Script for list insertion program

Our refinement tool takes script like this, constructs the list insertion program (as shown in Figure 1), and generates the corresponding proof obligations. At present, scripts are created using an ordinary text editor, but they could also be constructed via a graphical interface such as that provided by RRT [Nickson, 1994]. The tool has a simple prover which simplifies proof obligations, removing those conditions it can discharge using simple rewrite rules embodying laws about logic, arithmetic, arrays, etc, and records the proof obligations that remain. These are presented to the user as they arise to be checked, so the user can abort the derivation if an unprovable proof obligation is generated, or vouch for a proof obligation that is valid but can't be proved by the simplifier. A more detailed description of this tool, and further examples of the script notation, are given in [Groves, 1995].

5 Generic scripts: A counting-up loop

In constructing programs using the refinement calculus, we find that similar sequences of refinement steps occur whenever a particular programming construct or technique is used. For example, various kinds of loops lead to common sequences of refinement steps. We can use our script notation to build descriptions of such common sequences of steps in a way that allows them to then be easily reused in any situation where they are required.

We illustrate this by giving a script to construct a "counting-up" loop; i.e. a loop which uses a counter to iterate through a fixed range of values in ascending order (see Figure 3). This script is generic, in the sense that it can be applied to any specification, and metavariables are used in specifying parameters for rules. Thus, the initial specification contains metavariables W, PRE and POST, and subsequent refinement steps are described in terms of metavariables K, LO, HI and INV, which are given in place of the name of the loop counter, the initial and final values of the loop counter, and the loop invariant, respectively.

The script constructs a declaration for K and creates a loop with $K \neq \mathit{HI}$ as its guard. It inserts assignments to initialise K to LO, and to increment K within the loop. It also inserts a specification (Init) before the loop, requiring W to be initialised so that INV holds when

$$
\begin{array}{l}
S_1 == W\colon\left[\, PRE \;\middle/\; POST \,\right] \\
S_1 \;\lhd\; \mathit{IntroVar}(K) \;\;\rhd\; [\![\; \_ \bullet S_2 \;]\!] \\
S_2 \;\lhd\; \mathit{SplitSpec}(INV) \;\;\rhd\; S_3 \,;\, S_4 \\
S_3 \;\lhd\; \mathit{FollowingAssign}(K := LO) \;\;\rhd\; \mathit{Init} \,;\, \_ \\
S_4 \;\lhd\; \mathit{IntroDo}(INV, K \neq HI, HI - K) \;\;\rhd\; \mathbf{do}\ \_ \rightarrow S_5\ \mathbf{od} \\
S_5 \;\lhd\; \mathit{FollowingAssign}(K := K + 1) \;\;\rhd\; \mathit{Maint} \,;\, \_
\end{array}
$$

Figure 3: Script for counting-up loop

$K = LO$, and a specification (Maint) within the loop, indicating how W has to be modified in order to maintain the invariant when K is increased. Further refinement steps will be needed to implement these specifications depending on the particular situation, according to the values of these parameters.

The form of the program fragment constructed by this script is shown in Figure 4. Note that the substitution $INV[K \leftarrow K + 1]$ arising from the application of *FollowingAssign* cannot be performed because INV is unbound.

$$
\begin{array}{ll}
[\![\;\; \mathbf{var}\ K \bullet & \\
\quad W\colon\left[\, PRE \;\middle/\; INV[K \leftarrow LO] \,\right]; & (\mathit{Init}) \\
\quad K := LO; & \\
\quad \mathbf{do}\ K \neq HI \rightarrow & \\
\qquad W\colon\left[\, INV \wedge K \neq HI \;\middle/\; INV[K \leftarrow K + 1] \,\right]; & (\mathit{Maint}) \\
\qquad K := K + 1 & \\
\quad \mathbf{od} & \\
]\!] &
\end{array}
$$

Figure 4: Counting-up loop program

We use a script like this by making a copy and providing values for the parameters. This can be done either by textually replacing occurrences of a parameter name with the required value, or by using syntactic definitions.[2] For example, to construct a loop to find the sum s of an array a of size n, we might provide the definitions:

[2] Clearly a better parameterisation mechanism is warranted, but this approach works and allows the basic ideas to be tested.

