Investigating Miraculous Specifications

Sharon Flynn
Information Technology Centre,
National University of Ireland, Galway
Galway, Ireland

Abstract

In order to use expressions as the basis of a specification language, we admit undefinedness, and introduce non-determinism through the use of a choice operator. We extend expressiveness of the language by allowing choice from a set of values. Such a set could be infinite, giving unbounded non-determinism, or it could be empty, producing miracles. In this paper we treat the miraculous specification, examining its uses and highlighting related problems. In particular, we find that miracles promote the possibility of specification in parts, and piecewise refinement. However, their undesirable properties mean that we must limit their use. A biased choice operator is introduced as a method of totalising miraculous expressions. Finally, the formation of miraculous functions is considered with reference to their use and manipulation.

1 Introduction

The subject matter for this paper has grown out of research carried out into the area of a refinement calculus for expressions [5]. The calculus could be used in a number of ways: to extend the imperative refinement calculus by allowing specification and refinement using more abstract expressions; to provide the basis for a calculus to allow the development of imperative programs from specification expressions; or to provide the basis of a framework for the formal development of functional programs from specifications. The calculus parallels the work of Back [1], Morris [9, 10] and Morgan [8] on the refinement calculus for imperative programs.

We take the view that a refinement calculus consists of a specification language, which contains a programming language as sublanguage, a refinement relation between specifications, and a set of laws to allow the formal development of programs from specifications.

In order to extend the expressiveness of a language of expressions, which may include integer, boolean, product and function expressions, as well as sets, bags and sequences, it is not uncommon to add a choice operator for non-determinism. Sondergaard and Sestoft, in their paper [15], examine various forms of non-determinism in functional languages. In [5] we admit both undefined expressions and non-deterministic expressions using a bottom (undefined) value and a choice operator respectively. Our choice operator is unusual, however, since as well as being erratic, it can be used to choose elements from infinite sets. Moreover, we decided not to restrict its use to non-empty sets. Choice from an empty set, known as a miracle, is what is described in this paper.

The remainder of the paper is set out as follows: section 2 describes in more detail what is meant by undefinedness, non-determinism and refinement, as applied to expressions. It also describes the specification expression, which allows us to choose a value from a set. Section 3 shows how miracles are formed, and how they can be used for partial specifications and piecewise refinement. Section 4 examines the problems associated with miracles, and how their occurrences can be controlled. Section 5 shows how partial functions are formed and explores their uses, including the structuring of specifications.

2 Background

The specification language of the refinement calculus is described in detail in [5] using a set of axioms. In this paper we give an informal description of some of the more unusual features of the language, omitting axioms, except those

necessary for the explanation of miraculous expressions. We begin with a treatment of undefined expressions, and then move onto non-deterministic expressions and specification expressions. Finally, we treat the refinement relation, and describe what it means for one expression to refine another.

2.1 Undefinedness

Undefined values necessarily occur in any mathematical language of expressions. Simple examples, with explanations, include

$4/0$	division by zero
$0/0 = 1$	division by zero
$\sqrt{-5}$	when complex numbers are not considered
$hd\langle\rangle$	trying to return the first element of the empty sequence

Although it is clear that such simple expressions do not result in a well-defined value, it is not so clear what should be the outcome of such expressions as

$$(\forall\, n : \mathbb{Z} \mid \bullet n = 0 \lor n/n = 1)$$

$$(\forall\, S : Seq\, T \mid \bullet S = \langle\rangle \lor S = hd\, S \frown tl\, S)$$

where, if the first disjunct is true, the second must be undefined. The first expression states the property that for any integer n, either n is zero, or $n/n = 1$. The second states a property of sequences, that either a sequence is empty, or it is composed of its head and its tail. Undefined expressions are unavoidable, the problem lies in how to handle them. This problem is well documented, with various suggestions for its treatment, in *e.g.* [4, 7].

We make the decision to handle undefinedness explicitly. In order to allow reasoning about such expressions, we augment each type T with a special value '$\bot_T$', usually pronounced "bottom", which represents the undefined value of type T. For example, we say that the result of the evaluation of the expression $4/0$ is $\bot_{\mathbb{Z}}$. We shall drop the subscript in '$\bot_T$' if the type T is clear from the context, or is irrelevant. The undefined expression $\bot_T$ will also be used to represent a "don't care" value, where the specifier doesn't care about the result. This is in keeping with the treatments of [12, 16].

We now need to consider how expressions behave when their constituents are possibly undefined. In most cases it is appropriate to enforce strictness, *i.e.* an operator will yield $\bot$ when applied to $\bot$. So, for example, the expression $(4/0 + 3)$ is undefined, as is the expression $(0/0 = 1)$.

