The Essence of Expression Refinement

Martin Schwenke[1] and Brendan Mahony[2]

[1] Department of Computer Science
The Australian National University
Martin.Schwenke@cs.anu.edu.au
[2] Defence Science and Technology Organisation
Department of Defence, Australia[***]
Brendan.Mahony@dsto.defence.gov.au

Abstract. Expression refinement semantics are often complex. In particular, semantics for the relatively simple notion of function application are often disproportionately complex. Monads are frequently championed as the 'right' tool for structuring the semantics of function application. In this paper, we consider the interaction between monadic semantics and refinement semantics. We demonstrate that a monad extended with a refinement ordering becomes an elegant tool for structuring the semantics of expression refinement. We do not claim that using monads to structure the semantics of function application is a new idea, nor do we introduce novel semantic models for non-deterministic languages. We simply demonstrate that the structure provided by monads ensures a consistent semantic approach. We illustrate this further by adding a facility for output to one of our languages.

1 Introduction

Expression refinement allows specifications of values to be transformed into executable expressions written in a programming language. Recently Morris [9] has described *weakest precondition* semantics for nondeterministic expressions and an associated specification language. He defines a *refinement* ordering between elements of his language based on weakest precondition semantics. Morris also allows an assignment construct that manipulates *state*, forging a connection between expression refinement and the *imperative refinement calculus* of Back [1], Morgan [7] and Morris [8].

Earlier work by Schwenke and Robinson [11] defines similar semantics for a pure expression specification language. Their language provides *generalised expression specifications*, while Morris defines only binary choices (although he uses a generalised choice in an example). Recently, Mahony [6] has formalised, in Isabelle/HOL [10], a refinement calculus that involves generalised expression specifications and imperative constructs.

[***] This paper was written during the author's visit to the CSIRO Mathematical and Information Sciences Division as part of the DSTO/CSIRO Fellowship Programme.

Wadler [12] argues that monads are the correct class of models for interpreting expression languages. As an example, he adds nondeterminism to an interpreter for a functional language using a *monad of lists*. We demonstrate that the expression refinement languages described above can be given monadic semantics. In fact, for each of these languages we give semantics having the structure of a monad extended with a refinement ordering. We refer to such a structure as a refinement monad. The comprehension of this structure in expression refinement semantics both clarifies and allows a simpler presentation of the semantics of nondeterministic expression languages. As a demonstration of this, we use different monads to structure the semantics of a range of languages that allow different amounts and types of nondeterminism and state manipulation. We find that monads allow the semantics of function application to be characterised and compared in an elegant manner.

This paper brings together two algebraic concepts that have not been combined before (or at least not in this context). It does not attempt to solve all of the problems in the field of expression refinement, but merely highlights a conceptual tool, which may prove valuable in the solution of these problems. We freely admit that the semantic models presented in this paper are carefully chosen to demonstrate the power of refinement-monadic semantics. We do not claim that these models are more powerful or that they result in more useful expression refinement languages than those we consider 'disproportionately complex' (for example, the domain theoretic semantic models of Ward [13] or Bunkenburg [3]). We simply make the modest claim that our understanding of these semantic models has benefited from structuring them using refinement monads. We believe that the elegant and concise accounts of these models we are able to present in this paper are strong confirmation of the power of the refinement monad concept.

In general, our style of presentation is quite informal, with an emphasis on conceptual simplicity rather than formal detail. However, much of the work presented in this paper has been formalised in Isabelle/HOL [6].

2 Monads

A monad is a triple $(M, unit_M, bind_M)$, where M is a type constructor and $unit_M$ and $bind_M$ have the following types

$$unit_M \quad : \quad \alpha \to M\,\alpha \tag{1}$$
$$\_\ bind_M\ \_ \quad : \quad M\,\alpha \to (\alpha \to M\,\beta) \to M\,\beta \tag{2}$$

The $unit_M$ function allows a monadic value to be constructed from an ordinary value by lifting it into its monadic representation. The $bind_M$ function takes a monadic value and applies a function to it, yielding a monadic value. The function being applied is a *monad-valued function*, mapping an ordinary value of one type into a monadic value of another type. Such functions are quite simple because they do not need to cope with the structure of the monad on their input.

