PROLOGICAL FEATURES IN A FUNCTIONAL SETTING AXIOMS AND IMPLEMENTATIONS

RALF HINZE
Institut für Informatik III, Universität Bonn, Römerstraße 164
53117 Bonn, Germany
E-mail: ralf@uran.informatik.uni-bonn.de

The purpose of this paper is twofold. First, we show that Prological features can be smoothly integrated into a functional language like Haskell. The resulting 'language', embedded Prolog, lacks some concepts such as logical variables but it inherits all of Haskell's strengths eg static polymorphic typing, higher order functions etc. Technically, the integration is achieved using monads and monad transformers. One of the main innovations is the definition of a backtracking monad transformer which allows to combine backtracking with exception handling and interaction. Second, we work towards an axiomatization of the operations through which the computational features are accessed. Equations are used to lay down the meaning of the various operations and their interrelations enabling the programmer to reason about programs in a simple calculational style.

1 Introduction

Many proposals have been made for the integration of functional and logic programming, see[1] for a lucid account. We add another suggestion to this never ending list which has the remarkable property of being simple. We propose to marry both concepts via an embedding of Prolog — or rather, Prolog's control operations, into Haskell. An embedded language inherits the infrastructure of the host language. Thus embedded Prolog enjoys all of Haskell's strengths: static polymorphic typing, higher-order functions etc. Embedded Prolog additionally incorporates many of the features prescribed by the ISO draft standard proposal[2]: control constructs, all solution collecting functions, and error handling facilities. Some features, however, are lacking because they do not go well together with the use of Haskell as a host language: the concept of a logical variable and Prolog's data base operations. Logical variables are not supported because we want to retain Haskell's data types and its pattern matching facilities: a Haskell list remains a list in embedded Prolog. Database operations such as *assert* and *retract* are critical since they allow to write self-modifiable programs. The manipulation of a global state, however, does not pose any problems. We do not consider this extension here solely for reasons of space. In view of these differences the term "embedded Prolog" may be misleading; it was chosen primarily to emphasize the fact that we consider sequential Prolog and not its abstract Horn logic core (see Section 4.1).

Some attempts have been made to explain the semantics of full Prolog, both denotational [3] and operational [4]. We add a proposal to the third strand, the axiomatic approach. The main objective of the axiomatic approach is to aid the programmer in stating and proving properties about programs. Since we employ Haskell as a host language the idea suggests itself to utilize equational logic for reasoning about embedded Prolog programs. We hope to convince the reader that this approach is both attractive and feasible. The axiomatic approach has one further advantage: it is modular, new computational features can be taken into account simply by extending the set of axioms.

Technically, the embedding of Prolog into Haskell is achieved using monads and monad transformers. Both concepts have been proposed by Moggi [5,6] as a means to structure denotational semantics. Monads have received a great deal of attention in the functional programming community since then. Wadler [7] distinguishes between internal and external uses of monads: internally, they serve as a structuring technique for writing programs and externally as a means for extending a language. The IO monad [8] which provides interaction with the environment serves as a good example of the latter type. Examples of internal uses are given in [9]. Our use of monads lies on the borderline: while it is true that embedded Prolog adds new features to Haskell it does not strictly increase its power. In fact, we will see that every feature can be implemented in Haskell itself.

The implementation of embedded languages is well supported by Haskell's class system. Following [10] we define a class for each computational feature; embedded Prolog programs access the required features simply by calling the appropriate class methods. In a sense we employ the class system to define a clean and rigid interface between the application code and the implementation code. To implement a computational feature we must provide an instance of the respective class; to implement embedded Prolog we must develop a monad which is simultaneously instance of all computational classes. One can imagine that it is a tedious and error-prone task to program such a monad from the scratch. That is the point where monad transformers come into play: in essence a monad transformer extends a given monad by a certain feature. A monad which supports a variety of features is built by applying a sequence of transformers to a base monad such as IO.

One of the main innovations of the paper is the definition of a backtracking monad transformer which adds backtracking to an arbitrary monad. Among other things it allows to study the interplay of backtracking and interaction which was previously not possible. In addition we propose a simple scheme for encapsulating monads. Encapsulation of computations is vital since it allows, for instance, to turn an embedded Prolog program into a pure function.

The paper is organized as follows. Section 2 defines monads and introduces the different constructs of the embedded language. For each linguistic feature — nondeterminism, exception handling, and interaction — a subclass of the basic monadic class is given. Section 3 presents several motivating examples demonstrating among other things the use of higher order computations. A first attempt at an axiomatization of the computational primitives appears in Section 4. Section 5 describes the construction of a monad supporting the various linguistic features. Perhaps surprisingly, the implementation appears to compare favourably to that of logic languages like Prolog or Mercury [11] which offer non-determinism as a 'language primitive'. Section 6 provides some evidence for this claim. Finally, in Section 7 we relate our work to other approaches and suggest some directions for future work.

2 Monads

Think of a monad as an abstract type for computations which comes equipped with two principal operations.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{return} & :: & a \to m\ a \\
(\ggg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b \\
(\gg) & :: & m\ a \to m\ b \to m\ b \\
\multicolumn{3}{l}{m \gg n = m \ggg\!= \lambda\_ \to n}
\end{array}
\end{array}
$$

An element of $m\ a$ represents a computation which yields a value of type a. The trivial computation which immediately returns the value a is denoted by $\mathit{return}\ a$. The operator $(\ggg\!=)$, commonly called 'bind', combines two computations: $m \ggg\!= k$ applies k to the result of the computation m. The derived operation $(\gg)$ provides a handy shortcut if one is not interested in the result of the first computation; $m \gg n$ in effect sequences m and n. Passing by we note that $(\gg)$ roughly corresponds to the 'logical and'. Furthermore, the predicate which always succeeds is given by

$$
\begin{array}{l}
\mathit{true} :: (\mathit{Monad}\ m) \Rightarrow m\ () \\
\mathit{true} = \mathit{return}\ ()\ \ .
\end{array}
$$

Thus *return* and $(\ggg\!=)$ constitute the conjunctive kernel of embedded Prolog.

Building upon *return* and $(\ggg\!=)$ we can define a couple of auxiliary functions which will prove useful in the sequel: the operator $(\circledcirc)$, sometimes called Kleisli composition, composes two functions involving computations. Contrary to the usual composition it also takes care of computational effects.

$$\begin{aligned} &(\circledcirc) \quad :: (\mathit{Monad}\ m) \Rightarrow (b \to m\ c) \to (a \to m\ b) \to (a \to m\ c) \\ &f \circledcirc g = \lambda a \to g\ a \gg\!= f \end{aligned}$$

Functional programmers often use the function *map* to apply a function to each element of a given list. However, *map* can be generalized to work over an arbitrary monad: $\mathit{map}\ f\ m$ applies f to the result of the computation m.

$$\begin{aligned} &\mathit{map} \qquad :: (\mathit{Monad}\ m) \Rightarrow (a \to b) \to m\ a \to m\ b \\ &\mathit{map}\ f\ m = m \gg\!= \mathit{return} \circ f \end{aligned}$$

2.1 Encapsulation

The experienced programmer will probably remark that the class definition of monads is incomplete: *return* gets us into a monad, $(\gg\!=)$ get us around, but no operation is designated to get out of a monad. The following class definition fills the gap.

$$\begin{aligned} &\textbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Run}\ m\ \textbf{where} \\ &\quad \mathit{run} :: m\ a \to a \end{aligned}$$

The function *run* turns a computation into a proper value. In other words it encapsulates a computation thereby restricting the extent of all computational features involved.

