Monadic-style Backtracking

Ralf Hinze

Abstract

Lists are ubiquitous in functional programming. The list constructor forms an instance of a monad capturing non-deterministic computations. Despite its popularity the list monad suffers from serious drawbacks: It relies in an essential way on lazy evaluation, it is inefficient, and it is not modular. We develop an alternative based on continuations, which remedies these shortcomings. Essential use is made of constructor classes and second-order types, which sets the work apart from other approaches. Continuation-based backtracking monads behave amazingly well in practice: If an optimizing compiler is used, their performance is commensurate to that of logic languages. The class mechanism greatly eases the task of adding features to the basic machinery. We study three extensions in detail: control operations, all solutions functions, and exception handling.

Contents

1 The monadic machinery

In the following we briefly review the basic facts about monads. For a more gentle introduction to the topic see, for example, Wadler's papers (1990) or (1995). The definitions and examples are given in Haskell 1.3 syntax (Peterson & Hammond, 1996) augmented by multiple parameter constructor classes (Jones, 1995).

1.1 Class definitions

From the programmer's eye view we can think of a monad as an abstract data type which comes equipped with two basic operations.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m\,\mathbf{where} \\
\quad (\gg=) :: m\ a \to (a \to m\ b) \to m\ b \\
\quad (\gg) :: m\ a \to m\ b \to m\ b \\
\quad \mathit{return} :: a \to m\ a \\
\quad m \gg k = m \gg= \lambda\_ \to k
\end{array}
$$

Note that $(\gg=)$ is sometimes called *bind*; alternative names for *return* include *unit* and *result*. Intuitively, $m\ a$ denotes a computation which returns a value of type a: $\mathit{return}\ a$ is the empty computation which immediately returns a; $m \gg= k$ runs m and feeds the result into k. The two functions must be related by the following laws.

$$
\begin{array}{lrcl}
(1) & \mathit{return}\ a \gg= k & = & k\ a \\
(2) & m \gg= \mathit{return} & = & m \\
(3) & m \gg= (\lambda a \to k\ a \gg= \ell) & = & (m \gg= k) \gg= \ell
\end{array}
$$

If we ignore that the second argument of $(\gg=)$ involves a binding operation, we have that $(\gg=)$ is associative with *return* as its left and right unit. The laws are easier to remember if we rephrase them in terms of the Kleisli composition.

$$
\begin{array}{l}
(@) :: (\mathit{Monad}\ m) \Rightarrow (b \to m\ c) \to (a \to m\ b) \to (a \to m\ c) \\
f\ @\ g = \lambda a \to g\ a \gg= f
\end{array}
$$

Looking at the types we see that $(\gg=)$ corresponds to the reverse function application and that $(@)$ is similar to the ordinary composition of functions. However, contrary to the pure operations $(\gg=)$ and $(@)$ may additionally involve actions 'behind the scenes'. Using the Kleisli composition the monoidal structure becomes apparent.

$$
\begin{array}{lrcl}
(1') & f\ @\ \mathit{return} & = & f \\
(2') & \mathit{return}\ @\ f & = & f \\
(3') & (f\ @\ g)\ @\ h & = & f\ @\ (g\ @\ h)
\end{array}
$$

Monads differ in the additional operations they offer to the user: A state monad, for instance, provides means to manipulate the state; an exception monad enables its user to handle exceptional situations etc. It makes sense to prescribe the operations a particular monad must support. For each feature we shall introduce a subclass of *Monad* which lists the operations supporting this feature. This extensive use of the class system has two benefits: (1) The features an application requires are reflected in its type, see Section 2 and 7.2 for numerous examples. (2) Overloading the operations greatly eases the task of constructing monads which support a variety of different features. This will be the topic of Section 3.

To support nondeterministic programming a monad has to provide two additional operations: *zero* and $(+\!\!+)$. The term *zero* denotes a computation which fails, $(+\!\!+)$ realizes a nondeterministic choice.

$$
\begin{array}{l}
\mathbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadZero}\ m\,\mathbf{where} \\
\quad \mathit{zero} :: m\ a \\
\mathbf{class}\ (\mathit{MonadZero}\ m) \Rightarrow \mathit{MonadPlus}\ m\,\mathbf{where} \\
\quad (+\!\!+) :: m\ a \to m\ a \to m\ a
\end{array}
$$

Instances of these classes are supposed to satisfy additional laws. Usually *zero* is required to be left and right zero of $(\gg\!=)$. However, the latter demand is too strong: Suppose that the monad also offers the possibility of raising exceptions — think of exceptions as hard failures, *zero* being a soft failure. We would then expect that $\mathit{raise}\ e \gg\!= k = \mathit{raise}\ e$ holds. Consequently, *zero* cannot be a *right* zero.

$$
\begin{array}{lrcl}
(4) & \mathit{zero} \gg\!= k & = & \mathit{zero} \\
(5) & m \gg \mathit{zero} & \stackrel{?}{=} & \mathit{zero}
\end{array}
$$

The question mark indicates that (5) is optional. The operations *zero* and $(+\!\!+)$ form a monoid, ie *zero* is the identity for $(+\!\!+)$, and $(+\!\!+)$ is associative. We do not expect $(+\!\!+)$ to be commutative: Nondeterminism can generally only be approximated, so the order in which alternatives are presented usually matters. On the same grounds $(+\!\!+)$ need not be idempotent.

$$
\begin{array}{lrcl}
(6) & \mathit{zero} +\!\!+ m & = & m \\
(7) & m +\!\!+ \mathit{zero} & = & m \\
(8) & (m +\!\!+ n) +\!\!+ o & = & m +\!\!+ (n +\!\!+ o) \\
(9) & (m \gg\!= k) +\!\!+ (n \gg\!= k) & = & (m +\!\!+ n) \gg\!= k \\
(10) & (m \gg\!= k) +\!\!+ (m \gg\!= \ell) & \stackrel{?}{=} & m \gg\!= \lambda a \to k\ a +\!\!+ \ell\ a
\end{array}
$$

The last two equations are concerned with the interplay of bind and plus: $(\gg\!=)$ is expected to distribute leftward through $(+\!\!+)$ but not necessarily rightward. The following examples illustrates why (10) fails in general: Let $m = \mathit{return}\ a_1 +\!\!+ \cdots +\!\!+ \mathit{return}\ a_n$ then

$$
\begin{array}{cl}
 & (m \gg\!= k) +\!\!+ (m \gg\!= \ell) \\
= & \{\ \text{by (9) and by (1)}\ \} \\
 & k\ a_1 +\!\!+ \cdots +\!\!+ k\ a_n +\!\!+ \ell\ a_1 +\!\!+ \cdots +\!\!+ \ell\ a_n \\
\neq & k\ a_1 +\!\!+ \ell\ a_1 +\!\!+ \cdots +\!\!+ k\ a_n +\!\!+ \ell\ a_n \\
= & \{\ \text{by (9) and by (1)}\ \} \\
 & m \gg\!= \lambda a \to k\ a +\!\!+ \ell\ a
\end{array}
$$

Table 1. *The relationship of monads to cf grammars and moded logic programs*

monads	cf grammars	moded logic programs
$m \gg n$	$m\ n$	$m,\ n$
$return\ ()$	ϵ	$true$
$zero$	$\emptyset$	$false$
$m +\!\!+ n$	$m \mid n$	$m;\ n$
$(\gg\!=)$	semantic actions	output unification

Hence, (10) only holds if either $(+\!\!+)$ is commutative or m is unambiguous ie succeeds at most once. It is instructive to reformulate the distributivity laws in terms of the Kleisli composition. To this end we first raise $(+\!\!+)$ to functions of type $a \to m\ b$ (the arrows of the Kleisli category).

$$(+\!\!\!+\!\!\!+) :: (MonadPlus\ m) \Rightarrow (a \to m\ b) \to (a \to m\ b) \to (a \to m\ b)$$
$$f +\!\!\!+\!\!\!+ g = \lambda a \to f\ a +\!\!+ g\ a$$

Then the laws become

$$\begin{array}{lrcl} (9') & f @ g +\!\!\!+\!\!\!+ f @ h & = & f @ (g +\!\!\!+\!\!\!+ h) \\ (10') & f @ h +\!\!\!+\!\!\!+ g @ h & = & (f +\!\!\!+\!\!\!+ g) @ h. \end{array}$$

Remark

Monads with plus capture two fundamental notions of computing: sequencing and alternation. Table 1 relates monads with plus to other formal systems: context-free grammars and moded logic programming languages. It is interesting to note that cf grammars have per se no notion of 'computed value'. The counterpart of bind — a semantic action — is often only introduced when parser generators come into play. The tight relationship between monads and cf grammars is further highlighted in Section 2.2 where we exemplify how to derive a recursive descent parsers from a cf grammar.

Nondeterministic programming is not a very new idea. It dates back at least to the early seventies when the archetype of logic languages, Prolog, was conceived. Table 1 shows that the relationship between monads and logic programs is quite direct: sequencing corresponds to logical conjunction, alternation to disjunction. Note, however, that logic programs are more flexibel: A logic program defines a set of *relations* ie there is no notion of input and output. The well-known `append` relation, for instance, may be used to concatenate two lists or to split a list into two. In Haskell we are forced to define two functions, one for each mode: $[a] \to [a] \to m\ [a]$ and $[a] \to m\ ([a], [a])$. However, once the modes are fixed the translation of logic programs into monadic-style functional programs is almost mechanical. Section 2.1 contains several examples.

End of remark

A state monad must provide an update operation.

$$\begin{array}{l}\textbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadState}\ st\ m\,\textbf{where}\\ \quad \mathit{update} :: (st \to st) \to m\ st\end{array}$$

The operation *update* applies its argument to the current state, and returns the old state. Hence we can read the state by passing *update* the identity function, and set the state to s by passing it *const s*. We refrain from listing laws about *update*.

A parser monad is a monad with *zero* — a parse may fail — which offers access to the next token of the input. Clearly, a state monad with *zero* makes a parser monad.

$$\begin{array}{l}\textbf{class}\ (\mathit{MonadZero}\ m) \Rightarrow \mathit{MonadParser}\ tok\ m\,\textbf{where}\\ \quad \mathit{item} :: m\ tok\\ \textbf{instance}\ (\mathit{MonadZero}\ m, \mathit{MonadState}\ [tok]\ m) \Rightarrow \mathit{MonadParser}\ tok\ m\,\textbf{where}\\ \quad \mathit{item} = \mathit{update}\ \mathit{tail} \ggg\!= \lambda x \to \textbf{case}\ x\,\textbf{of}\ [\,] \to \mathit{zero};\ a : \_ \to \mathit{return}\ a\end{array}$$

The operation *item* returns the first token of the input list and fails if the list is empty. The reader may wonder why we have overloaded the function *item*. Alternatively, we could have turned it into a generic function of type

$$(\mathit{MonadZero}\ m, \mathit{MonadState}\ [tok]\ m) \Rightarrow m\ tok$$

simply by stripping off the class and instance declarations. However, overloading has the advantage that it is possible to specialize *item* for some instances in the interests of efficiency (see Section 4.4).

We have taken the view of monads as an abstract data type: *return* gets us into a monad, and $(\ggg\!=)$ gets us around. However, we have not yet devised a way to get out of a monad. Usually, this is done on a case by case basis. We adopt a more systematic approach which is inspired by the categorical concept of a natural transformation. First note that the most obvious choice, an operation of type $m\ a \to a$, does not take us very far. For example, if $m\ a$ is a nondeterministic computation we can in general not expect to obtain a single solution only. For this special case $m\ a \to [\,]\ a$ appears to be a better choice. Abstracting the type constructor $[\,]$ away we define:[†]

$$\begin{array}{l}\textbf{class}\ \mathit{Run}\ m\ n\,\textbf{where}\\ \quad \mathit{run} :: m\ a \to n\ a\\ \textbf{instance}\ \mathit{Run}\ m\ m\,\textbf{where}\\ \quad \mathit{run} = \mathit{id}\end{array}$$

Now, to enumerate the solutions of a nondeterministic computation of type $M\ a$ we simply apply $\mathit{run} :: M\ a \to [\,]\ a$. If we are only interested in the first solution we could use $\mathit{run} :: M\ a \to \mathit{Maybe}\ a$.

[†] Note that the instance declaration is cheated: Since it overlaps with another one given in Section 3.1 we are, in fact, forced to specialize it for each m of interest.

1.2 Derived functions

For the examples following we need a couple of auxiliary functions which are defined below. Some of these definitions employ so-called do expressions which provide a more readable, first-order syntax for $(\gg=)$ and $(\gg)$. The syntax and semantics of do expressions are given by the following identities:

$$\begin{array}{rcl}
\textbf{do}\ \{m; e\} & = & m \gg \textbf{do}\ \{e\} \\
\textbf{do}\ \{p \leftarrow m; e\} & = & m \gg= \lambda p \rightarrow \textbf{do}\ \{e\} \\
\textbf{do}\ \{\textbf{let}\, bs; e\} & = & \textbf{let}\, bs \,\textbf{in do}\ \{e\} \\
\textbf{do}\ \{e\} & = & e
\end{array}$$

Note that p must be 'failure-free' in the second equation, see the Haskell 1.3 Report (Peterson & Hammond, 1996) for a precise definition of failure-free. All the patterns we employ — variables and tuples of variables — do satisfy this constraint.

$$\begin{array}{l}
\mathit{when}, \mathit{unless} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Bool} \rightarrow m\ () \rightarrow m\ () \\
\mathit{when}\ b\ m = \textbf{if}\ b\,\textbf{then}\ m\,\textbf{else}\ \mathit{return}\ () \\
\mathit{unless}\ b\ m = \textbf{if}\ b\,\textbf{then}\ \mathit{return}\ ()\textbf{else}\ m
\end{array}$$

The *when* construct provides a form of conditional execution: The second argument is executed if the first evaluates to *True*. The operation *unless* is similar to *when* except that its complements the Boolean condition.

$$\begin{array}{l}
\mathit{guard} :: (\mathit{MonadZero}\ m) \Rightarrow \mathit{Bool} \rightarrow m\ () \\
\mathit{guard}\ b = \textbf{if}\ b\,\textbf{then}\ \mathit{return}\ ()\textbf{else}\ \mathit{zero}
\end{array}$$

The operation *guard* maps a Boolean value to the corresponding monad operation: *True* to *return* () and *False* to *zero* (cf Table 1).

$$\begin{array}{l}
(\triangleright) :: (\mathit{MonadZero}\ m) \Rightarrow m\ a \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow m\ a \\
m \triangleright p = \textbf{do}\ \ a \leftarrow m;\ \textbf{if}\ p\ a\,\textbf{then}\ \mathit{return}\ a\,\textbf{else}\ \mathit{zero}
\end{array}$$

The operation $(\triangleright)$ is called a filter: $m \triangleright p$ runs m yielding a value a; if $p\ a$ holds then a is returned, otherwise it fails.

We have already stated that monads with *zero* and $(+\!\!+)$ closely correspond to context-free grammars: $(\gg=)$ implements sequencing and $(+\!\!+)$ alternation. However, cf grammars are seldom used in practice. Usually one employs a variation of EBNF since the additional notation eases the specification of formal languages. The extensions EBNF provides are easily defined in the monadic framework.

An optional occurrence of m is given by $\mathit{opt}\ m\ a$, where a is the value returned if m does not occur.

$$\begin{array}{l}
\mathit{opt} :: (\mathit{MonadPlus}\ m) \Rightarrow m\ a \rightarrow a \rightarrow m\ a \\
\mathit{opt}\ m\ a = m +\!\!+ \mathit{return}\ a
\end{array}$$

Note the similarity to the definition of the EBNF meta symbol: $[m] = m \mid \epsilon$. Zero or more occurrences of m are defined accordingly: $m^* = mm^* \mid \epsilon$.

$$
\begin{array}{l}
\mathit{many} :: (\mathit{MonadPlus}\ m) \Rightarrow m\ a \rightarrow m\ [a] \\
\mathit{many}\ m = \mathit{ms} \\
\quad \textbf{where}\ \mathit{ms} = \textbf{do}\ \ a \leftarrow m \\
\qquad\qquad\qquad\qquad x \leftarrow \mathit{ms} \\
\qquad\qquad\qquad\qquad \mathit{return}\ (a : x) \\
\qquad\qquad\qquad +\!\!\!+ \mathit{return}\ []
\end{array}
$$

The notation $m\langle s\rangle$ is often used to denote one or more occurrences of m separated by s: $m\langle s\rangle = m(sm)^*$.

$$
\begin{array}{l}
\mathit{sepBy} :: (\mathit{MonadPlus}\ m) \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ [a] \\
\mathit{sepBy}\ m\ s = \textbf{do}\ \ a \leftarrow m \\
\qquad\qquad\qquad\quad x \leftarrow \mathit{many}\ (\textbf{do}\ \ s;\ m) \\
\qquad\qquad\qquad\quad \mathit{return}\ (a : x)
\end{array}
$$

Finally, we provide an operation which tests whether a parser based on a state monad accepts a given sequence and in case it does returns its semantic value.

$$
\begin{array}{l}
\mathit{accepts} :: (\mathit{MonadZero}\ m, \mathit{MonadState}\ [\mathit{tok}]\ m) \Rightarrow m\ a \rightarrow [\mathit{tok}] \rightarrow m\ a \\
\mathit{accepts}\ p\ s = \textbf{do}\ \ \mathit{old} \leftarrow \mathit{update}\ (\lambda\_ \rightarrow s) \\
\qquad\qquad\qquad\quad a \leftarrow p \\
\qquad\qquad\qquad\quad t \leftarrow \mathit{update}\ (\lambda\_ \rightarrow \mathit{old}) \\
\qquad\qquad\qquad\quad \mathit{guard}\ (\mathit{null}\ (t\,`\mathit{asTypeOf}`\,s)) \\
\qquad\qquad\qquad\quad \mathit{return}\ a
\end{array}
$$

Note that the guard checks whether the input is completely consumed. Furthermore note that the old state, ie the old token list, is saved before m is called and re-installed upon exit. This ensures that nested calls to *accept* work properly, see, Section 4.4 for an application. *Technical remark:* The type constraint $t\,`\mathit{asTypeOf}`\,s$ is necessary as to ensure that the two *update* operations refer to the same internal state.