However, we do want to have the ability to reason about undefined expressions. For example, it is desirable that the two quantified expressions above should hold. Enforcing strictness of the boolean operators would result in these being undefined. This leads us to new versions of the disjunction and conjunction operators which are symmetric and which satisfy the equivalences:

$$X \land False \equiv False$$

$$X \lor True \equiv True$$

for arbitrary (possibly undefined) logical expression X. It is possible to give formal rules to define these extended operators. We will also use other non-strict operators, including equivalence $\equiv$ and refinement $\sqsubseteq$.

One issue which arises when considering possibly undefined expressions is that of *monotonicity*. An operation *op* is monotonic with respect to an ordering $\sqsubseteq$ if, for any expressions E and F with $E \sqsubseteq F$, we have $E' \sqsubseteq F'$, where E' and F' are the results of applying *op* to E and F respectively. The new versions of conjunction and disjunction retain monotonicity (with respect to the definedness ordering) and are equivalent to their 2-valued counterparts when terms are well-defined. Other non-strict operators, including equivalence which is essential for reasoning within the language, are non-monotonic. This operator allows us to assert such equivalences as $(4/0 \equiv \bot_{\mathbb{Z}})$.

In order to distinguish undefined terms in specifications, a non-strict, non-monotonic operator δ is introduced. For any expression E of any type, $\delta\, E$ is *True* if E is well-defined, and *False* otherwise. Clearly $\neg\delta\ \bot_T$ holds for any type T. A selection of axioms concerning δ can be found in appendix A.

2.2 Non-Determinism

To allow greater flexibility and to increase abstractness in specifications, we introduce the possibility of non-determinism in expressions. In a non-deterministic expression, any one of a number of possible outcomes is acceptable. For example, a familiar non-deterministic specification is to search a sequence for the index of a particular value. If the value occurs more than once in the sequence, it doesn't matter whether the first, the last, or any other occurrence of that value is found.

We admit non-determinism by introducing the choice operator '$[]$'. For E and F expressions of the same type T, the expression $E [] F$, also of type T, denotes the non-deterministic choice between the two expressions. Evaluation of $E [] F$ could result in the evaluation of E or the evaluation of F, but we don't know or care which. Choice enjoys the properties of commutativity, associativity and idempotency.

Non-determinism is often modelled in terms of sets of possible outcomes. For example, the expression 3 has one possible outcome, namely the value 3. The expression $\sqrt{4}$, on the other hand, has two possible outcomes, the elements of the set $\{-2, 2\}$. The set of possible outcomes of an expression $E [] F$, then, contains the possible outcomes of expression E and the possible outcomes of expression F.

Facilitating non-determinism in the expression language is not a simple matter of just introducing the choice operator $[]$. We also need to consider how other operators of the language behave in the presence of non-deterministic operands. Most operators, such as integer addition, distribute over choice. So, for example, $(3 [] 4) + 7 \equiv 10 [] 11$. A few operators, such as equivalence, refinement and some of the boolean operators, do not distribute.

We must also consider the definedness properties of a possibly non-deterministic expression $E [] F$. In terms of sets of possible outcomes, the undefined integer $\bot_{\mathbb{Z}}$ has $\{\bot_{\mathbb{Z}}\}$ as its set of possible outcomes, while the expression $3 [] \bot_{\mathbb{Z}}$ is modelled by $\{3, \bot_{\mathbb{Z}}\}$. However, we say that both expressions are undefined. We make the decision that $\delta(E [] F)$ should hold only when both E and F are well-defined, $\delta E \wedge \delta F$. This means that $\neg\delta(E [] F)$ holds if either E or F has $\bot$ as a possible outcome. So, $\delta \bot$ is *False*, as is $\delta(3 [] \bot)$. In contrast, $\delta(3 [] 4)$ is *True*, as is $\delta 3$.

If an expression E yields a single, well-defined outcome, then we say that E is *proper* and we write ΔE. For example, $\Delta 3$ is *True*, while $\Delta \bot$, $\Delta(3 [] \bot)$ and $\Delta(3 [] 4)$ are all *False*. When all expressions are proper, the specification language reduces to the normal, everyday expressions involving familiar types such as integers, booleans, tuples, functions *etc.* Intuitively, it should be clear that if an expression is proper ΔE, then it is well-defined δE. Some axioms concerning Δ can be found in appendix A.

2.3 Specification Expressions

We introduce a new operation on sets called *generalised choice* and write this $[]/$. It is based on using the choice operator $[]$ with reduce for sets. If S is a non-empty, possibly infinite set of type $\mathbb{P} T$, then the expression $[]/S$ has type T and can be interpreted as 'choose any element of S'. For example

$$[]/\{3, 4, 5, 6\} \equiv 3 [] 4 [] 5 [] 6$$

Expressions of the form $[]/S$ are termed *specification expressions* [16]. Notice that we do not restrict specification expressions to choice over a finite set. Thus we may form the expression $[]/\mathbb{Z}$ which means 'choose any integer'.

