Science of Computer Programming 14 (1990) 25-42
North-Holland

A FUNCTIONAL THEORY OF EXCEPTIONS

Mike SPIVEY

Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, UK

Communicated by C.B. Jones
Received August 1988
Revised June 1989

Abstract. Exceptions are a feature often provided by programming languages to deal with computations which may fail. This paper argues that lazy functional programming not only makes a built-in exception mechanism unnecessary, but provides a powerful tool for developing and transforming programs that use exceptions. The basic idea is the simple one of augmenting each type with a distinguished error value; this idea is made practical for writing programs and reasoning about them through the use of higher-order functions. An advantage is that simple equational arguments can be used to reason about the programs.

Throughout the paper, the problem of simplifying algebraic expressions using rewriting rules is used as a source of motivation and examples.

1. Introduction

Exceptions are a programming language feature often advocated as a way of dealing with computations which may fail, such as in implementing backtracking search. For example, the LCF system [4, 7] for machine-assisted reasoning is based on the functional programming language ML, and uses the exception mechanism of ML as part of the implementation of proof tactics. Paulson [6] describes the implementation of rewriting in the Cambridge version of LCF, and the examples in this paper are inspired by his account.

Although they allow tactics to be written concisely, exceptions have the disadvantage that simple equational reasoning can no longer be used to derive programs, to prove their properties, or to improve them by transformation. Many programming languages, including ML, do not make it explicit in the type of a function whether the function can raise an exception, so making it difficult to determine where equational reasoning is applicable and where it is not.

The goal of this paper is to show how a functional programming language with lazy evaluation makes a built-in exception mechanism unnecessary. Under lazy evaluation, exceptions can be implemented as an abstract data type, rather than as a feature of the programming language. A small collection of higher-order combining forms allows the propagation of failure and the selection of alternatives to be

expressed. These functions obey algebraic laws which allow programs using them to be derived and transformed.

A standard technique in giving a denotational semantics to programming languages with exceptions is to adjoin a distinguishable error element to each semantic domain. The same technique is exploited in this paper, but the construction is used not *outside* the programming language—in building semantic domains—but *inside* it, in defining the result type of functions which may fail. This way of implementing exceptions preserves the simple semantics of a purely functional programming language, allowing programs to be derived by simple equational reasoning without explicit appeal to semantics. Lazy evaluation makes the implementation efficient by ensuring, for example, that a second alternative will not be evaluated before the first one has failed.

Wadler [10] has also argued that exceptions are rendered unnecessary by lazy evaluation; he replaces each function which may raise an exception with a function which returns a *list* of results, with the possibility that this "list of successes" may be empty in the case of failure. The model of exceptions used here is simpler, but many of the transformation laws continue to hold in Wadler's more elaborate model.

In the first part of this paper, a small collection of higher-order exception-handling functions is defined, and they are applied to the problem of simplifying algebraic expressions using rewriting rules. The process of rewriting is split into three parts, represented by three higher-order functions. These functions can be combined in various ways to give different rewriting strategies.

The second part of the paper develops a set of algebraic laws obeyed by the exception-handling functions. As an illustration of the power of these laws, one of the three rewriting functions defined earlier is transformed to a more efficient but less compact form. The style of derivation owes much to the work of Bird [1, 2], in which laws about data types are expressed as equations between functions.

The third part of the paper explores the algebra of partial functions implemented using the exception mechanism. Higher-level operations such as composition of partial functions can be implemented using the lower-level primitives introduced earlier and standard constructions from category theory.

2. Programming notation

The programs developed in this paper are written in a simple functional programming language with lazy evaluation and polymorphic typing, a variant of the notation used in the book [3]. A similar language has been implemented by Turner [9] under the name MIRANDA, a trademark of Research Software Limited. A program in this language is a collection of function definitions, each consisting of a declaration of the type of the function and a number of equations giving its values for various arguments. Unusual features of the notation used here and significant differences from the notation of [3] and [9] are noted below.

If f is a function of type $\alpha \to \beta$ and a is a list of type *list* α, then $f * a$ is a list of type *list* β, defined by

$$f * [x_0, x_1, \ldots, x_{n-1}] = [f x_1, f x_2, \ldots, f x_{n-1}].$$

In MIRANDA, $f * a$ is written *map f a*.

If $\oplus$ is an associative operator, $\oplus/$ is the reduction operator defined so that

$$\oplus/[x_0, x_1, \ldots, x_{n-1}] = x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}.$$

If $\oplus$ has an identity element e, it is taken as the definition of $\oplus/[\,]$, so that the following laws hold:

$$\oplus/[\,] = e, \tag{1}$$

$$\oplus/[x] = x, \tag{2}$$

$$\oplus/(a +\!\!+ b) = (\oplus/a) \oplus (\oplus/b). \tag{3}$$

The last of these laws generalizes to many lists as follows:

$$\oplus/+\!\!+/s = \oplus/(\oplus/) * s. \tag{4}$$

Strictly speaking, the notation $\oplus/$ is not allowed in MIRANDA, where it is necessary to make explicit the result returned for the empty list. In place of $\oplus/$, the expression *foldr* $(\oplus)$ e should appear, where *foldr* (for "fold from the right") is a function defined so that

$$foldr\ (\oplus)\ e\ [x_0, x_1, \ldots, x_{n-1}] = x_0 \oplus (x_1 \oplus (\cdots (x_{n-1} \oplus e) \cdots)).$$

This calculation is especially efficient when $\oplus$ sometimes ignores its right argument, as the disjunction operator $\vee$ does, because the rest of the list can then be discarded immediately: for example, the expression *foldr* $(\vee)$ *False a* is evaluated by scanning the list a only as far as the first occurrence of *True*; if one is found, the rest of a is discarded, and the expression is reduced to *True* without further calculation.