1.3 Monad morphisms

Each mathematical structure comes equipped with structure preserving maps, so do monads: A monad morphism from M to N is a function $\eta :: M\ a \rightarrow N\ a$ that preserves *return* and $(\gg\!=)$.

$$
\begin{array}{lrcl}
(11) & \eta\ \mathit{return} & = & \mathit{return} \\
(12) & \eta\ (m \gg\!= k) & = & \eta\ m \gg\!= (\eta \circ k)
\end{array}
$$

A function that satisfies only equation (11) is termed premonad morphism.

We will see that monad morphisms play a central rôle in adding features to a monad. A monad morphism $\eta :: M\ a \rightarrow N\ a$ is called isomorphism iff there is a morphism $\mu :: N\ a \rightarrow M\ a$ such that $\eta \circ \mu = \mathit{id}$ and $\mu \circ \eta = \mathit{id}$. Isomorphic monads exhibit the same computational behaviour. Thus we are free, for example, to replace a monad by an isomorphic, but more efficient one. This will be the main theme of Section 4.

Additional operations a monad supports must be preserved as well.

$$\begin{array}{lrcl}
(13) & \eta\ zero & = & zero \\
(14) & \eta\ (m +\!\!+\ n) & = & \eta\ m +\!\!+\ \eta\ n \\
(15) & \eta\ (update\ s) & = & update\ s
\end{array}$$

All in all the equations ensure that a computation does not change its behaviour when it is executed in a different monad: Let c be overloaded with respect to the monad operations and let $\eta :: M\ a \to N\ a$ be a monad morphism, then we have that $\eta\ (c :: M\ a) = c :: N\ a$.

2 Applications

This section presents several examples of monadic-style backtracking and parsing.

2.1 Pure backtracking

The following examples are inspired by similar ones curling around the 'Biblical family database' in the introductory textbook on Prolog by Sterling and Shapiro (1986). Let us assume that the Biblical family database is given by two relations: $person \subseteq String$ and $child \subseteq (String, String)$ such that $(p, c) \in child$ holds if c is the child of p. How do we represent these relations in the monadic framework? As a first try we could realize the relation $r \subseteq T$ as a nondeterministic computation $r :: (MonadPlus\ m) \Rightarrow m\ T$. While this certainly works it is for many applications too inefficient, for instance, if we are interested in the childs of a given person. The database expert will notice that the last query calls for an index on the first argument of *child*. Accordingly, a Prolog expert will recognize that *child* is called in the mode `(in,out)`. An index is easy to implement: We represent *child* by a function of type $(MonadPlus\ m) \Rightarrow String \to m\ String$. Thus *child s* enumerates the childs of *s*. If *s* has *n* childs, *child s* succeeds exactly *n* times.[‡]

```
person :: (MonadPlus m) => m String
person = return "abraham" ++ return "haran" ++ return "isaac"
            ++return "lot" ++ return "milcah" ++ return "nachor"
            ++return "sarah" ++ return "terach" ++ return "yiscah"
```

[‡] Note that we use or rather misuse Haskell's pattern matching facility as an index mechanism. Of course, nothing prevents us except time and space from employing a sophisticated data structure, say, a B-tree instead.

$$
\begin{aligned}
&child :: (MonadPlus\ m) \Rightarrow String \rightarrow m\ String\\
&child\ \texttt{"terach"} = return\ \texttt{"abraham"} \mathbin{+\!\!+} return\ \texttt{"nachor"} \mathbin{+\!\!+} return\ \texttt{"haran"}\\
&child\ \texttt{"abraham"} = return\ \texttt{"isaac"}\\
&child\ \texttt{"haran"} = return\ \texttt{"lot"} \mathbin{+\!\!+} return\ \texttt{"milcah"} \mathbin{+\!\!+} return\ \texttt{"yiscah"}\\
&child\ \texttt{"sarah"} = return\ \texttt{"isaac"}\\
&child\ \_ = zero
\end{aligned}
$$

Using *person* and *child* we can easily list all pairs of persons for which the child relationship holds.

$$\mathbf{do}\ \ p \leftarrow person;\ c \leftarrow child\ p;\ return\ (p, c)$$

Note that the last equation of *child* explicitly states that persons not listed before have no childs. Negative information of this kind is left implicit in the corresponding logic program.

The transitive closure of the child relationship is given by *descendant*.

$$
\begin{aligned}
&descendant :: (MonadPlus\ m) \Rightarrow String \rightarrow m\ String\\
&descendant = transitiveClosure\ child
\end{aligned}
$$

$$
\begin{aligned}
&transitiveClosure :: (MonadPlus\ m) \Rightarrow (a \rightarrow m\ a) \rightarrow (a \rightarrow m\ a)\\
&transitiveClosure\ m = m \mathbin{+\!\!+\!\!+} (transitiveClosure\ m\ @\ m)
\end{aligned}
$$

The definition of *descendant* in terms of *transitiveClosure* nicely illustrates the use of higher-order functions for capturing common patterns of computation. In a first-order language we are forced to repeat the definition of *transitiveClosure* for every base relation.

The code of *transitiveClosure* exhibits a slight inefficiency: The base relation m is called twice at each level of recursion. This can be avoided by right factorising *transitiveClosure*, ie by applying equation (10) respectively (10′).

$$
\begin{aligned}
&transitiveClosure' :: (MonadPlus\ m) \Rightarrow (a \rightarrow m\ a) \rightarrow (a \rightarrow m\ a)\\
&transitiveClosure'\ m = (return \mathbin{+\!\!+\!\!+} transitiveClosure'\ m)\ @\ m
\end{aligned}
$$

Unfortunately, *transitiveClosure´* and *transitiveClosure* are not equal, since (10) fails to be valid in general: We have seen in Section 1.1 that the solutions are enumerated in a different order. But perhaps one does not care.

Assume that the monad *BacktrM* is an instance of *MonadPlus* which implements backtracking. Then we can list all descendants of Terach as follows.

$$run\ (descendant\ \texttt{"terach"} :: BacktrM\ String) :: [\,]\ String$$

The construction of a suitable backtracking monad is discussed at length in Sections 3 and 4.

A common technique employed by Prolog programmers is generate-and-test. One of the standard instances of this technique tackles the 8 queens problem: eight queens must be placed on a standard chess board so that no two pieces are threatening each other under the rules of chess. Of course, we solve the problem generalized to n. The algorithmic idea is quite simple: The queens are placed columnwise from right to left; each new position is checked against the previously placed queens.

$$
\begin{aligned}
&queens :: (MonadPlus\ m) \Rightarrow Int \rightarrow m\ [Int] \\
&queens\ n = place\ n\ [1..n]\ []\ []
\end{aligned}
$$

The third argument of *place* records the threatened positions on the up-diagonal; the fourth argument on the down-diagonal.

$$
\begin{aligned}
&place :: (MonadPlus\ m) \Rightarrow Int \rightarrow [Int] \rightarrow [Int] \rightarrow [Int] \rightarrow m\ [Int] \\
&place\ 0\ rs\ d1\ d2 = return\ [] \\
&place\ i\ rs\ d1\ d2 = \mathbf{do}\ \ (q, rs') \leftarrow select\ rs \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ q1 = q - i \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ q2 = q + i \\
&\qquad\qquad\qquad\qquad guard\ (q1 \text{`}notElem\text{`} d1) \\
&\qquad\qquad\qquad\qquad guard\ (q2 \text{`}notElem\text{`} d2) \\
&\qquad\qquad\qquad\qquad qs \leftarrow place\ (i - 1)\ rs'\ (q1 : d1)\ (q2 : d2) \\
&\qquad\qquad\qquad\qquad return\ (q : qs)
\end{aligned}
$$

The auxiliary function *select* nondeterministically chooses an element from a list. The selected element and the residue are returned.

$$
\begin{aligned}
&select :: (MonadPlus\ m) \Rightarrow [a] \rightarrow m\ (a, [a]) \\
&select\ [] = zero \\
&select\ (a : x) = return\ (a, x) \\
&\qquad\qquad\quad +\!\!+\ \mathbf{do}\ \ (b, x') \leftarrow select\ x \\
&\qquad\qquad\qquad\qquad return\ (b, a : x')
\end{aligned}
$$

The following call determines the number of solutions to the 8 queens problem.

$$length\ (run\ (queens\ 8 :: BacktrM\ [Int]) :: []\ [Int])$$

Remark
Let us finally point out that some features of logic languages are difficult to recast in the monadic framework. The concept of a *logical variable* certainly counts to the more difficult ones. Logic programmers use them for programming incomplete data structures or for solving constraint problems. Interestingly, open data structures may be simulated through higher-order functions (Burton, 1989) — the difference list $[a_1, \cdots, a_n \mid x] - x$ corresponds to the λ-expression $\lambda x \rightarrow a_1 : \cdots : a_n : x$. Constraint problems like the map coloring problem (Sterling & Shapiro, 1986, Page 212) seem to require significantly more programming efforts.
End of remark

2.2 Parsing

Recursive descent parsers can be conveniently expressed in the monadic framework. Building upon the auxiliary functions introduced in Section 1.2 we present a parser for Prolog terms.

A Prolog term is either a variable or a function symbol applied to zero or more arguments. Terms are represented using the data type *Term*.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Term} = \mathit{Var}\ \mathit{String}\\
&\qquad\qquad\quad \mid \mathit{Fun}\ \mathit{String}\ [\mathit{Term}]
\end{aligned}
$$

The concrete representation of a term is given by the following cf grammar.

$$
\begin{array}{rcl}
\mathit{term} & \to & \mathit{var}\\
 & \mid & \mathit{fun}\ [\ \mathit{args}\ ]\\
\mathit{args} & \to & \texttt{(}\ \mathit{term}\langle \texttt{,} \rangle\ \texttt{)}\\
\mathit{var} & \to & \mathit{upper}\ \mathit{alpha}^{*}\\
\mathit{fun} & \to & \mathit{lower}\ \mathit{alpha}^{*}
\end{array}
$$

The corresponding monadic parser essentially augments this grammar with semantic actions. More precisely it constructs the internal representation of the parsed term returning an element of *Term*.

$$
\begin{aligned}
&\mathit{term} :: (\mathit{MonadPlus}\ m, \mathit{MonadParser}\ \mathit{Char}\ m) \Rightarrow m\ \mathit{Term}\\
&\mathit{term} = \mathbf{do}\ v \leftarrow \mathit{var}\\
&\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Var}\ v)\\
&\qquad\qquad\quad {+\!\!+}\,\mathbf{do}\ f \leftarrow \mathit{fun}\\
&\qquad\qquad\qquad\quad \mathit{as} \leftarrow \mathit{opt}\ \mathit{args}\ []\\
&\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Fun}\ f\ \mathit{as})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{args} :: (\mathit{MonadPlus}\ m, \mathit{MonadParser}\ \mathit{Char}\ m) \Rightarrow m\ [\mathit{Term}]\\
&\mathit{args} = \mathbf{do}\ \mathit{lit}\ \texttt{'('}\\
&\qquad\qquad\qquad\quad \mathit{as} \leftarrow \mathit{sepBy}\ \mathit{term}\ (\mathit{lit}\ \texttt{','})\\
&\qquad\qquad\qquad\quad \mathit{lit}\ \texttt{')'}\\
&\qquad\qquad\qquad\quad \mathit{return}\ \mathit{as}
\end{aligned}
$$

It is interesting to see that we can also handle lexical matters quite easily: Usually one agrees upon that lexical units such as identifiers or brackets may be separated by white space. The ‘parser transformer’ *lexical* accomplishes this task.

$$
\begin{aligned}
&\mathit{lexical} :: (\mathit{MonadPlus}\ m, \mathit{MonadParser}\ \mathit{Char}\ m) \Rightarrow m\ a \to m\ a\\
&\mathit{lexical}\ p = \mathbf{do}\ a \leftarrow p;\ \mathit{many}\ \mathit{white};\ \mathit{return}\ a
\end{aligned}
$$

Each function, which parses a lexical unit, is wrapped up in a call to *lexical*.

$$
\begin{aligned}
&\mathit{var}, \mathit{fun} :: (\mathit{MonadPlus}\ m, \mathit{MonadParser}\ \mathit{Char}\ m) \Rightarrow m\ \mathit{String}\\
&\mathit{var} = \mathit{lexical}\ (\mathbf{do}\ a \leftarrow \mathit{upper};\ x \leftarrow \mathit{many}\ \mathit{alpha};\ \mathit{return}\ (a : x))\\
&\mathit{fun} = \mathit{lexical}\ (\mathbf{do}\ a \leftarrow \mathit{lower};\ x \leftarrow \mathit{many}\ \mathit{alpha};\ \mathit{return}\ (a : x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{lit} :: (\mathit{MonadPlus}\ m, \mathit{MonadParser}\ \mathit{Char}\ m) \Rightarrow \mathit{Char} \to m\ \mathit{Char}\\
&\mathit{lit}\ a = \mathit{lexical}\ (\mathit{item} \rhd (== a))
\end{aligned}
$$

The basic parsers *lower*, *upper*, *alpha*, and *white* each process a single character.

$$\begin{aligned}
&\mathit{lower}, \mathit{upper}, \mathit{alpha}, \mathit{white} :: (\mathit{MonadParser}\ \mathit{Char}\ m) \Rightarrow m\ \mathit{Char}\\
&\mathit{lower} = \mathit{item} \rhd \mathit{isLower}\\
&\mathit{upper} = \mathit{item} \rhd \mathit{isUpper}\\
&\mathit{alpha} = \mathit{item} \rhd \mathit{isAlpha}\\
&\mathit{white} = \mathit{item} \rhd \mathit{isSpace}
\end{aligned}$$

Assume that we have an instance of *MonadParser* at hand, say, *ParserM*. Building upon this monad the function *parse* converts the concrete representation of a term into its internal representation.

$$\begin{aligned}
&\mathit{parse} :: \mathit{String} \to \mathit{Maybe}\ \mathit{Term}\\
&\mathit{parse}\ s = \mathit{run}\ (\mathit{accepts}\ (\mathbf{do}\ \ \mathit{many}\ \mathit{white};\ \mathit{term})\ s :: \mathit{ParserM}\ \mathit{Char}\ \mathit{Term})
\end{aligned}$$

We have hinted only at the basics of monadic or functional parsing; Hutton and Meijer (1996) give a more in-depth treatment of the topic.

3 Constructing monads

This section is concerned with the construction of monads which support the operations used in the previous applications.

3.1 Monad transformers

Each of the class definitions listed in Section 1.1 defines a certain 'feature' that a monad might support. Note that this list is far from being exhaustive: In the sections to follow we will introduce four more features; many others are conceivable. One can imagine that it is a tedious and error-prone task to program a monad from the scratch that supports all these features. We employ a more modular approach instead: For each class we define a *monad transformer* that adds the respective feature to a given monad. Old features are preserved by a process called *lifting*.

The idea of a monad transformer is again due to Moggi (1990): Each of the monad transformers defined in this paper already appears in *loc. cit.* albeit in an abstract form. Moggi's approach was later extended by Liang, Hudak, and Jones (1995) who are especially concerned with the problem of lifting operations through monad transformers. In the sequel we employ a mild variant of their work.

A monad transformer is a type constructor τ which takes a monad m to a monad $\tau\ m$.

$$\begin{aligned}
&\mathbf{class}\ (\mathit{Monad}\ m, \mathit{Monad}\ (\tau\ m)) \Rightarrow \mathit{MonadT}\ \tau\ m\mathbf{where}\\
&\quad \mathit{up} :: m\ a \to \tau\ m\ a\\
&\quad \mathit{down} :: \tau\ m\ a \to m\ a
\end{aligned}$$

The member function *up* embeds a computation in m into the monad $\tau\ m$. The function *down* goes the way back thus forgetting the additional structure $\tau\ m$ is assumed to provide. It is near at hand to require both *up* and *down* to be monad morphisms. While this certainly makes sense for *up*, it is often only possible to construct a *premonad* morphism from $\tau\ m$ to m (see Section 3.2).

