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

Procedures in the Refinement Calculus: A New Approach?

Lindsay Groves

Technical Report CS-TR-96/7
July 1996

Abstract

We examine the use of procedures in the refinement calculus and show that the traditional approach leads to certain problems when programs are not constructed in a strictly top-down manner. These problems arise because a procedure name becomes associated with the implementation when the procedure body is refined, and we examine some ways in which they may be avoided. We argue that, because procedures are not required in order to express procedural abstractions, the primary purpose of using procedures in program refinement is to do with packaging the final program, and that decisions about how a program is packaged into procedures should be made separately from algorithm design decisions. We present an alternative approach based on this rationale which avoids the aforementioned problems, and discuss ways in which it can be supported by a refinement tool.

Publishing Information

Seventh BCS/FACS Refinement Workshop, Bath, July 1996.

Procedures in the Refinement Calculus: A New Approach?

Lindsay Groves

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

Abstract

We examine the use of procedures in the refinement calculus and show that the traditional approach leads to certain problems when programs are not constructed in a strictly top-down manner. These problems arise because a procedure name becomes associated with the implementation when the procedure body is refined, and we examine some ways in which they may be avoided. We argue that, because procedures are not required in order to express procedural abstractions, the primary purpose of using procedures in program refinement is to do with packaging the final program, and that decisions about how a program is packaged into procedures should be made separately from algorithm design decisions. We present an alternative approach based on this rationale which avoids the aforementioned problems, and discuss ways in which it can be supported by a refinement tool.

1 Introduction

Procedures have long been recognised as one of the principle mechanisms for structuring computer programs. They allow a program text to be broken into conceptually separate parts that can be understood in isolation, and allow the effort of constructing a program to be broken into logically independent parts (possibly undertaken by different people). They also allow the same program fragment to be used in different parts of a program (with parameter substitutions to adapt to specific use as appropriate) without rewriting and without concern for how they are implemented. Procedures can also be collected into larger structures such as modules or libraries, so that they can be reused across many different programs.

Procedures are in principle not strictly necessary, since any program that can be written with procedures can also be written without them (assuming that the programming language has some kind of loop structure and can use at least as much storage space as recursive procedures could use for the stack), so the reasons for using procedures are essentially pragmatic — to do with making the programming task more manageable. For this reason (presumably), much of the early work in program verification and program refinement did not consider procedures. For example, the original work by Hoare [14] and Dijkstra [6] did not include procedures, though in both cases, procedures were subsequently added (see [15], [8], [7]). Similarly, early work on the refinement calculus ([1], [3], [18], [21], [23]) did not include procedures, but extensions to handle procedures are presented in [2] and [19] (also see [20] and [29]). In that work, however, the main emphasis has been on providing suitable semantics and establishing important properties for procedures and procedure calls. There has been little analysis of the precise role of procedures in formal derivations, and few examples of their use.

In this paper, we examine the use of procedures in the refinement calculus. We show that the treatment of procedures given by Back [2] and Morgan [19] leads to some methodological difficulties, essentially because when a procedure body is refined, the procedure name becomes associated with the result of the refinement. We outline a number of ways in which these problems might be addressed, and review the role of procedures in programming, and in formal programming in particular.

Our motivation here is to understand the way in which procedures are used in the refinement calculus and how we can best provide support for procedures in a refinement tool. This leads us to consider how and when the decision is made to introduce a procedure declaration or a procedure call, and how these decisions are manifested in the dialogue that takes place between a programmer and a refinement tool. We conclude that procedures should be seen primarily as a device for packaging the final program, and that the decision to package part of a program as a procedure should be separated from the basic design of the algorithm embodied in the program. During the design process, the main role of procedures is to allow procedural abstractions to be labelled, and that facility can be provided without procedures.

We present an alternative approach to incorporating procedures in the refinement calculus based on these ideas and discuss the support that might be provided for this approach by a refinement tool.

We are considering program refinement in the context of a refinement tool (e.g. [28], [11], [26], [4]). Thus, we view refinement as beginning with a specification and proceeding by the application of refinement rules, which have been shown to be valid. We assume that the refinement tool is equipped with a suitable set of rules, though the user may be permitted to add further rules, provided their validity can be established. We assume familiarity with the basic concepts and notation of the refinement calculus, as presented in [20].

2 The traditional approach

Morgan [19] and Back [2] both define the semantics for calls on parameterless, non-recursive procedures, in terms of the *copy rule* or *substitution principle*. Writing a block which declares a procedure p with body T and scope S as $[\![\ \mathbf{proc}\ p\ \widehat{=}\ T \bullet S\ ]\!]$, and a call on procedure p as p, we get[1]:

Definition *Procedure declaration* For any programs S and T, and procedure name p:

$$[\![\ \mathbf{proc}\ p\ \widehat{=}\ T \bullet S\ ]\!] = S[p \setminus T]$$

That is, a program containing a procedure declaration $\mathbf{proc}\ p\ \widehat{=}\ T$ is the same as one in which all occurrences of p, within the scope of this declaration, are replaced by T. Local variables within S are renamed if necessary to avoid variable capture — thus, T can't refer to variables declared within S as global variables.