The approach taken is not undebatable: in most cases *run* will be a partial function. Consider, for instance, a nondeterministic computation: what should *run* yield if the computation has no or more than one solution? We offer a simple, yet effective solution to this problem. The key idea is to provide additional computational primitives so that the programmer can explicitly deal with issues like this. In the case of nondeterministic computations we supply an operation, called *sols*, which collects all solutions of a given computation. Since $\mathit{sols}\ m$ has exactly one solution it can be safely encapsulated: $\mathit{run}\ (\mathit{sols}\ m)$ yields a list of all solutions m generates.

Not every monad can be encapsulated: the *IO* monad which provides interaction with the environment is one exception: a computation which possibly depends on external devices or on user input cannot be turned into a value without sacrificing referential transparency. For that reason a variant of *Run* is introduced which defines a mapping into the *IO* monad.

$$\begin{aligned} &\textbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Perform}\ m\ \textbf{where} \\ &\quad \mathit{perform} :: m\ a \to \mathit{IO}\ a \end{aligned}$$

2.2 Backtracking

The following class definition models a small, but significant part of Prolog's backtracking core.

$$\begin{array}{l}\textbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Backtr}\ m\ \textbf{where} \\ \quad \begin{array}{lll} \mathit{fail} & :: & m\ a \\ (\mid) & :: & m\ a \to m\ a \to m\ a \\ \mathit{once} & :: & m\ a \to m\ (\mathit{Maybe}\ a) \\ \mathit{sols} & :: & m\ a \to m\ [a] \end{array}\end{array}$$

The constant *fail* denotes a failing computation, $(\mid)$ realizes a nondeterministic choice. Both operations have a simple operational interpretation: $m \mid n$ means 'try m first, if it fails try n', *fail* means 'the current computation is a cul-de-sac try another one'. The operational reading indicates that the term 'nondeterministic choice' is really a misnomer: $(\mid)$ is not commutative, the order in which the alternatives are presented usually matters, see Section 4.1.

The operations *once* and *sols* encode computational behaviour using the data types *Maybe* and $[\,]$: *once* m returns *Nothing* if m fails and *Just* a where a is the first solution of m otherwise. Accordingly, *sols* m returns the list of all solutions of m. A word of warning is appropriate: *once* differs from Prolog's *once* in that the former succeeds exactly once whereas the latter may fail. It is not difficult, however, to define the latter in terms of the former (the other way round is slightly harder and left as an exercise to the reader).

$$\begin{array}{lll} \mathit{atMostOnce} & :: & (\mathit{Backtr}\ m) \Rightarrow m\ a \to m\ a \\ \mathit{atMostOnce}\ m & = & \mathit{once}\ m \gg\!= \mathit{maybe}\ \mathit{fail}\ \mathit{return} \end{array}$$

Having defined the backtracking core it is time to elaborate on the differences between Prolog and embedded Prolog. A Prolog procedure defines a relationship between objects. The well-known procedure *append* (X, Y, Z), for example, defines a relation between lists: it is true if Z is the catenation of X and Y. Since *append* defines a relation it may be used in different modes: to catenate two lists or to split a list into two. This property has inspired the characterization of Prolog as a 'relational programming language'. In embedded Prolog we cannot define a true relation; we must commit ourselves to a certain mode: to catenate two lists we simple use $(+\!\!+)$, for splitting a list into two we define

$$\begin{array}{lll} \mathit{split} & :: & (\mathit{Backtr}\ m) \Rightarrow [a] \to m\ ([a],[a]) \\ \mathit{split}\ [\,] & = & \mathit{return}\ ([\,],[\,]) \\ \mathit{split}\ (a : x) & = & \mathit{return}\ ([\,], a : x) \\ & \mid & \mathit{split}\ x \gg\!= \lambda(y,z) \to \mathit{return}\ (a : y, z)\ . \end{array}$$

Generally, a *directed* relation between a and b is represented by a function of type $a \to m\ b$ where m is a backtracking monad. It is debatable whether the restriction to directed relations is severe. Two observations suggest that this might not be the case. First, true relational programming is hard to achieve: the goal $\mathit{append}\ (X, [1,2], X)$, for instance, does not fail but diverges instead. A high percentage of Prolog programs probably defines only directed relations. Second, the importance of modes seems to be widely accepted as documented by their integration into the logic language Mercury [11].

We conclude the section by noting that the second-orderness of $(\mid)$, *once*, and *sols* poses no problems since computations are first-class citizens in Haskell. Here is another useful higher-order combinator which implements the union of directed relations.

$$
\begin{array}{ll}
(\oplus) & :: (\mathit{Backtr}\ m) \Rightarrow (a \to m\ b) \to (a \to m\ b) \to (a \to m\ b) \\
f \oplus g & = \lambda a \to f\ a \mid g\ a
\end{array}
$$

2.3 Exception handling

We provide one operation for signaling and one for trapping exceptional situations.

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Err} = \mathit{String} \\
\mathbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Exc}\ m\ \mathbf{where} \\
\quad \mathit{raise} :: \mathit{Err} \to m\ a \\
\quad \mathit{try}\ \ \ :: m\ a \to m\ (\mathit{Either}\ \mathit{Err}\ a)
\end{array}
$$

Their operational behaviour is as follows: $\mathit{try}\ m$ executes m, if m signals an exception by executing, say, $\mathit{raise}\ e$ then $\mathit{try}\ m$ returns $\mathit{Left}\ e$, otherwise it returns $\mathit{Right}\ a$ where a is the result of m.

Note that the Prolog ISO Standard incorporates two similar constructs, termed *throw* and *catch*. The control construct *throw* is tantamount to *raise*, *catch* is a mild variant of *try* and can be easily defined in terms of it (the converse also holds and is left as an exercise to the reader).

$$
\begin{array}{ll}
\mathit{catch} & :: (\mathit{Exc}\ m) \Rightarrow m\ a \to (\mathit{Err} \to m\ a) \to m\ a \\
\mathit{catch}\ m\ h & = \mathit{try}\ m \gg\!= \mathit{either}\ h\ \mathit{return}
\end{array}
$$

We prefer *try* to *catch* for a simple reason: the axioms for *try* are more readable than the corresponding ones for *catch*.

2.4 Input and output

For reasons of space we restrict the I/O system sublanguage to the basic operations for terminal input and output.

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{InOut}\ m\ \textbf{where}\\
&\quad \mathit{out} :: \mathit{String} \rightarrow m\ ()\\
&\quad \mathit{inp} :: m\ \mathit{String}
\end{aligned}
$$

The operation *out s* writes *s* to the standard output device, *inp* reads a line of characters from the standard input device.

