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

Eliminating intermediate data structures by data refinement

Lindsay Groves

Technical Report CS-TR-96/6
July 1996

Abstract

Many computing problems can be described as a composition of two relations (or functions) linked by an intermediate data structure, such as a list or set. In the interests of separation of concerns and reuse of general techniques, it is often desirable to implement the two relations separately and combine their implementations. In the interests of efficiency, however, it is often desirable to eliminate the intermediate data structure from the implementation, and this is vital in cases where the intermediate structure is infinite. Various techniques have been developed for eliminating (or at least restricting) intermediate structures in functional and logic programs, including the use of special execution mechanisms (e.g. lazy evaluation and coroutining of goals) and fold/unfold transformations. In formal derivation of imperative programs it appears more common to consider both relations together from the outset, with consequent loss of separation of concerns and reusability of general derivations.

In this paper, we show that intermediate data structures can be eliminated using data refinement. This allows separate relations in the specification to be considered separately in deriving an algorithm, while still arriving at an efficient implementation. It is hoped that this will make it easier to reuse general derivation strategies.

Publishing Information

Fifth Australasian Refinement Workshop, Brisbane, April 1996.

Eliminating intermediate data structures by data refinement

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

Abstract

Many computing problems can be described as a composition of two relations (or functions) linked by an intermediate data structure, such as a list or set. In the interests of separation of concerns and reuse of general techniques, it is often desirable to implement the two relations separately and combine their implementations. In the interests of efficiency, however, it is often desirable to eliminate the intermediate data structure from the implementation, and this is vital in cases where the intermediate structure is infinite. Various techniques have been developed for eliminating (or at least restricting) intermediate structures in functional and logic programs, including the use of special execution mechanisms (e.g. lazy evaluation and coroutining of goals) and fold/unfold transformations. In formal derivation of imperative programs it appears more common to consider both relations together from the outset, with consequent loss of separation of concerns and reusability of general derivations.

In this paper, we show that intermediate data structures can be eliminated using data refinement. This allows separate relations in the specification to be considered separately in deriving an algorithm, while still arriving at an efficient implementation. It is hoped that this will make it easier to reuse general derivation strategies.

1 Introduction

In deriving programs from specifications, we often encounter specifications in which two relations (or functions) are linked by an intermediate data structure, such as a list or set. In implementing such a specification, we generally have to choose one of two approaches. We can derive algorithms for the two relations separately, and compose them in the same way that they are composed in the specification. Or we can attempt to address both relations at once and avoid having to create the intermediate data structure. The first approach is attractive from the point of view of separation of concerns, and often allows reuse of standard techniques, but is often inefficient because of the need to create and then traverse or deconstruct the intermediate data structure. The second approach is exactly the opposite: it is attractive because the resulting implementation is more efficient, and unattractive because the derivation is more complex and can use fewer standard techniques.

A number of techniques have been developed which may provide the benefits of both approaches. In functional and logic programs, we can use special execution mechanisms, such as lazy evaluation and coroutining of goals, to avoid creating parts of an intermediate data structure that are not needed. We can also use transformations, such as fold/unfold, to remove intermediate data structures. Techniques such as coroutines and iterators may be able to provide these advantages in imperative programs, but it is not clear how they can be incorporated into a formal derivation process.

In this paper, we present an approach to deriving imperative programs within the refinement calculus (e.g. [Morgan, 1994], [Morgan and Vickers, 1994]) which allows separate relations in a specification to be considered separately in deriving an algorithm, while still arriving at an efficient implementation. Our approach is based on the use of data refinement (e.g. [Morgan, 1988], [Morgan & Gardiner, 1990], [Morris, 1989]) in which the data structure represented is implicit, and its elements are constructed by the accessing operations.

In Section 2, we present a simple example showing how intermediate data structures arise in the situtations described above, and in Section 3 we show the typical way in which intermediate data structures are avoided in such situtations. In Section 4, we show how our approach can be applied to remove the intermediate data structure in this simple example, and compare two variants of the approach. In Section 5 we apply our approach to a more interesting example, and in Section 6 we present out conclusions.

