I Went Down To The Crossroads: Conjoining Catamorphisms

Martin Schwenke

Department of Computer Science
The Australian National University
Martin.Schwenke@cs.anu.edu.au

Abstract. This paper attempts to address software specification, design and implementation reuse by bringing together work from a number of areas of program development. Refinement calculi are based on wide-spectrum languages that include abstract, logical specifications, which are transformed into executable programs. Functional and relational calculi allow programs and specifications to be manipulated using higher-order operations in powerful algebraic setting. Promotion is used in the Z specification notation to allow simple operations to be reused within a more complex framework. Specification conjunction has also been used in refinement calculi as an aid to reuse. We provide a series of examples that bring various elements of these areas together. Our examples centre on programs that can be elegantly specified by a calculation of all permutations of an input list, combined with some restriction on these permutations. This paper represents work in progress, so some of the examples are incomplete and, therefore, do not serve as convincing positive examples of the method. Also, much of the work has not been completely formalised. Our main contribution is to show that the combined approach can work in some cases and, when it does work, it is extremely profitable.

1 Introduction

The *imperative refinement calculi* of Back [2], Morgan [11] and Morris [12] allow imperative programs to be developed that are correct with respect to their specifications. The specifications are usually *logical specifications*, containing predicates that constrain their behaviour. Although these calculi meet their goal of making program development a mathematical activity, they operate at quite a *low level of abstraction*, and we believe that they are difficult to use.

Functional programming languages allow programs to be written at a *higher level of abstraction* that imperative languages. According to Hughes [8], one reason for this is the use of *higher-order functions*. Functional programming calculi, such as Bird [4], are based on higher-order functions and contain correspondingly high-order rules for manipulating programs. One reason for this is that higher-order functions provide a level of indirection when they operate on data structures, allowing for 'pointless' rules that deal with compositions of functions rather than the application of functions to variables. Relational calculi, such as

that of Aarts, Backhouse, et al [1], add nondeterminism and also provide a very high-level of abstraction. We believe that the elegance and level of abstraction found in these calculi are due to a profound understanding of the data structures involved.

There has been a substantial amount of work on refinement calculi targeted at functional programming languages (expression refinement calculi; for examples, see Norvell and Hehner [14], Ward [21], Schwenke and Robinson [17], Bunkenburg [5], Morris [13], Mahony [10], Schwenke and Mahony [16]). However, we believe that none of this work attempts to exploit the use of higher-order functions, based on known data structures, to help make refinement easier.

In the Z notation (Spivey [18]) low-level operations that operate on some type are often *promoted* so that they operate on collections involving that type. This method of specification reuse is similar to the way that functional languages allow operations to be 'plugged' into the framework provided by higher-order functions. Another method of combining specifications, which is also found in Z, is conjunction. Ward [20] and Groves [6] have explored the use of conjunction in the refinement calculus as a means of building specifications from components, and possibly reusing components. Mahony [9] provides a theoretical basis for conjunction in the refinement calculus and explores the relationship between conjunction and promotion.

We provide a series of examples involving all of these ideas. The style of presentation if fairly informal. We assume that we have an expression refinement calculus that provides all of the standard, useful refinement rules, like weakening preconditions and strengthening postconditions. Most of the expression refinement calculi mentioned above are at least partially suitable. We introduce notation and describe semantics informally. We hope to convince the reader that our examples would not be much more difficult if they were presented formally. Some of the examples are incomplete, and it is unclear whether they could actually be completed in the desired style.

2 Conjunction

Conjunction operators are well known in specification notations, but are less familiar in programming languages. We use $\cap$ to represent conjunction across our wide-spectrum language. The meaning is obvious if relational (or set-based) semantics are used, and is fairly intuitive for predicate transformer semantics. In particular, conjoining two function specifications means conjoining their preconditions and postconditions respectively.

$$\begin{array}{c}(\lambda\ x \bullet P_1 \succ \sqcap\ y \mid R_1) \\ \cap \\ (\lambda\ x \bullet P_2 \succ \sqcap\ y \mid R_2)\end{array} \quad = \quad \left(\lambda\ x \bullet \begin{pmatrix} P_1 \\ \wedge \\ P_2 \end{pmatrix} \succ \sqcap\ y \middle| \begin{matrix} R_1 \\ \wedge \\ R_2 \end{matrix}\right) \tag{1}$$

In the above function specifications, P_i represents a precondition that restricts input x, and the output y is chosen to satisfy a postcondition R_i.

