Deriving Monad Transformers

January 1999

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* `ralf@informatik.uni-bonn.de`)

Abstract

In a paper about pretty printing J. Hughes introduced two fundamental techniques for deriving programs from their specification, where a specification consists of a signature and properties that the operations of the signature are required to satisfy. Very briefly, the first technique, the term implementation, represents the operations by terms and works by defining a mapping from operations to observations — this mapping can be seen as defining a simple interpreter. The second, the context-passing implementation, represents operations as functions from their calling context to observations. We apply both techniques to derive among others a backtracking monad transformer which adds backtracking to an arbitrary monad. In addition to the usual backtracking operations — failure and nondeterministic choice — the prolog cut and an operation for delimiting the effect of a cut are supported.

1 Introduction

Why should one derive a program from its specification? Ideally, a derivation explains and motivates the design choices taken in a particular implementation. At best a derivation eliminates the need for so-called eureka steps which are usually inevitable if a program is explained, say, by means of example.

In a paper about pretty printing J. Hughes (1995) introduced two fundamental techniques for deriving programs from their specification. Both techniques provide the programmer with considerable guidance in the process of program derivation. To illustrate their utility and versatility we apply the proposed framework to derive several monad transformers which among other things add backtracking to an arbitrary monad. Briefly, a monad transformer is a mapping on monads which augments a given monad by a certain computational feature such as state, exceptions, or nondeterminism. Traditionally, monad transformers are introduced in a single big eureka step. Even the recent introductory textbook on functional programming (Bird, 1998) fails to explain the particular definitions of monad transformers. After defining an exception monad transformer R. Bird remarks: “Why have we chosen to write [...]? The answer is: because it works.”. Building upon Hughes’ techniques we will try to provide a more satisfying answer. The reader should be prepared, however, that the results are somewhat different from the standard textbook examples.

The paper is organized as follows. Sec. 2 reviews monads and monad transformers. Sec. 3 introduces Hughes' techniques by means of a simple example. Sec. 4 applies the framework to derive a backtracking monad transformer which adds backtracking to an arbitrary monad. Finally, Sec. 5 extends the design of Sec. 4 to include additional control constructs: Prolog's cut and an operation for delimiting the effect of cut. Finally, Sec. 6 concludes and points out directions for future work.

2 Preliminaries

Monads have been proposed by Moggi as a means to structure denotational semantics (1990; 1991). Wadler popularized Moggi's idea in the functional programming community by using monads to structure functional programs (1990; 1992; 1995). In Haskell 98 (Peyton Jones [editor] *et al.*, 1998) monads are captured by the following class definition.

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{return} & :: & a \to m\ a \\
(\gg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b \\
(\gg) & :: & m\ a \to m\ b \to m\ b \\
\mathit{fail} & :: & \mathit{String} \to m\ a \\
m \gg n & = & m \gg\!= \mathit{const}\ n \\
\mathit{fail}\ s & = & \mathit{error}\ s
\end{array}
\end{array}
$$

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 which yields a value of type a. The trivial computation which 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. The operation *fail* is useful for signaling error conditions and will be used to this effect. Note that *fail* does not stem from the mathematical concept of a monad, but has been added to the monad class for pragmatic reasons, see (Peyton Jones *et al.*, 1998, Sec. 3.14).

The operations are required to satisfy the following so-called *monad laws*.

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

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

$$(m \gg\!= k_1) \gg\!= k_2 \quad = \quad m \gg\!= (\lambda a \to k_1\ a \gg\!= k_2) \tag{M3}$$

For an explanation of the laws we refer the reader to (Bird, 1998, Sec. 10.3). Note that *fail* is intentionally left unspecified.

Different monads are distinguished by the computational features they support. Each computational feature is typically accessed through a number of additional operations. For instance, a backtracking monad additionally supports the operations *false* and $(|)$ denoting failure and nondeterministic choice. It is relatively easy to construct a monad which supports only a single computational feature. Unfortunately, there is no uniform way of combining two monads which support different computational features. The reason is simply that two features may interact in different ways. There is, however, a uniform method for augmenting a given monad by a certain computational feature. This method is captured by the following class definition which introduces *monad transformers* (Liang *et al.*, 1995).

$$\begin{array}{l} \textbf{class}\ \mathit{Transformer}\ \tau\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{promote} & :: & (\mathit{Monad}\ m) \Rightarrow m\ a \to \tau\ m\ a \\ \mathit{observe} & :: & (\mathit{Monad}\ m) \Rightarrow \tau\ m\ a \to m\ a \end{array} \end{array}$$

A monad transformer is basically a type constructor τ that takes a monad m to a monad $\tau\ m$. It must additionally provide two operations: an operation for embedding computations from the underlying monad into the transformed monad and an inverse operation which allows to observe 'augmented' computations in the underlying monad. Since *observe* forgets structure, it will in general be a partial function. Turning to the laws we require promotion to respect the monad operations.

$$\mathit{promote}\ (\mathit{return}\ a) = \mathit{return}\ a \tag{P1}$$

$$\mathit{promote}\ (m \gg\!= k) = \mathit{promote}\ m \gg\!= (\mathit{promote} \cdot k) \tag{P2}$$

These laws determine *promote* as a *monad morphism*. In general, *promote* should respect every operation the underlying monad provides in order to guarantee that a program that does not use new features behaves the same in the underlying and in the transformed monad. The counterpart of *promote* is not quite a monad morphism.

$$\mathit{observe}\ (\mathit{return}\ a) = \mathit{return}\ a \tag{O1}$$

$$\mathit{observe}\ (\mathit{promote}\ m \gg\!= k) = m \gg\!= (\mathit{observe} \cdot k) \tag{O2}$$

The second law is weaker than the corresponding law for *promote*. It is unreasonable to expect more since computations in $\tau\ m$ can, in general, not be mimicked in m.

3 Adding abnormal termination

This section reviews Hughes' technique by means of a simple example. We show how to augment a given monad by an operation which allows to terminate a computation abnormally. Monads with additional features are introduced as subclasses of *Monad*.

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

The call *raise e* terminates the current computation. This property is captured by the following law:

$$\mathit{raise}\ e \gg\!= k = \mathit{raise}\ e\ , \tag{R1}$$

which formalizes that *raise e* is a left zero of $(\gg\!=)$. Now, let's try to derive a monad transformer for this feature. Beforehand, we must determine how *raise e* is observed in the base monad. We specify

$$\mathit{observe}\ (\mathit{raise}\ e) \quad = \quad \mathit{fail}\ e\ , \tag{O3}$$

which appears to be the only reasonable choice since we know nothing of the underlying monad.

Remark. It is somehow unsatisfactory that exceptions signaled by *raise* cannot be trapped. We decided, however, not to consider additional operations (such as *handle*) in order to keep the introductory example short and simple.

3.1 A term implementation

The term implementation represents operations simply by terms of the algebra and works by defining an interpreter for the language. Since we have four operations — *return*, ($\gg=$), *raise*, and *promote* — the datatype which implements the term algebra consequently comprises four constructors. We adopt the convention that monad transformers are given names that are all in upper case. For the constructor names we re-use the names of the operations with the first letter in upper case; operators like ($\gg=$) are prefixed by a colon.

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{RAISE}\ m\ a & = & \mathit{Return}\ a \\
& \mid & (\mathit{RAISE}\ m\ b) :\!\gg= (b \to \mathit{RAISE}\ m\ a) \\
& \mid & \mathit{Raise}\ \mathit{Exception} \\
& \mid & \mathit{Promote}\ (m\ a)
\end{array}$$

Note that the definition involves an existentially quantified type variable (Läufer & Odersky, 1992) in the type of ($:\!\gg=$). Though Haskell 98 does not allow existential quantification, all major Haskell systems provide the necessary extensions.

Now, each of the operations *return*, ($\gg=$), *raise*, and *promote* is implemented by the corresponding constructor. In other words, the operations do nothing. All the work is performed by *observe* which can be seen as defining a tiny interpreter for the monadic language. Except for one case the definition of *observe* is straightforward.

$$\begin{array}{lcl}
\mathit{observe}\ (\mathit{Return}\ a) & = & \mathit{return}\ a \\
\mathit{observe}\ (m :\!\gg= k) & = & \qquad\qquad \\
\mathit{observe}\ (\mathit{Promote}\ m) & = & m \\
\mathit{observe}\ (\mathit{Raise}\ e) & = & \mathit{fail}\ e
\end{array}$$

Can we fill in the blank on the right-hand side? Unfortunately not, it appears impossible to define *observe* ($m :\!\gg= k$) in terms of its constituents. The only way out of this dilemma is to make a further case distinction on m.

$$\begin{array}{lcl}
\mathit{observe}\ (\mathit{Return}\ a :\!\gg= k) & = & \mathit{observe}\ (k\ a) \\
\mathit{observe}\ ((m :\!\gg= k_1) :\!\gg= k_2) & = & \mathit{observe}\ (m :\!\gg= (\lambda a \to k_1\ a :\!\gg= k_2)) \\
\mathit{observe}\ (\mathit{Promote}\ m :\!\gg= k) & = & m \gg= (\mathit{observe} \cdot k) \\
\mathit{observe}\ (\mathit{Raise}\ e :\!\gg= k) & = & \mathit{fail}\ e
\end{array}$$

Voilá. Each equation is a simple consequence of the monad laws and the laws for *observe*. In particular, the second equation employs (M3), the associative law for

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{RAISE}\ m\ a & = & \mathit{Return}\ a \\
 & | & \forall b.(\mathit{RAISE}\ m\ b) :\!\!\gg\!\!= (b \to \mathit{RAISE}\ m\ a) \\
 & | & \mathit{Raise}\ \mathit{Exception} \\
 & | & \mathit{Promote}\ (m\ a) \\
\mathbf{instance}\ \mathit{Monad}\ (\mathit{RAISE}\ m)\ \mathbf{where} & & \\
\quad \mathit{return} & = & \mathit{Return} \\
\quad (\gg\!=) & = & (:\!\!\gg\!\!=) \\
\quad \mathit{fail} & = & \mathit{Raise} \\
\mathbf{instance}\ \mathit{Raise}\ (\mathit{RAISE}\ m)\ \mathbf{where} & & \\
\quad \mathit{raise} & = & \mathit{Raise} \\
\mathbf{instance}\ \mathit{Transformer}\ \mathit{RAISE}\ \mathbf{where} & & \\
\quad \mathit{promote} & = & \mathit{Promote} \\
\quad \mathit{observe}\ (\mathit{Return}\ a) & = & \mathit{return}\ a \\
\quad \mathit{observe}\ (\mathit{Return}\ a :\!\!\gg\!\!= k) & = & \mathit{observe}\ (k\ a) \\
\quad \mathit{observe}\ ((m :\!\!\gg\!\!= k_1) :\!\!\gg\!\!= k_2) & = & \mathit{observe}\ (m :\!\!\gg\!\!= (\lambda a \to k_1\ a :\!\!\gg\!\!= k_2)) \\
\quad \mathit{observe}\ (\mathit{Raise}\ e :\!\!\gg\!\!= k) & = & \mathit{fail}\ e \\
\quad \mathit{observe}\ (\mathit{Promote}\ m :\!\!\gg\!\!= k) & = & m \gg\!= (\mathit{observe} \cdot k) \\
\quad \mathit{observe}\ (\mathit{Raise}\ e) & = & \mathit{fail}\ e \\
\quad \mathit{observe}\ (\mathit{Promote}\ m) & = & m
\end{array}
$$

Fig. 1. A term implementation of *RAISE*.

$(\gg\!=)$, to reduce the size of $(:\!\!\gg\!\!=)$'s first argument. This rewrite step is analogous to rotating a binary tree to the right. Fig. 1 summarizes the term implementation. Note that we use GHC/Hugs syntax for existential quantification (Peyton Jones, 1998): the existentially quantified variable is bound by an explicit *universal* quantifier written *before* the constructor.