2 Intermediate data structures

Suppose we want a program to find the sum of the squares of the first n positive integers.[1]

In a functional language, we might write $ss(n) = sum(squares(n))$, where $squares(n)$ returns the squares of the first n positive integers (as a list) and $sum(l)$ returns the sum of the elements in list l.[2] In Prolog, we might write $ss(N, R)$:- $squares(N, L), sum(L, R)$, again using a list for the intermediate structure. In Z, we could also define sum and $squares$ as functions, or we might define sum and $squares$ as separate schemas and combine them using schema composition; we can also model the intermediate result more accurately as a set rather than a list, since no two positive integers have the same square and the order in which numbers are squared and summed is unimportant.

In each case, we decompose the problem into two parts: a "*producer*" (*squares*), which creates an intermediate data structure, and a "*consumer*" (*sum*), which processes this data structure to compute the desired result. It would be quite natural in this example, and in others like it, to develop algorithms for the producer and consumer separately and combine them.

In the refinement calculus (e.g. [Morgan, 1994], [Morgan and Vickers, 1994]), we might begin with a specification statement:[3]

$$r\colon \left[r = \textstyle\sum\{x^2 \mid 1 \le x \le n\} \right] \qquad (i)$$

This specification can be implemented by splitting the set construction and summation into separate conjuncts which can be addressed by separate statements:

$$(i) \quad \sqsubseteq \quad (\mathit{Equivalent\ Postcondition})$$
$$r\colon \left[\exists s \bullet s = \{x^2 \mid 1 \le x \le n\} \land r = \textstyle\sum s \right] \qquad (ii)$$

[1] This is a popular, if rather trivial example, but serves to demonstrate the main ideas.

[2] In fact, we might define *squares* as the composition of a function which constructs a list of the first n positive integers and one which creates a list of the squares of the numbers in a given list. We might also define *squares* using a map and *sum* using a fold operation.

[3] We will assume that n and all other numbers used are naturals.

$(ii) \quad \sqsubseteq \quad$ (*Replace Existential by Local Variable*)[4]

$$\begin{array}{l} \textbf{var } s \bullet \\ r\colon \left[s = \{x^2 \mid 1 \leq x \leq n\} \wedge r = \sum s \right] \end{array} \qquad (iii)$$

$(iii) \quad \sqsubseteq \quad$ (*Split Specification*)

$$s\colon \left[s = \{x^2 \mid 1 \leq x \leq n\} \right]; \qquad (iv)$$

$$r\colon \left[s = \{x^2 \mid 1 \leq x \leq n\} \;\big/\; r = \sum s \right] \qquad (v)$$

We can now implement (iv) and (v) separately. In each case the development is quite straightforward, requiring a loop with a simple invariant that can be obtained using standard strategies. The loop invariant for the producer (iv) is:

$$0 \leq k \leq n \wedge s = \{x^2 \mid 1 \leq x \leq k\}$$

The loop invariant for the consumer (v) is:

$$r = \sum (s_0 - s)$$

where s_0 is the value of s at the beginning of the loop.

We would probably do these refinements by copying or mimicking previously worked out derivations. In a refinement tool (e.g. [Vickers, 1990], [Groves, Nickson and Utting, 1992], citeNickson:thesis), we might well have tactics, scripts or derived rules which perform these derivations.

The resulting program is:

$$\begin{array}{l}
\llbracket\ \textbf{var } s \bullet \\
\quad \llbracket\ \textbf{var } k \bullet \\
\quad\quad k := 0;\ s := \varnothing; \\
\quad\quad \textbf{do } k \neq n \rightarrow \\
\quad\quad\quad s := s \cup \{(k+1)^2\}; \\
\quad\quad\quad k := k + 1 \\
\quad\quad \textbf{od} \\
\quad \rrbracket; \\
\quad r := 0; \\
\quad \textbf{do } s \neq \varnothing \rightarrow \\
\quad\quad s := s - \{\mathit{Choose}(s)\}; \\
\quad\quad r := r + \mathit{Choose}(s) \\
\quad \textbf{od} \\
\rrbracket
\end{array}$$

where $\mathit{Choose}(s)$ returns an arbitrary element of s.[5]

[4] See Appendix.