Parameters are treated separately via a substitution mechanism, and recursion is handled separately via a fixed point construct (recursion block). Vickers and Morgan [29] give a differ-

[1] $S[p \setminus T]$ denotes the result of replacing all free occurrences of p in S by T.

ent semantics, based on invariants, but the results and approach are, for the purposes of this discussion, essentially the same (see, however, Section 4).

With these semantics, we obtain a number of laws which can be used as transformation rules in developing programs with procedures.

We can introduce a procedure declaration in anticipation of later use, by wrapping its declaration around an existing program:

Law *Declare Procedure 1* For any programs S and T, and procedure name p, where p is not free in S or T:

$$S = [\![\ \mathbf{proc}\ p\ \ \widehat{=}\ \ T \bullet S\ ]\!]$$

A procedure may also be introduced to name an existing program fragment, which is then replaced by a call on the new procedure:

Law *Declare Procedure 2* For any program S and procedure name p, where p is not free in S:

$$S = [\![\ \mathbf{proc}\ p\ \ \widehat{=}\ \ S \bullet p\ ]\!]$$

These laws are not very convenient to use in practice. The first provides no motivation for introducing the procedure declaration. The second only introduces a single call on the procedure, and its scope is only that call. Additional laws can be used to widen the scope of the procedure declaration (see [29]), but that is rather clumsy.

The following law is more flexible [2][2]:

Law *Declare Procedure 3* Let $\mathcal{C}$ be a program schema and d be a program name, and let $\mathcal{C}[X]$ be the result of replacing all occurrences of d in $\mathcal{C}$ by the program X. Then, for any program T and procedure name p, where p is not free in $\mathcal{C}$ or T:

$$\mathcal{C}[T] = [\![\ \mathbf{proc}\ p\ \ \widehat{=}\ \ T \bullet \mathcal{C}[p]\ ]\!]$$

Here we can choose an arbitrary program component, T, to be turned into a procedure, and then introduce the procedure declaration and some number of calls on the procedure in one step. In applying this law to a program P, we need to find a schema $\mathcal{C}$ such that $\mathcal{C}[T] = P$. If T occurs more than once in P, there will be several such schemas, and the one we use will determine which occurrences of T get replaced by calls. Note that *Declare Procedure 2* is a special case of *Declare Procedure 3* where $\mathcal{C}[X] = X$, and *Declare Procedure 1* is a special case of *Declare Procedure 3* where d does not occur in $\mathcal{C}$.

We can also introduce calls on an existing procedure using the following law [29]:

Law *Call Procedure 1* Let $\mathcal{C}$ be a program schema and d be a program name, and let $\mathcal{C}[X]$ be the result of replacing all occurrences of d in $\mathcal{C}$ by the program X. Then, for any program T and procedure name p, where p is not free in T:

$$[\![\ \mathbf{proc}\ p\ \ \widehat{=}\ \ T \bullet \mathcal{C}[T]\ ]\!] = [\![\ \mathbf{proc}\ p\ \ \widehat{=}\ \ T \bullet \mathcal{C}[p]\ ]\!]$$

[2] Back doesn't state this as a law, but presents it as part of the method.

Thus, within the scope of a procedure declaration, occurrences of the procedure body may be replaced by calls on the procedure. As in *Declare Procedure 3*, if T occurs more than once in $\mathcal{C}[T]$, the schema we use will determine which occurrences of T get replaced by calls.

Having introduced a procedure declaration, we need to be able to further refine both the body of the procedure and the scope of the declaration. This can be done via the the following monotonicity laws for procedures[3].

Law *Monotonicity of Procedure Body* If $T \sqsubseteq T'$ then

$$[\![\ \mathbf{proc}\ p\ \widehat{=}\ T \bullet S\ ]\!] \sqsubseteq [\![\ \mathbf{proc}\ p\ \widehat{=}\ T' \bullet S\ ]\!]$$

Law *Monotonicity of Procedure Scope* If $S \sqsubseteq S'$ then

$$[\![\ \mathbf{proc}\ p\ \widehat{=}\ T \bullet S\ ]\!] \sqsubseteq [\![\ \mathbf{proc}\ p\ \widehat{=}\ T \bullet S'\ ]\!]$$

These laws suggest that the body and scope of a procedure declaration can be developed independently; we will see, however, that this can be problematic. *Monotonicity of Procedure Body* exploits the fact that specifications and implementations can both be expressed within the same language. We regard the specification of a procedure to be the body associated with the procedure when it is declared.