What about correctness? First of all, the definition of *observe* is exhaustive. It is furthermore terminating since the size of $(:\!\!\gg\!\!=)$'s left argument is steadily decreasing. We can establish termination using a so-called *polynomial interpretation* of the operations (Dershowitz & Jouannaud, 1990).

$$
\begin{array}{rcl}
\mathit{Return}_\tau\ a & = & 1 \\
m :\!\!\gg_\tau n & = & 2 \times m + n \\
\mathit{Raise}_\tau\ e & = & 1 \\
\mathit{Promote}_\tau\ m & = & 1
\end{array}
$$

A multivariate polynomial of n variables is associated with each n-ary operation. For each equation $\mathit{observe}\ \ell = \ldots\ \mathit{observe}\ r\ \ldots$ we must show that $\tau\ \ell > \tau\ r$ for all variables (ranging over positive integers). Note that we consider bind only for the special case that the result of the first argument is ignored. The inclusion of $m :\!\!\gg\!\!= k$ in its full generality is feasible but technically more involving since the interpretation of k depends on the value m computes.

Does the implementation satisfy its specification? Since we are working in the free algebra, the laws do not hold: the expressions *Return a* and $\mathit{Return}\ a :\!\!\gg\!\!= \mathit{Return}$, for example, are distinct, unrelated terms. The laws of the specification only hold *under observation*. The monad laws, for instance, become

$$\begin{aligned} observe\ (return\ a \ggg\!= k) &= observe\ (k\ a) \\ observe\ (m \ggg\!= return) &= observe\ m \\ observe\ ((m \ggg\!= k_1) \ggg\!= k_2) &= observe\ (m \ggg\!= (\lambda a \rightarrow k_1\ a \ggg\!= k_2))\ . \end{aligned}$$

The first and the third are direct consequences of *observe*'s definition. The second can be shown by induction on m. Fortunately, we can live with the weakened laws, since the only way to run computations of type *RAISE m* is to use *observe*.

3.2 A simplified term implementation

Can we do better than the naive term implementation? A major critic of the first attempt is that the operations do not exploit the algebraic laws. It is conceivable that we can work with a subset of the term algebra. For instance, we need not represent both *Raise e* and *Raise e* $:\ggg\!=$ *Return*. A rather systematic way to determine the required subset of terms is to program a simplifier for the datatype *RAISE*, which exploits the algebraic laws as far as possible. It turns out that we only need to modify *observe* slightly.

$$\begin{aligned} &simplify && :: && RAISE\ m\ a \rightarrow RAISE\ m\ a \\ &simplify\ (Return\ a) && = && Return\ a \\ &simplify\ (Return\ a :\ggg\!= k) && = && simplify\ (k\ a) \\ &simplify\ ((m :\ggg\!= k_1) :\ggg\!= k_2) && = && simplify\ (m :\ggg\!= (\lambda a \rightarrow k_1\ a :\ggg\!= k_2)) \\ &simplify\ (Raise\ e :\ggg\!= k) && = && Raise\ e \\ &simplify\ (Promote\ m :\ggg\!= k) && = && Promote\ m :\ggg\!= (simplify \cdot k) \\ &simplify\ (Raise\ e) && = && Raise\ e \\ &simplify\ (Promote\ m) && = && Promote\ m \end{aligned}$$

Inspecting the right hand sides we see that we require ($:\ggg\!=$) only in conjunction with *Promote*. Since *promote m* is furthermore equivalent to *promote m* $\ggg\!=$ *return* we can, in fact, restrict ourselves to the following subset of the term algebra.

$$\begin{aligned} \mathbf{data}\ RAISE\ m\ a \ &= \ Return\ a \\ &\mid \ PromoteBind\ (m\ b)\ (b \rightarrow RAISE\ m\ a) \\ &\mid \ Raise\ Exception \end{aligned}$$

Following Hughes (1995) we call elements of the new datatype *simplified terms*. We avoid the term normalform or canonical form since distinct terms may not necessarily be semantically different. For instance, *return a* can be represented both by *Return a* and *PromoteBind* (*return a*) *Return*. Nonetheless, using this representation the definition of *observe* is much simpler. It is, in fact, directly based on the laws (O1)–(O3). The complete implementation appears in Fig. 2. If we are only interested in defining a monad (not a monad transformer), then we can omit the constructor *PromoteBind*. The resulting datatype corresponds exactly to the standard definition of the exception monad.

What about efficiency? The naive implementation — or rather, the first definition

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{RAISE}\ m\ a & = & \mathit{Return}\ a \\
 & \mid & \forall b.\mathit{PromoteBind}\ (m\ b)\ (b \to \mathit{RAISE}\ m\ a) \\
 & \mid & \mathit{Raise}\ \mathit{Exception} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Monad}\ (\mathit{RAISE}\ m)\ \textbf{where}} \\
\quad \mathit{return}\ a & = & \mathit{Return}\ a \\
\quad \mathit{Return}\ a \ggg\!= k & = & k\ a \\
\quad (\mathit{PromoteBind}\ m\ \kappa) \ggg\!= k & = & \mathit{PromoteBind}\ m\ (\lambda a \to \kappa\ a \ggg\!= k) \\
\quad \mathit{Raise}\ e \ggg\!= k & = & \mathit{Raise}\ e \\
\quad \mathit{fail}\ s & = & \mathit{Raise}\ s \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Raise}\ (\mathit{RAISE}\ m)\ \textbf{where}} \\
\quad \mathit{raise}\ e & = & \mathit{Raise}\ e \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Transformer}\ \mathit{RAISE}\ \textbf{where}} \\
\quad \mathit{promote}\ m & = & \mathit{PromoteBind}\ m\ \mathit{Return} \\
\quad \mathit{observe}\ (\mathit{Return}\ a) & = & \mathit{return}\ a \\
\quad \mathit{observe}\ (\mathit{PromoteBind}\ m\ k) & = & m \ggg\!= (\mathit{observe} \cdot k) \\
\quad \mathit{observe}\ (\mathit{Raise}\ e) & = & \mathit{fail}\ e
\end{array}
$$

Fig. 2. A simplified term implementation of *RAISE*.

of *observe* has a running time which is proportional to the size of the computation. Unfortunately, the 'improved' term implementation has a quadratic worst-case behaviour. Consider the expression

$$\mathit{observe}\ (\cdots\ ((\mathit{promote}\ (\mathit{return}\ 0) \ggg\!= \mathit{inc}) \ggg\!= \mathit{inc})\ \cdots \ggg\!= \mathit{inc})\ .$$

where *inc* is given by $\mathit{inc}\ n = \mathit{promote}\ (\mathit{return}\ (n+1))$. Since the amortized running time of bind is proportional to the size of its first argument, it takes $O(n^2)$ steps to evaluate the expression above. Bad luck.

3.3 A context-passing implementation

Since we cannot improve the implementation of the operations without sacrificing the runtime efficiency, let us try to improve the definition of *observe*. For a start, we can avoid some pattern matching if we specialize *observe* for $m \ggg\!= k$. To this end we replace the equations concerning $(\ggg\!=)$ by the single equation

$$\mathit{observe}\ (m :\!\ggg\!= \kappa) \quad = \quad \mathit{observe}_1\ m\ \kappa$$

and define $\mathit{observe}_1$ by

$$
\begin{array}{lcl}
\mathit{observe}_1\ (\mathit{Return}\ a)\ \kappa & = & \mathit{observe}\ (\kappa\ a) \\
\mathit{observe}_1\ (m :\!\ggg\!= k)\ \kappa & = & \mathit{observe}_1\ m\ (\lambda a \to k\ a :\!\ggg\!= \kappa) \\
\mathit{observe}_1\ (\mathit{Promote}\ m)\ \kappa & = & m \ggg\!= \lambda a \to \mathit{observe}\ (k\ a) \\
\mathit{observe}_1\ (\mathit{Raise}\ e)\ \kappa & = & \mathit{fail}\ e\ .
\end{array}
$$

Interestingly, the parameter κ is used twice in conjunction with *observe*. In an attempt to eliminate the mutual recursive dependence on *observe* we could try to pass $\mathit{observe} \cdot \kappa$ as a parameter instead of κ. We formalize this idea as follows (we baptize the new observer function $\underline{observe}$).

$$\begin{aligned}\underline{observe}\ m\ \underline{\kappa} &= observe\ (m \mathbin{:\gg=} \kappa)\\ \Longleftarrow\quad \underline{\kappa}\ a &= observe\ (\kappa\ a)\end{aligned}\tag{1}$$

It is immediate that this specification works for *Return* a, *Promote* m, and *Raise* e. It remains to infer the definition for $m \mathbin{:\gg=} k$. We assume that the precondition (1) holds — note that the equation number refers to the precondition only — and reason.

$$\begin{aligned}
& \underline{observe}\ (m \mathbin{:\gg=} k)\ \underline{\kappa}\\
=\ & \quad \{\text{ specification and assumption (1) }\}\\
& observe\ ((m \mathbin{:\gg=} k) \mathbin{:\gg=} \kappa)\\
=\ & \quad \{\text{ definition } observe\ \}\\
& observe\ (m \gg= (\lambda a \to k\ a \mathbin{:\gg=} \kappa))\\
=\ & \quad \{\text{ specification }\}\\
& \underline{observe}\ m\ (\lambda a \to observe\ (k\ a \mathbin{:\gg=} \kappa))\\
=\ & \quad \{\text{ specification and assumption (1) }\}\\
& \underline{observe}\ m\ (\lambda a \to \underline{observe}\ (k\ a)\ \underline{\kappa})
\end{aligned}$$

Voilá. The dependence on *observe* has vanished. To summarize, $\underline{observe}$ is given by

$$\begin{aligned}
\underline{observe}\ (Return\ a) &= \lambda\underline{\kappa} \to \underline{\kappa}\ a\\
\underline{observe}\ (m \mathbin{:\gg=} k) &= \lambda\underline{\kappa} \to \underline{observe}\ m\ (\lambda a \to \underline{observe}\ (k\ a)\ \underline{\kappa})\\
\underline{observe}\ (Raise\ e) &= \lambda\underline{\kappa} \to fail\ e\\
\underline{observe}\ (Promote\ m) &= \lambda\underline{\kappa} \to m \gg= \underline{\kappa}\ .
\end{aligned}$$

Note that the constructors appear only on the left-hand sides. This means that we are finally able to remove the interpretative layer, ie *return* a can be implemented directly by $\lambda\underline{\kappa} \to \underline{\kappa}\ a$ instead of *Return*. Silently, we have converted the term implementation into a *context-passing implementation*. To see why the term 'context-passing' is appropriate, re-consider the specification of the context-passing implementation.

$$\begin{aligned}m\ \underline{\kappa} &= observe\ (m \gg= \kappa)\\ \Longleftarrow\quad \underline{\kappa}\ a &= observe\ (\kappa\ a)\end{aligned}\tag{2}$$

The parameter $\underline{\kappa}$ of m can be seen as a representation of m's calling context $observe\ (\bullet \gg= \kappa)$ — here and in what follows we represent a context by an expression that has a hole in it. This is the nub of the story: every operation knows the context in which it is called and it is furthermore able to access and to rearrange the context. This gives the implementor a much greater freedom of manoeuvre as compared to the simplified term algebra. For instance, $(\gg=)$ can use the associative law to improve efficiency. By contrast, $(\gg=)$ of the simplified term variety does not know of any outer binds and consequently falls into the efficiency trap.