$W == s$
$PRE == 0 \leq n$
$POST == s = \sum_{i=1}^{n} a(i)$
$K == k$
$LO == 0$
$HI == n$
$INV == LO \leq K \leq HI \wedge s = \sum_{i=1}^{k} a(i)$

The script will then construct a loop as above, with the following specifications for *Init* and *Maint*:

$$s\colon \left[0 \leq n \;\middle/\; 0 \leq 0 \leq n \wedge s = \textstyle\sum_{i=1}^{0} a(i) \right] \tag{Init}$$

$$s\colon \left[0 \leq k \leq n \wedge s = \textstyle\sum_{i=1}^{k} a(i) \wedge k \neq n \;\middle/\; 0 \leq k+1 \leq n \wedge s = \textstyle\sum_{i=1}^{k+1} a(i) \right] \tag{Maint}$$

These can be refined to $s := 0$ and $s := s + a(k+1)$, respectively.

6 Example: Insertion sort

We will now show how the scripts we have seen in previous sections can be combined and adapted to give a script to derive an insertion sort algorithm. Suppose we wish to sort an array a, of length n. We begin with the specification:

$$a\colon \left[n = \#a \wedge a = b \;\middle/\; \mathbf{asc}\, a \wedge a \approx b \right]$$

An obvious way to sort an array is to sort successively longer prefixes of the array. This will require a counting-up loop, where the loop counter is the length of the sorted prefix, ranging from 0 to n. We should thus be able to use the generic counting-up loop script to create this loop. To instantiate the script, we first replace W by a, PRE by $n = \#a \wedge a = b$, and $POST$ by $\mathbf{asc}\, a \wedge a \approx b$. We then instantiate the rule parameters, giving a variable name, say k, for K, 0 for LO, and n for HI. We do these substitutions by textually replacing all occurrences of W, PRE, $POST$, K, LO and HI by the required values.

Next, we must find a suitable loop invariant to instantiate INV. While k varies between 0 and n, the loop will preserve the property that $a(1..k)$ is ascending, and is a permutation of its original value (i.e. $b(1..k)$), while the contents of $a(k+1\,..\,n)$ will be unchanged. Thus, the loop invariant is $0 \leq k \leq n \wedge \mathbf{asc}\, a(1\,..\,k) \wedge a(1\,..\,k) \approx b(1\,..\,k) \wedge a(k+1\,..\,n) = b(k+1\,..\,n)$. We instantiate INV by using a syntactic definition.

The script at this stage is shown in Figure 5, and the program it constructs, after minor simplification of *Init* and *Maint*, is shown in Figure 6. We now consider how to implement the specifications *Init* and *Main*.

Inspecting the specification for *Init*, we realise that the postcondition is implied by the precondition, so no action is required. We could thus refine *Init* to a **skip** statement, but that is rather inelegant. Instead, we will avoid creating *Init* by simply replacing the line

$S_3 \;\lhd\; FollowingAssign(k := 0) \;\rhd\; Init \;;\, \_$

$$S_1 == a\colon \left[n = \#a \wedge a = b \;\Big/\; \mathbf{asc}\, a \wedge a \approx b \right]$$

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

$$\mathit{INV} == 0 \le k \le n \wedge \mathbf{asc}\, a(1 \mathinner{..} k) \wedge a(1 \mathinner{..} k) \approx b(1 \mathinner{..} k) \wedge a(k+1 \mathinner{..} n) = b(k+1 \mathinner{..} n)$$

$$S_2 \;\lhd\; \mathit{SplitSpec}(\mathit{INV}) \;\;\rhd\; S_3 \,;\, S_4$$

$$S_3 \;\lhd\; \mathit{FollowingAssign}(k := 0) \;\;\rhd\; \mathit{Init} \,;\, \_$$

$$S_4 \;\lhd\; \mathit{IntroDo}(\mathit{INV}, k \ne n, n-k) \;\;\rhd\; \mathbf{do}\; \_ \to S_5 \;\mathbf{od}$$

$$S_5 \;\lhd\; \mathit{FollowingAssign}(k := k+1) \;\;\rhd\; \mathit{Maint} \,;\, \_$$

