Iteratees

Christoph-Simon Senjak

Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

TCS Oberseminar Wintersemester 2015/16

I/O in purely functional languages

- IO-Monad with handles
- Uniqueness types with handles
- Lazy I/O

IO-Monad with handles - 1

```
appendF h = hIsEOF h >>=
            \ e -> if e then return ()
                   else ( hGetSome h 4096 >>=
                          \bs->hPut stdout bs>>=
                          \ _ -> appendF h )

appendFS [] = return ()
appendFS (x : l) = openFile x ReadMode >>=
                   \h -> ( appendF h >>=
                           \_ -> hClose h) >>=
                   \_ -> appendFS l

main = getArgs >>= appendFS
```

IO-Monad with handles - 2

```
appendF h = do e <- hIsEOF h
               if e then return ()
               else do bs <- hGetSome h 4096
                       hPut stdout bs
                       appendF h

appendFS [] = return ()
appendFS (x : l) = do h <- openFile x ReadMode
                      appendF h
                      hClose h
                      appendFS l

main = getArgs >>= appendFS
```

IO-Monad with handles

- \+ Control over memory and buffer sizes.
- \+ Good for fine-tuning.
- \+ Predictable.
- \+ Separates state from purely functional code.
- \- Complicated to use.
- \- Complicated to reason about formally.

Uniqueness Types

Example from
http://www.inf.ufsc.br/~jbosco/cleanBookI.pdf:

```
module hello
import StdEnv
Start :: *World -> *World
Start world
 # (c,world)  = stdio world
   c          = fwrites "What is your name?\n" c
   (name,c)   = freadline c
   c          = fwrites ("Hello " +++ name) c
   (_,c)      = freadline c
   (ok,world) = fclose c world
 | not ok     = abort "Cannot close console"
 | otherwise  = world
```

Uniqueness Types

- \+ Control over memory and buffer sizes.
- \+ Predictable.
- \+ Comparably simple to use (though verbose).
- \- Requires support in the type system.

Lazy I/O

```
getFile n = do h <- openFile n ReadMode
               bs <- hGetContents h
               hClose h
               return bs

catFiles l = do seq <- sequence (map getFile l)
                return $ concat seq

main = getArgs >>= catFiles >>= hPut stdout
```

Lazy I/O

+ Simple and intuitive to use.
+ Not verbose.
- No control over memory and buffer sizes.
- Unpredictable.
- “Simulates” purity, pretends absence of stateful operations.

Our Implementation(s) of Deflate

We use the notions of

- **Parsability**:

  $$\text{Parsable}\, R :\Leftrightarrow \forall_l.(\lambda_X.X + \neg X)(\exists_{a,l_1,l_2}.R\, a\, l_1 \wedge l = l_1 +\!\!+ l_2)$$

  This extracts (roughly) to

  ```
  [A] -> Maybe ([B], [A], [A])
  ```

  and reads as much of the argument as possible, returns the extracted/converted cleartext, the consumed list, and the remaining list.

- **Strong Uniqueness**:

  $$\begin{aligned}\text{StrongUnique}\, R :\Leftrightarrow \quad & \forall_{a,b,l_a,l'_a,l_b,l'_b}.l_a +\!\!+ l'_a = l_b +\!\!+ l'_b \rightarrow \\ & R\, a\, l_a \rightarrow R\, b\, l_b \rightarrow a = b \wedge l_a = l_b\end{aligned}$$

$\Rightarrow$ Composable, pluggable, so ideal for parsing. But relies on lazy I/O for big files.

Our Implementation(s) of Deflate

Alternatives:

- Uniqueness Types – hard to realize in Coq: Either extend the type system, or work in an own semantic model (like VST does).
- State Monads – very hard to reason about non-trivial examples (in absence of special tactic collections like IMP or VST).

Iteratees

(See http://okmij.org/ftp/Haskell/Iteratee/describe.pdf)

```
data Iteratee i o
    = Done o
    | Next (Maybe i -> Iteratee i o)

type Enumerator i o =
    Iteratee i o -> Iteratee i o

listEnumerator :: [i] -> Enumerator i o
listEnumerator _ it@(Done _) = it
listEnumerator [] (Next f) = f Nothing
listEnumerator (x : l) (Next f) =
  listEnumerator l (f (Just x))
```

Iteratees

For actual I/O, in Haskell it is usually implemented as monad transformer:

```
data Iteratee m i o
    = Done (m o)
    | Next (Maybe i -> m (Iteratee m i o))

type Enumerator m i o = Iteratee m i o -> m o

type Enumeratee m i o a =
  Iteratee m i a ->
  Iteratee m o (Iteratee m i a)
```

Iteratees

- We can add additional branches, for example to pass more than one character at a time.
- An iterator that returns an iterator is called *enumeratee*.
- Such constructs can be "piped" into one another, similar to unix pipes and processes.
- The syntax can be made similar to that of lazy I/O, while retaining control over cache and memory.

Iteratees in Coq

In Coq, we want strong guarantees:

```
Inductive Iteratee {I O : Set}
    {R : list I -> O -> Prop} (l1 : list I) :=
| Done : forall o, R l1 o -> Iteratee l1
| Error : string ->
           (forall l2 o, ~ R (l1 ++ l2) o)
          -> Iteratee l1
| Proceed : (forall l2, Iteratee (l1 ++ l2))
                     -> Iteratee l1.
```

This is only a first prototype. It will likely change, depending on what we actually need to prove our theorems.

Conclusion and Future Work

- By using Iteratees, we hope to make the trusted codebase of our Deflate implementation smaller: In theory, only the extraction to OCaml and a small File-I/O wrapper should be needed (where now we use GHC and lazy I/O).
- The same technique could then be used for other verified parsers.
- Can a similar concept be defined for other stateful operations (like Arrays)?

The Final Slide