1 Delimited continuations in Haskell

This section describes programming with delimited control in Haskell. Delimited
control, like its instance, exceptions, is an effect. Therefore, we have to use
monads. We will be using the monad Cont from a monad transformer library.
Although the Cont monad is familiar to many Haskell programmers, we will
nevertheless introduce it below. As well shall see, Cont is the monad for the
delimited control.

1.1 Introduction to the Cont monad programming

For the introduction we use one of the earlier OchaCaml examples

```
reset (3 + shift (λk → 5*2)) − 1
```

and re-write it in Haskell in the monadic style. The re-writing is systematic,
even mechanical:

```
t1 = liftM2 (−)
   (reset
    (liftM2 (+) (return 3)
     (shift (λk → liftM2 (*) (return 5) (return 2))))))
   (return 1)
```

The code does look like scheme, doesn’t it? The code mentions several identifiers
yet to be introduced: reset, shift, liftM2, and return. While one may guess the
meaning of reset and shift from the first part of the tutorial, liftM2 and return
may look obscure to some. They are defined in Haskell standard libraries. Their
types will help us understand their meaning. We will ask the Haskell interpreter
GHCi to show the type of 1 and the type of return 1:

```
*> : type 1
1 :: Num a ⇒ a
*> : type (return 1)
(return 1) :: (Num a, Monad m) ⇒ m a
```

Whereas 1 is a number, return 1 is a computation (in some monad m) that
produces the number, and may also do something else like printing – or, in our
case, throwing exceptions and performing other so-called control effects. Types
indeed can tell us a lot about expressions. After all, a type is an approximation
of expression’s behavior, outlining the behavior of an expression without running
it.

Likewise, comparing the type of the ordinary subtraction (−) with the type
of liftM2 (−) may give us a clue about liftM2:

```
*> : type (−)
(−) :: Num a ⇒ a → a → a
*> : type liftM2 (−)
liftM2 (−) :: (Num a, Monad m) ⇒ m a → m a → m a
```
Can you now guess what \texttt{liftM2} does?

The Haskell interpreter has figured out (or, inferred) the type of the overall expression \texttt{t1}, and can tell it to us

\begin{verbatim}
*>:t t1
\end{verbatim}

\begin{verbatim}
t1 :: Cont w Integer
\end{verbatim}

It is an effectful expression, within a particular monad \texttt{Cont w}, which is parametrized by the so-called answer-type, to be discussed in more detail in below. To see the result, we have to \textit{run} the expression, executing all its effects and obtaining its eventual result:

\begin{verbatim}
*>:runC t1
\end{verbatim}

\begin{verbatim}
-- 9
\end{verbatim}

The expression \texttt{t1} looks ugly, even to a Schemer. We can make it prettier:

\begin{verbatim}
t1' = liftM2 (\sim)
(reset
(liftM2 (+) (return 3)
(shift (\lambda k \rightarrow return (5*2))))
(return 1)
\end{verbatim}

resorting to a monad law. Can you tell which law we have used, and how?

A few \textit{syntactic}, this time, embellishments – defining infix operators for ‘lifted’ numeric operations

\begin{verbatim}
infixl 6 \sim!
infixl 7 \star!
\end{verbatim}

\begin{verbatim}
(\sim!),(+!),(\star!) :: (Num a, Monad m) \Rightarrow m a \rightarrow m a \rightarrow m a
(\sim!) = liftM2 (\sim)
(+!) = liftM2 (+)
(\star!) = liftM2 (\star)
\end{verbatim}

make the expression prettier still:

\begin{verbatim}
t12 = reset (return 3 +! shift (\lambda k \rightarrow return (5*2))) \sim! return 1
\end{verbatim}

The expression looks almost like the one in OchaCaml. The remaining \texttt{returns} betray implicit effects of our expressions and of the operations on them. We can banish these \texttt{returns} and make the code look exactly like the OchaCaml code. We leave this task as a homework for an interested reader. In this tutorial, we shall keep \texttt{return}, as a reminder of effects.

