How to calculate with nondeterministic functions

Richard Bird[1] and Florian Rabe[2]

[1] Department of Computer Science, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
[2] Laboratoire de Recherche en Informatique, University Paris Sud
Rue Noetzlin 91405 Orsay Cedex, France

Abstract. While simple equational reasoning is adequate for the calculation of many algorithms from their functional specifications, it is not up to the task of dealing with others, particularly those specified as optimisation problems. One approach is to replace functions by relations, and equational reasoning by reasoning about relational inclusion. But such a wholesale approach means one has to adopt a new and sometimes subtle language to argue about the properties of relational expressions. A more modest proposal is to generalise our powers of specification by allowing certain nondeterministic, or multi-valued functions, and to reason about refinement instead. Such functions will not appear in any final code. Refinement calculi have been studied extensively over the years and our aim in this article is just to explore the issues in a simple setting and to justify the axioms of refinement using the semantics suggested by Morris and Bunkenburg.

1 Introduction

We set the scene by considering the following Haskell definition for an archetypal optimisation problem:

$$\begin{aligned} &mcc :: [Item] \to Candidate \\ &mcc = minWith\ cost \cdot candidates \end{aligned}$$

The function *mcc* computes a candidate with minimum cost. The function *minWith* can be defined by

$$\begin{aligned} &minWith :: Ord\ b \Rightarrow (a \to b) \to [a] \to a \\ &minWith\ f = foldr1\ smaller \\ &\qquad\qquad \textbf{where}\ smaller\ x\ y = \textbf{if}\ f\ x \leqslant f\ y\ \textbf{then}\ x\ \textbf{else}\ y \end{aligned}$$

Applied to a finite, nonempty list of candidates, *minWith cost* returns the first candidate with minimum cost. The function *candidates* takes a finite list of items and returns a finite, nonempty list of candidates. We will suppose that the construction uses *foldr*:

$$\begin{aligned} &candidates :: [Item] \to [Candidate] \\ &candidates\ xs = foldr\ step\ [c_0]\ xs \\ &\qquad\qquad \textbf{where}\ step\ x\ cs = concatMap\ (additions\ x)\ cs \end{aligned}$$

The value c_0 is some default candidate for an empty list of items. The function *concatMap* is defined by

$$concatMap\ f = concat \cdot map\ f$$

and $additions :: Item \to Candidate \to [Candidate]$ takes a new item and a candidate and constructs a nonempty list of extended candidates. For example, if the candidates were the permutations of a list, then c_0 would be the empty list and *additions* x would be a list of all the ways x can be inserted into a given permutation. For example,

$$additions\ 1\ [2,4,3] = [[1,2,4,3],[2,1,4,3],[2,4,1,3],[2,4,3,1]]$$

A greedy algorithm for *mcc* arises as the result of successfully fusing the function *minWith cost* with *candidates*. Operationally speaking, instead of building the complete list of candidates and then selecting a best one, we construct a single best candidate at each step. The usual formulation of the fusion rule for *foldr* states that

$$foldr\ f\ (h\ e)\ xs = h\ (foldr\ g\ e\ xs)$$

for all finite lists xs provided the fusion condition

$$h\ (g\ x\ y) = f\ x\ (h\ y)$$

holds for all x and y. In fact the fusion condition is required to hold only for all y of the form $y = foldr\ g\ e\ xs$; this version is called *context-sensitive* fusion.

For our problem, $h = minWith\ cost$ and $g = step$ but f is unknown. Abbreviating *candidates* xs to cs, the context-sensitive fusion condition reads

$$minWith\ cost\ (step\ x\ cs) = add\ x\ (minWith\ cost\ cs)$$

for some function *add*. To see if it holds, and to discover *add* in the process, we can reason:

$$\begin{array}{ll} & minWith\ cost\ (step\ x\ cs) \\ = & \quad \{ \text{ definition of } step \text{ } \} \\ & minWith\ cost\ (concatMap\ (additions\ x)\ cs) \\ = & \quad \{ \text{ distributive law (see below) } \} \\ & minWith\ cost\ (map\ (minWith\ cost \cdot additions\ x)\ cs) \\ = & \quad \{ \text{ define } add\ x = minWith\ cost \cdot additions\ x \text{ } \} \\ & minWith\ cost\ (map\ (add\ x)\ cs) \\ = & \quad \{ \text{ greedy condition (see below) } \} \\ & add\ x\ (minWith\ cost\ cs) \end{array}$$

The distributive law used in the second step is the fact that

$$minWith\ f\ (concat\ xss) = minWith\ f\ (map\ (minWith\ f)\ xss)$$

provided xss is a finite list of finite, nonempty lists. Equivalently,

$$minWith\ f\ (concatMap\ g\ xs) = minWith\ f\ (map\ (minWith\ f \cdot g)\ xs)$$

provided xs is a finite list and g returns finite, nonempty lists. The proof of the distributivity law is straightforward but we omit details.

Summarising this short calculation, we have shown that

$$mcc = foldr\ add\ c_0\ \textbf{where}\ add\ x = minWith\ cost \cdot additions\ x$$

provided the following *greedy condition* holds for all x and xs:

$$minWith\ cost\ (map\ (add\ x)\ cs) = add\ x\ (minWith\ cost\ cs)$$

where $cs = candidates\ xs$.

