Programming Language Concepts, CS2104
Lecture 7

Types, ADT, Haskell, Components
Reminder of Last Lecture

- Tupled Recursion
- Exceptions
Overview

- Types
- Abstract Data Types
- Haskell
- Design Methodology
Dynamic Typing

- Oz/Scheme uses dynamic typing, while Java uses static typing.
- In dynamic typing, each value can be of arbitrary types that is only checked at runtime.

Advantage of dynamic types
- no need to declare data types in advance
- more flexible

Disadvantage
- errors detected late at runtime
- less readable code
Type Notation

- Every value has a type which can be captured by:
  \[ e :: type \]

- Type information helps program development/documentation

- Many functions are designed based on the type of the input arguments
List Type

- Based on the type hierarchy
  - \( \langle \text{Value} \rangle, \langle \text{Record} \rangle, \ldots \)
  - \( \langle \text{Record} \rangle \subseteq \langle \text{Value} \rangle \)
    - The Record type is a subtype of the Value type
  - List is either \( \text{nil} \) or \( X \mid X_r \)
    where \( X_r \) is a list and \( X \) is an arbitrary value
  - \( \langle \text{List} \rangle ::= \text{nil} \mid \langle \text{Value} \rangle \mid \langle \text{List} \rangle \)
Polymorphic List

- Usually all elements of the same type
- Polymorphic list with elements of T type
  \[
  \langle \text{List } T \rangle ::= \text{nil} \mid \langle T \rangle \mid \langle \text{List } T \rangle
  \]
  - T is a type variable
  - \langle \text{List } ? \rangle \text{ is a type constructor}
  - \langle \text{List } \langle \text{Int} \rangle \rangle : \text{a list whose elements are integers}
  - \langle \text{List } \langle \text{Value} \rangle \rangle \text{ is equal to } \langle \text{List} \rangle
Polymorphic Binary Tree

- Binary trees
  \[ \langle \text{BTree T} \rangle ::= \text{leaf} \mid \text{tree}( \text{key: } \langle \text{Literal} \rangle \text{ value: } \text{T} \text{ left: } \langle \text{BTree T} \rangle \text{ right: } \langle \text{BTree T} \rangle ) \]

- Binary tree representing a dictionary mapping keys to values
- Binary tree is:
  - either a leaf (atom leaf), or
  - an internal node with label tree, with left and right subtrees, a key and a value
- Key is of literal type and the value is of type T
Types for procedures and functions

- The type of a procedure where $T_1 \ldots T_n$ are the types of its arguments can be represented by:

  \[ \langle \text{proc} \{\$ T_1 \ldots T_n\} \rangle \]

  or

  \[ \{T_1 \ldots T_n\} \rightarrow () \]
On Types: procedures and functions

- The type of a function where \( T_1 \ldots T_n \) are the types of the arguments, and \( T \) is the type of the result is:
  \[
  \langle \text{fun} \ \{ \$ \ T_1 \ldots T_n \} : T \rangle
  \]
  or
  \[
  \{ T_1 \ldots T_n \} \rightarrow T
  \]
- Append :: \( \{ \langle \text{List} \rangle \langle \text{List} \rangle \} \rightarrow \langle \text{List} \rangle \)
  or precisely :: \( \{ \langle \text{List A} \rangle \langle \text{List A} \rangle \} \rightarrow \langle \text{List A} \rangle \)
Constructing Programs from Type

- Programs that takes lists has a form that corresponds to the list type
- Code should also follow type, e.g:
  
  ```
  case Xs of
    nil then (expr1) % base case
    [] X|Xr then (expr2) % recursive call
  end
  ```
Constructing Programs from Type

- Helpful when the type gets complicated
- *Nested lists* are lists whose elements can be lists
- Exercise: “Find the number of elements of a nested list”
  \[
  Xs = \begin{bmatrix}
  \end{bmatrix}
  \]

\[
\text{\{Length } Xs \text{\}} = 5
\]

\[
\text{declare}
Xs1=\begin{bmatrix}
[1 & 2] & 4 & nil
\end{bmatrix}
\]

\[
\text{\{Browse } Xs1 \text{\}} \rightarrow \begin{bmatrix}
[1 & 2] & 4 & nil
\end{bmatrix}
\]

\[
Xs2=\begin{bmatrix}
[1 & 2] & 4
\end{bmatrix} | nil
\]

\[
\text{\{Browse } Xs2 \text{\}} \rightarrow \begin{bmatrix}
[[[1 & 2] & 4]]
\end{bmatrix}
\]
Constructing Programs from Type

- Nested lists type declaration

\[
\langle \text{NList } T \rangle ::= \text{nil} \mid \langle \text{NList } T \rangle \mid \langle \text{NList } T \rangle \mid \langle \text{NList } T \rangle \ \text{(T is not nil nor a cons)}
\]