We also require conjunction to be monotonic with respect to refinement.

$$\frac{a \sqsubseteq a', \quad b \sqsubseteq b'}{a \cap b \quad \sqsubseteq \quad a' \cap b'} \tag{2}$$

Our belief is that conjunction can be put to good use, 'welding' together programs of identical structure. This is particularly true when programs are written using higher-order operations. More generally, if two programs can be written using a common recursive or iterative framework, then the conjunction of the two can be written using a single instance of that framework. That is, function application distributes weakly over refinement.

$$f\ a \cap f\ b \quad \sqsubseteq \quad f\ (a \cap b) \tag{3}$$

We can take f to be a higher-order function that represents the common framework. The above condition holds under relational semantics even if f is a relation instead of a function. It probably also holds under predicate transformer semantics.

3 A Singular Insertion Sort

We provide a specification of sorting, and provide a high-level derivation that produces an *insertion sort* algorithm. We characterise the derivation in this section as being *singular* because we specify a program that produces a single output and describe how it is related to its input. This contrasts with the derivation in the next section, which explicitly uses set-valued functions.

We begin with a specification of *sort*, as follows:

$$\begin{array}{lcl} sort & :: & (Ord\ \alpha) \Rightarrow [\alpha] \rightarrow [\alpha] \\ sort & = & perm \cap ascendingOutput \end{array} \tag{4}$$

The notation used is based on Haskell [15]. The function *sort* is defined between lists of elements $[\alpha]$. The elements can be of any type that has an ordering $(Ord\ \alpha)$, permitting operations such as $<$ (less than). We use the algebraic 'cons' lists found in functional programming languages, rather than the function-based sequences found in Z. An informal interpretation of the above specification is that to sort a list we must produce an ascending permutation of the input.

3.1 Defining and Refining Permutations

The relation *perm* takes a list and returns a list containing the same items as the original.

$$\begin{array}{lcl} perm & :: & [\alpha] \rightarrow [\alpha] \\ perm & = & (\lambda\ xs \bullet (\sqcap ys \mid items\ ys = items\ xs)) \end{array} \tag{5}$$

The body of the λ-abstraction is a generalised choice, or a logical specification of an expression. Note that although we are using 'cons' lists we use a Z-style *items* function to convert lists to bags.

We assume that we can implement *perm* using *foldr*. *foldr* is probably the best known example of a *catamorphism*, which is a special type of operation that employs recursion over an inductive datatype.

$$perm \quad \sqsubseteq \quad foldr\ include\ [] \tag{6}$$

The *initial value* of the computation is the empty list, [], and *foldr* moves along the input list, using the *accumulation relation*, *include*, to add each item to its output.

The specification of *include* is quite relaxed. It insists that all the items in the input list are present in the output, along with the new item being included.

$$\begin{array}{l} include :: \alpha \to [\alpha] \to [\alpha] \\ include = (\lambda\ x\ xs \bullet (\sqcap ys \mid items\ ys = [\![x]\!] \uplus items\ xs)) \end{array} \tag{7}$$

If we are calculating permutations, it seems that an efficient refinement of *include* might be one that doesn't do any permuting of the partial results! That is, we can replace *include* by a specification that maintains the order of items in the input list, but can put the new item in any position. We call this relation *inject*.

$$\begin{array}{l} inject :: \alpha \to [\alpha] \to [\alpha] \\ inject = \left(\lambda\ x\ xs \bullet \left(\sqcap ys \;\middle|\; \left(\exists\ xs_1, xs_2 \bullet \begin{array}{c} xs = xs_1 +\!\!+ xs_2 \\ ys = xs_1 +\!\!+ [x] +\!\!+ xs_2 \end{array}\right)\right)\right) \end{array} \tag{8}$$

We claim that

$$include \quad \sqsubseteq \quad inject \tag{9}$$

and need to prove that the postcondition has been strengthened, but leave this as an exercise.

The simplest and most efficient refinement of *inject* is (:) (*cons*), which always chooses x_1 to be empty, thus adding the new value at the beginning of the input list. This results in a very boring permutation, since:

$$foldr\ (:)\ [] \quad = \quad id \tag{10}$$

That is, implementing *perm* by refining *include* to (:) simply produces the identity function. All that we have found is that a list is a permutation of itself. Although we have taken the refinement of *perm* past the point where it can be part of an implementation of *sort*, we have some useful intermediate results. Also, even though the final step is not terribly interesting, it is a good sanity check.