It is quite instructive to infer the operations of the context-passing implementation from the scratch using the above specification. Fig. 3 summarizes the calcu-

<u>specification</u>

$$
\begin{array}{l}
m\ \underline{\kappa} = \mathit{observe}\ (m \mathbin{>\!\!>\!\!=} \kappa) \\
\Longleftarrow \quad \underline{\kappa}\ a = \mathit{observe}\ (\kappa\ a) \qquad (2)
\end{array}
$$

$$
\begin{array}{ll}
 & (\mathit{return}\ a)\ \underline{\kappa} \\
= & \quad \{\ \text{spec. and ass. (2)}\ \} \\
 & \mathit{observe}\ (\mathit{return}\ a \mathbin{>\!\!>\!\!=} \kappa) \\
= & \quad \{\ \text{(M1)}\ \} \\
 & \mathit{observe}\ (\kappa\ a) \\
= & \quad \{\ \text{ass. (2)}\ \} \\
 & \underline{\kappa}\ a
\end{array}
$$

$$
\begin{array}{ll}
 & (m \mathbin{>\!\!>\!\!=} k)\ \underline{\kappa} \\
= & \quad \{\ \text{spec. and ass. (2)}\ \} \\
 & \mathit{observe}\ ((m \mathbin{>\!\!>\!\!=} k) \mathbin{>\!\!>\!\!=} \kappa) \\
= & \quad \{\ \text{(M3)}\ \} \\
 & \mathit{observe}\ (m \mathbin{>\!\!>\!\!=} (\lambda a \to k\ a \mathbin{>\!\!>\!\!=} \kappa)) \\
= & \quad \{\ \text{spec.}\ \} \\
 & m\ (\lambda a \to \mathit{observe}\ (k\ a \mathbin{>\!\!>\!\!=} \kappa)) \\
= & \quad \{\ \text{spec. and ass. (2)}\ \} \\
 & m\ (\lambda a \to k\ a\ \underline{\kappa})
\end{array}
$$

$$
\begin{array}{ll}
 & (\mathit{raise}\ e)\ \underline{\kappa} \\
= & \quad \{\ \text{spec. and ass. (2)}\ \} \\
 & \mathit{observe}\ (\mathit{raise}\ e \mathbin{>\!\!>\!\!=} \kappa) \\
= & \quad \{\ \text{(R1)}\ \} \\
 & \mathit{observe}\ (\mathit{raise}\ e) \\
= & \quad \{\ \text{(O3)}\ \} \\
 & \mathit{fail}\ e
\end{array}
$$

$$
\begin{array}{ll}
 & (\mathit{promote}\ m)\ \underline{\kappa} \\
= & \quad \{\ \text{spec. and ass. (2)}\ \} \\
 & \mathit{observe}\ (\mathit{promote}\ m \mathbin{>\!\!>\!\!=} \kappa) \\
= & \quad \{\ \text{(O2)}\ \} \\
 & m \mathbin{>\!\!>\!\!=} \lambda a \to \mathit{observe}\ (\kappa\ a) \\
= & \quad \{\ \text{ass. (2)}\ \} \\
 & m \mathbin{>\!\!>\!\!=} \underline{\kappa}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{observe}\ m \\
= & \quad \{\ \text{(M2)}\ \} \\
 & \mathit{observe}\ (m \mathbin{>\!\!>\!\!=} \mathit{return}) \\
= & \quad \{\ \text{spec.}\ \} \\
 & m\ (\lambda a \to \mathit{observe}\ (\mathit{return}\ a)) \\
= & \quad \{\ \text{(O1)}\ \} \\
 & m\ \mathit{return}
\end{array}
$$

Fig. 3. Deriving a context-passing implementation of *RAISE*.

lations. Note that each monad law, the law for *raise*, and each law for *observe* is employed exactly once. In other words, the laws of the specification are necessary and sufficient for deriving an implementation.

It remains to determine the type of the new monad transformer. This is most easily accomplished by inspecting the definition of *promote*. Note that *promote* m equals $(\mathbin{>\!\!>\!\!=})\ m$ and recall that $(\mathbin{>\!\!>\!\!=})$ possesses the type $\forall a.\forall b.m\ a \to (a \to m\ b) \to m\ b$ which is equivalent to $\forall a.m\ a \to (\forall b.(a \to m\ b) \to m\ b)$. Consequently, the new transformer has type $\forall b.(a \to m\ b) \to m\ b$. So, while the term implementation requires existential quantification, the context-passing implementations makes use of universal quantification. The final implementation appears in Fig. 4.[1] The cognoscenti has certainly recognized that the implementation is identical with the definition of the *continuation monad transformer* (Liang *et al.*, 1995). Only the types are different: *RAISE* involves rank-2 types while the continuation monad transformer is additionally parameterized with the so-called answer type ($\mathit{CONT}\ \mathit{ans}\ m\ a = (a \to m\ \mathit{ans}) \to m\ \mathit{ans}$). The transformer *RAISE* m constitutes the smallest extension of m which allows to add *raise*. Note, for instance, that

[1] Note that *RAISE* must actually be defined using **newtype** instead of **type**. This, however, introduces an additional data constructor which affects the readability of the code. Instead we employ **type** declarations as if they worked as **newtype** declarations.

$$
\begin{array}{lll}
\textbf{type}\ RAISE\ m\ a & = & \forall b.(a \to m\ b) \to m\ b \\
\textbf{instance}\ (Monad\ m) \Rightarrow Monad\ (RAISE\ m)\ \textbf{where} & & \\
\quad return\ a & = & \lambda\underline{\kappa} \to \underline{\kappa}\ a \\
\quad m \gg= k & = & \lambda\underline{\kappa} \to m\ (\lambda a \to k\ a\ \underline{\kappa}) \\
\quad fail\ s & = & raise\ s \\
\textbf{instance}\ (Monad\ m) \Rightarrow Raise\ (RAISE\ m)\ \textbf{where} & & \\
\quad raise\ e & = & \lambda\underline{\kappa} \to fail\ e \\
\textbf{instance}\ Transformer\ RAISE\ \textbf{where} & & \\
\quad promote\ m & = & \lambda\underline{\kappa} \to m \gg= \underline{\kappa} \\
\quad observe\ m & = & m\ return
\end{array}
$$

Fig. 4. A context-passing implementation of *RAISE*.

callcc is definable in *CONT ans m* but not in *RAISE m*. We will see in Sec. 4.3 that rank-2 types have definite advantages over parameterized types.

4 Adding backtracking

By definition, a *backtracking monad* is a monad with two additional operations: the constant *false*, which denotes failure[2], and the binary operation $(\mid)$, which denotes nondeterministic choice. The class definition contains a third operation, termed *cons*, which provides a handy shortcut for $return\ a \mid m$.

$$
\begin{array}{lll}
\textbf{class}\ (Monad\ m) \Rightarrow Backtr\ m\ \textbf{where} & & \\
\quad false & :: & m\ a \\
\quad (\mid) & :: & m\ a \to m\ a \to m\ a \\
\quad cons & :: & a \to m\ a \to m\ a \\
\quad cons\ a\ m & = & return\ a \mid m \\
true & :: & (Monad\ m) \Rightarrow m\ () \\
true & = & return\ ()
\end{array}
$$

The operations are required to satisfy the following laws.

$$
\begin{array}{rcll}
false \mid m & = & m & \text{(B1)} \\
m \mid false & = & m & \text{(B2)} \\
(m \mid n) \mid o & = & m \mid (n \mid o) & \text{(B3)} \\
false \gg= k & = & false & \text{(B4)} \\
(m \mid n) \gg= k & = & (m \gg= k) \mid (n \gg= k) & \text{(B5)} \\
(promote\ m \gg= k) \mid m & = & promote\ m \gg= \lambda a \to k\ a \mid m & \text{(B6)}
\end{array}
$$

That is, *false* and $(\mid)$ form a monoid; *false* is a left zero of $(\gg=)$, and $(\gg=)$ distributes leftward through $(\mid)$. The most interesting law is probably the last one.

[2] The name *false* is not entirely satisfactory; *fail* would be more appropriate. Unfortunately, *fail* is already in use.

Since we aim at defining a backtracking monad transformer, we must also specify the interaction of promoted operations with $(\mid)$. Consider *promote m* as a deterministic computation, ie a computation which succeeds exactly once. Then (B6) formalizes our intuition that a deterministic computation can be pushed out of a disjunction's left branch. Finally, we must specify how the backtracking operations are observed in the base monad.

$$\begin{aligned} \mathit{observe}\ \mathit{false} &= \mathit{fail}\ \texttt{"false"} && \text{(O4)} \\ \mathit{observe}\ (\mathit{return}\ a \mid m) &= \mathit{return}\ a && \text{(O5)} \end{aligned}$$

So we can observe the first answer of a nondeterministic computation.

Remark. It may seem unreasonable that *false* is mapped onto *fail* `"false"`. However, we can always arrange that a nondeterministic computation succeeds at least once — wrap it up in $\mathit{map}\ \mathit{Just}\ \bullet \mid \mathit{return}\ \mathit{Nothing}$ — so this choice is not crucial. Furthermore, we will see that the particular definition of *observe* plays little rôle in the derivation of the monad transformer.

Remark. Note that axiom (B5) is the first *non-linear* equation. Applying it from left to right may possibly double the size of a computation. This implies, of course, that we can no longer expect that the running time of a computation is linear to its size. As a simple example, consider the program fragment

$$\begin{array}{l} \mathbf{do\ let}\ \mathit{choose} = \mathit{return}\ \mathit{False} \mid \mathit{return}\ \mathit{True} \\ \quad a_1 \leftarrow \mathit{choose} \\ \quad \ldots \\ \quad a_n \leftarrow \mathit{choose} \\ \quad \mathbf{if}\ \mathit{eval}\ e\ [a_1, \ldots, a_n] \\ \qquad \mathbf{then}\ \mathit{true} \\ \qquad \mathbf{else}\ \mathit{false}\ \ , \end{array}$$

which checks whether a Boolean formula with n variables is satisfiable.

4.1 A term implementation

The free term algebra of the backtracking monad is given by the following type definition.

$$\begin{array}{lcl} \mathbf{data}\ \mathit{BACKTR}\ m\ a & = & \mathit{Return}\ a \\ & \mid & (\mathit{BACKTR}\ m\ b) :\gg= (b \to \mathit{BACKTR}\ m\ a) \\ & \mid & \mathit{False} \\ & \mid & \mathit{BACKTR}\ m\ a :\mid \mathit{BACKTR}\ m\ a \\ & \mid & \mathit{Promote}\ (m\ a) \end{array}$$

Let's try to derive an interpreter for this language. The definition of the base cases follows immediately from the specification. For $m \mathbin{:\!\gg\!=} k$ we obtain

$$
\begin{aligned}
\mathit{observe}\ (\mathit{Return}\ a \mathbin{:\!\gg\!=} k) &= \mathit{observe}\ (k\ a)\\
\mathit{observe}\ ((m \mathbin{:\!\gg\!=} k_1) \mathbin{:\!\gg\!=} k_2) &= \mathit{observe}\ (m \mathbin{:\!\gg\!=} (\lambda a \to k_1\ a \mathbin{:\!\gg\!=} k_2))\\
\mathit{observe}\ (\mathit{False} \mathbin{:\!\gg\!=} k) &= \mathit{fail}\ \texttt{"false"}\\
\mathit{observe}\ ((m \mathbin{:\!|} n) \mathbin{:\!\gg\!=} k) &= \mathit{observe}\ ((m \mathbin{:\!\gg\!=} k) \mathbin{:\!|} (n \mathbin{:\!\gg\!=} k))\\
\mathit{observe}\ (\mathit{Promote}\ m \mathbin{:\!\gg\!=} k) &= m \gg\!= (\mathit{observe} \cdot k)\ .
\end{aligned}
$$

Similarly, for $m \mathbin{:\!|} n$ we make a case distinction on m.

$$
\begin{aligned}
\mathit{observe}\ (\mathit{Return}\ a \mathbin{:\!|} \mathrm{f}) &= \mathit{return}\ a\\
\mathit{observe}\ (m \mathbin{:\!\gg\!=} k \mathbin{:\!|} \mathrm{f}) &= \\
\mathit{observe}\ (\mathit{False} \mathbin{:\!|} \mathrm{f}) &= \mathit{observe}\ \mathrm{f}\\
\mathit{observe}\ ((m \mathbin{:\!|} n) \mathbin{:\!|} \mathrm{f}) &= \mathit{observe}\ (m \mathbin{:\!|} (n \mathbin{:\!|} \mathrm{f}))\\
\mathit{observe}\ (\mathit{Promote}\ m \mathbin{:\!|} \mathrm{f}) &= m
\end{aligned}
$$