That all seems simple enough. However, the fly in the ointment is that, in order to establish the greedy condition when there may be more than one candidate in cs with minimum cost, we need to prove the very strong fact that

$$cost\ c \leqslant cost\ c' \ \Leftrightarrow\ cost\ (add\ x\ c) \leqslant cost\ (add\ x\ c') \tag{1}$$

for all candidates c and c' in cs. To see why, observe that if c is the first candidate with minimum cost in a list of candidates, then $add\ x\ c$ has to be the first candidate with minimum cost in the list of extended candidates. This follows from our definition of $minWith$ which selects the first element with minimum cost in a list of candidates. To ensure that the extension of a candidate c' earlier in the list has a larger cost we have to show that

$$cost\ c' > cost\ c \ \Rightarrow\ cost\ (add\ x\ c') > cost\ (add\ x\ c) \tag{2}$$

for all c and c' in cs. To ensure that the extension of a candidate c' later in the list does not have a smaller cost we have to show that

$$cost\ c \leqslant cost\ c' \ \Rightarrow\ cost\ (add\ x\ c) \leqslant cost\ (add\ x\ c') \tag{3}$$

for all c and c' in cs. The conjunction of (2) and (3) is (1). The problem is that (1) is so strong that it rarely holds in practice. As evidence for this assertion, the appendix briefly discusses one example. A similar condition is needed if, say, $minWith$ returned the last element in a list with minimum cost, so the problem is not to do with the specific definition of $minWith$. What we really need is a form of reasoning that allows us to establish the necessary fusion condition from the simple monotonicity condition (3) alone, and the plain fact of the matter is that equational reasoning with any definition of $minWith$ is simply not adequate to provide it.

It follows that we have to abandon equational reasoning. One approach is to replace our functional framework with a relational one, and to reason instead about the inclusion of one relation in another. Such an approach has been suggested in a number of places, including our own [1]. But, for the purposes of

presenting a simple introduction to the subject of greedy algorithms in Haskell, this solution is way too drastic, more akin to a heart transplant than a tube of solvent for occasional use. The alternative, if it can be made to work smoothly, is to introduce nondeterministic functions, also called *multi-valued* functions in mathematics, and to reason about refinement.

The necessary intuitions and syntax are introduced in Section 2. Section 3 gives a formal calculus and Section 4 a denotational semantics for our language. The soundness of the semantics establishes the consistency of the calculus. We have formalised syntax, calculus, and semantics in the logical framework LF [2] and are in the process of also formalizing the soundness proof; the formalisation is not given in this paper but is available online[3].

2 Nondeterminism and refinement

Suppose we introduce *MinWith* as a nondeterministic function, specified only by the condition that if x is a possible value of *MinWith* f xs, where xs is a finite nonempty list, then x is an element of xs and for all elements y of xs we have $f\ x \leqslant f\ y$. Note the initial capital letter: *MinWith* is not part of Haskell. It is not our intention to extend Haskell with nondeterministic functions; instead nondeterminism is simply there to extend our powers of specification and cannot appear in any final algorithm.

Suppose we define $y \leftarrow F\ x$ to mean that y is one possible output of the nondeterministic function F applied to a value x. In words, y is a possible *refinement* of the nondeterministic expression $F\ x$. For example, $1 \leftarrow \mathit{MinWith}\ (\mathit{const}\ 0)\ [1,2]$ and $2 \leftarrow \mathit{MinWith}\ (\mathit{const}\ 0)\ [1,2]$. More generally, if E_1 and E_2 are possibly nondeterministic expressions of the same type T, we will write $E_1 \leftarrow E_2$ to mean that for all values v of T we have

$$v \leftarrow E_1 \quad \Rightarrow \quad v \leftarrow E_2$$

We define two nondeterministic expressions of the same type to be equal if they both have the same set of refinements: $E_1 = E_2$ if

$$v \leftarrow E_1 \quad \Leftrightarrow \quad v \leftarrow E_2$$

for all v. Equivalently,

$$E_1 = E_2 \quad \Leftrightarrow \quad E_1 \leftarrow E_2 \wedge E_2 \leftarrow E_1$$

which just says that $\leftarrow$ is anti-symmetric. Our task is to make precise the exact rules allowed for reasoning about $\leftarrow$ and to prove that these rules do not lead to contradictions.

To illustrate some of the pitfalls that have to be avoided, we consider three examples. First, here is the distributive law again in which *minWith* is replaced by *MinWith*:

[3] https://github.com/florian-rabe/nondet

$$MinWith\ f\ (concat\ xss) = MinWith\ f\ (map\ (MinWith\ f)\ xss)$$

If this equation is to hold for all finite, nonempty lists xss of finite, nonempty lists, and we do indeed want it to, then it has to mean there is no refinement of one side that is not also a refinement of the other side. It does *not* mean that the equation should hold for all possible implementations of $MinWith$, and it cannot mean that because it is false. Suppose we define $minWith$ to return the *second* best candidate in a list of candidates, or the only best candidate if there is only one. In particular,

$$\begin{array}{ll} minWith\ (const\ 0)\ (concat\ [[a],[b,c]]) & = b \\ minWith\ (const\ 0)\ (map\ (minWith\ (const\ 0))\ [[a],[b,c]]) & = c \end{array}$$

The results are different so the distributive law fails. What the distributive law has to mean is the conjunction of the following two assertions, in which M abbreviates $MinWith\ cost$:

$$\begin{array}{lcl} x \leftarrow M\ (concat\ xss) & \Rightarrow & (\exists xs : xs \leftarrow map\ M\ xss \wedge x \leftarrow M\ xs) \\ (xs \leftarrow map\ M\ xss \wedge x \leftarrow M\ xs) & \Rightarrow & x \leftarrow M\ (concat\ xss) \end{array}$$

It is easy enough to show that these two assertions do hold though we omit details.

