International Journal of Foundations of Computer Science
© World Scientific Publishing Company

PROLOG'S CONTROL CONSTRUCTS IN A FUNCTIONAL SETTING — AXIOMS AND IMPLEMENTATION

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

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The purpose of this article is twofold. First, we show that Prolog's control constructs can be smoothly integrated into a functional language like Haskell. The resulting 'language', termed embedded Prolog, incorporates many of the features prescribed by the Prolog ISO standard: control constructs including the cut, all solution collecting functions, and error handling facilities. 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 us 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. The axiomatization is applied to show that each finite computation has a simple canonical form.

Keywords: Prolog, Haskell, axiomatic semantics, monads, monad transformers, continuations.

1. Introduction

Many proposals have been made for the integration of functional and logic programming, see [13] 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 constructs — into Haskell [37]. 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 Prolog ISO standard [19]: control constructs including the cut, all solution collecting functions, and error handling fa-

cilities. 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 us to write self-modifiable programs. The manipulation of a global state, however, does not pose any problems. 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 5.2).

Some attempts have been made to explain the semantics of full Prolog, both denotational [3] and operational [6]. 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 Haskell is employed 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 [32, 33] as a means to structure denotational semantics. Monads have received a great deal of attention in the functional programming community since then. Wadler [45] 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 [36], which provides interaction with the environment, serves as a good example of the latter type. Examples of internal uses are given in [46]. 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 Liang, Hudak, and Jones [25] 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 that 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 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 that 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 this article is the definition of a backtracking monad transformer, which adds backtracking to an arbitrary monad. Among other things it allows us to study the interplay between 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 us to turn an embedded Prolog program into a pure function.

The article is organized as follows. Section 2 reviews Haskell's predefined types and gives a short account of constructor classes. Section 3 defines monads and introduces the different constructs of the embedded language. For each linguistic feature — nondeterminism, cut, exception handling, and interaction — a subclass of the basic monadic class is given. Section 4 presents several motivating examples demonstrating among other things the use of higher order computations. An axiomatization of the computational primitives is discussed in Section 5. The axiomatization is applied to show that each finite computation has a simple canonical form. Section 6 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 [41], which offer nondeterminism as a 'language primitive'. Section 7 provides some evidence for this claim. Finally, in Section 8 we relate our work to other approaches.

The programs are written in Haskell 98 [37], a recent consolidation of the language, using a few extensions of the class and of the type system. The use of extensions will be announced as we go along.

2. Preliminaries

This section briefly reviews the predefined data types $[\,]$, *Maybe*, and *Either* and gives a short account of Haskell's constructor classes [20]. The cognoscenti may safely skip this section.

The ubiquitous datatype of polymorphic lists is given by the following datatype declaration.

$$
\begin{array}{lcl}
\textbf{data}\ [a] & = & [\,] \mid a : [a] \\
\mathit{list} & :: & b \to (a \to b \to b) \to [a] \to b \\
\mathit{list}\ \mathit{nil}\ \mathit{cons}\ [\,] & = & \mathit{nil} \\
\mathit{list}\ \mathit{nil}\ \mathit{cons}\ (a : \mathit{as}) & = & \mathit{cons}\ a\ (\mathit{list}\ \mathit{nil}\ \mathit{cons}\ \mathit{as})
\end{array}
$$

The function *list* is the so-called *fold-functional* [40], also known as catamorphism [27], of the list datatype (in Haskell *list* is predefined as *foldr*). It implements a general recursion scheme on lists. Essentially, *list* replaces the list constructors $[\,]$ and (:) by functions of appropriate types.

The datatype *Maybe* represents optional values of type a; its fold-functional is given by *maybe*.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Maybe}\ a & = & \mathit{Nothing} \mid \mathit{Just}\ a \\
\mathit{maybe} & :: & b \to (a \to b) \to \mathit{Maybe}\ a \to b \\
\mathit{maybe}\ \mathit{nothing}\ \mathit{just}\ \mathit{Nothing} & = & \mathit{nothing} \\
\mathit{maybe}\ \mathit{nothing}\ \mathit{just}\ (\mathit{Just}\ a) & = & \mathit{just}\ a
\end{array}
$$

Finally, the datatype *Either* implements the disjoint union of two types; its fold-functional is accordingly termed *either*.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Either}\ \ell\ r & = & \mathit{Left}\ \ell \mid \mathit{Right}\ r \\
\mathit{either} & :: & (\ell \to a) \to (r \to a) \to \mathit{Either}\ \ell\ r \to a \\
\mathit{either}\ \mathit{left}\ \mathit{right}\ (\mathit{Left}\ a) & = & \mathit{left}\ a \\
\mathit{either}\ \mathit{left}\ \mathit{right}\ (\mathit{Right}\ a) & = & \mathit{right}\ a
\end{array}
$$

Haskell's constructor classes support abstraction of common features among type constructors. They are best explained by means of an example. Functional programmers often use the function *map* to apply a function to each element of a given list.

$$\mathit{map} \quad :: \quad (a \to b) \to ([a] \to [b])$$

This so-called *mapping function* can be defined for arbitrary polymorphic datatypes. For instance, the mapping function on *Maybe* is

$$
\begin{array}{lcl}
\mathit{mapMaybe} & :: & (a \to b) \to (\mathit{Maybe}\ a \to \mathit{Maybe}\ b) \\
\mathit{mapMaybe}\ f\ \mathit{Nothing} & = & \mathit{Nothing} \\
\mathit{mapMaybe}\ f\ (\mathit{Just}\ a) & = & \mathit{Just}\ (f\ a)\ .
\end{array}
$$

All mapping functions have a similar type and functionality. By introducing a constructor class we can capture this common idiom.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\
\quad \mathit{map} \quad :: \quad (a \to b) \to (f\ a \to f\ b)
\end{array}
$$

In *map*'s type the concrete type constructors, [] and *Maybe*, have been replaced by a type variable, which additionally appears as an argument of the class *Functor*. By giving so-called instance declarations we can make [] and *Maybe* instances of that class.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ []\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{map}\ f\ [] & = & [] \\
\mathit{map}\ f\ (a : as) & = & f\ a : \mathit{map}\ f\ as
\end{array} \\
\textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{map}\ f\ \mathit{Nothing} & = & \mathit{Nothing} \\
\mathit{map}\ f\ (\mathit{Just}\ a) & = & \mathit{Just}\ (f\ a)
\end{array}
\end{array}
$$

3. Embedded Prolog

This section introduces monads as a constructor class and defines the computational primitives of embedded Prolog. For each feature — nondeterminism, cut, exception handling, and interaction — a subclass of the basic monadic class is given.

3.1. Monads

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

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} & & \\
\quad \mathit{return} & :: & a \to m\ a \\
\quad (\gg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b \\
\quad (\gg) & :: & m\ a \to m\ b \to m\ b \\
\quad m \gg n & = & m \gg\!= \lambda_ \to n
\end{array}
$$

The essential idea of monads is to distinguish between *computations* and *values*. This distinction is reflected on the type level: an element of $m\ a$ represents a computation that yields a value of type a. A computation may involve, for instance, state, exceptions, or nondeterminism. In this article we are, of course, mainly concerned with the latter computational feature.

The trivial computation that immediately returns the value a is denoted by $\mathit{return}\ a$. The operator $(\gg\!=)$, commonly called 'bind', combines two computations: $m \gg\!= k$ 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'. The predicate that always succeeds is given by

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

The operations *return* and $(\gg\!=)$ constitute the conjunctive kernel of embedded Prolog.

Building upon *return* and $(\gg\!=)$ we can define an auxiliary function that 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.

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

Despite appearances, *Functor* and *Monad* are closely related: every monad can be made an instance of *Functor*.[a]

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Functor}\ m\ \textbf{where} \\
\quad \mathit{map}\ f\ m \quad = \quad m \gg\!= \mathit{return} \circ f
\end{array}
$$

Here, $\mathit{map}\ f\ m$ applies f to the result of the computation m.

3.2. 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

[a] The instance declaration is not legal Haskell 98 since the instance head, the argument of *Functor*, is not of the form $T\ a_1 \ldots a_k$. Here, we use an extension of the class system [34] implemented both in GHC and in Hugs.

operation is designated to get out of a monad. The following class definition fills the gap.

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

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 among others an operation, called *sols*, which collects all solutions of a given computation. Since *sols m* has exactly one solution, it can be safely encapsulated: *run* (*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 a notable exception: a computation that 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 that defines a mapping into the *IO* monad.

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

3.3. Backtracking

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

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

The constant *fail* denotes a failing computation, (|) realizes a nondeterministic choice. We agree upon that (≫=) and (≫) take precedence over (|). The operations *fail* and (|) constitute the disjunctive kernel of embedded Prolog. 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: (|) is not commutative, the order in which the alternatives are presented usually matters, see Section 5.8.

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. Both *once* m and *sols* m succeed exactly once. Among other things they are useful for encapsulating nondeterministic computations: if m has type $m\ a$, then *run* (*once* m) of type *Maybe* a computes the first solution of m and *run* (*sols* m) of type $[a]$ yields a list of all solutions. Furthermore, the operation *once* will prove important for the axiomatization of the backtracking core as it allows us to characterize deterministic computations, see Section 5.2.

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.

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

The other way round is equally easy.

$$\begin{array}{lcl} \mathit{once} & :: & (\mathit{Backtr}\ m) \Rightarrow m\ a \to m\ (\mathit{Maybe}\ a) \\ \mathit{once}\ m & = & \mathit{atMostOnce}\ (\mathit{map}\ \mathit{Just}\ m \mid \mathit{return}\ \mathit{Nothing}) \end{array}$$

Using *once* we can also implement Prolog's *negation as failure*.

$$\begin{array}{lcl} \mathit{naf} & :: & (\mathit{Backtr}\ m) \Rightarrow m\ a \to m\ () \\ \mathit{naf}\ m & = & \mathit{once}\ m \gg\!= \mathit{maybe}\ \mathit{true}\ (\lambda_ \to \mathit{fail}) \end{array}$$

The operation *naf* reverts the notions of success and failure: if m fails, then *naf* m succeeds and vice versa.

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 simply use $(+\!\!+)$, for splitting a list into two we define

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

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 *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 [41].

We conclude the section by noting that the second-orderness of $(|)$, *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.

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

3.4. The cut

Prolog offers a predefined predicate, called cut and denoted !, for improving time and space behaviour of programs. We introduce this control construct as a subclass of *Backtr*.

$$\begin{array}{l} \textbf{class}\ (Backtr\ m) \Rightarrow Cut\ m\ \textbf{where} \\ \quad \begin{array}{lcl} ! & :: & m\ () \\ call & :: & m\ a \to m\ a \end{array} \end{array}$$

The cut succeeds exactly once and returns (). As a side-effect it discards *all* previous alternatives or choice points. The operation *call* may be used to delimit the effect of a cut: *call* m executes m; if the cut is invoked in m, it discards only the choices made since m was called. Note that in Prolog the effect of a cut is local to a procedure: a cut commits Prolog to all choices made since the so-called parent goal was invoked, see [42].

A useful derived operation is

$$\begin{array}{lcl} only & :: & (Cut\ m) \Rightarrow a \to m\ a \\ only\ a & = & ! \gg return\ a\ , \end{array}$$

which combines the cut and *return*. Using *only* we can re-implement the function *atMostOnce*

$$\begin{array}{lcl} atMostOnce & :: & (Cut\ m) \Rightarrow m\ a \to m\ a \\ atMostOnce\ m & = & call\ (m \gg\!= only)\ , \end{array}$$

which shows that *once* is in some sense a tamed variant of the cut. In Prolog the cut is the basis for implementing the negation as failure rule. Here is the corresponding definition in embedded Prolog — recall that $(\gg)$ takes precedence over $(|)$.

$$\begin{array}{lcl} naf & :: & (Cut\ m) \Rightarrow m\ a \to m\ () \\ naf\ m & = & call\ (m \gg ! \gg fail \mid true) \end{array}$$

The definition of *naf* makes use of a common programming idiom, the so-called *cut-fail combination*. When $! \gg fail$ is executed, control is passed to the nearest enclosing invocation of *call*, which in turn instantly fails.