[5] Note that since *Choose* is a function, two occurrences of $\mathit{Choose}(s)$ will return the same value.

Developing the algorithm in this way allows the producer and consumer to be developed separately, achieving separation of concerns. But this at the cost of creating the intermediate structure s and then immediately taking it apart again. Since the consumer only needs to process each element of s once, and can process elements in the same order that the producer produces them, we should be able to somehow combine producer and consumer into a single loop, or perhaps interleave their execution.

3 Avoiding intermediate data structures

One way of avoiding intermediate data structures in deriving programs like this is to simply develop a single loop from the outset. In this case the loop invariant is more complicated, and harder to justify/motivate, since it needs to address the requirements of both producer and consumer at once. Essentially, we need to anticipate the effect of creating the intermediate data structure and work out how to get by without it!

In the example above, we anticipate that elements of $\{x^2 \mid 1 \leq x \leq n\}$ can be generated and summed in a single loop, and choose the following loop invariant:

$$0 \leq k \leq n \wedge r = \sum\{x^2 \mid 1 \leq x \leq k\}$$

and thus refine (i) to:

$$\begin{array}{l} [\![\ \textbf{var}\ k \bullet \\ \quad k, r\colon \left[0 \leq k \leq n \wedge r = \sum\{x^2 \mid 1 \leq x \leq k\}\right]; \\ \quad k, r\colon \left[0 \leq k \leq n \wedge r = \sum\{x^2 \mid 1 \leq x \leq k\} \Big/ r = \sum\{x^2 \mid 1 \leq x \leq n\}\right] \\ ]\!] \end{array}$$

In implementing the loop body (assuming that we developing the loop a Dijkstra/Gries-like manner [Dijkstra, 1976], [Gries, 1981]), we end up having to implement the specification:

$$r\colon \left[0 \leq k < n \wedge r = \sum\{x^2 \mid 1 \leq x \leq k\} \Big/ 0 \leq k{+}1 \leq n \wedge r = \sum\{x^2 \mid 1 \leq x \leq k{+}1\}\right]$$

This can be implemented by $r := r{+}(k{+}1)^2$, but the reasoning required to show (or realise) that is certainly more complicated than any of the reasoning required in the previous version. Although in this example, neither derivation is especially complicated, in more complex examples, the difference would be more marked.

The resulting program is:

$$\begin{array}{l} [\![\ \textbf{var}\ k \bullet \\ \quad k := 0;\ r := 0; \\ \quad \textbf{do}\ k \neq n \rightarrow \\ \qquad r := r + (k+1)^2; \\ \qquad k := k + 1 \\ \quad \textbf{od} \\ ]\!] \end{array}$$

In this development we have avoided introducing an intermediate data structure, at the cost of explicitly addressing both producer and consumer together. The question is: Can we achieve the efficiency of this implementation, while retaining the separation of concerns in the development that we had in the initial version?

4 Eliminating intermediate data structures

In functional or logic programming languages, we can sometimes reduce the overhead of creating and deconstructing intermediate data structures by using special evaluation methods, such as lazy evaluation or coroutining. These have the effect of interleaving execution of producer and consumer, but still requires the intermediate data structure to be constructed and disassembled. These approaches are thus only an improvement if the consumer does not need to traverse all of the intermediate structure and can thus foreshorten its construction (this is especially useful when the intermediate structure is infinite). These methods also impose additional housekeeping overhead which may compromise efficiency.