Let us re-write another previously seen OchaCaml example

\begin{verbatim}
fst (reset (fun () \rightarrow let x = ("hi","bye") in (x, x)))
\end{verbatim}

into Haskell:

\begin{verbatim}
t13 = liftM fst (reset $ do
  x <- shift (\lambda k \rightarrow return ("hi","bye"))
  return (x,x))
\end{verbatim}
The \texttt{let} form of OchaCaml binds the result of a potentially effectful expression to a local variable. In Haskell, we use the \texttt{do} form for that purpose.

### 1.2 Applying the extracted continuations

Earlier in the tutorial we have learned how to extract a delimited continuation as a function, which we can later apply to various arguments. If we are going to use the captured continuation within a single expression, we can combine extraction with use:

\[
t_2 = \text{reset} \ (\text{return} \ 3 +! \ \text{shift} \ (\lambda k \rightarrow \text{return} \ (k \ (5 \ast 2)))) -! \ \text{return} \ 1
\]

\[
* > \text{runC} \ t_2
\]

\[
\text{---} \ 12
\]

Rather than returning the extracted delimited continuation, we return the result of the expression that uses that continuation. This is the most frequent pattern of using \texttt{shift}.

We can apply the captured continuation more than once within the same expression:

\[
t_3 :: \text{Cont} \ \text{Int} \ \text{Int}
\]

\[
t_3 = \text{reset} \ (\text{return} \ 2 +! \ \text{shift} \ (\lambda k \rightarrow \text{return} \ $ k \ (k \ 10))) +! \ \text{return} \ 1
\]

This time we have explicitly specified the expected type of the expression in its signature. We did not have to do that: GHCi could have inferred the type. Writing signatures of all top-level definitions is considered a good style, regardless of whether they could be inferred. After all, if we do not have even a vague idea of what a new function is to do, perhaps we should not rush into writing its code.

The type of \(t_3\) states that it is a computation that produces an \texttt{Int} and may also throw `exceptions` of the type \texttt{Int}. Therefore, we obtain an \texttt{Int} either way. Can you determine the result of running the expression in your head, without using GHCi? (Hint: if you have trouble, read further about the bubble-up semantics, and then do the exercise using that semantics. After doing that, see \S\textcolor{red}{1.4.3}.)

### 1.3 The \texttt{Cont} monad

The \texttt{Cont} monad used so far comes from a monad transformer library, a part of the Haskell Platform. For reference, we show its definition below. As any monad, \texttt{Cont} is defined by a type constructor, which is, in our case, is parametrized by the answer-type \(w\).

\[
\text{newtype} \ \texttt{Cont} \ w \ a = \text{Cont}\{\text{runCont} : (a \rightarrow w) \rightarrow w\}
\]
To complete the specification, we have to define two basic operations on \( \text{Cont} \ w \ a \), return and \((\gg=)\) (pronounced ‘bind’). In other words, we have to make \( \text{Cont} \ w \) an instance of the class \textbf{Monad}:

\[
\text{instance Monad (Cont w) where}
\]

\[
\begin{align*}
\text{return} \ x = & \text{Cont} (\lambda k \to k \ x) \\
\text{Cont} m \gg= f = & \text{Cont} (\lambda k \to (\text{runCont} \ (f \ v) \ k)) \\
\end{align*}
\]

Each monad is also an applicative functor:

\[
\text{instance Functor (Cont w) where}
\]

\[
\begin{align*}
\text{fmap} \ f \ (\text{Cont} \ m) = & \text{Cont} (\lambda k \to (\text{fmap} \ (k \circ f)) \\
\end{align*}
\]

\[
\text{instance Applicative (Cont w) where}
\]

\[
\begin{align*}
\text{pure} = & \text{return} \\
\text{m} \ll= a = m \gg= \lambda h \to \text{fmap} \ h \ a \\
\end{align*}
\]