For the remaining two examples, define

$$Choose\ x\ y = MinWith\ (const\ 0)\ [x, y]$$

so $x \leftarrow Choose\ x\ y$ and $y \leftarrow Choose\ x\ y$. Do we have

$$double\ (Choose\ 1\ 2) = Choose\ 1\ 2 + Choose\ 1\ 2$$

where $double\ x = x + x$? The answer is no, because

$$\begin{array}{ll} & x \leftarrow double\ (Choose\ 1\ 2) \\ \Leftrightarrow & \exists y : y \leftarrow Choose\ 1\ 2 \wedge x = double\ y \\ \Leftrightarrow & x = 2 \vee x = 4 \end{array}$$

while

$$\begin{array}{ll} & x \leftarrow Choose\ 1\ 2 + Choose\ 1\ 2 \\ \Leftrightarrow & \exists y, z : y \leftarrow Choose\ 1\ 2 \wedge z \leftarrow Choose\ 1\ 2 \wedge x = y + z \\ \Leftrightarrow & x = 2 \vee x = 3 \vee x == 4 \end{array}$$

We have only that $double\ (Choose\ x\ y) \leftarrow Choose\ x\ y + Choose\ x\ y$.

For the third example, it is easy enough to show, for all f_1, f_2 and x that

$$Choose\ (f_1\ x)\ (f_2\ x) = Choose\ f_1\ f_2\ x$$

but it would be wrong to conclude by η conversion that

$$\lambda x.\ Choose\ (f_1\ x)\ (f_2\ x) = Choose\ f_1\ f_2$$

We have

$$f \leftarrow \lambda x.\ Choose\ (f_1\ x)\ (f_2\ x) \Leftrightarrow \forall x : f\ x = f_1\ x \vee f\ x = f_2\ x$$

However,

$$f \leftarrow Choose\ f_1\ f_2 \Leftrightarrow (\forall x : f\ x = f_1\ x) \vee (\forall x : f\ x = f_2\ x)$$

The results are different. The η rule, namely $f = \lambda x.\ f\ x$, does not hold if f is a nondeterministic function such as $Choose\ f_1\ f_2$.

What else do we want? Certainly, we want a refinement version of the fusion law for *foldr*, namely that over finite lists we have

$$foldr\ f\ e'\ xs \leftarrow H\ (foldr\ g\ e\ xs)$$

for all finite lists xs provided that $e' \leftarrow H\ e$ and $f\ x\ (H\ y) \leftarrow H\ (g\ x\ y)$. Here is the proof of the fusion law. The base case is immediate and the induction step is as follows:

$$\begin{array}{ll} & foldr\ f\ e'\ (x : xs) \\ = & \quad \{ \text{ definition of } foldr \ \} \\ & f\ x\ (foldr\ f\ e'\ xs) \\ \leftarrow & \quad \{ \text{ induction, and monotonicity of refinement (see below) } \} \\ & f\ x\ (H\ (foldr\ g\ e\ xs)) \\ \leftarrow & \quad \{ \text{ fusion condition, and monotonicity of refinement } \} \\ & H\ (g\ x\ (foldr\ g\ e\ xs)) \\ = & \quad \{ \text{ definition of } foldr \ \} \\ & H\ (foldr\ g\ e\ (x : xs)) \end{array}$$

The appeal to the monotonicity of refinement is the assertion

$$E_1 \leftarrow E_2 \ \Rightarrow \ F\ E_1 \leftarrow F\ E_2$$

So this condition is also required to hold.

Let us see what else we might need by redoing the calculation of the greedy algorithm for *mcc*. This time we start with the specification

$$mcc \leftarrow MinWith\ cost \cdot candidates$$

For the fusion condition we reason:

$$\begin{array}{ll} & MinWith\ cost\ (step\ x\ cs) \\ = & \quad \{ \text{ definition of } step \ \} \\ & MinWith\ cost\ (concatMap\ (additions\ x)\ cs) \\ = & \quad \{ \text{ distributive law } \} \\ & MinWith\ cost\ (map\ (MinWith\ cost \cdot additions\ x)\ cs) \\ \rightarrow & \quad \{ \text{ suppose } add\ x \leftarrow MinWith\ cost \cdot additions\ x \ \} \end{array}$$

$$
\begin{array}{ll}
 & \mathit{MinWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{add}\ x)\ \mathit{cs}) \\
\rightarrow & \quad \{\ \text{greedy condition (see below)}\ \} \\
 & \mathit{add}\ x\ (\mathit{MinWith}\ \mathit{cost}\ \mathit{cs})
\end{array}
$$

We write $E_1 \rightarrow E_2$ as an alternative to $E_2 \leftarrow E_1$. The second step makes use of the distributive law, and the third step is an instance of the monotonicity of refinement.

Let us now revisit the greedy condition. This time we only have to show

$$\mathit{add}\ x\ (\mathit{MinWith}\ \mathit{cost}\ \mathit{cs}) \leftarrow \mathit{MinWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{add}\ x)\ \mathit{cs})$$

where $\mathit{add}\ x \leftarrow \mathit{MinWith}\ \mathit{cost} \cdot \mathit{additions}\ x$. Unlike the previous version, this claim follows from the monotonicity condition (3). To spell out the details, suppose c is a candidate in cs with minimum cost. We have only to show that

$$\mathit{add}\ x\ c \leftarrow \mathit{MinWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{add}\ x)\ \mathit{cs})$$

Equivalently, that

$$\mathit{cost}\ (\mathit{add}\ x\ c) \leqslant \mathit{cost}\ (\mathit{add}\ x\ c')$$

for all candidates c' on cs. But this follows from (3) and the fact that $\mathit{cost}\ c \leqslant \mathit{cost}\ c'$.

Summarising, we can now define $\mathit{mcc} = \mathit{foldr}\ \mathit{add}\ c_0$ provided (3) holds for a suitable refinement of add. Unlike the previous calculation, the new one is sufficient to deal with most examples of greedy algorithms, at least when candidate generation is expressed in terms of foldr.

