DERIVING BACKTRACKING MONAD TRANSFORMERS
Functional Pearl

RALF HINZE

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

September, 2000
(Pick the slides at `.../~ralf/talks.html#T16`.)

Combinator libraries

If you want to design and implement a combinator library, I recommend reading

> John Hughes. *The design of a pretty-printing library.* In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 53–96. Springer-Verlag, 1995.

beforehand.

Hughes shows how to derive an implementation from a specification, where a specification consists of a *signature* and *properties* that the operations of the signature must satisfy.

In the paper I apply Hughes' technique to derive a backtracking monad transformer that adds backtracking to an arbitrary monad (additionally, the prolog cut and an operation for delimiting the effect of a cut are supported).

For the talk I chose to simplify matters slightly: we restrict ourselves to computations of type $M\ ()$, that is, we ignore the results of computations.

In other words, we neither consider monads nor monad transformers. Fortunately, the generalization to monads and monad transformers is fairly straightforward.

Backtracking: signature

$$\begin{array}{|l|}\hline \mathbf{data}\ M \\ \mathit{false}, \mathit{true} :: M \\ (\vee), (\wedge) \quad :: M \to M \to M \\ \hline\end{array}$$

It is important to have a function from the new type to an old type.

$$\boxed{\mathit{observe} :: M \to \mathit{Int}}$$

The observer function $\mathit{observe}$ counts the number of successes.

Backtracking: properties

$$\begin{aligned}
\mathit{false} \vee x &= x = x \vee \mathit{false} \\
(x \vee y) \vee z &= x \vee (y \vee z) \\
\mathit{false} \wedge x &= \mathit{false} \\
\mathit{true} \wedge x &= x = x \wedge \mathit{true} \\
(x \wedge y) \wedge z &= x \wedge (y \wedge z) \\
(x \vee y) \wedge z &= (x \wedge z) \vee (y \wedge z)
\end{aligned}$$

$$\begin{aligned}
\mathit{observe}\ \mathit{false} &= 0 \\
\mathit{observe}\ (\mathit{true} \vee x) &= \mathit{succ}\ (\mathit{observe}\ x)
\end{aligned}$$

A term implementation, the easy part

$$\mathbf{data}\ \mathit{Backtr} = \mathit{False} \mid \mathit{True} \mid \mathit{Backtr} :\!\vee\ \mathit{Backtr} \mid \mathit{Backtr} :\!\wedge\ \mathit{Backtr}$$

Each of the operations false, true, $(\vee)$, and $(\wedge)$ is simply implemented by the corresponding constructor.

$$\begin{aligned}
&\mathit{false} = \mathit{False} \\
&\mathit{true}\ = \mathit{True} \\
&(\vee)\ \ = (:\!\vee) \\
&(\wedge)\ \ = (:\!\wedge)
\end{aligned}$$

In other words, the operations do nothing.

A term implementation, the hard part

All the work is performed by $\mathit{observe}$, which can be seen as an interpreter.

$$\begin{aligned}
&\mathit{observe}\ \mathit{False} = 0 \\
&\mathit{observe}\ \mathit{True}\ = 1
\end{aligned}$$

It is tempting to set $\mathit{observe}\ (m :\!\vee\ n) = \mathit{observe}\ m + \mathit{observe}\ n$, but this equation does not follow immediately from the laws. Instead, we proceed by making a further case distinction on m.

$$\begin{aligned}
\mathit{observe}\ (\mathit{False} :\!\vee\ \mathrm{f}) &= \mathit{observe}\ \mathrm{f} \\
\mathit{observe}\ (\mathit{True} :\!\vee\ \mathrm{f}) &= \mathit{succ}\ (\mathit{observe}\ \mathrm{f}) \\
\mathit{observe}\ ((m :\!\vee\ n) :\!\vee\ \mathrm{f}) &= \mathit{observe}\ (m :\!\vee\ (n :\!\vee\ \mathrm{f}))
\end{aligned}$$

Now we are stuck. How do we simplify $\mathit{observe}\ ((m :\!\wedge\ n) :\!\vee\ \mathrm{f})$?