Having given two definitions of both *atMostOnce* and *naf* the question naturally arises as to whether they are equivalent in each case. It turns out that the two variants of *atMostOnce* are, in fact, equivalent, while the variants of *naf* are subtly different, see Sections 5.8 and 5.10.

3.5. Exception handling

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

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

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

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.

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

The converse holds as well.

$$\begin{array}{lcl}
\mathit{try} & :: & (\mathit{Exc}\ m) \Rightarrow m\ a \to m\ (\mathit{Either}\ \mathit{Err}\ a) \\
\mathit{try}\ m & = & \mathit{catch}\ (\mathit{map}\ \mathit{Right}\ m)\ (\mathit{return} \circ \mathit{Left})
\end{array}$$

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

3.6. Input and output

The I/O system sublanguage offers basic operations for terminal and file input and output.

$$\begin{array}{lcl}
\textbf{class } (\mathit{Monad}\ m) \Rightarrow \mathit{InOut}\ m \textbf{ where} & & \\
\quad \mathit{out} & :: & \mathit{String} \to m\ () \\
\quad \mathit{inp} & :: & m\ \mathit{String} \\
\quad \mathit{outFile} & :: & \mathit{FilePath} \to \mathit{String} \to m\ () \\
\quad \mathit{inpFile} & :: & \mathit{FilePath} \to m\ \mathit{String}
\end{array}$$

The operation *out s* writes *s* to the standard output device, *inp* reads a line of characters from the standard input device. Likewise, *outFile path s* appends *s* to the file named *path* and *inpFile path* reads the file named *path* and returns its contents as a string.

4. 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 ($\gg\!=$) and ($\gg$). The syntax and semantics of **do**-

expressions are given by the following identities:[b]

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

The **do**-expressions above use braces and semi-colons to indicate the begin and end of commands. In the examples below we will employ layout to convey the same information. Briefly, Haskell's layout rule ("offside rule") is activated if the open brace following **do** is missing. Then the indentation of the next symbol is remembered. For each following line, if it is indented more, then the previous command is continued; if it is indented the same amount, a new command begins; and if it is indented less, then the **do**-expression ends. Note that the same rule also applies to **where**, **let**, and **case**, see [37].

4.1. Transitive closure

The famous textbook by Sterling and Shapiro [42] introduces Prolog's basic constructs using the running example of a 'Biblical family database'. Here is a small excerpt reformulated in monadic terms.

```
child              :: (Backtr m) => String -> m String
child "terach"     =  return "abraham" | return "nachor" | return "haran"
child "abraham"    =  return "isaac"
child "haran"      =  return "lot" | return "milcah" | return "yiscah"
child "sarah"      =  return "isaac"
child _            =  fail
```

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. By contrast, 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.

```
grandChild  :: (Backtr m) => String -> m String
grandChild  =  child <=< child
```

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

[b] Note that the second equation holds only if matching against the pattern p cannot fail. All the patterns we employ — variables and tuples of variables — satisfy this constraint.

transitive closure.

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

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*.

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

Note that *transitiveClosure* m and *transitiveClosure'* m are generally not equivalent if m involves computational effects such as output, see Section 5.8.

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.

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

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.

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

Here, *elem* is the list membership function. Note that the type of the function reflects the computational features involved in its definition.

4.2. 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.

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

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

```
place              ::  (Backtr m) ⇒ Int → [Int] → [Int] → [Int] → m [Int]
place 0 rs d1 d2   =   return []
place i rs d1 d2   =   do (q, rs') ← select rs
                          let q1 = q − i
                          let q2 = q + i
                          guard (not (elem q1 d1))
                          guard (not (elem q2 d2))
                          qs ← place (i − 1) rs' (q1 : d1) (q2 : d2)
                          return (q : qs)
```

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

```
select             ::  (Backtr m) ⇒ [a] → m (a, [a])
select []          =   fail
select (a : x)     =   return (a, x)
                   |   do {(b, x') ← select x; return (b, a : x')}
```

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

```
guard    ::  (Backtr m) ⇒ Bool → m ()
guard b  =   if b then true else fail .
```

4.3. 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 [30]: *trace m msg* behaves as m except that the user is informed when (i) m is called the first time, (ii) m is exited upon success, (iii) m fails, and (iv) m is re-entered upon backtracking.

```
trace                ::  (Backtr m, InOut m) ⇒ m a → String → m a
trace m msg          =   do (say "call" | say "fail" ≫ fail)
                            a ← m
                            (say "exit" | say "redo" ≫ fail)
                            return a
    where say s      =   out (s ++ ": " ++ msg ++ "\n")
```

Double use is made of the idiom $m_1 \gg (\iota_1 \mid \iota_2 \gg fail) \gg m_2$, where ι_1 and ι_2 succeed exactly once, 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 $\iota_2 \gg fail$ 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 5. Given the definition of *tchild*

```
tchild     ::  (Backtr m, InOut m) ⇒ String → m String
tchild a   =   trace (child a) ("child " ++ a)
```

one can show, for instance, that *transitiveClosure' tchild* `"abraham"` ≫ *fail* is equivalent to

```
    out "call: child abraham"  ≫  out "exit: child abraham"
≫   out "call: child isaac"    ≫  out "fail: child isaac"
≫   out "redo: child abraham"  ≫  out "fail: child abraham"
≫   fail .
```

A proof of this claim appears in Section 5.9.

4.4. 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.

```
shell      ::  (Backtr m, InOut m, Show a) ⇒ m a → m ()
shell m    =   do a ← m
                  out (show a ++ "\n" ++ "more? ")
                  s ← inp
                  guard (s /= "y")
               |  out "no."
```

4.5. Parsing

A popular application of backtracking monads is recursive descent parsing [18]. In this section we briefly review the basic idea, illustrate the use of ! and *call* for improving the space behaviour of parsers, and show how to combine parsing with interaction.

Basically, a parser is a function that takes a list of tokens and returns a pair consisting of an abstract syntax tree and the unconsumed suffix of the token list: $[Token] \to (a, [Token])$. A nondeterministic parser is simply a directed relation of that type.[c]

```
type Token              =  String
newtype Parser m a      =  [Token] → m (a, [Token])
```

Note that essentially the same approach is taken in Prolog's definite clause grammars [42]. On a more abstract level a parser combines backtracking with a *backtrackable state*. If *m* is a backtracking monad, then *Parser m* can be made into an

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

instance of *Monad* and *Backtr*.

```
instance (Monad m) ⇒ Monad (Parser m) where
    return a   =  λinp → return (a, inp)
    m ≫= k     =  λinp → m inp ≫= λ(a, inp') → k a inp'
instance (Backtr m) ⇒ Backtr (Parser m) where
    fail       =  λinp → fail
    m ∣ n      =  λinp → m inp ∣ n inp
```

The implementation of *once* and *sols* is left as an exercise to the reader.

Parsing with monads is popular mainly because grammars in EBNF can be easily converted into recursive descent parsers. For each grammar construction such as sequencing, alternation, or repetition there is a corresponding parser combinator. Of course, (≫=) implements sequencing and (∣) alternation. Repetition is a derived operation.

```
many              ::  (Backtr m) ⇒ m a → m [a]
many m            =   ms
    where ms      =   do a ← m
                         as ← ms
                         return (a : as)
                  ∣   return []
```

The combinator *many* is not specific to parsing. Its type signature shows that *many* may be used for arbitrary backtracking monads. Its use is, however, pointless for pure nondeterministic computations, for which *many m* simply loops.

Terminal symbols are recognized by *literal*.

```
literal      ::  (Backtr m) ⇒ Token → Parser m Token
literal a    =   satisfy (a ==)
```

The parser *satisfy p* determines whether the next input token belongs to a class of symbols specified by the predicate *p*.

```
satisfy      ::  (Backtr m) ⇒ (Token → Bool) → Parser m Token
satisfy p    =   λinp → case inp of [] → fail
                                    a : as | p a → return (a, as)
                                           | otherwise → fail
```

Finally, we require a primitive for applying a parser to a given list of input tokens.

```
apply          ::  (Backtr m) ⇒ Parser m a → [Token] → Parser m a
apply p inp    =   λinp' → do (a, rest) ← p inp
                              guard (null rest)
                              return (a, inp')
```

Note that the guard in the second line checks whether the input is completely consumed by the parser.

Let us illustrate the monadic parsing combinators using the following sample grammar for a toy functional language. This example is inspired by a similar one given in [39].

$$
\begin{array}{lcl}
\mathit{program} & \longrightarrow & \mathit{decl}^* \\
\mathit{decl} & \longrightarrow & \mathit{var}\ \mathit{var}^*\ \texttt{=}\ \mathit{exp}\ \texttt{;} \\
\mathit{exp} & \longrightarrow & \mathit{var}\ \mathit{aexp}^* \mid \mathit{aexp} \\
\mathit{aexp} & \longrightarrow & \mathit{int} \mid \texttt{(}\ \mathit{exp}\ \texttt{)}
\end{array}
$$

The corresponding parser essentially augments the grammar with code for constructing the abstract syntax tree. Here is the parser for declarations.

```
data Decl  =  Decl Var [Var] Exp
decl       ::  (Backtr m) ⇒ Parser m Decl
decl       =  do f ← var
                 as ← many var
                 literal "="
                 e ← exp
                 literal ";"
                 return (Decl f as e)
```

The code nicely exemplifies the direct correspondence between EBNF notation and monadic parsing combinators. Fig. 1 presents the complete parser for the example grammar.

It is well-known that backtracking parsers inherently suffer from a serious *space leak*. Consider the definition of $m \mid n$ and note that the input is passed to both m and n. Thus, the input cannot be garbage collected until n is invoked. If m parses a large part of the input, then a great deal of space is required to hold these already-parsed tokens, just in case m finally fails and n needs to be considered. Judicious use of ! can help to improve the situation. Before demonstrating the use of the cut, we must lift it into the realm of parsers.

```
instance (Cut m) ⇒ Cut (Parser m) where
    !       =  λinp → ! ≫= λa → return (a, inp)
    call m  =  λinp → call (m inp)
```

Now, re-consider the definition of *decl*. This production is the only one that contains the literal '='. Once the equal sign has been recognized, there is no need ever to backtrack beyond this point.[d]

```
decl  ::  (Cut m) ⇒ Parser m Decl
decl  =  do f ← var
            as ← many var
            literal "="
            !
            e ← exp
            literal ";"
            return (Decl f as e)
```

[d] Interestingly, this property also holds for the Haskell grammar.

```
data Program  =  Program [Decl]
data Decl     =  Decl Var [Var] Exp
data Exp      =  Int Int
              |  App Var [Exp]
type Var      =  String

program       ::  (Backtr m) ⇒ Parser m Program
program       =   map Program (many decl)
decl          ::  (Backtr m) ⇒ Parser m Decl
decl          =   do f ← var
                     as ← many var
                     literal "="
                     e ← exp
                     literal ";"
                     return (Decl f as e)
exp, aexp     ::  (Backtr m) ⇒ Parser m Exp
exp           =   do f ← var
                     as ← many aexp
                     return (App f as)
              |   aexp
aexp          =   map Int int
              |   do literal "("
                     e ← exp
                     literal ")"
                     return e
var           ::  (Backtr m) ⇒ Parser m String
var           =   satisfy (all isAlpha)
int           ::  (Backtr m) ⇒ Parser m Int
int           =   map read (satisfy (all isDigit))
```

Fig. 1. A parser for the example grammar.

Recall that the cut discards *all* previous choice points effectively freeing the space retained by (|).

The cut may also be used to implement a more space economical variant of *many*. Note that *many m* returns *all* possible parses with the longest first. Even when m is deterministic, *many m* returns $n+1$ solutions if m can be applied n times in sequence. Here is *many*, rewritten with ! and *call*.

```
many'          :: (Cut m) ⇒ m a → m [a]
many' m        =  call ms
    where ms   =  do a ← m
                     !
                     as ← ms
                     return (a : as)
                  | return []
```

Once m succeeds, the cut discards the second branch, *return* [], and alternative solutions to m. The effect of the cut, however, does not extend beyond *many' m*.