3.2 Ascending Output

The relation *ascendingOutput* ignores its input, but always outputs a list that is in ascending order.

$$\begin{array}{l} ascendingOutput :: (Ord\,\alpha) \Rightarrow \beta \rightarrow [\alpha] \\ ascendingOutput = (\lambda\ z \bullet (\sqcap\, ys \mid isAscending\, ys)) \end{array} \tag{11}$$

For completeness we define *isAscending* and provide an implementation.

$$\begin{array}{l} isAscending\ ::\ (Ord\,\alpha) \Rightarrow [\alpha] \rightarrow \mathbb{B} \\ isAscending\ =\ (\lambda\ xs \bullet (\forall\ i, j \mid 0 \leq i < length\, xs - 1 \bullet xs!!i \leq xs!!(i+1))) \\ isAscending \sqsubseteq\ (\lambda\ xs \bullet all\,(zipWith\,(\leq)\,xs\,(tail\,xs))) \end{array} \tag{12}$$

3.3 A Refinement Rule For *foldr*

The specification of *sort* (4) suggests that we wish to strengthen an implementation of *perm* so that its output satisfies *isAscending*. This leads us to investigate how we can refine a given specification by a *foldr* operation. Our investigations lead us very quickly to an instantiation of the structural induction proof rule for lists.

$$\frac{P(i), \quad (\forall\ x, xs \bullet P(xs) \Rightarrow P(f\ x\ xs))}{(\forall\ ys \bullet P(foldr\, f\ i\ ys))} \tag{13}$$

We can rewrite this as a refinement rule by instantiating the variables in the consequence with specifications that satisfy the conditions in the assumption.

$$\begin{array}{lcl} (\lambda\ xs \bullet (\sqcap\, ys \mid P(ys))) & \sqsubseteq & foldr\ (\lambda\ x\ xs \bullet P(xs) \succ\!\!- \sqcap\ ys \mid P(ys)) \\ & & (\sqcap\, i \mid P(i)) \end{array} \tag{14}$$

That is, we begin with a specification of a function, mapping a list to a list, where the result satisfies P. We can refine this by a *foldr* operation, as long as the initial value satisfies P, and the accumulation function maintains P.

3.4 An Ascending Permutation

We can now implement *ascendingOutput* (11) via (14).

$$\begin{array}{lcl} ascendingOutput & \sqsubseteq & foldr\ (\lambda\ x\ xs \bullet isAscending\ xs \succ\!\!- \sqcap\ ys \mid isAscending\ ys) \\ & & (\sqcap\, i \mid isAscending\, i) \end{array} \tag{15}$$

It is easy to prove that the initial value is refined by the empty list.

$$(\sqcap\, i \mid isAscending\, i) \quad \sqsubseteq \quad [] \tag{16}$$

We have now refined both conjuncts of (4). We choose *inject* as a suitable refinement of *perm*.

$$\begin{array}{lll} sort & \sqsubseteq & foldr\ \left(\lambda\ x\ xs \bullet \left(\sqcap ys \;\middle|\; \left(\exists\ xs_1, xs_2 \bullet \begin{array}{c} xs = xs_1 +\!\!+ xs_2 \\ ys = xs_1 +\!\!+ [x] +\!\!+ xs_2 \end{array}\right)\right)\right)\ [] \\ & & \cap \\ & & foldr\ \left(\lambda\ x\ xs \bullet isAscending\ xs \rightarrowtail \sqcap\ ys \,|\, isAscending\ ys\right)\ [] \end{array} \tag{17}$$

We can use (3) to produce a refinement involving a single use of *foldr*

$$sort \quad \sqsubseteq \quad foldr\ insert\ [] \tag{18}$$

and define *insert* as the conjunction of the two accumulation relations, as follows.

$$\begin{array}{lll} insert & = & \lambda\ x\ xs \bullet isAscending\ xs \rightarrowtail \\ & & \left(\sqcap ys \;\middle|\; \left(\exists\ xs_1, xs_2 \bullet \begin{array}{c} isAscending\ ys \\ xs = xs_1 +\!\!+ xs_2 \\ ys = xs_1 +\!\!+ [x] +\!\!+ xs_2 \end{array}\right)\right) \end{array} \tag{19}$$

A reasonable implementation of *insert* is

$$insert \quad \sqsubseteq \quad \left(\lambda\ x\ xs \bullet \left(\sqcap ys \;\middle|\; \left(\exists\ xs_1, xs_2 \bullet \begin{array}{c} (xs_1, xs_2) = break\ (>x)\ xs \\ ys = xs_1 +\!\!+ [x] +\!\!+ xs_2 \end{array}\right)\right)\right) \tag{20}$$

where *break* splits its input list before the first element that satisfies the given condition.