In functional and logic programming languages, we can also often eliminate intermediate data structures by performing transformations, such as fold/unfold. This approach allows the kind of development we are looking for: in the initial development, the producer and consumer can be derived separately, and transformations are then used to remove the intermediate data structure and modify the producer and consumer so that they operate in tandem. These techniques often only work for programs in limited forms; most assume that the intermediate data structure is a list, though some will work for trees.

In an imperative language, such as is used in the refinement calculus, the corresponding transformation would be to merge two loops. While this is certainly possible (and such "loop jamming" transformations are used in optimising compilers), they again only work for loops in particular forms. We would prefer to find a more general approach which can be integrated more naturally in the development method.

The approach we propose is to eliminate the intermediate data structure using data refinement (e.g. [Morgan, 1988], [Morgan & Gardiner, 1990], [Morris, 1989]). We begin as in Section 2 and then expand either producer (vi) or consumer (v), and then perform a data refinement. The basic idea is to replace the intermediate data structure, which may be a composite structure such as a set, bag or list, by a much simpler representation, and make the operation of adding an element to the intermediate data structure, or of selecting and removing an element from this structure, do the work of computing the next element.

4.1 Data refining the producer

Suppose we begin as in Section 2 and expand the producer (vi), but not the consumer (v). This gives:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ s \bullet \\
\quad [\![\ \mathbf{var}\ k \bullet \\
\qquad k := 0;\ s := \varnothing; \\
\qquad \mathbf{do}\ k \neq n \rightarrow \\
\qquad\qquad s := s \cup \{(k+1)^2\}; \\
\qquad\qquad k := k+1 \\
\qquad \mathbf{od} \\
\quad]\!]; \\
\quad r\colon \left[\, s = \{x^2 \mid 1 \leq x \leq n\} \;\middle/\; r = \sum s \,\right] \\
]\!]
\end{array}
$$

We can now perform a data refinement to implement s by r, so that r is the sum of the elements in s, i.e. with coupling invariant:

$$r = \sum s$$

Performing the data refinement, we delete the declaration of s (since r is global, it doesn't need to be declared), and make the following replacements:[6]

$$
\begin{array}{lcl}
s := \varnothing & \longrightarrow & r := 0 \\
s := s \cup \{(k+1)^2\} & \longrightarrow & r := r + (k+1)^2 \\
r\colon \left[\, s = \{x^2 \mid 1 \leq x \leq n\} \;\middle/\; r = \sum s \,\right] & \longrightarrow & \mathbf{skip}
\end{array}
$$

The resulting program is:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ k \bullet \\
\quad k := 0;\ r := 0; \\
\quad \mathbf{do}\ k \neq n \rightarrow \\
\qquad r := r + (k+1)^2; \\
\qquad k := k+1 \\
\quad \mathbf{od} \\
]\!]; \\
\mathbf{skip}
\end{array}
$$

This can now be simplified to remove the **skip** statement, giving the same program that we obtained in Section 3.

[6]The data refinements are not explained in detail here; they can be derived using standard techniques, such as those described in [Morgan, 1988] or [Morgan & Gardiner, 1990], [Morris, 1989].

4.2 Data refining the consumer

Now suppose we begin as in Section 2 and expand the consumer (v), but not the producer (vi). This gives:

$$
\begin{array}{l}
[\![\ \textbf{var}\ s\ \bullet \\
\quad s\colon\left[s = \{x^2 \mid 1 \le x \le n\} \right]; \\
\quad r := 0; \\
\quad \textbf{do}\ s \ne \varnothing \rightarrow \\
\qquad r := r + \mathit{Choose}(s); \\
\qquad s := s - \{\mathit{Choose}(s)\} \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