The remaining operations to capture, delimit and run the \text{Cont} monad computations are not part of the Haskell Platform libraries. These operations are easy to define, as shown below:

\[
\begin{align*}
\text{runC} :: & \text{Cont} \ w \ w \to w \\
\text{runC} \ m = & \text{runCont} \ m \ \text{id} \\
\text{reset} :: & \text{Cont} \ a \ a \to \text{Cont} \ w \ a \\
\text{reset} = & \text{return} \circ \text{runC} \\
\text{shift} :: & ((a \to w) \to \text{Cont} \ w \ w) \to \text{Cont} \ w \ a \\
\text{shift} \ f = & \text{Cont} \ (\text{runC} \circ f) \\
\end{align*}
\]

1.4 Justifying the \text{Cont} implementation of delimited control

How do we know that the above definitions of \text{shift} and \text{reset} are ‘correct’ and that computed results shall always match those of OchaCaml? We need a specification for delimited control, and we need to demonstrate that the \text{Cont} implementation matches the specification.

1.4.1 The bubble-up semantics

For specification we take the so-called ‘bubble-up semantics’, which was the original semantics of delimited control (prompt/control) introduced by Felleisen. The bubble-up semantics was re-discovered for the so-called \( \lambda \mu \)-calculus, which is the calculus for classical logic.

The operator \text{shift} introduces a bubble:

\[
\text{shift} \ \text{body} \rightarrow \ \circ \text{id} \ \text{body}
\]

The bubble percolates up, devouring the neighboring operations:
(泡 k body) +e1  →  泡(λx →(k x) +e1) body
(泡 k body) e1  →  泡(λx →(k x) e1) body
f (泡 k body)  →  泡(λx →f (k x)) body
if (泡 k body) then e1 else e2  →  泡(λx →if (k x) then e1 else e2) body

The operator reset ‘pricks’ (or, eliminates) the bubble:
reset (泡 k body)  →  reset (body (λx →reset (k x)))

If an expression evaluates to a value rather than a bubble, reset just returns the value.
reset value  →  value

1.4.2 Proving that the implementation matches the specification

To start with, we η-expand the definition of shift :
shift body = Cont (λk →runC (body (λu →runCont (return u) k)))

to make clear the representation of the bubble in the Cont monad:
泡 ctx body = Cont (λk →runC (body (λu →runCont (ctx u) k)))

We will now demonstrate, using equational reasoning, that the Cont bubble propagation matches the rules of the bubble-up semantics. We show the detailed proof for one propagation rule:
(泡 k body) e1  →  泡(λx →(k x) e1) body

The others are similar.

It the applicative/monadic notation, the application of 泡 k body to an effectful expression e1 is written as 泡 ctx body <<< e – or, expanding <<< in terms of bind, 泡 ctx body >>>=λh →(fmap h e). We calculate:
泡 ctx body <<< e
≡
泡 ctx body >>>=λh →(fmap h e)
≡
Cont (λks →runC (body (λu →runCont (ctx u) ks))) >>>=
  λh →(fmap h e)
≡
Cont (λk →
  (λks →runC (body (λu →runCont (ctx u) ks)))
  (λν →runCont ((λh →(fmap h e)) ν) k))
≡
Cont (λk →
  (λks →runC (body (λu →runCont (ctx u) ks)))
  (λν →runCont (fmap ν e) k))
≡
Cont (λk →
runC (body (λu → runCont (ctx u) (λv → runCont (fmap v e) k))))

− an inner expression is almost the same as
− let g = (λv → fmap v e) in
− ctx u >>= g ≡
− Cont (λk1 → runCont (ctx u) (λv → runCont (g v) k1))
− modulo the replacement of k1 with k
≡
Cont (λk → runC (body (λu → runCont (ctx u >>= λv → fmap v e) k)))
≡
let ctx' = ctx u >>= λv → fmap v e in
Cont (λk → runC (body (λu → runCont (ctx' u) k)))
≡
let ctx' = ctx u >>= λv → fmap v e in
泡 (λu → ctx u >>= λv → fmap v e) body
≡
泡 (λu → ctx u <> e) body

Which laws justify each step in the above equational derivation?