The notation of *sections* provides a way of applying a binary operator to only one of its arguments. If $\oplus$ is a binary operator, then $(x\oplus)$ is the function which maps y to $x \oplus y$, and $(\oplus y)$ is the function which maps x to $x \oplus y$. For example, $(+1)$ is the function which adds one to its argument, and $(x{:})$ is the function which adjoins x to the front of a list. The section notation can be used to state another law, which relates concatenation $+\!\!+/$ to the mapping operator $*$:

$$f * (+\!\!+/s) = +\!\!+/(f*) * s. \tag{5}$$

Here, $(f*) * s$ is the list of lists which results from applying f to the members of each member of the list of lists s. The parentheses around a section $(x\oplus)$ or $(y\oplus)$ are often omitted where this does not lead to ambiguity.

3. The data type *maybe*

Some programs can either terminate successfully and produce an output or terminate unsuccessfully and produce no output. For example, a program which

tries to simplify an algebraic expression using a rewriting rule may either succeed, producing a simplified expression as output, or fail because the rewriting rule cannot be applied to the expression. In ML, such programs are written as functions which may raise an exception, but in our simple programming language this possibility is not open to us. Instead, we shall use programs which return an object of the type *maybe* α, defined by:

$$maybe\ \alpha ::= Just\ \alpha \mid Nothing.$$

The idea is that the function should return *Just* x if it succeeds in producing the result x, and should return *Nothing* if it fails. This means that the possibility of a function's raising an exception is made explicit in its type. Here is a function which tries to subtract 3 from its argument, but fails if the result would be negative:

$$\begin{aligned} &less3 :: num \to maybe\ num \\ &less3\ x = Just\ (x-3), \quad \textbf{if } x \geqslant 3 \\ &\qquad = Nothing, \qquad \textbf{otherwise}. \end{aligned}$$

Failure may be detected by use of the infix operator ?, defined as follows:

$$\begin{aligned} &(?) :: maybe\ \alpha \to maybe\ \alpha \to maybe\ \alpha \\ &Nothing\ ?\ y = y \\ &(Just\ z)\ ?\ y = Just\ z. \end{aligned}$$

If x succeeds, the expression $x\ ?\ y$ returns the same result as x; otherwise it returns the same result as y. If both x and y fail, $x\ ?\ y$ fails as well. For example, the expression *less*3 x ? *Just* x always succeeds, returning three less than x if this is nonnegative, and x itself otherwise.

Under lazy evaluation, an expression of the form $x\ ?\ y$ does not call for evaluation of y unless x fails. It is the need to match x with the patterns *Nothing* and *Just* z which forces it to be evaluated; if it succeeds, it is returned without evaluation of y. This means we can write the expression *Cheap* ? *Expensive* and know that *Expensive* will not be evaluated unless *Cheap* fails.

Sometimes it is necessary to select from a list of expressions the first one which succeeds; this is achieved by the reduction ?/. If the list is empty, or all its elements come to *Nothing*, then the result will be *Nothing* too. Lazy evaluation makes the definition of ?/ using *foldr* particularly efficient, because elements of the list need be evaluated only until one of them succeeds.

It is worth noting that in a language with applicative-order evaluation and true exceptions, it is not possible to make the same separation between calculating a list of attempts and selecting the first one which succeeds. For example, it would not be possible to define a function g by the single equation,

$$g\ a = ?/[f\ x \mid x \leftarrow a],$$

because the subexpression $[f\ x \mid x \leftarrow a]$ would fail completely if $f\ x$ failed for any element x of the list a. In such a language, it would be necessary to make explicit

the recursion encapsulated in the reduction ?/:

$$g\,[\,] = \textbf{fail}$$
$$g(x:a) = f\,x\ \textbf{orelse}\ g\,a.$$

Here, **fail** is an expression which raises an exception when it is evaluated, and the expression E **orelse** E' is evaluated by evaluating E, and if it fails, evaluating E' instead.

An ordinary total function may be applied to an argument which may have failed using the operator •, which simply passes on the failure:

$$(\bullet) :: (\alpha \to \beta) \to \mathit{maybe}\ \alpha \to \mathit{maybe}\ \beta$$
$$f \bullet \mathit{Nothing} = \mathit{Nothing}$$
$$f \bullet (\mathit{Just}\ z) = \mathit{Just}\ (f\,z)$$

An example of the use of • is the function *sql*3, defined by

$$\mathit{sql}3 :: \mathit{num} \to \mathit{maybe}\ \mathit{num}$$
$$\mathit{sql}3\ x = \mathit{sqrt} \bullet (\mathit{less}3\ x).$$

In effect, the expression *sql*3 x calculates $\sqrt{x-3}$, provided this does not involve taking the square root of a negative number; otherwise, it returns *Nothing*.

The model for exceptions presented here is a simple one which allows an expression to succeed in only one way. Another model for exceptions makes the results of functions into *lists of successes* [10]: for example, a function which solves a puzzle might return a list containing all the solutions instead of just one solution. If there are no solutions, such a function would just return the empty list. Many of the functions which act on simple exceptions can be translated into this richer model: for example, the operator ? becomes the concatenation operator ++ and • becomes the mapping operator *. Many of the laws about exceptions correspond to familiar laws about lists under this translation.

4. Terms and substitutions

An application of these ideas is in simplifying algebraic expressions by rewriting. For present purposes, an expression or *term* is either a simple variable, which might be written x, y, z, etc., or it is a *compound term* $f(a_1, \ldots, a_n)$, obtained by applying a function symbol f to zero or more arguments $a_1, \ldots, a_n$ which are themselves terms. If the number of arguments n is zero, the term is simply a constant. Terms can be represented by members of the type *term*, defined by:

$$\mathit{term} ::= \mathit{Var}\ \mathit{var} \mid \mathit{Func}\ \mathit{func}\ (\mathit{list}\ \mathit{term}).$$

The types *var* and *func* represent variables and function symbols respectively; for simplicity, we may take them both to be the type *char* of characters.