Now, *up* allows us to lift *zero* through an *arbitrary* monad transformer. To lift the operation *update* which has a slightly more general type we can, in fact, use the morphism condition (15).

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{MonadZero}\ m, \mathit{MonadT}\ \tau\ m) \Rightarrow \mathit{MonadZero}\ (\tau\ m)\textbf{where} \\
\quad \mathit{zero} = \mathit{up}\ \mathit{zero} \\
\textbf{instance}\ (\mathit{MonadState}\ \mathit{st}\ m, \mathit{MonadT}\ \tau\ m) \Rightarrow \mathit{MonadState}\ \mathit{st}\ (\tau\ m)\textbf{where} \\
\quad \mathit{update} = \mathit{up} \circ \mathit{update}
\end{array}
$$

The operation $(+\!\!+)$, however, must be treated on a case by case basis, since the relevant equation, (14), does not provide a constructive definition. Espinosa (1995) proposes a slightly more general approach based on higher-order monads which allows to lift $(+\!\!+)$ uniformly. However, the approach fails for more complex operations like *handle* (see Section 7.1) as well. Furthermore, there is no way to turn the state monad transformer to be defined in Section 3.2 into a higher-order monad. Consequently we refrain from introducing the additional machinery.

Turning to the topic of running a monad transformer τ, respectively an application $\tau\ m$, we note that *down* already allows us to move from $\tau\ m\ a$ to $m\ a$. We will see that monads combining several features are typically built using a *chain* of monad transformers. To run such a tower of monads in one go we apply *run* transitively to the computation in m.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{MonadT}\ \tau\ m, \mathit{Run}\ m\ n) \Rightarrow \mathit{Run}\ (\tau\ m)\ n\textbf{where} \\
\quad \mathit{run} = \mathit{run} \circ \mathit{down}
\end{array}
$$

3.2 State monad transformer

We have seen in Section 1.1 that a state monad with *zero* makes a parser monad. The easiest way to build such a monad is via a *state monad transformer*.[§]

$$
\textbf{type}\ \mathit{StateT}\ \mathit{st}\ m\ a = \mathit{st} \to m\ (\mathit{st}, a)
$$

Applying the state monad transformer to a monad m yields a computation of type $\mathit{st} \to m\ (\mathit{st}, a)$: The current state is passed as an argument, the updated state and the computed value are returned as results.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{StateT}\ \mathit{st}\ m)\textbf{where} \\
\quad m \gg\!= k = \lambda s \to m\ s \gg\!= \lambda(s', a) \to k\ a\ s' \\
\quad \mathit{return}\ a = \lambda s \to \mathit{return}\ (s, a) \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadState}\ \mathit{st}\ (\mathit{StateT}\ \mathit{st}\ m)\textbf{where} \\
\quad \mathit{update}\ \mathit{st} = \lambda s \to \mathit{return}\ (\mathit{st}\ s, s)
\end{array}
$$

[§] Note that we have cheated a bit in the definition of *StateT*. The instance declarations which follow require *StateT* to be a data type rather than a type synonym. Since the introduction of a superfluous data constructor affects the readability of the code, we use type synonyms instead.

Lifting a computation c into $StateT\ st\ m$ works as follows: ignoring the current state c is run yielding a which is then paired with the current state and returned. Projecting a computation in $StateT\ st\ m$ onto m is equally simple: We supply an undefined initial state and discard the final state. If one is interested in the final state it must be returned explicitly: **do** $\cdots; s \leftarrow update\ id; return\ (s, \cdots)$.

$$
\begin{aligned}
&\mathbf{instance}\ (Monad\ m) \Rightarrow MonadT\ (StateT\ st)\ m\,\mathbf{where}\\
&\quad up\ m = \lambda s \rightarrow m \gg\!= \lambda a \rightarrow return\ (s, a)\\
&\quad down\ m = m\ \bot \gg\!= \lambda(\_, a) \rightarrow return\ a
\end{aligned}
$$

The reader may convince herself that *up* is a monad morphism and *down* a premonad morphism. Lifting $(+\!\!+)$ is straightforward: The state s is passed to each of the alternatives. Note that the instance declaration for *zero* is redundant; it is listed only for ease of reference.

$$
\begin{aligned}
&\mathbf{instance}\ (MonadZero\ m) \Rightarrow MonadZero\ (StateT\ st\ m)\mathbf{where}\\
&\quad zero = \lambda s \rightarrow zero\\
&\mathbf{instance}\ (MonadPlus\ m) \Rightarrow MonadPlus\ (StateT\ st\ m)\mathbf{where}\\
&\quad m +\!\!+ n = \lambda s \rightarrow m\ s +\!\!+ n\ s
\end{aligned}
$$

3.3 Maybe monad

The *Maybe* monad implements a restricted form of backtracking: Each computation either fails or succeeds exactly once. On the one hand this has the unfortunate consequence that once a computation succeeds further alternatives are discarded.

(16) $return\ a +\!\!+ m \quad = \quad return\ a$

On the other hand *Maybe* is entirely sufficient for the implementation of unambiguous parsers, eg for *term* and *args*. The data type is given by the following definition; the names are due to Spivey (1989):

$$\mathbf{data}\ Maybe\ a = Nothing \mid Just\ a$$

$$
\begin{aligned}
&maybe :: b \rightarrow (a \rightarrow b) \rightarrow Maybe\ a \rightarrow b\\
&maybe\ n\ f\ Nothing = n\\
&maybe\ n\ f\ (Just\ x) = f\ x
\end{aligned}
$$

The function *maybe* is the fold-functional for *Maybe*, see (Sheard & Fegaras, 1993).

$$
\begin{aligned}
&\mathbf{instance}\ Monad\ Maybe\mathbf{where}\\
&\quad m \gg\!= k = maybe\ zero\ k\ m\\
&\quad return = Just\\
&\mathbf{instance}\ MonadZero\ Maybe\mathbf{where}\\
&\quad zero = Nothing\\
&\mathbf{instance}\ MonadPlus\ Maybe\mathbf{where}\\
&\quad m +\!\!+ n = maybe\ n\ Just\ m
\end{aligned}
$$

We are now able to construct a parser monad by applying *StateT* [*tok*] to *Maybe*. As indicated above *Maybe* is to weak to be used for the pure backtracking examples.

$$\textbf{type}\ BacktrM = Maybe \quad \text{-- does not work}$$
$$\textbf{type}\ ParserM\ tok = StateT\ [tok]\ Maybe$$

The monad *Maybe* satisfies all the relevant laws listed in Section 1.1 with the notable exception of (9): Taking $m = Just\ False$, $n = Just\ True$, and $k = guard$ the lhs yields $return\ ()$ while the rhs evaluates to *zero*. To put it more abstractly property (16) is incompatible with leftward distributivity.

Let us finally note that it is quite easy to turn *Maybe* into a monad transformer: $MaybeT\ m\ a = m\ (Maybe\ a)$. However, we do not need this extra generality.

3.4 List monad

The most prominent example for a monad supporting full backtracking is the list monad.

$$\textbf{data}\ [a] = [\,]\ |\ a : [a]$$

$$list :: b \to (a \to b \to b) \to [a] \to b$$
$$list\ e\ (*)\ [\,] = e$$
$$list\ e\ (*)\ (a : x) = a * list\ e\ (*)\ x$$

The cognoscenti will certainly notice that *list* amounts to the fold functional for lists (also known as *foldr*). It proves very useful for defining the monad operations.

$$\begin{array}{l} \textbf{instance}\ Monad\ [\,]\ \textbf{where} \\ \quad m \ggg\!= k = list\ zero\ (\lambda a\ n \to k\ a +\!\!\!+\ n)\ m \\ \quad return\ x = [x] \\ \textbf{instance}\ MonadZero\ [\,]\ \textbf{where} \\ \quad zero = [\,] \\ \textbf{instance}\ MonadPlus\ [\,]\ \textbf{where} \\ \quad m +\!\!\!+\ n = list\ n\ (:)\ m \end{array}$$

The list monad supports full backtracking and may be used to implement nondeterministic parsers.

$$\textbf{type}\ BacktrM = [\,]$$
$$\textbf{type}\ ParserM\ tok = StateT\ [tok]\ [\,]$$

The monad [] satisfies equations (1) – (9). Right distributivity fails since $(+\!\!\!+)$ is not commutative.

Why don't we generalize the construction above to a monad transformer using $ListT\ m\ a = m\ [a]$? It appears that lists compose only with so-called commutative monads, see (Jones & Duponcheel, 1993). For that reason we will use lists only as a base monad.

4 Efficient backtracking monads

Lists are ubiquitous in functional programming. Thus it comes as no surprise that they also serve as a prominent example for monads. However, the list monad suffers at least from three problems: (1) It relies in an essential way on lazy evaluation. Thus we cannot use this technique in a strict language like Standard ML. (2) It is inefficient: Both $(\gg=)$ and $(+\!\!+)$ are sensitive to the number of successes of their first argument. (3) There is no obvious way to turn it into a monad transformer. In the following we will gradually develop an alternative to the list monad which remedies these shortcomings.

4.1 Endomorphism monad transformer

Let us first make the efficiency problem more explicit. Turning back to the Biblical family database of Section 2.1 assume that we have constructed an index, say, a binary tree for the unary relation *person*. The index allows us to check quickly whether a given string denotes a person.

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

$$\begin{aligned}&\mathit{persons} :: \mathit{Tree}\ \mathit{String}\\ &\mathit{persons} = \mathit{Node}\ \cdots\ \texttt{"milcah"}\ (\mathit{Node}\ \cdots\ \texttt{"terach"}\ \cdots)\end{aligned}$$

To avoid redundancies we further define the original relation in terms of *persons*.

$$\begin{aligned}&\mathit{labels} :: (\mathit{MonadPlus}\ m) \Rightarrow \mathit{Tree}\ a \to m\ a\\ &\mathit{labels}\ \mathit{Empty} = \mathit{zero}\\ &\mathit{labels}\ (\mathit{Node}\ \ell\ a\ r) = \mathit{labels}\ \ell +\!\!+ \mathit{return}\ a +\!\!+ \mathit{labels}\ r\end{aligned}$$

$$\begin{aligned}&\mathit{person} :: (\mathit{MonadPlus}\ m) \Rightarrow m\ \mathit{String}\\ &\mathit{person} = \mathit{labels}\ \mathit{persons}\end{aligned}$$

The function *labels* enumerates the labels of a tree. Now, if we use the list monad for m the cost of computing *labels* t is in the worst case quadratic to the size of t. This bad behaviour is caused by $(+\!\!+)$ whose running time is proportional to the size of its first argument. The standard solution to this problem is to 'functionalize' *labels* — the additional parameter is sometimes called *accumulator*.

$$\begin{aligned}&\mathit{labels}' :: (\mathit{MonadPlus}\ m) \Rightarrow \mathit{Tree}\ a \to m\ a\\ &\mathit{labels}'\ t = \mathit{labelsPlus}\ t\ \mathit{zero}\end{aligned}$$

$$\begin{aligned}&\mathit{labelsPlus} :: (\mathit{MonadPlus}\ m) \Rightarrow \mathit{Tree}\ a \to (m\ a \to m\ a)\\ &\mathit{labelsPlus}\ \mathit{Empty} = \mathit{id}\\ &\mathit{labelsPlus}\ (\mathit{Node}\ \ell\ a\ r) = \mathit{labelsPlus}\ \ell \circ (\lambda f \to \mathit{return}\ a +\!\!+ f) \circ \mathit{labelsPlus}\ r\end{aligned}$$

Note that the same technique is applied in Haskell's *Show* class to guarantee linear complexity of the *show* functions. The following equation relates *labelsPlus* to *labels*.

$$(17)\quad \mathit{labelsPlus}\ t\ f \quad = \quad \mathit{labels}\ t +\!\!+ f$$

This trick works fine. However, it is annoying that we have to rewrite a piece of code to get a better runtime performance. A better alternative is ready at hand: We wrap this trick up into a monad transformer!

$$\textbf{type}\ \mathit{EndoT}\ m\ a = m\ a \to m\ a$$

Inspecting the code for *labelsPlus* we find that we already know how to define *return*, *zero*, $(+\!\!+)$, and *down*. It remains to implement $(\gg\!=)$ and *up*. To see how a definition of bind might look like let us first invent a suitable correctness criterium. Combining (17) with the definition of *labels´* we get:

$$(18)\quad m\,f \quad = \quad m\ \mathit{zero} +\!\!+ f$$

Thus the work is reduced to finding a definition for $(m \gg\!= k)\ \mathit{zero}$ which is solved by pushing *zero* inwards: $m\ \mathit{zero} \gg\!= \lambda a \to k\ a\ \mathit{zero}$.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{MonadPlus}\ m) \Rightarrow \mathit{Monad}\ (\mathit{EndoT}\ m)\textbf{where} \\
\quad m \gg\!= k = \lambda f \to (m\ \mathit{zero} \gg\!= \lambda a \to k\ a\ \mathit{zero}) +\!\!+ f \\
\quad \mathit{return}\ a = \lambda f \to \mathit{return}\ a +\!\!+ f \\
\textbf{instance}\ (\mathit{MonadPlus}\ m) \Rightarrow \mathit{MonadT}\ \mathit{EndoT}\ m\textbf{where} \\
\quad \mathit{up}\ m = \lambda f \to m +\!\!+ f \\
\quad \mathit{down}\ m = m\ \mathit{zero} \\
\textbf{instance}\ \mathit{MonadZero}\ (\mathit{EndoT}\ m)\textbf{where} \\
\quad \mathit{zero} = \mathit{id} \\
\textbf{instance}\ \mathit{MonadPlus}\ (\mathit{EndoT}\ m)\textbf{where} \\
\quad m +\!\!+ n = m \circ n
\end{array}
$$

Note again that the instance declaration for *MonadZero* is redundant since it follows from the generic declaration given in Section 3.1: $\mathit{zero} = \mathit{up}\ \mathit{zero} = \lambda f \to \mathit{zero} +\!\!+ f = \mathit{id}$. Now, using *EndoT* [] in place of [] the original definition of *labels* runs in linear time. The approach taken is, in fact, very appealing: Instead of rewriting a piece of code we simply supply alternative instances for the overloaded operations.

We devote the rest of this section to the formal justification of this approach: We show that *EndoT m* does, in fact, form a monad and that *EndoT m* is isomorphic to *m*. The correctness criterium (18) plays, of course, a central rôle in this undertaking.

Lemma 1 Let m be a monad with *zero* and $(+\!\!+)$ which satisfies (6) – (8) and let c be a computation in *EndoT m* built up from *return*, *up*, and *zero* using the combinators $(\gg\!=)$ and $(+\!\!+)$. Then c satisfies equation (18).

It should be clear that we cannot expect arbitrary elements of $m\ a \to m\ a$ to satisfy the correctness criterion. Consider, for example, $\lambda f \to \mathit{return}\ a$ which discards the 'failure continuation' f. Thus, it is a good idea to turn $\mathit{EndoT}\ m\ a$ into an abstract data type. In the sequel we take $\mathit{EndoT}\ m\ a$ as the set of all functions $m\ a \to m\ a$ which satisfy equation (18).

Theorem 1 Let m be a monad with *zero* and $(+\!\!+)$ which satisfies (6) – (8). Then *EndoT m* forms a monad with *zero* and $(+\!\!+)$.

We have introduced *EndoT* solely for reasons of efficiency. The following theorem shows that *EndoT* does not add new structure to the base monad m. Thus, we may safely replace a backtracking monad m by *EndoT* m.

Theorem 2 Let m be a monad with *zero* and $(+\!\!+)$ which satisfies (6) – (8). Then *up* is a monad isomorphism of m onto *EndoT* m with its inverse given by *down*.

4.2 Continuation monad transformer

Did we succeed in defining an efficient backtracking monad? Not quite, *EndoT* m works fine as long as the computation does not involve $(\ggg\!=)$: Since bind still resorts to $(+\!\!+)$, we have only meliorated the efficiency problem.