Figure 5: Script for outer loop for sort program

in the script by

$$S_3 \;\lhd\; \mathit{IntroAssign}(k := 0)$$

The reason this occurred is due to the generality of the counting-up loop script. Although this problem only requires the loop counter to be initialised, some counting-up loops (e.g. to find the sum or maximum of an array) do require other variables to be initialised. The ability to make simple changes like this is part of the flexibility we gain over derived rules (which cannot be adapted in this way) and tactics (where the possibility that a component might be refined to **skip** would need to be anticipated and code to test for it incorporated in the tactic).

$$
\begin{array}{ll}
[\![\; \mathbf{var}\; k \bullet & \\
\quad a\colon \left[a = b \wedge n = \#a \;\Big/\; \begin{array}{c} \mathbf{asc}\, a(1 \mathinner{..} 0) \\ a(1 \mathinner{..} 0) \approx b(1 \mathinner{..} 0) \\ a(1 \mathinner{..} n) = b(1 \mathinner{..} n) \end{array} \right]; & (\mathit{Init}) \\
\quad k := 0; & \\
\quad \mathbf{do}\; k \ne n \to & \\
\qquad a\colon \left[\begin{array}{c} 0 \le k < n \\ \mathbf{asc}\, a(1 \mathinner{..} k) \\ a(1 \mathinner{..} k) \approx b(1 \mathinner{..} k) \\ a(k+1 \mathinner{..} n) = b(k+1 \mathinner{..} n) \end{array} \;\Big/\; \begin{array}{c} \mathbf{asc}\, a(1 \mathinner{..} k+1) \\ a(1 \mathinner{..} k+1) \approx b(1 \mathinner{..} k+1) \\ a(k+2 \mathinner{..} n) = b(k+2 \mathinner{..} n) \end{array} \right]; & (\mathit{Maint}) \\
\qquad k := k+1 & \\
\quad \mathbf{od} & \\
]\!] &
\end{array}
$$

Figure 6: Outer loop for sort program

We now turn our attention to the specification *Maint* in the loop body. The presence of $\mathbf{asc}\, a(1 \mathinner{..} k)$ in the precondition and $\mathbf{asc}\, a(1 \mathinner{..} k+1)$ in the postcondition is reminiscent of the specification for insertion in an ordered list (Section 3), so we will endeavor to use the list insertion script here. For easy comparison, the two specifications are shown in Figure 7, where the list insertion specification is labelled as *Insert*. Note that the b's in the two specifications are different constants.

$$a\colon \left[\begin{array}{c} 0 \leq k < n \\ \mathbf{asc}\, a(1\mathinner{..}k) \\ a(1\mathinner{..}k) \approx b(1\mathinner{..}k) \\ a(k+1\mathinner{..}n) = b(k+1\mathinner{..}n) \end{array} \middle/ \begin{array}{c} \mathbf{asc}\, a(1\mathinner{..}k+1) \\ a(1\mathinner{..}k+1) \approx b(1\mathinner{..}k+1) \\ a(k+2\mathinner{..}n) = b(k+2\mathinner{..}n) \end{array} \right] \qquad (\mathit{Maint})$$

$$a\colon \left[\begin{array}{c} \mathbf{asc}\, a(1\mathinner{..}k) \\ a(1\mathinner{..}k) = b \end{array} \middle/ \begin{array}{c} \mathbf{asc}\, a(1\mathinner{..}k+1) \\ a(1\mathinner{..}k+1) \approx b \frown \langle x \rangle \end{array} \right] \qquad (\mathit{Insert})$$

Figure 7: Specifications *Maint* and *Insert*

We will now consider the differences between these specifications, and how we can use the list insertion script in implementing *Main*. One option would be to package the list insertion algorithm as a procedure and insert a procedure call. We prefer, however, to do the insertion in-line, since this leads to the familiar form of this algorithm. Also note that the correspondence required between a specification we wish to implement and the initial specification of a script we wish to reuse is not as stringent as that required for a procedure call. If the script we are reusing does not address the entire specification, we can adapt it to do so. A particular use of a script may also rely on properties of the algorithm derived by the script which are not stated in its initial specification. For example, in reusing the list insertion script, we will need to ensure that the list insertion code does not disturb the contents of $a(k+1..n)$, though that was not part of the list insertion specification.