Finally, let us show how to combine parsing and interaction. Assume for the sake of example that we want to augment our toy language by a simple facility for the inclusion of files (analogous to C's `#include` directive).

```
incldecl  ⟶  include var ;
```

The intention is that `include path;` behaves as if the declaration were replaced by the contents of the file `path`. Now, to implement the corresponding parser we must make *Parser m* an instance of *InOut*. We will, however, postpone the necessary declaration until Section 6.1, where it is given in a more general setting.

```
incldecl  ::  (Backtr m, InOut m) ⇒ Parser m Program
incldecl  =   do literal "include"
                 path ← var
                 cnts ← inpFile path
                 ds ← apply program (lexer cnts)
                 literal ";"
                 return ds
```

The parser *incldecl* determines the file name and uses *inpFile* to read its contents. Assuming a predefined function *lexer* :: *String* → [*Token*] the parser *program* is then applied to the tokenized input. The resulting abstract syntax tree is finally returned. Note that *apply* automatically saves and restores the current input so that nested calls to *apply* work, in fact, properly.

5. Axioms

This section is concerned with the axiomatic characterization of the proposed computational primitives. The primary purpose of the axiomatic approach is to aid the programmer in stating and proving properties about programs. Using the axioms we will be able, for instance, to answer questions like the following.

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

What is the meaning of $sols\ (out\ s \mid !) \gg raise\ e)$?

Are the two definitions $naf_1\ m = once\ m \gg\!= maybe\ true\ (\lambda_ \to fail)$ and $naf_2\ m = call\ (m \gg ! \gg fail \mid true)$ equivalent?

As the main result of this section we present a normal form for the language defined in Section 3. This result shows that the axioms are expressive enough to transform each finite computation into a canonical form.

5.1. Monads

For each computational feature we will specify a handful of axioms that try to capture our intuition about this feature. To begin with the basic monadic operations must be related by

$$return\ a \gg\!= k \quad = \quad k\ a \tag{M1}$$

$$m \gg\!= return \quad = \quad m \tag{M2}$$

$$(m \gg\!= k) \gg\!= \ell \quad = \quad m \gg\!= (\lambda a \to k\ a \gg\!= \ell)\ . \tag{M3}$$

These so-called *monad laws* are easier to remember if we rephrase them in terms of the Kleisli composition.

$$f \circledcirc return \quad = \quad f \tag{M1'}$$

$$return \circledcirc f \quad = \quad f \tag{M2'}$$

$$f \circledcirc (g \circledcirc h) \quad = \quad (f \circledcirc g) \circledcirc h \tag{M3'}$$

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

We have noted in Section 3.1 that every monad can be considered as a functor. If $map\ f\ m$ is defined as $m \gg\!= return \circ f$, then it automatically satisfies the so-called *functor laws*:

$$map\ id \quad = \quad id \tag{F1}$$

$$map\ (f \circ g) \quad = \quad map\ f \circ map\ g\ . \tag{F2}$$

Both laws can be easily derived from the monad laws.

5.2. 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 that 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 $(\mid)$ form a monoid, ie *fail* is the neutral element of $(\mid)$ and $(\mid)$ is associative.

$$\begin{aligned} \mathit{fail} \mid m &= m && \text{(B1)} \\ m \mid \mathit{fail} &= m && \text{(B2)} \\ (m \mid n) \mid o &= m \mid (n \mid o) && \text{(B3)} \end{aligned}$$

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

$$\begin{aligned} \mathit{fail} \gg\!= k &= \mathit{fail} && \text{(B4)} \\ (m \mid n) \gg\!= k &= (m \gg\!= k) \mid (n \gg\!= k) && \text{(B5)} \end{aligned}$$

We have paraphrased $m \mid n$ 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. Eq.(B5) provides the missing link relating $(m \mid n) \gg\!= k$ to a simple disjunction.

Here are the laws for *once*.

$$\begin{aligned} \mathit{once}\ \mathit{fail} &= \mathit{nothing} && \text{(O1)} \\ \mathit{once}\ (\mathit{return}\ a \mid m) &= \mathit{just}\ a && \text{(O2)} \\ (\mathit{once}\ m \gg\!= k) \mid n &= \mathit{once}\ m \gg\!= \lambda a \to k\ a \mid n && \text{(O3)} \\ \mathit{once}\ (\mathit{once}\ m \gg\!= k) &= \mathit{once}\ m \gg\!= \lambda a \to \mathit{once}\ (k\ a) && \text{(O4)} \end{aligned}$$

The auxiliary functions *nothing* and *just* are given by

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

The first two equations formalize the informal description of *once*: *once* m returns *Nothing* if m fails and *Just* a, where a is the first solution of m, otherwise. 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 (O4). Furthermore, (O2) implies that *return* a is deterministic. Finally, (O3) 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.

$$\begin{aligned} \mathit{nil} &:: (\mathit{Monad}\ m) \Rightarrow m\ [a] \\ \mathit{nil} &= \mathit{return}\ [\,] \\ \mathit{cons} &:: (\mathit{Monad}\ m) \Rightarrow a \to m\ [a] \to m\ [a] \\ \mathit{cons}\ a\ \mathit{ms} &= \mathbf{do}\ \{as \leftarrow \mathit{ms}; \mathit{return}\ (a : as)\} \end{aligned}$$

The operations *nil* and *cons* raise [] and (:) to the level of computations.

$$\begin{aligned}
&\mathit{sols}\ \mathit{fail} && = && \mathit{nil} && \text{(S1)}\\
&\mathit{sols}\ (\mathit{return}\ a \mid m) && = && \mathit{cons}\ a\ (\mathit{sols}\ m) && \text{(S2)}\\
&\mathit{sols}\ (\mathit{once}\ m \gg\!= k) && = && \mathit{once}\ m \gg\!= \lambda a \to \mathit{sols}\ (k\ a) && \text{(S3)}\\
&\mathit{once}\ (\mathit{sols}\ m) && = && \mathit{map}\ \mathit{Just}\ (\mathit{sols}\ m) && \text{(S4)}
\end{aligned}$$

The first two equations formalize that *sols* m collects all solutions m generates. Eq.(S3) is similar to (O3) and (O4); it states that a deterministic computation may be pushed out of a call to *sols*. Finally, (S4) identifies *sols* as deterministic.

5.3. The cut

The cut is characterized by the following equations.

$$\begin{aligned}
(! \gg m) \mid n &= ! \gg m && \text{(!1)}\\
! \gg (m \mid n) &= m \mid ! \gg n && \text{(!2)}\\
! \gg \mathit{true} &= ! && \text{(!3)}\\
\mathit{once}\ (! \gg m) &= \mathit{once}\ m && \text{(!4)}\\
\mathit{sols}\ (! \gg m) &= \mathit{sols}\ m && \text{(!5)}
\end{aligned}$$

The first equation formalises our intuition that a cut discards *past* choice points, ie alternatives that appear 'above' or to its left. On the other hand, the cut does not affect *future* choice points, ie alternatives that appear to its right. This fact is captured by (!2), which shows that a cut may be pushed into the right branch of a disjunction. Eq.(!3) simply records that ! returns (). Finally, (!4) and (!5) lay down that the effect of a cut does not extend outside *once* or *sols*. This is in accordance with the Prolog ISO standard [19]. Section 5.8 furthermore explains why this is also a sensible choice.

The axioms have a number of interesting consequences. Replacing m by *true* in (!1) and applying (!3) we obtain $! \mid m = !$. Similarly, replacing n by *fail* in (!2) we obtain

$$! \gg m \quad = \quad m \mid ! \gg \mathit{fail}\ . \qquad (1)$$

Combining the two derived laws we can, for instance, show that the cut is idempotent, ie $! \gg ! = !$.

$$\begin{aligned}
! \gg ! &= ! \mid ! \gg \mathit{fail} && \text{Eq.(1)}\\
&= ! && ! \mid m = !
\end{aligned}$$

Furthermore, the cut commutes with deterministic computations: if d is deterministic, then $d \gg\!= \mathit{only} = ! \gg d$. The proof essentially uses the fact that a deterministic computation can be pushed out of a disjunction.

$$! \gg d \quad = \quad d \mid ! \gg \mathit{fail} \qquad \text{Eq.(1)}$$

$$
\begin{array}{lll}
= & d \gg\!= \lambda a \to \mathit{return}\ a \mid ! \gg \mathit{fail} & d \text{ is det.} \\
= & d \gg\!= \lambda a \to ! \gg \mathit{return}\ a & \text{Eq.(1)} \\
= & d \gg\!= \mathit{only} & \text{Def. } \mathit{only}
\end{array}
$$

The operation *call* satisfies a similar set of equations as *once* or *sols*.

$$
\begin{array}{lll}
\mathit{call}\ \mathit{fail} & = & \mathit{fail} \qquad \text{(C1)} \\
\mathit{call}\ (\mathit{return}\ a \mid m) & = & \mathit{return}\ a \mid \mathit{call}\ m \qquad \text{(C2)} \\
\mathit{call}\ (\mathit{once}\ m \gg\!= k) & = & \mathit{once}\ m \gg\!= \lambda a \to \mathit{call}\ (k\ a) \qquad \text{(C3)} \\
\mathit{call}\ (! \gg m) & = & \mathit{call}\ m \qquad \text{(C4)}
\end{array}
$$

Thus *call* m behaves essentially like m except that any cut inside m has only local effect.

5.4. Exception handling

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

The operations *raise* and *try* are expected to satisfy the following laws.

$$
\begin{array}{lll}
\mathit{raise}\ e \gg\!= k & = & \mathit{raise}\ e \qquad \text{(E1)} \\
\mathit{try}\ (\mathit{return}\ a) & = & \mathit{return}\ (\mathit{Right}\ a) \qquad \text{(E2)} \\
\mathit{try}\ (\mathit{raise}\ e) & = & \mathit{return}\ (\mathit{Left}\ e) \qquad \text{(E3)} \\
\mathit{try}\ (\mathit{try}\ m) & = & \mathit{map}\ \mathit{Right}\ (\mathit{try}\ m) \qquad \text{(E4)} \\
\mathit{try}\ (\mathit{try}\ m \gg\!= k) & = & \mathit{try}\ m \gg\!= \lambda a \to \mathit{try}\ (k\ a) \qquad \text{(E5)}
\end{array}
$$

Signaling an error terminates the current computation: *raise* e is a left zero of bind. Eq.(E2) and (E3) formalize the operational description of *try*. Using *try* we can characterize safe computations, ie computations that do not raise an exception: m is *safe* iff $\mathit{try}\ m = \mathit{map}\ \mathit{Right}\ m$. Eq.(E4) shows that *try* m is safe itself. Eq.(E5) formalises that a safe computation can be pushed out of a call to *try*. Note that (E5) implies (E4). However, (E4) is additionally listed as (E5) does not hold in general, see below.

The interaction of exception handling and backtracking is explained by the following equations.

$$
\begin{array}{lll}
\mathit{once}\ (\mathit{raise}\ e) & = & \mathit{raise}\ e \qquad \text{(E6)} \\
\mathit{try}\ \mathit{fail} & = & \mathit{fail} \qquad \text{(E7)} \\
\mathit{try}\ (\mathit{return}\ a \mid m) & = & \mathit{return}\ (\mathit{Right}\ a) \mid \mathit{try}\ m \qquad \text{(E8)} \\
\mathit{try}\ (! \gg m) & = & \mathit{try}\ m \qquad \text{(E9)}
\end{array}
$$

Eq.(E6) lays down that *raise* e propagates through *once*. Despite appearance (E6) has far reaching consequences: it means that *raise* e is deterministic and it

implies $raise\ e \mid m = raise\ e$, $sols\ (raise\ e) = raise\ e$, $call\ (raise\ e) = raise\ e$, and $! \gg raise\ e = raise\ e$. The first equation shows that $raise\ e$ discards past choice points. Here is a simple calculational proof of the claim.

$$\begin{aligned} raise\ e \mid m &= raise\ e \gg\!= \lambda a \to return\ a \mid m && \text{(M2) and } raise\ e \text{ is det.} \\ &= raise\ e && \text{(E1)} \end{aligned}$$

The laws for *try* have been chosen to conform to the Prolog ISO standard [19]. In particular (E8) lays down, that *try* is re-satisfiable. Note that (E8) implies (E2). Eq.(E9) specifies that *try* is not transparent to cut.