Recall that *EndoT* m does not resort to the operations of m in defining *zero* and $(+\!\!+)$. The question naturally arises whether we can conceive a monad transformer which has the same property with respect to *return* and $(\ggg\!=)$? As unlikely as this may sound there is an off the shelf solution at hand: the continuation monad transformer. The approach is completely analogous to the one of the preceding section where we have replaced a computation m by a computation m' such that $m'\,f = m +\!\!+ f$. Accordingly, we now replace a computation m by a computation m' such that $m'\ \kappa = m \ggg\!= \kappa$. Since m' takes the (success) continuation as a parameter m' is said to be expressed in *continuation-passing style* (CPS). Now, what do we know about the type of m'? Recall that $(\ggg\!=)$ possesses the type $\forall a.\forall b.m\ a \to (a \to m\ b) \to m\ b$ which is equivalent to $\forall a.m\ a \to \forall b.(a \to m\ b) \to m\ b$. Since m' equals $(m\ggg\!=)$ we may assign m' the type $\forall b.(a \to m\ b) \to m\ b$. The monad definition consequently involves second-order types.¶

$$
\begin{array}{l}
\textbf{type}\ \mathit{ContT}\ m\ a = \forall ans.(a \to m\ ans) \to m\ ans \\
\ \textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{ContT}\ m)\textbf{where} \\
\qquad m \ggg\!= k = \lambda\kappa \to m\ (\lambda a \to k\ a\ \kappa) \\
\qquad \mathit{return}\ a = \lambda\kappa \to \kappa\ a
\end{array}
$$

Note that no use is made of m's monad structure. Furthermore both *return* and $(\ggg\!=)$ run in constant time. The monad structure of m is only required for lifting operations into *ContT* m and for projecting computations in *ContT* m onto m.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadT}\ (\mathit{ContT})\ m\textbf{where} \\
\qquad \mathit{up}\ m = \lambda\kappa \to m \ggg\!= \kappa \\
\qquad \mathit{down}\ m = m\ \mathit{return} \\
\ \textbf{instance}\ (\mathit{MonadZero}\ m) \Rightarrow \mathit{MonadZero}\ (\mathit{ContT}\ m)\textbf{where} \\
\qquad \mathit{zero} = \lambda\kappa \to \mathit{zero} \\
\ \textbf{instance}\ (\mathit{MonadPlus}\ m) \Rightarrow \mathit{MonadPlus}\ (\mathit{ContT}\ m)\textbf{where} \\
\qquad m +\!\!+ n = \lambda\kappa \to m\ \kappa +\!\!+ n\ \kappa
\end{array}
$$

¶ At the time of writing only the Haskell B compiler, release 0.9999.3, supports local universal quantification. Its typechecker is based on work by Didier (1994).

Combining *ContT* with *EndoT* we get efficient backtracking and parser monads.

$$\textbf{type}\ BacktrM\ a = ContT\ (EndoT\ [\,])\ a$$
$$\textbf{type}\ ParserM\ tok\ a = ContT\ (StateT\ [tok]\ (EndoT\ [\,]))\ a$$

In the latter type definition we have enclosed the state monad transformer in calls to *ContT* and *EndoT*. Two alternative orderings are possible but they are, in fact, less efficient.

As with *EndoT* we have introduced *ContT* solely to improve runtime performance. In the remainder we demonstrate that *ContT m* is, in fact, isomorphic to *m*. The following lemma is the counterpart to Lemma 1.

Lemma 2 Let m be a monad with *zero* and $(+\!\!+)$ which satisfies (4) and (9) and let c be a computation in *ContT m* built up from *return*, *up*, and *zero* using the combinators $(\gg\!=)$ and $(+\!\!+)$. Then c satisfies

(19) $c\ \kappa \quad = \quad c\ return \gg\!= \kappa.$

Remark
The proof given in the appendix shows the proposition from the scratch using simple algebraic manipulations. The reader may wonder whether a more elegant proof based on parametricity (Wadler, 1989) is possible. It is well-known that for the second-order type $\forall ans.(a \to ans) \to ans$ a simplified variant of (19), ie $c\ \kappa = \kappa\ (c\ id)$, follows directly from the parametricity theorem. However, *ContT* involves the higher-order type variable m and a generalization of the parametricity theorem to higher-order types is not known to the author. Furthermore note that it is necessary that m satisfies property (9). In *ContT Maybe*, for example, equation (19) fails: Take $c = return\ False +\!\!+ return\ True$ and $\kappa = guard$.
End of remark

As before we reserve *ContT m a* for the set of all functions $\forall ans.(a \to m\ ans) \to m\ ans$ which satisfy equation (19).

Theorem 3 Let m be a monad with *zero* and $(+\!\!+)$ which satisfies (4) and (9). The map *up* is a monad isomorphism of m onto *ContT m* with its inverse given by *down*.

Remark
CPS is usually employed to model sophisticated control operations like *callcc* or *shift* and *reset* (Danvy & Filinski, 1990). Theorem 3 clearly implies that they cannot be defined within *ContT m*. If we are interested in the increased power of expression we are forced to generalize *ContT* by parameterizing it with the answer type.

$$\textbf{type}\ PContT\ ans\ m\ a = (a \to m\ ans) \to m\ ans$$

The instance declaration for *Monad* remains unchanged. We are now able, for instance, to implement the *shift* operation defined by Danvy and Filinski (1990).

$$shift :: (Monad\ m) \Rightarrow ((a \to PContT\ ans'\ m\ ans) \to PContT\ ans\ m\ ans) \to PContT\ ans\ m\ a$$
$$shift\ h = \lambda\kappa \to h\ (\lambda a \to \lambda\kappa' \to \kappa\ a \gg\!= \kappa')\ return$$

End of remark

4.3 A different perspective

It is well-known that backtracking can be implemented using continuation passing style (Mellish & Hardy, 1984). In the sequel we give a short account of the traditional approach and examine the relationship to the one given in the previous sections.

To implement backtracking a program is first converted to CPS the continuation argument specifying what to do in case of success. Ignoring the success continuation corresponds to failure. Alternatives are tried successively: $m +\!\!+ n$ reads as 'try m first and then try n'. Thus, alternation is implemented by sequencing! This technique may be easily expressed in our framework:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadZero}\ (\mathit{ContT}\ m) \textbf{where} \\ \quad \mathit{zero} = \lambda\kappa \to \mathit{return}\ \bot \\ \textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadPlus}\ (\mathit{ContT}\ m) \textbf{where} \\ \quad m +\!\!+ n = \lambda\kappa \to m\ \kappa \gg n\ \kappa\end{array}$$

In view of the previous sections it is probably surprising that $(+\!\!+)$ is defined in terms of $(\gg)$. However, that is exactly the approach taken in an imperative setting, see, for example, Wirth's solution of the n queens problem (Wirth, 1979). The reader may wonder whether the laws concerning *zero* and $(+\!\!+)$ are still satisfied. This is indeed the case: Equations (4), (6), (8), and (9) go through without problems; equation (7) holds if every computation in m returns $\bot$. From the definitions above it is easy to see that this condition holds for *zero* and $(+\!\!+)$ (we tacitly assume that these operations are the only ones available). The latter point makes clear that we employ m's sequential structure only: Values play no rôle in a failing computation.

Using *IO* as a base monad we can print the solutions of the 8 queens problem as follows:

$$(\mathit{queens}\ 8 :: \mathit{ContT}\ \mathit{IO}\ [\mathit{Int}])\ \mathit{print}.$$

Note that the initial continuation, *print*, specifies what to do with a generated solution. This contrasts to the previous section where we supplied *return* to run an element of *ContT*. If we used *return* here as well we would merely obtain the value of the rightmost computation branch which is a failure branch, ie $(\mathit{queens}\ 8 :: \mathit{ContT}\ \mathit{IO}\ [\mathit{Int}])\ \mathit{return}$ yields $\mathit{return}\ \bot$.

Of course, we are usually interested in getting a list of all solutions. Let us discuss an imperative and a functional approach to this problem. A truly imperative way to construct the desired list is to use a writer monad instead of the *IO* monad.

$$\begin{array}{l}\textbf{type}\ \mathit{WriterM}\ o\ m\ a = (m\ o,\ a) \\ \textbf{instance}\ (\mathit{MonadPlus}\ m) \Rightarrow \mathit{Monad}\ (\mathit{WriterM}\ o\ m) \textbf{where} \\ \quad m \ggg k = \textbf{let}\ (x,\ a) = m;\ (y,\ b) = k\ a \textbf{in}\ (x +\!\!+ y,\ b) \\ \quad \mathit{return}\ a = (\mathit{zero},\ a) \\ \\ \mathit{write} :: (\mathit{MonadPlus}\ m) \Rightarrow o \to \mathit{WriterM}\ o\ m\ () \\ \mathit{write}\ o = (\mathit{return}\ o,\ ())\end{array}$$

The writer monad defined above is a bit unconventional: Normally, the output is required to to form a monoid (Jones & Duponcheel, 1993). To minimize concepts we use a monad with *zero* and $(+\!\!+)$ instead. This is not a severe restriction, however, since the most interesting monoids, $[a]$ and $[a] \to [a]$, form monads as well. To collect the solutions of the 8 queens problem we replace *IO* by *WriterM* [*Int*] [] and *print* by *write*.

$$fst\ ((queens\ 8 :: ContT\ (WriterM\ [Int]\ [])\ [Int])\ write)$$

For reasons of efficiency it is, of course, better to use *EndoT* [] as the base monad of *WriterM*.

We have noted above that the values computed in the base monad are never used. So why not dispense with them? Surprisingly, if we strip the unnecessary code off $ContT\ (WriterM\ a\ m)\ a$ boils down to $ContT\ m\ a$.

As Danvy and Filinski (1990) point out there is also a purely functional solution to the problem of collecting all solutions: For the base monad take yet another instance of the continuation monad transformer which in turn is based on the list monad. Actually, Danvy and Filinski do not use monads, but is not hard to recast their construction in monadic terms. *Technical remark:* Recall that the values computed in the base monad are not required. Consequently we must use the variant of *ContT*, which is parameterized with the answer type, for the base monad.

$$(queens\ 8 :: ContT\ (PContT\ [Int]\ [])\ [Int])\ (\lambda a \to \lambda\kappa \to a : \kappa\ \bot)\ (\lambda\_ \to [])$$

The initial inner continuation constructs the list of answers. The initial outer continuation terminates the list. But see, since the values computed in the base monad are of no interest $PContT\ ans\ m\ a = (a \to m\ ans) \to m\ ans$ boils down to $EndoT\ m\ ans = m\ ans \to m\ ans$. Furthermore note that the initial inner continuation amounts to *return* and the initial outer continuation to *zero*. So we obtain again $ContT\ (EndoT\ [])$ which may be considered as the 'essence of implementing backtracking by continuation passing'.

Remark
Danvy and Filinski (1990) show how to simulate backtracking using the control operators *shift* and *reset*. The computation *zero*, for instance, may be expressed as follows (*zero* is termed *fail* in *loc. cit.*).

$$\begin{array}{l}\mathbf{instance}\ (Monad\ m) \Rightarrow MonadZero\ (PContT\ ans\ m)\mathbf{where}\\ \quad zero = shift\ (\lambda c \to return\ \bot)\end{array}$$

The relation to the declaration given above is immediate: Unfolding the definition of *shift* yields the former.
End of remark

4.4 Backtracking monad transformer

The monad transformer $ContT \circ EndoT$ turns a backtracking monad into a more efficient one. Interestingly the dependence on the base monad is quite loose: Its operations are only used for implementing *up* and *down*. This can be seen by unfolding $ContT \circ EndoT$ into a single monad transformer which we will call *BacktrT*.

$$
\begin{array}{l}
\textbf{type}\ Cps\ a\ ans = (a \to ans) \to ans \\
\textbf{type}\ Endo\ ans = ans \to ans
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ BacktrT\ m\ a = \forall ans.Cps\ a\ (Endo\ (m\ ans)) \\
\textbf{instance}\ Monad\ (BacktrT\ m) \textbf{where} \\
\quad return\ a = \lambda\kappa \to \kappa\ a \\
\quad m \ggg k = \lambda\kappa \to m\ (\lambda a \to k\ a\ \kappa) \\
\textbf{instance}\ (Monad\ m) \Rightarrow MonadT\ (BacktrT)\ m \textbf{where} \\
\quad up\ m = \lambda\kappa\, f \to (m \ggg \lambda a \to \kappa\ a\ zero) +\!\!+ f \\
\quad down\ m = m\ (\lambda a\, f \to return\ a +\!\!+ f)\ zero \\
\textbf{instance}\ MonadZero\ (BacktrT\ m) \textbf{where} \\
\quad zero = \lambda\kappa \to id \\
\textbf{instance}\ MonadPlus\ (BacktrT\ m) \textbf{where} \\
\quad m +\!\!+ n = \lambda\kappa \to m\ \kappa \circ n\ \kappa
\end{array}
$$

Now, assume that the base monad is merely an instance of *Monad*. Is there a way of turning *BacktrT* into a monad transformer which respects the morphism conditions on *up* and *down*? The answer is in the affirmative and quite simple, too.

$$
\begin{array}{l}
\textbf{instance}\ (Monad\ m) \Rightarrow MonadT\ (BacktrT)\ m \textbf{where} \\
\quad up\ m = \lambda\kappa\, f \to m \ggg \lambda a \to \kappa\ a\ f \\
\quad down\ m = m\ (\lambda a\_ \to return\ a)\ (error\ \texttt{"no solution"})
\end{array}
$$

Note that *down* generates an error if the computation in *BacktrT m* fails. Furthermore only the first solution is returned to the base monad; further solutions are simply ignored. Nonetheless *down* is a premonad morphism and *up* a monad morphism. Furthermore we have succeeded in programming a backtracking transformer: *BacktrT* adds *zero* and $(+\!\!+)$ to an arbitrary monad.

Theorem 4 Let *m* be a monad, then *BacktrT m* forms a monad with *zero* and $(+\!\!+)$ which satisfies (4) and (6) – (9).

The backtracking monad transformer has many interesting applications: *BacktrT* (*StateT st m*) adds backtracking and a *global* state to *m*; *BacktrT IO* enables us to perform *IO* during a non-deterministic computation. The following example illustrates the use of the latter monad. The function *trace* implements a simple trace facility based on the 4 port procedure model of logic languages (Mellish & Clocksin, 1995): *trace m msg* behaves as m except that the user is informed when (1) m is called the first time, (2) m is exited upon success, (3) m fails, and (4) m is re-entered upon backtracking.

$$
\begin{array}{l}
\mathbf{class}\ (Monad\ m) \Rightarrow MonadIO\ m\,\mathbf{where} \\
\quad outStr :: String \to m\ () \\
\quad inFile :: FilePath \to m\ String \\
\mathbf{instance}\ MonadIO\ IO\,\mathbf{where} \\
\quad outStr = putStr \\
\quad inFile = readFile \\
\mathbf{instance}\ (MonadIO\ m, MonadT\ t\ m) \Rightarrow MonadIO\ (t\ m)\,\mathbf{where} \\
\quad outStr = up \circ outStr \\
\quad inFile = up \circ inFile
\end{array}
$$

$$
\begin{array}{l}
trace :: (MonadPlus\ m, MonadIO\ m) \Rightarrow m\ a \to String \to m\ a \\
trace\ m\ msg = \mathbf{do}\ \ (out\ \texttt{"call"} +\!\!+\ \mathbf{do}\ \ out\ \texttt{"fail"};\ zero) \\
\qquad\qquad\qquad\qquad a \leftarrow m \\
\qquad\qquad\qquad\qquad (out\ \texttt{"exit"} +\!\!+\ \mathbf{do}\ \ out\ \texttt{"redo"};\ zero) \\
\qquad\qquad\qquad\qquad return\ a \\
\quad \mathbf{where}\ out\ s = outStr\ (s +\!\!+ \texttt{": "} +\!\!+ msg +\!\!+ \texttt{"\textbackslash n"})
\end{array}
$$

Note that the — highly operational — definition of *trace* takes for granted that equation (5) does *not* hold, otherwise $out\ \texttt{"call"} +\!\!+\ \mathbf{do}\ \ out\ \texttt{"fail"};\ zero$ would simplify to $out\ \texttt{"call"}$. The following expression which traces the computation of *child*'s transitive closure

$$tchild\ a = trace\ (child\ a)\ (\texttt{"child "} +\!\!+ show\ a)$$

$$run\ ((\mathbf{do}\ \ transitiveClosure'\ tchild\ \texttt{"terach"};\ zero) :: BacktrT\ IO\ String) :: IO\ String$$

generates the following output:

```
call: child "terach"
exit: child "terach"
call: child "abraham"
exit: child "abraham"
call: child "isaac"
fail: child "isaac"
redo: child "abraham"
fail: child "abraham"
redo: child "terach"
exit: child "terach"
call: child "nachor"
fail: child "nachor"
redo: child "terach"
exit: child "terach"
call: child "haran"
exit: child "haran"
call: child "lot"
fail: child "lot"
redo: child "haran"
exit: child "haran"
call: child "milcah"
fail: child "milcah"
redo: child "haran"
exit: child "haran"
call: child "yiscah"
fail: child "yiscah"
redo: child "haran"
fail: child "haran"
redo: child "terach"
fail: child "terach"
Program error: no solution
```

Since the generation of the transitive closure is driven by the call to *zero* (this is called a failure driven loop) the computation terminates with an error message. It is instructive to replace *transitiveClosure´* by *transitiveClosure* to visualize the overhead involved in the latter definition.