3 Examples

The following examples are intended to illustrate the use of the computational primitives giving a feel for the differences between Prolog and embedded Prolog. Some of these definitions employ so-called **do**-expressions which provide a more readable, first-order syntax for $(\ggg\!=)$ and $(\gg)$. The syntax and semantics of **do**-expressions are given by the following identities:[a]

$$
\begin{aligned}
&\textbf{do}\ \{m; e\} &&= m \gg \textbf{do}\ \{e\}\\
&\textbf{do}\ \{p \leftarrow m; e\} &&= m \ggg\!= \lambda p \rightarrow \textbf{do}\ \{e\}\\
&\textbf{do}\ \{\textbf{let}\ bs; e\} &&= \textbf{let}\ bs\ \textbf{in}\ \textbf{do}\ \{e\}\\
&\textbf{do}\ \{e\} &&= e\ .
\end{aligned}
$$

3.1 Transitive closure

The famous textbook by Sterling and Shapiro [13] introduces Prolog's basic constructs using the running example of a 'Biblical family database'. Here is a small excerpt reformulated in monadic terms.

$$
\begin{aligned}
&\mathit{child} &&:: (\mathit{Backtr}\ m) \Rightarrow \mathit{String} \rightarrow m\ \mathit{String}\\
&\mathit{child}\ \texttt{"terach"} &&= \mathit{return}\ \texttt{"abraham"} \mid \mathit{return}\ \texttt{"nachor"} \mid \mathit{return}\ \texttt{"haran"}\\
&\mathit{child}\ \texttt{"abraham"} &&= \mathit{return}\ \texttt{"isaac"}\\
&\mathit{child}\ \texttt{"haran"} &&= \mathit{return}\ \texttt{"lot"} \mid \mathit{return}\ \texttt{"milcah"} \mid \mathit{return}\ \texttt{"yiscah"}\\
&\mathit{child}\ \texttt{"sarah"} &&= \mathit{return}\ \texttt{"isaac"}\\
&\mathit{child}\ \_ &&= \mathit{fail}
\end{aligned}
$$

[a] Note that p must be 'failure-free' in the second equation, see the Haskell 1.4 Report [12] for a precise definition. All the patterns we employ — variables and tuples of variables — do satisfy this constraint.

Note that the last equation of *child* explicitly states that persons not listed before have no children. This proviso is necessary since *child* `"lot"` would otherwise provoke a runtime error. Negative information of this kind is left implicit in the corresponding Prolog program.

Using the combinators defined in the previous sections we can define a couple of useful relations: by composing *child* with itself we obtain the *grandChild* relation.

$$\begin{aligned}
&\mathit{grandChild} :: (\mathit{Backtr}\ m) \Rightarrow \mathit{String} \rightarrow m\ \mathit{String}\\
&\mathit{grandChild} = \mathit{child} \circledcirc \mathit{child}
\end{aligned}$$

The example illustrates that the Kleisli composition corresponds to the relational composition if the underlying structure is a backtracking monad. Recall that the operator ($\oplus$) defines the union of two relations. Combining the two yields the transitive closure.

$$\begin{aligned}
&\mathit{descendant} && :: (\mathit{Backtr}\ m) \Rightarrow \mathit{String} \rightarrow m\ \mathit{String}\\
&\mathit{descendant} && = \mathit{transitiveClosure}\ \mathit{child}\\
&\mathit{transitiveClosure} && :: (\mathit{Backtr}\ m) \Rightarrow (a \rightarrow m\ a) \rightarrow (a \rightarrow m\ a)\\
&\mathit{transitiveClosure}\ m && = m \oplus (\mathit{transitiveClosure}\ m \circledcirc m)
\end{aligned}$$

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 factorising *transitiveClosure*.

$$\begin{aligned}
&\mathit{transitiveClosure}' && :: (\mathit{Backtr}\ m) \Rightarrow (a \rightarrow m\ a) \rightarrow (a \rightarrow m\ a)\\
&\mathit{transitiveClosure}'\ m && = (\mathit{return} \oplus \mathit{transitiveClosure}'\ m) \circledcirc m
\end{aligned}$$

Note that *transitiveClosure* and *transitiveClosure′* are generally not equivalent if m involves computational effects such as output.

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.

$$\mathit{child}\ \texttt{"abraham"} = \ldots \mid \mathit{return}\ \texttt{"terach"} \qquad \text{-- cycle}$$

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.

$$
\begin{array}{lll}
\mathit{transitiveClosure''} & :: & (\mathit{Exc}\ m, \mathit{Backtr}\ m, \mathit{Eq}\ a) \Rightarrow (a \to m\ a) \to (a \to m\ a) \\
\mathit{transitiveClosure''}\ m\ a & = & \mathit{aux}\ a\ [a] \\
\quad \textbf{where}\ \mathit{aux}\ b\ \mathit{as} & = & \textbf{do}\ c \leftarrow m\ b \\
 & & \qquad \textbf{if}\ c \in \mathit{as}\ \textbf{then}\ \mathit{raise}\ \texttt{"cycle"} \\
 & & \qquad\qquad \textbf{else}\ \mathit{return}\ c \mid \mathit{aux}\ c\ (c : \mathit{as})
\end{array}
$$

Note that the type of the function reflects the computational features involved in its definition.

3.2 Queens problem

A common technique employed by Prolog programmers is generate-and-test. One of the standard instances of this technique tackles the n queens problem: n queens must be placed on a standard chess board so that no two pieces are threatening each other under the rules of chess. 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.

$$
\begin{array}{lll}
\mathit{queens} & :: & (\mathit{Backtr}\ m) \Rightarrow \mathit{Int} \to m\ [\mathit{Int}] \\
\mathit{queens}\ n & = & \mathit{place}\ n\ [1\,..\,n]\ [\,]\ [\,]
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{place} & :: & (\mathit{Backtr}\ m) \Rightarrow \mathit{Int} \to [\mathit{Int}] \to [\mathit{Int}] \to [\mathit{Int}] \to m\ [\mathit{Int}] \\
\mathit{place}\ 0\ \mathit{rs}\ d_1\ d_2 & = & \mathit{return}\ [\,] \\
\mathit{place}\ i\ \mathit{rs}\ d_1\ d_2 & = & \textbf{do}\ (q, \mathit{rs}') \leftarrow \mathit{select}\ \mathit{rs} \\
 & & \qquad \mathit{guard}\ (q - i \notin d_1) \\
 & & \qquad \mathit{guard}\ (q + i \notin d_2) \\
 & & \qquad \mathit{qs} \leftarrow \mathit{place}\ (i-1)\ \mathit{rs}'\ (q - i : d_1)\ (q + i : d_2) \\
 & & \qquad \mathit{return}\ (q : \mathit{qs})
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{select} & :: & (\mathit{Backtr}\ m) \Rightarrow [a] \to m\ (a, [a]) \\
\mathit{select}\ [\,] & = & \mathit{fail} \\
\mathit{select}\ (a : x) & = & \mathit{return}\ (a, x) \\
 & & \mid\ \textbf{do}\ (b, x') \leftarrow \mathit{select}\ x; \mathit{return}\ (b, a : x')
\end{array}
$$

Finally, *guard* maps a Boolean value to the corresponding computation:

$$\begin{array}{l} \mathit{guard} \quad :: (\mathit{Backtr}\ m) \Rightarrow \mathit{Bool} \to m\ () \\ \mathit{guard}\ b = \textbf{if}\ b\ \textbf{then}\ \mathit{true}\ \textbf{else}\ \mathit{fail}\ . \end{array}$$

3.3 A simple tracer

The following two examples illustrate the combination of backtracking and interaction.

The function *trace* implements a simple trace facility based on the 4 port procedure model of Prolog [14]: *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.

$$\begin{array}{lll} \mathit{trace} & :: & (\mathit{Backtr}\ m, \mathit{InOut}\ m) \Rightarrow m\ a \to \mathit{String} \to m\ a \\ \mathit{trace}\ m\ \mathit{msg} & = & \textbf{do}\ (\mathit{say}\ \texttt{"call"} \mid \textbf{do}\ \mathit{say}\ \texttt{"fail"}; \mathit{fail}) \\ & & \quad a \leftarrow m \\ & & \quad (\mathit{say}\ \texttt{"exit"} \mid \textbf{do}\ \mathit{say}\ \texttt{"redo"}; \mathit{fail}) \\ & & \quad \mathit{return}\ a \\ \multicolumn{3}{l}{\quad \textbf{where}\ \mathit{say}\ s = \mathit{out}\ (s +\!\!+ \texttt{":␣"} +\!\!+ \mathit{msg} +\!\!+ \texttt{"\textbackslash n"})} \end{array}$$

Double use is made of the idiom $m_1 \gg (\iota_1 \mid \iota_2 \gg \mathit{fail}) \gg m_2$ which provides the key to *trace*'s definition. If m_1 succeeds ι_1 is executed and then m_2. If m_2 subsequently fails ι_1 is retried. This attempt, however, fails since ι_1 succeeds exactly once. In turn $\iota_2 \gg \mathit{fail}$ is executed which fails, as well, passing control to m_1. To summarize we have that ι_1 is executed on the way from m_1 to m_2 and ι_2 on the way back.