The expressive power of the generalised choice operator is realised when it is used with set comprehensions. An initial specification can be given by defining the properties required of a solution using a predicate P say, forming the set of all elements which satisfy that property $\{x \in T : P\,x\}$, and then using $[]/$ to choose any one of those elements. Provided it can be proven that there is a solution, *i.e.* $(\exists\, x \in T \bullet P)$, then the set $\{x \in T : P\,x\}$ is non-empty, and the specification is given as

$$[]/\{x \in T : P\,x\}$$

which may, of course, be a non-deterministic expression. For example

$[]/\{x \in \mathbb{Z} : 0 \leqslant x : 2 * x\}$	Any even natural
$[]/\{s \in \mathbb{P}\mathbb{Z} : \#s = 10\}$	Any integer set with exactly 10 elements

2.4 Refinement

Informally, for an expression E to be refined by an expression F, written $E \sqsubseteq F$, it should be the case that every possible 'evaluation' of F is also a possible 'evaluation' of E, or is better defined than some possible 'evaluation' of E. So, a specification is refined by reducing non-determinism or by increasing definedness. For example, we expect the following refinements to hold

$$2 \parallel 3 \sqsubseteq 2 \tag{1}$$

$$[\!]/\mathbb{N} \sqsubseteq 2 \parallel 3$$

$$(\textbf{fun}\ x \in \mathbb{Z} : x + 2 \parallel x + 3) \sqsubseteq (\textbf{fun}\ x \in \mathbb{Z} : x + 2)$$

$$(\textbf{fun}\ x \in \mathbb{N} : x + 2 \parallel x + 3) \sqsubseteq (\textbf{fun}\ x \in \mathbb{Z} : x + 2) \tag{2}$$

The first two examples are simple cases of reducing non-determinacy, while the third example reduces non-determinacy within the body of a function. The last example reduces non-determinacy, but also increases definedness since the function on the left gives an undefined result for any negative integer, while that on the right is defined for every integer.

In terms of sets of possible evaluations, expression E is refined by expression F iff everything in the set of evaluations of F is better defined than something in the set of possible evaluations of E. This intuitive notion is exactly the Smyth ordering for sets, as described in [14]. The formal semantics of refinement, as described in [5], is based on this Smyth ordering.

We advocate the process of program development by stepwise refinement, starting with an initial specification S_0 and building a sequence of specifications $S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$ so that each S_i, for $1 \leqslant i \leqslant n$ is an acceptable replacement for S_{i-1}, and S_n is a program. Since the aim is to derive programs in steps, it is required that the refinement relation is transitive. Then, from a sequence of refinements of the form $S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$, we can conclude that S_n is a correct implementation of the initial specification S_0. In fact, refinement is a pre-order, since every specification refines itself. In general, a refinement relation need not be anti-symmetric. In fact our relation is not since, for example, we have the refinements

$$2 \parallel \bot \sqsubseteq \bot$$

$$\bot \sqsubseteq 2 \parallel \bot$$

In the first case, the refinement is obtained by reducing non-determinism, while in the second definedness is increased. However, the two expressions are not equivalent.

As well as refinements proceeding stepwise, it is also important that refinement can occur piecewise. This means that an expression may be refined by refining one, or more, subexpressions,

$$(E \sqsubseteq F) \Rightarrow (G[E/x] \sqsubseteq G[F/x])$$

This states exactly the property that G must be monotonic (with respect to refinement) at the position x where the refined subexpression occurs. Refinement can occur only in monotonic positions.

The refinement relation is defined in [5] using six axioms, which are listed in appendix A.

3 Introducing Miracles

In section 2.3 we introduced the specification expression, of the form $[\!]/S$ for S a non-empty, possibly infinite set. Miracles occur when we ask the question, what is meant by $[\!]/\{\}$?

We introduce an identity for choice, which we give the fictitious value $\top$, pronounced "top". So, we have that $\top \parallel E \equiv E$ for any expression E. From our observations about sets of possible values, and interpreting choice as the union of such sets, it must follow that $\top$ corresponds to the empty set of possible values.

Since choice now has an identity, it follows from the rules for reduce that $[\!]/\{\} \equiv \top$.

We notice that $\top$ is distinct from $\bot$, and so it must be well-defined $\delta\ \top$. But $\top$ is not a proper value, so we assert $\neg \Delta\ \top$.

Since the set of possible outcomes of the expression $\top$ is empty, and so a subset of every set, it follows that $\top$ refines every expression, *i.e.* $E \sqsubseteq \top$ for arbitrary expression E. This means that $\top$ cannot be implemented; if it could, the programmer would have a very simple job. This is why $\top$ is referred to as a *miracle*.