Along the same lines a parser monad transformer can be defined.

$$\textbf{type}\ \mathit{ParserT}\ \mathit{tok}\ a = \forall \mathit{ans}.\mathit{Cps}\ a\ ([\mathit{tok}] \to \mathit{Endo}\ (m\ \mathit{ans}))$$

Note that *ParserT tok* is actually a bit more general than $\mathit{ContT}\ (\mathit{StateT}\ [\mathit{tok}]\ (\mathit{EndoT}\ m))\ a = \forall \mathit{ans}.\mathit{Cps}\ a\ ([\mathit{tok}] \to \mathit{Endo}\ (m\ ([\mathit{tok}], \mathit{ans})))$. We specify only the interesting instance declarations; the remaining ones correspond essentially to those of *BacktrT*.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{MonadState}\ [\mathit{tok}]\ (\mathit{ParserT}\ \mathit{tok}\ m)\textbf{where} \\
\quad \mathit{update}\ \mathit{st} = \lambda\kappa\ s \to \kappa\ s\ (\mathit{st}\ s) \\
\textbf{instance}\ \mathit{MonadParser}\ \mathit{tok}\ (\mathit{ParserT}\ \mathit{tok}\ m)\textbf{where} \\
\quad \mathit{item} = \lambda\kappa\ s \to \textbf{case}\ s\textbf{of}\ [\,] \to \mathit{id};\ a : x \to \kappa\ a\ x
\end{array}
$$

Using *ParserT IO* it is easy to define a parser which directly processes include or import directives. The following is an excerpt of a parser which handles C like preprocessor directives.

$$
\begin{array}{l}
\mathit{process}, \mathit{line} :: (\mathit{MonadPlus}\ m, \mathit{MonadParser}\ \mathit{Char}\ m, \mathit{MonadIO}\ m) \Rightarrow m\ \mathit{Defs} \\
\mathit{process} = \textbf{do}\ \ \mathit{xs} \leftarrow \mathit{sepBy}\ \mathit{line}\ (\mathit{lit}\ \texttt{'\textbackslash n'}) \\
\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{concat}\ \mathit{xs}) \\
\mathit{line} = \cdots \\
\qquad\qquad\quad +\!\!\!+ \textbf{do}\ \ \mathit{lits}\ \texttt{"\#include"} \\
\qquad\qquad\qquad\quad \mathit{path} \leftarrow \mathit{string} \\
\qquad\qquad\qquad\quad \mathit{cnts} \leftarrow \mathit{inFile}\ \mathit{path} \\
\qquad\qquad\qquad\quad \mathit{accepts}\ \mathit{process}\ \mathit{cnts}
\end{array}
$$

Recall that *accept* ‘pushes’ the current token list so that nested calls to *accept* work properly.

4.5 Benchmarks

The purpose of this section is twofold. First, we would like to show that continuation-based backtracking monads compare favourably to the list monad. Of course, it is not difficult to construct a program along the lines of *labels* (see Section 4.1) which exemplifies the improved asymptotic runtime complexity of *ContT* (*EndoT* []) or *BacktrT IO*. The point is that even for apparently uncritical applications like the n queens problem continuation-based backtracking monads are superior to [].

Second, we will compare the efficiency of Haskell’s simulation of backtracking to that of logic languages like Prolog or Mercury (Somogyi *et al.*, 1995) which offer non-determinism as a ‘language primitive’. To get a somewhat broader picture we also include C and Standard ML (Milner *et al.*, 1990) in the competition. Due to the multi-lingual nature of the competition we will consider only a single example: For a given n the number of solutions to the n queens problem must be calculated.

Let us first comment on the various implementations of the problem. The C version listed in Appendix B corresponds to the *ContT IO* instance of the *queens* program: We have unfolded the monadic operations and subsequently converted them to C language constructs (*select* $[1..n]$ becomes the driving `for` loop). The list operations are furthermore replaced by bit operations. Nonetheless the C implementation has a rather functional flavour: Apart from the counter increment it is side-effect free.

Two Prolog programs participate in the contest: The first is a transliteration of the Haskell program. The second also listed in Appendix B is due to Thom Frühwirth and employs unification instead of arithmetic operations to check for attacking queens.

The Mercury variant corresponds to the first Prolog version enriched with type, mode, and determinism declarations. Unfortunately it is not possible to adapt Thom Frühwirth's program due to the use of open data structures which the Mercury compiler does not support.

The Haskell programs are based with one exception (see below) on the program of Section 2.1. Since Haskell currently neither supports universally quantified types nor multiple parameter classes the author was forced to by-pass Haskell's class system which has probably a positive effect on the running times. For simulating backtracking we use three different monads: [] (list), *ContT* (*EndoT* []) (fun. CPS), and *ContT* (*ST s*) (imp. CPS). Note that the state transformer monad, *ST s*, is an extension of Haskell (Launchbury & Peyton Jones, 1994) which provides updatable references and arrays. An updatable reference is used in *ContT* (*ST s*) to implement the solution counter.

The Standard ML code is commensurate to the Haskell code. For the imperative variant we employ Standard ML's impure features, ie sequencing and references.

Table 2 lists the results of the benchmark for $n = 11, 12, 13$, and 14. The columns display the absolute CPU time (average of three runs) and the time relative to the fastest implementation. Note that the benchmarks were run during the night to reduce the effects of background loads. Turning to the Haskell timings we see that the CPS monads are 50% faster than the list monad. The overhead in the list monad can quite easily be made explicit: We simply replace the do syntax by list comprehensions[‖]. It is well-known that list comprehensions may be translated such that exactly one $(:)$ operation is performed for each element of the resulting list (Wadler, 1987). The effect on the running time is as expected: The respective variant is quite as fast as the variants based on CPS.

Remark

The efficient translation scheme described in (Wadler, 1987) utilizes, of course, the technique discussed in Section 4.1. In effect *EndoT m* is specialized for lists which allows a more direct and efficient definition of *return* and $(\ggg=)$. We omit the in-

[‖] The required changes are, in fact, minor which is not surprising since do expressions are inspired by monad comprehensions.

Table 2. *Benchmark results*

#queens	11		12		13		14	
#solutions	2680		14200		73712		365596	
			C (`gcc-2.7.2 -O`)					
	0.11[a]	1[b]	0.61	1	3.53	1	21.93	1
			Prolog (`eclipse-3.5.2`)					
standard	5.74	52.2	31.35	51.4	182.56	51.7	1139.60	52.0
Früwirth's	5.72	52.0	31.45	51.6	181.74	51.5	1137.15	51.9
			Mercury (`mercury-0.6 -O6`)					
standard	3.0	27.3	18.33	30.0	123.0	34.8	896.0	40.9
			Haskell (`ghc-0.29 -O`)					
list	6.62	60.2	36.91	60.5	218.16	61.8	1352.70[c]	61.7
list compr.	4.23	38.5	23.53	38.6	139.49	39.5	891.24[c]	40.6
fun. CPS	4.09	37.2	22.78	37.3	135.08[d]	38.3	855.40[d]	39.0
imp. CPS	4.16	37.8	23.23	38.1	137.46	38.9	868.22	39.6
bit set	2.04	18.5	10.71	17.6	60.05[d]	17.0	363.38[d]	16.6
list transf.	5.89	53.5	32.44	53.2	191.51	54.3	1182.07[c]	53.9
			Standard ML (`sml-1.09`)					
list	1205.74	10961.3	15512.62	25430.5				
fun. CPS	10.71	97.4	59.18	97.0	328.10	92.9	2043.02	93.2
imp. CPS	8.55	77.7	47.60	78.0	284.00	80.5	1745.09	79.6

[a] Seconds of user time on a (loaded) Sun Ultrasparc-1
[b] Time relative to `gcc-2.7.2 -O`
[c] Runtime option `-H32M`
[d] Runtime option `-K8M`

stance declarations for *MonadZero* and *MonadPlus* since they are identical to those of *EndoT m*.

$$
\begin{array}{l}
\textbf{type}\ LT\ a = [a] \to [a] \\
\textbf{instance}\ Monad\ LT\ \textbf{where} \\
\quad return\ a = \lambda f \to a : f \\
\quad m \ggg k = \lambda f \to list\ f\ (k)\ (m\ zero)
\end{array}
$$

The naîve translation scheme on the other hand employs the operations of the ordinary list monad.
End of remark

The Standard ML implementation nicely demonstrates that the list monad is unsuited for strict languages. [We are probably dwelling on the obvious. Albeit this fact is barely mentioned elsewhere.]

Among the different languages C is the clear winner. The Haskell implementation based on CPS is a factor of 40 slower. This is not surprising, however, since C employs bit fields while Haskell uses lists to represent the queens' positions. If we use bit fields in Haskell as well the factor reduces to 16. Surprisingly, Haskell respectively the Glasgow Haskell Compiler (Peyton Jones *et al.*, 1993) compares favourably with Prolog and Mercury. Due to the use of type, mode and determinism systems the latter language is known to generate very fast code (Somogyi *et al.*, 1995). The running times, however, are commensurate to that of Haskell. Even more surprisingly, Haskell lies in front of Standard ML of New Jersey (Appel & MacQueen, 1991). Traditionally, strict languages have a reputation of being more efficient than their non-strict counterparts. The author has no conclusive explanation for this opposite behaviour. However, it is certainly the case that the Glasgow Haskell Team has done an excellent job — Hartel (1994) relates the Glasgow Haskell Compiler to other systems. All in all we may conclude that backtracking based on monads appears to be a serious alternative to logic languages (but see the remark at the end of Section 2.1).

5 Adding control

Having succeeded in programming an efficient backtracking monad we now extend it with widely used features such as control operations, all solution functions and exception handling.

Often one ist interested in getting only a single solution: the operator *once* may be used in this case.

$$
\begin{array}{l}
\textbf{class}\ (\mathit{MonadPlus}\ m) \Rightarrow \mathit{MonadCtrl}\ m \textbf{where} \\
\quad \mathit{once} :: m\ a \to m\ a \\
\quad (?) :: m\ a \to m\ a \to m\ a \\
\quad m\ ?\ n = \mathit{once}\ (m + n)
\end{array}
$$

The member function *once* makes a computation unambiguous, ie *once m* succeeds at most once, whence its name. If the underlying monad is not commutative with respect to $(+)$ we expect *once* to yield the *first* solution.

The binary operator $(?)$ is a restricted variant of $(+)$: $m\ ?\ n$ yields one solutions of m, unless m fails, in which case it yields one solution of n. We expect *once* to satisfy, at least, the following identities; for reasons of space we omit the corresponding laws for $(?)$.

$$
\begin{array}{lrcl}
(20) & \mathit{once}\ \mathit{zero} & = & \mathit{zero} \\
(21) & \mathit{once}\ (\mathit{return}\ a + m) & = & \mathit{return}\ a \\
(22) & (\mathit{once}\ m \ggg k) + (\mathit{once}\ m \ggg \ell) & = & \mathit{once}\ m \ggg \lambda a \to k\ a + \ell\ a
\end{array}
$$

Equation (22) is really a restriction of (10): We have seen in Section 1.1 that (10) only holds if either $(+)$ is commutative or m is unambiguous. Using *once* we enforce the latter condition. Each of the two operations also induces a condition which a monad morphism is required to satisfy.

$$(23)\quad \eta\ (once\ m) \;=\; once\ (\eta\ m)$$
$$(24)\quad \eta\ (m \mathbin{?} n) \;=\; \eta\ m \mathbin{?} \eta\ n$$

If the default implementation of $(?)$ is used, (24) follows immediately from (23). Thus default methods reduce both implementation efforts and proof obligations.

The operator $(?)$ is commonly employed to define a more efficient variant of *many*. The naïve implementation of Kleene's star given in Section 1.2 yields not only the longest possible sequence but also every shorter sequence. The following variant remedies this problem.

```
many :: (MonadCtrl m) ⇒ m a → m [a]
many m = ms
  where ms = do a ← m
                x ← ms
                return (a : x)
             ? return []
```

The implementation of the *once* operator pôses little problems:

```
instance MonadCtrl Maybe where
  once = id
instance MonadCtrl [] where
  once = list zero (λa _ → return a)
```

Note that $(+\!\!+)$ and $(?)$ coincide in the *Maybe* monad. The most interesting instance declaration is the one for *BacktrT m*. It appears that it is possible to implement the *once* operator *independently* of the underlying base monad.

```
instance MonadCtrl (BacktrT m) where
  once m = λκ f → m (λa _ → κ a f) f
```

Recall that $\kappa :: a \to m\ ans \to m\ ans$ is the *success continuation* and $f :: m\ ans$ the *failure continuation*. Now, if m succeeds it calls the success continuation passing it the computed value and possible alternatives. We simply discard these 'choice points' and install the original failure continuation.

Lifting *once* through the various monad transformers is equally straightforward. For *EndoT* and *ContT* we have simply unfolded the equation $once = up \circ once \circ down$ which is based on the isomorphism results.

```
instance (MonadCtrl m) ⇒ MonadCtrl (StateT st m) where
  once m = λs → once (m s)
instance (MonadCtrl m) ⇒ MonadCtrl (EndoT m) where
  once m = λf → once (m zero) ++ f
instance (MonadCtrl m) ⇒ MonadCtrl (ContT m) where
  once m = λκ → once (m return) ≫= κ
```

6 Adding introspection

Assume that we want to write a function which counts how often a non-deterministic computation succeeds. The only way to realize such a function is via *run*:

$$\begin{aligned}&\mathit{count} :: (\mathit{Monad}\ m, \mathit{Run}\ m\ []) \Rightarrow m\ a \to m\ \mathit{Int}\\ &\mathit{count}\ m = \mathit{return}\ (\mathit{length}\ (\mathit{run}\ m))\end{aligned}$$

This approach, however, suffers from a serious drawback: It is not modular. If we execute m all computational features m might involve, say, state or exception handling are decoupled from the current flow of control, ie m is executed in isolation.

The crux is that *MonadPlus* offers per se no introspective features; more precisely there is no way to determine the solutions of a computation *within* a backtracking monad. The following extension of *MonadPlus* remedies this shortcoming.

$$\begin{aligned}&\mathbf{class}\ (\mathit{MonadPlus}\ m) \Rightarrow \mathit{MonadSoln}\ m\,\mathbf{where}\\ &\quad \mathit{solutions} :: m\ a \to m\ [a]\end{aligned}$$

The member function *solutions m* succeeds exactly once returning the list of *m*'s successes in order of their generation. For stating the numerous laws *solutions* involves the following function proves useful.

$$\begin{aligned}&(\oplus) :: (\mathit{Monad}\ m) \Rightarrow m\ [a] \to m\ [a] \to m\ [a]\\ &m \oplus n = \mathbf{do}\ \ x \leftarrow m;\ y \leftarrow n;\ \mathit{return}\ (x +\!\!+ y)\end{aligned}$$

The cognoscenti will notice the similarity of the equations below with the definition of the list monad transformer: The latter are obtained from the former by dropping the calls to *solutions*.

$$\begin{aligned}
&(25)\quad \mathit{solutions}\ (\mathit{return}\ a) &&= \mathit{return}\ [a]\\
&(26)\quad \mathit{solutions}\ (m \ggg\!= k) &&= \mathit{solutions}\ m \ggg\!=\\
& && \qquad \mathit{list}\ (\mathit{return}\ [])\ (\lambda a\ n \to \mathit{solutions}\ (k\ a) \oplus n)\\
&(27)\quad \mathit{solutions}\ \mathit{zero} &&= \mathit{return}\ []\\
&(28)\quad \mathit{solutions}\ (m +\!\!+ n) &&= \mathit{solutions}\ m \oplus \mathit{solutions}\ n\\
&(29)\quad \mathit{solutions}\ (\mathit{once}\ m) &&= \mathit{solutions}\ m \ggg\!= \mathit{return} \circ \mathit{list}\ []\ (\lambda a\ _ \to [a])
\end{aligned}$$

The morphism condition for *solution* should be quite obvious:

$$(30)\quad \eta\ (\mathit{solutions}\ m) \quad = \quad \mathit{solutions}\ (\eta\ m)$$

The implementation of this feature turns out to be quite simple except, perhaps, for *BacktrT*. The backtracking monad transformer runs the given computation supplying initial success and failure continuations which construct the desired list.

$$\begin{aligned}
&\mathbf{instance}\ \mathit{MonadSoln}\ \mathit{Maybe}\,\mathbf{where}\\
&\quad \mathit{solutions} = \mathit{return} \circ \mathit{maybe}\ []\ (\lambda a \to [a])\\
&\mathbf{instance}\ \mathit{MonadSoln}\ []\,\mathbf{where}\\
&\quad \mathit{solutions}\ x = [x]\\
&\mathbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadSoln}\ (\mathit{BacktrT}\ m)\,\mathbf{where}\\
&\quad \mathit{solutions}\ m = \lambda\kappa\, f \to m\ (\lambda a\, f \to f \ggg\!= \lambda x \to \mathit{return}\ (a : x))\ (\mathit{return}\ []) \ggg\!= \lambda x \to \kappa\ x\, f
\end{aligned}$$

If m is the identity monad the latter definition of *solutions* boils down to $\lambda\kappa \rightarrow \kappa\ (m\ (:)\ [])$.