The unit_M and bind_M functions must satisfy the following laws:

$$(\mathit{unit}_M\, c)\ \mathit{bind}_M\ k \quad = \quad k\ c \tag{3}$$

$$m\ \mathit{bind}_M\ \mathit{unit}_M \quad = \quad m \tag{4}$$

$$(m\ \mathit{bind}_M\ k)\ \mathit{bind}_M\ h \quad = \quad m\ \mathit{bind}_M\ (\lambda\ a \bullet (k\ a)\ \mathit{bind}_M\ h) \tag{5}$$

This presentation is similar to that of Wadler [12]. He describes interpreters for functional languages with features such as error handling, state, output, nondeterminism and call-by-name. In each case, monads are used to define the language interpreter in a structured way, while maintaining referential transparency. Sect. 5 is based on Wadler's treatment of nondeterminism, but uses a more abstract monad.

3 Refinement Monads

In order to better structure the semantics of function application in expression refinement languages, we suggest the extension of the monad structure with a partial order

$$\_ \sqsubseteq \_ \quad : \quad M\,\alpha \leftrightarrow M\,\alpha \tag{6}$$

which determines when one expression is a 'suitable' replacement for another. By suitable, we generally mean either better defined or less nondeterministic. In this context, the partial order is referred to as the *refinement order.*

For the moment, we defer the question of how the refinement relation should interact with the other monadic constructs. Instead we turn our attention to the relationship between refinement and the monadic model of functions. Since, in a monadic semantics, functions are modelled as objects of type $\alpha \rightarrow M\,\beta$, it is natural to propose a model of refinement on functions that is simply the basic refinement relation applied pointwise.

$$f \sqsubseteq g \quad \equiv \quad (\forall\ x \bullet f\,x \sqsubseteq g\,x) \tag{7}$$

This is the last consideration we will give to the application of refinement related concepts to monadic function models. It is a straightforward exercise to apply each such concept to the function case through this technique of pointwise application.

Now we return to the question of the necessary interaction between refinement and bind_M. As with most refinement related operators we require simply that bind_M be monotonic in both it arguments. Thus we add two further axioms to the standard monadic axioms.

$$m \sqsubseteq m' \quad \Rightarrow \quad (\forall\ f \bullet m\ \mathit{bind}_M\ f \ \sqsubseteq\ m'\ \mathit{bind}_M\ f) \tag{8}$$

$$f \sqsubseteq f' \quad \Rightarrow \quad (\forall\ m \bullet m\ \mathit{bind}_M\ f \ \sqsubseteq\ m\ \mathit{bind}_M\ f') \tag{9}$$

We believe the refinement monad to be an ideal tool for comprehending and structuring the relationship between function application and refinement. In order to demonstrate its power, we now proceed to present refinement monad semantics for a series of expression refinement languages of increasing complexity.

4 An Ordinary Functional Language

The first language that we treat is a simple functional language. This language is deterministic and has no mutable state. The language contains the following interesting constructs.

$$\begin{array}{lcll} exp & \leftarrow & term & \text{(abstract terms)} \\ exp & \leftarrow & exp.exp & \text{(function application)} \\ exp & \leftarrow & (\lambda\ ident \bullet exp) & \text{(functions)} \end{array}$$

We use a metalanguage that involves logic, sets and λ-calculus to define a semantics for our language. In the concrete language we use an infix dot to indicate function application, while in the metalanguage we use juxtaposition. The syntactic class *term* consists of terms of the metalanguage that we allow in our concrete language for ease of presentation. Such terms might involve integers, metavariables, Booleans, arithmetic operators and logical operators. Note that we have omitted many common constructs such as alternation and lists because we wish to focus on function application.

We define semantics for our concrete language via a simple mapping into our metalanguage. The simple functional language, is conceptually a subset of the metalanguage, so the mapping is quite trivial. We complicate things a little by using the identity refinement monad *Id* in our semantics, but this serves to introduce the monadic approach for modelling function abstractions and application; providing the essential structure on which the semantics of every expression language should be based.

$$Id\,\alpha \quad = \quad \alpha \qquad (10)$$

$$unit_{Id}\,c \quad = \quad c \qquad (11)$$

$$e\ bind_{Id}\,f \quad = \quad f\,e \qquad (12)$$

$$e \sqsubseteq e' \quad \equiv \quad e = e' \qquad (13)$$

The identity monad is a bit of syntactic sugar for nothing, where $bind_{Id}$ performs function application. This is evident in the definition of refinement, which is simply equality of meanings.