Again, we help ourselves by making a further case distinction on m.

$$\begin{aligned}
\mathit{observe}\ ((\mathit{False} :\!\wedge\ k) :\!\vee\ \mathrm{f}) &= \mathit{observe}\ \mathrm{f} \\
\mathit{observe}\ ((\mathit{True} :\!\wedge\ k) :\!\vee\ \mathrm{f}) &= \mathit{observe}\ (k :\!\vee\ \mathrm{f}) \\
\mathit{observe}\ (((m :\!\vee\ n) :\!\wedge\ k) :\!\vee\ \mathrm{f}) &= \mathit{observe}\ ((m :\!\wedge\ k) :\!\vee\ ((n :\!\wedge\ k) :\!\vee\ \mathrm{f})) \\
\mathit{observe}\ (((m :\!\wedge\ n) :\!\wedge\ k) :\!\vee\ \mathrm{f}) &= \mathit{observe}\ ((m :\!\wedge\ (n :\!\wedge\ k)) :\!\vee\ \mathrm{f})
\end{aligned}$$

On case remains: $\mathit{observe}\ (m :\!\wedge\ n)$. The corresponding equations can be easily determined by partially evaluating $\mathit{observe}\ ((m :\!\wedge\ n) :\!\vee\ \mathit{False})$.

Correctness

The definition of $observe$ is exhaustive.

It is furthermore terminating.

However, since we are working in the free term algebra, the laws only hold *under observation*.

$$
\begin{aligned}
observe\ (false \vee x) = observe\ x &= observe\ (x \vee false)\\
observe\ ((x \vee y) \vee z) &= observe\ (x \vee (y \vee z))\\
observe\ (false \wedge x) &= observe\ false\\
&\ldots
\end{aligned}
$$

A context-passing implementation, step 1

Now, let us try to improve the efficiency of the term implementation.

For a start, we can avoid some pattern matching if we specialize $observe$ for $e :\!\vee\ \mathrm{f}$, $(e :\!\wedge\ c) :\!\vee\ \mathrm{f}$, and $e :\!\wedge\ c$. We specify

$$
\begin{aligned}
observed\ e\ \mathrm{f} &= observe\ (e :\!\vee\ \mathrm{f})\\
observecd\ e\ c\ \mathrm{f} &= observe\ ((e :\!\wedge\ c) :\!\vee\ \mathrm{f})\\
observec\ e\ c &= observe\ (e :\!\wedge\ c).
\end{aligned}
$$

Given this specification we can easily derive the following implementations of $observed$ and $observecd$ ($observec$ is a special version of $observecd$).

$$
\begin{aligned}
&observed\ False &&= \lambda \mathrm{f} \to observe\ \mathrm{f}\\
&observed\ True &&= \lambda \mathrm{f} \to succ\ ((observe\ \mathrm{f}))\\
&observed\ (m :\!\vee\ n) &&= \lambda \mathrm{f} \to observed\ m\ (n :\!\vee\ \mathrm{f})\\
&observed\ (m :\!\wedge\ n) &&= \lambda \mathrm{f} \to observecd\ m\ n\ \mathrm{f}
\end{aligned}
$$

$$
\begin{aligned}
&observecd\ False &&= \lambda c\ \mathrm{f} \to observe\ \mathrm{f}\\
&observecd\ True &&= \lambda c\ \mathrm{f} \to observed\ c\ \mathrm{f}\\
&observecd\ (m :\!\vee\ n) &&= \lambda c\ \mathrm{f} \to observecd\ m\ c\ ((n :\!\wedge\ c) :\!\vee\ \mathrm{f})\\
&observecd\ (m :\!\wedge\ n) &&= \lambda c\ \mathrm{f} \to observecd\ m\ (n :\!\wedge\ c)\ \mathrm{f}
\end{aligned}
$$

Step 2

Now, note that the parameter f is eventually passed to $observe$ in each case. Likewise, c is eventually passed to $observed$.

These observations suggest that we could try to advance the function calls and pass $observe\ \mathrm{f}$ instead of f and similarly $observed\ c$ instead of c.

$$
\begin{aligned}
\underline{observed}\ e\ \underline{\mathrm{f}} = observe\ (e :\!\vee\ \mathrm{f}) &\iff \underline{\mathrm{f}} = observe\ \mathrm{f}\\
\underline{observecd}\ e\ \underline{c}\ \underline{\mathrm{f}} = observe\ (e :\!\wedge\ c :\!\vee\ \mathrm{f}) &\iff \underline{\mathrm{f}} = observe\ \mathrm{f}\\
&\qquad \wedge\ \ \underline{c} = \underline{observed}\ c
\end{aligned}
$$

Note that we actually pass $\underline{observed}\ c$ rather than $observed\ c$ since we want to avoid dependencies on the 'old' code.