An element of *term* is either of the form *Var v*, where *v* is of type *var*, or of the form *Func f a*, where *f* is of the type *func*, and *a* is a list of immediate subterms. For example, the term $f(f(x, y), h(a))$ is represented by the following element of the type *term*:

Func 'f' [*Func* 'f' [*Var* 'x', *Var* 'y'], *Func* 'h' [*Func* 'a' []]].

Note that the constant *a* has been represented by a function symbol with an empty list of arguments.

The function *subterms* returns a list of the subterms of a term *t*, paired with the paths by which they can be reached from the root of *t*. Paths are represented by list of numbers: the empty list means the root of *t* itself, whilst *i*:*k* means "take the *i*th immediate subterm (counting from zero), then follow the path *k*":

```
path == list num.
```

The list compiled by *subterms* is in lexicographic order of path, and omits the trivial subterms consisting of a single variable.

```
subterms :: term → list (path × term)
subterms (Var v) = []
subterms (Func f a) = [([], Func f a)] ++ list_subs a

list_subs :: list term → list (path × term)
list_subs a = [(i : k, u) | i ← [0..#a − 1];
                            (k, u) ← subterms (a ! i)]
```

The expression *list_subs a* returns the list of all the subterms of elements of the list of terms *a*, labelling each with the element of *a* it came from.

As an example, here is a list of paths in the term $f(f(x, y), h(a))$ together with the subterms at their ends:

[]	$f(f(x, y), h(a))$
[0]	$f(x, y)$
[1]	$h(a)$
[1, 0]	a.

If *t* is a term and *k* is a path inside it, *replace t k u* is the result of replacing the subterm of *t* at the end of path *k* with the term *u*;

```
replace :: term → path → term → term
replace t [] u = u
replace (Func f a) (i : k) u
        = Func f (update a i (replace (a ! i) k u)).
```

The function *update* is defined so that *update a i y* is a copy of the list a with the i'th element replaced by y:

$$update :: list\ \alpha \to num \to \alpha \to list\ \alpha$$
$$update\ (x:a)\ 0\ y = y:a$$
$$update\ (x:a)\ (i+1)\ y = x:(update\ a\ i\ y).$$

The definitions of both these functions can be cast in terms of composition rather than application: these alternative definitions are stated here as laws, because they will be useful in reasoning about programs later:

$$replace\ t\ [\,] = id \tag{6}$$
$$replace\ (Func\ f\ a)\ (i:k) = Func\ f \cdot update\ a\ i \cdot replace\ (a\ !\ i)\ k \tag{7}$$
$$update\ (x:a)\ 0 = (:a) \tag{8}$$
$$update\ (x:a)\ (i+1) = (x:) \cdot update\ a\ i. \tag{9}$$

Here, $(x:)$ is the function which maps a list a to $x:a$, and $(:a)$ is the function which maps an element x to $x:a$.

We shall not need to know much about substitutions, except that one term can be matched against another to give a matching substitution or failure, and a substitution can be applied to a term to give an instantiated term:

$$match :: term \to term \to maybe\ sub$$
$$subst :: term \to sub \to term.$$

For example, if t is the term $f(f(x, y), z)$ and u is the term $f(f(a, w), f(w, b))$, then *match t u* succeeds with the substitution

$$s = \{x \mapsto a,\ y \mapsto w,\ z \mapsto f(w, b)\}.$$

This substitution shows what to substitute for the variables of t to get a copy of u, and in fact $subst\ t\ s = u$. If v is the term $f(x, f(y, z))$, then *subst v s* is the term $f(a, f(w, f(w, b)))$: it is the result of rewriting u with the equation

$$f(f(x, y), z) = f(x, f(y, z)).$$

Substitutions might be represented by association lists of variables and terms, or even by functions from variables to terms (see [3]), but the details are unimportant here.

5. Rewriting

An *equation* like $f(f(x, y), z) = f(x, f(y, z))$ can be regarded as just a pair of terms:

$$equation == term \times term.$$

An equation asserts that its two sides are equal whatever the values of the variables. This justifies the basic step of rewriting: using an equation as a rewriting rule from left to right. We can express this idea as a strategy, or function from *term* to *term* which may fail:

$$strategy == term \to maybe\ term.$$

If t is an instance of l, the expression $rewrite\ (l, r)\ t$ returns the corresponding instance of r; otherwise it fails. Here is the definition of *rewrite*:

$$rewrite :: equation \to strategy$$
$$rewrite\ (l, r)\ t = subst\ r \bullet match\ l\ t.$$

The operator • has been used in this definition to avoid substituting into r if the match of l with t fails. To continue the example, if e is the equation $f(f(x, y), z) = f(x, f(y, z))$ and u is the term $f(f(a, w), f(w, b))$, then $rewrite\ e\ u$ would succeed with the result $f(a, f(w, f(w, b)))$.

Two rewriting strategies can be combined so that the second one is tried if the first one fails. This is achieved with the operator ?? on strategies:

$$(??) :: strategy \to strategy \to strategy$$
$$(rw1\ ??\ rw2)\ t = rw1\ t\ ?\ rw2\ t.$$

An identity element for ?? is the strategy *fail*, which always fails:

$$fail :: strategy$$
$$fail\ t = Nothing.$$

A list of strategies can be combined by using the first one which succeeds. This is the purpose of the function *many_rules*, defined by reducing the list of strategies with ??:

$$many\_rules :: list\ strategy \to strategy$$
$$many\_rules = ??/.$$

Of course, $many\_rules\ [\,] = fail$.