We can now perform a data refinement to implement s by a variable k, so that s is the set of squares of numbers from $k+1$ to n, i.e. with coupling invariant:

$$0 \le k \le n \wedge s = \{x^2 \mid k+1 \le x \le n\}$$

Performing the data refinement, we replace the declaration of s by a declaration of k, and make the following replacements:

$$
\begin{array}{lcl}
s\colon\left[s = \{x^2 \mid 1 \le x \le n\} \right] & \longrightarrow & k := 0 \\
s \ne \varnothing & \longrightarrow & k \ne n \\
r := r + \mathit{Choose}(s) & \longrightarrow & r := r + (k+1)^2 \\
s := s - \{\mathit{Choose}(s)\} & \longrightarrow & k := k+1
\end{array}
$$

In performing this data refinement, we have chosen to implement $\mathit{Choose}(s)$ by returning the smallest element in s, viz. $(k+1)^2$, which then allows $s := s - \{\mathit{Choose}(s)\}$ to be implemented by $k := k+1$.

The resulting program is:

$$
\begin{array}{l}
[\![\ \textbf{var}\ k\ \bullet \\
\quad k := 0;\ r := 0; \\
\quad \textbf{do}\ k \ne n \rightarrow \\
\qquad r := r + (k+1)^2; \\
\qquad k := k+1 \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

This is now the same program that we obtained in Section 3.

4.3 Comparison

In the above derivations, we ended up with the same program as we did by developing a single loop directly (Section 3), but the reasoning is simpler. In both cases, the loop we developed has a simpler invariant than the one in Section 3, because it is establishing a simpler postcondition. The additional complexity is handled by the data refinement step, but this is done incrementally so the burden is lessened.

It is interesting to compare the loop invariants and coupling invariants for the two versions:

	Loop invariant	Coupling invariant
Producer	$0 \leq k \leq n \wedge s = \{x^2 \mid 1 \leq x \leq k\}$	$r = \sum s$
Consumer	$r = \sum(s_0 - s)$	$0 \leq k \leq n \wedge s = \{x^2 \mid k+1 \leq x \leq n\}$

Thus, the relationship between the two versions is essentially a swapping in the roles of the loop invariant and coupling invariant, with both being complemented in the consumer (i.e. sum of $s_0 - s$ rather than of s, and k being the set of squares of numbers from $k+1$ to n, rather than of 1 to k).

For this example, it makes little difference whether we do the data refinement on the producer or on the consumer. In the next section, we will see an example where this choice is more important.

5 An example using searching

We will now consider an example where the consumer is a search that may terminate before it has processed all of the intermediate structure. In this case, data refining the consumer has definite advantages, since we can avoid creating all of the elements of the intermediate structure.

Suppose we want a program to determine whether all the elements in an array $a[1\,..\,n]$ are different. Clearly, we need to compare pairs of elements in a until we have compared all possible pairs or until we find an identical pair. Since equality is symmetric, we only need to consider pairs where the indexes are in ascending order. We thus begin with the specification:

$$r\colon\left[\, r \equiv (\forall i, j \in 1\,..\,n \mid i < j \bullet a[i] \neq a[j]) \,\right] \qquad (i)$$

An "obvious" approach to implementing this program would be to use two loops to generate pairs of indexes in a. This involves finding two loop invariants, and arranging some way to exit from both loops if an identical pair is found, which usually involves a redundant test.

An alternative approach is to split the specification into two parts, one to construct the set of ascending pairs of indexes from $1\,..\,n$, and one to determine whether any of the pairs in this set index identical elements in a. Thus, we refine (i) as follows:

(i) $\sqsubseteq$ (*Equivalent Postcondition*)

$$r\colon\left[\, \exists p \bullet p = \{(u,v) \mid u, v \in 1\,..\,n \wedge u < v\} \wedge r \equiv (\forall (i,j) \in p \bullet a[i] \neq a[j]) \,\right] \qquad (ii)$$

(ii) $\sqsubseteq$ (*Replace Existential by Local Variable*)

var p $\bullet$

$$r\colon\left[\, p = \{(u,v) \mid u, v \in 1\,..\,n \wedge u < v\} \wedge r \equiv (\forall (i,j) \in p \bullet a[i] \neq a[j]) \,\right] \qquad (iii)$$