The use of these laws can be illustrated by Morgan's example, of sorting three integers p, q and r into increasing order, assuming that the operators $\sqcap$ and $\sqcup$ are not available in the target language [20, Chap. 11].

Morgan's derivation is as follows[4]:

$$p, q, r\!: \left[\mathit{true} \Big/ \begin{array}{c} p \le q \le r \\ \{\!|\ p, q, r\ |\!\} = \{\!|\ p_0, q_0, r_0\ |\!\} \end{array} \right]$$

$$\begin{array}{lll}
 & \sqsubseteq & p, q := p \sqcap q, p \sqcup q; \\
 & & q, r := q \sqcap r, q \sqcup r; \\
 & & p, q := p \sqcap q, p \sqcup q \\
 & \sqsubseteq & \mathbf{proc}\ \mathit{Sort}\ \widehat{=}\ p, q := p \sqcap q, p \sqcup q \bullet \qquad (i) \\
 & & \mathit{Sort}; \\
 & & q, r := q \sqcap r, q \sqcup r; \\
 & & \mathit{Sort} \\
(i) & \sqsubseteq & \mathbf{if}\ p \le q \rightarrow \mathbf{skip}\ [\!]\ p \ge q \rightarrow p, q := q, p\ \mathbf{fi}
\end{array}$$

The first refinement is quite standard, and is justified by laws such as *Sequential Composition* and *Assignment* or *Following/Leading Assignment*. In fact, some ingenuity (or perhaps foresight) is required to get the first and third assignments the same — this seems to be typical of the massaging needed to get a specification into the required form to introduce a procedure call.

[3] The first of these is given by Back [2]. Both are implicit in Morgan's book [20], though he doesn't give the laws explicitly. Both are also consequences of the monotonicity theorem given by Vickers and Morgan [29], where they are proved jointly.

[4] $x \sqcap y$ is the greatest lower bound (i.e. the minimum) of x and y, $x \sqcup y$ is the least upper bound (i.e. the maximum) of x and y, and $\{\!|\ \ldots\ |\!\}$ denotes bag display.

The second refinement can be justified in a number of ways. We could apply *Declare Procedure 1* to the result of the first refinement to introduce the procedure declaration, then use *Call Procedure 1* to introduce the two calls. Or, we could apply *Declare Procedure 2* to the first occurrence of $p, q := p \sqcap q, p \sqcup q$ to introduce the declaration and one call, then use a distribution law to widen the scope of the declaration and use *Call Procedure 1* to introduce the second call. Alternatively, we could use *Declare Procedure 3* to introduce the declaration and both calls in one step.

The third refinement is justified by *Monotonicity of Procedure Body*, along with laws for **if**, **skip** and assignment statements.

The program at this stage is:

$$
\begin{array}{l}
[\![\ \textbf{proc}\ \mathit{Sort}\ \widehat{=}\ \textbf{if}\ p \leq q \rightarrow \textbf{skip}\ [\!]\ p \geq q \rightarrow p, q := q, p\ \textbf{fi}\ \bullet \\
\quad \mathit{Sort}; \\
\quad q, r := q \sqcap r, q \sqcup r; \\
\quad \mathit{Sort} \\
]\!]
\end{array}
$$

This program is obtained by collecting the results of the previous refinement steps, which can be performed automatically by a refinement tool. This collecting process relies implicitly on the monotonicity of the various program constructors with respect to refinement, and the transitivity of refinement.

Morgan goes on to show that the remaining assignment can be handled using substitution; we will ignore that for now (see Section 6).

3 Problems with the traditional approach

Strict application of the laws presented above can lead to some undesirable consequences in program development. These problems arise principally because using the *Monotonicity of Procedure Body* law causes the procedure name to become associated with the implementation of the procedure body rather than its specification. We identify two such problems, and illustrate them with simple examples.

3.1 Procedure calls need to be introduced before the procedure is implemented

The first problem is that this way of handling procedures effectively requires that all calls to a procedure should be introduced *before* the procedure is implemented (i.e. before the body of the procedure is refined). To illustrate this, let us consider Morgan's 3-sort example again, and see what would happen if we had done the derivation in a different order. We might, for example, have split the initial specification into two steps, introduced a procedure and turned the first step into a procedure call, and implemented the procedure, before then refining the second step. Thus:

$$
p, q, r \colon \left[\mathit{true} \ \middle/ \ \begin{array}{c} p \leq q \leq r \\ \{\!|\ p, q, r\ |\!\} = \{\!|\ p_0, q_0, r_0\ |\!\} \end{array} \right]
$$

$$
\begin{array}{lll}
 & \sqsubseteq & p, q := p \sqcap q, p \sqcup q; \\
 & & p, q, r := p \sqcap r, q \sqcap (p \sqcup r), q \sqcup r \\
 & \sqsubseteq & \textbf{proc } \mathit{Sort} \ \widehat{=} \ p, q := p \sqcap q, p \sqcup q \bullet \qquad (i) \\
 & & \mathit{Sort}; \\
 & & p, q, r := p \sqcap r, q \sqcap (p \sqcup r), q \sqcup r \qquad (ii) \\
(i) & \sqsubseteq & \textbf{if } p \leq q \rightarrow \textbf{skip} \ [\!] \ p \geq q \rightarrow p, q := q, p \textbf{ fi} \\
(ii) & \sqsubseteq & \mathrm{q}, r := q \sqcap r, q \sqcup r; \\
 & & p, q := p \sqcap q, p \sqcup q
\end{array}
$$