The interaction of exception handling and backtracking is, in fact, quite subtle. It turns out that (E5) does not hold for monads that support both computational features, see Section 5.8. In this case a computation can only be pushed out of a call to *try* if it is both safe *and* deterministic. Hence, in general we must be content with the following weakening of (E5): if s is safe and deterministic, then

$$try\ (s \gg\!= k) \quad = \quad s \gg\!= \lambda a \to try\ (k\ a)\ . \qquad \text{(E5')}$$

5.5. 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

$$\begin{aligned} once\ \iota &= map\ Just\ \iota && \text{(IO1)} \\ try\ \iota &= map\ Right\ \iota && \text{(IO2)} \end{aligned}$$

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 (IO2) for $\iota = inp$.

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

$$\begin{aligned} out\ "" &= true && \text{(IO3)} \\ out\ (s_1 +\!\!+ s_2) &= out\ s_1 \gg out\ s_2 && \text{(IO4)} \end{aligned}$$

5.6. 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 $(\gg\!=)$.

$$\begin{aligned} \eta\ (return\ a) &= return\ a && (\eta 1) \\ \eta\ (m \gg\!= k) &= \eta\ m \gg\!= \lambda a \to \eta\ (k\ a) && (\eta 2) \end{aligned}$$

A function that satisfies only ($\eta 1$) is termed premonad morphism. We will see that monad morphisms play a central rôle in adding features to a monad, see Section 6.1.

5.7. 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 6.2). 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

$$\eta\ (\iota \gg\!= k) \quad = \quad \iota \gg\!= \lambda a \to \eta\ (k\ a) \tag{$\eta 3$}$$

must hold for $\eta = perform$.

5.8. Equations that do not hold in general

By now we have seen a multitude of laws that every implementation shall satisfy. It is, however, also interesting and sometimes even revealing to learn which equations do not hold in general.

The nondeterministic choice is in general 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*.

$$\begin{array}{lcl} repeat & :: & (Backtr\ m) \Rightarrow m\ () \\ repeat & = & true \mid repeat \end{array}$$

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

The dual properties of (B4) and (B5) do not hold: it is neither true that *fail* is a right zero of ($\gg\!=$) nor that ($\gg\!=$) distributes rightward through ($\mid$). For instance, $out\ s \gg fail$ does not equal *fail*. If this were true, the tracer of Section 4.3 would make no sense. Furthermore note that the idiom $m \gg fail$ is known to Prolog programmers as a *failure driven loop*. To see that right distributivity does not hold in general consider $out\ s \gg (true \mid true)$, which is not the same as $(out\ s \gg true) \mid (out\ s \gg true)$. Interestingly, the 'optimization' of *transitiveClosure* in Section 4.1 rests on right distributivity. For that reason *transitiveClosure* m and *transitiveClosure'* m behave differently if m involves computational effects such as output: since m is called twice in *transitiveClosure* m, the output is doubled as compared to *transitiveClosure'* m.

Turning to the all solutions function one may wonder whether (S2) may be generalized to $sols\ (m \mid n) = sols\ m \oplus sols\ n$ where ($\oplus$) is given by

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

The operator ($\oplus$) lifts ($+\!\!+$) to the level of computations. We have that *nil* is a unit of ($\oplus$), and that ($\oplus$) is associative. Unfortunately, the generalization of (S2) fails for computations involving the cut. Take, for instance, $m = !$ and $n = true$. Using the generalized law we could infer $sols\ (m \mid n) = return\ [(), ()]$. However, since the cut discards past choice points, we actually have $sols\ (m \mid n) = return\ [()]$. In a sense, (S2) captures strict left-to-right evaluation.

The effect of a cut shall be limited by *once*. Can we alternatively make *once* transparent to cut postulating $once\ (! \gg m) = ! \gg once\ m$? To see that this is not a viable choice consider $once\ (! \gg (true \mid true)) = once\ (true \mid ! \gg true) = just\ ()$. Applying the proposed law, however, we obtain $! \gg just\ ()$. Interestingly, the invalidity of the above equation also implies that ! is *not* deterministic though it succeeds exactly once. To see why just note that $once\ ! = map\ Just\ !$ implies $once\ (! \gg m) = ! \gg once\ m$.

Finally, let us discuss two invalid equations involving the *try* operator. The first, $try\ (m \mid n) = try\ m \mid try\ n$, aims at generalizing (E8). Setting $m = raise\ e$ and noting that $raise\ e \mid m = raise\ e$ a contradiction can be easily derived. The second, (E5), can be considered as a natural counterpart to (O4) formalising our false intuition that a safe computation can be pushed out of a call to *try*. It fails, however, for similar reasons as the first equation. Take, for instance, $m = true \mid true$ and $k = \lambda_ \to raise\ e$. A straightforward calculation shows that $try\ (try\ m \gg\!= k)$ can be simplified to $return\ (Left\ e)$. On the other hand, if we apply (E5) first, we obtain $return\ (Left\ e) \mid return\ (Left\ e)$.

5.9. Examples

In this section we will take a look at some example derivations, which demonstrate the applicability and the expressibility of the axiomatic approach. Among other things we will answer the questions posed in the introduction to this section. Beforehand let us state some simple consequences of the axioms. The first corollary gathers our knowledge about deterministic computations.

Corollary 1

1. *return a, once m, sols m, raise e, and the IO actions are deterministic.*
2. *If d is deterministic, then*

$$\begin{aligned}
(d \gg\!= k) \mid n &= d \gg\!= \lambda a \to k\ a \mid n \\
once\ (d \gg\!= k) &= d \gg\!= \lambda a \to once\ (k\ a) \\
sols\ (d \gg\!= k) &= d \gg\!= \lambda a \to sols\ (k\ a) \\
d \gg\!= only &= ! \gg d \\
call\ (d \gg\!= k) &= d \gg\!= \lambda a \to call\ (k\ a)\ .
\end{aligned}$$

The second corollary is accordingly concerned with safe computations.

Corollary 2

1. *return a, fail, try m, and the IO actions are safe.*

2. If s is deterministic and safe, then

$$try\ (s \gg\!= k) \quad = \quad s \gg\!= \lambda a \rightarrow try\ (k\ a)\ .$$

We will see that Corollary 1 plays a central rôle in the following calculations. An immediate consequence of Corollary 1 is $(out\ s \gg m) \mid n = out\ s \gg (m \mid n)$, which answers the first question posed in the introduction to this section. Furthermore, we can simplify $sols\ (out\ s \mid !\ \gg raise\ e)$ to $out\ s \gg raise\ e$ using a straightforward calculation.

$$\begin{array}{lll}
 & sols\ (out\ s \mid !\ \gg raise\ e) & \\
= & out\ s \gg sols\ (true \mid !\ \gg raise\ e) & \text{Cor. 1} \\
= & out\ s \gg cons\ ()\ (sols\ (!\ \gg raise\ e)) & \text{(S2)} \\
= & out\ s \gg cons\ ()\ (sols\ (raise\ e)) & \text{(!5)} \\
= & out\ s \gg cons\ ()\ (raise\ e) & sols\ (raise\ e) = raise\ e \\
= & out\ s \gg raise\ e & \text{(E1)}
\end{array}$$

The next example is concerned with nonterminating computations. The function *dots* defined below employs a *repeat loop* to output an infinite sequence of dots.

$$\begin{array}{lll}
dots & :: & (Backtr\ m, InOut\ m) \Rightarrow m\ () \\
dots & = & repeat \gg out\ "." \gg fail
\end{array}$$

A simple calculation shows that *dots* is equal to $out\ "." \gg dots$.

$$\begin{array}{lll}
dots & = & repeat \gg out\ "." \gg fail & \text{Def. } dots \\
 & = & (true \mid repeat) \gg out\ "." \gg fail & \text{Def. } repeat \\
 & = & out\ "." \gg fail \mid repeat \gg out\ "." \gg fail & \text{(B5) and (M1)} \\
 & = & out\ "." \gg fail \mid dots & \text{Def. } dots \\
 & = & out\ "." \gg (fail \mid dots) & \text{Cor. 1} \\
 & = & out\ "." \gg dots & \text{(B1)}
\end{array}$$

Using the axioms we can, of course, also establish negative results. Consider the two implementations of negation as failure given in Section 3.3 and in Section 3.4. They are equivalent except for one subtle case, namely when the argument contains a cut-fail combination. For $m = !\ \gg fail$ the first definition of $naf\ m$ evaluates to

$$\begin{array}{lll}
 & = & once\ (!\ \gg fail) \gg\!= maybe\ true\ (\lambda_ \rightarrow fail) & \text{Def. } naf \\
 & = & nothing \gg\!= maybe\ true\ (\lambda_ \rightarrow fail) & \text{(!4) and (O1)} \\
 & = & true\ , & \text{(M1)}
\end{array}$$

while the second definition yields

$$\begin{array}{lll}
 & = & call\ (!\ \gg fail \gg !\ \gg fail \mid true) & \text{Def. } naf \\
 & = & call\ (!\ \gg fail \gg !\ \gg fail) & \text{(!1)} \\
 & = & fail\ . & \text{(C4), (B4), and (C1)}
\end{array}$$

The crux is that the cut, which is passed as an argument to *naf*, inadvertently discards the success branch. This observation suggests a simple cure of the problem; we must additionally 'protect' the argument of *naf*.

$$\begin{array}{lcl} naf & :: & (Cut\ m) \Rightarrow m\ a \to m\ () \\ naf\ m & = & call\ (call\ m \gg !\ \gg fail \mid true) \end{array}$$

Except for the types, this definition is indeed equivalent to the first one, see Section 5.10. As an aside, note that this definition also exactly corresponds to the Prolog implementation of `not`.

```
not X :- X, !, fail.
not X.
```

The first clause employs the so-called meta-variable facility, which is a syntactic convenience for `call(X)`.

As a slightly larger example let us finally prove the claim made in Section 4.3, namely that *transitiveClosure' tchild* `"abraham"` $\gg$ *fail* is equivalent to

$$\begin{array}{lll} & out\ \texttt{"call: child abraham"} & \gg \quad out\ \texttt{"exit: child abraham"} \\ \gg & out\ \texttt{"call: child isaac"} & \gg \quad out\ \texttt{"fail: child isaac"} \\ \gg & out\ \texttt{"redo: child abraham"} & \gg \quad out\ \texttt{"fail: child abraham"} \\ \gg & fail\ . & \end{array}$$

For the subsequent calculations it will be convenient to abbreviate *tchild* to tc and *transitiveClosure' tchild* to tc^*. The failure-driven loop has the following effect.

$$\begin{array}{lll} & tc^*\ a \gg fail & \\ = & (tc\ a \ggg\!= \lambda b \to return\ b \mid tc^*\ b) \gg fail & \text{Def. } tc^* \\ = & tc\ a \ggg\!= \lambda b \to (return\ b \mid tc^*\ b) \gg fail & \text{(M3)} \\ = & tc\ a \ggg\!= \lambda b \to tc^*\ b \gg fail & \text{(B5), (M1), and (B1)} \end{array}$$

Using this equality, we can now conclude that

$$\begin{array}{lll} & tc^*\ \texttt{"abraham"} \gg fail & \\ = & tc\ \texttt{"abraham"} \ggg\!= \lambda b \to tc^*\ b \gg fail & \text{see above} \\ = & (c_a \mid f_a \gg fail) \gg (e_a \mid r_a \gg fail) \gg tc^*\ \texttt{"isaac"} \gg fail\ . & \text{Def. } tc \end{array}$$

Here, c_a abbreviates the somewhat lengthy *out* `"call: child abraham"`; f_a, e_a, and r_a are defined accordingly. An auxiliary calculation determines the value of tc^* `"isaac"` $\gg$ *fail*.

$$\begin{array}{lll} & tc^*\ \texttt{"isaac"} \gg fail & \\ = & tc\ \texttt{"isaac"} \ggg\!= \lambda b \to tc^*\ b \gg fail & \text{see above} \\ = & (c_i \mid f_i \gg fail) \gg fail & \text{Def. } tc \\ = & c_i \gg fail \mid f_i \gg fail & \text{(B5) and (B4)} \\ = & c_i \gg f_i \gg fail & \text{Cor. 1 and (B1)} \end{array}$$

Now, we continue:

$$
\begin{array}{lll}
= & (c_a \mid f_a \gg \mathit{fail}) \gg (e_a \mid r_a \gg \mathit{fail}) \gg c_i \gg f_i \gg \mathit{fail} & \text{see above} \\
= & (c_a \mid f_a \gg \mathit{fail}) \gg (e_a \gg c_i \gg f_i \gg \mathit{fail} \mid r_a \gg \mathit{fail}) & \text{(B5) and (B4)} \\
= & (c_a \mid f_a \gg \mathit{fail}) \gg e_a \gg c_i \gg f_i \gg r_a \gg \mathit{fail} & \text{Cor. 1 and (B1)} \\
= & c_a \gg e_a \gg c_i \gg f_i \gg r_a \gg \mathit{fail} \mid f_a \gg \mathit{fail} & \text{(B5) and (B4)} \\
= & c_a \gg e_a \gg c_i \gg f_i \gg r_a \gg f_a \gg \mathit{fail} \ , & \text{Cor. 1 and (B1)}
\end{array}
$$

which establishes the result.