Unfortunately, one case remains. There is no obvious way to simplify the term $\mathit{observe}\ (m \mathbin{:\!\gg\!=} k \mathbin{:\!|} \mathrm{f})$. As usual, we help ourselves by making a further case distinction on m.

$$
\begin{aligned}
\mathit{observe}\ ((\mathit{Return}\ a \mathbin{:\!\gg\!=} k) \mathbin{:\!|} \mathrm{f}) &= \mathit{observe}\ (k\ a \mathbin{:\!|} \mathrm{f})\\
\mathit{observe}\ (((m \mathbin{:\!\gg\!=} k_1) \mathbin{:\!\gg\!=} k_2) \mathbin{:\!|} \mathrm{f}) &= \mathit{observe}\ ((m \mathbin{:\!\gg\!=} \lambda a \to k_1\ a \mathbin{:\!\gg\!=} k_2) \mathbin{:\!|} \mathrm{f})\\
\mathit{observe}\ ((\mathit{False} \mathbin{:\!\gg\!=} k) \mathbin{:\!|} \mathrm{f}) &= \mathit{observe}\ \mathrm{f}\\
\mathit{observe}\ (((m \mathbin{:\!|} n) \mathbin{:\!\gg\!=} k) \mathbin{:\!|} \mathrm{f}) &= \mathit{observe}\ ((m \mathbin{:\!\gg\!=} k) \mathbin{:\!|} ((n \mathbin{:\!\gg\!=} k) \mathbin{:\!|} \mathrm{f}))\\
\mathit{observe}\ ((\mathit{Promote}\ m \mathbin{:\!\gg\!=} k) \mathbin{:\!|} \mathrm{f}) &= m \gg\!= \lambda a \to \mathit{observe}\ (k\ a \mathbin{:\!|} \mathrm{f})
\end{aligned}
$$

Voilá. We have succeeded in building an interpreter for backtracking. The complete implementation appears in Fig. 5. Now, what about correctness? Clearly, the case distinction is exhaustive. To establish termination we can use the following polynomial interpretation.

$$
\begin{aligned}
\mathit{Return}_{\tau}\ a &= 2\\
m \mathbin{:\!\gg}_{\tau}\ n &= m^2 \times n\\
\mathit{False}_{\tau} &= 2\\
m \mathbin{:\!|}_{\tau}\ n &= 2 \times m + n\\
\mathit{Promote}_{\tau}\ m &= 2
\end{aligned}
$$

As before, the laws of the specification only hold under observation.

4.2 A simplified term implementation

Let us take a brief look at the simplified term implementation. Inspecting the definition of *observe* — recall that a simplifier is likely to make the same case distinction as *observe* — we see that we need at most six terms: *False*, *Return* a, *Return* $a \mathbin{:\!|} \mathrm{f}$, *Promote* m, *Promote* $m \mathbin{:\!\gg\!=} k$, and *Promote* $m \mathbin{:\!|} \mathrm{f}$. We can eliminate three of them using $\mathit{return}\ a = \mathit{cons}\ a\ \mathit{false}$, $\mathit{promote}\ m = \mathit{promote}\ m \gg\!= \mathit{return}$, and $\mathit{promote}\ m \mathbin{:\!|} \mathrm{f} = \mathit{promote}\ m \gg\!= \lambda a \to \mathit{cons}\ a\ \mathrm{f}$. This explains the following

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{BACKTR}\ m\ a & = & \mathit{Return}\ a \\
 & \mid & \forall b.(\mathit{BACKTR}\ m\ b) :\!\!\gg\!\!= (b \to \mathit{BACKTR}\ m\ a) \\
 & \mid & \mathit{False} \\
 & \mid & \mathit{BACKTR}\ m\ a :\!\mid \mathit{BACKTR}\ m\ a \\
 & \mid & \mathit{Promote}\ (m\ a)
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Monad}\ (\mathit{BACKTR}\ m)\ \textbf{where}} \\
\quad \mathit{return} & = & \mathit{Return} \\
\quad (\gg\!=) & = & (:\!\!\gg\!\!=) \\
\quad \mathit{fail}\ s & = & \mathit{False} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Backtr}\ (\mathit{BACKTR}\ m)\ \textbf{where}} \\
\quad \mathit{false} & = & \mathit{False} \\
\quad (\mid) & = & (:\!\mid) \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Transformer}\ \mathit{BACKTR}\ \textbf{where}} \\
\quad \mathit{promote} & = & \mathit{Promote} \\
\quad \mathit{observe}\ (\mathit{Return}\ a) & = & \mathit{return}\ a \\
\quad \mathit{observe}\ (\mathit{Return}\ a :\!\!\gg\!\!= k) & = & \mathit{observe}\ (k\ a) \\
\quad \mathit{observe}\ ((m :\!\!\gg\!\!= k_1) :\!\!\gg\!\!= k_2) & = & \mathit{observe}\ (m :\!\!\gg\!\!= (\lambda a \to k_1\ a :\!\!\gg\!\!= k_2)) \\
\quad \mathit{observe}\ (\mathit{False} :\!\!\gg\!\!= k) & = & \mathit{fail}\ \texttt{"false"} \\
\quad \mathit{observe}\ ((m :\!\mid n) :\!\!\gg\!\!= k) & = & \mathit{observe}\ ((m :\!\!\gg\!\!= k) :\!\mid (n :\!\!\gg\!\!= k)) \\
\quad \mathit{observe}\ (\mathit{Promote}\ m :\!\!\gg\!\!= k) & = & m \gg\!= (\mathit{observe} \cdot k) \\
\quad \mathit{observe}\ \mathit{False} & = & \mathit{fail}\ \texttt{"false"} \\
\quad \mathit{observe}\ (\mathit{Return}\ a :\!\mid \mathrm{f}) & = & \mathit{return}\ a \\
\quad \mathit{observe}\ ((\mathit{Return}\ a :\!\!\gg\!\!= k) :\!\mid \mathrm{f}) & = & \mathit{observe}\ (k\ a :\!\mid \mathrm{f}) \\
\quad \mathit{observe}\ (((m :\!\!\gg\!\!= k_1) :\!\!\gg\!\!= k_2) :\!\mid \mathrm{f}) & = & \mathit{observe}\ ((m :\!\!\gg\!\!= (\lambda a \to k_1\ a :\!\!\gg\!\!= k_2)) :\!\mid \mathrm{f}) \\
\quad \mathit{observe}\ ((\mathit{False} :\!\!\gg\!\!= k) :\!\mid \mathrm{f}) & = & \mathit{observe}\ \mathrm{f} \\
\quad \mathit{observe}\ (((m :\!\mid n) :\!\!\gg\!\!= k) :\!\mid \mathrm{f}) & = & \mathit{observe}\ ((m :\!\!\gg\!\!= k) :\!\mid ((n :\!\!\gg\!\!= k) :\!\mid \mathrm{f})) \\
\quad \mathit{observe}\ ((\mathit{Promote}\ m :\!\!\gg\!\!= k) :\!\mid \mathrm{f}) & = & m \gg\!= \lambda a \to \mathit{observe}\ (k\ a :\!\mid \mathrm{f}) \\
\quad \mathit{observe}\ (\mathit{False} :\!\mid \mathrm{f}) & = & \mathit{observe}\ \mathrm{f} \\
\quad \mathit{observe}\ ((m :\!\mid n) :\!\mid \mathrm{f}) & = & \mathit{observe}\ (m :\!\mid (n :\!\mid \mathrm{f})) \\
\quad \mathit{observe}\ (\mathit{Promote}\ m :\!\mid \mathrm{f}) & = & m \\
\quad \mathit{observe}\ (\mathit{Promote}\ m) & = & m
\end{array}
$$

Fig. 5. A term implementation of *BACKTR*.

definition of simplified terms.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{BACKTR}\ m\ a & = & \mathit{False} \\
 & \mid & \mathit{Cons}\ a\ (\mathit{BACKTR}\ m\ a) \\
 & \mid & \mathit{PromoteBind}\ (m\ b)\ (b \to \mathit{BACKTR}\ m\ a)
\end{array}
$$

In essence, the simplified term algebra is an extension of the datatype of polymorphic lists with *False* corresponding to [] and *Cons* corresponding to (:). The additional constructor *PromoteBind* makes the difference between a monad and a monad transformer. The implementation of the operations is listed in Fig. 6. Note that *observe*'s definition is a transliteration of the laws (O3)–(O5). Since the term implementation is a direct descendent of the list monad, it inherits also its efficiency problems. The amortized running time of ($\mid$), for instance, is proportional

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{BACKTR}\ m\ a & = & \mathit{False} \\
 & | & \mathit{Cons}\ a\ (\mathit{BACKTR}\ m\ a) \\
 & | & \forall b.\mathit{PromoteBind}\ (m\ b)\ (b \rightarrow \mathit{BACKTR}\ m\ a) \\
\textbf{instance}\ \mathit{Monad}\ (\mathit{BACKTR}\ m)\ \textbf{where} & & \\
\quad \mathit{return}\ a & = & \mathit{Cons}\ a\ \mathit{False} \\
\quad \mathit{False} \ggg\!= k & = & \mathit{False} \\
\quad \mathit{Cons}\ a\ m \ggg\!= k & = & k\ a \mid (m \ggg\!= k) \\
\quad \mathit{PromoteBind}\ m\ k_1 \ggg\!= k_2 & = & \mathit{PromoteBind}\ m\ (\lambda a \rightarrow k_1\ a \ggg\!= k_2) \\
\quad \mathit{fail}\ s & = & \mathit{False} \\
\textbf{instance}\ \mathit{Backtr}\ (\mathit{BACKTR}\ m)\ \textbf{where} & & \\
\quad \mathit{false} & = & \mathit{False} \\
\quad \mathit{False} \mid n & = & n \\
\quad \mathit{Cons}\ a\ m \mid n & = & \mathit{Cons}\ a\ (m \mid n) \\
\quad \mathit{PromoteBind}\ m\ k \mid n & = & \mathit{PromoteBind}\ m\ (\lambda a \rightarrow k\ a \mid n) \\
\quad \mathit{cons} & = & \mathit{Cons} \\
\textbf{instance}\ \mathit{Transformer}\ \mathit{BACKTR}\ \textbf{where} & & \\
\quad \mathit{promote}\ m & = & \mathit{PromoteBind}\ m\ \mathit{return} \\
\quad \mathit{observe}\ \mathit{False} & = & \mathit{error}\ \texttt{"false"} \\
\quad \mathit{observe}\ (\mathit{Cons}\ a\ m) & = & \mathit{return}\ a \\
\quad \mathit{observe}\ (\mathit{PromoteBind}\ m\ k) & = & m \ggg\!= (\mathit{observe} \cdot k)
\end{array}
$$

Fig. 6. A simplified term implementation of *BACKTR*.

to the size of its first argument. Since *BACKTR* additionally manages promoted operations, the trouble is, of course, doubled.