The program at this stage is:

$$
\begin{array}{l}
[\![\ \textbf{proc } \mathit{Sort} \ \widehat{=} \ \textbf{if } p \leq q \rightarrow \textbf{skip} \ [\!] \ p \geq q \rightarrow p, q := q, p \textbf{ fi} \bullet \\
\quad \mathit{Sort}; \\
\quad q, r := q \sqcap r, q \sqcup r; \\
\quad p, q := p \sqcap q, p \sqcup q \qquad (iii) \\
]\!]
\end{array}
$$

Now, we can't refine the last assignment (iii) to a call on *Sort* using *Call Procedure 1*, because the body of *Sort* has already been refined!

We could refine (iii) to an **if** statement and then turn it into a procedure call, but that is pointless repetition of effort. Imagine having to refine a sort specification to exactly the same implementation of the same sorting algorithm used by a sort procedure before being able to use it!

Of course, we have already refined the previous occurrence of $p, q := p \sqcap q, p \sqcup q$ to the **if** statement which is now the body of *Sort*, but the laws we have don't allow us to exploit that. We could facilitate this by introducing a new law:

Law *Call Procedure 2* Let $\mathcal{C}$ be a program schema and d be a program name, and let $\mathcal{C}[X]$ be the result of replacing all occurrences of d in $\mathcal{C}$ by the program X. Then, for any programs S and T, and procedure name p, where p is not free in T and $S \sqsubseteq T$:

$$
[\![\ \textbf{proc } p \ \widehat{=} \ T \bullet \mathcal{C}[S] \ ]\!] \sqsubseteq [\![\ \textbf{proc } p \ \widehat{=} \ T \bullet \mathcal{C}[p] \ ]\!]
$$

While this allows us to complete the derivation above, it does not entirely solve our problems, as we shall now see.

3.2 Calls may depend on properties of the implementation

In the above example, the implementation of *Sort* differed from its specification only in that the specification was expressed in non-executable form, and the problem arose when we wanted to introduce a call after the body of the procedure had been refined. A different problem can arise when the implementation of a procedure is a proper refinement (i.e. not equivalent to) its specification. This is illustrated by the following example.

Suppose a program P contains the specification:

$$x\colon \left[true \;\middle/\; x = 1 \vee x = 2 \right]$$

and suppose that we decide to introduce a procedure with this specification as its body and P as its scope, replace the occurrence of the specification by a call on the procedure, and then refine the procedure body to $x := 1$. We thus obtain[5]:

$$\begin{array}{rcl} P & = & \ldots\ x\colon \left[true \;\middle/\; x = 1 \vee x = 2 \right]\ \ldots \\ & \sqsubseteq & [\![\ \mathbf{proc}\ p\ \widehat{=}\ x\colon \left[true \;\middle/\; x = 1 \vee x = 2 \right]\ \bullet\ \ldots\ p\ \ldots]\!] \\ & \sqsubseteq & [\![\ \mathbf{proc}\ p\ \widehat{=}\ x := 1\ \bullet\ \ldots\ p\ \ldots]\!] \end{array}$$

If we later encounter the statement $x := 1$ elsewhere within P (or its refinement), we could then replace that by a call on p using *Call Procedure 1* (or *Call Procedure 2*).

This refinement, however, violates one of the fundamental principles of using procedures — that we should be able to replace one implementation of a procedure by another without affecting the correctness of a program which uses the procedure. According to this principle, we should be able to replace the implementation of p by $x := 2$, since it also satisfies the original specification; but of course, that may not satisfy the specification for the later call we introduced!

The problem here is that a procedure call may be introduced in a context which relies on properties of the implementation of the procedure which were not required by the original procedure body (i.e. the specification of the procedure), and which may not be preserved by alternative implementations.

For a more realistic example, suppose we have a non-deterministic specification for a program to locate some occurrence of x in an array A (assuming, for simplicity, that there is at least one such occurrence), and we choose to implement this specification as a procedure, using a linear search algorithm. Using the above laws, we could later introduce a call on this procedure in a situation which relied on finding the first occurrence of x in A. But that refinement could be invalidated if we subsequently replaced the implementation of the procedure by a different search algorithm.