So far, we can rewrite a whole term with an equation, and we can try a list of rewriting strategies until one succeeds. A third element of rewriting is the idea of rewriting not just a term in its entirety, but also its subterms. If a subterm u of a term t can be rewritten, then we can rewrite t by replacing the subterm u by its rewritten form. This idea is expressed by the function *inside*, which takes a rewriting strategy and applies it to subterms of a term t, returning the result of replacing in t the first one that can be rewritten:

$$inside :: strategy \to strategy$$
$$inside\ rw\ t = ?/[replace\ t\ k \bullet rw\ u \mid (k, u) \leftarrow subterms\ t].$$

As an example of the use of *inside*, suppose e is the equation $a = b$, where a and b are constants: this equation would be represented by the pair

(*Func* '*a*' [ ], *Func* '*b*' [ ]). Let t be the term $f(x, g(f(a, y)))$. The expression

inside (*rewrite e*) *t*

calls for the equation e to be tried on each subterm of t in depth-first order. In fact, there is exactly one subterm where the left-hand side of e matches, the subterm a at the end of the path [1, 0, 0]. The expression succeeds with the term $f(x, g(f(b, y)))$ obtained by replacing the subterm of t at [1, 0, 0] with b.

The three parts *rewrite*, *many_rules* and *inside* can be put together in various ways to make different rewriting strategies. For example, here is a strategy which applies a list of equations to a term, first trying each rule at the root, then trying each one at the subterms:

reduce :: *list equation* → *strategy*
reduce = *inside* · *many_rules* · (*rewrite* ∗).

The function (*rewrite* ∗) makes the list of equations into a list of strategies; these are combined with *many_rules* to give a single strategy which tries the equations in term. Finally, *inside* takes this strategy and applies it at each subterm.

6. Laws of list comprehension

In this section and the next are collected a number of laws expressed as equations; this section contains general laws about list comprehension, and the next contains more specialized laws about the exception-handling operators defined in Section 3. The use of these laws in program transformation is illustrated in Section 8.

Most of the laws of list comprehension given here are simple translations of laws from Bird's theory of lists [1]. The advantage of stating the laws in terms of comprehensions rather than in terms of mappings and reductions is that there is less need to invent names for auxiliary functions introduced during derivations. The first of law relates comprehension to the mapping operator ∗:

$$f * a = [fx \mid x \leftarrow a]. \tag{10}$$

This law can be used as the definition of comprehensions with a single generator: $[E \mid x \leftarrow a]$ is defined to mean $f * a$, where f is defined by $fx = E$.

The following laws can be proved by translating them into laws about ∗ using (10). In these laws, the notation $E1_x^{E2}$ means a copy of the expression $E1$ in which the variable x has been replaced by the expression $E2$, provided there is no capture of variables. If f is defined by $fx = E1$, then $E1_x^{E2}$ is equal to $fE2$.

$$[E \mid x \leftarrow [\,]] = [\,] \tag{11}$$

$$[E \mid x \leftarrow [y]] = [E_x^y] \tag{12}$$

$$[E \mid x \leftarrow a + b] = [E \mid x \leftarrow a] + [E \mid x \leftarrow b] \tag{13}$$

$$[g\,E \mid x \leftarrow a] = g * [E \mid x \leftarrow a] \tag{14}$$

$$[E \mid x \leftarrow f * a] = [E^{fy}_{x} \mid y \leftarrow a] \tag{15}$$

$$[E1 \mid x \leftarrow [E2 \mid y \leftarrow a]] = [E1^{E2}_{x} \mid y \leftarrow a]. \tag{16}$$

Law (15) can be used to justify "changing the variable" in a list comprehension. Since $[1..n] = (+1) * [0..n-1]$, it follows that

$$[E \mid i \leftarrow [1..n]] = [E^{i+1}_{i} \mid j \leftarrow [0..n-1]] \tag{17}$$

(Here, $(+1)$ is the function which adds one to its argument.)

If a is a list of pairs, $[E \mid (x, y) \leftarrow a]$ is often written for the list of values take by E as x and y take as value the first and second components of successive elements of a. This notation is made precise by the following definition:

$$[E \mid (x, y) \leftarrow a] = [E^{fstz, sndz}_{x,y} \mid z \leftarrow a].$$

By use of this definition, all the laws of list comprehension can be extended to include this notation.

As law (10) defines comprehensions with one generator, so the following law defines those with two generators in terms of concatenation $+\!+/$:

$$[E \mid x \leftarrow a; y \leftarrow b] = +\!+/[[E \mid y \leftarrow b] \mid x \leftarrow a]. \tag{18}$$

The nesting of the scope of y within that of x on the right-hand side of this law reflects the fact that the second generator "varies faster" than the first.

Law (4) gives a result about reducing over a comprehension with two generators:

$$\oplus/[E \mid x \leftarrow a; y \leftarrow b] = \oplus/[\oplus/[E \mid y \leftarrow b] \mid x \rightarrow a]. \tag{19}$$

Here is the proof:

$$\begin{aligned}
& \oplus/[E \mid x \leftarrow a; y \leftarrow b] \\
= \quad & \{(18)\} \\
& \oplus/+\!+/[[E \mid y \leftarrow b] \mid x \leftarrow a] \\
= \quad & \{(4)\} \\
& \oplus/(\oplus/) * [[E \mid y \leftarrow b] \mid x \leftarrow a] \\
= \quad & \{(14)\} \\
& \oplus/[\oplus/[E \mid y \leftarrow b] \mid x \rightarrow a].
\end{aligned}$$

Also, law (5) allows law (16) to be generalized to the case where the inner comprehension has two generators:

$$[E1 \mid x \leftarrow [E2 \mid y \leftarrow a; z \leftarrow b]] = [E1^{E2}_{x} \mid y \leftarrow a; z \leftarrow b]. \tag{20}$$