The most significant difference between the two specifications is that *Insert* requires x as an input. Since we are attempting to extend the sorted prefix $a(1\mathinner{..}k)$ of a, whilst ensuring $a(1\mathinner{..}k+1) \approx b(1\mathinner{..}k+1) \wedge a(k+2\mathinner{..}n) = b(k+2\mathinner{..}n)$, we must take x to be $a(k+1)$.

We now need to decide how to provide the value for x. We could either textually replace x by $a(k+1)$ in the script, or introduce x as a local variable and initialise it to $a(k+1)$.[3] Since *Insert* can alter elements of a (and will alter $a(k+1)$), but not x, the latter option is preferred.

We thus insert steps into the script to declare x as a local variable and initialise it to $a(k+1)$. Since the remaining specification must not alter x, we use the *Leading Assignment* rule (see Section 2), which removes the variable assigned from the frame of the second component of the sequence it creates. The result of applying these steps to *Maint* is:

$$\begin{array}{l} [\![\ \mathbf{var}\ x \bullet \\ \quad x := a(k+1); \\ \quad a\colon \left[\begin{array}{c} 0 \leq k < n \\ \mathbf{asc}\, a(1\mathinner{..}k) \\ a(1\mathinner{..}k) \approx b(1\mathinner{..}k) \\ a(k+1\mathinner{..}n) = b(k+1\mathinner{..}n) \\ x = a(k+1) \end{array} \middle/ \begin{array}{c} \mathbf{asc}\, a(1\mathinner{..}k+1) \\ a(1\mathinner{..}k+1) \approx b(1\mathinner{..}k+1) \\ a(k+2\mathinner{..}n) = b(k+2\mathinner{..}n) \end{array} \right] \\ \quad]\!] \end{array} \qquad (\mathit{Maint}_2)$$

[3]These choices are analagous to the choice between pass-by-name and pass-by-value parameters in a procedure call.

$Maint_2$ is now fairly similar to *Insert*, so we can apply a variant of the list insertion script to it. We append a copy of the list insertion script, omitting the first line (since the initial specification is already provided here), renaming its labels to avoid clashes, and ensuring that the first refinement is applied to $Maint_2$. We modify the loop invariant, deleting the conjunct $a(1 \mathinner{..} k+1 \backslash i) \approx b$, and adding conjuncts $0 \leq k < n \wedge x = b(k+1) \wedge a(1 \mathinner{..} k+1 \backslash i) \approx b(1 \mathinner{..} k) \wedge a(k+2 \mathinner{..} n) = b(k+2 \mathinner{..} n)$. The final script is shown in Figure 8.

The additional non-trivial proof obligations which arise (ignoring index constraints in the premises) are:

- For $i := k+1$:

$$a(1 \mathinner{..} k) \approx b(1 \mathinner{..} k) \\ \Rightarrow a(1 \mathinner{..} k+1 \backslash i) \approx b(1 \mathinner{..} k)$$

- For $a(i) := a(i-1)$:

$$a(1 \mathinner{..} k+1 \backslash i) \approx b(1 \mathinner{..} k) \\ \Rightarrow a[i := a(i-1)](1 \mathinner{..} k+1 \backslash i-1) \approx b(1 \mathinner{..} k)$$

- For $a(i) := x$:

$$a(1 \mathinner{..} k+1 \backslash i) \approx b(1 \mathinner{..} k) \wedge x = b(k+1) \\ \Rightarrow a[i := a(i-1)](1 \mathinner{..} k+1) \approx b(1 \mathinner{..} k+1)$$

These are easily verified, so the derivation is now complete. The resulting program is shown in Figure 9.

The derivation we have ended up with is similar to the one given in [Morgan, 1990] (Chapter 11), though Morgan's version "bubbles" the value of $a(k+1)$ down to its required position rather than storing it in a local variable. Morgan's derivation makes more use of syntactic definitions, and is simplified by treating some conditions, such as $n = \#a$ and $a \approx b$ (Morgan's equivalent is $A = \textbf{bag}\ a$), informally as local invariants. The treatment of local invariants, definitions, and other forms of context, within a refinement tool is discussed in [Nickson, 1994].

We have been able to make use of two previous derivations, which could also be used for many other algorithms. To do this, we only needed to add two additional steps (to declare and initialise x), modify the loop invariant and prove three new proof obligations. Of course, being able to do this relied on the existing scripts being in a suitable form — different ways of implementing the list insertion algorithm would have lead to different sorting algorithms, and might have been more difficult to adapt. This is, of course, true of any software components we attempt to reuse. By reusing refinement scripts, rather than code, the resulting proof obligations provide a check that the code we insert does in fact perform as required.