The meaning of metalanguage terms and function application are defined using $unit_{Id}$ and $bind_{Id}$ respectively. The meaning of function abstractions is determined by interpreting their bodies.

$$[\![c]\!] \quad = \quad unit_{Id}\,c \qquad (14)$$

$$[\![f.e]\!] \quad = \quad [\![e]\!]\ bind_{Id}\ [\![f]\!] \qquad (15)$$

$$[\![\,(\lambda\ x \bullet e(x))]\!] \quad = \quad (\lambda\ x \bullet [\![e(x)]\!]) \qquad (16)$$

In the remaining sections we use more interesting refinement monads to capture the semantics of function application, but the only change to (14), (15) and (16) will be the choice of monad. That is, as we introduce languages with additional constructs and more complex semantics, we will not need to make radical changes to the form of the semantics of our core language constructs. The changes are confined to the definitions of $unit_M$ and $bind_M$.

5 An Expression Language with Nondeterminism

We wish to add *demonic choice* to our simple language. We define the following two syntactic elements.

$$\begin{array}{lcll} exp & \leftarrow & exp \sqcap exp & \text{(binary demonic choice)} \\ exp & \leftarrow & (\sqcap\, ident \mid term\,) & \text{(generalised demonic choice)} \end{array}$$

The binary demonic choice has two explicit elements available, and selects the less useful of the two. The generalised demonic choice selects the least desirable value, represented by an identifier *ident*, that satisfies a Boolean formula *term*. Such terms are not completely trivial because we allow them to refer to metalanguage variables, such as the bound variables in function abstractions, but they cannot contain choices.[1]

The semantics for function application in our concrete language are defined using the set refinement monad.

$$\begin{array}{rcll} Set\,\alpha & = & \mathbb{P}\,\alpha & (17) \\ unit_{Set}\; c & = & \{c\} & (18) \\ s\; bind_{Set}\, f & = & \{x \mid (\exists\; y \bullet y \in s \wedge x \in f\; y)\,\} & (19) \\ e \sqsubseteq e' & \equiv & e \supseteq e' & (20) \end{array}$$

Each nondeterministic expression is interpreted as the set of values it can evaluate to. The definition of $unit_M$ tells us that a metaterm is interpreted as the singleton set containing it, while that of $bind_{Set}$ takes a set and a set-valued function as arguments and returns a set-valued result. We define refinement by mapping concrete expressions into the metalanguage and comparing the resulting sets.

The meaning of metalanguage terms, function application and functions are the same as (14), (15) and (16), differing only in the choice of monad.

Binary and generalised choice are modelled by union and set comprehension respectively.

$$\begin{array}{rcll} [\![a \sqcap b]\!] & = & [\![a]\!] \cup [\![b]\!] & (21) \\ [\![\, (\sqcap\, x \mid P\,)]\!] & = & \{x \mid P\,\} & (22) \end{array}$$

The set monad semantics for our expression language and refinement is quite limited. There is only room for demonic choice; angelic choice and undefinedness can not be treated because the set monad has only a single dimension of choice.

6 A Slightly Different Construction

There is an alternative construction of the model given in Sect. 5. In the style of Back and von Wright [2], we can model sets using the *simple expression-predicate*

[1] Note that this paper does not to treat choices of functions, which can be handled via a pointwise lifting of choice to functions.

refinement monad. Simple expression-predicates make an assertion about a given value, but can't reference program variables.

$$\begin{aligned}
\mathit{PredSet}\,\alpha &= \alpha \rightarrow \mathbb{B} && (23)\\
\mathit{unit}_{\mathit{PredSet}}\ c &= (\lambda\ x \bullet x = c) && (24)\\
p\ \mathit{bind}_{\mathit{PredSet}}\ f &= (\lambda\ x \bullet (\exists\ y \bullet p\, y \wedge f\, y\, x)) && (25)\\
e \sqsubseteq e' &\equiv (\forall\ x \bullet e\, x \Leftarrow e'\, x) && (26)
\end{aligned}$$