5.10. A normal form

In this section we show that each computation — or rather, computational term — can be transformed into a simple canonical form. This normal form has the usual property that two syntactically different computations in normal form are also semantically different, that is they exhibit a different computational behaviour. This result suggests that the axiomatization is in a sense complete. We impose, however, two restrictions. First of all, we do not consider infinite computations such as *dots*. Second, we exclude computations that depend on external information such as *inp* or *inpFile path*. For simplicity, we additionally restrict ourselves to *out s* as the sole IO action.

Which shape does a computational term in normal form take on? Since we consider nondeterministic computations, a computation may generate multiple solutions as in $\mathit{return}\ a_1 \mid \cdots \mid \mathit{return}\ a_n$. A computation may additionally produce output — before the first solution, between solutions, and after the last solution. This suggest the following canonical form.

$$\mathit{out}\ s_1 \gg \mathit{return}\ a_1 \mid \cdots \mid \mathit{out}\ s_n \gg \mathit{return}\ a_n \mid \mathit{out}\ s_{n+1} \gg \mathit{fail}$$

Note that this form also accounts for pure computations since $m = \mathit{out}\ "" \gg m$. Instead of *fail* a computation may alternatively end with a cut-fail combination or with an untrapped exception, which motivates the following definition.

Definition 1 *A computational term built from return, fail, !, raise, and out using the combining forms* $(\gg\!=)$, $(\mid)$, *once, sols, call, and try is in normal form if it equals* $\mathit{out}\ s_1 \gg \mathit{return}\ a_1 \mid \cdots \mid \mathit{out}\ s_n \gg \mathit{return}\ a_n \mid \mathit{out}\ s_{n+1} \gg \alpha$ *where* $n \geqslant 0$ *and* α *takes one of the following three forms:* fail, $! \gg \mathit{fail}$, *or* $\mathit{raise}\ e$.

Computational terms serve as an example for *higher-order abstract syntax*. Denote by $C\ a$ the set of all computational terms that return a value of type a. Then in $m \gg\!= k$, m is of type $C\ a$ and k is a *function* of type $a \to C\ b$, which yields a computational term for each element of type a. Note, that $C\ a$ is not expressible as a Haskell data type. First of all, $(\gg\!=)$ cannot be turned into a constructor since its type contains a 'free' type variable, which would require some form of existential types [24]. Even worse, the target types of ! and *out* are substitution instances of $C\ a$. To the best of the author's knowledge only the type system described in [12] allows for type-specialized constructors.

The normalization of a computational term is defined inductively on the structure of $C\ a$.

$$
\begin{array}{lcl}
\textsc{nf} & :: & C\ a \to C\ a \\
\textsc{nf}\ (return\ a) & = & return_{nf}\ a \\
\textsc{nf}\ (m \gg\!= k) & = & \textsc{nf}\ m \gg\!=_{nf} \textsc{nf} \circ k \\
\textsc{nf}\ fail & = & fail_{nf} \\
\textsc{nf}\ (m \mid n) & = & \textsc{nf}\ m \mid_{nf} \textsc{nf}\ n \\
\textsc{nf}\ (once\ m) & = & once_{nf}\ (\textsc{nf}\ m) \\
\textsc{nf}\ (sols\ m) & = & sols_{nf}\ (\textsc{nf}\ m) \\
\textsc{nf}\ ! & = & !_{nf} \\
\textsc{nf}\ (call\ m) & = & call_{nf}\ (\textsc{nf}\ m) \\
\textsc{nf}\ (raise\ e) & = & raise_{nf}\ e \\
\textsc{nf}\ (try\ m) & = & try_{nf}\ (\textsc{nf}\ m) \\
\textsc{nf}\ (out\ s) & = & out_{nf}\ s
\end{array}
$$

The function NF maps a computational term to its normal form. In essence each construct is replaced by a corresponding function that operates on normal forms. The normal forms of the basic operations are defined as follows.

$$
\begin{array}{lcl}
return_{nf}\ a & = & out\ "" \gg return\ a \mid out\ "" \gg fail \\
fail_{nf} & = & out\ "" \gg fail \\
!_{nf} & = & out\ "" \gg return\ () \mid out\ "" \gg ! \gg fail \\
raise_{nf}\ e & = & out\ "" \gg raise\ e \\
out_{nf}\ s & = & out\ s \gg return\ () \mid out\ "" \gg fail
\end{array}
$$

To improve readability we will henceforth abbreviate $\omega \mid out\ "" \gg fail$ to ω. The disjunction of normal forms essentially appends its two arguments unless the first argument ends with $! \gg fail$ or $raise\ e$. In this case the second argument is discarded.

$$
\begin{array}{lcl}
(\omega \mid out\ s \gg fail) \mid_{nf} \omega' & = & \omega \mid s \rhd \omega' \\
(\omega \mid out\ s \gg ! \gg fail) \mid_{nf} \omega' & = & \omega \mid out\ s \gg ! \gg fail \\
(\omega \mid out\ s \gg raise\ e) \mid_{nf} \omega' & = & \omega \mid out\ s \gg raise\ e
\end{array}
$$

The auxiliary function $(\rhd)$, termed prepend, is given by

$$
\begin{array}{lcl}
x \rhd (out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid \sigma) & & \\
\quad \mid n \geqslant 1 & = & out\ (x +\!\!+ s_1) \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid \sigma \\
x \rhd (out\ s_1 \gg \alpha) & = & out\ (x +\!\!+ s_1) \gg \alpha\ .
\end{array}
$$

Note that $x \rhd m$ implements $out\ x \gg m$. The bind operation falls back on disjunction and prepend. It uses the fact that $fail$ and $raise\ e$ are left zeros of $(\gg\!=)$.

$$
\begin{array}{l}
(out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid \sigma) \gg\!=_{nf} k \\
\quad = \quad s_1 \rhd k\ a_1 \mid_{nf} \cdots \mid_{nf} s_n \rhd k\ a_n \mid_{nf} \sigma
\end{array}
$$

The definition of $once_{nf}$ is straightforward.

$$
\begin{array}{lcl}
once_{nf}\ (out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid \sigma) & & \\
\quad \mid n \geqslant 1 & = & out\ s_1 \gg return\ (Just\ a_1) \\
once_{nf}\ (out\ s_1 \gg fail) & = & out\ s_1 \gg return\ Nothing \\
once_{nf}\ (out\ s_1 \gg ! \gg fail) & = & out\ s_1 \gg return\ Nothing \\
once_{nf}\ (out\ s_1 \gg raise\ e) & = & out\ s_1 \gg raise\ e
\end{array}
$$

Note that the second and the third case are treated alike as *once* is not transparent to cut. The same remark applies to $sols_{nf}$.

$$
\begin{aligned}
&sols_{nf}\ (out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid out\ s_{n+1} \gg fail) \\
&\quad = \quad out\ (s_1 +\!\!+ \cdots +\!\!+ s_n +\!\!+ s_{n+1}) \gg return\ [a_1, \ldots, a_n] \\
&sols_{nf}\ (out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid out\ s_{n+1} \gg\ ! \gg fail) \\
&\quad = \quad out\ (s_1 +\!\!+ \cdots +\!\!+ s_n +\!\!+ s_{n+1}) \gg return\ [a_1, \ldots, a_n] \\
&sols_{nf}\ (out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid out\ s_{n+1} \gg raise\ e) \\
&\quad = \quad out\ (s_1 +\!\!+ \cdots +\!\!+ s_n +\!\!+ s_{n+1}) \gg raise\ e
\end{aligned}
$$

The definition of $call_{nf}$ mirrors the fact that *call* m behaves exactly as m except that the effect of a cut does not extend outside *call* m.

$$
\begin{aligned}
call_{nf}\ (\omega \mid out\ s_{n+1} \gg fail) &= \omega \mid out\ s_{n+1} \gg fail \\
call_{nf}\ (\omega \mid out\ s_{n+1} \gg\ ! \gg fail) &= \omega \mid out\ s_{n+1} \gg fail \\
call_{nf}\ (\omega \mid out\ s_{n+1} \gg raise\ e) &= \omega \mid out\ s_{n+1} \gg raise\ e
\end{aligned}
$$

Finally, try_{nf} is defined as

$$
\begin{aligned}
try_{nf}\ (\omega \mid out\ s_{n+1} \gg fail) &= right\ \omega \mid out\ s_{n+1} \gg fail \\
try_{nf}\ (\omega \mid out\ s_{n+1} \gg\ ! \gg fail) &= right\ \omega \mid out\ s_{n+1} \gg fail \\
try_{nf}\ (\omega \mid out\ s_{n+1} \gg raise\ e) &= right\ \omega \mid out\ s_{n+1} \gg return\ (Left\ e)\ ,
\end{aligned}
$$

where *right* is given by

$$
\begin{aligned}
&right\ (out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n) \\
&\quad = \quad out\ s_1 \gg return\ (Right\ a_1) \mid \cdots \mid out\ s_n \gg return\ (Right\ a_n)\ .
\end{aligned}
$$

The following theorem shows the correctness of NF.

Theorem 1 *Let m be a computational term, then* NF $m = m$ *and* NF m *is in normal form.*

Proof. The proof proceeds by structural induction on the structure of $C\ a$. It is easy to see that the auxiliary functions take normal forms to normal forms, which immediately implies the second part of the theorem. It remains to show that the auxiliary functions preserve the meaning of the operations. The proofs are entirely straightforward, so we content ourselves with two exemplary cases. The first calculation shows $!_{nf} = !$.

$$
\begin{aligned}
!_{nf} &= out\ "" \gg return\ () \mid out\ "" \gg\ ! \gg fail && \text{Def. } !_{nf} \\
&= return\ () \mid\ ! \gg fail && out\ "" \gg m = m \\
&= \ ! \gg return\ () && \text{Eq.(1)} \\
&= \ ! && (!3)
\end{aligned}
$$

The second calculation proves $\omega_1 \mid_{nf} \omega_2 = \omega_1 \mid \omega_2$ where ω_1 and ω_2 are in normal form. **Case** $\omega_1 = \omega \mid out\ s \gg fail$:

$$
\begin{aligned}
\omega_1 \mid_{nf} \omega_2 &= \omega \mid s \rhd \omega_2 && \text{Def. } (\mid_{nf}) \\
&= \omega \mid out\ s \gg \omega_2 && x \rhd m = out\ x \gg m \\
&= \omega \mid (out\ s \gg fail \mid \omega_2) && \text{(B1) and Cor. 1} \\
&= \omega_1 \mid \omega_2\ . && (B3)
\end{aligned}
$$

Case $\omega_1 = \omega \mid out\ s \gg\ !\ \gg fail$:

$$\begin{array}{rcll} \omega_1 \mid_{nf} \omega_2 & = & \omega \mid out\ s \gg\ !\ \gg fail & \text{Def. } (\mid_{nf}) \\ & = & \omega \mid out\ s \gg (!\ \gg fail \mid \omega_2) & (!1) \\ & = & \omega \mid (out\ s \gg\ !\ \gg fail \mid \omega_2) & \text{Cor. 1} \\ & = & \omega_1 \mid \omega_2\ . & \text{(B3)} \end{array}$$

Case $\omega_1 = \omega \mid out\ s \gg raise\ e$: analogous. □

Let us conclude the section by giving a sample application of Theorem 1 proving the equality of two derived operations. Consider the two definitions of *atMostOnce* given in Section 3.3 and in Section 3.4. Since the two variants rely on different primitives — the first uses *once* while the second depends on ! and *call* — we cannot reasonably expect to find a 'direct' proof of their equality. Just imagine that we add an additional computational primitive that interacts differently with *once* and !. We can, however, show that they behave the same for all computational terms built according to Def. 1. To this end it suffices to show that the two definitions are equal for computational terms in normal form. Let $m = out\ s_1 \gg return\ a_1 \mid \cdots \mid out\ s_n \gg return\ a_n \mid out\ s_{n+1} \gg \alpha$. If $n \geqslant 1$ then we have that $atMostOnce\ m$ equals $out\ s_1 \gg return\ a_1$ in both cases. Otherwise we obtain $out\ s_{n+1} \gg fail$ for $\alpha = fail$ and $\alpha = !\ \gg fail$ and $out\ s_{n+1} \gg raise\ e$ for $\alpha = raise\ e$. Using a similar argument we can also prove that the definitions of *naf* given in Section 3.3 and in Section 5.9 coincide.