7 Adapting a Derivation: Stable Sort

We will now consider how the insertion sort script we developed in the previous section can be further adapted to construct a stable sorting algorithm. We first extend the derivation to construct an algorithm which sorts on keys, rather than on whole array elements. We then consider the additional requirement that the sort be stable.

$$S_1 == a\colon \left[n = \#a \wedge a = b \;\middle/\; \mathbf{asc}\, a \wedge a \approx b \right]$$

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

$$\mathit{INV} == 0 \le k \le n \wedge \mathbf{asc}\, a(1\,..\,k) \wedge a(1\,..\,k) \approx b(1\,..\,k) \wedge a(k+1\,..\,n) = b(k+1\,..\,n)$$

$$S_2 \;\lhd\; \mathit{SplitSpec}(\mathit{INV}) \;\rhd\; S_3 \,;\, S_4$$

$$S_3 \;\lhd\; \mathit{IntroAssign}(k := 0)$$

$$S_4 \;\lhd\; \mathit{IntroDo}(\mathit{INV}, k \neq n, n-k) \;\rhd\; \mathbf{do}\ \_ \rightarrow S_5\ \mathbf{od}$$

$$S_5 \;\lhd\; \mathit{FollowingAssign}(k := k+1) \;\rhd\; \mathit{Maint} \,;\, \_$$

$$\mathit{Maint} \;\lhd\; \mathit{IntroVar}(x) \;\rhd\; [\![\;\_ \bullet \mathit{Maint}_1 \;]\!]$$

$$\mathit{Maint}_1 \;\lhd\; \mathit{LeadingAssign}(x := a(k+1)) \;\rhd\; \_ \,;\, \mathit{Maint}_2$$

$$I_1 \;\lhd\; \mathit{IntroVar}(i) \;\rhd\; [\![\;\_ \bullet I_2 \;]\!]$$

$$\begin{aligned} \mathit{Inv} == {} & 0 \le k < n \wedge x = b(k+1) \wedge a(k+2\,..\,n) = b(k+2\,..\,n) \\ & 1 \le i \le k+1 \wedge \mathbf{asc}\, a(1\,..\,k+1 \backslash i) \wedge \\ & a(1\,..\,k+1 \backslash i) \approx b(1\,..\,k) \wedge x \le a(i+1\,..\,k+1) \end{aligned}$$

$$I_2 \;\lhd\; \mathit{SplitSpec}(\mathit{Inv} \wedge a(1\,..\,i-1) \le x) \;\rhd\; I_3 \,;\, I_4$$

$$I_3 \;\lhd\; \mathit{SplitSpec}(\mathit{Inv}) \;\rhd\; I_5 \,;\, I_6$$

$$I_5 \;\lhd\; \mathit{IntroAssign}(i := k+1)$$

$$I_6 \;\lhd\; \mathit{IntroDo}(\mathit{Inv}, i > 1 \wedge x \le a(i-1), i) \;\rhd\; \mathbf{do}\ \_ \rightarrow I_7\ \mathbf{od}$$

$$I_7 \;\lhd\; \mathit{FollowingAssign}(i := i-1) \;\rhd\; I_8 \,;\, \_$$

$$I_8 \;\lhd\; \mathit{IntroAssign}(a(i) := a(i-1))$$

$$I_4 \;\lhd\; \mathit{IntroAssign}(a(i) := x)$$

Figure 8: Script for insertion sort program

7.1 Sorting on keys

Suppose that a is an array of records with fields u and v (both integers), and that we wish to sort a on field u. We will write $a(i).u$ to refer to the u field of $a(i)$, $a(i\,..\,j).u$ for the sequence $\langle a(i).u, \cdots, a(j).u \rangle$, and $a.u$ for $a(1\,..\,\#a).u$. Using this notation, the new specification is:

$$a\colon \left[n = \#a \wedge a = b \;\middle/\; a \approx b \wedge \mathbf{asc}\, a.u \right] \qquad (S_1)$$