The result matches the conclusion of the bubble-up semantics rule:

(泡 k body) e1 → (泡 (λx → (k x) e1) body

Let us check that the Cont bubble elimination matches the specification:
reset (泡 ctx body) → reset (body (λx → reset (ctx x)))

We calculate:

reset (泡 ctx body)
≡
reset (Cont (λk → runC (body (λu → runCont (ctx u) k))))
≡
return (runC (Cont (λk → runC (body (λu → runCont (ctx u) k)))))
≡
return ((λk → runC (body (λu → runCont (ctx u) k))) id)
≡
return (runC (body (λu → runCont (ctx u) id))
≡
reset (body (λu → runCont (ctx u) id))
≡
reset (body (λu → runC (ctx u)))

If the captured continuation, passed to the body, were of the type a → Cont w a, we would have added return. The result of the equational reasoning would have had matched the specification to the letter, keeping in mind that reset is return ◦ runC. We observe that the continuation captured by shift is always a pure function (that is, has no effects). Our definition of shift made this fact explicit in the type of the captured continuation. Therefore, we do not need the spurious return.
Finally the Cont-monad reset when applied to a value returns the value, as required by the bubble-up semantics:

\[
\begin{align*}
\text{reset} \ (\text{return} \ v) & \equiv \\
\text{reset} \ (\text{Cont} \ (\lambda k \rightarrow k \ v)) & \equiv \\
\text{return} \ (\text{runC} \ (\text{Cont} \ (\lambda k \rightarrow k \ v))) & \equiv \\
\text{return} \ ((\lambda k \rightarrow k \ v) \ \text{id}) & \equiv \\
\text{return} \ v & \equiv
\end{align*}
\]

Our implementation of delimited control indeed matches the specification.

1.4.3 Bubble-up semantics in practice

If determining the result of \( t3 \), §1.2, in your head was difficult, let us see how the bubble-up semantics helps. The bubble-up semantics makes determining the result of any expression with shift a pure mechanical operation:

\[
\begin{align*}
\text{runC} \ (\text{reset} \ (\text{return} \ 2 \ \ast! \ \text{shift} \ (\lambda k \rightarrow \text{return} \ $ k (k \ 10))) \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{shift introduces the 逃跑} \\
\text{runC} \ (\text{reset} \ (\text{return} \ 2 \ \ast! \ \text{逃跑} \ \text{return} \ (\lambda k \rightarrow \text{return} \ $ k (k \ 10))) \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{the 逃跑 propagates up and devours return} \ 2 \ \ast! \\
\text{runC} \ (\text{reset} \\
\text{  (逃跑} \ (\lambda x \rightarrow \text{return} \ 2 \ \ast! \ \text{return} \ x) \\
\text{  \ (\lambda k \rightarrow \text{return} \ $ k (k \ 10))) \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{simplifying using the monad law} \\
\text{runC} \ (\text{reset} \\
\text{  (逃跑} \ (\lambda x \rightarrow \text{return} \ (2 \ \ast x)) \\
\text{  \ (\lambda k \rightarrow \text{return} \ $ k (k \ 10))) \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{reset pricks the 逃跑} \\
\text{runC} \ (\text{reset} \ ((\lambda k \rightarrow \text{return} \ $ k (k \ 10))) \ (\lambda x \rightarrow \text{runC} \ (\text{return} \ (2 \ \ast x)))) \\
\text{  \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{runC} \circ \text{return} \equiv \text{id} \\
\text{runC} \ (\text{reset} \ ((\lambda k \rightarrow \text{return} \ $ k (k \ 10))) \ (\lambda x \rightarrow 2 \ \ast x)) \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{beta-reduction} \\
\text{runC} \ (\text{reset} \ (\text{return} \ 40) \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{reset of a value} \\
\text{runC} \ (\text{return} \ 40) \ +! \ \text{return} \ 1) & \equiv \ --- \ \text{monad law (the addition of pure expressions)} \\
\text{runC} \ (\text{return} \ 41) & \equiv \\
\end{align*}
\]

41