The miracle expression $\top$ on its own is not a useful specification, and so we want to avoid its use. For example, we do not want to form the specification $[]/S$ unless we can prove that S is non-empty. Such a proof is a burden on the specifier. Further, given an initial specification expression E, there is nothing to stop the developer from over-refining E, perhaps in a sequence of steps, to the miraculous specification, thereby resulting in something which is unimplementable.

So, in introducing $\top$, we have introduced a problem. However, $\top$ has its uses, which we explore next, and so we are reluctant to throw it away. Instead we will consider, in section 4, how to handle it safely, and without placing any additional burden on the specifier or implementor.

3.1 Potentially Partial Expressions

We define a *total* expression to be one which has a non-empty set of possible outcomes. Otherwise, if the expression has no possible outcomes, not even the undefined outcome, we say that it is *partial*. We say that an expression of the form $[]/S$ is *potentially partial*, since it may 'evaluate' to $\top$, in the case where S is empty.

We introduce the concept of a *guarded expression*. The intuitive meaning of an expression $P \to E$, where P is a boolean expression called the *guard*, is such that: if P is *True* then $P \to E \equiv E$; if P is *False* then $P \to E \equiv \top$; and otherwise $P \to E \equiv \bot$. The axioms are, with $E : T$,

$$
\begin{aligned}
&\mathit{True} \to E \equiv E \\
&\mathit{False} \to E \equiv \top \\
&\neg \Delta P \Rightarrow (P \to E \equiv \bot_T)
\end{aligned}
$$

These axioms have been formed to facilitate case-based reasoning. To prove something about an expression $P \to E$ it is convenient to consider three cases, $P \equiv \mathit{True}$, $P \equiv \mathit{False}$ and $\neg \Delta P$.

Since an expression of the form $P \to E$ may 'evaluate' to $\top$, guarded expressions are potentially partial.

Guarded expressions are used to form *alternation expressions*. An alternation expression is of the form $P_1 \to E_1 [] \ldots [] P_n \to E_n$. Any guard P_i which evaluates to *False* has the result that the guarded expression $P_i \to E_i$ effectively disappears from the alternation. If all the guards are proper, then the alternation is such that some expression E_j for which the corresponding guard P_j evaluates to *True* will be chosen and evaluated. For example, the alternation

$$x \geqslant 0 \to \text{`+'} \; [] \; x \leqslant 0 \to \text{`}-\text{'}$$

will evaluate to '+' if the integer x is positive, to '$-$' if x is negative, and to either '+' or '$-$' if x is 0. An alternation expression is potentially partial, since all guards may be *False*.

Alternation expressions allow the facility for specification in parts because each 'case' of a specification can be considered in isolation, giving a guarded expression, and then combined using choice to form an alternation. When a particular case does not apply, for example when $x \geqslant 0$ does not apply in the above expression, the partial expression $x \geqslant 0 \to$ '+' 'disappears' from the specification, *i.e.* '+' is no longer a possible evaluation. This is based on the fact that $\top$ is the identity for choice.

Moreover, since $[]$ is monotonic with respect to the refinement operator, each guarded expression can be refined in isolation, resulting in piecewise refinement for each 'case' of the specification.

Experience with the Z specification language has shown that it is a useful feature to allow a specification to be constructed in parts. Partial specifications mean that a single aspect of the problem can be focussed upon in isolation, and the complete specification obtained by assembling the parts. The beauty of guarded expressions is that they can be easily combined using choice. We return to this theme in section 5.1 when we consider partial functions.

We now examine how we can ensure that a complete specification is total.

4 Managing Miracles

Although the introduction of $\top$ brings great expressive power to the language and greatly facilitates the piecewise refinement of expressions, it is nonetheless a very dangerous expression, as explained in section 3.

The solution is to control occurrences of potentially partial expressions so that every specification of the language, whether an initial specification or one calculated by refinement from a previous specification, is total (though it may contain partial sub-expressions).

We first show how potentially partial expressions can be recognised syntactically, and then show how restrictions can be imposed to ensure that every specification is a total expression.

4.1 Recognising Potentially Partial Expressions

Potentially partial expressions can occur in exactly 2 possible ways:

- from a generalised choice, $[]/S$
- from a guarded expression, $P \to E$

In the first case, the expression $[]/S$ is partial when S is the empty set; in the second case, the expression $P \to E$ is partial when P is *False*. There are no other constructs where partiality might be created. All other language constructs are total. So, it is only in the cases of generalised choice and guarding where we need to be concerned about the possible introduction of the miraculous expression $\top$. Both of these cases are recognisable syntactically.

Potentially partial expressions are defined as the smallest subset of expressions satisfying

- Expressions of the form $[]/S$ are potentially partial.
- Expressions of the form $P \to E$ are potentially partial.
- If E is potentially partial then so is $E \,[]\, F$, for arbitrary F.