We have concentrated on greedy algorithms and the function $\mathit{MinWith}$, but there is another nondeterministic function ThinBy, which is needed in the study of thinning algorithms. Not every optimisation problem can be solved by a greedy algorithm, and between the extremes of maintaining just one candidate at each step and maintaining all possible candidates, there is the option of keeping only a subset of candidates in play. That is where ThinBy comes in. It is a function with type

$$\mathit{ThinBy} :: (a \rightarrow a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a]$$

Thus $\mathit{ThinBy}\ (\ll)\ \mathit{xs}$ takes a comparison function $\ll$ and a list xs as arguments and returns a subsequence ys of xs such that for all x in xs there is a y in ys with $y \ll x$. The subsequence is not specified further, so ThinBy is nondeterministic. We mention ThinBy to show that there is more than one nondeterministic function of interest in the study of deriving algorithms from specifications.

The task now before us is to find a suitable axiomatisation for a theory of refinement and to give a model to show the soundness and consistency of the axioms. Essentially, this axiomatisation is the one proposed in [3,4] but simplified by leaving out some details inessential for our purposes.

3 An axiomatic basis

Rather than deal with specific nondeterministic functions such as *MinWith* and *ThinBy*, we can phrase the required rules in terms of a binary choice operator ($\sqcap$). Thus,

$$E_1 \sqcap E_2 = \mathit{MinWith}\ (\mathit{const}\ 0)\ [E_1, E_2]$$

We also have

$$\mathit{MinWith}\ f\ xs = \mathit{foldr1}\ (\sqcap)\ [x \mid x \leftarrow xs, \mathit{and}\ [f\ x \leqslant f\ y \mid y \leftarrow xs]]$$

so *MinWith* can be defined in terms of ($\sqcap$). Below we write $\sqcap/$ for *foldr1* ($\sqcap$). Thus $\sqcap/$ takes a finite, nonempty list of arguments and returns an arbitrary element of the list.

To formulate the axioms we need a language of types and expressions, and we choose the simply-typed lambda calculus. Types are given by the grammar

$$T ::= B \mid T \to T$$

B consists of the base types, such as *Int* and *Bool*. We could have included pair types explicitly, as is done in [3], but for present purposes it is simpler to omit them. Expressions are given by the grammar

$$E ::= C \mid V \mid \sqcap / [E_1, E_2, ..., E_n] \mid E\ E \mid \lambda V : T.\, E$$

where $n > 0$ and each of $E_1, E_2, ..., E_n$ are expressions of the same type. We omit the type of the bound variable in a λ-abstraction if it can be inferred, and we write $E_1 \sqcap E_2$ for $\sqcap/\ [E_1, E_2]$. Included in the constants C are constant functions such as the addition function $+$ on integers (written infix as usual) and integer literals $0, 1, -1, ...$ The typing rules are standard; in particular, $\sqcap/\ [E_1, E_2, ..., E_n]$, has type T if all E_i do.

Boolean formulas are formed using equality $E_1 = E_2$ and refinement $E_1 \leftarrow E_2$ of expressions as well as universal and existential quantification and the propositional connectives in the usual way. We use the same type of Booleans both for programs and for formulas about them, but only some Boolean expressions are practical in programs (e.g., propositional connectives and equality at base types). Additionally, in order to state the axioms, we need a predicate $\mathit{pure}(E)$ to distinguish a subclass of expressions, called *pure* expressions. The intention is to define a semantics in which a pure expression denotes a single value, except for lambda abstractions with impure bodies, which denote a set of functions. We add rules such that $\mathit{pure}(E)$ holds if E is

- a constant C applied to any number of pure arguments (including C itself if there are no arguments),
- a lambda abstraction (independent of whether its body is pure).

Like any predicate symbol, purity is closed under equality, i.e., if E_1 is pure and we can prove $E_1 = E_2$, then so is E_2. For example, 2 and $E_1 + E_2$ for pure E_1 and E_2 are pure because 2 and $+$ are constants. Also $\lambda y.\, 1 \sqcap y$ is pure because it is a lambda abstraction, and $(\lambda x.\, \lambda y.\, x \sqcap y)\ 1$ is pure because it is equal by β-reduction (see below) to the former. Furthermore, $2 \sqcap 2$ is pure because it is equal to 2 (using the axioms given below), but $(\lambda y.\, 1 \sqcap y)\ 2$ and $1 \sqcap 2$ are both impure. In what follows we use lowercase letters for pure expressions and uppercase letters for possibly impure expressions.

The reason for introducing pure expressions is in the statement of our first two axioms, the rules of β and η conversion. The β rule is that if e is a pure expression, then

$$(\lambda x.\, E)\ e = E\ (x := e) \tag{4}$$

where $E\ (x := e)$ denotes the expression E with all free occurrences of x replaced by e. Intuitively, the purity restriction to β-reduction makes sense because the bound variable of the lambda abstraction only ranges over values and therefore may only be substituted with pure expressions.

The η rule asserts that if f is a pure function, then

$$f = \lambda x.\, f\ x \tag{5}$$

The purity restriction to η-expansion makes sense because lambda-abstractions are always pure and thus can never equal an impure function.

Our notion of purity corresponds to the *proper* expressions of [3] except that we avoid the axiom that variables are pure. Our first draft used that axiom, but we were unable to formalise the calculus until we modified that aspect. The reason why the axiom is problematic is that it forces a distinction between meta-variables (which may be impure) and object variables (which must be pure). That precludes using higher-order abstract syntax when representing and reasoning about the language, e.g., in a logical framework like [2], and highly complicates the substitution properties of the language. However, just like in [3], our binders will range only over values, which our calculus captures by adding a purity assumption for the bound variable whenever traversing into the body of a binder. For example, the ξ rule for equality reasoning under a lambda becomes:

$$\frac{\mathit{pure}(x)\ \vdash\ E = F}{\vdash\ \lambda x.E = \lambda x.F}$$

As we will see below, without the above purity restrictions we could derive a contradiction with the remaining five axioms, which are as follows:

$$E_1 \leftarrow E_2 \Leftrightarrow \forall x : x \leftarrow E_1 \Rightarrow x \leftarrow E_2 \tag{6}$$

$$E_1 = E_2 \Leftrightarrow \forall x : x \leftarrow E_1 \Leftrightarrow x \leftarrow E_2 \tag{7}$$

$$x \leftarrow \sqcap/\ [E_1, E_2, ..., E_n] \Leftrightarrow x \leftarrow E_1\ \vee\ x \leftarrow E_2\ \vee\ ...\ \vee\ x \leftarrow E_n \tag{8}$$

$$x \leftarrow F\ E \Leftrightarrow \exists f, e : f \leftarrow F \wedge e \leftarrow E \wedge x \leftarrow f\ e \tag{9}$$

$$f \leftarrow \lambda x.\, E \Leftrightarrow \forall x : f\ x \leftarrow E \tag{10}$$

Recall that free lower case variables range over pure expressions only, i.e., the free variables x and f are assumed pure.