7. Laws of exceptions

In this section are collected some laws obeyed by the exception-handling operators such as ? and •. The first two laws state that the operator ? is associative, and

Nothing is a left and right identity element. Implicit appeal has already been made to these laws in the use of the reduction ?/.

$$(x \mathbin{?} y) \mathbin{?} z = x \mathbin{?} (y \mathbin{?} z) \tag{21}$$

$$\mathit{Nothing} \mathbin{?} x = x = x \mathbin{?} \mathit{Nothing}. \tag{22}$$

Three important laws link the operator • with ?, functional composition $(\cdot)$, and the identity function:

$$f \bullet (x \mathbin{?} y) = (f \bullet x) \mathbin{?} (f \bullet y) \tag{23}$$

$$(g \cdot f) \bullet x = g \bullet (f \bullet x) \tag{24}$$

$$\mathit{id} \bullet x = x. \tag{25}$$

All three of these laws can be proved by a simple case analysis on x. As is common when one operator distributes over another one, we adopt the convention that • binds tighter than ?, so that the right-hand side of law (23) could be written without brackets. We also make • associate to the right, so that the right-hand side of law (24) could be written without brackets also.

By abstraction, law (24) can be expressed as an equation between functions:

$$(g \cdot f)\bullet = g\bullet \cdot f\bullet. \tag{26}$$

Law (23) generalizes to a list of arguments using the reduction ?/ and the mapping operator $*$:

$$f \bullet (?/a) = ?/(f\bullet) * a. \tag{27}$$

This law may be combined with law (14) about list comprehensions to give the following result:

$$?/[f \bullet E \mid x \leftarrow a] = f \bullet (?/[e \mid x \leftarrow a]). \tag{28}$$

8. Improving the *inside* function

The function *inside* has a pleasingly direct definition, but it is rather inefficient, because the process of searching for a subterm which can be rewritten is separated from the replacement of that subterm by its rewritten form. A more efficient version of *inside* would combine these two processes into a single traversal of the term. We might regard the version above as a clear but inefficient specification, from which a more efficient but more complicated implementation could be derived by transformation. The laws of list comprehension and exceptions make such a transformation possible, as this section shows.

As a first step, let us examine the function *list_rw*, which searches a list of terms for one which can be rewritten, and replaces it by its rewritten form. Here is a specification in the style of the previous section:

$$\mathit{list\_rw} :: \mathit{strategy} \to \mathit{list\ term} \to \mathit{maybe}\ (\mathit{list\ term})$$
$$\mathit{list\_rw}\ \mathit{rw}\ a = ?/[\mathit{update}\ a\ i \bullet \mathit{rw}\ (a\ !\ i) \mid i \leftarrow [0..\#a-1]]$$

If the list is empty, we may calculate as follows:

$$
\begin{aligned}
& \mathit{list\_rw}\ \mathit{rw}\ [\,] \\
= \quad & \{\text{definition of } \mathit{list\_rw}\} \\
& ?/[\mathit{update}\ [\,]\ i \bullet \mathit{rw}\ ([\,]\ !\ i) \mid i \leftarrow [\,]] \\
= \quad & \{\text{empty generator (11)}\} \\
& ?/[\,] \\
= \quad & \{(1), (22)\} \\
& \mathit{Nothing}.
\end{aligned}
$$

So we derive

$$\mathit{list\_rw}\ \mathit{rw}\ [\,] = \mathit{Nothing}.$$

If the list is non-empty, we calculate:

$$
\begin{aligned}
& \mathit{list\_rw}\ \mathit{rw}\ (x : a) \\
= \quad & \{\text{definition of } \mathit{list\_rw}\} \\
& ?/[\mathit{update}\ (x : a)\ i \bullet \mathit{rw}\ ((x : a)\ !\ i) \mid i \leftarrow [0..\#a]] \\
= \quad & \{\text{split off } i = 0\text{: (13), (12), (3), (2)}\} \\
& \mathit{update}\ (x : a)\ 0 \bullet \mathit{rw}\ ((x : a)\ !\ 0) \\
& ?\ (?/[\mathit{update}\ (x : a)\ i \bullet \mathit{rw}\ ((x : a)\ !\ i) \mid i \rightarrow [1..\#a]]) \\
= \quad & \{(8), \text{definition of } !, (17)\} \\
& (: a) \bullet \mathit{rw}\ x \\
& ?\ (?/[\mathit{update}\ (x : a)\ (j+1) \bullet \mathit{rw}\ ((x : a)\ !\ (j+1)) \mid j \leftarrow [0..\#a-1]]) \\
= \quad & \{(9), (24), \text{definition of } !\} \\
& (: a) \bullet \mathit{rw}\ x \\
& ?\ (?/[(x :) \bullet \mathit{update}\ a\ j \bullet \mathit{rw}\ (a\ !\ j) \mid j \leftarrow [0..\#a-1]]) \\
= \quad & \{(28)\} \\
& (: a) \bullet \mathit{rw}\ x \\
& ?\ (x :) \bullet (?/[\mathit{update}\ a\ j \bullet \mathit{rw}\ (a\ !\ j) \mid j \leftarrow [0..\#a-1]]) \\
= \quad & \{\text{definition of } \mathit{list\_rw}\} \\
& (: a) \bullet \mathit{rw}\ x\ ?\ (x :) \bullet \mathit{list\_rw}\ \mathit{rw}\ a.
\end{aligned}
$$

We have derived another clause in a recursive definition of *list_rw*:

$$\mathit{list\_rw}\ \mathit{rw}\ (x : a) = (: a) \bullet \mathit{rw}\ x\ ?\ (x :) \bullet \mathit{list\_rw}\ \mathit{rw}\ a.$$

Together, the two clauses we have derived are a recursive definition of *list_rw*.