Turning to the issue of lifting things become interesting. Suppose that m is an instance of *MonadSoln* $\cdot$, then $StateT\ st\ m\ a = st \rightarrow m\ (st, a)$ adds a 'backtrackable state' to m. Now, if we collect the solutions of $m\ s$ (see below) each solution is paired with the final state of that computation path. But what is the successor state of $solutions\ (m\ s)$? Clearly, taking one of the computed states is more or less arbitrary. Consequently, we pair the list of solutions with the *initial* state. Note that this variant is also implied by morphism condition (30).

$$
\begin{array}{l}
\textbf{instance}\ (MonadSoln\ m) \Rightarrow MonadSoln\ (StateT\ st\ m)\textbf{where} \\
\quad solutions\ m = \lambda s \rightarrow solutions\ (m\ s) \gg\!= \lambda x \rightarrow return\ (s, map\ snd\ x) \\
\textbf{instance}\ (MonadSoln\ m) \Rightarrow MonadSoln\ (EndoT\ m)\textbf{where} \\
\quad solutions\ m = \lambda f \rightarrow solutions\ (m\ zero) +\!\!+ f \\
\textbf{instance}\ (MonadSoln\ m) \Rightarrow MonadSoln\ (ContT\ m)\textbf{where} \\
\quad solutions\ m = \lambda\kappa \rightarrow solutions\ (m\ return) \gg\!= \kappa
\end{array}
$$

7 Adding exception handling

Often it is useful to signal and to trap exceptional situations. This section shows how to add dynamic exception handling in the style of Standard ML (Milner *et al.*, 1990) to a monad. Since this topic has already been addressed elsewhere, see (Spivey, 1989), we keep the presentation rather terse and concentrate on the interplay of exception handling and backtracking.

The rest of this section is organized as follows: Section 7.1 introduces the basic class definitions. Applications extending those of Section 2 are given in Section 7.2. Finally, Section 7.3 shows how to implement the new features.

7.1 Class definitions and derived functions

The computation *zero* says 'the current computation path is a cul-de-sac try another one'. Failure should be distinguished from exceptional situations such as runtime errors, eg division by zero. For that reason we introduce an additional operation termed *raise* which signals an error.

$$
\begin{array}{l}
\textbf{class}\ (Monad\ m) \Rightarrow MonadError\ err\ m\textbf{where} \\
\quad raise :: err \rightarrow m\ a
\end{array}
$$

The argument of *raise* further describes the error. Of course, *raise e* is a left zero of bind: signaling an error terminates the current computation.

$$
\begin{array}{lrcl}
(31) & raise\ e \gg\!= k & = & raise\ e \\
(32) & cut\ (raise\ e +\!\!+ m) & = & raise\ e \\
(33) & solutions\ (raise\ e) & = & raise\ e
\end{array}
$$

The last two equations state that exceptions propagate through *cut* and *solutions*. Equation (33) has an interesting consequence: *solutions m* only succeeds if *no* computation path in m raises an exception.

As usual a new operation calls for a new condition on monad morphisms.

$$(34) \quad \eta\ (raise\ e) \quad = \quad raise\ e$$

As with *zero* and *update* we may use the above condition to lift *raise* through an arbitrary monad transformer.

$$\begin{array}{l} \textbf{instance}\ (MonadError\ err\ m, MonadT\ t\ m) \Rightarrow MonadError\ err\ (t\ m)\ \textbf{where} \\ \quad raise = up \circ raise \end{array}$$

An exception may be catched using an exception handler.

$$\begin{array}{l} \textbf{class}\ (MonadError\ exc\ m) \Rightarrow MonadExc\ exc\ m\ \textbf{where} \\ \quad handle :: m\ a \to (exc \to m\ a) \to m\ a \end{array}$$

The operational behaviour of $m\text{`}handle\text{`}h$ is as follows: m is executed; if m raises an exception, h is applied to *raise*'s argument, otherwise the handler h is ignored.

For the examples following we will require the function *try* which represents successful and exceptional outcome using the disjoint union *Either* (see Section 7.3).

$$\begin{array}{l} try :: MonadExc\ exc\ m \Rightarrow m\ a \to m\ (Either\ exc\ a) \\ try\ m = (m \gg\!= (return \circ Right))\text{`}handle\text{`}(return \circ Left) \end{array}$$

Note that *try m* never raises an exception. The definition shows that instances of *MonadExc exc m* are capable of introspection, ie computational behaviour on the monadic level can be raised to the object level encoded into a suitable data type.

Instances of *MonadExc* are supposed to satisfy the following laws.

$$\begin{array}{lrcl} (35) & return\ a\text{`}handle\text{`}h & = & return\ a \\ (36) & raise\ e\text{`}handle\text{`}h & = & h\ e \\ (37) & m\text{`}handle\text{`}raise & = & m \\ (38) & m\text{`}handle\text{`}(\lambda s \to h\ s\text{`}handle\text{`}j) & = & (m\text{`}handle\text{`}h)\text{`}handle\text{`}j \end{array}$$

Equation (37) states that re-raising an exception is equivalent to not handling it. Equation (38) amounts to the associative law for *handle*.

Things become interesting if we combine soft and hard failure. How do these two kinds of failure interact? What, for example, is the meaning of $m = return\ a +\!\!+ raise\ e +\!\!+ return\ b$? Of course, there are several possible answers to this question. One particular attractive one is the following: m succeeds three times, two times normally and once abnormally. Thus, *raise e* may be interpreted as 'another kind of success'. More operationally speaking we have that choice points are unaffected by calls to *raise*. Accordingly, *handle* may be considered as a variant of $(\gg\!=)$. This correspondence in mind it comes as no surprise that *zero* is a left zero of *handle* and that *handle* distributes leftward through $(+\!\!+)$. As with $(\gg\!=)$ rightward distribution is optional.

$$\begin{array}{lrcl} (39) & zero\text{`}handle\text{`}h & = & zero \\ (40) & (m\text{`}handle\text{`}h) +\!\!+ (n\text{`}handle\text{`}h) & = & (m +\!\!+ n)\text{`}handle\text{`}h \\ (41) & (m\text{`}handle\text{`}h) +\!\!+ (m\text{`}handle\text{`}j) & \stackrel{?}{=} & m\text{`}handle\text{`}\lambda s \to h\ s +\!\!+ j\ s \end{array}$$

Using left distributivity and equations (35) and (36) we may prove, for example, that

$$(\mathit{return}\ a +\!\!+\ \mathit{raise}\ e +\!\!+\ \mathit{return}\ b)\,`\mathit{handle}`\,h \quad = \quad \mathit{return}\ a +\!\!+\ h\ e +\!\!+\ \mathit{return}\ b.$$

Many alternative behaviours are conceivable: Eclipse Prolog, for instance, implements *handle* (called *block*) such it is non re-satisfiable, ie the expression above yields *return a*. The Prolog ISO Standard specifies that *handle* (called *catch*) is re-satisfiable but restricts *raise* (called *throw*) to ignore further choice points:

(42) $\mathit{raise}\ e +\!\!+\ m \quad = \quad \mathit{raise}\ e.$

Note that this identity is incompatible with equation (40). Lifting *raise* into *BacktrT* implements exactly this behaviour since both the success and the failure continuation are discarded.

Since *handle* has a type similar to $(\gg\!=)$ we obtain its morphism condition from equation (12) by substituting *handle* for $(\gg\!=)$.

(43) $\eta\ (m\,`\mathit{handle}`\,h) \quad = \quad \eta\ m\,`\mathit{handle}`\,(\eta \circ h)$

Lifting *handle* through the various monad transformers works as follows: If an exception is raised in a state monad the current state is discarded. Thus the handler is run in the old state. For *EndoT* and *ContT* we build again on the isomorphism results: both instance declarations are derived from $m\,`\mathit{handle}`\,h = \mathit{up}\ (\mathit{down}\ m\,`\mathit{handle}`\,\mathit{down} \circ h)$ by partial evaluation.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{MonadExc}\ \mathit{exc}\ m) \Rightarrow \mathit{MonadExc}\ \mathit{exc}\ (\mathit{StateT}\ \mathit{st}\ m)\,\textbf{where} \\
\quad m\,`\mathit{handle}`\,h = \lambda s \to m\ s\,`\mathit{handle}`\,\lambda a \to h\ a\ s \\
\textbf{instance}\ (\mathit{MonadExc}\ \mathit{exc}\ m) \Rightarrow \mathit{MonadExc}\ \mathit{exc}\ (\mathit{EndoT}\ m)\,\textbf{where} \\
\quad m\,`\mathit{handle}`\,h = \lambda f \to (m\ \mathit{zero}\,`\mathit{handle}`\,\lambda a \to h\ a\ \mathit{zero}) +\!\!+\ f \\
\textbf{instance}\ (\mathit{MonadExc}\ \mathit{exc}\ m) \Rightarrow \mathit{MonadExc}\ \mathit{exc}\ (\mathit{ContT}\ m)\,\textbf{where} \\
\quad m\,`\mathit{handle}`\,h = \lambda \kappa \to (m\ \mathit{return}\,`\mathit{handle}`\,\lambda a \to h\ a\ \mathit{return}) \gg\!= \kappa
\end{array}
$$

The most challenging instance declaration is that of *BacktrT*. The key idea is to encode computational behaviour using suitable data types: success and failure are represented by *Maybe*; normal and abnormal termination by *Either*.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{MonadExc}\ \mathit{exc}\ m) \Rightarrow \mathit{MonadExc}\ \mathit{exc}\ (\mathit{BacktrT}\ m)\,\textbf{where} \\
\quad m\,`\mathit{handle}`\,h = \lambda \kappa\ f \to \textbf{let}\ c\ (\mathit{Left}\ e) = h\ e\ \kappa\ f \\
\qquad\qquad\qquad\qquad\qquad\quad c\ (\mathit{Right}\ \mathit{Nothing}) = f \\
\qquad\qquad\qquad\qquad\qquad\quad c\ (\mathit{Right}\ (\mathit{Just}\ (a, f'))) = \kappa\ a\ f' \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{try}\ (m\ (\lambda a\ f' \to \mathit{return}\ (\mathit{Just}\ (a, \mathit{try}\ f' \gg\!= c)))\ (\mathit{return}\ \mathit{Nothing})) \gg\!= c
\end{array}
$$

The initial continuations m is supplied with perform these encodings; the continuation c then decodes the results and calls either the handler, the failure or the success continuation. Note that alternative choice points are passed through c after applying c 'recursively' to them: $\mathit{try}\ f' \gg\!= c$.

7.2 Applications

Before implementing these features let us first illustrate their use. Returning to Section 2 we find that both groups of examples benefit from the possibility of exception raising and handling. *Technical remark:* In order to avoid large context expressions we introduce shortcuts for extensions of *MonadError String*.

$$\begin{array}{l}\textbf{class}\ (\mathit{MonadError}\ \mathit{String}\ m, \mathit{MonadPlus}\ m) \Rightarrow \mathit{MonadErrPlus}\ m \\ \textbf{instance}\ (\mathit{MonadError}\ \mathit{String}\ m, \mathit{MonadPlus}\ m) \Rightarrow \mathit{MonadErrPlus}\ m\end{array}$$

$$\begin{array}{l}\textbf{class}\ (\mathit{MonadError}\ \mathit{String}\ m, \mathit{MonadCtrl}\ m) \Rightarrow \mathit{MonadErrCtrl}\ m \\ \textbf{instance}\ (\mathit{MonadError}\ \mathit{String}\ m, \mathit{MonadCtrl}\ m) \Rightarrow \mathit{MonadErrCtrl}\ m\end{array}$$

It is well-known that the implementation of the transitive closure works only if the base relation defines an acyclic graph — which should certainly be the case for *child*. For the sake of example let us fake the *child* relation.

$$\begin{array}{l}\mathit{child}\ \texttt{"abraham"} = \cdots +\!\!+\ \mathit{return}\ \texttt{"terach"} \quad \text{-- cycle} \\ \mathit{child}\ \texttt{"haran"} = \cdots +\!\!+\ \mathit{return}\ \texttt{"terach"} \quad \text{-- cycle}\end{array}$$

Now, *descendant* "terach" produces an infinite list of solutions: The first loop in the graph is entered and unrolled ad infinitum. To block this loop we keep track of the nodes we have already visited. If a cycle is detected an exception is raised which signals that the input data is faulty.

$$\begin{array}{l}\mathit{descendant} :: (\mathit{MonadErrPlus}\ m) \Rightarrow \mathit{String} \rightarrow m\ \mathit{String} \\ \mathit{descendant}\ a = \mathit{transitiveClosure}\ \mathit{child}\ [a]\ a\end{array}$$

$$\begin{array}{l}\mathit{transitiveClosure} :: (\mathit{MonadErrPlus}\ m, \mathit{Eq}\ a) \Rightarrow (a \rightarrow m\ a) \rightarrow [a] \rightarrow (a \rightarrow m\ a) \\ \mathit{transitiveClosure}\ m\ \mathit{ancestors}\ a = \textbf{do}\ \ b \leftarrow m\ a \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{when}\ (b \in \mathit{ancestors})\ (\mathit{raise}\ \texttt{"cycle"}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}\ b +\!\!+\ \mathit{transitiveClosure}\ m\ (b : \mathit{ancestors})\ b\end{array}$$

Assume that *BacktrExcM* is an instance of *MonadErrPlus* then

$$\textbf{type}\ \mathit{Res} = \mathit{Either}\ \mathit{String}\ \mathit{String}$$

$$\mathit{run}\ (\mathit{try}\ (\mathit{descendant}\ \texttt{"terach"}) :: \mathit{BacktrExcM}\ \mathit{Res}) :: [\,]\ \mathit{Res}$$

yields nine solutions *including* two errors indicating that a loop has been detected. Alternatively, we may decide to ignore loops:

$$\mathit{run}\ (\mathit{descendant}\ \texttt{"terach"}\text{`}\mathit{handle}\text{`}\lambda\texttt{"cycle"} \rightarrow \mathit{zero} :: \mathit{BacktrExcM}\ \mathit{String}) :: [\,]\ \mathit{String}.$$

Now we obtain seven solutions. Note that the handler replaces an exception by failure, ie it turns hard into soft failure.

Another example which illustrates the usefulness of having two kinds of failure is parsing. In order to provide good error messages a syntax error should be reported as early as possible. For instance, a missing right parenthesis presumably indicates an error and should be signaled as such. The following definitions show how to add error code to the parsing routines of Section 2.2. Note that we additionally replace $(+\!\!+)$ by its restricted variant $(?)$.

$$
\begin{array}{l}
term :: (MonadErrCtrl\ m, MonadParser\ Char\ m) \Rightarrow m\ Term \\
term = \mathbf{do}\ \ v \leftarrow var \\
\qquad\qquad\qquad\qquad return\ (Var\ v) \\
\qquad\qquad\quad ?\ \mathbf{do}\ f \leftarrow fun \\
\qquad\qquad\qquad\qquad as \leftarrow opt\ args\ [\,] \\
\qquad\qquad\qquad\qquad return\ (Fun\ f\ as) \\
\qquad\qquad\quad ?\ raise\ \texttt{"term expected"}
\end{array}
$$

Note that the idiom $m\ ?\ raise\ e$ used above and below turns soft into hard failure.

$$
\begin{array}{l}
args :: (MonadErrCtrl\ m, MonadParser\ Char\ m) \Rightarrow m\ [Term] \\
args = \mathbf{do}\ \ lit\ \texttt{'('} \\
\qquad\qquad\qquad\qquad as \leftarrow sepBy\ term\ (lit\ \texttt{','}) \\
\qquad\qquad\qquad\qquad lit\ \texttt{')'}\ ?\ raise\ \texttt{"')' missing"} \\
\qquad\qquad\qquad\qquad return\ as
\end{array}
$$

The test whether a parser accepts a given string is augmented with error code as well.

$$
\begin{array}{l}
accepts :: (MonadErrCtrl\ m, MonadState\ [tok]\ m) \Rightarrow m\ a \rightarrow [tok] \rightarrow m\ a \\
accepts\ p\ s = \mathbf{do}\ \ old \leftarrow update\ (\lambda\_ \rightarrow s) \\
\qquad\qquad\qquad\qquad a \leftarrow once\ p \\
\qquad\qquad\qquad\qquad t \leftarrow update\ (\lambda\_ \rightarrow old) \\
\qquad\qquad\qquad\qquad unless\ (null\ (t\ \text{`}asTypeOf\text{`}\ s))\ (raise\ \texttt{"incomplete parse"}) \\
\qquad\qquad\qquad\qquad return\ a \\
\qquad\qquad\qquad\ \ ?\ raise\ \texttt{"no parse"}
\end{array}
$$

Assume that *ParserExcM* is a suitable monad combining backtracking, state and exceptions. The routine *parse* puts things together: It returns either the internal representation of the string or an error message indicating why the parse has failed.

$$
\begin{array}{l}
\mathbf{type}\ Res' = Either\ String\ Term \\
\\
parse :: String \rightarrow Either\ String\ Term \\
parse\ s = run\ (try\ (accepts\ (\mathbf{do}\ \ many\ white;\ term)\ s) :: ParserExcM\ Char\ Res')
\end{array}
$$

7.3 Exception monad transformer

For implementing exception handling features we utilize the technique of monad transformers once again. In an exception monad a computation either succeeds gracefully or aborts with an exception. These two possible outcomes are represented by the following disjoint union.

$$
\begin{array}{l}
\mathbf{data}\ Either\ a\ b = Left\ a \mid Right\ b \\
\\
either :: (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow Either\ a\ b \rightarrow c \\
either\ f\ g\ (Left\ a) = f\ a \\
either\ f\ g\ (Right\ b) = g\ b
\end{array}
$$

An exceptional outcome is represented by *Left e*, where *e* is the exception; a normal outcome is accordingly represented by *Right a*, where *a* is the computed value. As before we employ the fold-functional to define functions on *Either*. The exception monad transformer simply composes its argument with *Either exc*.