4 A Plural Insertion Sort

We use a *plural* version of insertion sort to motivate further discussion. By plural we mean that the set of all possible results is returned, which doesn't make much of a difference for sorted lists, since all results will be the same. However, there are many problems where it is useful to consider a set of possible solutions rather than a single solution. For example, we might be interested in calculating all possible permutations of a list instead of a just a single permutation.

Fortunately, we have at our disposal Λ, the *power transpose* operator (Bird and de Moor [3]), which takes a relation and turns it into a set-valued function. For the example of permutations we can simply write

$$perms \quad = \quad \Lambda\, perm \tag{21}$$

to describe the function that returns the set of all permutation of a given list.

Even more fortunately, we have a simple rule for calculating the power transpose of a relation that is expressed in terms of *foldr*.

$$\Lambda\,(foldr\ f\ i) \quad = \quad foldr\ \left(\lambda\ x \bullet \bigcup \circ\, mapSet\,(\Lambda\,(f\ x))\right)\{i\} \tag{22}$$

This is an instance of a more general rule for relational catamorphisms given by Bird and de Moor [3].

If we consider that

$$(\Lambda\,(\mathit{insert}\,x))\,\mathit{xs} \quad = \quad \{\mathit{insert}\,x\,\mathit{xs}\} \quad = \quad (\mathit{singleton} \circ \mathit{insert}\,x)\,\mathit{xs} \tag{23}$$

then $\Lambda\,\mathit{sort}$ can be written as follows.

$$\Lambda\,\mathit{sort} \quad \sqsubseteq \quad \mathit{foldr}\,\Big(\lambda\ x \bullet \bigcup \circ\, \mathit{mapSet}\,(\mathit{singleton} \circ \mathit{insert}\,x)\Big)\,\{[]\} \tag{24}$$

We have shown that we can phrase an initial specification in the singular and still construct a plural implementation.

5 Filtering Permutations

We can generalise from the specification of *sort* given in (4) to a more generic shape of specification. Consider specifying a function f via the conjunction of a function g that produces a list, and an extra requirement on the output list P. We can write this as

$$f \quad = \quad g \cap (\lambda\ x \bullet (\sqcap\, ys \mid P(ys))) \tag{25}$$

We can implement $\Lambda\,f$ as follows.

$$\Lambda\,f \quad \sqsubseteq \quad \mathit{filterSet}\ P \circ \Lambda\,g \tag{26}$$

That is, we can calculate all of the possible outputs of g, and then discard those outputs that don't satisfy P. Groves [6] has proved some similar, more general results to do with implementing conjunctions using sequential composition in the imperative refinement calculus.

We could have used this as a very naive, though very simple, approach for implementing *sort*.

$$\Lambda\,\mathit{sort} \quad \sqsubseteq \quad \mathit{filterSet}\ \mathit{isAscending} \circ \mathit{perms} \tag{27}$$

The best case complexity is quite good when combined with laziness!

Focusing on the body of the accumulation relation in (24) we can observe the following property.

$$\Lambda\,\mathit{insert}\,x \quad \sqsubseteq \quad \mathit{filterSet}\ \mathit{isAscending} \circ \Lambda\,\mathit{inject}\,x \tag{28}$$

By generalising *isAscending* to an arbitrary predicate P, we get

$$\frac{P([])}{\begin{array}{c}\mathit{perms} \cap (\lambda\ \mathit{xs} \bullet (\sqcap\, ys \mid P(ys))) \\ \sqsubseteq \\ \mathit{foldr}\,\Big(\lambda\ x \bullet \bigcup \circ\, \mathit{mapSet}\,(\mathit{filterSet}\ P \circ \Lambda\,\mathit{inject}\,x)\Big)\,\{[]\}\end{array}} \tag{29}$$

This is quite a useful general result, and is much more efficient than calculating all permutations and then filtering them.

6 Eight Queens

The eight queens problem is well known to most computer scientists. The goal is to place eight queens on a chess board so that no queen can attack another. Since a queen can move either horizontally or vertically, like a rook, or diagonally, like a bishop, we can use conjunction to specify the problem.

$$eightQueens \quad = \quad eightRooks \cap eightBishops \tag{30}$$

The eight rooks problem is easy to specify.

$$eightRooks \quad = \quad zip\,(perm\,[{'a'}..{'h'}])\,(perm\,[1..8]) \tag{31}$$

We take the eight distinct files and permute them, do the same for the ranks, and then construct pairs.