From (6) and (7) we obtain that $(\leftarrow)$ is reflexive, transitive and anti-symmetric. From (8) we obtain that $(\sqcap)$ is associative, commutative and idempotent. Axioms (8) and (9) are sufficient to establish

$$F\ (\sqcap/\ [E_1, E_2, ..., E_n]) = \sqcap/\ [F\ E_1, F\ E_2, ..., F\ E_n] \tag{11}$$

Here is the proof:

$$\begin{array}{ll} & x \leftarrow F\ (\sqcap/\ [E_1, E_2, ..., E_n]) \\ \Leftrightarrow & \quad \{\ (9)\ \} \\ & \exists f, e : f \leftarrow F \wedge e \leftarrow \sqcap\ /\ [E_1, E_2, ..., E_n] \wedge x \leftarrow f\ e \\ \Leftrightarrow & \quad \{\ (8)\ \} \\ & \exists i, f, e : f \leftarrow F \wedge e \leftarrow E_i \wedge x \leftarrow f\ e \\ \Leftrightarrow & \quad \{\ (9)\ \} \\ & \exists i : x \leftarrow F\ E_i \\ \Leftrightarrow & \quad \{\ (8)\ \} \\ & x \leftarrow \sqcap\ /\ [F\ E_1, F\ E_2, ..., F\ E_n] \end{array}$$

It follows from (11) and (4) that

$$(\lambda x.\ x + x)\ (1 \sqcap 2) = (\lambda x.\ x + x)\ 1 \sqcap (\lambda x.\ x + x)\ 2 = 2 \sqcap 4$$

If, however, (4) was allowed to hold for arbitrary expressions, then we would have

$$(\lambda x.\ x + x)\ (1 \sqcap 2) = (1 \sqcap 2) + (1 \sqcap 2) = 2 \sqcap 3 \sqcap 4$$

which is a contradiction.

We can also show, for example, that $\lambda x.\ x \sqcap 3$ and $id \sqcap const\ 3$ are different functions even though they are extensionally the same:

$$(\lambda x.\ x \sqcap 3)\ x = x \sqcap 3 = (id \sqcap const\ 3)\ x$$

Consider the function $h = \lambda f.\ f\ 1 + f\ 2$. We have by β reduction that

$$h\ (\lambda x.\ x \sqcap 3) = (\lambda x.\ x \sqcap 3)\ 1 + (\lambda x.\ x \sqcap 3)\ 2 = (1 \sqcap 3) + (2 \sqcap 3) = 3 \sqcap 4 \sqcap 5 \sqcap 6$$

while, on account of (11), we have

$$h\ (id \sqcap const\ 3) = h\ id \sqcap h\ (const\ 3) = (1 + 2) \sqcap (3 + 3) = 3 \sqcap 6$$

Thus two nondeterministic functions can be extensionally equal without being the same function. That explains the restriction of the η rule to pure functions. Finally, (9) gives us that

$$\begin{array}{l} G_1 \leftarrow G_2 \Rightarrow F \cdot G_1 \leftarrow F \cdot G_2 \\ F_1 \leftarrow F_2\ \Rightarrow F_1 \cdot G \leftarrow F_2 \cdot G \end{array}$$

where $(\cdot) = (\lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x))$.

To complete the presentation of the calculus, we need to give the rules for the logical operators used in the axioms. The rule for the propositional connectives are the standard ones and are omitted. But the rules for the quantifies are subtle because we have to ensure the quantifiers range over pure expressions only. In single-conclusion natural deduction style, these are

$$\frac{\mathit{pure}(x) \vdash F}{\vdash \forall x{:}F} \qquad \frac{\vdash \forall x{:}F \quad \vdash \mathit{pure}(e)}{\vdash F(x{:=}e)}$$

$$\frac{\vdash F(x{:=}e) \quad \vdash \mathit{pure}(e)}{\vdash \exists x{:}F} \qquad \frac{\vdash \exists x{:}F \quad \mathit{pure}(x),\, F \vdash G}{\vdash G}$$

Here $\mathit{pure}(e)$ is the purity predicate, whose axioms are described above.

4 A denotational semantics

To establish the consistency of the axiomatisation we give a denotational semantics for nondeterministic expressions. As the target language of our semantics, we use standard set theory, with the notations $A \to B$ and $\lambda x \in A.b$ for functions (with $\in A$ omitted if clear).

Overview The basic intuition of the interpretation function $[\![-]\!]$ is given in the following table where we write $\mathbb{P}^*\ A$ for the set of non-empty subsets of A:

Syntax	Semantics
type T	set $[\![T]\!]$
context declaring $x : T$	environment mapping $\rho : x \mapsto [\![T]\!]$
expression $E : T$	non-empty subset $[\![E]\!] \in \mathbb{P}^*[\![T]\!]$
refinement $E_1 \leftarrow E_2$	subset $[\![E_1]\!]_\rho \subseteq [\![E_2]\!]_\rho$
function type $S \to T$	set–valued functions $[\![S]\!] \to \mathbb{P}^*[\![T]\!]$
choice $E_1 \sqcap E_2$	union $[\![E_1]\!]_\rho \cup [\![E_2]\!]_\rho$
purity $\mathit{pure}(E)$ for $E : T$ $[\![E]\!]_\rho$ is generated by a single $v \in [\![T]\!]$	

Thus, types denotes sets, and non-deterministic expressions denote sets of values. Functions are set-valued, and choice is simply union.