$$
\begin{array}{l}
\mathbf{type}\ \mathit{ExcT}\ \mathit{exc}\ m\ a = m\ (\mathit{Either}\ \mathit{exc}\ a)\\
\mathbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{ExcT}\ \mathit{exc}\ m)\mathbf{where}\\
\quad m \ggg\!= k = m \ggg\!= \mathit{either}\ (\mathit{return} \circ \mathit{Left})\ (k)\\
\quad \mathit{return} = \mathit{return} \circ \mathit{Right}\\
\mathbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadError}\ \mathit{err}\ (\mathit{ExcT}\ \mathit{err}\ m)\mathbf{where}\\
\quad \mathit{raise} = \mathit{return} \circ \mathit{Left}\\
\mathbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadExc}\ \mathit{exc}\ (\mathit{ExcT}\ \mathit{exc}\ m)\mathbf{where}\\
\quad m\text{`}\mathit{handle}\text{`}h = m \ggg\!= \mathit{either}\ (h)\ (\mathit{return} \circ \mathit{Right})
\end{array}
$$

A computation in *m* always succeeds; it is lifted by wrapping it up in a call to *Right*. The other way round exceptions are mapped to runtime errors using the predefined function *error*.

$$
\begin{array}{l}
\mathbf{instance}\ (\mathit{Show}\ \mathit{exc}, \mathit{Monad}\ m) \Rightarrow \mathit{MonadT}\ (\mathit{ExcT}\ \mathit{exc})\ m\mathbf{where}\\
\quad \mathit{up}\ m = m \ggg\!= (\mathit{return} \circ \mathit{Right})\\
\quad \mathit{down}\ m = m \ggg\!= \mathit{either}\ (\mathit{error} \circ \mathit{show})\ \mathit{return}
\end{array}
$$

It is straightforward to show that *up* is a monad morphism and *down* a premonad morphism. Since *ExcT* composes the base monad from the right the lifting of the remaining operations becomes almost trivial.

$$
\begin{array}{l}
\mathbf{instance}\ (\mathit{MonadZero}\ m) \Rightarrow \mathit{MonadZero}\ (\mathit{ExcT}\ \mathit{exc}\ m)\mathbf{where}\\
\quad \mathit{zero} = \mathit{zero}\\
\mathbf{instance}\ (\mathit{MonadPlus}\ m) \Rightarrow \mathit{MonadPlus}\ (\mathit{ExcT}\ \mathit{exc}\ m)\mathbf{where}\\
\quad m +\!\!+ n = m +\!\!+ n\\
\mathbf{instance}\ (\mathit{MonadCtrl}\ m) \Rightarrow \mathit{MonadCtrl}\ (\mathit{ExcT}\ \mathit{exc}\ m)\mathbf{where}\\
\quad \mathit{once}\ m = \mathit{once}\ (m)\\
\mathbf{instance}\ (\mathit{MonadSoln}\ m) \Rightarrow \mathit{MonadSoln}\ (\mathit{ExcT}\ \mathit{exc}\ m)\mathbf{where}\\
\quad \mathit{solutions}\ m = \mathit{solutions}\ (m) \ggg\!= \mathit{check}\ \mathit{id}
\end{array}
$$

$$
\begin{array}{l}
\mathit{check}\ k\ [\,] = \mathit{return}\ (\mathit{Right}\ (k\ [\,]))\\
\mathit{check}\ k\ (\mathit{Left}\ e : \_) = \mathit{return}\ (\mathit{Left}\ e)\\
\mathit{check}\ k\ (\mathit{Right}\ a : y) = \mathit{check}\ (\lambda x \to k\ (a : x))\ y
\end{array}
$$

The monads, required in the applications above, are constructed as follows.

$$
\begin{array}{l}
\mathbf{type}\ \mathit{BacktrExcM} = \mathit{ExcT}\ \mathit{String}\ [\,]\\
\mathbf{type}\ \mathit{ParserExcM}\ \mathit{tok} = \mathit{StateT}\ [\mathit{tok}]\ (\mathit{ExcT}\ \mathit{String}\ [\,])
\end{array}
$$

It is quite instructive to unfold the definition of *ParserExcM tok*:

$$
\begin{array}{lcl}
\mathit{ParserExcM}\ \mathit{tok}\ a & = & \mathit{StateT}\ [\mathit{tok}]\ (\mathit{ExcT}\ \mathit{String}\ [\,])\ a\\
 & = & [\mathit{tok}] \to [\mathit{Either}\ \mathit{String}\ ([\mathit{tok}], a)].
\end{array}
$$

The resulting type expression makes explicit that the input list is discarded if an exception is raised. This is in accordance with our view of exceptions as hard failures. Of course, nothing prevents us from passing the input list explicitly to the handler, say, for generating a suitable error message. More efficient handlers are obtained using *BacktrT* and *ParserT tok* (note that *Id* is the identity monad).

$$\textbf{type}\ \mathit{BacktrExcM}\ a = \mathit{ExcT}\ \mathit{String}\ (\mathit{BacktrT}\ \mathit{Id})\ a$$
$$\textbf{type}\ \mathit{ParserExcM}\ \mathit{tok}\ a = \mathit{ExcT}\ \mathit{String}\ (\mathit{ParserT}\ a\ \mathit{Id})\ a$$

Remark
As an aside we would like to point out that an exception monad may be used to implement a restricted form of backtracking (à la *Maybe*).

$$\begin{array}{l}\textbf{instance}\ (\mathit{MonadError}\ ()\ m) \Rightarrow \mathit{MonadZero}\ m\textbf{where}\\ \quad \mathit{zero} = \mathit{raise}\ ()\\ \textbf{instance}\ (\mathit{MonadExc}\ ()\ m) \Rightarrow \mathit{MonadPlus}\ m\textbf{where}\\ \quad m +\!\!+\ n = m\text{`}\mathit{handle}\text{`}\lambda() \rightarrow n\end{array}$$

This suggests a way of simulating (a weak form of) backtracking in a language like Standard ML. This approach has been taken, for example, by Paulson (1991) to implement recursive descent parsers.
End of remark

References

Appel, A. W., & MacQueen, D. B. (1991). Standard ML of New Jersey. *Pages 1–13 of:* Maluszyński, J., & Wirsing, M. (eds), *Programming language implementation and logic programming (plilp)*. Lecture Notes in Computer Science, no. 528. Springer-Verlag.

Burton, F.W. (1989). A note on higher-order functions versus logical variables. *Information processing letters*, **31**, 91–95.

Danvy, Olivier, & Filinski, Andrzey. (1990). Abstracting control. *Pages 151–160 of: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, June 1990, Nice*. ACM-Press.

Espinosa, David A. (1995). *Semantic lego*. Ph.d. thesis, Columbia University.

Hartel, P. H. 1994 (December). *Benchmarking implementations of lazy functional languages II – two years later*. Tech. rept. CS-94-21. Dept. of Comp. Sys, Univ. of Amsterdam.

Hutton, Graham, & Meijer, Erik. 1996 (January). *Monadic parser combinators*. Submitted for publication.

Jones, Mark P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.

Jones, Mark P., & Duponcheel, Luc. 1993 (December). *Composing monads*. Tech. rept. YALEU/DCS/RR-1004. Department of Computer Science, Yale University.

Launchbury, John, & Peyton Jones, Simon L. 1994 (June). Lazy functional state threads. *Pages 24–35 of: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, Orlando, Florida. SIGPLAN Notices,* 29(6), June 1994.

Liang, Sheng, Hudak, Paul, & Jones, Mark. 1995 (January). Monad transformers and modular interpreters. *Pages 333–343 of: Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California.*

Mellish, Chris, & Hardy, Steve. (1984). Integrating Prolog into the Poplog environment. *Pages 533–535 of:* Campbell, J.A. (ed), *Implementations of prolog*. Ellis Horwood Limited.

Mellish, C.S., & Clocksin, W.F. (1995). *Programming in prolog*. Fourth edition edn. Berlin: Springer-Verlag.

Milner, Robin, Tofte, Mads, & Harper, Robert. (1990). *The definition of Standard ML*. The MIT Press.

Moggi, Eugenio. (1990). *An abstract view of programming languages*. Tech. rept. ECS-LFCS-90-113. Department of Computer Science, Edinburgh University.

Paulson, Lawrence C. (1991). *ML for the working programmer*. Cambridge University Press.

Peterson, John, & Hammond, Kevin. 1996 (May). *Report on the programming language Haskell — a non-strict, purely functional language, version 1.3*. Tech. rept. Yale University, Department of Computer Science. YALEU/DCS/RR-1106.

Peyton Jones, Simon L, Hall, Cordy, Hammond, Kevin, Partain, Will, & Wadler, Phil. (1993). The Glasgow Haskell compiler: a technical overview. *Proceedings of the uk joint framework for information technology (jfit) technical conference, keele*.

Rémy, Didier. (1994). Programming objects with ML-ART: An extension to ML with abstract and record types. *Pages 321–346 of:* Hagiya, Masami, & Mitchell, John C. (eds), *International symposium on theoretical aspects of computer software*. Sendai, Japan: Springer-Verlag.

Sheard, Tim, & Fegaras, Leonidas. (1993). A fold for all seasons. *Pages 233–242 of: Proceedings 6th ACM SIGPLAN/SIGARCH international conference on functional programming languages and computer architecture, FPCA'93, Copenhagen, Denmark*. ACM-Press.

Somogyi, Zoltan, Henderson, Fergus, & Conway, Thomas. 1995 (February). Mercury: an efficient purely declarative logic programming language. *Pages 499–512 of: Proceedings of the australian computer science conference*.

Spivey, M. (1989). Term rewriting without exceptions. *Science of computer programming*.

Sterling, Leon, & Shapiro, Ehud. (1986). *The art of prolog: Advanced programming techniques*. The MIT Press.

Wadler, Philip. (1987). *List comprehensions*. Series in Computer Science. Prentice Hall International. Chap. 7, pages 127–138.

Wadler, Philip. (1989). Theorems for free! *Pages 347–359 of: Fpca '89: The fourth international conference on functional programming languages and computer architecture, london, uk*. New York: ACM-Press.

Wadler, Philip. (1990). Comprehending monads. *Pages 61–78 of: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*. ACM-Press.

Wadler, Philip. (1995). Monads for functional programming. Jeuring, J., & Meijer, E. (eds), *Advanced Functional Programming, Proceedings of the Båstad Spring School*. Lecture Notes in Computer Science, no. 925. Springer-Verlag.

Wirth, Niklaus. (1979). *Algorithmen und Datenstrukturen*. 2nd edn. Teubner, Stuttgart.

A Proofs

The proofs of the propositions are simple throughout: They can be given using straightforward equational reasoning. There is, however, some danger of redundancy if all of them are undertaken from the scratch. For that reason we chose to re-organize the propositions slightly. We start with two lemmas which provide useful shortcuts.

Lemma 3 Let $up :: M\ a \to N\ a$ be a bijection with its inverse given by *down*. If *down* satisfies (11) and (12) and M is a monad, then N is a monad as well.

Proof
Property (1):

$$
\begin{array}{ll}
 & return\ a \gg\!= k \\
= & \{\ up \circ down = id\ \} \\
 & up\ (down\ (return\ a \gg\!= k)) \\
= & \{\ \text{by (12)}\ \} \\
 & up\ (down\ (return\ a) \gg\!= (down \circ k)) \\
= & \{\ \text{by (11)}\ \} \\
 & up\ (return\ a \gg\!= (down \circ k)) \\
= & \{\ \text{by (1)}\ \} \\
 & up\ (down\ (k\ a)) \\
= & \{\ up \circ down = id\ \} \\
 & k\ a
\end{array}
$$

Property (2):

$$
\begin{array}{ll}
 & m \gg\!= return \\
= & \{\ up \circ down = id\ \} \\
 & up\ (down\ (m \gg\!= return)) \\
= & \{\ \text{by (12)}\ \} \\
 & up\ (down\ m \gg\!= (down \circ return)) \\
= & \{\ \text{by (11)}\ \} \\
 & up\ (down\ m \gg\!= return) \\
= & \{\ \text{by (2)}\ \} \\
 & up\ (down\ m) \\
= & \{\ up \circ down = id\ \} \\
 & m
\end{array}
$$

Property (3):

$$
\begin{array}{ll}
 & m \gg\!= \lambda a \to k\ a \gg\!= \ell \\
= & \{\ up \circ down = id\ \} \\
 & up\ (down\ (m \gg\!= \lambda a \to k\ a \gg\!= \ell)) \\
= & \{\ \text{by (12)}\ \} \\
 & up\ (down\ m \gg\!= \lambda a \to down\ (k\ a \gg\!= \ell)) \\
= & \{\ \text{by (12)}\ \} \\
 & up\ (down\ m \gg\!= \lambda a \to down\ (k\ a) \gg\!= (down \circ \ell)) \\
= & \{\ \text{by (3)}\ \} \\
 & up\ ((down\ m \gg\!= \lambda a \to down\ (k\ a)) \gg\!= (down \circ \ell)) \\
= & \{\ \text{by (12)}\ \} \\
 & up\ (down\ (m \gg\!= k) \gg\!= (down \circ \ell)) \\
= & \{\ \text{by (12)}\ \} \\
 & up\ (down\ ((m \gg\!= k) \gg\!= \ell)) \\
= & \{\ up \circ down = id\ \} \\
 & (m \gg\!= k) \gg\!= \ell
\end{array}
$$

□

Lemma 4 Let $up :: M\ a \to N\ a$ be a bijection with its inverse given by *down*. If *up* satisfies (11), so does *down*. Ditto for (12).

Proof
Property (11):

$$\begin{aligned}
& down\ (return\ a) \\
= \ & \{\ \text{by (11)}\ \} \\
& down\ (up\ (return\ a)) \\
= \ & \{\ down \circ up = id\ \} \\
& return\ a
\end{aligned}$$

Property (12):

$$\begin{aligned}
& down\ (m \gg\!= k) \\
= \ & \{\ up \circ down = id\ \} \\
& down\ (up\ (down\ m) \gg\!= (up \circ down \circ k)) \\
= \ & \{\ \text{by (12)}\ \} \\
& down\ (up\ (down\ m \gg\!= (down \circ k))) \\
= \ & \{\ down \circ up = id\ \} \\
& down\ m \gg\!= (down \circ k)
\end{aligned}$$

□

The two lemmas suggest to prove the bijection property first since this cuts the work down by half.

Proof of Lemma 1
We show the proposition by structural induction on c. Note that we extend Lemma 1 slightly by taking the combinators *once*, *solutions*, and *handle* into account. Case $c = return\ a$:

$$\begin{aligned}
& (return\ a)\ f \\
= \ & \{\ \text{def. } return\ \} \\
& return\ a +\!\!+ f \\
= \ & \{\ \text{by (7)}\ \} \\
& (return\ a +\!\!+ zero) +\!\!+ f \\
= \ & \{\ \text{def. } return\ \} \\
& (return\ a)\ zero +\!\!+ f
\end{aligned}$$

Case $c = m \gg\!= k$:

$$\begin{aligned}
& (m \gg\!= k)\ f \\
= \ & \{\ \text{def. } (\gg\!=)\ \} \\
& (m\ zero \gg\!= \lambda a \to k\ a\ zero) +\!\!+ f \\
= \ & \{\ \text{by (7)}\ \} \\
& ((m\ zero \gg\!= \lambda a \to k\ a\ zero) +\!\!+ zero) +\!\!+ f \\
= \ & \{\ \text{def. } (\gg\!=)\ \} \\
& (m \gg\!= k)\ zero +\!\!+ f
\end{aligned}$$

Case $c = up\ m$:

$$\begin{array}{ll} & (up\ m)\ f \\ = & \{\ \text{def. } up\ \} \\ & m \mathbin{+\!\!+} f \\ = & \{\ \text{by (7)}\ \} \\ & (m \mathbin{+\!\!+} zero) \mathbin{+\!\!+} f \\ = & \{\ \text{def. } up\ \} \\ & (up\ m)\ zero \mathbin{+\!\!+} f \end{array}$$

Case $c = zero$:

$$\begin{array}{ll} & zero\ f \\ = & \{\ \text{def. } zero\ \} \\ & f \\ = & \{\ \text{by (6)}\ \} \\ & zero \mathbin{+\!\!+} f \\ = & \{\ \text{def. } zero\ \} \\ & zero\ zero \mathbin{+\!\!+} f \end{array}$$

Case $c = m \mathbin{+\!\!+} n$:

$$\begin{array}{ll} & (m \mathbin{+\!\!+} n)\ f \\ = & \{\ \text{def. } (\mathbin{+\!\!+})\ \} \\ & m\ (n\ f) \\ = & \{\ \text{ind. hyp. } n\ \} \\ & m\ (n\ zero \mathbin{+\!\!+} f) \\ = & \{\ \text{ind. hyp. } m\ \} \\ & m\ zero \mathbin{+\!\!+} (n\ zero \mathbin{+\!\!+} f) \\ = & \{\ \text{by (8)}\ \} \\ & (m\ zero \mathbin{+\!\!+} n\ zero) \mathbin{+\!\!+} f \\ = & \{\ \text{ind. hyp. } m\ \} \\ & m\ (n\ zero) \mathbin{+\!\!+} f \\ = & \{\ \text{def. } (\mathbin{+\!\!+})\ \} \\ & (m \mathbin{+\!\!+} n)\ zero \mathbin{+\!\!+} f \end{array}$$

Cases $c = once\ m$, $c = solutions\ m$, $c = m`handle`h$: Since the definitions of these operations are based on the isomorphism result, the proofs are nearly identical. For that reason we list only the one for *handle*.

$$\begin{array}{ll} & (m`handle`h)\ f \\ = & \{\ \text{def. } handle\ \} \\ & (m\ zero`handle`\lambda a \to h\ a\ zero) \mathbin{+\!\!+} f \\ = & \{\ \text{by (7)}\ \} \\ & ((m\ zero`handle`\lambda a \to h\ a\ zero) \mathbin{+\!\!+} zero) \mathbin{+\!\!+} f \\ = & \{\ \text{def. } handle\ \} \\ & (m`handle`h)\ zero \mathbin{+\!\!+} f \end{array}$$

□

Lemma 5 Let $up\ m = \lambda f \to m +\!\!+ f$, then up is a bijection of $m\ a$ onto $EndoT\ m\ a$ with its inverse given by $down\ m = m\ zero$.