4.2 Restricting the Syntax

We don't want to eliminate potentially partial expressions completely. We've seen that guarded expressions are very useful when used with choice to form alternation expressions. Generalised choice expressions are also extremely useful specification tools. We do, however, intend to ensure that potentially partial expressions are never used directly with operators (other than choice), constructors or function application. None of these can create partiality, but they would propagate it.

What is required is a way of 'totalising' potentially partial expressions, *i.e.* transform them into total expressions, so that they can be used freely in specifications. We introduce a new operator, biased choice $\overleftarrow{[]}$, which always chooses its left operand if possible. Intuitively, $E \overleftarrow{[]} F$ is equivalent to E if E is total, otherwise $E \overleftarrow{[]} F$ is equivalent to F. Biased choice is associative and idempotent, but not commutative. It is strict in its left argument and distributes over choice to the right. We have the axioms

$$(E \equiv \top) \Rightarrow (E \overleftarrow{[]} F \equiv F)$$
$$(E \not\equiv \top) \Rightarrow (E \overleftarrow{[]} F \equiv E)$$

Most importantly, the expression $E \overleftarrow{[]} F$ is guaranteed to be total if F is. This means that given a potentially partial expression, such as $P \to E$, it can be 'totalised' by combining it with a total 'alternative' F, giving an expresson of the form $P \to E \overleftarrow{[]} F$.

We now give the extra restrictions placed on expressions of the specification language. The use of potentially partial expressions is such that they may only be:

- operands of $[\!]$ – thus forming a new potentially partial expression;
- the left operand of $\overleftarrow{[\!]}$ – thus forming a total expression;
- operands of $\equiv$, $\sqsubseteq$, Δ and δ – thus forming total expressions.

5 Partial Functions

Consider what happens when we allow abstraction over guarded expressions to form *partial functions*, of the form $(\mathbf{fun}\ x \in T : B \to E)$ where E is typically a large expression.

Note the distinction between a total function and a total expression. A function expression can be total, while still being a partial function, *i.e.* its body is potentially partial. Such functions are total expressions and, as such, there is no restriction on where they may occur, subject to typing conditions.

Now consider the application of a partial function to some argument for which a result has not been specified in the function body. According to the axioms of the language which govern function application, the result is the value $\top$. So, for example, the result of the application

$$(\mathbf{fun}\ x \in \mathbb{Z} : x \geqslant 0 \to \text{`}+\text{'})(-7)$$

is $\top$ and thus the expression is not total.

This example illustrates that, although in order to form the expression $(f\ e)$, representing the application of function f to argument e, both f and e must be total, it is possible that the new expression $(f\ e)$ is not total.

The result of allowing such applications is that a new form of potentially partial expression has been admitted, that of a function application. Rather than complicating specifications by requiring that all expressions of the form $(f\ e)$ are totalised, we instead insist that all functions occurring within an expression are total functions.

We now consider how partial functions can be used.

5.1 Structuring Large Specifications

During the construction of a specification we claim that it is useful to allow an abstraction over a non-total expression, *i.e.* the formation of a partial function, with the intention that it be combined with other, possibly partial, functions at a later stage. In the same way that partial expressions are used for small specifications, partial functions are a useful concept in the language because they permit large specifications to be constructed in parts, with separation of concerns a major issue.

The intention is that a specification is written describing a result in a certain, perhaps error-free, case, generally of the form $(\mathbf{fun}\ x \in T : G \to E)$ where E is typically a large expression. The "error" case is described separately, perhaps of the form $(\mathbf{fun}\ x \in T : \neg G \to F)$. These two partial functions should be combined to form a new specification given by $(\mathbf{fun}\ x \in T : G \to E\ [\!]\ \neg G \to F)$. For example, the searching function for sequences of type *SeqT* could be written as

$$(\mathbf{fun}\ S \in SeqT, x \in T : [\!]/\{i \in \{0..\#S-1\} : S[i] = x\}) \tag{3}$$

This is a partial function since it yields $\top$ if the given x does not occur in the sequence. It could be made into a total function by combining it, for example, with a function which returns a default error value if the given value x does not occur in the sequence.

The Z specification language [13] permits the construction of specifications by combining schemas, which can be compared to partial functions. In a Z specification it is usual to combine schemas for partial specifications using schema disjunction. We propose a similar method for combining partial functions.