4.3 A context-passing implementation

In Sec. 3.3 we have seen that the context-passing implementation essentially removes the interpretative layer from the 'naive' term implementation. If we apply the same steps to the new definition of *observe*, we can derive very systematically a context-passing implementation of backtracking. We leave the details to the reader and sketch only the main points. Firstly, from the case analysis which *observe* performs we may conclude that the most complex context has the form $\mathit{observe}\ (\bullet \ggg\!= \kappa \mid \mathrm{f})$. All other contexts can be rewritten into this form.

$$
\begin{array}{lcl}
\mathit{observe}\ \bullet & = & \mathit{observe}\ (\bullet \ggg\!= \mathit{return} \mid \mathit{false}) \\
\mathit{observe}\ (\bullet \ggg\!= \kappa) & = & \mathit{observe}\ (\bullet \ggg\!= \kappa \mid \mathit{false}) \\
\mathit{observe}\ ((\bullet \mid \mathrm{f}) \ggg\!= \kappa) & = & \mathit{observe}\ (\bullet \ggg\!= \kappa \mid \mathrm{f} \ggg\!= \kappa) \\
\mathit{observe}\ (\bullet \mid \mathrm{f}) & = & \mathit{observe}\ (\bullet \ggg\!= \mathit{return} \mid \mathrm{f}) \\
\mathit{observe}\ (\bullet \ggg\!= \kappa \mid \mathrm{f}) & & \\
\mathit{observe}\ ((\bullet \mid \mathrm{f}_1) \mid \mathrm{f}_2) & = & \mathit{observe}\ (\bullet \ggg\!= \mathit{return} \mid (\mathrm{f}_1 \mid \mathrm{f}_2))
\end{array}
$$

Secondly, if we inspect the equations which are concerned with $\mathit{observe}\ (\bullet \ggg\!= \kappa \mid \mathrm{f})$ we see that f appears once in the context $\mathit{observe}\ \bullet$. Likewise, κ is used twice in the context $observe\ (\bullet\ a :\!\mid \mathrm{f})$. These observations motivate the following specification.

$$
\begin{aligned}
& m\ \underline{\kappa}\ \underline{\mathrm{f}} = observe\ (m \gg\!= \kappa \mid \mathrm{f}) \\
&\quad \Longleftarrow \quad \underline{\mathrm{f}} = observe\ \mathrm{f} && (3)\\
&\quad \wedge \quad (\underline{\kappa}\ a\ \underline{\mathrm{f}'} = observe\ (\kappa\ a \mid \mathrm{f}') \ \Longleftarrow\ \underline{\mathrm{f}'} = observe\ \mathrm{f}') && (4)
\end{aligned}
$$

The nice thing about Hughes' technique is that mistakes made at this point will be discovered later when the operations are derived. For instance, it may seem unnecessary that $\underline{\kappa}$ is parameterized with $\underline{\mathrm{f}'}$. However, if we simply postulate $\underline{\kappa}\ a = observe\ (\kappa\ a \mid \mathrm{f})$, then we will not be able to derive a definition for $(\mid)$. Better still, one can develop the specification above while making the calculations. The derivation of *false*, for instance, motivates assumption (3); the derivation of *return* suggests $\underline{\kappa}\ a = observe\ (\kappa\ a \mid \mathrm{f})$ or assumption (4) and the derivation of $(\mid)$ confirms that (4) is the right choice. The complete derivation appears in Fig. 7. Interestingly, each equation of the specification is employed exactly once.

It remains to determine the type of the backtracking monad transformer. If we assume that the second parameter, the so-called *failure continuation*, has type $m\ b$, then the first parameter, the so-called *success continuation*, is of type $a \to m\ b \to m\ b$. It follows that the type of the new transformer is $\forall a.(a \to m\ b \to m\ b) \to m\ b \to m\ b$. Again, the answer type is universally quantified. We will see shortly why this is a reasonable choice. Fig. 8 summarizes the implementation.

Reconsider Fig. 7 and note that the derivation of *return*, $(\gg\!=)$, *false*, and $(\mid)$ is completely independent of *observe*'s specification. The laws (O4) and (O5) are only required in the derivation of *observe*. Only *promote* relies on (O3) which, however, appears to be the only sensible way to observe promoted operations. This suggests that we can define different observations without changing the definitions of the other operations. In other words, we may generalize the specification as follows (φ is an arbitrary observer function).

$$
\begin{aligned}
& m\ \underline{\kappa}\ \underline{\mathrm{f}} = \varphi\ (m \gg\!= \kappa \mid \mathrm{f}) && (5)\\
&\quad \Longleftarrow \quad \underline{\mathrm{f}} = \varphi\ \mathrm{f} \\
&\quad \wedge \quad (\underline{\kappa}\ a\ \underline{\mathrm{f}'} = \varphi\ (\kappa\ a \mid \mathrm{f}') \ \Longleftarrow\ \underline{\mathrm{f}'} = \varphi\ \mathrm{f}') \\
&\quad \wedge \quad \varphi\ (promote\ m \gg\!= k) = m \gg\!= (\varphi \cdot k)
\end{aligned}
$$

To illustrate the use of the generalized specification assume that we want to collect all solutions of a nondeterministic computation. To this end we specify an observation *solve* of type $(Monad\ m) \Rightarrow BACKTR\ m\ a \to m\ [a]$:

$$
\begin{aligned}
solve\ false &= return\ [\,] && \text{(S1)}\\
solve\ (return\ a \mid m) &= a \lhd solve\ m && \text{(S2)}\\
solve\ (promote\ m \gg\!= k) &= m \gg\!= (solve \cdot k)\ , && \text{(S3)}
\end{aligned}
$$

where $(\lhd)$ is given by

$$
\begin{aligned}
(\lhd) \quad &:: \quad (Monad\ m) \Rightarrow a \to m\ [a] \to m\ [a]\\
a \lhd ms \quad &= \quad ms \gg\!= \lambda as \to return\ (a : as)\ .
\end{aligned}
$$

specification

$$
\begin{aligned}
& m\ \underline{\kappa}\ \underline{\mathrm{f}} = \mathit{observe}\ (m \gg\!= \kappa \mid \mathrm{f}) \\
\Longleftarrow\quad & \underline{\mathrm{f}} = \mathit{observe}\ \mathrm{f} && (3) \\
\wedge\quad & (\underline{\kappa}\ a\ \underline{\mathrm{f}'} = \mathit{observe}\ (\kappa\ a \mid \mathrm{f}') \ \Longleftarrow\ \underline{\mathrm{f}'} = \mathit{observe}\ \mathrm{f}') && (4)
\end{aligned}
$$

$$
\begin{aligned}
& (\mathit{return}\ a)\ \underline{\kappa}\ \underline{\mathrm{f}} \\
= & \quad \{\ \text{spec. and ass. (3) \& (4)}\ \} \\
& \mathit{obs}\ (\mathit{return}\ a \gg\!= \kappa \mid \mathrm{f}) \\
= & \quad \{\ \text{(M1)}\ \} \\
& \mathit{obs}\ (\kappa\ a \mid \mathrm{f}) \\
= & \quad \{\ \text{ass. (3) \& (4)}\ \} \\
& \underline{\kappa}\ a\ \underline{\mathrm{f}}
\end{aligned}
$$

$$
\begin{aligned}
& (m \gg\!= k)\ \underline{\kappa}\ \underline{\mathrm{f}} \\
= & \quad \{\ \text{spec. and ass. (3) \& (4)}\ \} \\
& \mathit{obs}\ ((m \gg\!= k) \gg\!= \kappa \mid \mathrm{f}) \\
= & \quad \{\ \text{(M3)}\ \} \\
& \mathit{obs}\ (m \gg\!= (\lambda a \to k\ a \gg\!= \kappa) \mid \mathrm{f}) \\
= & \quad \{\ \text{spec. and ass. (3)}\ \} \\
& m\ (\lambda a\ \underline{\mathrm{f}'} \to \mathit{obs}\ (k\ a \gg\!= \kappa \mid \mathrm{f}'))\ \underline{\mathrm{f}} \\
= & \quad \{\ \text{spec. and ass. (4)}\ \} \\
& m\ (\lambda a\ \underline{\mathrm{f}'} \to k\ a\ \underline{\kappa}\ \underline{\mathrm{f}'})\ \underline{\mathrm{f}}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{false}\ \underline{\kappa}\ \underline{\mathrm{f}} \\
= & \quad \{\ \text{spec. and ass. (3) \& (4)}\ \} \\
& \mathit{obs}\ (\mathit{false} \gg\!= \kappa \mid \mathrm{f}) \\
= & \quad \{\ \text{(B4)}\ \} \\
& \mathit{obs}\ (\mathit{false} \mid \mathrm{f}) \\
= & \quad \{\ \text{(B1)}\ \} \\
& \mathit{obs}\ \mathrm{f} \\
= & \quad \{\ \text{spec. and ass. (3)}\ \} \\
& \underline{\mathrm{f}}
\end{aligned}
$$

$$
\begin{aligned}
& (m \mid n)\ \underline{\kappa}\ \underline{\mathrm{f}} \\
= & \quad \{\ \text{spec. and ass. (3) \& (4)}\ \} \\
& \mathit{obs}\ ((m \mid n) \gg\!= \kappa \mid \mathrm{f}) \\
= & \quad \{\ \text{(B5)}\ \} \\
& \mathit{obs}\ ((m \gg\!= \kappa \mid n \gg\!= \kappa) \mid \mathrm{f}) \\
= & \quad \{\ \text{(B3)}\ \} \\
& \mathit{obs}\ (m \gg\!= \kappa \mid (n \gg\!= \kappa \mid \mathrm{f})) \\
= & \quad \{\ \text{spec. and ass. (4)}\ \} \\
& m\ \underline{\kappa}\ (\mathit{obs}\ (n \gg\!= \kappa \mid \mathrm{f})) \\
= & \quad \{\ \text{spec. and ass. (3) \& (4)}\ \} \\
& m\ \underline{\kappa}\ (n\ \underline{\kappa}\ \underline{\mathrm{f}})
\end{aligned}
$$

$$
\begin{aligned}
& (\mathit{pro}\ m)\ \underline{\kappa}\ \underline{\mathrm{f}} \\
= & \quad \{\ \text{spec. and ass. (3) \& (4)}\ \} \\
& \mathit{obs}\ (\mathit{pro}\ m \gg\!= \kappa \mid \mathrm{f}) \\
= & \quad \{\ \text{(B6)}\ \} \\
& \mathit{obs}\ (\mathit{pro}\ m \gg\!= (\lambda a \to \kappa\ a \mid \mathrm{f})) \\
= & \quad \{\ \text{(O3)}\ \} \\
& m \gg\!= \lambda a \to \mathit{obs}\ (\kappa\ a \mid \mathrm{f}) \\
= & \quad \{\ \text{ass. (3) \& (4)}\ \} \\
& m \gg\!= \lambda a \to \underline{\kappa}\ a\ \underline{\mathrm{f}}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{obs}\ m \\
= & \quad \{\ \text{(M2) and (B2)}\ \} \\
& \mathit{obs}\ (m \gg\!= \mathit{return} \mid \mathit{false}) \\
= & \quad \{\ \text{spec.}\ \} \\
& m\ (\lambda a\ \underline{\mathrm{f}'} \to \mathit{obs}\ (\mathit{return}\ a \mid \mathrm{f}'))\ (\mathit{obs}\ \mathit{false}) \\
= & \quad \{\ \text{(O4)}\ \} \\
& m\ (\lambda a\ \underline{\mathrm{f}'} \to \mathit{obs}\ (\mathit{return}\ a \mid \mathrm{f}'))\ (\mathit{fail}\ \ldots) \\
= & \quad \{\ \text{(O5)}\ \} \\
& m\ (\lambda a\ \underline{\mathrm{f}'} \to \mathit{return}\ a)\ (\mathit{fail}\ \ldots)
\end{aligned}
$$

Fig. 7. Deriving a context-passing implementation of *BACKTR*.

An implementation for *solve* can be readily derived if we specialize (5) for $\kappa = \mathit{return}$ and $\mathrm{f} = \mathit{false}$. We obtain

$$
\begin{aligned}
& \varphi\ m = m\ (\oplus)\ e && (6) \\
\Longleftrightarrow\quad & \varphi\ \mathit{false} = e \\
\wedge\quad & \varphi\ (\mathit{return}\ a \mid m) = a \oplus \varphi\ m \\
\wedge\quad & \varphi\ (\mathit{promote}\ m \gg\!= k) = m \gg\!= (\varphi \cdot k)\ .
\end{aligned}
$$

Note that the 'only if' part is a simple consequence of the implementation equations.