Now, let us derive $\underline{observecd}$.

Assume that $\underline{\mathrm{f}} = observe\ \mathrm{f}$ and $\underline{c} = \underline{observed}\ c$.

Case $e = False$:

$$
\begin{array}{ll}
 & \underline{observecd}\ False\ \underline{c}\ \underline{\mathrm{f}} \\
= & \quad \{ \text{ assumptions and specification } \} \\
 & observe\ (False :\wedge c :\vee \mathrm{f}) \\
= & \quad \{ \ false \wedge x = false \text{ and } false \vee x = x \ \} \\
 & observe\ \mathrm{f} \\
= & \quad \{ \ \underline{\mathrm{f}} = observe\ \mathrm{f} \ \} \\
 & \underline{\mathrm{f}}
\end{array}
$$

Case $e = True$:

$$
\begin{array}{ll}
 & \underline{observecd}\ True\ \underline{c}\ \underline{\mathrm{f}} \\
= & \quad \{ \text{ assumptions and specification } \} \\
 & observe\ (True :\wedge c :\vee \mathrm{f}) \\
= & \quad \{ \ true \wedge x = x \ \} \\
 & observe\ (c :\vee \mathrm{f}) \\
= & \quad \{ \ \underline{\mathrm{f}} = observe\ \mathrm{f} \text{ and specification } \} \\
 & \underline{observed}\ c\ \underline{\mathrm{f}} \\
= & \quad \{ \ \underline{c} = \underline{observed}\ c \ \} \\
 & \underline{c}\ \underline{\mathrm{f}}.
\end{array}
$$

$$
\begin{array}{ll}
 & \underline{observecd}\ (m :\vee n)\ \underline{c}\ \underline{\mathrm{f}} \\
= & \quad \{ \text{ assumptions and specification } \} \\
 & observe\ ((m :\vee n) :\wedge c :\vee \mathrm{f}) \\
= & \quad \{ \ (x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z) \ \} \\
 & observe\ ((m :\wedge c :\vee n :\wedge c) :\vee \mathrm{f}) \\
= & \quad \{ \ (x \vee y) \vee z = x \vee (y \vee z) \ \} \\
 & observe\ ((m :\wedge c) :\vee ((n :\wedge c) :\vee \mathrm{f})) \\
= & \quad \{ \ \underline{c} = \underline{observed}\ c \text{ and specification } \} \\
 & \underline{observecd}\ m\ \underline{c}\ (observe\ ((n :\wedge c) :\vee \mathrm{f})) \\
= & \quad \{ \ \underline{\mathrm{f}} = observe\ \mathrm{f},\ \underline{c} = \underline{observed}\ c, \text{ and specification } \} \\
 & \underline{observecd}\ m\ \underline{c}\ (\underline{observecd}\ n\ \underline{c}\ \underline{\mathrm{f}}).
\end{array}
$$

$$
\begin{array}{ll}
 & \underline{observecd}\ (m :\wedge n)\ \underline{c}\ \underline{\mathrm{f}} \\
= & \quad \{ \text{ assumptions and specification } \} \\
 & observe\ ((m :\wedge n) :\wedge c :\vee \mathrm{f}) \\
= & \quad \{ \ (x \wedge y) \wedge z = x \wedge (y \wedge z) \ \} \\
 & observe\ (m :\wedge (n :\wedge c) :\vee \mathrm{f}) \\
= & \quad \{ \ \underline{\mathrm{f}} = observe\ \mathrm{f} \text{ and specification } \} \\
 & \underline{observecd}\ m\ (\underline{observed}\ (n :\wedge c))\ \underline{\mathrm{f}} \\
= & \quad \{ \text{ definition } \underline{observed} \ \} \\
 & \underline{observecd}\ m\ (\underline{observecd}\ n\ (\underline{observed}\ c))\ \underline{\mathrm{f}} \\
= & \quad \{ \ \underline{c} = \underline{observed}\ c \ \} \\
 & \underline{observecd}\ m\ (\underline{observecd}\ n\ \underline{c})\ \underline{\mathrm{f}}.
\end{array}
$$