In order to derive a program for this specification, we modify the script for the insertion sort derivation, so that it compares keys, rather than whole elements of a. Working through the steps in the insertion sort script shown in Figure 8, we find there are only five changes required:

- Replace $\mathbf{asc}\, a$ in the postcondition of S_1 by $\mathbf{asc}\, a.u$
- Replace $\mathbf{asc}\, a(1\,..\,k)$ in INV by $\mathbf{asc}\, a(1\,..\,k).u$
- Replace $\mathbf{asc}\, a(1\,..\,k+1 \backslash i)$ in Inv by $\mathbf{asc}\, a(1\,..\,k+1 \backslash i).u$
- Replace $x < a(i+1\,..\,k+1)$ in Inv by $x.u < a(i+1\,..\,k+1).u$

$$
\begin{array}{l}
[\![\ \mathbf{var}\ k \bullet \\
\quad k := 0; \\
\quad \mathbf{do}\ k \neq n \rightarrow \\
\qquad [\![\ \mathbf{var}\ x \bullet \\
\qquad\quad x := a(k+1); \\
\qquad\quad [\![\ \mathbf{var}\ i \bullet \\
\qquad\qquad i := k+1; \\
\qquad\qquad \mathbf{do}\ i > 1 \wedge x < a(i-1) \rightarrow \\
\qquad\qquad\qquad a(i) := a(i-1); \\
\qquad\qquad\qquad i := i-1 \\
\qquad\qquad \mathbf{od}; \\
\qquad\qquad a(i) := x \\
\qquad\quad]\!] \\
\qquad]\!]; \\
\qquad k := k+1 \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

Figure 9: Insertion sort program

- Replace $x < a(i-1)$ in the inner loop guard by $x.u < a(i-1).u$

The resulting program is identical to that in Figure 9, except that the guard for the inner loop has $x.u < a(i-1).u$ rather than $x < a(i-1)$.

Several of the proof obligations are now a little more complicated, since they involve keys and sequences of keys, but these are mainly notational complications and the proof obligations are easily proved.

7.2 Stability

Now, consider the requirement that the sort be stable, i.e. whenever two elements of a have the same key, their order is preserved during sorting. We will formalise this requirement by adding a conjunct $stable(a,b)$ to the postcondition of our specification, where $stable$ is defined as:

$$
\begin{array}{ll}
stable(a,b) \mathrel{\widehat{=}} & (\ \forall i, j, i', j' \in 1 \mathinner{..} \#a \bullet \\
& \quad i < j \wedge b(i).u = b(j).u \wedge a(i') = b(i) \wedge a(j') = b(j) \Rightarrow i' < j')
\end{array}
$$

Thus, the new specification is:

$$
a\colon \Big[\, n = \#a \wedge a = b \;\Big/\; a \approx b \wedge \mathbf{asc}\ a.u \wedge stable(a,b) \,\Big]
$$

We now consider how the derivation must be modified to accomodate the stability requirement, and the additional proof obligations that arise. Clearly, both loops will have to preserve a stability property, so we will need to modify the loop invariants. For the outer loop, we add the conjunct $\mathit{stable}(a(1 \mathinner{..} k), b(1 \mathinner{..} k))$. For the inner loop, we add the conjunct $\mathit{stable}(a(1 \mathinner{..} k+1 \backslash i), b(1 \mathinner{..} k))$. With these changes, we get the following new proof obligations:

- For $i := k+1$:

$$\begin{array}{l} \mathit{stable}(a(1 \mathinner{..} k), b(1 \mathinner{..} k)) \\ \Rightarrow \mathit{stable}(a(1 \mathinner{..} k+1 \backslash i), b(1 \mathinner{..} k)) \end{array}$$

- For $a(i) := a(i-1)$:

$$\begin{array}{l} \mathit{stable}(a(1 \mathinner{..} k+1 \backslash i), b(1 \mathinner{..} k)) \\ \Rightarrow \mathit{stable}(a[i := a(i-1)](1 \mathinner{..} k+1 \backslash i-1), b(1 \mathinner{..} k)) \end{array}$$

- For $a(i) := x$:

$$\begin{array}{l} \mathit{stable}(a(1 \mathinner{..} k+1 \backslash i), b(1 \mathinner{..} k)) \wedge x = b(k+1) \wedge a(1 \mathinner{..} i-1).u \le x.u \le a(i \mathinner{..} k).u \\ \Rightarrow \mathit{stable}(a[i := a(i-1)](1 \mathinner{..} k+1), b(1 \mathinner{..} k+1)) \end{array}$$