5.1.1 Combining Partial Functions

Our aim is to define an operator $\dot{\cup}$ which will take two partial functions and combine them such that

$$(\mathbf{fun}\ x \in T : E)\ \dot{\cup}\ (\mathbf{fun}\ x \in T : F) \quad \equiv \quad (\mathbf{fun}\ x \in T : E \,[\!]\, F)$$

Since the formation and combination of partial functions appears to be a purely syntactic notion, it makes sense that the definition of $\dot{\cup}$ should also be syntactic. Restrictions to occurrences of $\dot{\cup}$ are that it is used only with function types. The functions must be of the form $(\mathbf{fun}\ x \in T : E)$, or a choice between functions of this form. The two defining rules for $\dot{\cup}$ are, therefore

$$\begin{array}{rcl}(\mathbf{fun}\ x \in T : E)\ \dot{\cup}\ (\mathbf{fun}\ x \in T : F) & \hat{=} & (\mathbf{fun}\ x \in T : E \,[\!]\, F) \\ f\ \dot{\cup}\ (g_1 \,[\!]\, g_2) & \hat{=} & (f\ \dot{\cup}\ g_1) \,[\!]\, (f\ \dot{\cup}\ g_2)\end{array}$$

Since choice is commutative, so also is $\dot{\cup}$.

Taking the union of two partial functions yields another partial function. We define another version of union, a biased union, which can be used to obtain a total function. A function $(f \overset{\leftarrow}{\cup} g)$ when applied to an argument e will result in $(f\ e)$ if it is total and otherwise $(g\ e)$. The definition is purely syntactic, with the defining rules given by

$$\begin{array}{rcl}(\mathbf{fun}\ x \in T : E) \overset{\leftarrow}{\cup} (\mathbf{fun}\ x \in T : F) & \hat{=} & (\mathbf{fun}\ x \in T : E \overset{\leftarrow}{[\!]}\ F) \\ f \overset{\leftarrow}{\cup} (g_1 \,[\!]\, g_2) & \hat{=} & (f \overset{\leftarrow}{\cup} g_1) \,[\!]\, (f \overset{\leftarrow}{\cup} g_2)\end{array}$$

Commutativity does not hold, in general, for $\overset{\leftarrow}{\cup}$. Moreover, $\overset{\leftarrow}{\cup}$ does not left-distribute over choice, which is why the left argument of $\overset{\leftarrow}{\cup}$ may not be a choice between functions. We see that a function $f \overset{\leftarrow}{\cup} g$ is guaranteed to be a total function if g is. Thus, the biased union can be used to form total functions.

5.2 Manipulating Partial Functions

We have suggested the use of partial functions as a means to construct a specification piecewise, so that the partial functions can be combined to form a complete specification. However, we may also want to manipulate partial functions. This means allowing certain higher-order functions to be applied to potentially partial functions.

In general, partial functions are not permitted as arguments to higher-order functions, for the reason that this might introduce partiality into a specification. For example, let *sqrt* be the partial function

$$(\mathbf{fun}\ x \in \mathbb{Z} : (x \geqslant 0) \to \sqrt{x})$$

Now what should be the result of mapping f over a set of integers which may contain negative elements? The answer is not clear.

However, we propose a class of higher-order functions which may be applied to partial functions, and for which the resulting application is guaranteed to be total. Consider a higher-order function which takes two arguments, a possibly partial function f of type $\mathbb{Z} \to \mathbb{Z}$ and a string (sequence of characters) s. The result is a total function of type $\mathbb{Z} \to (\mathbb{Z} \times \mathit{String})$ which behaves in the following way: when applied to an argument x, if $(f\ x)$ is total then it returns the pair consisting of the value $(f\ x)$ and the string 'ok', otherwise it returns the pair $(0, s)$. The specification can be expressed by

$$\begin{array}{rcl}\text{totalise} & \hat{=} & (\mathbf{fun}\ f \in \mathbb{Z} \to \mathbb{Z}, s \in \mathit{String} : \\ & & \quad (\mathbf{fun}\ x \in \mathbb{Z} : ((f \overset{\leftarrow}{\cup} \text{zero})x, (x \in \mathit{dom}\, f \to \text{'}\mathit{ok}\text{'}) \overset{\leftarrow}{[\!]}\ s)))\end{array}$$

where zero $\hat{=}$ $(\mathbf{fun}\ x \in \mathbb{Z} : 0)$, and the function *dom*, when applied to a partial function f, returns the set of values for which f has been specified. Notice that the 'totalise' function, being a total function, can now be used to totalise a partial function. Without the possibility of having partial functions, we could not specify this higher-order function.

5.3 With Non-deterministic Arguments

We also examine the behaviour of partial functions when applied to non-deterministic arguments. For example, consider the following expression, which is not syntactically correct according to our syntax restrictions.

$$(\mathbf{fun}\, x \in \mathbb{Z} : x = 0 \to E)(0 \,[\!]\, 1) \tag{4}$$

Since function application distributes over choice, it is reasonable to assume that this should be the same as

$$(\mathbf{fun}\, x \in \mathbb{Z} : x = 0 \to E)0 \,[\!]\, (\mathbf{fun}\, x \in \mathbb{Z} : x = 0 \to E)1$$

Function application with deterministic arguments is governed by the substitution rule, giving

$$0 = 0 \to E \,[\!]\, 1 = 0 \to E$$

The first guard is *True*, and the second guard is *False*, so the expression is equivalent to just E.