Turning to *inside* itself, we consider first the case where the term being rewritten is a variable:

$$
\begin{aligned}
& \mathit{inside}\ \mathit{rw}\ (\mathit{Var}\ v) \\
= \quad & \{\text{definition of } \mathit{inside}\} \\
& ?/[\mathit{replace}\ (\mathit{Var}\ v)\ k \bullet \mathit{rw}\ u \mid (k, u) \leftarrow \mathit{subterms}\ (\mathit{Var}\ v)] \\
= \quad & \{\text{definition of } \mathit{subterms}, (11)\} \\
& ?/[\,]
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{(1), (22)\} \\
&\quad \mathit{Nothing}.
\end{aligned}
$$

We derive

$$\mathit{inside}\ rw\ (\mathit{Var}\ v) = \mathit{Nothing}.$$

If the term is an application of a function symbol, we reason as follows:

$$
\begin{aligned}
&\quad \mathit{inside}\ rw\ (\mathit{Func}\ f\ a) \\
&= \quad \{\text{definition of } \mathit{inside}\} \\
&\quad ?/[\mathit{replace}\ (\mathit{Func}\ f\ a)\ k \bullet rw\ u \mid (k, u) \leftarrow \mathit{subterms}\ (\mathit{Func}\ f\ a)] \\
&= \quad \{\text{definition of } \mathit{subterms}, (13), (12), (3), (2)\} \\
&\quad \mathit{replace}\ (\mathit{Func}\ f\ a)\ [\,] \bullet rw (\mathit{Func}\ f\ a) \\
&\quad ?\ (?/[\mathit{replace}\ (\mathit{Func}\ f\ a)\ k \bullet rw\ u \mid (k, u) \leftarrow \mathit{list\_subs}\ a])
\end{aligned}
$$

We simplify the two parts of this expression separately. For the first part:

$$
\begin{aligned}
&\quad \mathit{replace}\ (\mathit{Func}\ f\ a)\ [\,] \bullet rw\ (\mathit{Func}\ f\ a) \\
&= \quad \{(6)\} \\
&\quad \mathit{id} \bullet rw\ (\mathit{Func}\ f\ a) \\
&= \quad \{(25)\} \\
&\quad rw\ (\mathit{Func}\ f\ a).
\end{aligned}
$$

For the second part:

$$
\begin{aligned}
&\quad ?/[\mathit{replace}\ (\mathit{Func}\ f\ a)\ k \bullet rw\ u \mid (k, u) \leftarrow \mathit{list\_subs}\ a] \\
&= \quad \{\text{definition of } \mathit{list\_subs}\} \\
&\quad ?/[\mathit{replace}\ (\mathit{Func}\ f\ a)\ k \bullet rw\ u \mid (k, u) \leftarrow [(i : j, u) \mid i \leftarrow [0..\#a-1]; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (j, u) \leftarrow \mathit{subterms}\ (a\ !\ i)]] \\
&= \quad \{(20)\} \\
&\quad ?/[\mathit{replace}\ (\mathit{Func}\ f\ a)\ (i : j) \bullet rw\ u \mid i \leftarrow [0..\#a-1]; (j, u) \leftarrow \mathit{subterms}\ (a\ !\ i)] \\
&= \quad \{(7)\} \\
&\quad ?/[(\mathit{Func}\ f \cdot \mathit{update}\ a\ i \cdot \mathit{replace}\ (a\ !\ i)\ j) \bullet rw\ u \\
&\quad \mid i \leftarrow [0..\#a-1]; (j, u) \leftarrow \mathit{subterms}\ (a\ !\ i)] \\
&= \quad \{(24) \text{ twice}, (28)\} \\
&\quad \mathit{Func}\ f \bullet (?/[\mathit{update}\ a\ i \bullet (\mathit{replace}\ (a\ !\ i)\ j \bullet rw\ u) \\
&\qquad\qquad \mid u \leftarrow [0..\#a-1]; (j, u) \leftarrow \mathit{subterms}\ (a\ !\ i)]) \\
&= \quad \{(19)\} \\
&\quad \mathit{Func}\ f \bullet (?/[?/[\mathit{update}\ a\ i \bullet \mathit{replace}\ (a\ !\ i)\ j \bullet rw\ u) \\
&\qquad\qquad\qquad \mid (j, u) \leftarrow \mathit{subterms}\ (a\ !\ i)] \mid i \leftarrow [0..\#a-1]]) \\
&= \quad \{(28)\}
\end{aligned}
$$

$$
\begin{aligned}
& Func\, f \bullet (?/[update\ a\ i \bullet (?/[replace\ (a\ !\ i)\ j \bullet rw\ u \\
& \qquad\qquad \mid (j, u) \leftarrow subterms\ (a\ !\ i)]) \\
& \qquad \mid i \leftarrow [0..\#a - 1]]) \\
= \quad & \{\text{definition of } inside\} \\
& Func\, f \bullet (?/[update\ a\ i \bullet inside\ rw\ (a\ !\ i) \mid i \leftarrow [0..\#a - 1]]) \\
= \quad & \{\text{definition of } list\_rw\} \\
& Func\, f \bullet list\_rw\ (inside\ rw)\ a.
\end{aligned}
$$

Putting the two parts together, we have derived:

$$
\begin{aligned}
& inside\ rw\ (Func\, f\, a) \\
& \qquad = rw\ (Func\, f\, a)\ ?\ Func\, f \bullet list\_rw\ (inside\ rw)\ a.
\end{aligned}
$$

In summary, here is the code for *inside* which we have derived:

$$
\begin{aligned}
& inside :: strategy \to strategy \\
& inside\ rw\ (Var\ v) = Nothing \\
& inside\ rw\ (Func\, f\, a) \\
& \qquad = rw\ (Func\, f\, a)\ ?\ Func\, f \bullet list\_rw\ (inside\ rw)\ a \\
& \\
& list\_rw :: strategy \to list\ term \to maybe\ (list\ term) \\
& list\_rw\ rw\ [\,] = Nothing \\
& list\_rw\ rw\ (x : a) = (: a) \bullet rw\ x\ ?\ (x :) \bullet list\_rw\ rw\ a.
\end{aligned}
$$