Given a value v of type α, we can apply a value p of type $\mathit{PredSet}\,\alpha$ to v to determine if v is a member of p, thus function evaluation takes the place of set membership. Singleton sets constructed via $\mathit{unit}_{\mathit{PredSet}}$ merely check if a given value is equal to their sole element, while the definition of $\mathit{bind}_{\mathit{PredSet}}$ is similar to (19). Refinement on simple expression-predicates is defined in a pointwise manner. It is easy to show that $\mathit{Set}\,\alpha$ and $\mathit{PredSet}\,\alpha$ are isomorphic. We introduce the simple expression-predicate monad because it is easier to generalise it to allow treatment of angelic nondeterminism, undefinedness and state.

The definitions of the choice constructs are similar to those in Sect. 5.

$$\begin{aligned}
[\![a \sqcap b]\!] &= (\lambda\ x \bullet [\![a]\!]\, x \vee [\![b]\!]\, x) && (27)\\
[\![(\sqcap x \mid P)]\!] &= (\lambda\ x \bullet P) && (28)
\end{aligned}$$

7 Adding Angelic Nondeterminism

We extend the above language with binary angelic choice and generalised angelic choice. We also allow expressions to be augmented with assumptions.

exp	$\leftarrow$	$\mathit{exp} \sqcup \mathit{exp}$	(binary angelic choice)
exp	$\leftarrow$	$(\sqcup\, \mathit{ident} \mid \mathit{term})$	(generalised angelic choice)
exp	$\leftarrow$	$\mathit{term} \succ \mathit{exp}$	(assumption)

Angelic choice is the dual of demonic choice in that it always makes the most useful possible selection. The assumption *term* is a predicate written in the metalanguage.

In order to model expressions with angelic choice we introduce the simple expression predicate transformer refinement monad.

$$\begin{aligned}
\mathit{SPT}\,\alpha &= (\alpha \rightarrow \mathbb{B}) \rightarrow \mathbb{B} && (29)\\
\mathit{unit}_{\mathit{SPT}}\ c &= (\lambda\ \phi \bullet \phi\, c) && (30)\\
e\ \mathit{bind}_{\mathit{SPT}}\ f &= (\lambda\ \phi \bullet e\ (\lambda\ x \bullet (f\, x)\, \phi)) && (31)\\
e \sqsubseteq e' &\equiv (\forall\ \phi \bullet e\, \phi \Rightarrow e'\, \phi) && (32)
\end{aligned}$$

A simple expression-predicate transformer is essentially a set of sets. The outer set represents angelic choice and the inner sets represent demonic choice. Another view of the transformer model is that it is a function which tells us whether the result of evaluating an expression is certain to satisfy a given property; the property is represented by the simple expression-predicate argument to the transformer. Simple expression-predicates are referred to as *postfunctions* by Schwenke and Robinson [11], because of their role as arguments to simple expression-predicate transformers. The $unit_{SPT}$ function constructs the transformer which returns true for every postfunction satisfied by its argument. The definition of $bind_{SPT}$ must be considered carefully. Suppose that e is a transformer, that f is a transformer-valued function, and that ϕ is an expression-predicate postfunction. Suppose further that x is the result of evaluating e. The bind of f to e ensures ϕ if and only if e ensures that $f\ x$ ensures ϕ. Refinement is defined in the usual manner for predicate transformers. That is, a refinement must satisfy every property (postfunction) satisfied by the original expression. This definition of expression refinement is similar to that of Morris [9].