We could say that the evaluation of expression (4) *looks ahead* to determine which choice, if any, gives a total result. Similarly, we expect the expression

$$(\mathbf{fun}\, x \in \mathbb{Z} : x = 0 \to E)([\!]/\mathbb{Z})$$

to behave in the same way, by *choosing* the integer value 0. This could prove to be a very useful property of the application of partial functions to non-deterministic arguments.

5.4 Specification Intersection?

We allow the formation of partial functions as abstractions over partial expressions, and combine them using a union operator, which is defined syntactically. This operator is similar to the disjunction operator used to combine schemas in Z. There also exists a conjunction operator for schemas in Z. We consider how a corresponding intersection operator might be used in our language.

In section 5 we used partial functions to specify different cases of a problem. These are then combined, using the union operator, such that

$$(\mathbf{fun}\, x \in T : P \to E) \,\dot{\cup}\, (\mathbf{fun}\, x \in T : \neg P \to F)$$

is equivalent to

$$(\mathbf{fun}\, x \in T : P \to E \,[\!]\, \neg P \to F)$$

Given the two specification expressions

$$[\!]/\{x \in \mathbb{Z} \;:\; 0 \leqslant x \leqslant 20\} \qquad [\!]/\{x \in \mathbb{Z} \;:\; \mathrm{even}\, x\}$$

an intersection of the two specifications should result in the expression

$$[\!]/\{x \in \mathbb{Z} \;:\; (0 \leqslant x \leqslant 20) \wedge \mathrm{even}\, x\}$$

Apart from investigating whether or not such a facility would be useful, it would also be interesting to see if a suitable syntactic definition could be given in the language. Such an operator, among others, is described by Frappier in [6] using a relational approach. The main concern is that either of the two specification expressions could be refined (piecewise) to such a point that the intersection no longer exists, resulting in an unimplementable specification.

6 Conclusions

The distinction between possibly undefined and possibly partial expressions is not usually so explicit. We have treated partiality as the dual of undefinedness, with respect to choice, *i.e.* $\top$ is the identity of choice, while $\bot$ is (almost) the zero for choice. The concept of partial expressions is useful since specifications can be built in parts, while each part may be manipulated and refined as a complete unit.

However, since partial expressions are not implementable, we found it necessary to control the occurrences of potentially partial expressions in specifications. This led to the introduction of the biased choice operator $\overleftarrow{[]}$ in order to totalise specifications. Unfortunately, $\overleftarrow{[]}$ is not monotonic with respect to refinement, in general, which causes some problems. It would be more elegant to treat partiality in the same unrestricted way that we have treated undefinedness.

We extended the concept of partial expressions to partial functions, which can be used purely as a syntactic device to structure specifications. The use of partial functions is demonstrated in [5] with a specification of a printing control system.

The specification language has been given a semantics based on sets, with powerdomain theory used to give a meaning to recursive functions.

6.1 Some Comparisons

There are many possible alternatives to the treatment of undefined expressions, as illustrated by the work of Cliff Jones in the area of handling partial functions [4, 7]. For example, the approach taken in the Z specification language [13] is to avoid function application entirely by treating functions as relations. This has the advantage that it would also handle non-determinism quite easily. The disadvantage is that this approach leads to more complicated formulations of properties, making specifications more difficult to write. We further note that the specification expression could be written in Z using the μ-expression, and relaxing the rules forcing deterministic results.

The approach we take to treat the undefined value explicitly, using an extension of classical logic, is similar to the approach used in the logic of partial functions (LPF) used for reasoning about specifications in VDM [2]. However, our implication $\Rightarrow$ is defined differently, and is reflexive. Further, while LPF is three-valued, our logic also deals with non-deterministic values.

Both Morris [11] and Bunkenburg [3] use a logic where terms may be non-deterministic. The choice operator in this treatment is demonic, making $\bot$ a zero for choice. The choice operator in our treatment is erratic.

Acknowledgements

I would like to thank two anonymous referees who pointed out numerous errors in an earlier version of this paper, and who provided interesting and helpful comments.

References

[1] R.J.R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593–624, 1988.

[2] H. Barringer, J.H. Chen, and C.B. Jones. A Logic Covering Undefinedness in Program Proofs. *Acta Informatica*, 21:251–269, 1984.

[3] A. Bunkenburg. *Expression Refinement: Imperative Programs.* Phd thesis, Department of Computing Science, University of Glasgow, 1997.

[4] J.H. Cheng and C.B. Jones. On the Usability of Logics which handle Partial Functions. In Carroll Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1990.

[5] S. Flynn. *A Refinement Calculus for Expressions.* Phd thesis, Department of Computing Science, University of Glasgow, 1997.