We may also assign a 'logical' meaning to *trace* using the axioms to be presented in Section 4. Given the definition of *tchild*

$$\begin{array}{l} \mathit{tchild} \quad :: (\mathit{Backtr}\ m, \mathit{InOut}\ m) \Rightarrow \mathit{String} \to m\ \mathit{String} \\ \mathit{tchild}\ a = \mathit{trace}\ (\mathit{child}\ a)\ (\texttt{"child␣"} +\!\!+ a) \end{array}$$

one can show, for instance, that *transitiveClosure tchild* "abraham" $\gg$ *fail* is equivalent to

$$\begin{array}{l} \quad \mathit{out}\ \texttt{"call:␣child␣abraham"} \gg \mathit{out}\ \texttt{"exit:␣child␣abraham"} \\ \gg \mathit{out}\ \texttt{"redo:␣child␣abraham"} \gg \mathit{out}\ \texttt{"fail:␣child␣abraham"} \\ \gg \mathit{fail}\ . \end{array}$$

3.4 A simple Prolog shell

The function *shell* constitutes a rudimentary, interactive Prolog shell: *shell m* displays the first solution of m and prompts the user for instructions — if m has no solutions it simply says "no." and stops. If the user types "y" the next solution is presented, otherwise it terminates.

$$
\begin{aligned}
&\mathit{shell} &&:: (\mathit{Backtr}\ m, \mathit{InOut}\ m, \mathit{Show}\ a) \Rightarrow m\ a \rightarrow m\ ()\\
&\mathit{shell}\ m &&= \mathbf{do}\ a \leftarrow m\\
& && \qquad\quad \mathit{out}\ (\mathit{show}\ a + \!\!+\ \texttt{"\textbackslash n"} + \!\!+\ \texttt{"more?␣"})\\
& && \qquad\quad s \leftarrow \mathit{inp}\\
& && \qquad\quad \mathit{guard}\ (s \not\equiv \texttt{"y"})\\
& && \quad\ \mid\ \mathit{out}\ \texttt{"no."}
\end{aligned}
$$

4 Axioms

This section is concerned with the axiomatic characterization of the proposed computational primitives. Using the axioms we will be able to answer questions like the following two.

Are $(\mathit{out}\ s \gg m) \mid n$ and $\mathit{out}\ s \gg (m \mid n)$ equivalent?

What is the meaning of $\mathit{sols}\ (\mathit{out}\ s \mid \mathit{raise}\ e)$?

First of all, the basic monadic operations must be related by

$$\mathit{return}\ a \ggg\!= k \ = k\ a \tag{1}$$

$$m \ggg\!= \mathit{return} \ = m \tag{2}$$

$$(m \ggg\!= k) \ggg\!= \ell = m \ggg\!= (\lambda a \rightarrow k\ a \ggg\!= \ell)\ . \tag{3}$$

The laws are easier to remember if we rephrase them in terms of the Kleisli composition.

$$f \circledcirc \mathit{return} \ = f \tag{1'}$$

$$\mathit{return} \circledcirc f \ = f \tag{2'}$$

$$f \circledcirc (g \circledcirc h) = (f \circledcirc g) \circledcirc h \tag{3'}$$

Now the monoidal structure becomes apparent: $(\circledcirc)$ is associative with return as its left and right unit.

4.1 Backtracking

Before listing the axioms for the backtracking primitives we should stress that our intention is to axiomatize sequential Prolog and not its abstract Horn logic core. We will see that Prolog has a much weaker theory, many of the laws which are sound from a logical point of view do not hold for Prolog. This is, of course, due to the use of backtracking, a depth-first search strategy which is known to be incomplete. However, it is because of this search strategy that the addition of other computational features such as interaction or exception handling makes sense: if a computation involves, for instance, interaction a strict left to right execution is indispensable.

Turning to the laws we have that *fail* and $(\shortmid)$ form a monoid:

$$
\begin{aligned}
\mathit{fail} \shortmid m &= m && (4)\\
m \shortmid \mathit{fail} &= m && (5)\\
(m \shortmid n) \shortmid o &= m \shortmid (n \shortmid o) \ . && (6)
\end{aligned}
$$

Generally, $(\shortmid)$ is neither commutative nor idempotent, that is to say order and multiplicity of solutions matters. Take as an extreme example the definition of Prolog's control construct *repeat*.

$$
\begin{aligned}
&\mathit{repeat} :: (\mathit{Backtr}\ m) \Rightarrow m\ ()\\
&\mathit{repeat} = \mathit{true} \shortmid \mathit{repeat}
\end{aligned}
$$

The call *repeat* generates the same solution infinitely often. Note the importance of the order of the alternatives: $\mathit{repeat} = \mathit{repeat} \shortmid \mathit{true}$ does not work. The example furthermore demonstrates that *true* is not a zero of $(\shortmid)$.

The next two equations are concerned with the interplay of $(\ggg\!=)$ with *fail* and $(\shortmid)$: *fail* is a left zero of $(\ggg\!=)$, and $(\ggg\!=)$ distributes leftward through $(\shortmid)$.

$$
\begin{aligned}
\mathit{fail} \ggg\!= k &= \mathit{fail} && (7)\\
(m \shortmid n) \ggg\!= k &= (m \ggg\!= k) \shortmid (n \ggg\!= k) && (8)
\end{aligned}
$$

We have paraphrased $m \shortmid n$ as 'try m first, if it fails try n'. This is, of course, an oversimplification: even if m succeeds it may be necessary to try n because a 'future' computation which depends on m fails. Equation 8 provides the missing link relating $(m \shortmid n) \ggg\!= k$ to a simple disjunction.

The dual properties do not hold: it is neither true that *fail* is a right zero of $(\ggg\!=)$ nor that $(\ggg\!=)$ distributes rightward through $(\shortmid)$. Here are two counterexamples: $\mathit{out}\ s \gg \mathit{fail}$ does not equal *fail*[b], and $(\mathit{out}\ s \gg m) \shortmid (\mathit{out}\ s \gg n)$ is not the same as $\mathit{out}\ s \gg (m \shortmid n)$. Note that the idiom $m \gg \mathit{fail}$ is known to Prolog programmers as a 'failure driven loop'.

[b] If this were true the tracer of Section 3.3 would make no sense.

Here are the laws for *once*.

$$\begin{aligned}
&\mathit{nothing} :: (\mathit{Monad}\ m) \Rightarrow m\ (\mathit{Maybe}\ a)\\
&\mathit{nothing} = \mathit{return}\ \mathit{Nothing}
\end{aligned}$$

$$\begin{aligned}
&\mathit{once}\ \mathit{fail} &&= \mathit{nothing} && (9)\\
&\mathit{once}\ (\mathit{return}\ a \mid m) &&= \mathit{return}\ (\mathit{Just}\ a) && (10)\\
&(\mathit{once}\ m \ggg\!= k) \mid n &&= \mathit{once}\ m \ggg\!= \lambda a \to k\ a \mid n && (11)\\
&\mathit{once}\ (\mathit{once}\ m \ggg\!= k) &&= \mathit{once}\ m \ggg\!= \lambda a \to \mathit{once}\ (k\ a) && (12)
\end{aligned}$$

Recall that *once* m succeeds exactly once, in other words it is a deterministic predicate. Using *once* we can characterize deterministic computations: m is deterministic iff $\mathit{once}\ m = \mathit{map}\ \mathit{Just}\ m$. Of course, *once* m is deterministic itself. To see this replace k by *return* in 12. Furthermore, Equation 10 implies that *return* a is deterministic. Finally, Equation 11 shows that a deterministic computation can be pushed out of a disjunction.