4 Some alternatives

The problems illustrated above occurred because when we refine the body of a procedure, using the *Monotonicity of Procedure Body* law, the name of the procedure becomes associated with the implementation, rather than the specification (the original body). In order to ensure that calls on a procedure are based on the specification, rather than the implementation, we must ensure that the procedure name is associated with the specification in some way when a call is introduced. There are several ways in which we might consider achieving this. For example:

- We could insist that all calls on a procedure be introduced before the procedure is implemented. This may seem sensible for a strict top-down development, but is far too restrictive in general. It does not allow us to exploit libraries of previously written procedures, and does not account for the kinds of modifications we might make during program maintenance.

[5]More formally, in place of $\ldots X \ldots$, write $\mathcal{C}[X]$ for suitable program schema $\mathcal{C}$.

- We could keep both the specification and the implementation of a procedure as separate components, so that the specification remains unchanged when the body is implemented. Back [2] proposes a structure like this, which also allows alternative implementations of procedures. In our notation, Back's structure looks something like:

$$
\begin{array}{l}
\llbracket\ \mathbf{proc}\ p\ \mathrel{\widehat{=}}\ T\ \bullet \\
\qquad S \\
\quad \mathbf{where} \\
\qquad p \sqsubseteq \mathbf{proc}\ p'\ \mathrel{\widehat{=}}\ T' \\
\rrbracket
\end{array}
$$

  where T is the specification of the procedure and T' is its implementation; p' is a name associated with the implementation which can be used for describing alternative implementations. Although Back doesn't indicate how this construct would be used, one would presumably require that T' be a refinement of T, and allow further refinement of S and T', but not of T. A similar approach is being taken in the PRT refinement tool being developed at the University of Queensland [27]. This approach is closer to the way procedures are handled in Hoare logic [15], and has been incorporated in programming languages such as Alphard [31] and Euclid [17]. From a refinement point of view, it can be seen as recording information about the development of a program as part of the program text; this information is irrelevant to the final program and would be ignored by a compiler.

- We could regard a procedure declaration as a meta-linguistic device which adds a new primitive statement to the programming language, for which we also add a new refinement rule. Thus, on seeing a procedure declaration **proc** $p \mathrel{\widehat{=}} T$, we add a rule $T \sqsubseteq p$ to our set of available refinement rules. We can then refine the procedure body as before, but use this new rule, rather than *Call Procedure 1* (or *Call Procedure 2*) to introduce calls on p.

- The treatment of procedures given in Vickers and Morgan [29] can be modified to give a rule in which the binding for a procedure name doesn't change when the procedure body gets refined. Their approach, however, is based on a modified weakest precondition, indexed by a context and an environment. While that may ultimately prove to be necessary and/or desirable, we prefer to explore possibilities within standard weakest precondition semantics at present.

5 The role of procedures in program refinement

Before choosing one of these alternatives, or some other, we should pause to review just what we are trying to achieve. In particular, we need to reconsider why we use procedures, and how we want to use them within a formal development process.

In this discussion, we are primarily concerned with procedures that are both declared and used within the same program component (i.e. within the same development). Thus, we will not address the situation where the program being developed is the implementation of a procedure which has been specified as part of a higher-level design (e.g. based on a schema in a Z specification), or with implementing a procedure to be provided as part of a library for others to use. In both cases, the decision to create a procedure is made externally to the part of the development process we are considering, and the procedure will not be used within the program being implemented. Similarly, we are not concerned with using a procedure which is already part of an available library.

We indicated in Section 1 that procedures are used for several different purposes. Some of these are to do with program design (abstraction, hiding implementation details); some are to do with program understanding, modification and reuse; some are to do with implementation considerations (reducing size of compiled code). Although, from a practical point of view, there are advantages in gaining all of these benefits from a single mechanism, it is important to be able to separate them when necessary ([19], [13]). In particular, since our main concern is with the process of formal program derivation, we wish to separate concerns that are relevant during program derivation from considerations that are relevant only to the final program.

Certainly considerations of the size of the compiled code are only relevant to the final program (indeed, they should be the concern of the compiler, not the programmer). Considerations of size and structure of the program text are also essentially to do with the final program, where we are concerned about understandability and adaptability — though similar concerns may apply to a partially completed program during program derivation.

According to Morgan [20, Chap. 11], the most significant advantage of using procedures is that they avoid repeating the refinement of a specification every time it occurs. In the context of a refinement tool, however, this can be achieved in other ways. If we notice that a specification we wish to refine has already been refined elsewhere, and we are content to refine it in the same way, we can simply copy the necessary part of the refinement (this may be done, for example, by copying part of a refinement script [9]). Alternatively, if we expect the same refinement to be required many times, we may introduce a derived rule [28] or tactic ([11], [26]), which can then be invoked by name just as a procedure can. We have already seen that a procedure declaration can be thought of as adding a refinement rule; this reinforces the view that procedures should be somehow regarded as a meta-linguistic device.