6. Implementation

Each of the class definitions listed in Section 3 defines a certain 'feature' or computational behaviour. In order to implement embedded Prolog we must develop a monad that 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 scratch. 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*.

6.1. Monad transformers

The idea of a monad transformer is again due to Moggi [32]: some of the monad transformers defined in this article already appear in *loc. cit.* albeit in an abstract form. Moggi's approach was later extended by Liang, Hudak, and Jones [25], 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 basically a type constructor τ that takes a monad m to a monad $\tau\ m$.

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

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.

Interestingly, we have already seen an example for a monad transformer: the type constructor *Parser* defined in Section 4.5. In essence, *Parser* augments a monad with state. The *MonadT* instance is given by

```
instance MonadT Parser where
    up m     = λinp → m ≫= λa → return (a, inp)
    down m   = m (error "no input given") ≫= λ(a, _) → return a  .
```

Note that *up m* simply runs m while preserving the current state (ie input). Conversely, *down m* runs m by supplying an undefined initial state and discarding the final state.

Both *up* and *down* have immediate applications. The premonad morphism *down* can be used to provide 'generic' instance declarations for *Run* and *Perform*.

```
instance (MonadT τ, Run m, Monad (τ m)) ⇒ Run (τ m) where
    run      = run ∘ down
instance (MonadT τ, Perform m, Monad (τ m)) ⇒ Perform (τ m) where
    perform  = perform ∘ down
```

To encapsulate a computation c of type $\tau\ m\ a$ we simply encapsulate *down c* of type $m\ a$. Note that $\eta_1 \circ \eta_2$ is a premonad morphism if both η_1 and η_2 are. Furthermore, $\eta_1 \circ \eta_2$ satisfies (η3) if both η_1 and η_2 do. As an aside, note that the instance context *Monad* $(\tau\ m)$ is necessary in both cases because *Run* and *Perform* are subclasses of *Monad*.

The monad morphism *up* is useful for lifting the I/O operations through an arbitrary monad transformer.

```
instance (MonadT τ, InOut m, Monad (τ m)) ⇒ InOut (τ m) where
    out s          = up (out s)
    inp            = up inp
    outFile path s = up (outFile path s)
    inpFile path   = up (inpFile path)
```

Generic implementations enjoy generic properties: given the above definition of *out s* it is easy to show that $\tau\ m$ satisfies (IO3) and (IO4) if m does and *up* is a monad morphism.

6.2. Base monads

The identity monad and the *IO* monad serve as base monads, upon which the monad transformers are applied. The identity monad identifies computations and

values. It is given by the following definition.

```
newtype Id a    =  a
instance Monad Id where
    return a    =  a
    a >>= f     =  f a
instance Run Id where
    run         =  id
```

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, (M1)–(M3), are satisfied.

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

```
instance Perform IO where
    perform   =  id
instance InOut IO where
    out       =  putStr
    inp       =  getLine
    outFile   =  appendFile
    inpFile   =  readFile
```

6.3. 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*.

```
newtype ExcT m a   =  m (Either Err a)
instance (Monad m) => Monad (ExcT m) where
    m >>= k        =  m >>= either (return . Left) k
    return         =  return . Right
instance (Monad m) => Exc (ExcT m) where
    raise          =  return . Left
    try m          =  map Right m
```

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*.

```
instance MonadT ExcT where
    up m       =  map Right m
    down m     =  m >>= either error return
```

Theorem 2 *Let m be a monad, then ExcT m is a monad satisfying* (E1)–(E5) *and* (IO2). *Furthermore, up is a monad morphism and down a premonad morphism. If m is an I/O monad, then down additionally satisfies* (η3).

Proof. The proofs of the different axioms are entirely straightforward. Note, however, that we tacitly assume that the datatype *Either* is unlifted, ie does not contain $\perp$. Otherwise, (M2) does not hold. □

6.4. The list monad

Most publications on monads employ lists for implementing nondeterministic computations: each element of a list represents a solution, the nondeterministic choice is implemented by concatenation.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ [\,]\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{return}\ a & = & [a] \\
m \gg\!= k & = & \mathit{list}\ \mathit{fail}\ (\lambda a\ n \to k\ a \mid n)\ m
\end{array} \\
\textbf{instance}\ \mathit{Backtr}\ [\,]\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{fail} & = & [\,] \\
m \mid n & = & m +\!\!+ n
\end{array}
\end{array}
$$

The list monad suffers from at least four problems. (i) It relies in an essential way on lazy evaluation. Thus we cannot use it in a strict language like Standard ML. (ii) It is inefficient: both ($\gg\!=$) and ($\mid$) are sensitive to the number of successes of their first argument. (iii) It cannot implement the cut. (iv) There is no obvious way to turn it into a monad transformer. Let us briefly indicate the reason for the last deficiency. It appears that the base monad can only sensibly combined with the list monad by functor composition.

$$
\begin{array}{lcl}
\textbf{newtype}\ \mathit{ListT}\ m\ a & = & m\ [a] \\
\mathit{single} & :: & (\mathit{Monad}\ m) \Rightarrow a \to m\ [a] \\
\mathit{single}\ a & = & \mathit{return}\ [a] \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{ListT}\ m)\ \textbf{where} & & \\
\quad \begin{array}{lcl}
\mathit{return}\ a & = & \mathit{single}\ a \\
m \gg\!= k & = & m \gg\!= \mathit{list}\ \mathit{nil}\ (\lambda a\ f \to k\ a \oplus f)
\end{array} & & \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Backtr}\ (\mathit{ListT}\ m)\ \textbf{where} & & \\
\quad \begin{array}{lcl}
\mathit{fail} & = & \mathit{nil} \\
m \mid n & = & m \oplus n
\end{array} & &
\end{array}
$$

Note that the definition of each operation is more or less inevitable. Unfortunately, $\mathit{ListT}\ m$ does only satisfy the monad laws, if m is a so-called *commutative monad* [22]. The crux is that $m \mid n = m \oplus n$ executes both m *and* n before the list of solutions is returned. If the base monad is not commutative, then (M3) fails. Take, for instance, $m = \mathit{IO}$ and consider $((\mathit{tr}\ 1 \mid \mathit{tr}\ 2) \gg\!= \mathit{tr}) \gg\!= (\mathit{tr} \mathbin{⦶} \mathit{tr})$ where tr is given by

$$
\begin{array}{lcl}
\mathit{tr} & :: & (\mathit{Monad}\ m, \mathit{InOut}\ m) \Rightarrow \mathit{Int} \to m\ \mathit{Int} \\
\mathit{tr}\ n & = & \textbf{do}\ \{\, \mathit{out}\ (\mathit{show}\ n); \mathit{return}\ n \,\}\ .
\end{array}
$$

If executed the above computation outputs 12121122. However, if we rebracket the expression applying (M3), we obtain 12111222.

In the sequel we will derive an alternative to $ListT$, which remedies the shortcomings mentioned above.

6.5. Backtracking monad transformer

A major disadvantage of the list monad is its inefficiency: the running time of $(\mid)$ alias $(+\!\!+)$ is proportional to the length of its first argument. Now, part of the functional programming folklore is a transformation technique for eliminating expensive calls to $(+\!\!+)$, see, for instance, [5, 17]. The technique is, for instance, applied in Haskell's *Show* class to guarantee linear complexity of the *show* functions. Instead of focusing on the list monad let us 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 $m = m'\ fail$. 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

$$\begin{array}{lcl} up\ m & = & \lambda f \to m \mid f \\ down\ m & = & m\ fail \end{array}$$

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

$$m\ f \quad = \quad m\ fail \mid f \tag{2}$$

holds. Using these prerequisites we can actually *derive* the monad operations of $m\ a \to m\ a$. To derive $fail$ we reason:

$$\begin{array}{lcll} fail & = & up\ fail & up \text{ should preserve } fail \\ & = & \lambda f \to fail \mid f & \text{Def. } up \\ & = & \lambda f \to f\ . & \text{(B1)} \end{array}$$

Thus, $fail$ equals the identity function. The derivation of $(\mid)$ proceeds as follows.

$$\begin{array}{lcll} m \mid n & = & up\ (down\ (m \mid n)) & up \circ down = id \\ & = & up\ (down\ m \mid down\ n) & down \text{ should preserve } (\mid) \\ & = & \lambda f \to (m\ fail \mid n\ fail) \mid f & \text{Def. } up/down \\ & = & \lambda f \to m\ fail \mid (n\ fail \mid f) & \text{(B3)} \\ & = & \lambda f \to m\ (n\ f) & \text{Eq.(2)} \end{array}$$

We see that $(\mid)$ boils down to the composition of functions. Interestingly, neither $fail$ nor $(\mid)$ depends on m's primitives. Unfortunately, this does not hold for return and bind. The latter operation, for instance, is given by

$$m \gg\!= k \quad = \quad \lambda f \to (m\ fail \gg\!= \lambda a \to k\ a\ fail) \mid f\ .$$

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

$$\begin{array}{lcl} up\ n & = & \lambda\kappa \to n \gg\!= \kappa \\ down\ n & = & n\ return \end{array}$$

and an isomorphism condition

$$n\ \kappa \quad = \quad n\ return \gg\!= \kappa\ . \tag{3}$$

Again, we can use these definitions to *derive* the monad operations. For *return* we obtain:

$$\begin{array}{lcll} return\ a & = & up\ (return\ a) & up \text{ should preserve } return \\ & = & \lambda\kappa \to return\ a \gg\!= \kappa & \text{Def. } up \\ & = & \lambda\kappa \to \kappa\ a\ , & \text{(M1)} \end{array}$$

and for bind:

$$\begin{array}{lcll} m \gg\!= k & = & up\ (down\ (m \gg\!= k)) & up \circ down = id \\ & = & up\ (down\ m \gg\!= down \circ k) & down \text{ should preserve } (\gg\!=) \\ & = & \lambda\kappa \to (m\ return \gg\!= \lambda a \to k\ a\ return) \gg\!= \kappa & \text{Def. } up/down \\ & = & \lambda\kappa \to m\ return \gg\!= \lambda a \to k\ a\ return \gg\!= \kappa & \text{(M3)} \\ & = & \lambda\kappa \to m\ (\lambda a \to k\ a\ \kappa)\ . & \text{Eq.(3)} \end{array}$$

Let us determine the type of the computations obtained by the two abstraction steps. 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 $n\ b = m\ b \to m\ b$, we arrive at the following definition.

$$\begin{array}{lcl} \textbf{type}\ CPS\ a\ ans & = & (a \to ans) \to ans \\ \textbf{newtype}\ BacktrT\ m\ a & = & \forall ans.CPS\ a\ (m\ ans \to m\ ans) \\ \textbf{instance}\ Monad\ (BacktrT\ m)\ \textbf{where} & & \\ \quad return\ a & = & \lambda\kappa \to \kappa\ a \\ \quad m \gg\!= k & = & \lambda\kappa \to m\ (\lambda a \to k\ a\ \kappa) \end{array}$$

The cognoscenti would certainly recognize that the above definition is identical to the definition of the *continuation monad transformer* [25]. Only the types are different: *BacktrT* involves second-order types[e] while the continuation monad transformer is additionally parameterized with the so-called answer type (*ContT ans m a*

[e] Haskell 98 does not admit local universal quantification. However, all major Haskell systems, GHC, HBC, and Hugs, provide the necessary extensions. We refer the interested reader to the work of Rémy [38] and Jones [21].