It has been possible to derive this more efficient version of *inside* thanks to the algebraic properties of the exception-handling operators such as • and ?, which can be expressed only by making the propagation of exceptions explicit. However, the code can be translated into a language where propagation is implicit. It might be translated as follows:

$$
\begin{aligned}
& inside :: (term \to term) \to term \to term \\
& inside\ rw\ (Var\ v) = \textbf{fail} \\
& inside\ rw\ (Func\, f\, a) \\
& \qquad = rw\ (Func\, f\, a)\ \textbf{orelse}\ Func\, f\ (list\_rw\ (inside\ rw)\ a) \\
& \\
& list\_rw :: (term \to term) \to list\ term \to list\ term \\
& list\_rw\ rw\ [\,] = \textbf{fail} \\
& list\_rw\ rw\ (x : a) = (rw\ x) : a\ \textbf{orelse}\ x : (list\_rw\ rw\ a).
\end{aligned}
$$

9. The algebra of strategies

Rewriting strategies are one example of *partial functions*. More generally, a partial function from a type α to a type β is a function in $\alpha \to maybe\ \beta$. Define $\alpha \leftrightarrow \beta$ to

be this type of partial functions:

$$\alpha \mapsto \beta == \alpha \to maybe\ \beta.$$

Rewriting strategies are elements of the type $term \mapsto term$.

Two functions in $\alpha \mapsto \beta$ may be combined with the associative operator ??, whose definition we now generalize as follows:

$$(??) :: (\alpha \mapsto \beta) \to (\alpha \mapsto \beta) \to (\alpha \mapsto \beta)$$
$$(f\ ??\ g)\ x = f\ x\ ?\ g\ x.$$

The operator ?? is analogous to the *orelse* tactical of LCF; as before, it has the identity element *fail*, whose definition generalizes as follows:

$$fail :: \alpha \mapsto \beta$$
$$fail\ x = Nothing.$$

Partial functions may also be composed with the sequencing operator $\circ$, analogous to the tactical *then* in LCF. It is defined as follows:

$$(\circ) :: (\beta \mapsto \gamma) \to (\alpha \mapsto \beta) \to (\alpha \mapsto \gamma)$$
$$g \circ f = prop \cdot g\bullet \cdot f,$$

where *prop* is a function which propagates exceptions, merging the two failure values *Nothing* and *Just Nothing* in the type $maybe\ (maybe\ \alpha)$. Here is its definition:

$$prop :: maybe\ (maybe\ \alpha) \to maybe\ \alpha$$
$$prop\ Nothing = Nothing$$
$$prop\ (Just\ x) = x.$$

The following laws about *prop* can be proved by case analysis. Here is a law which asserts the equality of two different ways of combining *prop* with itself to make a function of type $maybe\ (maybe\ (maybe\ \alpha)) \to maybe\ \alpha$:

$$prop \cdot prop\bullet = prop \cdot prop. \tag{29}$$

On the left-hand side of this equation, both occurrences of *prop* have type $maybe\ (maybe\ \alpha) \to maybe\ \alpha$; on the right, the first occurrence has this type, but the second has type $maybe\ (maybe\ (maybe\ \alpha)) \to maybe\ (maybe\ \alpha)$.

A second law shows how the $\bullet$ operator interacts with *prop*. If $f : \alpha \mapsto \beta$, then

$$prop \cdot (f\bullet)\bullet = f\bullet \cdot prop. \tag{30}$$

These laws can be used to show that $\circ$ is associative: if $f : \alpha \mapsto \beta$, $g : \beta \mapsto \gamma$, and

$h : \gamma \mapsto \delta$, then

$$
\begin{aligned}
& (h \circ g) \circ f \\
= \quad & \{\text{definition of } \circ\} \\
& prop \cdot (h \circ g)\bullet \cdot f \\
= \quad & \{\text{definition of } \circ\} \\
& prop \cdot (prop \cdot g\bullet \cdot g)\bullet \cdot f \\
= \quad & \{(26) \text{ twice}\} \\
& prop \cdot prop\bullet \cdot (h\bullet)\bullet \cdot g\bullet \cdot f \\
= \quad & \{(29)\} \\
& prop \cdot prop \cdot (h\bullet)\bullet \cdot g\bullet \cdot f \\
= \quad & \{(30)\} \\
& prop \cdot h\bullet \cdot prop \cdot g\bullet \cdot f \\
= \quad & \{\text{definition of } \circ\} \\
& prop \cdot h\bullet \cdot (g \circ f) \\
= \quad & \{\text{definition of } \circ\} \\
& h \circ (g \circ f).
\end{aligned}
$$

This argument uses nothing but the definition of $\circ$ in terms of *prop* and $\bullet$, law (26), and the laws (29) and (30) about *prop*.

As we saw earlier, there is another model for exceptions in which *maybe* α is identified with the type *list* α lists of elements of α. In this model, the operator $\bullet$ is the same as the mapping operator $*$, and *prop* is the same as $+\!+/$. Law (26) is just the familiar law

$$(g \cdot f)* = g* \cdot f*,$$

about $*$, law (29) is the same as law (5), and law (30) becomes

$$+\!+/ \cdot (+\!+/)* = +\!+/ \cdot +\!+/.$$

This asserts the equivalence of two ways of flattening a list of lists into a simple list. Both $\bullet$ and *prop* can be defined in the list-of-successes model of exceptions, and they satisfy the same laws as in the simpler model. In consequence, we can use the same definition of the composition operator $\circ$, and the proof that it is associative is precisely the same.