The solution collecting primitive *sols* satisfies a similar set of equations. To explain the interplay of *sols* with *fail* and $(\mid)$ some definitions are helpful.

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

The operator $(\oplus)$ essentially lifts $(+\!\!+)$ to the level of computations. Note that *nil* is a unit of $(\oplus)$, and that $(\oplus)$ is associative.

$$\begin{aligned}
&\mathit{sols}\ \mathit{fail} &&= \mathit{nil} && (13)\\
&\mathit{sols}\ (\mathit{return}\ a) &&= \mathit{return}\ [a] && (14)\\
&\mathit{sols}\ (m \mid n) &&= \mathit{sols}\ m \oplus \mathit{sols}\ n && (15)\\
&\mathit{sols}\ (\mathit{once}\ m \ggg\!= k) &&= \mathit{once}\ m \ggg\!= \lambda a \to \mathit{sols}\ (k\ a) && (16)\\
&\mathit{once}\ (\mathit{sols}\ m) &&= \mathit{map}\ \mathit{Just}\ (\mathit{sols}\ m) && (17)
\end{aligned}$$

Equation 16 is similar to 11 and 12; it states that a deterministic computation may be pushed out of a call to *sols*. Equation 17 identifies *sols* as deterministic.

4.2 Exception handling

This section lists properties of the exception handling primitives. We will see that the interplay of exception handling and backtracking is particularly interesting.

The operation *raise e* is expected to satisfy the following laws.

$$\mathit{raise}\ e \ggg k \quad = \mathit{raise}\ e \tag{18}$$

$$\mathit{once}\ (\mathit{raise}\ e) = \mathit{raise}\ e \tag{19}$$

Signaling an error terminates the current computation: *raise e* is a left zero of bind. Equation 19 lays down that *raise e* propagates through *once*. Despite appearance 19 has far reaching consequences: it means that *raise e* is deterministic and it implies both $\mathit{raise}\ e \mid m = \mathit{raise}\ e$ and $\mathit{sols}\ (\mathit{raise}\ e) = \mathit{raise}\ e$. The former equation shows that *raise e* ignores further choice points. Here is a simple calculational proof of the claim.

$$\begin{aligned}
\mathit{raise}\ e \mid m &= (\mathit{raise}\ e \ggg k) \mid m && \text{by } 18\\
&= (\mathit{once}\ (\mathit{raise}\ e) \ggg k) \mid m && \text{by } 19\\
&= \mathit{once}\ (\mathit{raise}\ e) \ggg \lambda a \to k\ a \mid m && \text{by } 11\\
&= \mathit{raise}\ e \ggg \lambda a \to k\ a \mid m && \text{by } 19\\
&= \mathit{raise}\ e && \text{by } 18
\end{aligned}$$

The *try* construct plays a similar rôle for exception handling as *once* for backtracking. For that reason we obtain similar laws.

$$\mathit{try}\ \mathit{fail} \qquad\qquad = \mathit{fail} \tag{20}$$

$$\mathit{try}\ (\mathit{return}\ a \mid m) = \mathit{return}\ (\mathit{Right}\ a) \mid \mathit{try}\ m \tag{21}$$

$$\mathit{try}\ (\mathit{raise}\ e) \qquad = \mathit{return}\ (\mathit{Left}\ e) \tag{22}$$

$$\mathit{try}\ (\mathit{try}\ m \ggg k) \ = \mathit{try}\ m \ggg \lambda a \to \mathit{try}\ (k\ a) \tag{23}$$

The laws have been chosen to conform to the Prolog ISO standard [2]. In particular Equation 21 lays down, that *try* is re-satisfiable. This implies that *try m* behaves as *map Right m* provided *m* does not raise an exception. Using *try* we can characterize *safe* computations, ie computations which do not raise an exception: *m* is safe iff $\mathit{try}\ m = \mathit{map}\ \mathit{Right}\ m$. Having said that we see that Equation 23 formalizes that a safe computation can be pushed out of a call to *try*.

4.3 Input and output

For simplicity let us assume that I/O actions are deterministic and safe. Let ι be an I/O operation such as *out s* or *inp* then

$$\mathit{once}\ \iota = \mathit{map}\ \mathit{Just}\ \iota \tag{24}$$

$$\mathit{try}\ \iota \quad = \mathit{map}\ \mathit{Right}\ \iota \tag{25}$$

are expected to hold. The second assumption is probably debatable: *inp* typically raises an exception if the end of the input is reached. We can take this into account simply by dropping 25 for $\iota = inp$.

Certain I/O actions may require additional laws. The following two equations, for instance, further constrain *out*.

$$out\ \texttt{""} = return\ () \tag{26}$$

$$out\ (s_1 +\!\!+\ s_2) = out\ s_1 \gg out\ s_2 \tag{27}$$

Now the time has come to answer the questions posed in the introduction to this section. Using $out\ s \gg m = once\ (out\ s) \gg m$ and 11 we see that $(out\ s \gg m) \mid n$ and $out\ s \gg (m \mid n)$ are indeed equivalent. Furthermore we can simplify $sols\ (out\ s \mid raise\ e)$ to $out\ s \gg raise\ e$ using a straightforward calculation.

$$\begin{aligned} sols\ (out\ s \mid raise\ e) &= sols\ (out\ s) \oplus sols\ (raise\ e) && \text{by 15} \\ &= (out\ s \gg return\ [()]) \oplus raise\ e && \text{by 16} \\ &= out\ s \gg raise\ e && \text{def. } (\oplus) \text{ and by 18} \end{aligned}$$

4.4 Monad morphisms

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

$$\eta\ (return\ a) = return\ a \tag{28}$$

$$\eta\ (m \gg\!= k) = \eta\ m \gg\!= (\eta \circ k) \tag{29}$$

A function that satisfies Equation 28 only is termed premonad morphism. We will see that monad morphisms play a central rôle in adding features to a monad, see Section 5. Of course, if we extend a backtracking monad we will require that the backtracking operations are preserved as well:

$$\eta\ fail = fail \tag{30}$$

$$\eta\ (m \mid n) = \eta\ m \mid \eta\ n\ . \tag{31}$$

4.5 Encapsulation

Which laws are *run* and *perform* supposed to satisfy? Both forget computational structure. The only thing we can realistically expect from *run* is that it maps *return a* to *a*. To put it abstractly *run* should be a premonad morphism from m to Id, the identity monad (cf Section 5.1). Accordingly, *perform* must be a premonad morphism from m to IO. Additionally, we require *perform* to preserve I/O actions: let ι be an I/O operation, then

$$perform\ (\iota \ggg\!= k) = \iota \ggg\!= (perform \circ k) \tag{32}$$

must hold.

5 Monad transformers

Each of the class definitions listed in Section 2 defines a certain 'feature' or computational behaviour. 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 [5]: 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 [10] 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 $\tau\ m$.

$$\begin{array}{l}
\mathbf{class}\ MonadT\ \tau\ \mathbf{where} \\
\quad up \quad\ \ :: (Monad\ m) \Rightarrow m\ a \rightarrow \tau\ m\ a \\
\quad down :: (Monad\ m) \Rightarrow \tau\ m\ a \rightarrow m\ a
\end{array}$$

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. Since τ adds structure it is natural to require *up* to be a monad morphism and *down* to be a premonad morphism.