$= (a \to m\ ans) \to m\ ans$). 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 that allows us to add backtracking. Note, for instance, that *callcc* is definable in $ContT\ ans\ m$ but not in $BacktrT\ m$.

The monad transformer, $BacktrT\ m$, has the amazing property of being a monad regardless of m! The dependence on the base monad has completely vanished. Let's see if this is also proves to be true for *fail* and $(|)$. We reason:

$$\begin{array}{lcll} fail & = & up\ fail & up \text{ should preserve } fail \\ & = & \lambda\kappa \to fail \gg\!= \kappa & \text{Def. } up \\ & = & \lambda\kappa \to fail & \text{(B4)} \\ & = & \lambda\kappa \to id\ , & \text{Def. } fail \end{array}$$

and

$$\begin{array}{lcll} m \mid n & = & up\ (down\ (m \mid n)) & up \circ down = id \\ & = & up\ (down\ m \mid down\ n) & down \text{ should preserve } (|) \\ & = & \lambda\kappa \to (down\ m \mid down\ n) \gg\!= \kappa & \text{Def. } up \\ & = & \lambda\kappa \to down\ m \gg\!= \kappa \mid down\ n \gg\!= \kappa & \text{(B5)} \\ & = & \lambda\kappa \to up\ (down\ m)\ \kappa \mid up\ (down\ n)\ \kappa & \text{Def. } up \\ & = & \lambda\kappa \to m\ \kappa \mid n\ \kappa & up \circ down = id \\ & = & \lambda\kappa \to m\ \kappa \circ n\ \kappa\ . & \text{Def. } (|) \end{array}$$

Voilà. Neither *fail* nor $(|)$ require a primitive operation of the base monad m. In other words, we have succeeded in defining a *backtracking monad transformer* that adds backtracking to an arbitrary monad. Here is the necessary $MonadT$ instance.

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

The definition for $up\ m$ is obtained by composing the two *up*s introduced above. The result obtained must additionally be simplified taking into account that m is always deterministic. A similar remark applies to *down*.

$$\begin{array}{l} \textbf{instance}\ (Monad\ m) \Rightarrow Backtr\ (BacktrT\ m)\ \textbf{where} \\ \quad \begin{array}{lcl} fail & = & \lambda\kappa \to id \\ m \mid n & = & \lambda\kappa \to m\ \kappa \circ n\ \kappa \\ once\ m & = & up\ (m\ first\ nothing) \\ sols\ m & = & up\ (m\ cons\ nil) \end{array} \\ \begin{array}{lcl} first & :: & (Monad\ m) \Rightarrow a \to m\ (Maybe\ a) \to m\ (Maybe\ a) \\ first\ a\ _ & = & return\ (Just\ a) \end{array} \end{array}$$

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 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 only the first solution; *cons* on the other hand executes the failure continuation and constructs the desired list of solutions.

As matters stands $BacktrT\ m$ cannot implement the cut operator. We postpone a discussion of the necessary modifications until the next section.

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*.

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

The call $try\ (m\ first\ nothing)$ performs the encodings; the function δ then decodes the results and calls the appropriate continuations. Now, in order to make $try\ m$ re-satisfiable we must additionally pass the failure continuation, which *first* simply discards, to δ. Let us introduce a suitable datatype for this purpose.

$$
\begin{array}{lcl}
\textbf{data}\ Answer\ m\ a & = & No \\
 & | & Yes\ a\ (m\ (Answer\ m\ a)) \\
answer & :: & (Monad\ m) \Rightarrow m\ b \rightarrow (a \rightarrow m\ b \rightarrow m\ b) \\
 & & \qquad\qquad \rightarrow Answer\ m\ a \rightarrow m\ b \\
answer\ no\ yes\ No & = & no \\
answer\ no\ yes\ (Yes\ a\ m) & = & yes\ a\ (m \gg\!= answer\ no\ yes)
\end{array}
$$

The constructor *No* corresponds to *Nothing*; *Yes* extends *Just* by an additional field for the failure continuation. Note that the failure continuation yields an element of $Answer\ m\ a$, ie *Answer* is recursively defined. As usual, *answer* denotes the fold-functional of *Answer*.

$$
\begin{array}{l}
\textbf{instance}\ (Exc\ m) \Rightarrow Exc\ (BacktrT\ m)\ \textbf{where} \\
\quad \begin{array}{lcl}
raise & = & up \circ raise \\
try\ m & = & \lambda\kappa\ f \rightarrow \textbf{let}\ \delta\ (Left\ e) = \kappa\ (Left\ e)\ f \\
 & & \qquad\qquad \delta\ (Right\ No) = f \\
 & & \qquad\qquad \delta\ (Right\ (Yes\ a\ f')) = \kappa\ (Right\ a)\ (try\ f' \gg\!= \delta) \\
 & & \qquad\ \textbf{in}\ try\ (m\ yes\ no) \gg\!= \delta \\
no & :: & (Monad\ m) \Rightarrow m\ (Answer\ m\ a) \\
no & = & return\ No \\
yes & :: & (Monad\ m) \Rightarrow a \rightarrow m\ (Answer\ m\ a) \rightarrow m\ (Answer\ m\ a) \\
yes\ a\ m & = & return\ (Yes\ a\ m)
\end{array}
\end{array}
$$

The code shows that the decoder function δ is applied recursively to the failure continuation f'. This is necessary since f' may raise exceptions that must be trapped.

Using the backtracking monad transformer we are able to construct the monads required in the applications of Section 4: *BacktrT Id* supports backtracking, *BacktrT* (*ExcT IO*) supports backtracking, exception handling, and interaction. Furthermore, the implementation is correct with respect to the axiomatization given in Section 5.

Theorem 3 *Let m be a monad, then BacktrT m is a monad satisfying* (B1)–(B5), (O1)–(O4), (S1)–(S4) *and* (IO1). *Furthermore, up is a monad morphism and down a premonad morphism. If m is an exception monad satisfying* (E1)–(E5) *and* (IO2), *then BacktrT m additionally satisfies* (E1)–(E4), (E5'), (E6)–(E8), *and* (IO2). *If m is an I/O monad, then down additionally satisfies* (η3).

Proof. The proofs of the different axioms are mostly straightforward, so we content ourselves with two examples. Some of the calculations can be simplified by noting that deterministic computations in *BacktrT m* equal *up n* for some n. For instance, (O3) can be shown as follows.

$$
\begin{array}{lll}
 & (up\ m \gg\!= k) \mid n & \\
= & \lambda\kappa\ f \to (up\ m \gg\!= k)\ \kappa\ (n\ \kappa\ f) & \text{Def. } (\mid) \\
= & \lambda\kappa\ f \to (up\ m)\ (\lambda a \to k\ a\ \kappa)\ (n\ \kappa\ f) & \text{Def. } (\gg\!=) \\
= & \lambda\kappa\ f \to m \gg\!= \lambda a \to k\ a\ \kappa\ (n\ \kappa\ f) & \text{Def. } up \\
= & \lambda\kappa\ f \to m \gg\!= \lambda a \to (k\ a \mid n)\ \kappa\ f & \text{Def. } (\mid) \\
= & \lambda\kappa\ f \to (up\ m)\ (\lambda a \to (k\ a \mid n)\ \kappa)\ f & \text{Def. } up \\
= & up\ m \gg\!= \lambda a \to k\ a \mid n & \text{Def. } (\gg\!=)
\end{array}
$$

Since $once\ m = up\ (m\ first\ nothing)$ the claim follows immediately. In order to prove (E8) we require that the base monad m satisfies (E2).

$$
\begin{array}{lll}
 & try\ (return\ a \mid m) & \\
= & \lambda\kappa\ f \to try\ ((return\ a \mid m)\ yes\ no) \gg\!= \delta & \text{Def. } try \\
= & \lambda\kappa\ f \to try\ ((return\ a)\ yes\ (m\ yes\ no)) \gg\!= \delta & \text{Def. } (\mid) \\
= & \lambda\kappa\ f \to try\ (yes\ a\ (m\ yes\ no)) \gg\!= \delta & \text{Def. } return \\
= & \lambda\kappa\ f \to try\ (return\ (Yes\ a\ (m\ yes\ no))) \gg\!= \delta & \text{Def. } yes \\
= & \lambda\kappa\ f \to return\ (Right\ (Yes\ a\ (m\ yes\ no))) \gg\!= \delta & \text{(E2)} \\
= & \lambda\kappa\ f \to \kappa\ (Right\ a)\ (try\ (m\ yes\ no) \gg\!= \delta) & \text{(M1) and Def. } \delta \\
= & \lambda\kappa\ f \to \kappa\ (Right\ a)\ ((try\ m)\ \kappa\ f) & \text{Def. } try \\
= & \lambda\kappa\ f \to return\ (Right\ a)\ \kappa\ ((try\ m)\ \kappa\ f) & \text{Def. } return \\
= & return\ (Right\ a) \mid try\ m & \text{Def. } (\mid)
\end{array}
$$

The remaining axioms are proved accordingly. □

6.6. Implementing the cut

The cut discards all past choice points. Since the alternatives of a computations are represented by the failure continuation, the idea suggests itself that the cut simply replaces the current failure continuation by an initial failure continuation, which represents 'global' or 'total' failure. However, the realization of this idea is hindered by the fact that there is no single initial continuation. Each of the constructs *down*, *once*, *sols*, and *try* supplies a different initialization: *once*, for instance, uses *nothing* while *sols* employs *nil*. Thus, in order to implement the cut we must first standardize the initial failure continuation, which in turn implies that we must be more specific about the answer type. Currently, the answer type is universally quantified: $\forall ans.\ CPS\ a\ (m\ ans \to m\ ans)$. It appears that the type *Answer* defined in the previous section is a suitable datatype for representing answers (hence its name). Consequently, we specialize the backtracking monad transformer as follows.

$$\textbf{newtype}\ CutT\ m\ a \quad = \quad \forall ans.CPS\ a\ (m\ (Answer\ m\ ans) \to m\ (Answer\ m\ ans))$$

The function *no* serves as the standard initial failure continuation. We can now implement the cut operation as follows.

$$! \quad = \quad \lambda\kappa\ f \to \kappa\ ()\ no$$

The cut simply calls the success continuation replacing the failure continuation by *no*. If the cut is immediately followed by *fail*, *no* is called in effect terminating the current computation. Hence, the cut-fail combination boils down to $\lambda\kappa\ f \to no$.

Before discussing the implementation of the remaining operations let us briefly sketch an alternative approach. Instead of specializing the answer type we could alternatively supply an additional 'cut continuation', ie the initial failure continuation is passed as an explicit parameter.

$$\textbf{newtype}\ CutT'\ m\ a \quad = \quad \forall ans.CPS\ a\ (m\ ans \to m\ ans \to m\ ans)$$

Using this representation the cut is implemented by

$$! \quad = \quad \lambda\kappa\ c\ f \to \kappa\ ()\ c\ c\ .$$

This implementation naturally offers a greater degree of freedom. It allows us, for instance, to implement a variant of *try* that is transparent to cut, which is not possible with the first approach.