A final generalization step yields:

$$
\begin{array}{ll}
\underline{observecd}\ \mathit{False} & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow \underline{\mathrm{f}} \\
\underline{observecd}\ \mathit{True} & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow \underline{c}\ \underline{\mathrm{f}} \\
\underline{observecd}\ (m :\vee\ n) & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow \underline{observecd}\ m\ \underline{c}\ (\underline{observecd}\ n\ \underline{c}\ \underline{\mathrm{f}}) \\
\underline{observecd}\ (m :\wedge\ n) & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow \underline{observecd}\ m\ (\underline{observecd}\ n\ \underline{c})\ \underline{\mathrm{f}}
\end{array}
$$

NB. $\underline{observecd}$ is closed; it does not depend on any other function.

Step 3: the final code

As a last step we remove the interpretative layer, which is straightforward since the constructors only appear on the left-hand sides of the equations.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{type}\ \mathit{Backtr} = (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow \mathit{Int} \rightarrow \mathit{Int}} \\
\mathit{false} & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow \underline{\mathrm{f}} \\
\mathit{true} & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow \underline{c}\ \underline{\mathrm{f}} \\
m \vee n & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow m\ \underline{c}\ (n\ \underline{c}\ \underline{\mathrm{f}}) \\
m \wedge n & = \lambda \underline{c}\ \underline{\mathrm{f}} \rightarrow m\ (n\ \underline{c})\ \underline{\mathrm{f}} \\
\mathit{observe}\ m & = m\ \mathit{succ}\ 0
\end{array}
$$

NB. We have reinvented the Church numerals!

To see why the term ‘context-passing’ is appropriate, consider the final specification of the context-passing implementation.

$$
\begin{array}{l}
op\ \underline{c}\ \underline{\mathrm{f}} = \mathit{observe}\ (op \wedge c \vee \mathrm{f}) \\
\quad \Longleftarrow \quad \underline{\mathrm{f}} = \mathit{observe}\ \mathrm{f} \\
\quad \wedge \quad \underline{c}\ \underline{\mathrm{f}'} = \mathit{observe}\ (c \vee \mathrm{f}') \Longleftarrow \underline{\mathrm{f}'} = \mathit{observe}\ \mathrm{f}'
\end{array}
$$

The arguments $\underline{c}$ and $\underline{\mathrm{f}}$ can be seen as a representation of op's calling context $\mathit{observe}\ (\bullet \wedge k \vee \mathrm{f})$.

In this case we only have one calling context so the implementation specializes to continuation-passing style ($\underline{c} \cong$ success continuation and $\underline{\mathrm{f}} \cong$ failure continuation).

Adding control: signature

$$
\begin{array}{ll}
!\,, \mathit{cutfalse} & :: M \\
\mathit{call} & :: M \rightarrow M
\end{array}
$$

The cut succeeds exactly once and discards *all* previous alternatives. The operation call delimits the effect of cut. The operation $\mathit{cutfalse}$ captures a common programming idiom in Prolog, the so-called cut-fail combination.

Adding control: properties

$$! = true \vee cutfalse$$
$$cutfalse = \, ! \wedge false$$

$$cutfalse \wedge x = cutfalse$$
$$cutfalse \vee x = cutfalse$$

$$\begin{aligned} call\ false &= false \\ call\ cutfalse &= false \\ call\ (true \vee m) &= true \vee call\ m \\ observe\ (call\ m) &= observe\ m \end{aligned}$$

A term implementation, sketch

Problem: we cannot simplify $observe\ (call\ m \wedge k)$. There is no axiom that specifies the interaction of $call$ with $(\wedge)$.