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

$$p\colon\Big[\, p = \{(u,v) \mid u,v \in 1\mathinner{..}n \wedge u < v\} \,\Big]\,; \qquad (iv)$$

$$r\colon\Big[\, p = \{(u,v) \mid u,v \in 1\mathinner{..}n \wedge u < v\} \;\Big/\; r \equiv (\forall (i,j) \in p \bullet a[i] \neq a[j]) \,\Big] \qquad (v)$$

We could now implement each of these specifications separately, and end up with a program using set p to communicate between the two parts. Instead, we will implement the consumer (v) and then data refine to remove p. We refine (v) with the following loop invariant:

$$p \subseteq p_0 \wedge r \equiv (\forall (i,j) \in p - p_0 \bullet a[i] \neq a[j])$$

where p_0 is the value of p at the beginning of the loop.

The resulting program (assuming suitable notation for handling pairs) is:

$$
\begin{array}{l}
[\![\ \textbf{var}\ p \bullet \\
\quad p\colon\Big[\, p = \{(u,v) \mid u,v \in 1\mathinner{..}n \wedge u < v\} \,\Big]\,; \\
\quad r := \mathit{true}; \\
\quad \textbf{do}\ p \neq \varnothing \wedge r \rightarrow \\
\qquad [\![\ \textbf{var}\ i,j \bullet \\
\qquad\quad (i,j) := \mathit{Choose}(p); \\
\qquad\quad \textbf{if}\ \ a[i] = a[j] \rightarrow r := \mathit{false} \\
\qquad\quad [\!]\ \ a[i] \neq a[j] \rightarrow \textbf{skip} \\
\qquad\quad \textbf{fi}; \\
\qquad\quad p := p - \{(i,j)\}; \\
\qquad]\!] \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

We now data refine so that p is represented by variables I and J, where p is the set of all ascending pairs that lie between (I,J) and (n,n), inclusive, in the standard lexical ordering on pairs. Thus, we used the coupling invariant:

$$(1,2) \leq (I,J) \leq (n-1,n) \wedge p = \{\, (u,v) \mid (I,J) \leq (u,v) \leq (n,n) \wedge u < v \,\}$$

where $\leq$ denotes lexical ordering on ordered pairs.

Performing the data refinement, we replace the declaration of p by declarations of I and J, and make the following replacements:

$$
\begin{array}{lcl}
p\colon\Big[\, p = \{(u,v) \mid u,v \in 1\mathinner{..}n \wedge u < v\} \,\Big] & \longrightarrow & I,J := 1,2 \\
p \neq \varnothing & \longrightarrow & I \neq n-1 \vee J \neq n \\
(i,j) := \mathit{Choose}(p) & \longrightarrow & i,j := I,J \\
p := p - \{(i,j)\} & \longrightarrow & \textbf{if}\ \ J < n \rightarrow J := J+1 \\
 & & [\!]\ \ J = n \rightarrow I,J := I+1, I+2 \\
 & & \textbf{fi}
\end{array}
$$

In performing this data refinement, we have chosen to implement $Choose(p)$ by returning the smallest element in p, viz. (I, J), which then allows $p := p - \{(i, j)\}$ to be implemented by replacing (I, J) by the next ascending pair in the lexical ordering.

The resulting program is:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ I, J \bullet \\
\quad I, J := 1, 2; \\
\quad r := true; \\
\quad \mathbf{do}\ (I \neq n-1 \vee J \neq n) \wedge r \rightarrow \\
\qquad [\![\ \mathbf{var}\ i, j \bullet \\
\qquad\quad i, j := I, J; \\
\qquad\quad \mathbf{if}\ \ a[i] = a[j] \rightarrow r := false \\
\qquad\quad [\!]\ \ a[i] \neq a[j] \rightarrow \mathbf{skip} \\
\qquad\quad \mathbf{fi}; \\
\qquad\quad \mathbf{if}\ \ J < n \rightarrow J := J + 1 \\
\qquad\quad [\!]\ \ J = n \rightarrow I, J := I + 1, I + 2 \\
\qquad\quad \mathbf{fi} \\
\qquad]\!] \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

This can now be simplified by deleting the declarations of i and j, and replacing all references to them by I and J, respectively.