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{BACKTR}\ m\ a & = & \forall b.(a \to m\ b \to m\ b) \to m\ b \to m\ b \\
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Monad}\ (\mathit{BACKTR}\ m)\ \textbf{where}} \\
\quad \mathit{return}\ a & = & \lambda \underline{\kappa} \to \underline{\kappa}\ a \\
\quad m \ggg k & = & \lambda \underline{\kappa} \to m\ (\lambda a \to k\ a\ \underline{\kappa}) \\
\quad \mathit{fail}\ s & = & \mathit{false} \\
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Monad}\ m) \Rightarrow \mathit{Backtr}\ (\mathit{BACKTR}\ m)\ \textbf{where}} \\
\quad \mathit{false} & = & \lambda \underline{\kappa} \to \mathit{id} \\
\quad m \mid n & = & \lambda \underline{\kappa} \to m\ \underline{\kappa} \cdot n\ \underline{\kappa} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Transformer}\ \mathit{BACKTR}\ \textbf{where}} \\
\quad \mathit{promote}\ m & = & \lambda \underline{\kappa}\ \underline{\mathrm{f}} \to m \ggg \lambda a \to \underline{\kappa}\ a\ \underline{\mathrm{f}} \\
\quad \mathit{observe}\ m & = & m\ (\lambda a\ \underline{\mathrm{f}} \to \mathit{return}\ a)\ (\mathit{fail}\ \texttt{"false"})
\end{array}
$$

Fig. 8. A context-passing implementation of *BACKTR*.

Instead of providing *solve* as an additional observer function we promote it into the backtracking monad.

$$
\begin{array}{lcl}
\mathit{sols} & :: & (\mathit{Monad}\ m) \Rightarrow \mathit{BACKTR}\ m\ a \to \mathit{BACKTR}\ m\ [a] \\
\mathit{sols}\ m & = & \mathit{promote}\ (m\ (\lhd)\ (\mathit{return}\ [\,]))
\end{array}
$$

This way we can use the all solution collecting function as if it were a new computational primitive. Since *promote* is a monad morphism, we furthermore know that *sols* satisfies suitable variants of (S1)–(S3). Note that the implementation of *sols* makes non-trivial use of rank-2 types. If we used a variant of *BACKTR* which is parameterized with the answer type, then *sols* cannot be assigned a type $t\ a \to t\ [a]$ for some t.

The context-passing implementation appears to differ significantly from the simplified term implementation or from the list monad. However, both implementations are derived from a common ancestor, the 'naive' term implementation. Hence, we should not be surprised if common structures rise to the surface. For instance, relation (6) closely resembles the *universal property* of the fold-operation on lists.

$$
\begin{array}{l}
\varphi\ \mathit{as} = \mathit{foldr}\ (\oplus)\ e\ \mathit{as} \\
\quad \Longleftrightarrow \quad \varphi\ [\,] = e \quad \wedge \quad \varphi\ (a : \mathit{as}) = a \oplus \varphi\ \mathit{as}
\end{array}
$$

The universal property states that $\mathit{foldr}\ (\oplus)\ e$ is the unique solution of the two equations specifying φ. Likewise, property (6) formalizes that $\lambda m \to m\ (\oplus)\ e$ is the unique solution of the corresponding three equations. This suggests that every list processing function which can be expressed as a fold is easily adapted to the context-passing implementation — provided the typing restrictions are met. Recall that the success and the failure continuation have types $a \to m\ b \to m\ b$ and $m\ b$. By contrast, *foldr* takes arguments of types $a \to b \to b$ and b. Again, this makes the difference between a monad and a monad transformer. The difference vanishes if we specialize *BACKTR* to the identity monad.

5 Adding control

Let's extend our language by two additional control constructs which have a definite Prolog flavour. The first, called cut and denoted $!$, allows to reduce the search space by dynamically pruning unwanted computation paths. The second, termed *call*, is provided for controlling the effect of cut. Both constructs are introduced as a subclass of *Backtr*.

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Backtr}\ m) \Rightarrow \mathit{Cut}\ m\ \textbf{where} \\
\quad \begin{array}{lcl}
! & :: & m\ () \\
\mathit{cutfalse} & :: & m\ a \\
\mathit{call} & :: & m\ a \to m\ a \\
! & = & \mathit{true} \mid \mathit{cutfalse} \\
\mathit{cutfalse} & = & ! \gg \mathit{false}
\end{array}
\end{array}
$$

The operational reading of $!$ and *call* is as follows. The cut succeeds exactly once and returns (). As a side-effect it discards *all* previous alternatives. The operation *call* delimits the effect of cut: $\mathit{call}\ m$ executes m; if the cut is invoked in m, it discards only the choices made since m was called. The class definition contains a third operation, called *cutfalse*, which captures a common programming idiom in Prolog, the so-called cut-fail combination (Sterling & Shapiro, 1986).

Note that instances of *Cut* are free to leave either $!$ or *cutfalse* unspecified since both operations are interdefinable. The default definitions already employ our knowledge about the properties of the operations which we shall consider next. Note that we sketch the axiomatization only briefly, for a more in-depth treatment the interested reader is referred to (Hinze, 1999). The cut is characterized by the following three equations.

$$
\begin{array}{rcll}
(! \gg m) \mid n & = & ! \gg m & (!1) \\
! \gg (m \mid n) & = & m \mid ! \gg n & (!2) \\
! \gg \mathit{true} & = & ! & (!3)
\end{array}
$$

The first equation formalizes our intuition that a cut discards *past* choice points, ie alternatives which appear 'above' or to its left. On the other hand, the cut does not affect *future* choice points, ie alternatives which appear to its right. This fact is captured by (!2). Axiom (!3) simply records that cut returns (). An immediate consequence of the axioms is $! = \mathit{true} \mid ! \gg \mathit{false}$ which explains the default definition of cut. To see why this relation holds replace m by *true* and n by *false* in (!2).

The operation *cutfalse* enjoys algebraic properties which are somewhat easier to remember: *cutfalse* is a left zero of both $(\gg=)$ and $(\mid)$.

$$
\begin{array}{rcll}
\mathit{cutfalse} \gg= k & = & \mathit{cutfalse} & (\mathrm{CF1}) \\
\mathit{cutfalse} \mid m & = & \mathit{cutfalse} & (\mathrm{CF2})
\end{array}
$$

We have remarked that $!$ and *cutfalse* are interdefinable. Likewise, the two sets of axioms are interchangeable. That is, we may either set $\mathit{cutfalse} = ! \gg \mathit{false}$ and take the equations for $!$ as axioms — the laws concerning *cutfalse* are then simple logical consequences — or vice versa.

Finally, *call* is required to satisfy the following properties.

$$\begin{aligned} \mathit{call}\ \mathit{false} &= \mathit{false} && \text{(C1)}\\ \mathit{call}\ (\mathit{return}\ a \mid m) &= \mathit{return}\ a \mid \mathit{call}\ m && \text{(C2)}\\ \mathit{call}\ (! \gg m) &= \mathit{call}\ m && \text{(C3)}\\ \mathit{call}\ (m \mid \mathit{cutfalse}) &= \mathit{call}\ m && \text{(C4)}\\ \mathit{call}\ (\mathit{promote}\ m \gg= k) &= \mathit{promote}\ m \gg= (\mathit{call} \cdot k) && \text{(C5)} \end{aligned}$$

Thus, $\mathit{call}\ m$ behaves essentially like m except that any cut inside m has only local effect. It remains to lay down how the new operations are observed in the underlying monad.

$$\mathit{observe}\ (\mathit{call}\ m) = \mathit{observe}\ m \qquad \text{(O6)}$$

Note that we need not specify the observation of $!$ and *cutfalse* since (C3), (C4), and (O6) imply $\mathit{observe}\ (! \gg m) = \mathit{observe}\ m$ and $\mathit{observe}\ (m \mid \mathit{cutfalse}) = \mathit{observe}\ m$.

5.1 A term implementation

The free term implementation faces two problems, one technical and one fundamental. Let us consider the technical problem first. Inspecting the type signature of cut, we find that cut cannot be turned into a constructor, because it does not have the right type. If we define a type, say, $CUT\ m\ a$, then $!$ must have exactly this type. Alas, its type signature only allows for a substitution instance, ie $CUT\ m\ ()$. Here, we stumble over the general problem that Haskell's **data** construct is not capable of expressing arbitrary polymorphic term algebras. To the best of the author's knowledge only the type system described in (Hanus, 1991) allows for type-specialized constructors. Fortunately, the axioms save the day. Since $!$ can be expressed in terms of *cutfalse* and this operation has a polymorphic type, we turn *cutfalse* into a constructor.

$$\begin{aligned} \mathbf{data}\ CUT\ m\ a \;=\;& \mathit{Return}\ a\\ \mid\;& (CUT\ m\ b) :\gg= (b \to CUT\ m\ a)\\ \mid\;& \mathit{False}\\ \mid\;& \mathit{CutFalse}\\ \mid\;& CUT\ m\ a :\mid CUT\ m\ a\\ \mid\;& \mathit{Call}\ (CUT\ m\ a)\\ \mid\;& \mathit{Promote}\ (m\ a) \end{aligned}$$

Turning to the definition of *observe* we encounter a problem of a more fundamental nature. For a start, we discover that the term $\mathit{observe}\ (\mathit{call}\ m \gg= k)$ cannot be simplified. If we make a further case distinction on m, we end up with $\mathit{observe}\ (\mathit{call}\ (\mathit{call}\ m \gg= k_1) \gg= k_2)$ which is not reducible either. The crux is that we have no axiom which specifies the interaction of *call* with $(\gg=)$. And rightly so. Each *call* opens a new scope for cut. Hence, we cannot reasonably expect that nested *call*s can be collapsed. This suggests to define two interpreters, one for *observe* and one for *call*, which means, of course, that the implementation is no longer based on the free term algebra. The resulting code, which is mostly straightforward, appears in Fig. 9. The equations involving *cutfalse* use the fact that *cutfalse* is a left zero of both $(\ggg\!=)$ and $(\shortmid)$, and that *call* maps *cutfalse* to *false*. Note that *observe* falls back on *call* to avoid duplication of code.

5.2 A simplified term implementation

For completeness, here is the simplified term algebra which augments the type *BACKTR* of Sec. 4.2 with an additional constructor for *cutfalse*.

$$\begin{array}{lcl}\mathbf{data}\ \mathit{CUT}\ m\ a & = & \mathit{False} \\ & \mid & \mathit{CutFalse} \\ & \mid & \mathit{Cons}\ a\ (\mathit{CUT}\ m\ a) \\ & \mid & \mathit{PromoteBind}\ (m\ b)\ (b \to \mathit{CUT}\ m\ a)\end{array}$$

In essence, we have lists with two different terminators, *False* and *CutFalse*. Interestingly, exactly this structure (without *PromoteBind*) has been used to give a denotational semantics for Prolog with cut (Billaud, 1990), where *cutfalse* and *call* are termed *esc* and *unesc*. Fig. 10 contains the implementation of the simplified term algebra.

5.3 A context-passing implementation

We have seen that the realization of cut and *call* is more demanding since there is no way to simplify nested invocations of *call*. With regard to the context-passing implementation this means that we must consider an infinite number of contexts. Using a grammar-like notation we can characterize the set of all possible contexts as follows.

$$\mathcal{C} \quad ::= \quad \mathit{observe}\ (\bullet \ggg\!= k \shortmid \mathrm{f}) \mid \mathcal{C}\ [\mathit{call}\ (\bullet \ggg\!= k \shortmid \mathrm{f})]$$

A context is either simple or of the form $C\ [\mathit{call}\ (\bullet \ggg\!= k \shortmid \mathrm{f})]$ where C is the enclosing context. Thus, contexts are organized in a list- or stack-like fashion. As usual we will represent operations as functions from contexts to observations. The main difference to Sec. 4.3 is that each operation must now consider two different contexts and that the contexts are recursively defined. Note, however, the duality between the term and the context-passing implementation: In Sec. 5.1 we had two interpreters, *call* and *observe*, and each interpreter had to consider each operation. Here we have two contexts and each operation must consider each context.