Additionally, for each type T, we will define the operation

$$[\![T]\!] \ni v \mapsto v^{\leftarrow} \in \mathbb{P}^*[\![T]\!],$$

which embeds the single (deterministic) values into the power set. We call it *refinement closure* because $v^{\leftarrow}$ is the set of all values that we want to allow as a refinement of v. This allows defining the refinement ordering $\leqslant_T$ on $[\![T]\!]$ by $v \leqslant_T w$ iff $v^{\leftarrow} \subseteq w^{\leftarrow}$. While we will not need it in the sequel, it is helpful to define because for every expression $E : T$, the set $[\![E]\!]$ will be downward closed with respect to $\leqslant_T$. One could add an expression $\bot$ as a value with no refinements other than itself, which denotes the empty set. But doing so would mean that $\bot$ would be a refinement of every expression, which we choose not to have. That explains the restriction to non-empty sets in our semantics. Note that $\leqslant_T$ is not the same as the usual approximation ordering on Haskell expressions of a given type with $\bot$ as the least element.

Choice and Refinement We define

$$[\![\sqcap/[E_1, ..., E_n]]\!]_\rho = [\![E_1]\!]_\rho \cup \ldots \cup [\![E_n]\!]_\rho$$

This captures our intuition that a choice refines to any of its arguments, i.e., it denotes all values denoted by any argument. This is tied to the intuition that the refinement property corresponds to the subset condition on denotations. For example, $E_1 \leftarrow E_1 \sqcap E_2$ corresponds to $[\![E_1]\!]_\rho \subseteq [\![E_1 \sqcap E_2]\!]_\rho$.

Pure expressions $e : T$ cannot be properly refined. At base types, they are interpreted as singletons. For the general case, we have to relax this idea somewhat and require only $[\![e]\!]_\rho = v^\leftarrow$ for some $v \in [\![T]\!]$.

Variables As usual, expressions with free variables are interpreted relative to an environment ρ. Analogously to variables ranging over pure expressions, the environment maps every variable $x : T$ to a value $v \in [\![T]\!]$ (but not to a subset of $[\![T]\!]$ as one might expect). Consequently, the denotation of a variable is defined by applying the refinement closure

$$[\![x]\!]_\rho = \rho(x)^\leftarrow$$

Base Types and Constants The interpretation of base types is straightforward, and we define

$$\begin{aligned} [\![\, \mathit{Int} \,]\!] &= \mathbb{Z} \\ [\![\, \mathit{Bool} \,]\!] &= \mathbb{B} \end{aligned}$$

Moreover, we define $v^\leftarrow = \{v\}$ for $v \in [\![B]\!]$ for every base type B. In particular, we have $v \leqslant_B w$ iff $v = w$. In other words, the refinement ordering on base types is *flat*.

We would like to interpret all constants C in this straightforward way as well, but that is not as easy. In general, we assume that for every user-declared constant $C : T$, a denotation $\overline{C} \in [\![T]\!]$ is provided. Then we define

$$[\![C]\!]_\rho = \overline{C}^\leftarrow.$$

However, we cannot simply assume that $\overline{C}$ is the standard denotation that we would use to interpret a deterministic type theory. For example, for $+ : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$, we cannot define $\overline{+}$ as the usual addition $+_\mathbb{Z} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ because we need a value $\overline{+} : \mathbb{Z} \to \mathbb{P}^*(\mathbb{Z} \to \mathbb{P}^*\mathbb{Z})$.

For first-order constants, i.e., constants $C : B_1 \to \ldots \to B_n \to B$ where B and all B_i are base types (e.g., the constant $+$), we can still lift the standard interpretation relatively easily: If $f : [\![B_1]\!] \to \ldots \to [\![B_n]\!] \to [\![B]\!]$ is the intended interpretation for C, we define

$$\overline{C} : [\![B_1]\!] \to \mathbb{P}^*([\![B_2]\!] \to \ldots \to \mathbb{P}^*([\![B_n]\!] \to \mathbb{P}^*[\![B]\!]) \ldots)$$

by

$$\overline{C} = \lambda x_1.\{\lambda x_2. \ldots \{\lambda x_n.\{f\, x_1\, \ldots,\, x_n\}\} \ldots\}$$

Because all B_i are base types, this yields we have $[\![C]\!]_\rho = \overline{C}^{\leftarrow} = \{\overline{C}\}$. For $n = 0$, this includes constants $C : B$, e.g., $[\![1]\!]_\rho = \{1\}$ and accordingly for all integer literals.

But we cannot systematically lift standard interpretations of higher-order constants C accordingly. Instead, we must provide $\overline{C}$ individually for each higher-order constant. But for the purposes of program calculation, this is acceptable because we only have to do it once for the primitive constants of the language. In [3], this subtlety is handled by restricting attention to first-order constants.

Functions We define the interpretation of function types as follows:

$$[\![S \to T]\!] = [\![S]\!] \to \mathbb{P}^*[\![T]\!]$$

and for $f \in [\![S \to T]\!]$ we define

$$f^{\leftarrow} = \{g : [\![S \to T]\!] \mid g(v) \subseteq f(v) \text{ for all } v \in [\![S]\!]\}$$

Thus, the refinement ordering on functions acts point-wise: $g \leqslant_{S\to T} f$ iff $g(v) \subseteq f(v)$ for all $v \in [\![S]\!]$.