We can verify the first two of these, but not the last. Since all items in $a(1 \mathinner{..} k+1 \backslash i)$ were to the left of (had lower indices than) x in b, any items in $a(1 \mathinner{..} k+1 \backslash i)$ with the same key as x must remain to the left of x, i.e. at index positions less than i. We could the verify the proof obligation if the last premise was $a(1 \mathinner{..} i-1).u \le x.u \le a(i \mathinner{..} k).u$. We can achieve this by replacing $x \le a(i-1)$ by $x < a(i-1)$ in the guard of the inner loop. With this modification, the other proof obligations are still provable, so the derivation is complete. The script resulting after both modifications is shown in Figure 10.

In this case, modifying the sort so that it is stable was quite easy. Indeed, if we had used $x < a(i-1)$ in the guard initially (which we might have done, simply to reduce the number of iterations required), the sort would have been stable already, and we would only have needed to do the additional proofs. If we began with an algorithm such as selection sort or quicksort, which may swap non-adjacent elements, modifying the algorithm so that it was stable would not be so easy.

8 Conclusions

In this paper, we have shown that a formal derivation can be described using a simple notation that is amenable to mechanical processing. Scripts in this notation can be used to generate programs and the corresponding proof obligations. These scripts can then be adapted and combined, allowing derivations to be reused, and providing a formal basis for program maintenance. It should also be straightforward to produce descriptions of derivations such as that presented in Section 3, but we have not done this yet.

We have a simple tool which generate programs and the corresponding proof obligations from refinement scripts. Its input is a minor variation of that presented here — the tool is implemented in Prolog and scripts are expressed using Prolog terms. The tool also attempts to discharge proof obligations using a library of rewrite rules, but will allow the derivation to proceed if it cannot. Although this tool is quite simple, describing derivations in the script

$$S_1 == a\colon \left[n = \#a \wedge a = b \;\middle/\; a \approx b \wedge \mathbf{asc}\, a.u \wedge \mathit{stable}(b, a) \right]$$

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

$$\begin{aligned} \mathit{INV} == {} & 0 \leq k \leq n \wedge \mathbf{asc}\, a(1 \mathinner{..} k).u \wedge a(1 \mathinner{..} k) \approx b(1 \mathinner{..} k) \wedge \\ & a(k+1 \mathinner{..} n) = b(k+1 \mathinner{..} n) \wedge \mathit{stable}(b, a) \end{aligned}$$

$$S_2 \;\lhd\; \mathit{SplitSpec}(\mathit{INV}) \;\;\rhd\; S_3 \,;\, S_4$$

$$S_3 \;\lhd\; \mathit{IntroAssign}(k := 0)$$

$$S_4 \;\lhd\; \mathit{IntroDo}(\mathit{INV}, k \neq n, n - k) \;\;\rhd\; \mathbf{do}\ \_ \rightarrow S_5\ \mathbf{od}$$

$$S_5 \;\lhd\; \mathit{FollowingAssign}(k := k + 1) \;\;\rhd\; \mathit{Maint} \,;\, \_$$

$$\mathit{Maint} \;\lhd\; \mathit{IntroVar}(x) \;\;\rhd\; [\![\ \_\bullet \mathit{Maint}_1\ ]\!]$$

$$\mathit{Maint}_1 \;\lhd\; \mathit{LeadingAssign}(x := a(k+1)) \;\;\rhd\; \_ \,;\, \mathit{Maint}_2$$

$$I_1 \;\lhd\; \mathit{IntroVar}(i) \;\;\rhd\; [\![\ \_\bullet I_2\ ]\!]$$

$$\begin{aligned} \mathit{Inv} == {} & 0 \leq k < n \wedge x = b(k+1) \wedge a(k+2 \mathinner{..} n) = b(k+2 \mathinner{..} n) \\ & 1 \leq i \leq k+1 \wedge \mathbf{asc}\, a(1 \mathinner{..} k+1 \backslash i).u \wedge \\ & a(1 \mathinner{..} k+1 \backslash i) \approx b(1 \mathinner{..} k) \wedge x.u \leq a(i+1 \mathinner{..} k+1) \wedge \mathit{stable}(a, b) \end{aligned}$$

