Summary: This lecture discusses the construction of a simple interpreter for Haskell.
Interpretation

- The simplest implementation of any programming language is an interpreter
- A typical interpreter has the following features
  - it evaluates programs directly in the source language (more-or-less)
  - little or no compilation time
  - good error messages
  - tracing facilities usually available
  - slow execution speed
  - e.g. Hugs, MacGofer
- Often, an interpreter is used for rapid prototyping of programs or testing ideas
  - good interpreters allow mixed execution of source code and compiled code
Interpretation of Haskell

• The **source language** is the language in which programs are written (in our case Haskell)

• The **implementation language** is the language in which the interpreter itself is written (could be anything, most commonly either the source language or C)
  – obviously the first interpreter for Haskell wasn’t written in Haskell!
  – c.f. boot-strapping

• We define the **abstract syntax** of the source language as a datatype in the implementation language
  – the abstract syntax defines the **information content** of each program construct

• The interpreter is a function that takes a value of this datatype and returns the WHNF of this value represented in the same datatype
\textbf{\(\beta\)-reduction and names}

- Consider the \(\beta\)-reduction \( (\lambda x. E) \ E' \to^{\beta} E[E'/x] \)
  
  - the value of \((\lambda x. E) \ E'\) is the value of \(E\) where each free instance of \(x\) is replaced by \(E'\)

- There are two standard methods of recording the replacement of \(x\) by \(E'\)

  1. evaluate the \textbf{original} \(E\) in an \textbf{environment} containing the association \(x \to E'\)
     
     - this is the basis of most interpretation techniques

  2. make a \textbf{new copy} of \(E\) with each instance of \(x\) replaced by \(E'\)
     
     - this is the basis of \textbf{graph reduction}, the most popular compilation technique for Haskell
Environment-based model

- Bindings of names to expressions/values are kept in a local environment
- Function application (β-reduction) involves binding the names of the formal arguments of the equation to the values of the actual arguments, adding these bindings to the environment and evaluating the body of the equation
- Sharing is expressed by having multiple instances of a name
- Given the equation
  \[ f \ x \ y \ z = x + g \ x \ z \]
  and the machine state
  \[(f \ A1 \ A2 \ A3, \rho)\]
  we go to the machine state
  \[(x + g \ x \ z, \rho[x \rightarrow A1, \ y \rightarrow A2, \ z \rightarrow A3])\]
- When the value of an argument is required, its environment entry is looked-up
- When the expression to which an argument is bound is evaluated, its environment entry is updated
Graph-based model

- Function application involves making a copy of the body of the equation with instances of the formal arguments replaced by the actual arguments.
- Sharing is expressed with multiple pointers to a sub-graph.
- When an expression is evaluated, the root node of its representation is updated to point to the representation of the evaluated expression.
- Given the equation
  \[ f(x, y, z) = x + g(x, z) \]
  and the machine state

we go to the machine state

\[ f \]
\[ + \]
\[ g \]
\[ A1 \]
\[ A3 \]
\[ A2 \]
\[ A3 \]
\[ A1 \]
A simple representation of programs

type Program = [Module]

type Module = (Name, [ExportedItem],
               [ImportedModule], [Decl])

data Decl = Typedecl ...
   | Datadecl ...
   | Classdecl ...
   | Instancedecl ...
   | Funcdecl Funcinfo

type Funcinfo = (Name, Type, NoOfArgs, [Equation])

type Equation = ([Pattern], Body, WhereClause)

type WhereClause = [Funcinfo]

data Body = Simple Exp
   | Guarded [(Exp, Exp)]

• This is an example of an **unfree datatype**
  – in a **free datatype**, every object that you can build
    using the constructors is a legal value
  – in an unfree datatype, you can build some objects that
    aren’t legal values
A simple representation of expressions

```
data Exp  =  INT Int
  |  BOOL Bool
  |  ... — other primitive datatypes
  |  ARG Name
  |  PRIM Name   — assumed to be strict
  |  FUNC Name   — assumed to be lazy
  |  APPN Exp Exp
  |  LET [(Name, Exp)] Exp
  |  ... — other expression constructs

type Name = String
```
A specification of environments

\[
\text{type } \text{Localenv} = ... \\
\]

— create a new empty environment
\[
\text{emptyenv :: Localenv} \\
\]

— get a binding from the environment
\[
\text{getlocal :: Name -> Localenv -> Exp} \\
\]

— add some bindings to the environment
\[
\text{extendenv :: Localenv -> [(Name, Exp)] -> Localenv} \\
\]

— defns of user-defined FUNCs are
— kept in a global environment
A simple environment-based interpreter

\[ \text{evalprogram :: Exp -> Exp} \]

\[ \text{evalprogram} = \text{eval emptyenv} \]

\[ \text{eval :: Localenv -> Exp -> Exp} \]

\[ \text{eval loc (INT i)} = \text{INT i} \]

\[ \text{eval loc (BOOL b)} = \text{BOOL b} \]

\[ \text{eval loc (ARG v)} = \text{getlocal v loc} \]

\[ \text{eval loc (PRIM f)} = \text{PRIM f} \]

\[ \text{eval loc (FUNC f)} = \text{FUNC f} \]

\[ \text{eval loc (APPN f x)} = \text{apply loc f [x]} \]

\[ \text{eval loc (LET ds e)} \]
\[ = \text{eval newloc e} \]
\[ \text{where newloc = extendenv loc (map susp ds)} \]
\[ \text{susp (i, e')} = (i, \text{eval newloc e'}) \]
Function application

apply :: Localenv -> Exp -> [Exp] -> Exp

apply loc x [] = eval loc x

apply loc (APPN f x) args = apply loc f (x : args)

apply loc (PRIM f) args
  | length args >= n = apply loc (funof f evalnargs) (drop n args)
  | otherwise = foldl APPN (PRIM f) args
where n = arityofprim f
  evalnargs = map (eval loc) (take n args)

apply loc (FUNC f) args
  | length args >= n = apply newloc body (drop n args)
  | otherwise = foldl APPN (FUNC f) args
where (argnames, body) = getglobal f
  n = length argnames
  suspnargs = map (eval loc) (take n args)
  newloc = extendenv loc (zip argnames suspnargs)

apply loc e args = apply loc (eval loc e) args
Function application

\[
\text{arityofprim} :: \text{Name} \rightarrow \text{Int} \\
\text{arityofprim} \ "+" = 2 \\
\text{arityofprim} \ "not" = 1 \\
\]

\[
\text{funof} :: \text{Name} \rightarrow [\text{Exp}] \rightarrow \text{Exp} \\
\text{funof} \ "+" [\text{INT} m, \text{INT} n] = \text{INT} (m+n) \\
\text{funof} \ "not" [\text{BOOL} b] = \text{BOOL} \ (\text{not} \ b) \\
\]