For example, there are nine functions of type $[\![\, \mathit{Bool} \to \mathit{Bool}\,]\!]$ with $\mathbb{B} = \{0, 1\}$ whose tables are as follows:

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
0	$\{0,1\}$	$\{0,1\}$	$\{0\}$	$\{1\}$	$\{0,1\}$	$\{0\}$	$\{0\}$	$\{1\}$	$\{1\}$
1	$\{0,1\}$	$\{0\}$	$\{0,1\}$	$\{0,1\}$	$\{1\}$	$\{0\}$	$\{1\}$	$\{0\}$	$\{1\}$

For example, $f_7 = \neg$ is the lifting of the usual negation function. The ordering $\leqslant_{\mathit{Bool}\to\mathit{Bool}}$ has top element f_0 and the four bottom elements f_5, f_6, f_7 and f_8.

Finally, the clauses for the denotation of λ and application terms are

$$[\![\lambda x : S.E]\!]_\rho = (\lambda v \in [\![S]\!].[\![E]\!]_{\rho(x:=v)})^{\leftarrow} \tag{12}$$

$$[\![F\, E]\!]_\rho = \bigcup\{f(e) \mid f \in [\![F]\!]_\rho,\, e \in [\![E]\!]_\rho\} \tag{13}$$

Here the notation $\rho(x := v)$ means the environment ρ extended with the binding of v to x. Because every expression in already interpreted as a set and function expressions must be interpreted as set-valued functions, a λ-abstraction can be interpreted essentially as the corresponding semantic function. We only need to apply the refinement closure. Equivalently, we could rewrite (12) using

$$(\lambda v \in [\![S]\!].[\![E]\!]_{\rho(x:=v)})^{\leftarrow} = \{f \mid f(v) \subseteq [\![E]\!]_{\rho(x:=v)} \text{ for all } v \in [\![S]\!]\}$$

The clause for application captures our intuition of monotonicity of refinement: $F\, E$ is interpreted by applying all possible denotations f of F to all possible denotations e of E; each such application returns a set, and we take the union of all these sets.

Formulas Because formulas are a special case of expressions, they are interpreted as non-empty subsets of $[\![Bool]\!] = \{0, 1\}$. We write $\top$ for the truth value $\{1\}$ denoting truth. The truth value $\{0, 1\}$ will never occur (unless the user wilfully interprets a constant in a way that returns it).

The denotation of all Boolean constants and expressions is as usual. The denotation of the quantifiers and the special predicates is defined by:

$$[\![E_1 \leftarrow E_2]\!]_\rho = \top \quad \text{iff} \quad [\![E_1]\!]_\rho \subseteq [\![E_2]\!]_\rho \tag{14}$$

$$[\![pure(E)]\!]_\rho = \top \quad \text{iff} \quad [\![E]\!]_\rho = v^{\leftarrow} \text{ for some } v \in [\![S]\!] \tag{15}$$

$$[\![\forall_S x : F]\!]_\rho = \top \quad \text{iff} \quad [\![F]\!]_{\rho(x:=v)} = \top \text{ for all } v \in [\![S]\!] \tag{16}$$

$$[\![\exists_S x : F]\!]_\rho = \top \quad \text{iff} \quad [\![F]\!]_{\rho(x:=v)} = \top \text{ for some } v \in [\![S]\!] \tag{17}$$

Note that the quantified variables seamlessly range only over values.

Soundness and Consistency We can now state the soundness of our calculus as follows:

Theorem 1 (Soundness). *If F is provable, then $[\![F]\!]_\rho = \top$ for every environment ρ for the free variables of F. In particular, if $E_1 \leftarrow E_2$ is provable, then $[\![E_1]\!]_\rho \subseteq [\![E_2]\!]_\rho$ for all environments ρ.*

Proof. As usual, the proof proceeds by induction on derivations.

In particular, we must justify the axioms (4) - (10). We concentrate on (4), which requires us to show

$$[\![(\lambda x : S.E)\, e]\!]_\rho = [\![E(x := e)]\!]_\rho$$

for all expressions E, all pure expressions e and all environments ρ. The proof divides into two cases according to the two axioms for purity: either e is an application of a constant to pure arguments, in which case $[\![e]\!]_\rho$ is a singleton set, or e is a lambda abstraction. For the former we will need the fact that if e is single-valued, then $[\![E(x := e)]\!]_\rho = [\![E]\!]_{\rho(x:=!{[\![e]\!]_\rho})}$ where $!\{v\} = v$. This *substitution lemma* can be proved by structural induction on E. That means we can argue:

$[\![(\lambda x : S.E)\, e]\!]_\rho$
$= \{13\}$
$\bigcup\{f(v) \mid f \in [\![\lambda x.E]\!]_\rho,\ v \in [\![e]\!]_\rho\}$
$= \{12\}$
$\bigcup\{f(v) \mid f(w) \subseteq [\![E]\!]_{\rho(x:=w)} \text{ for all } w \in [\![S]\!],\ v \in [\![e]\!]_\rho\}$
$= \{\text{subsumed sets can be removed from a union}\}$
$\bigcup\{f(v) \mid f(w) = [\![E]\!]_{\rho(x:=w)} \text{ for all } w \in [\![S]\!],\ v \in [\![e]\!]_\rho\}$
$= \{[\![e]\!]_\rho \subseteq [\![S]\!]\}$
$\bigcup\{[\![E]\!]_{\rho(x:=v)} \mid v \in [\![e]\!]_\rho\}$
$= \{e \text{ is single-valued}\}$
$[\![E]\!]_{\rho(x:=!{[\![e]\!]_\rho})}$
$= \{\text{substitution lemma}\}$
$[\![E(x := e)]\!]_\rho$

For the second case, where e is a lambda abstraction $\lambda y : T.\, F$, we need the fact that

$$[\![(\lambda x.E)\,(\lambda y.F)]\!]_\rho = [\![E]\!]_{\rho(x:=\lambda v.[\![F]\!]_{\rho(y:=v)})}$$