Given that the refinement calculus already provides procedural abstractions, in the form of specification statements, and we are already able to break a derivation into logically separate chunks, we do not need to use procedures for that purpose. The ability to name these procedural abstractions, and to refer to them by name, is useful to the programmer in constructing a program, and is likely to be helpful to someone else trying to understand the program and its derivation. Labelling procedural abstractions, however, does not require the use of procedures (in the sense that they are normally understood), though procedures are a traditional way of providing this facility. In the absence of recursion, we can view procedures as a meta-linguistic feature, allowing names to be associated with program fragments, which may be thought of as procedural abstractions, rather like macros.

Thus, we see that while procedures are clearly important in structuring the resulting program, the main advantage associated with procedures during program construction is that of labelling procedural abstractions — which can be done without procedures. Indeed, we claim that it is desirable to separate these ideas, since it is often useful to be able to label procedural abstractions in a refinement without making any commitment at that point to implementing the abstraction as a procedure. This is often done in an informal way. For example, in Wirth's step-wise refinement [30], a task is broken down into named steps which are then elaborated separately; any of these may end up being implemented as procedures. Dijkstra [5] presents parts of a program as fragments with descriptive labels and assumes that these will be assembled somehow in order to obtain the final program. Knuth's literate programming [16] provides a more systematic way of doing this.

Morgan's approach to handling parameters ([19], [20, Chap. 11]) is also consistent with this approach. After introducing procedure calls using substitutions, Morgan later decides whether to parameterise the procedure.

6 A new approach?

We propose an alternative way of handling procedures in the refinement calculus which separates the design of a program from the construction of the resulting program. Within the design, program components can be labelled and these labels can be used within other program components. A label associated with a specification remains associated with that specification even though it may be refined to an implementation. The decision as to whether the implementation of a labelled specification is coded in-line or as a procedure can be deferred until the design is complete.

In presenting a derivation, we can associate a name, n, with any program fragment, S, using the notation:

$$n :: \; S$$

In subsequent text, any occurrence of n is understood to refer to S.

This is the way that Hehner [12] treats procedures, and is essentially the way that labels are associated with abstractions by Wirth [30] and Dijkstra [5]. Also note that we already use a convention for labelling specifications in refinements anyway — this makes the association more precise. This notation can also be used in conjunction with Morgan's substitution notation (see refinement of $Sort3$ below).

To illustrate this approach, we present a version of the derivation of the program to sort three numbers, following the same steps as in Section 3.1:

$$Sort :: \; p, q, r\colon \left[true \; \Big/ \; \begin{array}{c} p \leq q \leq r \\ \{\!\mid p, q, r \mid\!\} = \{\!\mid p_0, q_0, r_0 \mid\!\} \end{array} \right]$$

$$\begin{array}{lll}
(Sort) & \sqsubseteq & Sort1 :: \; p, q := p \sqcap q, p \sqcup q; \\
 & & Sort2 :: \; p, q, r := p \sqcap r, q \sqcap (p \sqcup r), q \sqcup r \\
(Sort1) & \sqsubseteq & \mathbf{if}\ p \leq q \rightarrow \mathbf{skip} \ [\!]\ p \geq q \rightarrow p, q := q, p\ \mathbf{fi} \\
(Sort2) & \sqsubseteq & Sort3 :: \; q, r := q \sqcap r, q \sqcup r; \\
 & & Sort4 :: \; p, q := p \sqcap q, p \sqcup q \\
(Sort3) & \sqsubseteq & Sort1[\ \mathbf{value\ result}\ p, q \setminus q, r] \\
(Sort4) & \sqsubseteq & Sort1
\end{array}$$

In this derivation, we have gained the same benefit of labelling abstraction that we obtained before using a procedure; in particular, we have reused the refinement of $Sort1$ in our refinements of $Sort3$ and $Sort4$. But we have not yet committed ourselves to using any procedures.

From this derivation, we can now extract several different programs, according to which of the labelled constructs we choose to package as procedures.

We might choose not to use any procedures, in which case where the same fragment is the result of two refinements, it is duplicated (with substitutions if necessary). In this case, we obtain (after minor simplification) the program:

$$
\begin{array}{l}
\mathbf{if}\ p \le q \to \mathbf{skip} \ [\!]\ p \ge q \to p, q := q, p\ \mathbf{fi};\\
\mathbf{if}\ q \le r \to \mathbf{skip} \ [\!]\ q \ge r \to q, r := r, q\ \mathbf{fi};\\
\mathbf{if}\ p \le q \to \mathbf{skip} \ [\!]\ p \ge q \to p, q := q, p\ \mathbf{fi}
\end{array}
$$