Turning to the implementation details we will see that the greatest difficulty is to get the types right. The contexts are represented by a recursive datatype with two constructors: *OBCC* (which is an acronym for <u>o</u>bserve-<u>b</u>ind-<u>c</u>hoice <u>c</u>ontext) and *CBCC* (<u>c</u>all-<u>b</u>ind-<u>c</u>hoice <u>c</u>ontext). The first takes two arguments, the success and the failure continuation, while the second expects three arguments, the two continuations and the representation of the enclosing context. To infer their types

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{CUT}\ m\ a & = & \mathit{Return}\ a \\
 & \mid & \forall b.(\mathit{CUT}\ m\ b) :\gg= (b \to \mathit{CUT}\ m\ a) \\
 & \mid & \mathit{False} \\
 & \mid & \mathit{CutFalse} \\
 & \mid & \mathit{CUT}\ m\ a :\mid \mathit{CUT}\ m\ a \\
 & \mid & \mathit{Promote}\ (m\ a) \\
\textbf{instance}\ \mathit{Monad}\ (\mathit{CUT}\ m)\ \textbf{where} & & \\
\quad \mathit{return} & = & \mathit{Return} \\
\quad (\gg=) & = & (:\gg=) \\
\quad \mathit{fail}\ s & = & \mathit{False} \\
\textbf{instance}\ \mathit{Backtr}\ (\mathit{CUT}\ m)\ \textbf{where} & & \\
\quad \mathit{false} & = & \mathit{False} \\
\quad (\mid) & = & (:\mid) \\
\textbf{instance}\ \mathit{Cut}\ (\mathit{CUT}\ m)\ \textbf{where} & & \\
\quad \mathit{cutfalse} & = & \mathit{CutFalse} \\
\quad \mathit{call}\ (\mathit{Return}\ a) & = & \mathit{Return}\ a \\
\quad \mathit{call}\ (\mathit{Return}\ a :\gg= \kappa) & = & \mathit{call}\ (\kappa\ a) \\
\quad \mathit{call}\ ((m :\gg= \kappa_1) :\gg= \kappa_2) & = & \mathit{call}\ (m :\gg= (\lambda a \to \kappa_1\ a :\gg= \kappa_2)) \\
\quad \mathit{call}\ (\mathit{False} :\gg= \kappa) & = & \mathit{False} \\
\quad \mathit{call}\ (\mathit{CutFalse} :\gg= \kappa) & = & \mathit{False} \\
\quad \mathit{call}\ ((m :\mid n) :\gg= \kappa) & = & \mathit{call}\ ((m :\gg= \kappa) :\mid (n :\gg= \kappa)) \\
\quad \mathit{call}\ (\mathit{Promote}\ m :\gg= \kappa) & = & \mathit{Promote}\ m :\gg= (\mathit{call} \cdot \kappa) \\
\quad \mathit{call}\ \mathit{False} & = & \mathit{False} \\
\quad \mathit{call}\ \mathit{CutFalse} & = & \mathit{False} \\
\quad \mathit{call}\ (\mathit{Return}\ a :\mid \mathrm{f}) & = & \mathit{Return}\ a :\mid \mathit{call}\ \mathrm{f} \\
\quad \mathit{call}\ ((\mathit{Return}\ a :\gg= \kappa) :\mid \mathrm{f}) & = & \mathit{call}\ (\kappa\ a :\mid \mathrm{f}) \\
\quad \mathit{call}\ (((m :\gg= \kappa_1) :\gg= \kappa_2) :\mid \mathrm{f}) & = & \mathit{call}\ ((m :\gg= (\lambda a \to \kappa_1\ a :\gg= \kappa_2)) :\mid \mathrm{f}) \\
\quad \mathit{call}\ ((\mathit{False} :\gg= \kappa) :\mid \mathrm{f}) & = & \mathit{call}\ \mathrm{f} \\
\quad \mathit{call}\ ((\mathit{CutFalse} :\gg= \kappa) :\mid \mathrm{f}) & = & \mathit{False} \\
\quad \mathit{call}\ (((m :\mid n) :\gg= \kappa) :\mid \mathrm{f}) & = & \mathit{call}\ ((m :\gg= \kappa) :\mid ((n :\gg= \kappa) :\mid \mathrm{f})) \\
\quad \mathit{call}\ ((\mathit{Promote}\ m :\gg= \kappa) :\mid \mathrm{f}) & = & \mathit{Promote}\ m :\gg= \lambda a \to \mathit{call}\ (\kappa\ a :\mid \mathrm{f}) \\
\quad \mathit{call}\ (\mathit{False} :\mid \mathrm{f}) & = & \mathit{call}\ \mathrm{f} \\
\quad \mathit{call}\ (\mathit{CutFalse} :\mid \mathrm{f}) & = & \mathit{False} \\
\quad \mathit{call}\ ((m :\mid n) :\mid \mathrm{f}) & = & \mathit{call}\ (m :\mid (n :\mid \mathrm{f})) \\
\quad \mathit{call}\ (\mathit{Promote}\ m :\mid \mathrm{f}) & = & \mathit{Promote}\ m :\mid \mathit{call}\ \mathrm{f} \\
\quad \mathit{call}\ (\mathit{Promote}\ m) & = & \mathit{Promote}\ m \\
\textbf{instance}\ \mathit{Transformer}\ \mathit{CUT}\ \textbf{where} & & \\
\quad \mathit{promote} & = & \mathit{Promote} \\
\quad \mathit{observe}\ m & = & \mathit{observe}'\ (\mathit{call}\ m) \\
\mathit{observe}' & :: & (\mathit{Monad}\ m) \Rightarrow \mathit{CUT}\ m\ a \to m\ a \\
\mathit{observe}'\ (\mathit{Return}\ a) & = & \mathit{return}\ a \\
\mathit{observe}'\ (\mathit{Promote}\ m :\gg= \kappa) & = & m \gg= (\mathit{observe}' \cdot \kappa) \\
\mathit{observe}'\ \mathit{False} & = & \mathit{fail}\ \texttt{"false"} \\
\mathit{observe}'\ (\mathit{Return}\ a :\mid \mathrm{f}) & = & \mathit{return}\ a \\
\mathit{observe}'\ (\mathit{Promote}\ m :\mid \mathrm{f}) & = & m \\
\mathit{observe}'\ (\mathit{Promote}\ m) & = & m
\end{array}
$$

Fig. 9. A term implementation of *CUT*.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{CUT}\ m\ a & = & \mathit{False} \\
 & \mid & \mathit{CutFalse} \\
 & \mid & \mathit{Cons}\ a\ (\mathit{CUT}\ m\ a) \\
 & \mid & \forall b.\mathit{PromoteBind}\ (m\ b)\ (b \to \mathit{CUT}\ m\ a) \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Monad}\ (\mathit{CUT}\ m)\ \mathbf{where}} \\
\quad \mathit{return}\ a & = & \mathit{Cons}\ a\ \mathit{False} \\
\quad \mathit{False} \gg\!= k & = & \mathit{False} \\
\quad \mathit{CutFalse} \gg\!= k & = & \mathit{CutFalse} \\
\quad \mathit{Cons}\ a\ m \gg\!= k & = & k\ a \mid (m \gg\!= k) \\
\quad \mathit{PromoteBind}\ m\ \kappa \gg\!= k & = & \mathit{PromoteBind}\ m\ (\lambda a \to \kappa\ a \gg\!= k) \\
\quad \mathit{fail}\ s & = & \mathit{False} \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Backtr}\ (\mathit{CUT}\ m)\ \mathbf{where}} \\
\quad \mathit{false} & = & \mathit{False} \\
\quad \mathit{False} \mid n & = & n \\
\quad \mathit{CutFalse} \mid n & = & \mathit{CutFalse} \\
\quad \mathit{Cons}\ a\ m \mid n & = & \mathit{Cons}\ a\ (m \mid n) \\
\quad \mathit{PromoteBind}\ m\ \kappa \mid n & = & \mathit{PromoteBind}\ m\ (\lambda a \to \kappa\ a \mid n) \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Cut}\ (\mathit{CUT}\ m)\ \mathbf{where}} \\
\quad ! & = & \mathit{Cons}\ ()\ \mathit{CutFalse} \\
\quad \mathit{cutfalse} & = & \mathit{CutFalse} \\
\quad \mathit{call}\ \mathit{False} & = & \mathit{False} \\
\quad \mathit{call}\ \mathit{CutFalse} & = & \mathit{False} \\
\quad \mathit{call}\ (\mathit{Cons}\ a\ m) & = & \mathit{Cons}\ a\ (\mathit{call}\ m) \\
\quad \mathit{call}\ (\mathit{PromoteBind}\ m\ \kappa) & = & \mathit{PromoteBind}\ m\ (\mathit{call} \cdot \kappa) \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Transformer}\ \mathit{CUT}\ \mathbf{where}} \\
\quad \mathit{promote}\ m & = & \mathit{PromoteBind}\ m\ \mathit{return} \\
\quad \mathit{observe}\ \mathit{False} & = & \mathit{fail}\ \texttt{"false"} \\
\quad \mathit{observe}\ \mathit{CutFalse} & = & \mathit{fail}\ \texttt{"false"} \\
\quad \mathit{observe}\ (\mathit{Cons}\ a\ m) & = & \mathit{return}\ a \\
\quad \mathit{observe}\ (\mathit{PromoteBind}\ m\ \kappa) & = & m \gg\!= (\mathit{observe} \cdot \kappa)
\end{array}
$$

Fig. 10. A simplified term implementation of CUT.

it is useful to consider the specification of the context-passing implementation beforehand. The specification is similar to the one given in Sec. 4.3 except that we have two clauses, one for each context.

$$
\begin{array}{lll}
\multicolumn{2}{l}{m\ (\mathit{OBCC}\ \underline{\kappa}\ \underline{\mathrm{f}}) = \mathit{observe}\ (m \gg\!= \kappa \mid \mathrm{f})} & \\
\quad \Longleftarrow & \underline{\mathrm{f}} = \mathit{observe}\ \mathrm{f} & (4) \\
\quad \wedge & (\underline{\kappa}\ a\ \underline{\mathrm{f}'} = \mathit{observe}\ (\kappa\ a \mid \mathrm{f}') \ \Longleftarrow\ \underline{\mathrm{f}'} = \mathit{observe}\ \mathrm{f}') & (5) \\
\multicolumn{2}{l}{m \cdot \mathit{CBCC}\ \underline{\kappa}\ \underline{\mathrm{f}} = \mathit{call}\ (m \gg\!= \kappa \mid \mathrm{f})} & \\
\quad \Longleftarrow & \underline{\mathrm{f}} = \mathit{call}\ \mathrm{f} & (6) \\
\quad \wedge & (\underline{\kappa}\ a\ \underline{\mathrm{f}'} = \mathit{call}\ (\kappa\ a \mid \mathrm{f}') \ \Longleftarrow\ \underline{\mathrm{f}'} = \mathit{call}\ \mathrm{f}') & (7)
\end{array}
$$

The first clause closely corresponds to the specification of Sec. 4.3. For that reason we may assign the components of $\mathit{OBCC}\ \underline{\kappa}\ \underline{\mathrm{f}}$ the same types: $\underline{\mathrm{f}}$ has type $m\ b$ and

$\underline{\kappa}$ has type $a \to m\ b \to m\ b$ where b is the answer type. This implies that the type of contexts must be parameterized with m, a, and b.

$$\textbf{data}\ \mathcal{C}\ m\ a\ b \quad = \quad OBCC\ (a \to m\ b \to m\ b)\ (m\ b) \mid \ldots$$

The second clause of the specification has essentially the same structure as the first one. The main difference is that the components dwell in the transformed monad rather than in the underlying monad. Furthermore, $CBCC$ additionally contains the enclosing context which may have a different type. To illustrate, consider the context $C\ [\mathit{call}\ (\bullet \ggg\!= \kappa \shortmid \mathrm{f})]$ of type $\mathcal{C}\ m\ a\ b$. If we assume that the enclosing context C has type $\mathcal{C}\ m\ i\ b$ — there is no reason to require that C has the same argument type as the entire context, but it must have the same answer type — then f has type $CUT\ m\ i$ and κ has type $a \to CUT\ m\ i \to CUT\ m\ i$. This motivates the following definition.

$$\begin{array}{lcl}
\textbf{data}\ \mathcal{C}\ m\ a\ b & = & OBCC\ (a \to m\ b \to m\ b)\ (m\ b) \\
 & \mid & CBCC\ (a \to CUT\ m\ i \to CUT\ m\ i)\ (CUT\ m\ i)\ (\mathcal{C}\ m\ i\ b) \\
\textbf{type}\ CUT\ m\ a & = & \forall b. \mathcal{C}\ m\ a\ b \to m\ b
\end{array}$$

Note that the intermediate type is represented by an existentially quantified variable. The mutually recursive types $\mathcal{C}$ and CUT are quite interesting as they involve both universal and existential quantification, a combination of features the author has not seen before.