5.1 Base monads

The identity monad and the IO monad serve as base monads, upon which the monad transformers are applied. The identity monad is given by the following definition.[c]

[c]Note that we omit data constructors of **newtype** definitions to improve readability.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Id}\ a = a \\
\textbf{instance}\ \mathit{Monad}\ \mathit{Id}\ \textbf{where} \\
\quad \mathit{return}\ a \quad = a \\
\quad a \ggg\!= f \quad = f\ a \\
\textbf{instance}\ \mathit{Run}\ \mathit{Id}\ \textbf{where} \\
\quad \mathit{run} \quad = \mathit{id} \\
\textbf{instance}\ \mathit{Perform}\ \mathit{Id}\ \textbf{where} \\
\quad \mathit{perform} \quad = \mathit{return}
\end{array}
$$

Note that bind is equivalent to the reverse function application. Since the Kleisli composition boils down to the ordinary composition of functions it is immediate that the monad laws, 1–3, are satisfied.

The predefined monad of interactions, *IO*, is declared an instance of *Perform* and *InOut*.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Perform}\ \mathit{IO}\ \textbf{where} \\
\quad \mathit{perform} = \mathit{id} \\
\textbf{instance}\ \mathit{InOut}\ \mathit{IO}\ \textbf{where} \\
\quad \mathit{out} \quad = \mathit{putStr} \\
\quad \mathit{inp} \quad = \mathit{getLine}
\end{array}
$$

5.2 Exception monad transformer

In an exception monad a computation either succeeds gracefully or aborts with an exception. These two possible outcomes may be represented using the predefined type constructor *Either*, which is already employed in *try*'s type. 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. The exception monad transformer simply composes the base monad with *Either Err*.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{ExcT}\ m\ a = m\ (\mathit{Either}\ \mathit{Err}\ a) \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{ExcT}\ m)\ \textbf{where} \\
\quad m \ggg\!= k \quad = m \ggg\!= \mathit{either}\ (\mathit{return} \circ \mathit{Left})\ k \\
\quad \mathit{return} \quad = \mathit{return} \circ \mathit{Right} \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Exc}\ (\mathit{ExcT}\ m)\ \textbf{where} \\
\quad \mathit{raise} \quad = \mathit{return} \circ \mathit{Left} \\
\quad \mathit{try}\ m \quad = \mathit{map}\ \mathit{Right}\ m
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{instance } \mathit{MonadT}\ \mathit{ExcT} \textbf{ where} \\
\quad \mathit{up}\ m \quad\ \ = \mathit{map}\ \mathit{Right}\ m \\
\quad \mathit{down}\ m = m \ggg\!= \mathit{either}\ \mathit{error}\ \mathit{return} \\
\textbf{instance } (\mathit{Run}\ m) \Rightarrow \mathit{Run}\ (\mathit{ExcT}\ m) \textbf{ where} \\
\quad \mathit{run} \qquad = \mathit{run} \circ \mathit{down} \\
\textbf{instance } (\mathit{Perform}\ m) \Rightarrow \mathit{Perform}\ (\mathit{ExcT}\ m) \textbf{ where} \\
\quad \mathit{perform} = \mathit{perform} \circ \mathit{down}
\end{array}
$$

It is straightforward to show that *up* is a monad morphism and *down* a pre-monad morphism. The morphism *up* maybe used to lift the I/O operations through the exception monad transformer.

$$
\begin{array}{l}
\textbf{instance } (\mathit{InOut}\ m) \Rightarrow \mathit{InOut}\ (\mathit{ExcT}\ m) \textbf{ where} \\
\quad \mathit{out} = \mathit{up} \circ \mathit{out} \\
\quad \mathit{inp} = \mathit{up}\ \mathit{inp}
\end{array}
$$

5.3 Backtracking monad transformer

Virtually all publications on monads employ lists for implementing nondeterministic computations. The list monad, however, suffers at least from three problems: (1) It relies in an essential way on lazy evaluation. Thus we cannot use it in a strict language like Standard ML. (2) It is inefficient: both $(\ggg\!=)$ and $(\mid)$ are sensitive to the number of successes of their first argument. (3) There is no obvious way to turn it into a monad transformer.[d] In the sequel we will derive an alternative which remedies these shortcomings.

Part of the functional programming folklore is a transformation technique for eliminating expensive calls to $(+\!\!+)$. The technique is, for instance, applied in Haskell's *Show* class to guarantee linear complexity of the *show* functions. It is not hard to generalize this technique to an arbitrary backtracking monad: the essential idea is to transform a computation m into a function m' such that $m'\ f = m \mid f$ holds. An efficient variant of m is then obtained by passing *fail* to m', ie $m = m'\ \mathit{fail}$. Note that the type of m' is $m\ a \to m\ a$. The question naturally arises as to whether it is possible to turn $m\ a \to m\ a$ into a monad. Now, if this is the case then the mappings

$$
\begin{array}{l}
\mathit{up}\ m \quad\ \ = \lambda f \to m \mid f \\
\mathit{down}\ m = m\ \mathit{fail}
\end{array}
$$

[d] Setting $\mathit{ListT}\ m\ a = m\ [a]$ appears to work only if m is a so-called commutative monad[15]; otherwise property 8 does not hold.

must be monad morphisms. Furthermore, since we do not intend to add structure to m we can restrict ourselves to m's isomorphic copy in $m\ a \to m\ a$. It is not hard to see that *up* and *down* are mutually inverse iff

$$m\ f = m\ \mathit{fail} \mid f \tag{33}$$

holds. Using these prerequisites we can actually *derive* the monad operations of $m\ a \to m\ a$. For reasons of space we only show the calculation of $(\mid)$.

$$\begin{aligned}
m \mid n &= \mathit{up}\ (\mathit{down}\ (m \mid n)) && \mathit{up} \circ \mathit{down} = \mathit{id} \\
&= \mathit{up}\ (\mathit{down}\ m \mid \mathit{down}\ n) && \text{by } 31 \\
&= \lambda f \to (m\ \mathit{fail} \mid n\ \mathit{fail}) \mid f && \text{def. } \mathit{up}/\mathit{down} \\
&= \lambda f \to m\ \mathit{fail} \mid (n\ \mathit{fail} \mid f) && \text{by } 6 \\
&= \lambda f \to m\ (n\ f) && \text{by } 33
\end{aligned}$$

We see that $(\mid)$ boils down to the composition of functions. A similar calculation demonstrates that *fail* equals the identity function. Interestingly, neither operation depends on m's primitives. However, this does not hold for return and bind. The latter operation, for instance, is given by

$$m \ggg k = \lambda f \to (m\ \mathit{fail} \ggg \lambda a \to k\ a\ \mathit{fail}) \mid f \quad .$$

We can get rid of these dependencies via a second transformation which abstracts $(\ggg)$ away: each computation n is transformed into a function n' such that $n'\ k = n \ggg k$ holds. Going back we apply n' to *return*, ie $n = n'\ \mathit{return}$. This gives us two morphisms

$$\begin{aligned}
\mathit{up}\ n &= \lambda \kappa \to n \ggg \kappa \\
\mathit{down}\ n &= n\ \mathit{return}
\end{aligned}$$

and an isomorphism condition

$$n\ \kappa = n\ \mathit{return} \ggg \kappa \quad . \tag{34}$$

Again, we can use these definitions to *derive* the monad operations. We restrict ourselves to the calculation of bind.