Turning back to the first representation note that the *Monad* and *Exc* instances for $CutT$ are identical to those of $BacktrT$ since $CutT$ is a type specialization of $BacktrT$. The remaining operations can be adapted by inserting appropriate type

coercions.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{MonadT}\ \mathit{CutT}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{up}\ m & = & \lambda\kappa\, f \to m \gg\!= \lambda a \to \kappa\ a\ f \\
\mathit{down}\ m & = & m\ \mathit{yes}\ \mathit{no} \gg\!= \mathit{answer}\ \mathit{err}\ (\lambda a\ _ \to \mathit{return}\ a)
\end{array} \\
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Backtr}\ (\mathit{CutT}\ m)\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{fail} & = & \lambda\kappa \to \mathit{id} \\
m \mid n & = & \lambda\kappa \to m\ \kappa \circ n\ \kappa \\
\mathit{once}\ m & = & \mathit{up}\ (m\ \mathit{yes}\ \mathit{no} \gg\!= \mathit{answer}\ \mathit{nothing}\ \mathit{first}) \\
\mathit{sols}\ m & = & \mathit{up}\ (m\ \mathit{yes}\ \mathit{no} \gg\!= \mathit{answer}\ \mathit{nil}\ \mathit{cons})
\end{array}
\end{array}
$$

Note that expressions of the form $m\ \kappa\, f$, which appear in the $\mathit{BacktrT}$ instances, have been systematically replaced by $m\ \mathit{yes}\ \mathit{no} \gg\!= \mathit{answer}\ f\ \kappa$. This idiom is also used in the implementation of *call*.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Cut}\ (\mathit{CutT}\ m)\ \textbf{where} \\
\quad \begin{array}{lcl}
! & = & \lambda\kappa\, f \to \kappa\ ()\ \mathit{no} \\
\mathit{call}\ m & = & \lambda\kappa\, f \to m\ \mathit{yes}\ \mathit{no} \gg\!= \mathit{answer}\ f\ \kappa
\end{array}
\end{array}
$$

Finally, we are able to construct a monad that supports all the computational features introduced in Section 3:

$$\textbf{type}\ \mathit{Prolog} \quad = \quad \mathit{CutT}\ (\mathit{ExcT}\ \mathit{IO})\ .$$

Note that we may view *Prolog* as *the standard model of embedded Prolog*. The following theorem in conjunction with Theorem 2 implies its correctness.

Theorem 4 *In addition to the properties listed in Theorem 3 CutT m satisfies* (!1)–(!5), (C1)–(C4), *and* (E9).

Proof. For reasons of space we only show (!1) and (!2). The proof for (!1) is:

$$
\begin{array}{rcll}
(! \gg m) \mid n & = & \lambda\kappa\, f \to (! \gg m)\ \kappa\ (n\ \kappa\, f) & \text{Def. } (\mid) \\
& = & \lambda\kappa\, f \to !\ (\lambda_ \to m\ \kappa)\ (n\ \kappa\, f) & \text{Def. } (\gg\!=) \\
& = & \lambda\kappa\, f \to m\ \kappa\ \mathit{no} & \text{Def. } ! \\
& = & \lambda\kappa\, f \to !\ (\lambda_ \to m\ \kappa)\ f & \text{Def. } ! \\
& = & ! \gg m\ . & \text{Def. } (\gg\!=)
\end{array}
$$

Eq.(!2) is proved as follows.

$$
\begin{array}{rcll}
! \gg (m \mid n) & = & \lambda\kappa\, f \to !\ (\lambda_ \to (m \mid n)\ \kappa)\ f & \text{Def. } (\gg\!=) \\
& = & \lambda\kappa\, f \to (m \mid n)\ \kappa\ \mathit{no} & \text{Def. } ! \\
& = & \lambda\kappa\, f \to m\ \kappa\ (n\ \kappa\ \mathit{no}) & \text{Def. } (\mid) \\
& = & \lambda\kappa\, f \to m\ \kappa\ (!\ (\lambda_ \to n\ \kappa)\ f) & \text{Def. } ! \\
& = & \lambda\kappa\, f \to m\ \kappa\ ((! \gg n)\ \kappa\, f) & \text{Def. } (\gg\!=) \\
& = & m \mid ! \gg n & \text{Def. } (\mid)
\end{array}
$$

The remaining axioms are shown accordingly. □

Table 1. Benchmark results

#queens	13		14	
#solutions	73712		365596	
C (`gcc-2.7.2.3 -O`)	2.0[a]	1.0[b]	14.0	1.0
Prolog (`eclipse-3.5.2`)	288.5	144.2	1799.1	128.5
ditto using logical variables	130.9	65.5	803.8	57.4
Mercury (`mercury-0.8 -O6`)	109.0	54.5	772.0	55.1
Haskell 1.2 (`ghc-0.29 -O`)	100.2	50.1	630.6	45.0
ditto using bit fields	48.0	24.0	296.1	21.2
Haskell 98 (`ghc-4.04 -O`)	119.0	59.5	751.5	53.7
Standard ML (`sml-110`)	324.5	162.3	2011.3	143.7

[a] seconds of user time on a Sun Ultra 5/10, 320 MB (minimum of four runs)
[b] time relative to the fastest implementation

To summarize: *BacktrT* and *CutT* remedy the shortcomings of the list monad mentioned in Section 6.4.[f] (i) They are independent of the order of evaluation.[g] (ii) They are efficient: both ($\gg=$) and ($|$) take constant time. (iii) *CutT* allows a simple implementation of the cut. (iv) They add backtracking to an arbitrary monad.

7. 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 [41], which offer nondeterminism as a 'language primitive'. To get a somewhat broader picture we also include C and Standard ML [31] in the competition. Due to the multi-lingual 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 a general tendency.

The source code of the programs appears in [15]. A few comments 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 4.2 specialized to *BacktrT ID*, ie overloading is resolved to improve efficiency.

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 45 slower, the factor reduces to 20 if bit fields are used. Perhaps surprisingly, Haskell — or rather, the

[f] Interestingly, the two approaches are not unrelated: the backtracking monad *BacktrT Id a* ($= \forall ans.\ (a \to ans \to ans) \to ans \to ans$) corresponds exactly to the type of the fold-functional for lists, which in turn describes the encoding of lists in the second-order λ-calculus [21].

[g] Actually, in a strict language the failure continuation must be a function to prevent its evaluation, ie *BacktrT m a* must be defined as $\forall ans.\ CPS\ a\ (CPS\ ()\ (m\ ans))$.

Glasgow Haskell Compiler [35] — compares favourably with Prolog and Mercury. It lies even in front of Standard ML of New Jersey [2]. 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 [14] relates the Glasgow Haskell Compiler to other systems. Compared to Haskell 1.2 the Haskell 98 code as presented in this article is 20% slower. The loss of performance is probably due to the use of type classes.

8. Related work

Monads and monad transformers Monads have been proposed by Moggi as a means to structure denotational semantics [32, 33], an aspect which is also present in this article: the monad *Prolog* can be seen as a structured denotational semantics for embedded Prolog. In a series of papers [43, 44, 46] Wadler popularized Moggi's idea in the functional programming community by using monads to structure functional programs, in particular, interpreters. The question soon arose as how to construct monads that support a variety of features. Early attempts to combine monads by composition [23, 22] were only mildly successful. The list monad, for instance, composes only with commutative monads. Taking up an idea of Moggi, Espinosa [10] and later Liang, Hudak, and Jones [25] employed monad transformers to construct monads in a modular fashion. Our development in Section 6 is mainly based on the latter article.

Semantics There is an abundance of work on the semantics of Prolog's horn logic core (possibly extended by the negation as failure rule), see [26]. Comparatively few attempts have been made to cover the full language. De Bruin and de Vink [9] describe an operational semantics and a denotational continuation semantics for Prolog's control core (ie backtracking and cut ignoring unification). The continuation semantics employs three continuations (a success, a failure, and a cut continuation) corresponding closely to the monad transformer $CutT'$ of Section 6.6. Meijer [28] shows how to derive low-level machine implementations from the continuation semantics. An operational and a denotational semantics covering the same subset of Prolog is also presented by Billaud [4]. His work can be seen as a precursor to the axiomatization given in Section 5. He identifies so-called equivalence rules for the control constructs, which roughly correspond to the axioms (M1)–(M3), (B1)–(B5), (!1) and $! \gg ! = !$ (axiom (!2) is, however, missing). Andrews [1] later showed that Billaud's language can also be characterized declaratively using a Fitting-style model theory. The most comprehensive treatment of Prolog's semantics was given by Börger and Rosenzweig [6]. Using Gurevich's notion of evolving algebras (later called Abstract State Machines) [11] they provide a logical operational semantics for full Prolog including control constructs, database operations, all solution collecting predicates, and error handling.

The axiomatic approach taken in this paper originates, of course, in the work of

Moggi. A further source of inspiration is [46] where Wadler identifies properties of functional parsers. To the best of our knowledge, the unified treatment of control constructs including the cut, all solution predicates, exception handling, and interaction is original. A distinguished advantage of the axiomatic approach based on monads is that the programmer can use the same language for programming and for reasoning about programs. In particular, there is no need to refer to an external model as in the denotational or operational approach.

Implementation Implementing backtracking using so-called continuation-passing style (CPS) is by no means a new idea. An early account appears in [29] where CPS is used to integrate Prolog into the LISP dialect POP-11. Danvy and Filinski [8] show that backtracking can be implemented using two CPS conversions (ie $CPS\ a\ (CPS\ ()\ ans)$). Interestingly, both approaches can be recast in monadic terms and lead after some simplifications to the structure presented in Section 6.5 [15]. Success continuations are also employed in the Lisp-based Prolog interpreter of Carlsson [7]. In a paper about pretty printing [16] Hughes derives a so-called context-passing implementation of backtracking. The essential idea is to give a function access to its own calling context by passing it as an additional parameter. Hughes shows that for monads with *fail* and $(\mid)$ it is sufficient to consider contexts of the form $run\ (\bullet \gg\!= \kappa \mid f)$. Though the details differ, Hughes obtains after some simplifications essentially the same implementation as we do (ie $BacktrT$ specialized to Id). The main difference to the calculations of Section 6.5 is that Hughes derives a backtracking monad from scratch while we aim at improving the efficiency of a given backtracking monad.

Acknowledgements

Thanks are due to the FLOPS'98 referees and to the anonymous referee of the special issue who provided detailed and constructive comments. Furthermore, I would like to thank Jeremy Gibbons for improving my English.

References

1. James Andrews. A paralogical semantics for the Prolog cut. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 591–608, Cambridge, December 1995. MIT Press.
2. 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.
3. Bijan Arbab and Daniel M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4(4):309–329, December 1987.
4. Michel Billaud. Simple operational and denotational semantics for Prolog with cut. *Theoretical Computer Science*, 71(2):193–208, March 1990.
5. R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

6. Egon Börger and Dean Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, June 1995.
7. Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2:347–359, 1984.
8. 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.
9. A. de Bruin and E.P. de Vink. Continuation semantics for prolog with cut. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 1*, volume 351 of *Lecture Notes in Computer Science*, pages 178–192, Berlin, March 1989. Springer-Verlag.
10. David A. Espinosa. *Semantic Lego.* Ph.d. thesis, Columbia University, 1995.
11. Y. Gurevich. Evolving algebras. A tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.
12. Michael Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science*, 89(1):63–106, October 1991.
13. Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
14. 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.
15. Ralf Hinze. Efficient monadic-style backtracking. Technical Report IAI-TR-96-9, Institut für Informatik III, Universität Bonn, October 1996.
16. John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–93. Springer-Verlag, 1995.
17. R. John Muir Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, March 1986.
18. Graham Hutton and Erik Meijer. Functional Pearl: Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), July 1998.
19. *Information technology — Programming languages — Prolog — Part 1: General core.* ISO/IEC JTC 1/SC22/WG17, June 1995.
20. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.
21. 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.
22. Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.
23. David J. King and Philip Wadler. Combining monads. In J. Launchbury and P.M. Sansom, editors, *Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland*, Workshops in Computing, pages 134–143. Springer-Verlag, July 1993.
24. Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the 1992 ACM Workshop on ML and its Applications, San Francisco*, pages 78–91. Association for Computing Machinery, June 1992.

25. 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.

26. John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

27. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

28. Erik Meijer. *Calculating Compilers.* PhD thesis, Nijmegen University, 1992.

29. 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.

30. C.S. Mellish and W.F. Clocksin. *Programming in Prolog.* Springer-Verlag, Berlin, fourth edition edition, 1995.

31. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

32. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University, 1990.

33. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

34. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: Exploring the design space. In *Proceedings of the Haskell Workshop*, 1997.

35. 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.

36. 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.

37. Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.

38. 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.

39. Niklas Röjemo. Efficient parsing combinators. unpublished, May 1995.

40. Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark*, pages 233–242. ACM-Press, June 1993.

41. 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.

42. Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.
43. Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78. ACM-Press, June 1990.
44. Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Sante Fe, New Mexico*, pages 1–14, January 1992.
45. Philip Wadler. How to declare an imperative (invited paper). In John Lloyd, editor, *ILPS'95: International Logic Programming Symposium*. MIT Press, December 1995.
46. 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.