Solution: define two interpreters, one for $observe$ and one for $call$. This means, of course, that the implementation is no longer based on the free term algebra.

$$\begin{array}{ll} \mathbf{data}\ Cut & = False \\ & \mid\ CutFalse \\ & \mid\ True \\ & \mid\ Cut :\wedge Cut \\ & \mid\ Cut :\vee Cut \\ \ldots & \\ call\ ((False :\wedge k) :\vee s) & = call\ s \\ call\ ((CutFalse :\wedge k) :\vee s) & = False \\ call\ ((True :\wedge k) :\vee s) & = call\ (k :\vee s) \\ call\ (((m :\vee n) :\wedge k) :\vee s) & = call\ ((m :\wedge k) :\vee ((n :\wedge k) :\vee s)) \\ call\ (((m :\wedge n) :\wedge k) :\vee s) & = call\ ((m :\wedge (n :\wedge k)) :\vee s) \\ \ldots & \end{array}$$

A context-passing implementation, sketch

The context-passing implementation is *dual* to the term implementation: before we had two interpreters, $call$ and $observe$, and each interpreter had to consider each operation; now we have two contexts and each operation must consider each context.

$$Context \ ::= \ observe\ (\bullet \wedge k \vee \mathrm{f}) \mid Context[call\ (\bullet \wedge k \vee \mathrm{f})]$$

We implement the operations as functions from contexts to observations.

$$\begin{array}{ll} \mathbf{data}\ Context & = OC\ (Int \to Int)\ Int \\ & \mid\ CC\ (Cut \to Cut)\ Cut\ Context \\ \mathbf{type}\ Cut & = Context \to Int \end{array}$$

$$
\begin{array}{lll}
\mathit{false} & = \lambda ctx_0 \rightarrow & \mathbf{case}\ ctx_0\ \mathbf{of} \\
 & & OC\ \underline{c}\ \underline{f} \rightarrow \underline{f} \\
 & & CC\ \underline{c}\ \underline{f}\ ctx \rightarrow \underline{f}\ ctx \\
\mathit{true} & = \lambda ctx_0 \rightarrow & \mathbf{case}\ ctx_0\ \mathbf{of} \\
 & & OC\ \underline{c}\ \underline{f} \rightarrow \underline{c}\ \underline{f} \\
 & & CC\ \underline{c}\ \underline{f}\ ctx \rightarrow \underline{c}\ \underline{f}\ ctx \\
m \vee n & = \lambda ctx_0 \rightarrow & \mathbf{case}\ ctx_0\ \mathbf{of} \\
 & & OC\ \underline{c}\ \underline{f} \rightarrow m\ (OC\ \underline{c}\ (n\ (OC\ \underline{c}\ \underline{f}))) \\
 & & CC\ \underline{c}\ \underline{f}\ ctx \rightarrow m\ (CC\ \underline{c}\ (n \cdot CC\ \underline{c}\ \underline{f})\ ctx) \\
m \wedge k & = \lambda ctx_0 \rightarrow & \mathbf{case}\ ctx_0\ \mathbf{of} \\
 & & OC\ \underline{c}\ \underline{f} \rightarrow m\ (OC\ (\lambda \underline{f'} \rightarrow k\ (OC\ \underline{c}\ \underline{f'}))\ \underline{f}) \\
 & & CC\ \underline{c}\ \underline{f}\ ctx \rightarrow m\ (CC\ (\lambda \underline{f'} \rightarrow k \cdot CC\ \underline{c}\ \underline{f'})\ \underline{f}\ ctx)
\end{array}
$$

$$
\begin{array}{lll}
! & = \lambda ctx_0 \rightarrow & \mathbf{case}\ ctx_0\ \mathbf{of} \\
 & & OC\ \underline{c}\ \underline{f} \rightarrow \underline{c}\ 0 \\
 & & CC\ \underline{c}\ \underline{f}\ ctx \rightarrow \underline{c}\ \mathit{false}\ ctx \\
\mathit{cutfalse} & = \lambda ctx_0 \rightarrow & \mathbf{case}\ ctx_0\ \mathbf{of} \\
 & & OC\ \underline{c}\ \underline{f} \rightarrow 0 \\
 & & CC\ \underline{c}\ \underline{f}\ ctx \rightarrow \mathit{false}\ ctx \\
\mathit{call}\ m & = \lambda ctx_0 \rightarrow & m\ (CC\ (\lambda m \rightarrow \mathit{true} \vee m)\ \mathit{false}\ ctx_0) \\
\mathit{observe}\ m & = m\ (OC\ \mathit{succ}\ 0) &
\end{array}
$$

Summary

The *term implementation* represents operations by terms and works by defining a mapping from operations to observations — this mapping can be seen as defining a simple interpreter.

The *context-passing implementation* represents operations as functions from their calling context to observations. The context-passing implementation can be systematically derived from the term implementation.