Proof
Recall that $EndoT\ m\ a$ is reserved for the set of all functions $m\ a \to m\ a$ satisfying equation (18). '$up \circ down = id$':

$$\begin{aligned}
& up\ (down\ m) \\
= {} & \{\ \text{def.}\ down\ \} \\
& up\ (m\ zero) \\
= {} & \{\ \text{def.}\ up\ \} \\
& \lambda f \to m\ zero +\!\!+ f \\
= {} & \{\ \text{by (18)}\ \} \\
& m
\end{aligned}$$

'$down \circ up = id$':

$$\begin{aligned}
& down\ (up\ m) \\
= {} & \{\ \text{def.}\ up\ \} \\
& down\ (\lambda f \to m +\!\!+ f) \\
= {} & \{\ \text{def.}\ down\ \} \\
& m +\!\!+ zero \\
= {} & \{\ \text{by (7)}\ \} \\
& m
\end{aligned}$$

□

Proof of Theorem 1
We show first that up satisfies (11) and (12). Property (11):

$$\begin{aligned}
& up\ (return\ a) \\
= {} & \{\ \text{def.}\ up\ \} \\
& \lambda f \to return\ a +\!\!+ f \\
= {} & \{\ \text{def.}\ return\ \} \\
& return\ a
\end{aligned}$$

Property (12):

$$\begin{aligned}
& up\ (m \gg\!\!= k) \\
= {} & \{\ \text{def.}\ up\ \} \\
& \lambda f \to (m \gg\!\!= k) +\!\!+ f \\
= {} & \{\ down \circ up = id\ \} \\
& \lambda f \to (down\ (up\ m) \gg\!\!= (down \circ up \circ k)) +\!\!+ f \\
= {} & \{\ \text{def.}\ down\ \} \\
& \lambda f \to (up\ m\ zero \gg\!\!= \lambda a \to up\ (k\ a)\ zero) +\!\!+ f \\
= {} & \{\ \text{def.}\ (\gg\!\!=)\ \} \\
& up\ m \gg\!\!= (up \circ k)
\end{aligned}$$

Lemma 4 and Lemma 5 imply that $down$ satisfies (11) and (12) as well. Using Lemma 3 and Lemma 5 it follows that $EndoT\ m$ is a monad. Properties (6) – (8) are trivially satisfied, since $(m\ a \to m\ a, \circ, id)$ forms a monoid. □

Proof of Theorem 2

In the proof of Theorem 1 we have shown that *up* and *down* are monad morphisms. Hence the proposition follows immediately from Lemma 5. □

Proof of Lemma 2

We show the proposition by structural induction on c. Note that we extend Lemma 2 slightly by taking the combinators *once*, *solutions*, and *handle* into account. Case $c = \mathit{return}\ a$:

$$\begin{array}{ll}
 & (\mathit{return}\ a)\ \kappa \\
= & \{\ \text{def. } \mathit{return}\ \} \\
 & \kappa\ a \\
= & \{\ \text{by (1)}\ \} \\
 & \mathit{return}\ a \gg= \kappa \\
= & \{\ \text{def. } \mathit{return}\ \} \\
 & (\mathit{return}\ a)\ \mathit{return} \gg= \kappa
\end{array}$$

Case $c = m \gg= k$:

$$\begin{array}{ll}
 & (m \gg= k)\ \kappa \\
= & \{\ \text{def. } (\gg=)\ \} \\
 & m\ (\lambda a \to k\ a\ \kappa) \\
= & \{\ \text{ind. hyp. } m\ \} \\
 & m\ \mathit{return} \gg= \lambda a \to k\ a\ \kappa \\
= & \{\ \text{ind. hyp. } (k\ a)\ \} \\
 & m\ \mathit{return} \gg= \lambda a \to k\ a\ \mathit{return} \gg= \kappa \\
= & \{\ \text{by (3)}\ \} \\
 & (m\ \mathit{return} \gg= \lambda a \to k\ a\ \mathit{return}) \gg= \kappa \\
= & \{\ \text{ind. hyp. } m\ \} \\
 & (m\ (\lambda a \to k\ a\ \mathit{return})) \gg= \kappa \\
= & \{\ \text{def. } (\gg=)\ \} \\
 & (m \gg= k)\ \mathit{return} \gg= \kappa
\end{array}$$

Case $c = \mathit{up}\ m$:

$$\begin{array}{ll}
 & (\mathit{up}\ m)\ \kappa \\
= & \{\ \text{def. } \mathit{up}\ \} \\
 & m \gg= \kappa \\
= & \{\ \text{by (2)}\ \} \\
 & (m \gg= \mathit{return}) \gg= \kappa \\
= & \{\ \text{def. } \mathit{up}\ \} \\
 & (\mathit{up}\ m)\ \mathit{return} \gg= \kappa
\end{array}$$

Case $c = zero$:

$$\begin{aligned}
& zero\ \kappa \\
= & \quad \{ \text{ def. } zero \ \} \\
& zero \\
= & \quad \{ \text{ by } (4) \ \} \\
& zero \gg\!= \kappa \\
= & \quad \{ \text{ def. } zero \ \} \\
& zero\ return \gg\!= \kappa
\end{aligned}$$

Case $c = m +\!\!+\, n$:

$$\begin{aligned}
& (m +\!\!+\, n)\ \kappa \\
= & \quad \{ \text{ def. } (+\!\!+) \ \} \\
& m\ \kappa +\!\!+\, n\ \kappa \\
= & \quad \{ \text{ ind. hyp. } m \text{ and ind. hyp. } n \ \} \\
& (m\ return \gg\!= \kappa) +\!\!+ (n\ return \gg\!= \kappa) \\
= & \quad \{ \text{ by } (9) \ \} \\
& (m\ return +\!\!+\, n\ return) \gg\!= \kappa \\
= & \quad \{ \text{ def. } (+\!\!+) \ \} \\
& (m +\!\!+\, n)\ return \gg\!= \kappa
\end{aligned}$$

Cases $c = once\ m$, $c = solutions\ m$, $c = m\,`handle`\,h$: Since the definitions of these operations are based on the isomorphism result, the proofs are nearly identical. For that reason we list only the one for *handle*.

$$\begin{aligned}
& (m\,`handle`\,h)\ \kappa \\
= & \quad \{ \text{ def. } handle \ \} \\
& (m\ return\,`handle`\,\lambda a \to h\ a\ return) \gg\!= \kappa \\
= & \quad \{ \text{ by } (2) \ \} \\
& ((m\ return\,`handle`\,\lambda a \to h\ a\ return) \gg\!= return) \gg\!= \kappa \\
= & \quad \{ \text{ def. } handle \ \} \\
& (m\,`handle`\,h)\ return \gg\!= \kappa
\end{aligned}$$

□

Lemma 6 Let $up\ m = \lambda\kappa \to m \gg\!= \kappa$, then up is a bijection of $m\ a$ onto $ContT\ a$ with its inverse given by $down\ m = m\ return$.

Proof
Recall that $ContT\ m\ a$ is reserved for the set of all functions $\forall ans.(a \to m\ ans) \to m\ ans$ satisfying equation (19). '$up \circ down = id$':

$$\begin{aligned}
& up\ (down\ m) \\
= & \quad \{ \text{ def. } down \ \} \\
& up\ (m\ return) \\
= & \quad \{ \text{ def. } up \ \} \\
& \lambda\kappa \to m\ return \gg\!= \kappa \\
= & \quad \{ \text{ by } (19) \ \} \\
& m
\end{aligned}$$

'$down \circ up = id$':

$$
\begin{aligned}
& down\ (up\ m) \\
= \ & \{\ \text{def. } up\ \} \\
& down\ (\lambda\kappa \to m \gg\!= \kappa) \\
= \ & \{\ \text{def. } down\ \} \\
& m \gg\!= return \\
= \ & \{\ \text{by } (2)\ \} \\
& m
\end{aligned}
$$

□

Proof of Theorem 3
We show first that *up* satisfies (11) and (12). Property (11):

$$
\begin{aligned}
& up\ (return\ a) \\
= \ & \{\ \text{def. } up\ \} \\
& \lambda\kappa \to return\ a \gg\!= \kappa \\
= \ & \{\ \text{by } (1)\ \} \\
& \lambda\kappa \to \kappa\ a \\
= \ & \{\ \text{def. } return\ \} \\
& return\ a
\end{aligned}
$$

Property (12):

$$
\begin{aligned}
& up\ (m \gg\!= k) \\
= \ & \{\ \text{def. } up\ \} \\
& \lambda\kappa \to (m \gg\!= k) \gg\!= \kappa \\
= \ & \{\ \text{by } (3)\ \} \\
& \lambda\kappa \to m \gg\!= \lambda a \to k\ a \gg\!= \kappa \\
= \ & \{\ \text{def. } up\ \} \\
& \lambda\kappa \to m \gg\!= \lambda a \to up\ (k\ a)\ \kappa \\
= \ & \{\ \text{def. } up\ \} \\
& \lambda\kappa \to up\ m\ (\lambda a \to up\ (k\ a)\ \kappa) \\
= \ & \{\ \text{def. } (\gg\!=)\ \} \\
& up\ m \gg\!= (up \circ k)
\end{aligned}
$$

Lemma 4 and Lemma 6 imply that *down* satisfies (11) and (12) as well. Using Lemma 3 and Lemma 6 it follows that *ContT m* is a monad which in turn implies the proposition. □

Proof of Theorem 4
Property (1):

$$
\begin{aligned}
& return\ a \gg\!= k \\
= \ & \{\ \text{def. } (\gg\!=)\ \} \\
& \lambda\kappa \to return\ a\ (\lambda a \to k\ a\ \kappa) \\
= \ & \{\ \text{def. } return\ \} \\
& k\ a
\end{aligned}
$$

Property (2):

$$\begin{aligned}
& m \gg\!= return \\
= \;& \{ \text{ def. } (\gg\!=) \; \} \\
& \lambda\kappa \to m\,(\lambda a \to return\; a\; \kappa) \\
= \;& \{ \text{ def. } return \; \} \\
& m
\end{aligned}$$

Property (3):

$$\begin{aligned}
& m \gg\!= \lambda a \to k\; a \gg\!= \ell \\
= \;& \{ \text{ def. } (\gg\!=) \; \} \\
& \lambda\kappa \to m\,(\lambda a \to (k\; a \gg\!= \ell)\; \kappa) \\
= \;& \{ \text{ def. } (\gg\!=) \; \} \\
& \lambda\kappa \to m\,(\lambda a \to k\; a\,(\lambda b \to \ell\; b\; \kappa)) \\
= \;& \{ \text{ def. } (\gg\!=) \; \} \\
& \lambda\kappa \to (m \gg\!= k)\,(\lambda b \to \ell\; b\; \kappa) \\
= \;& \{ \text{ def. } (\gg\!=) \; \} \\
& (m \gg\!= k) \gg\!= \ell
\end{aligned}$$

Property (4):

$$\begin{aligned}
& zero \gg\!= k \\
= \;& \{ \text{ def. } (\gg\!=) \; \} \\
& \lambda\kappa \to zero\,(\lambda a \to k\; a\; \kappa) \\
= \;& \{ \text{ def. } zero \; \} \\
& \lambda\kappa \to id \\
= \;& \{ \text{ def. } zero \; \} \\
& zero
\end{aligned}$$

Property (6):

$$\begin{aligned}
& zero + m \\
= \;& \{ \text{ def. } (+\!\!+) \; \} \\
& \lambda\kappa \to zero\; \kappa \circ m\; \kappa \\
= \;& \{ \text{ def. } zero \; \} \\
& \lambda\kappa \to id \circ m\; \kappa \\
= \;& \{ \; id \circ f = f \; \} \\
& m
\end{aligned}$$

Property (7): analogous. Property (8):

$$
\begin{array}{cl}
 & (m \mathbin{+\!\!+} n) \mathbin{+\!\!+} o \\
= & \{ \text{ def. } (\mathbin{+\!\!+}) \ \} \\
 & \lambda\kappa \to (m \mathbin{+\!\!+} n)\ \kappa \circ o\ \kappa \\
= & \{ \text{ def. } (\mathbin{+\!\!+}) \ \} \\
 & \lambda\kappa \to (m\ \kappa \circ n\ \kappa) \circ o\ \kappa \\
= & \{ \ (f \circ g) \circ h = f \circ (g \circ h) \ \} \\
 & \lambda\kappa \to m\ \kappa \circ (n\ \kappa \circ o\ \kappa) \\
= & \{ \text{ def. } (\mathbin{+\!\!+}) \ \} \\
 & \lambda\kappa \to m\ \kappa \circ (n \mathbin{+\!\!+} o)\ \kappa \\
= & \{ \text{ def. } (\mathbin{+\!\!+}) \ \} \\
 & m \mathbin{+\!\!+} (n \mathbin{+\!\!+} o)
\end{array}
$$

Property (9):

$$
\begin{array}{cl}
 & (m \gg\!\!= k) \mathbin{+\!\!+} (n \gg\!\!= k) \\
= & \{ \text{ def. } (\mathbin{+\!\!+}) \ \} \\
 & \lambda\kappa \to (m \gg\!\!= k)\ \kappa \circ (n \gg\!\!= k)\ \kappa \\
= & \{ \text{ def. } (\gg\!\!=) \ \} \\
 & \lambda\kappa \to m\ (\lambda a \to k\ a\ \kappa) \circ n\ (\lambda a \to k\ a\ \kappa) \\
= & \{ \text{ def. } (\mathbin{+\!\!+}) \ \} \\
 & \lambda\kappa \to (m \mathbin{+\!\!+} n)\ (\lambda a \to k\ a\ \kappa) \\
= & \{ \text{ def. } (\gg\!\!=) \ \} \\
 & (m \mathbin{+\!\!+} n) \gg\!\!= k
\end{array}
$$

B Source code

The C code is modeled after the Haskell program listed in Section 2.1. Instead of lists we use bit sets, ie elements of `int`, to record the queens' positions. Apart from the assignment to the global variable `solutions` the code is purely functional. If we used `int` arrays instead of bit sets things would be different, of course.

```
int solutions;

void place(int n, int i, int rs, int d1, int d2) {
    int q;

    if (i == 0)
        solutions++;
    else
        for (q = 1; q <= n; q++) {
            int r  = 1 << q;
            int q1 = 1 << (q - i);
            int q2 = 1 << (q + i);

            if (!(r & rs | q1 & d1 | q2 & d2))
```

```
                place(n, i - 1, r | rs, q1 | d1, q2 | d2);
        }
}

void queens(int n) {
    solutions = 0;
    place(n, n, 0, 0, 0);
    printf("queens(%i):  %i solutions\n", n, solutions);
}
```

The following program is due to Thom Frühwirth. It relies on unification instead of arithmetic operations to check for attacking queens. The up- and the down diagonal are represented by open lists of fields.

```
queens(N, Qs) :-
        gen_list(N, Qs),
        place_queens(N, Qs, _, _).

gen_list(0, []).
gen_list(N, [_|L]) :-
        N > 0,
        M is N - 1,
        gen_list(M, L).

place_queens(0, _, _, _).
place_queens(I, Rs, D1, [_|D2]) :-
        I > 0,
        J is I - 1,
        place_queens(J, Rs, [_|D1], D2),
        place_queen(I, Rs, D1, D2).

place_queen(I, [I|_], [I|_], [I|_]).
place_queen(I, [_|Rs], [_|D1], [_|D2]):-
        place_queen(I, Rs, D1, D2).
```

The number of solutions is counted via a failure-driven loop which increments a global variable, an extra-logical feature many Prolog implementations offer. Alternatively, one may employ an 'all solutions' predicate which collects the solutions in an unsorted list. The latter technique is used in the Mercury version.

```
count(N, _) :-
        setval(solutions, 0),
        queens(N, _),
        getval(solutions, I),
        J is I + 1,
        setval(solutions, J),
        fail.
```

```
count(N, Solutions) :-
        getval(solutions, Solutions).
```