If we choose to implement just *Sort*1 as a procedure, and to parameterise this procedure, so as to accommodate the refinement of *Sort*3, we end up with the following, which is the same result as Morgan's:

$$
\begin{array}{l}
[\![\ \mathbf{proc}\ \mathit{Sort}(\mathbf{value\ result}\ x, y)\ \widehat{=}\\
\qquad \mathbf{if}\ x \le y \to \mathbf{skip}\ [\!]\ x \ge y \to x, y := y, x\ \mathbf{fi}\ \bullet\\
\quad \mathit{Sort}(p, q);\\
\quad \mathit{Sort}(q, r);\\
\quad \mathit{Sort}(p, q)\\
]\!]
\end{array}
$$

Alternatively, we might (just to be perverse!) choose to implement all of the named specifications as procedures, but to effect the substitution in the refinement of *Sort*3 by constructing a variant of the refinement of *Sort*1 rather than introducing parameters. Assuming that all the resulting procedures are declared in a single block, rather than being nested, we end up with:

$$
\begin{array}{l}
[\![\ \mathbf{proc}\ \mathit{Sort}\ \widehat{=}\ \mathit{Sort}1;\ \mathit{Sort}2;\\
\quad \mathbf{proc}\ \mathit{Sort}1\ \widehat{=}\ \mathbf{if}\ p \le q \to \mathbf{skip}\ [\!]\ p \ge q \to p, q := q, p\ \mathbf{fi};\\
\quad \mathbf{proc}\ \mathit{Sort}2\ \widehat{=}\ \mathit{Sort}3;\ \mathit{Sort}4;\\
\quad \mathbf{proc}\ \mathit{Sort}3\ \widehat{=}\ \mathbf{if}\ q \le r \to \mathbf{skip}\ [\!]\ q \ge r \to q, r := r, q\ \mathbf{fi};\\
\quad \mathbf{proc}\ \mathit{Sort}4\ \widehat{=}\ \mathit{Sort}1\ \bullet\\
\quad \mathit{Sort}\\
]\!]
\end{array}
$$

The three programs shown above (and several other possibilities) differ only in which specifications are packaged as procedures, and should be considered as different manifestations of the same basic design. With our approach, the decision as to which program is produced is separated from the refinement steps leading to this design.

In general, we regard a derivation as a collection of refinement pairs, $A \sqsubseteq B$, where A may be just a program label. To construct a program from such a derivation, we also provide a set of annotations or coding cues indicating which labelled components are to be packaged as procedures, and how they are to parameterised. The annotations might also explicitly indicate nesting levels for procedure declarations.

7 Tool support

We now consider how this approach to using procedures can be supported within a refinement tool.

First, we need a way of annotating labelled specifications to indicate how they are to be coded. This can be done using a simple command line interface, or using a point and click dialogue in a refinement tool with a window interface — this would work well, for example, if refinements were presented as structure diagrams [10]. Annotations can be provided or changed at any time, so need not interfere with the basic refinement process.

Next, we need an algorithm to construct the resulting program from a derivation and a set of annotations. This requires a variant of the usual collector, which traverses a refinement to construct the resulting program (see [26, Chap. 8]). Whereas the standard collector assumes that every refinement step corresponds to an application of a monotonicity or transitivity law for refinement, the new collector can instead treat a refinement step as an application of a procedure introduction law, as guided by the coding cues. Where the target of a refinement step is a label, the collector will introduce a procedure call if the specification referred to is marked as a procedure, and will copy the result of the refinement otherwise.

If substitution is used with a specification that is marked as a procedure, the collector will associate parameters with the procedure declaration and calls. Otherwise, the collector will insert code to declare local variables and make textual substitutions as necessary (see [20, Chap. 11]).

The collector can also check for cycles in a derivation. On detecting a cycle, the tool will ask for a variant to prove termination, and construct a recursion block or a recursive procedure. The collector could also have the option of introducing a loop where the recursion is linear.

Note that decisions about what specifications should correspond to procedures do not have to wait until the derivation is complete. We can modify the annotations and generate the corresponding version of the (possibly incomplete) program at any time. We can, for example, view different versions of a program obtained by either calling procedures or expanding the derivation in-line, which may be helpful for understanding dynamic properties of the program.

8 Conclusions

Our approach to handling procedures takes the separation of concerns advocated by Morgan [19] a step further, so that basic problem solving and algorithm design are separated from more concrete issues of coding and packaging of the final program. This allows us to concentrate, during program refinement, on assembling a collection of theorems (instances of the refinement relation and their proof obligations) that demonstrate that the initial problem can be solved constructively, and defer some questions of just how these results are combined to obtain the final program. This approach is (as far as I know) new in the context of the refinement calculus, but in fact draws on ideas that are much older[6] (e.g. [30], [5], [12]).