The definitions for the various choice constructs are unsurprising.

$$\llbracket a \sqcap b \rrbracket = (\lambda\ \phi \bullet \llbracket a \rrbracket\, \phi \wedge \llbracket b \rrbracket\, \phi) \tag{33}$$
$$\llbracket a \sqcup b \rrbracket = (\lambda\ \phi \bullet \llbracket a \rrbracket\, \phi \vee \llbracket b \rrbracket\, \phi) \tag{34}$$
$$\llbracket (\sqcap x \mid P) \rrbracket = (\lambda\ \phi \bullet (\forall\ x \mid P \bullet \phi\, x)) \tag{35}$$
$$\llbracket (\sqcup x \mid P) \rrbracket = (\lambda\ \phi \bullet (\exists\ x \mid P \bullet \phi\, x)) \tag{36}$$

An expression e augmented with an assumption P need only meet its requirement ϕ if the assumption is satisfied.

$$\llbracket P \succ\!\!- e \rrbracket = (\lambda\ \phi \bullet P \wedge \llbracket e \rrbracket\, \phi) \tag{37}$$

8 Stateful Predicate Transformer Semantics

We extend the language again, introducing the idea of state.

statement	$\leftarrow$	*term* := *exp*	(assignment)
exp	$\leftarrow$	*term* $\uparrow$	(state reference)

The *term* on the left-hand side of an assignment statement must be an *L-value*, mostly commonly a program variable. An L-value is a *term* of type

$$\alpha \rightarrow (\Sigma \rightarrow \Sigma)$$

where Σ represents an explicit state space. Given a value and a state, an L-value updates the state by storing the value in a given location. The *term* in a state reference expression must be an *R-value*. An R-value is a *term* of type $\Sigma \rightarrow \alpha$. Given a state, it returns the value stored in a given location.

The refinement monad of stateful expression predicate transformers is defined as follows.

$$PT\,\alpha = (\alpha \rightarrow \Sigma \rightarrow \mathbb{B}) \rightarrow (\Sigma \rightarrow \mathbb{B}) \tag{38}$$
$$unit_{PT}\, c = (\lambda\ \phi\sigma \bullet \phi\, c\, \sigma) \tag{39}$$
$$t\ bind_{PT}\, f = (\lambda\ \phi\sigma \bullet t\ (\lambda\ x \bullet f\, x\, \phi)\ \sigma) \tag{40}$$
$$e \sqsubseteq e' \equiv (\forall\ \phi\sigma \bullet e\, \phi\sigma \Rightarrow e'\, \phi\sigma) \tag{41}$$

Expression-predicate transformers are applied to expression predicates yielding state predicates. Expression predicates place an explicit restriction on a value in the context of the program state. The definitions of unit_{SP}, bind_{SP} and $\sqsubseteq$ have the same structure as their counterparts in Sect. 7, but now instead of the result being a simple Boolean term it is a state predicate. In order to distinguish them from the previous ones we include the state arguments, but the state can be abstracted out of these equations yielding exactly (30), (31) and (32). The definitions of the existing language constructs have the same form as (33–37), only the type of ϕ changes. They differ in the choice of monad and the addition of the state space.

The definition of the assignment statement is similar to that of Morris [9]. An assignment ensures a postcondition ϕ if and only if the expression ensures that updating the state with its value results in a state satisfying ϕ.

$$[\![lv := e]\!] \quad = \quad (\lambda\ \phi \bullet [\![e]\!]\ (\lambda\ y\,\sigma \bullet \phi\,(lv\,y\,\sigma))) \tag{42}$$

A state reference expression ensures a postfunction ϕ if and only if the value in the state location satisfies ϕ.

$$[\![rv \uparrow]\!] \quad = \quad (\lambda\ \phi\,\sigma \bullet \phi\,(rv\,\sigma)) \tag{43}$$

Interfaces between imperative constructs and expressions can be defined. For details see Mahony [6]. Since expression predicate transformer are compatible with ordinary predicate transformer semantics, the semantics given above for choice constructs and procedures (functions with side-effects) can also be applied to imperative statements.

9 A Language with Output and Demonic Nondeterminism

All of the above expression refinement semantics are already known in the literature, we have simply shown that they can all be structured according to the refinement monad approach. In order to show how easy it is to add novel language features when using monads in our semantics, we add an output expression to the language defined in Sect. 6. We are influenced by Wadler's addition of output to an ordinary functional language [12].

$$\mathit{exp} \quad \leftarrow \quad \textbf{out}\ \mathit{exp} \qquad\qquad \text{(output)}$$

The *Out* refinement monad models a set of strings and values.