$$\begin{aligned}
& m \ggg k \\
={}& \mathit{up}\ (\mathit{down}\ (m \ggg k)) && \mathit{up} \circ \mathit{down} = \mathit{id} \\
={}& \mathit{up}\ (\mathit{down}\ m \ggg \mathit{down} \circ k) && \text{by } 29 \\
={}& \lambda \kappa \to (m\ \mathit{return} \ggg \lambda a \to k\ a\ \mathit{return}) \ggg \kappa && \text{def. } \mathit{up}/\mathit{down} \\
={}& \lambda \kappa \to m\ \mathit{return} \ggg \lambda a \to k\ a\ \mathit{return} \ggg \kappa && \text{by } 3 \\
={}& \lambda \kappa \to m\ (\lambda a \to k\ a\ \kappa) && \text{by } 34
\end{aligned}$$

A similar calculation shows that *return* a amounts to $\lambda\kappa \to \kappa\ a$. Before defining the backtracking monad transformer let us first determine its type. Recall that $(\gg\!=)$ possesses the type $\forall a.\ \forall b.\ n\ a \to (a \to n\ b) \to n\ b$ which is equivalent to $\forall a.\ n\ a \to \forall b.\ (a \to n\ b) \to n\ b$. Since *up* n equals $(\gg\!=)\ n$ it consequently has the type $\forall b.\ (a \to n\ b) \to n\ b$. Taking the first abstraction step into account which yields $n\ a = m\ a \to m\ a$ we arrive at the following definition.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{BacktrT}\ m\ a = \forall \mathit{ans}.(a \to m\ \mathit{ans} \to m\ \mathit{ans}) \to m\ \mathit{ans} \to m\ \mathit{ans} \\
\textbf{instance}\ \mathit{Monad}\ (\mathit{BacktrT}\ m)\ \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{return}\ a & = \lambda\kappa \to \kappa\ a \\
m \gg\!= k & = \lambda\kappa \to m\ (\lambda a \to k\ a\ \kappa)
\end{array}
\end{array}
$$

The cognoscenti has certainly recognized that the above definition is identical with the definition of the continuation monad transformer [10]. Only the types are different: *BacktrT* involves second-order types[e] while the continuation monad transformer is additionally parameterized with the answer type ($\mathit{ContT}\ \mathit{ans}\ m\ a = (a \to m\ \mathit{ans}) \to m\ \mathit{ans}$). We have seen, however, that second-order types arise naturally from the second abstraction step. In a sense *BacktrT* m constitutes the smallest extension of m which allows to add backtracking. Note, for instance, that *callcc* is definable in *ContT ans* m but not in *BacktrT* m.

The backtracking monad transformer, *BacktrT* m, has the amazing property of being a monad regardless of m! The dependence on the base monad has completely vanished. Nearly the same holds for the *Backtr* instance: m is only required to be a monad.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Backtr}\ (\mathit{BacktrT}\ m)\ \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{fail} & = \lambda\kappa \to \mathit{id} \\
m \mid n & = \lambda\kappa \to m\ \kappa \circ n\ \kappa \\
\mathit{once}\ m & = \lambda\kappa\ f \to m\ \mathit{first}\ \mathit{nothing} \gg\!= \lambda x \to \kappa\ x\ f \\
\mathit{sols}\ m & = \lambda\kappa\ f \to m\ \mathit{cons}\ \mathit{nil} \gg\!= \lambda x \to \kappa\ x\ f
\end{array} \\
\begin{array}{ll}
\mathit{first} & :: (\mathit{Monad}\ m) \Rightarrow a \to m\ (\mathit{Maybe}\ a) \to m\ (\mathit{Maybe}\ a) \\
\mathit{first}\ a\ \_ & = \mathit{return}\ (\mathit{Just}\ a) \\
\mathit{cons} & :: (\mathit{Monad}\ m) \Rightarrow a \to m\ [a] \to m\ [a] \\
\mathit{cons}\ a\ \mathit{mx} & = \textbf{do}\ x \leftarrow \mathit{mx}; \mathit{return}\ (a : x)
\end{array}
\end{array}
$$

The definitions for *once* and *sols* are explained as follows. A computation in *BacktrT* m, say, n takes two arguments: a success continuation and a failure

---

[e] Haskell 1.4 does not admit local universal quantification. All major Haskell systems, however, provide the necessary extensions. We refer the interested reader to the work of Remy [16] and Jones [17].

continuation. If n succeeds it calls the success continuation passing it the computed value and the current failure continuation (cf *return*). If it fails it simply calls the failure continuation (cf *fail*). Both *once* and *sols* are implemented by supplying special success and failure continuations to their argument. The success continuation *first*, for example, discards the failure continuation and returns the first solution; *cons* on the other hand executes the failure continuation and constructs the desired list of solutions.

Turning to the definitions of *up* and *down* we should note that *up* is *not* obtained by composing the two *up*s introduced above since we no longer require m to be a backtracking monad. The same argument applies to *down*.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{MonadT}\ \mathit{BacktrT}\ \textbf{where} \\
\quad \mathit{up}\ m \qquad = \lambda\kappa\ f \rightarrow m \gg\!\!= \lambda a \rightarrow \kappa\ a\ f \\
\quad \mathit{down}\ m = m\ (\lambda a\ \_ \rightarrow \mathit{return}\ a)\ (\mathit{error}\ \texttt{"no\textvisiblespace solution"})
\end{array}
$$

Nevertheless *up* is a monad morphism and *down* a premonad morphism. We omit the instance declarations for *Run*, *Perform*, and *InOut* since they are the same as for *ExcT*.[f]

Note that *BacktrT* remedies the shortcomings of the list monad mentioned in the introduction to this section.[g] (1) It is independent of the order of evaluation. (2) It is efficient: both $(\gg\!\!=)$ and $(\mid)$ take constant time. (3) It adds backtracking to an arbitrary monad.

It remains to lift exception handling through the backtracking monad transformer. Let us first implement a non re-satisfiable variant of *try*. 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*.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Exc}\ m) \Rightarrow \mathit{Exc}\ (\mathit{BacktrT}\ m)\ \textbf{where} \\
\quad \mathit{raise}\ \ = \mathit{up} \circ \mathit{raise} \\
\quad \mathit{try}\ m = \lambda\kappa\ f \rightarrow \textbf{let}\ \mathit{decode}\ (\mathit{Left}\ e) = \kappa\ (\mathit{Left}\ e)\ f \\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{decode}\ (\mathit{Right}\ \mathit{Nothing}) = f \\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{decode}\ (\mathit{Right}\ (\mathit{Just}\ a)) = \kappa\ (\mathit{Right}\ a)\ f \\
\qquad\qquad\qquad\quad\ \textbf{in}\ \mathit{try}\ (m\ \mathit{first}\ \mathit{nothing}) \gg\!\!= \mathit{decode}
\end{array}
$$

The call *try* (*m first nothing*) performs the encodings; the function *decode* then decodes the results and calls the appropriate continuations. In order to make

[f] In fact, they are identical for all monad transformers. However, Haskell does not allow to share code by giving instance declarations like **instance** $(\mathit{Run}\ m, \mathit{MonadT}\ \tau) \Rightarrow \mathit{Run}\ (\tau\ m)$.
[g] Interestingly, both approaches are not unrelated: the backtracking monad *BacktrT Id a* $(= \forall \mathit{ans}.\,(a \rightarrow \mathit{ans} \rightarrow \mathit{ans}) \rightarrow \mathit{ans} \rightarrow \mathit{ans})$ corresponds exactly to the type of the fold-functional [18] for lists which in turn describes the encoding of lists in the second-order λ-calculus [17].