Category theory provides an explanation of why this works, because $\bullet$ and *prop* are two parts of a *monad*, a standard categorical concept. The third part of the monad is the partial function *succeed*, defined in the simpler model by:

$$
\begin{aligned}
& succeed :: \alpha \mapsto \alpha \\
& succeed\ x = Just\ x.
\end{aligned}
$$

In the list-of-successes model, *succeed* is the function $[\cdot]$ which maps each object x to the singleton $[x]$.

A construction due to Kleisli [5] shows how to make a new category from a monad, in which the arrows are analogous to our partial functions. Our definition

of the composition operator $\circ$ is a copy of the definition of composition in the Kleisli category, and the proof of its associativity is a copy of the proof in category theory. There are also categorical proofs that *succeed* is a left and right identity for $\circ$ from the laws:

$$prop \cdot succeed\bullet = id = prop \cdot succeed \tag{31}$$

$$succeed \cdot f = f\bullet \cdot succeed, \tag{32}$$

each of which holds in both models for exceptions.

Many of the laws obeyed by the operators can be seen in categorical terms. Briefly, the type constructor *maybe* is the object part of a functor from types to types; its arrow part is the operator $\bullet$. If $f : \alpha \to \beta$ then $f\bullet : maybe\ \alpha \to maybe\ \beta$, and laws (25) and (26) state that $\bullet$ respects identities and composition as a functor should. Operators such as ?, *prop* and *succeed* are natural transformations—see laws (23), (30) and (32) respectively. The equations needed for ($\bullet$, *succeed*, *prop*) to be a monad are laws (31) and (29).

The composition operator $\circ$ can be used to define a repetition operator for rewriting strategies. The function *repeat* takes a function in $\alpha \mapsto \alpha$ and applies it repeatedly until it fails. In the simple model of exceptions, the result is the last value obtained before failure.

$$repeat :: (\alpha \mapsto \alpha) \to (\alpha \mapsto \alpha)$$
$$repeat\, f = (repeat\, f \circ f)\ ??\ succeed.$$

A common application is to reduce a term to normal form with respect to a list of equations. This is achieved by the function *normalize*, defined by

$$normalize :: list\ equation \to term \to term$$
$$normalize\ eqns\ t = u \textbf{ where } Just\ u = repeat\ (reduce\ eqns)\ t.$$

It is quite safe to match the result of *repeat* with the pattern *Just u*, because if evaluation of *repeat* (*reduce eqns*) *t* terminates at all, it must terminate in success. Of course, reduction of the term *t* may not terminate, in which case neither does the evaluation.

In the list-of-successes model, *repeat f x* returns a list of all possible results from applying *f* repeatedly to *x*. This list is the post-order traversal of a search tree with *x* at the root, in which the immediate descendants of each node *y* are the results of applying *f* to *y*, if any. If the search tree is finite, then it will have a post-order traversal, and the first member of the traversal will be a leaf of the tree: in the application to term rewriting, a normal form for the term.

10. Conclusion

Exceptions as a control structure need to be built in to a programming language, and they prevent simple equational reasoning about programs. I have tried to show

how a lazy evaluation strategy turns exceptions from a control structure into a data structure, which can be defined within a functional programming language. There is a small price to pay for this simplification, and that is the need to make explicit in the type of a function the possibility that it will raise an exception, and to make explicit in the program the way exceptions are propagated. I have tried to show by example that the use of suitable higher-order functions on exceptions makes the price seem tolerable.

The advantage bought for this price is the ease of reasoning about programs with exceptions. Programs such as *inside* can be derived by rigorous transformation from simple but inefficient specifications, and algebraic laws about exception-handling functions can be established by rigorous mathematical argument.

Many of the operators on exceptions can be described abstractly using terminology from category theory, and the two models for exceptions share a common categorical specification. The proof of some of the algebraic properties of exception-handling functions—for example, the associativity of the *then* tactical—is based on this categorical specification, and so is independent of the model chosen. More generally, many of the laws in Bird's "theory of lists" [1, 2] can be seen as asserting that various categorical constructions are functors, natural transformations, adjunctions and so on. I work out some of the details of this view of lists in the paper [8].

Acknowledgement

I am grateful to Richard Bird and Phil Wadler for their wise advice and generous encouragement, and to Oriel College, Oxford, Rank Xerox (UK) Ltd., and the Science and Engineering Research Council of Great Britain for financial support.

References

[1] R.S. Bird, An introduction to the theory of lists, in: M. Broy, ed., *Logics of Programming and Calculi of Discrete Design* (Springer, Berlin, 1987) 3–42.

[2] R.S. Bird, A calculus of functions for program derivation, Tech. Monograph PRG-64, Oxford University Computing Laboratory, Oxford (1987).

[3] R.S. Bird and P.L. Wadler, *An Introduction to Functional Programming* (Prentice-Hall International, Hemel Hempstead, England, 1988).

[4] M. Gordon, R. Milner and C. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science **78** (Springer, Berlin, 1979).

[5] S. MacLane, *Categories for the Working Mathematician* (Springer, Berlin, 1971).

[6] L. Paulson, A higher-order implementation of rewriting, *Sci. Comput. Programming* **3** (1983) 119–149.

[7] L. Paulson, *Logic and Computation* (Cambridge University Press, Cambridge, 1988).

[8] J. M. Spivey, A categorical approach to the theory of lists, in: *Proceedings International Conference on Mathematics of Program Construction*, Enschede, Netherlands (1989).

[9] D. Turner, *An overview of* MIRANDA, *SIGPLAN Notices* (1986).

[10] P. L. Wadler, How to replace failure by a list of success, in: J.-P. Jouannaud, ed., *Proceedings Second International Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201** (Springer, Berlin, 1985).