$$I_2 \;\lhd\; \mathit{SplitSpec}(\mathit{Inv} \wedge a(1 \mathinner{..} i-1) \leq x) \;\;\rhd\; I_3 \,;\, I_4$$

$$I_3 \;\lhd\; \mathit{SplitSpec}(\mathit{Inv}) \;\;\rhd\; I_5 \,;\, I_6$$

$$I_5 \;\lhd\; \mathit{IntroAssign}(i := k + 1)$$

$$I_6 \;\lhd\; \mathit{IntroDo}(\mathit{Inv}, i > 1 \wedge x.u \leq a(i-1).u, i) \;\;\rhd\; \mathbf{do}\ \_ \rightarrow I_7\ \mathbf{od}$$

$$I_7 \;\lhd\; \mathit{FollowingAssign}(i := i - 1) \;\;\rhd\; I_8 \,;\, \_$$

$$I_8 \;\lhd\; \mathit{IntroAssign}(a(i) := a(i-1))$$

$$I_4 \;\lhd\; \mathit{IntroAssign}(a(i) := x)$$

Figure 10: Script for insertion sort program on keys

notation and running through the tool has proved to be a helpful in checking the details of formal derivations, and determining what laws they rely on.

At present, we construct scripts "manually" using an editor, but it would be straightfoward to construct them automatically from an interactive dialogue using a tool such as Red or RRT. Similarly, we currently perform the kinds of modifications to scripts desribed here and in [Groves, 1995] using an editor and then feeding the modified script to the tool. There is considerable scope for providing support for this within the tool, and automating parts of it. A major issue to be addressed here is the kind of matching required in order to identify parts of the script that need to be modified; for example, in modifying the insertion sort derivation to obtain a key sort, we had to replace occurrences of a and $a(i)$ by $a.u$ and $a(i).u$ — but not all. Without some reliable way of identifying the occurrences that should be replaced, the best we can provide is some form of query-replace facility which finds all the places where a replacement might be required and allows the user to decide which ones will be replaced.

Having program derivations described in this form raises a number of other interesting possibilities worth exploring. For instance, we can perform various transformations on derivations rather than on programs. Data refinements (especially with functional coupling invariants) can be performed by replacing abstract variables by concrete ones in the derivation. We could also introduce a procedure by extracting a fragment of a script, inserting a corresponding procedure declaration and replacing the removed fragment by a procedure call. Many of the "structure modifying" (as opposed to structure creating) refinement laws given in [Morgan, 1990] can also be treated as transformations on derivations; for example eliminating **skip** statements, removing branches with false guards from **if** statements, merging identical branches in **if** statements and collapsing nested alternations. Similarly, we can defer decisions about bindings for parameters in refinement rules (as suggested in [Nickson and Groves, 1994]) by leaving metavariables in a script while it is being constructed and subsequently binding them. Whether it is advantageous to treat these as transformations on derivations requires further investigation.

Treating these as operations as transformations on derivations also 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. Our view is that the programmer should have the freedom to make such steps explicit in the refinement history if they are felt to be useful, or remove them from the history if they are not.

References

[Back, 1981] R. J. R. Back. "On correct refinements of programs". *Jl. Computer and System Sciences* **23** (1981), pp49–68.

[Back, 1988] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica* **25** (1988), pp593–624.

[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.

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

[Groves, 1995] Lindsay Groves. "Adapting Formal Derivations". Presented at 4th Australasian Refinement Workshop, Sydney, 1995. Technical Report TR-CS-95/9, Victoria University of Wellington.

[Morgan, 1988] Carroll Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comp. Prog.* **11** (1988). Also in [Morgan, Robinson and Gardiner, 1988], pp58–71.

[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.

[Morris, 1987] J. M. Morris. "A theoretical basis for stepwise refinement and the programming calculus". *Science of Computer Programming* **9** (1987), pp287–306.

[Nickson and Groves, 1994] Raymond G. Nickson and Lindsay J. Groves. "Metavariables and Conditional Refinements in the Refinement Calculus" *Proc. 6th Refinement Workshop*, British Computer Society, London, January 1994.

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

[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.