Now that we have the types right, we can address the derivation of the various operations. Except for *promote* the calculations are analogous to those of Sec. 4.3. For *promote* m we must conduct an inductive proof to show that m propagates through the stack of contexts, ie $(\mathit{promote}\ m \ggg\!= k)\ c = m \ggg\!= \lambda a \to k\ a\ c$. The proof is left as an exercise to the reader. To derive cut we reason:

$$\begin{array}{cl}
 & ! \cdot CBCC\ \underline{\kappa}\ \underline{\mathrm{f}} \\
= & \quad \{\ \text{specification and assumptions (6) \& (7)}\ \} \\
 & \mathit{call}\ (!\ \ggg\!= \kappa \shortmid \mathrm{f}) \\
= & \quad \{\ \text{(!3), (M3), and (M1)}\ \} \\
 & \mathit{call}\ (!\ \gg \kappa\ () \shortmid \mathrm{f}) \\
= & \quad \{\ \text{(!1) and (!2)}\ \} \\
 & \mathit{call}\ (\kappa\ () \shortmid\ !\ \gg \mathit{false}) \\
= & \quad \{\ \text{assumption (7)}\ \} \\
 & \underline{\kappa}\ ()\ (\mathit{call}\ (!\ \gg \mathit{false})) \\
= & \quad \{\ \text{(C3) and (C1)}\ \} \\
 & \underline{\kappa}\ ()\ \mathit{false}\ \ .
\end{array}$$

The derivation for the context $OBCC$ proceeds completely analogous. For *call* we obtain:

$$\begin{array}{cl}
 & \mathit{call}\ m \\
= & \quad \{\ \text{(M2) and (B2)}\ \}
\end{array}$$

$$
\begin{array}{lll}
\textbf{data}\ \mathcal{C}\ m\ a\ b & = & OBCC\ (a \to m\ b \to m\ b)\ (m\ b) \\
& \mid & \forall i. CBCC\ (a \to CUT\ m\ i \to CUT\ m\ i)\ (CUT\ m\ i)\ (\mathcal{C}\ m\ i\ b) \\
\textbf{type}\ CUT\ m\ a & = & \forall b. \mathcal{C}\ m\ a\ b \to m\ b \\
\multicolumn{3}{l}{\textbf{instance}\ (Monad\ m) \Rightarrow Monad\ (CUT\ m)\ \textbf{where}} \\
\quad return\ a & = & \lambda c_0 \to \textbf{case}\ c_0\ \textbf{of} \\
& & \qquad OBCC\ \underline{\kappa}\ \underline{\mathrm{f}} \to \underline{\kappa}\ a\ \underline{\mathrm{f}} \\
& & \qquad CBCC\ \underline{\kappa}\ \underline{\mathrm{f}}\ c \to \underline{\kappa}\ a\ \underline{\mathrm{f}}\ c \\
\quad m \gg\!= k & = & \lambda c_0 \to \textbf{case}\ c_0\ \textbf{of} \\
& & \qquad OBCC\ \underline{\kappa}\ \underline{\mathrm{f}} \to m\ (OBCC\ (\lambda a\ \underline{\mathrm{f}'} \to k\ a\ (OBCC\ \underline{\kappa}\ \underline{\mathrm{f}'}))\ \underline{\mathrm{f}}) \\
& & \qquad CBCC\ \underline{\kappa}\ \underline{\mathrm{f}}\ c \to m\ (CBCC\ (\lambda a\ \underline{\mathrm{f}'} \to k\ a \cdot CBCC\ \underline{\kappa}\ \underline{\mathrm{f}'})\ \underline{\mathrm{f}}\ c) \\
\quad fail\ s & = & false \\
\multicolumn{3}{l}{\textbf{instance}\ (Monad\ m) \Rightarrow Backtr\ (CUT\ m)\ \textbf{where}} \\
\quad false & = & \lambda c_0 \to \textbf{case}\ c_0\ \textbf{of} \\
& & \qquad OBCC\ \underline{\kappa}\ \underline{\mathrm{f}} \to \underline{\mathrm{f}} \\
& & \qquad CBCC\ \underline{\kappa}\ \underline{\mathrm{f}}\ c \to \underline{\mathrm{f}}\ c \\
\quad m \mid n & = & \lambda c_0 \to \textbf{case}\ c_0\ \textbf{of} \\
& & \qquad OBCC\ \underline{\kappa}\ \underline{\mathrm{f}} \to m\ (OBCC\ \underline{\kappa}\ (n\ (OBCC\ \underline{\kappa}\ \underline{\mathrm{f}}))) \\
& & \qquad CBCC\ \underline{\kappa}\ \underline{\mathrm{f}}\ c \to m\ (CBCC\ \underline{\kappa}\ (n \cdot CBCC\ \underline{\kappa}\ \underline{\mathrm{f}})\ c) \\
\multicolumn{3}{l}{\textbf{instance}\ (Monad\ m) \Rightarrow Cut\ (CUT\ m)\ \textbf{where}} \\
\quad ! & = & \lambda c_0 \to \textbf{case}\ c_0\ \textbf{of} \\
& & \qquad OBCC\ \underline{\kappa}\ \underline{\mathrm{f}} \to \underline{\kappa}\ ()\ (fail\ \texttt{"false"}) \\
& & \qquad CBCC\ \underline{\kappa}\ \underline{\mathrm{f}}\ c \to \underline{\kappa}\ ()\ false\ c \\
\quad call\ m & = & \lambda c_0 \to m\ (CBCC\ cons\ false\ c_0) \\
\multicolumn{3}{l}{\textbf{instance}\ Transformer\ CUT\ \textbf{where}} \\
\quad promote\ m & = & \lambda c_0 \to \textbf{case}\ c_0\ \textbf{of} \\
& & \qquad OBCC\ \underline{\kappa}\ \underline{\mathrm{f}} \to m \gg\!= \lambda a \to \underline{\kappa}\ a\ \underline{\mathrm{f}} \\
& & \qquad CBCC\ \underline{\kappa}\ \underline{\mathrm{f}}\ c \to m \gg\!= \lambda a \to \underline{\kappa}\ a\ \underline{\mathrm{f}}\ c \\
\quad observe\ m & = & m\ (OBCC\ (\lambda a\ \underline{\mathrm{f}} \to return\ a)\ (fail\ \texttt{"false"}))
\end{array}
$$

Fig. 11. A context-passing implementation of CUT.

$$
\begin{array}{ll}
& call\ (m \gg\!= return \mid false) \\
= & \quad \{\ \text{specification}\ \} \\
& m \cdot CBCC\ (\lambda a\ \underline{\mathrm{f}'} \to call\ (return\ a \mid \mathrm{f}'))\ (call\ false) \\
= & \quad \{\ \text{(C1) and (C2)}\ \} \\
& m \cdot CBCC\ (\lambda a\ \underline{\mathrm{f}'} \to return\ a \mid call\ \mathrm{f}')\ false \\
= & \quad \{\ \underline{\mathrm{f}'} = call\ \mathrm{f}'\ \} \\
& m \cdot CBCC\ (\lambda a\ \underline{\mathrm{f}'} \to return\ a \mid \underline{\mathrm{f}'})\ false \\
= & \quad \{\ \text{definition}\ cons\ \} \\
& m \cdot CBCC\ cons\ false\ .
\end{array}
$$

Thus, *call* installs a new context with *cons* and *fail* as the initial failure continuations. The complete implementation appears in Fig. 11. The definition of an all solution collecting function analogous to that of Sec. 4.3 is left as an exercise to the reader.

6 Conclusion

Naturally, most of the credit goes to J. Hughes for introducing two wonderful techniques for deriving programs from their specification. Many of the calculations given in this paper already appear in (Hughes, 1995), albeit specialized to monads. However, the step from monads to monad transformers is not a big one and this is one of the pleasant findings. To be able to derive an implementation of Prolog's control core from a given axiomatization is quite remarkable. We have furthermore applied the techniques to derive state monad transformers, $STATE$, and exception monad transformers, EXC. In both cases the techniques worked well.

Some work remains to be done though. We did not address the problem of promotion in general. It is well known that different combinations of transformers generally lead to different semantics of the operations involved. For instance, composing $STATE$ with $BACKTR$ yields a backtracking monad with a backtrackable state, which is characterized as follows.

$$\begin{aligned} \mathit{store}\ s \gg \mathit{false} &= \mathit{false} \\ \mathit{store}\ s \gg (m \mid n) &= \mathit{store}\ s \gg m \mid \mathit{store}\ s \gg n \end{aligned}$$

Reversing the order of the two transformers results in a global state, which enjoys a different axiomatization.

$$\mathit{store}\ s \gg (m \mid n) = \mathit{store}\ s \gg m \mid n$$

For both variants it is straightforward to derive an implementation from the corresponding specification — in the first case $(\mid)$ is promoted through $STATE$, in the second case *store* is promoted through $BACKTR$. Unfortunately, some harder cases remain, where the author has not been able to *derive* a promotion in a satisfying way. The problematic operations are, in general, those where the interaction with $(\gg=)$ is not explicitly specified (as in the case of *call*).

References

Billaud, Michel. (1990). Simple operational and denotational semantics for Prolog with cut. *Theoretical computer science*, **71**(2), 193–208.

Bird, Richard. (1998). *Introduction to functional programming using Haskell*. 2nd edn. London: Prentice Hall Europe.

Dershowitz, Nachum, & Jouannaud, Jean-Pierre. (1990). Rewrite systems. *Chap. 6, pages 243–320 of:* van Leeuwen, Jan (ed), *Handbook of theoretical computer science, volume b: Formal models and semantics*. Elsevier Science Publishers B.V. (North Holland).

Hanus, Michael. (1991). Horn clause programs with polymorphic types: semantics and resolution. *Theoretical computer science*, **89**(1), 63–106.

Hinze, Ralf. (1999). *Prolog's control constructs in a functional setting — Axioms and implementation*. Submitted for publication.

Hughes, John. (1995). The design of a pretty-printing library. *Pages 53–93 of:* Jeuring, J., & Meijer, E. (eds), *Advanced functional programming*. Lecture Notes in Computer Science, vol. 925. Springer Verlag.

Läufer, Konstantin, & Odersky, Martin. 1992 (June). An extension of ML with first-class abstract types. *Pages 78–91 of: Proceedings of the 1992 ACM workshop on ML and its applications, san francisco*. Association for Computing Machinery.

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

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

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

Peyton Jones, Simon. (1998). *Explicit quantification in Haskell*. ⟨URL: http://research.microsoft.com/ Users/ simonpj/ Haskell/ quantification.html⟩.

Peyton Jones [editor], Simon, Hughes [editor], John, Augustsson, Lennart, Barton, Dave, Boutel, Brian, Burton, Warren, Fraser, Simon, Fasel, Joseph, Hammond, Kevin, Hinze, Ralf, Hudak, Paul, Johnsson, Thomas, Jones, Mark, Launchbury, John, Meijer, Erik, Peterson, John, Reid, Alastair, Runciman, Colin, & Wadler, Philip. 1998 (December). *Haskell 98 — A non-strict, purely functional language*.

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

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

Wadler, Philip. 1992 (January). The essence of functional programming. *Pages 1–14 of: Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Sante Fe, New Mexico*.

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