Since $perm \sqsubseteq id$ we can decide not to permute the files.

$$eightRooks \quad \sqsubseteq \quad zip\,[{'a'}..{'h'}]\,(perm\,[1..8]) \tag{32}$$

If we ignore the order of the pairs in the resulting list, this still leaves use with full coverage of the solution space. Refining out the *perm* of the ranks would leave us with a trivial solution where the rooks are arranged along the diagonal from $({'a'}, 1)$ to $({'h'}, 8)$.

The eight bishops problem doesn't appear to have as simple a solution as the eight rooks problem. A solution can be specified using a list of pairs of ranks and files, of length eight, where no two pairs occupy a common diagonal. We won't formalise this here, because we aren't confident that there is a useful solution involving such a formalisation. Another problem is that we haven't expressed the eight rooks problem in a form where *foldr* appears at the outermost level.

In conclusion, even though it is quite easy to express the eight queens problem using conjunction, there isn't necessarily a refinement sequence that maintains the conjunction for any length of time. However, it is still possible that there is such a solution.

7 Boggle

7.1 Introduction

Boggle is Parker Brothers trademark for its hidden word game. The object of the game is to find words in a four-by-four grid. The positions in the grid are occupied by dice that have a letter of the alphabet on each of their six faces. Before each round the dice are shaken into the grid in a random arrangement. Words are formed by moving vertically, horizontally or diagonally between the topmost faces of the dice. Each die may only be used once in each word. The problem that we are interested in is finding all possible words in a particular grid, relative to some dictionary.

7.2 Grids and Paths

Before considering the letters associated with the dice, it is useful to consider the paths that can be constructed between them. We begin by constructing a four-by-four grid.

$$grid \quad = \quad [(x, y) \mid x \leftarrow [1..4], y \leftarrow [1..4]] \tag{33}$$

A very naive way of specifying a solution is to consider all possible permutations of the grid elements, and restrict ourselves to those that are properly connected. We can then calculate all possible subsequences of each of these complete paths. In addition, we choose to work in a singular setting, specifying a single path.

$$path \quad = \quad (subsequence \circ (perm \cap outputConnected))\ grid \tag{34}$$

We consider the specification of *subsequence* to be uninteresting, so we turn our attention to specifying connected paths through the grid. We could do this via a simple Boolean test but, for reasons that will become obvious, we choose to do this by calculating the length of a path by considering the distance between each pair of path elements.

$$\begin{array}{lcl} pathLength\ p & = & \Sigma\ i \mid 0 \leq i < length\ p - 1 \bullet \\ & & \quad max\ abs\ ((fst\ (p!!i) - fst\ (p!!(i+1)))) \\ & & \qquad abs\ ((snd\ (p!!i) - snd\ (p!!(i+1)))) \end{array} \tag{35}$$

Now we can specify a connected path as one whose *pathLength* is less than the number of elements in the path.

$$isConnected\ p \quad = \quad pathLength\ p < length\ p \tag{36}$$

$$outputConnected \quad = \quad (\lambda\ xs \bullet (\sqcap\ ys \mid isConnected\ ys)) \tag{37}$$

In fact, the *pathLength* will be exactly one less that the *length*, but the above, more general definition turns out to be more useful. Also, note that *isConnected* handles any list of pairs of numbers, and isn't restricted to work with *grid*.

If we choose to implement *outputConnected* using *foldr*, via (14), we encounter a fairly serious problem. Although we require the final result to be connected, we don't necessarily want each intermediate result of *foldr* to be connected. That is, an intermediate result may not be connected, but may become connected if new elements are 'injected' into useful places. This is akin to choosing a loop invariant that is too strong, so possibly useful intermediate results are likely to be rejected.

We can provide a weaker version of *isConnected* that does the same job.

$$maxLength \quad = \quad 16 \tag{38}$$

$$isPossiblyConnected\ p \quad = \quad pathLength\ p < maxLength \tag{39}$$

This condition only causes a partial path to be rejected if its length already exceeds the maximum allowable length. For complete paths (of length 16),

this is equivalent to *isConnected*. However, this condition rejects fewer of the shorter, intermediate results, making it less efficient (although more correct) than *isConnected*.