This algorithm avoids the need for nested loops and the problem of exiting from nested loops. The reasoning by which we arrived at it was, we believe, simpler than that required to develop this algorithm directly, without the data refinement step.

6 Conclusions

We have shown that data refinement can be used to eliminate intermediate data structures in a way that allows separate requirements to be addressed separately during formal development, while still arriving at an efficient implementation. It is hoped that this approach will allow more extensive use of standard programming techniques, which may be stored as tactics or scripts in a refinement tool. In order for this to succeed, we will also need to develop similar kinds of support for reusing data refinements.

The data refinements used are somewhat usual, in that the data structure represented is never explicitly constructed; instead successive elements are computed by the accessing operations. The effect is rather like lazy evaluation, but does not require a language with lazy evaluation semantics. It is also similar in effect to a transformation to distribute a fold over a map operation, though more widely applicable, and to the kinds of transformation performed in partial evaluation.

In our examples, we have performed the data refinements by textual replacement. We could, instead, have implemented the intermediate data structure using calls to operations in a module with local state. In this case, the effect would be very much like using coroutines or iterators.

The examples we have studied so far have been quite simple, but suggest that the approach may be attractive for larger problems. It remains, however, to try the approach out on larger example.

Acknowledgements

The ideas contained in this paper were motivated by Ray Nickson's observation that the least common multiple of two numbers is the product of the bag maximum of their factors [Nickson, 1994]. I wanted to show this connection formally in the derivation of the *lcm* algorithm, but couldn't see how to do the data refinement, since the bags of factors were not constructed explicitly. Now I know! My thanks to Ray for providing this motivation, and for discussion of some of the results presented in the paper.

References

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

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

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

[Morgan, 1988] Carroll Morgan. "Auxiliary variables in data refinement". *Information Processing Letters* **29** (1988), pp293–296.

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

[Morgan & Gardiner, 1990] C. C. Morgan and P. H. B. Gardiner. "Data refinement by calculation". *Acta Informatica* **27** (1990), pp481–503.

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

[Morris, 1989] J. M. Morris. "Laws of data refinement". *Acta Informatica* **26** (1989), pp287–308.

[Nickson, 1994] Raymond 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, pp39–44.

Appendix

The derivations in Sections 2 and 5 use a law which allows an existentially quantified variable in the postcondition of a specification statement to be turned into a local variable, and *vice versa*. Although this law is clearly useful for this kind of derivation, I have not seen it previously, so present a proof for it here.

Law *Replace Existential by Local Variable*

If w and x are disjoint, and x is not free in pre, then

$$w\colon \left[pre \;\middle/\; \exists x \bullet post \right] \quad = \quad [\![\;\mathbf{var}\; x \bullet w\colon \left[pre \;\middle/\; post \right]]\!]$$

Proof Assume that x is not free in pre or Φ. Then:

$$\begin{array}{cl}
 & wp(w\colon \left[pre \;\middle/\; \exists x \bullet post \right], \Phi) \\
= & pre \wedge (\forall w \bullet (\exists x \bullet post) \Rightarrow \Phi) \\
= & pre \wedge (\forall w \bullet (\forall x \bullet \neg post) \vee \Phi) \\
= & (\forall x \bullet pre \wedge (\forall w \bullet \neg post \vee \Phi)) \\
= & (\forall x \bullet pre \wedge (\forall w \bullet post \Rightarrow \Phi)) \\
= & wp([\![\;\mathbf{var}\; x \bullet w\colon \left[pre \;\middle/\; post \right]]\!], \Phi)
\end{array}$$