[6] M. Frappier, A. Mili, and J. Desharnais. Program Construction by Parts. In B. Möller, editor, *Mathematics of Program Construction : Proceedings of the Third International Conference, MPC'95, Kloster Irsee, Germany*, number 947 in LNCS, pages 258–281. Springer-Verlag, 1995.

[7] C.B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

[8] C. Morgan. *Programming from Specifications.* Prentice Hall, U.K., 1990.

[9] J.M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9:287–306, 1987.

[10] J.M. Morris. Programs from Specifications. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, University of Texas at Austin Year of Programming Series, chapter 9, pages 81–115. Addison-Wesley, 1989.

[11] J.M. Morris. Undefinedness and Nondeterminacy in Program Proofs. To appear, 1996.

[12] T.S. Norvell and E.C.R. Hehner. Logical Specifications for Functional Programs. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction 1992*, number 669 in LNCS, pages 269–290. Springer-Verlag, 1993.

[13] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, U.K., second edition, 1996.

[14] M. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.

[15] H. Sondergaard and P. Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35:514–523, 1992.

[16] N. Ward. *A Refinement Calculus for Nondeterministic Expressions.* Phd thesis, The University of Queensland, February 1994.

A A Selection of Axioms

Axioms for δ and Δ For v a value, x a variable identifier, T a type and E any expression:

$$\Delta\, v$$
$$\Delta\, x$$
$$\neg\delta\, \bot_T$$
$$\Delta\, E \Rightarrow \delta\, E$$
$$\Delta(\delta\, E)$$
$$\Delta(\Delta\, E)$$

These axioms state that: all proper values and all variable expressions are proper (and hence well-defined); for every type T, $\bot_T$ is not defined; every expression that is proper is necessarily well-defined; and it is always determined whether an expression is proper or well-defined.

Axioms for $[\!]$ For E, F and G expressions of the same type:

$$E \,[\!]\, E \equiv E$$
$$E \,[\!]\, F \equiv F \,[\!]\, E$$
$$E \,[\!]\, (F \,[\!]\, G) \equiv (E \,[\!]\, F) \,[\!]\, G$$
$$\Delta(E \,[\!]\, F) \equiv \Delta\, E \wedge \Delta\, F \wedge (E \equiv F)$$
$$\delta(E \,[\!]\, F) \equiv \delta\, E \wedge \delta\, F$$

These axioms state that: choice is idempotent, symmetric and associative; the expression $E \,[\!]\, F$ is proper whenever E and F are proper and equivalent expressions; the expression $E \,[\!]\, F$ is well-defined exactly when both E and F are well-defined.

Axioms for Operators over Integers For improper terms, all of the operators over integers are strict and distribute over choice.

For $\oplus$ one of $+, -, *, /, mod, \sqcap, \sqcup, <$, with E, F and G integer expressions:

$$E \oplus (F \,[\!]\, G) \equiv (E \oplus F) \,[\!]\, (E \oplus G)$$
$$(E \,[\!]\, F) \oplus G \equiv (E \oplus G) \,[\!]\, (F \oplus G)$$
$$\delta(E \oplus F) \Rightarrow (\delta E \wedge \delta F)$$

The last axiom is an equivalence when $\oplus$ is one of $+, -, *, \sqcap, \sqcup, <$.

Attempts to divide by zero result in undefined terms. For $\oslash$ one of $/, mod$, and with ΔF,

$$\delta(E \oslash F) \equiv \delta E \wedge \delta F \wedge (F \neq 0)$$

Refinement Axioms The refinement relation is transitive

$$(E \sqsubseteq F) \wedge (F \sqsubseteq G) \Rightarrow (E \sqsubseteq G)$$

for E, F and G expressions of the same type.

The general refinement axiom is

$$(E \sqsubseteq F) \Leftarrow (\neg\delta E \vee (E \,[\!]\, F \equiv E))$$

where E and F are expressions of the same type. When E and F belong to a simple type, this is an equivalence, and may be used as the definition of refinement.

For function domains, with f and g functions of the same type, and Δf and Δg,

$$(f \sqsubseteq g) \equiv (\forall x : T \mid \bullet f\, x \sqsubseteq g\, x)$$

When refining non-deterministic expressions, with E, F and G expressions of the same type, S a set expression of type $\mathbb{P}\, T$, and with ΔG, ΔE and ΔS we have the axioms

$$(E \,[\!]\, F \sqsubseteq G) \equiv (E \sqsubseteq G \vee F \sqsubseteq G)$$
$$([\!]/S \sqsubseteq E) \equiv (\exists x : T \mid x \in S \bullet x \sqsubseteq E)$$

We assert that top $\top$ is the unique most-refined specification,

$$(\top \sqsubseteq E) \equiv (E \equiv \top)$$

for E any expression.