Our approach also shows that different programs can be obtained from the same design, simply by taking different coding options. We believe that this provides greater insight into the relationships between these programs than constructing one program and then transforming it into an equivalent version in a different style. This is also more attractive from a practical point of view. Various coding strategies can be built into a refinement tool, allowing it to assume more of the burden of handling tedious detail and allowing the programmer to concentrate on more important design decisions.

[6] Hence the "?" in the title!

Although we have developed this approach from consideration of handling procedures, it also presents interesting possibilities for handling other programming constructs and development styles. For example, we can allow a derivation to include alternative refinements, $S \sqsubseteq S_1, \cdots, S \sqsubseteq S_n$, for a specification S, and subsequently code these as an **if** statement, **if** $[\!]_i\ S_i$ **fi** (after ensuring that $[\!]_i\ S_i$ is non-miraculous). Thus, we can support the conditional refinement techniques described in [25]. We should also be able to support the kind of development style discussed by Naftalin [24], but we have not yet considered that in detail.

The most important lesson to be drawn from this paper, however, is about the relationship between refinement theory and methodology. In designing a formal development method, we must consider carefully the practical implications of the theorems we prove. While any refinement theorem can be used to justify transforming one program into another, it appears that some theorems should not in fact be used in this way!

Acknowledgements

Thanks to Gill Dobbie, Ian Hayes, Ray Nickson and Steve Reeves for comments on earlier drafts of this paper, and to Andrew Vignaux who implemented an version of the collector described in Section 7.

References

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

[2] R. J. R. Back. "Procedural Abstraction in the Refinement Calculus". Reports on computer science and mathematics 55, Abo Akademi, 1987.

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

[4] D. Carrington, I. Hayes, R. Nickson, G. Watson and J. Welsh. "A tool for developing correct programs by refinement". Technical Report 95–49, Software Verification Research Centre, University of Queensland, 1995.

[5] E.W. Dijkstra. "Notes on Structured Programming". In O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.

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

[7] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[8] D. Gries and G. Levin. "Assignment and procedure call proof rules". *TOPLAS* **2** (1980), pp564–579.

[9] Lindsay Groves. "Deriving Programs by Combining and Adapting Refinement Scripts". *Proc. 1995 Asia Pacific Software Engineering Conference*, IEEE Computer Society Press, 1995, pp354–363.

[10] Lindsay Groves. "Program Refinement with Structure Diagrams". *5th Australasian Refinement Workshop*, Brisbane, April 1996.

[11] Lindsay Groves, Raymond Nickson and Mark Utting. "A Tactic Driven Refinement Tool". *Proc. 5th Refinement Workshop*, Springer-Verlag, 1992, pp272–297.

[12] Eric C. R. Hehner. "**do** considered **od**: A contribution to the programming calculus". *Acta Informatica* **11** (1979), pp287–304.

[13] Eric C. R. Hehner. *The Logic of Programming.* Prentice-Hall, 1984.

[14] C.A.R. Hoare. "An axiomatic basis for computer programming". *C.A.C.M.* **12** (1969), pp576–583.

[15] C.A.R. Hoare. "Procedures and parameters: an axiomatic approach". In E. Engeler (ed.), *Symposium On Semantics of Algorithmic Languages*, Lecture Notes in Mathematics 188, Springer-Verlag, 1971, pp102-116.

[16] Donald E. Knuth. "Literate programming". *The Computer Journal* **27**, 2 (May 1984), pp97-111.

[17] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek. "Report on the Programming Language Euclid". *SIGPLAN Notices* **12**, 2 (Feb 1977), pp1-79.

[18] Carroll Morgan. "The specification statement". *TOPLAS* **10**, 3 (July 1988). Also in [22], pp7–30.

[19] Carroll Morgan. "Procedures, parameters, and abstraction: separate concerns". *Sci. Comp. Prog.* **11** (1988). Also in [22], pp58–71.

[20] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, Second Edition, 1994.

[21] Carroll Morgan and Ken Robinson. "Specification statements and refinement". *IBM Jnl. Res. Dev.* **31**,5 (Sept. 1987). Also in [22], pp31–57.

[22] Carroll Morgan and Trevor Vickers (Eds). *On the Refinement Calculus.* Springer-Verlag, 1994.

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

[24] M. Naftalin. "Informal Strategies in Design by Refinement". *Proc. Formal Methods Europe 1994.*

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

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

[27] Raymond Nickson. Private communication. November, 1995.

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

[29] Trevor Vickers and Carroll Morgan. Procedures and invariants in the refinement calculus. Technical Report TR-CS-94-04, Australian National University, May 1994.

[30] N. Wirth. "Program development by stepwise refinement". *CACM* **14** (April 1971), pp221–227.

[31] W. A. Wulf, R. L. London and M. Shaw. "An Introduction to the Construction and Verification of Alphard Programs". *IEEE Transactions on Software Engineering* **2** (1976), pp253–265.