$$\begin{aligned}
\mathit{Out}\,\alpha \quad &= \quad (\mathit{String} \times \alpha) \to \mathbb{B} && (44)\\
\mathit{unit}_{\mathit{Out}}\ c \quad &= \quad (\lambda\,(s,a) \bullet s = \texttt{""} \wedge a = c) && (45)\\
m\ \mathit{bind}_{\mathit{Out}}\ k \quad &= \quad \left(\lambda\,(s,a) \bullet \left(\exists\ r_1, r_2, b \bullet \begin{array}{c} s = r_1 +\!\!+\ r_2 \\ m\,(r_1, b) \wedge k\ b\,(r_2, a) \end{array}\right)\right) && (46)\\
e \sqsubseteq e' \quad &\equiv \quad (\forall\,(s,a) \bullet e\,(s,a) \Leftarrow e'\,(s,a)) && (47)
\end{aligned}$$

The type $Out\,\alpha$ is, of course, just the type $PredSet\,(String \times \alpha)$, so the definition of demonic choice generalises easily. The definition of $unit_{Out}$ is similar to (24), but explicitly indicates that metalanguage terms do not produce any output. The definition of $bind_{Out}$ is like (25), but concatenates the output strings resulting from evaluating the argument and the function body.

An output expression returns the result of evaluating its expression argument and concatenates a string representation of the value (generated by the function *showval*) to the output generated during the evaluation process. Care must be taken to cater for the fact that both value and string may be non-deterministic.

$$[\![\mathbf{out}\, e]\!] \quad = \quad (\lambda\,(s,a) \bullet (\exists\; r \bullet s = (showval\; a) +\!\!+ \star +\!\!+ r \wedge e\,(r,a))) \qquad (48)$$

The string $\star$ is a separator that is inserted between output items.

10 Discussion

We believe refinement monads to be the essence of structuring the semantics of function application in expression refinement languages. Forcing this structure into the definitions encourages uniformity between the different semantics for languages of varying complexity. It may be possible to elegantly combine the semantics of different language constructs using King and Wadler's work on combining monads [5]. It may also be possible to combine laws relating to monadic data structures with our monadic semantics, yielding higher-order refinement laws.

Adding nondeterministic choices to 'functional' languages has the potential to wreak havoc with referential transparency. One approach to this problem has been to use 'underdetermined' choice operators in order to avoid referential transparency problems, ensuring that 'the operator always makes the same choice' (Jones [4, p. 77-78]). Our use of $bind_M$ to model function application avoids any such confusion. Choices are sets and refinement is the act of making a choice. However, if an expression is refined in two different ways then the refinements are not necessarily equal.

We are also interested in exploring the possibility of treating predicate transformers at the level of the concrete language. The equivalence between $[\![e]\!]\,\phi$ and $[\![\,(\lambda\; x \bullet \phi\, x)\,.e]\!]$ can possibly be used to define a *wp* operator, allowing refinement to be treated entirely at the concrete level. This is intriguing.

Acknowledgements

We are grateful to Jim Grundy for valuable discussions regarding this paper. The three anonymous referees also provided helpful comments.

References

1. R.-J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, Feb. 1981.
2. R.-J. R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing*, 2(3):247–272, July 1990.
3. A. Bunkenburg. *Expression Refinement.* PhD thesis, Department of Computing Science, University of Glasgow, 1996.
4. C. B. Jones et al. *Mural: A formal development support system.* Springer-Verlag, London, 1991.
5. D. J. King and P. Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow, 1992*, Workshops in Computing, Ayr, Scotland, 1993. Springer-Verlag.
6. B. P. Mahony. Expression refinement in higher order logic. In 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.
7. C. Morgan and T. Vickers, editors. *On the Refinement Calculus.* Formal Approaches to Computing and Information Technology. Springer-Verlag, London, 1994.
8. 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.
9. J. M. Morris. Non-deterministic expressions and predicate transformers. *Information Processing Letters*, 61(5):1997, Mar. 1997.
10. L. C. Paulson. Isabelle's object-logics. Technical Report Technical Report 286, University of Cambridge, Computer Laboratory, 1993.
11. M. Schwenke and K. Robinson. What if? In *The 2rd Australasian Refinement Workshop*, Sept. 1992.
12. P. Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming Languages*, Albuquerque, Jan. 1992. ACM Press. Invited talk.
13. 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`.