This fact can be established as a corollary to the *monotonicity lemma* which asserts $[\![E]\!]_{\rho(x:=f)} \subseteq [\![E]\!]_{\rho(x:=g)}$ whenever $f(v) \subseteq g(v)$ holds for all $v \in [\![S]\!]$. for all expressions E and environments ρ. The monotonicity lemma can be proved by structural induction on E. The corollary above is now proved by reasoning

$[\![(\lambda x.E)\,(\lambda y.F)]\!]_\rho$
$= \{13\}$
$\bigcup\{h(f) \mid h \in [\![\lambda x.E]\!]_\rho,\ f \in [\![\lambda y.F]\!]_\rho\}$
$= \{\text{as in previous calculation}\}$
$\bigcup\{[\![E]\!]_{\rho(x:=f)} \mid f \in [\![\lambda y.F]\!]_\rho\}$
$= \{12\}$
$\bigcup\{[\![E]\!]_{\rho(x:=f)} \mid f(v) \subseteq [\![F]\!]_{\rho(y:=v)} \text{ for all } v \in [\![T]\!]\}$
$= \{\subseteq\text{-direction: monotonicity lemma; } \supseteq\text{-direction: } X \subseteq \bigcup Y \text{ if } X \in Y\}$
$[\![E]\!]_{\rho(x:=\lambda v.[\![F]\!]_{\rho(y:=v)})}$

It remains to show that the latter is equal to $[\![E(x := \lambda y.F)]\!]_\rho$. Here we proceed by structural induction on E. We omit the details. The other axioms are proved by similar reasoning.

As a straightforward consequence of soundness, we have

Theorem 2 (Consistency). *Our calculus is consistent, i.e., we cannot derive a contradiction.*

Proof. If we could derive a contradiction, then soundness would yield a contradiction in set theory.

Technically, our calculus is only consistent under the assumption that set theory is consistent. We can strengthen that result by using a much weaker target language than set theory for our semantics. Indeed, standard higher-order logic (using an appropriate definition of power set) is sufficient.

5 Summary

The need for nondeterministic functions arose while the first author was preparing a text on an introduction to Algorithm Design using Haskell. The book, which is co-authored by Jeremy Gibbons, will be published by Cambridge University Press next year. Two of the six parts of the book are devoted to greedy algorithms and thinning algorithms. To make the material as accessible as possible, we wanted to stay close to Haskell and that meant we did not want to make the move from functions to relations, as proposed for instance in [1]. Instead, we made use of just two nondeterministic functions, *MinWith* and *ThinBy* (or three if you count *MaxWith*), and reasoned about refinement rather than equality when the need arose. The legitimacy of the calculus, as propounded above,

is not given in the book. The problems associated with reasoning about nondeterminism were discussed at the Glasgow meeting of WG2.1 in 2016, when the second author came on board. Our aim has been to write a short and hopefully sufficient introduction to the subject of nondeterminism for functional programmers rather than logicians. In this enterprise we made much use of the very readable papers by Joe Morris and Alexander Bunkenberg.

References

1. Richard S. Bird and Oege de Moor. *The Algebra of Programming.* Prentice-Hall International Series in Computer Science, Hemel Hempstead, UK (1997).
2. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
3. Joseph M. Morris and Alexander Bunkenburg. Specificational functions. *ACM Transactions on Programming Languages and Systems*, 21 (3) (1999) pp 677–701.
4. Joseph M. Morris and Alexander Bunkenburg. Partiality and Nondeterminacy in Program Proofs *Formal Aspects of Computing* 10 (1998) pp 76–96.
5. Joseph M. Morris and Malcolm Tyrrell. Dually nondeterministic functions. *ACM Transactions on Programming Languages and Systems*, 30 (6), Article 34 (2008).

Appendix

Here is the example, known as the *paragraph* problem. Consider the task of dividing a list of words into a list of lines so that each line is subject to a maximum line width of w. Each line is a list of words and its width is the sum of the length of the words plus the number of inter-word spaces. There is an obvious greedy algorithm for this problem, namely to add the next word to the current line if it will fit, otherwise to start a newline with the word. For what cost function does the greedy algorithm produce a division with minimum cost?

The obvious answer is that such a division has the minimum possible number of lines. So it has, but we cannot calculate this algorithm from a specification involving *minWith length*. To see why, consider a list of words whose lengths are $[3, 6, 1, 8, 1, 8]$ (the words are not important, only their lengths matter). Taking $w = 12$, there are four shortest possible layouts, of which two are

$$p_1 = [[3, 6, 1], [8], [1, 8]]$$
$$p_2 = [[3, 6], [1, 8, 1], [8]]$$

Let $add\ x\ p$ be the function that adds the next word x to the end of the last line if the result will still fit into a width of 12, or else begins a new line. In particular

$$q_1 = add\ 2\ p_1 = [[3, 6, 1], [8], [1, 8], [2]]$$
$$q_2 = add\ 2\ p_2 = [[3, 6], [1, 8, 1], [8, 2]]$$

We have

$$length\ p_1 \leqslant length\ p_2 \wedge length\ q_1 > length\ q_2$$

so the monotonicity condition fails. The situation can be redeemed by strengthening the cost function to read

$$cost\ p = (length\ p, width\ (last\ p))$$

In words one paragraph costs less than another if its length is shorter, or if the lengths are equal and the width of the last line is shorter. Minimising *cost* will also minimise *length*. This time we do have

$$cost\ p \leqslant cost\ p' \Rightarrow cost\ (add\ x\ p) \leqslant cost\ (add\ x\ p')$$

as can be checked by considering the various cases, so the monotonicity condition holds. However, we also have

$$cost\ (add\ 5\ p_1) = cost\ (add\ 5\ p_2) = (4, 5)$$

and $cost\ p_2 < cost\ p_1$, so the strong monotonicity condition (1) fails.