try m re-satisfiable we must pass the failure continuation, which *first* simply discards, to *decode*.

$$
\begin{array}{l}
\textbf{instance } (\mathit{Exc}\ m) \Rightarrow \mathit{Exc}\ (\mathit{BacktrT}\ m)\ \textbf{where} \\
\quad \mathit{raise} \ \ = \mathit{up} \circ \mathit{raise} \\
\quad \mathit{try}\ m = \lambda\kappa\, f \rightarrow \\
\qquad\qquad \textbf{let } \mathit{decode}\ (\mathit{Left}\ e) = \kappa\ (\mathit{Left}\ e)\ f \\
\qquad\qquad\quad\ \ \mathit{decode}\ (\mathit{Right}\ \mathit{Nothing}) = f \\
\qquad\qquad\quad\ \ \mathit{decode}\ (\mathit{Right}\ (\mathit{Just}\ (a, f'))) = \kappa\ (\mathit{Right}\ a)\ f' \\
\qquad\qquad \textbf{in } \mathit{try}\ (m\ (\lambda a\ f' \rightarrow \mathit{return}\ (\mathit{Just}\ (a, \mathit{try}\ f' \gg\!= \mathit{decode})))\ \mathit{nothing}) \\
\qquad\qquad\quad\ \ \gg\!= \mathit{decode}
\end{array}
$$

The code shows that the decoder function *decode* is applied 'recursively' to the failure continuation: $\mathit{try}\ f' \gg\!= \mathit{decode}$. This is necessary since f' may, of course, raise exceptions which must be trapped.

Finally, we are able to construct the monads required in the applications of Section 3: *BacktrT Id* supports backtracking, *BacktrT* (*ExcT IO*) supports backtracking, exception handling, and interaction. Furthermore the implementations are correct wrt the axiomatization given in Section 4. A proof of this claim appears in [19]. As a final remark note that we may view *BacktrT* (*ExcT IO*) as a *standard model for embedded Prolog.*

6 Benchmarks

The purpose of this section is to compare the efficiency of Haskell's simulation of backtracking to that of logic languages like Prolog or Mercury [11] which offer non-determinism as a 'language primitive'. To get a somewhat broader picture we also include C and Standard ML [20] in the competition. Due to the multilingual nature of the competition we will consider only a toy example: for a given n the number of solutions to the n queens problem must be calculated. Of course, a single benchmark does not allow to establish a ranking of the different systems; if at all it may indicate general tendencies.

The source code of the programs appears in [19]. A few comments, however, are appropriate: the C program employs bit fields to record the queens' positions. Two Prolog programs participate in the contest: the first is a transliteration of the Haskell code. The second which is due to Thom Frühwirth employs unification instead of arithmetic operations to check for attacking queens. The Mercury implementation adds type, mode, and determinism declarations to the first Prolog variant. The Haskell 1.2 program is based on the program of Section 3.2 specialized to *BacktrT ID*, ie overloading is resolved to improve efficiency.

Table 1: Benchmark results

#queens	13		14	
#solutions	73712		365596	
C (`gcc-2.7.2 -O`)	4.0[a]	1.0[b]	27.0	1.0
Prolog (`eclipse-3.5.2`)	504.5	126.1	3173.7	117.5
ditto using logical variables	226.0	56.5	1401.0	51.9
Mercury (`mercury-0.6 -O6`)	153.0	38.3	1113.0	41.2
Haskell 1.2 (`ghc-0.29 -O`)	154.4	38.6	974.0	36.1
ditto using bit fields	68.9	17.2	415.8	15.4
Haskell 1.4 (`ghc-2.10 -O`)	619.1	154.8	3809.4	141.1
Standard ML (`sml-1.10`)	650.4	162.6	4046.5	149.9

[a]seconds of user time on a Sun Ultrasparc-1, 64 MB (minimum of four runs)
[b]time relative to the fastest implementation

Table 1 lists the results of the benchmark. Among the different languages C is the clear winner. The Haskell 1.2 implementation is a factor of 40 slower, the factor reduces to 15 if bit fields are used. Perhaps surprisingly, Haskell — or rather, the Glasgow Haskell Compiler [21] compares favourably with Prolog and Mercury. It lies even in front of Standard ML of New Jersey [22]. 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 [23] relates the Glasgow Haskell Compiler to other systems. Compared to Haskell 1.2 the 1.4 code as presented above is a factor of 4 slower. The loss of performance is partly due to the use of type classes and partly due to the fact that the Haskell 1.4 compiler is not yet as good as its predecessor.

7 Related and future work

Implementing backtracking using continuation-passing style (CPS) is by no means a new idea. An early account appears in [24] where CPS is used to integrate Prolog into the LISP dialect POP-11. Danvy and Filinski [25] show that backtracking can be implemented using two CPS conversions. Interestingly, both approaches can be recast in monadic terms and lead after some simplifications to the structure presented in Section 5.3 [19]. Success continuations are also employed in the Lisp-based Prolog interpreter of Carlsson [26].

The axiomatic approach is inspired by the work of Wadler [9] who identifies properties of functional parsers. To the best of our knowledge, the treatment of control and all solution predicates is original. However, a lot of work remains to be done: it is not clear, for instance, whether the given set of axioms is independent or complete. The existence of a standard model at least shows that it is consistent.

Acknowledgments

The author would like to thank the anonymous referees for many valuable comments on this paper.

References

1. Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
2. *PROLOG. Part 1, General Core, Committee Draft 1.0.* Number 92. ISO/IEC JTC1 SC22 WG17, 1992.
3. Bijan Arbab and Daniel M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4(4):309–329, December 1987.
4. Egon Börger and Dean Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, June 1995.
5. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dep. of Comp. Science, Edinburgh University, 1990.
6. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
7. Philip Wadler. How to declare an imperative (invited paper). In John Lloyd, editor, *ILPS'95: International Logic Programming Symposium*. MIT Press, December 1995.
8. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, Charleston, South Carolina, January 1993.
9. Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, Proceedings of the Båstad Spring School*, number 925 in Lecture Notes in Computer Science. Springer Verlag, May 1995.
10. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343, January 1995.
11. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.

12. J. Peterson and K. Hammond. Report on the programming language Haskell 1.4, a non-strict, purely functional language. Research Report YALEU/DCS/RR-1106, Yale University, Dep. of Comp. Science, March 1997.
13. Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.
14. C.S. Mellish and W.F. Clocksin. *Programming in Prolog*. Springer Verlag, Berlin, fourth edition edition, 1995.
15. Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Dep. of Comp. Science, Yale University, December 1993.
16. Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.
17. Mark P. Jones. First-class polymorphism with type inference. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 15–17 January 1997.
18. T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.
19. Ralf Hinze. Efficient monadic-style backtracking. Technical Report IAI-TR-96-9, Institut für Informatik III, Universität Bonn, October 1996.
20. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
21. Simon L Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele*, 1993.
22. A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP)*, number 528 in Lecture Notes in Computer Science, pages 1–13. Springer Verlag, 1991.
23. P. H. Hartel. Benchmarking implementations of lazy functional languages II – two years later. Technical Report CS-94-21, Dept. of Comp. Sys, Univ. of Amsterdam, December 1994.
24. Chris Mellish and Steve Hardy. Integrating Prolog into the Poplog environment. In J.A. Campbell, editor, *Implementations of Prolog*, pages 533–535. Ellis Horwood Limited, 1984.
25. Olivier Danvy and Andrzey Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, June 1990, Nice*, pages 151–160. ACM-Press, June 1990.
26. Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2:347–359, 1984.