There’s Something about Haskell

Lecture Notes for CS520
The Haskell Language

• Haskell is a “popular” functional programming language
  – Non-strict
  – Pure
  – Type classes
  – Monads
  – Higher ranked polymorphism and existential types
  – Guarded recursive datatypes
  – …
Haskell vs. ML

• ML (and its variants):
  – Call by value
  – Side effects (reference, exceptions, I/O, …)
  – …

• Haskell:
  – Call by name (lazy evaluation, non-strictness)
  – A pure language
  – …
Road Map

- The non-strictness in Haskell
- The support to side effects in Haskell
An Example of Call-by-Name

\[ f = \lambda x. \lambda y. \text{if } x = 0 \text{ then } 0 \text{ else } y \]

show \( f \ 0 \ (1+2) \)
show \( f \ 1 \ (1+2) \)

strict
non-strict

not evaluated
evaluated
Call by Name

• May save some computation
  – An expression is not evaluated if it is never used.

• May also duplicate some computation
  – let f = \lambda x. (x, x) in f (1+2)
Call by Need

• An efficient implementation of call-by-name
  – Employ the idea of memoization

• When accessing an expression
  – If it is already computed and cached, then use the cached value
  – otherwise, compute the expression to a value and put the value into cache
Call by Need

• Time overheads
  – Caused by the indirect memory access
  – Inspires the research of strictness analysis

• Space overheads
  – Caused by the heap space for caching
  – Inspires the research of “once upon a type”
Non-strictness and Infinite Data Structure

fil i l = filter p l where p x = not (x `mod` i == 0)
nat2 = 2 : {x + 1 | x ∈ nat2}
-- nat2 = [2, 3, 4, …]
gtp (x : xs) = x : gtp (fil x xs)
primes = gtp nat2
-- primes = [2, 3, 5, 7, …]

> Take 25 primes
> [2, 3, 5, 7, …, 83, 89, 97]
Road Map

• The non-strictness in Haskell
• The support to side effects in Haskell
Side Effects and Monad

• In Haskell side effects are supported through monads
• Monads
  – A category theory notion used in the formalization of the denotational semantics of programming languages
  – Introduced into real functional programming languages (Moggi88), and implemented in Haskell
  – Many language features can be captured by monads, such as exceptions, reference, I/O, continuations, non-determinism, etc.
Monads

• A monad is a unary type constructor associated with two operations
  – M: the type constructor of the kind: * → *
  – return: an operation of the type ∀ a. a → M a
  – Bind: an operation of the type
    ∀ a, b. M a → (a → M b) → M b
Monad Laws

• Left unit:
  – (return x) `bind` k = k x

• Right unit
  – m `bind` return = m

• Associativity
  – (m `bind` k) `bind` h =
    m `bind` (λ x. (k x) `bind` h)
An Interpreter Example

\[ E ::= n \mid E + E \]
\[ \text{data } E = I \text{ Int} \mid \text{Add } E \ E \]
\[ \text{data } \text{Id } t = \text{Id } t \]
\[ \text{return } x = \text{Id } x \]
\[ \text{bind } m \ k = \text{let } \text{Id } x = m \ \text{in } k \ x \]
\[ \text{type } M = \text{Id} \]

\[ \text{eval } :: E \rightarrow M \ \text{Int} \]
\[ \text{eval } (I \ n) = \text{return } n \]
\[ \text{eval } (\text{Add } e_1 \ e_2) = \]
\[ \quad \quad \text{eval } e_1 \ \text{`bind`} (\lambda \ x. \]
\[ \quad \quad \quad \text{eval } e_2 \ \text{`bind`} (\lambda \ y. \]
\[ \quad \quad \quad \quad \text{return } x + y)) \]


\[ E ::= n \mid E + E \mid E / E \]

\[
\begin{align*}
\text{data } &E = \text{I Int} \mid \text{Add } E \ E \mid \text{Div } E \ E \\
\text{data } &X \ t = \text{Norm } t \mid \text{Excp} \\
\text{return } &x = \text{Norm } x \\
\text{bind } &m \ k = \\
\text{case } &m \text{ of} \\
&\text{Norm } x \to k \ x \\
&\text{Excp} \to \text{Excp} \\
\text{type } &M = X \\
\text{eval } :: E \to M \ \text{Int} \\
\text{eval } (\text{I } n) &\text{ = …} \\
\text{eval } (\text{Add } e1 \ e2) &\text{ = …} \\
\text{eval } (\text{Div } e1 \ e2) &\text{ = } \\
&\text{eval } e1 \text{ `bind’ (λ } x. \\
&\text{eval } e2 \text{ `bind’ (λ } y. \\
&\text{if } y == 0 \text{ then Excp} \\
&\text{else return (x `div` y)))}
\end{align*}
\]
Enhance the Interpreter

type \( S \ t = \text{Int} \rightarrow X \ (t, \text{Int}) \)

\[
\text{return } x = \lambda s. \text{Norm} (s, x)
\]

\[
\text{bind } m \ k = \lambda s. \ \\
\begin{cases} \\
\text{case } m \ s \ of \\
\text{Norm} (s', x) & \rightarrow k \ x \ s' \\
\text{Excp} & \rightarrow \text{Excp}
\end{cases}
\]

tick :: S ()

\[
tick \ s = \text{Norm} (s + 1, \ () )
\]

raise :: \( \forall \ a. \ S \ a \)

\[
\text{raise} = \lambda s. \text{Excp}
\]

type \( M = S \)

eval :: E \rightarrow M \text{Int}

\[
eval (I \ n) = \ldots
\]

\[
eval (\text{Add} \ e1 \ e2) = \ldots
\]

\[
eval (\text{Div} \ e1 \ e2) = \\
\text{eval e1} \ `\text{bind}` (\lambda x. \\
\text{eval e2} \ `\text{bind}` (\lambda y. \\
\text{if } y == 0 \text{ then } \text{raise} \\
\text{else } \text{tick} \ `\text{bind}` (\lambda () . \\
\text{return (x } `\text{div} \ y))))
\]
Some Resources

• Haskell
  – http://www.haskell.org

• Monads
  – http://homepages.inf.ed.ac.uk/wadler/topics/monads.html