7.3 Incomplete Paths

There is a neater specification of paths that dispenses with the idea of calculating complete paths and then calculating subpaths. The subpaths can be directly specified via the following variation of *perm* (5).

$$path' = foldr\ maybeInclude\ [] \tag{40}$$

$$maybeInclude = \left(\lambda\ x\ xs \bullet \left(\sqcap ys \;\middle|\; \begin{array}{c} items\ ys = [\![x]\!] \uplus items\ xs \\ \vee \\ items\ ys = items\ xs \end{array}\right)\right) \tag{41}$$

That is, *maybeInclude* is like *include* (7), but doesn't necessarily include the new element. Refinements of *maybeInclude* follow a similar pattern to those in Sect. 3.1.

However, if this model is used, it is more difficult to specify the 'invariant'. We hope to give this more consideration in the future.

7.4 Mindboggling Conclusions?

We haven't yet done enough work on this example to decide whether we can take a neat specification of the problem involving conjunction, and perform a neat refinement that yields a neat result! However, the work that we've done so far does raise some interesting points.

8 Conclusions

There are examples where conjunction can be exploited to modularise the design of programs. This is particularly true when higher-order functions are used to build identical frameworks for the conjuncts, allowing a promotion-like mechanism to be used. The approach is unlikely to be suitable for implementing all programs that can be specified using conjunction and catamorphisms, but when it can be used it is quite elegant. In particular, our method of implementing an arbitrary restriction using *foldr* can introduce concerns about monotonicity; the problem is similar to choosing a loop invariant that is too strong. The approach is not universal, but it does show some promise.

Another conclusion that can be drawn from this work is that problems can often be solved in elegant ways by combining elegant techniques from a variety of elegant methodologies.

References

1. C. Aarts, R. C. Backhouse, P. Hoogendijk, E. Voermans, and J. van der Woude. A relational theory of datatypes. Working document., Dec. 1992.
2. R.-J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, Feb. 1981.
3. R. Bird and O. de Moor. The algebra of programming. In M. Broy, editor, *Deductive Program Design*, volume 152 of *NATO ASI Series*, pages 167–203, Berlin Heidelberg, 1996. Springer-Verlag. Proceedings of the NATO Advanced Study Institute on Deductive Program Design, held in Marktoberdorf, Germany, July 26 – August 7, 1994.
4. R. S. Bird. A calculus of functions for program derivation. In Turner [19], chapter 11.
5. A. Bunkenburg. *Expression Refinement*. PhD thesis, Department of Computing Science, University of Glasgow, 1996.
6. L. Groves. Adapting program derivations using program conjunction. In Grundy et al. [7], pages 145–164.
7. J. Grundy, M. Schwenke, and T. Vickers, editors. *International Refinement Workshop and Formal Methods Pacific 1998*, Discrete Mathematics and Theoretical Computer Science, Singapore, 1998. Springer-Verlag.
8. J. Hughes. Why functional programming matters. In Turner [19], chapter 2, pages 17–42.
9. B. P. Mahony. The least conjunctive refinement and promotion in the refinement calculus. Submitted.
10. B. P. Mahony. Expression refinement in higher order logic. In Grundy et al. [7], pages 230–249.
11. C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Formal Approaches to Computing and Information Technology. Springer-Verlag, London, 1994.
12. J. M. Morris. Programs from specifications. In E. W. Dijkstra, editor, *The Formal Development of Programs and Proofs*, Year of Programming, pages 81–115. Addison Wesley, 1990.
13. J. M. Morris. Non-deterministic expressions and predicate transformers. *Information Processing Letters*, 61(5):1997, Mar. 1997.
14. T. S. Norvell and E. C. R. Hehner. Logical specifications for functional programs. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 269–290. Springer-Verlag, 1993. Proceedings of the 2nd Workshop on the Mathematics of Program Construction held at Oxford, June/July 1992.
15. J. Peterson and K. Hammond (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.4). Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, Feb. 1997.
16. M. Schwenke and B. Mahony. The essence of expression refinement. In Grundy et al. [7], pages 324–333.
17. M. Schwenke and K. Robinson. Semantics of expression specifications. In *The 3rd Australasian Refinement Workshop*, Apr. 1994.
18. J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 1989.

19. D. A. Turner, editor. *Research Topics in Functional Programming*. Year of Programming. Addison-Wesley, 1990.
20. N. Ward. Adding specification constructors to the refinement calculus. In *The 2rd Australasian Refinement Workshop*, Sept. 1992. Rough draft.
21. N. Ward. *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, The Department of Computer Science, The University of Queensland, Feb. 1994. `ftp://ftp.cs.uq.oz.au/pub/Thesis/nigel_ward.ps.Z`.