- General structure:

```haskell
case Xs
  of nil then \langle \text{expr1} \rangle \ % \ \text{base case}
  [] X|Xr andthen \{\text{IsList X}\} \ then
    \langle \text{expr2} \rangle \ % \ \text{recursive calls for X and Xr}
  [] X|Xr then
    \langle \text{expr3} \rangle \ % \ \text{recursive call for Xr}
end
```
Constructing Programs from Type

- Length :: \(\{\text{NList T}\} \rightarrow \{\text{Int}\}\)

- fun \{Length Xs\} 
  case Xs 
  of nil then 0 % base case 
  [] X|Xr andthen \{IsList X\} then 
    \{Length X\} + \{Length Xr\} 
  [] X|Xr then 
    1 + \{Length Xr\} 
  end 
  end

- fun \{IsList L\} 
  L == nil orelse 
  \{Label L\} == '|' andthen \{Width L\} == 2 
  end
Summary so far

- Type Notation
- Polymorphic Types
- Function types
- Constructing programs from type
Abstract Data Types
Data Types

- Data type
  - set of values
  - operations on these values

- Primitive data types
  - records
  - numbers
  - …

- Abstract data types
  - completely defined by its operations (interface)
  - implementation can be changed without changing use
Motivation

- Sufficient to understand interface only
- Software components can be developed independently when they are used through interfaces.
- Developers need not know implementation details
Outlook

- How to *define* abstract data types
- How to *organize* abstract data types
- How to *use* abstract data types
Abstract data types (ADTs)

- A type is *abstract* if it is completely defined by its set of operations/functionality.
- Possible to change the implementation of an ADT without changing its use.
- ADT is described by a set of procedures
  - Including how to create a value of the ADT
- These operations are the only thing that a user of ADT can assume
Example: stack

- Assume we want to define a new data type \(\text{stack } T\) whose elements are of any type \(T\)
- We define the following operations (with type definitions)

\[
\begin{align*}
\langle \text{fun } \{ \text{NewStack} \} : \langle \text{stack } T \rangle \rangle \\
\langle \text{fun } \{ \text{Push } \langle \text{stack } T \rangle \langle T \rangle \} : \langle \text{stack } T \rangle \rangle \\
\langle \text{proc } \{ \text{Pop } \langle \text{stack } T \rangle ?\langle T \rangle ?\langle \text{stack } T \rangle \} \rangle \\
\langle \text{fun } \{ \text{IsEmpty } \langle \text{stack } T \rangle \} : \langle \text{Bool} \rangle \rangle
\end{align*}
\]
Example: stack (algebraic properties)

- Algebraic properties are logical relations between ADT’s operations.
- Operations normally satisfy certain laws (properties).
  - \{IsEmpty \{NewStack\}\} = true
  - For any stack \( S \), \{IsEmpty \{Push S\}\} = false
  - For any \( E \) and \( S \), \{Pop \{Push S E\} E S\} holds
  - For any stack \( S \), \{Pop \{NewStack\} S\} raises error
stack (implementation I) using lists

fun {NewStack} nil end
fun {Push S E} E|S end
proc {Pop E|S ?E1 ?S1}
    E1 = E
    S1 = S
end
fun {IsEmpty S} S==nil end
stack (implementation II) using tuples

fun {NewStack} emptyStack end
fun {Push S E} stack(E S) end
proc {Pop stack(E S) E1 S1}
  E1 = E
  S1 = S
end
fun {IsEmpty S} S==emptyStack end
Why is Stack Abstract?

- A program that uses the stack will work with either implementation (gives the same result)

```plaintext
declare Top S4  
% ... either implementation  
S1={NewStack}  
S2={Push S1 2}  
S3={Push S2 5}  
{Pop S3 Top S4}  
{Browse Top}  \rightarrow 5
```
What is a Dictionary?

- A **dictionary** is a *finite mapping* from a set of simple constants to a set of language entities.
- The constants are called **keys** because they provide a unique the path to each entity.
- We will use atoms or integers as constants.
- **Goal:** create the mapping dynamically, i.e., by adding new keys during the execution.
Example: Dictionaries

- Designing the interface of Dictionary
  
  MakeDict :: {} → Dict
  returns new dictionary

  DictMember :: {Dict Feature} → Bool
  tests whether feature is member of dictionary

  DictAccess :: {Dict Feature} → Value
  return value of feature in Dict

  DictAdjoin :: {Dict Feature Value} → Dict
  return adjoined dictionary with value at feature

- Interface depends on purpose, could be richer.
Implementing the Dict ADT

- Two possible implementations are
  - based on pairlists
  - based on records

- Regardless of implementation, programs using the ADT should work!
  - the interface is a *contract* between use and implementation
Dict: List of Pairs

fun {MakeDict}
   nil
end

fun {DictMember D F}
   case D of
      nil then false
      [] G#X|Dr then if G==F then true
         else {DictMember Dr F} end
   end
end

Example: telephone book
[name1#62565243 name2#67893421 taxi1#65221111...]
Dict: Records

fun {MakeDict} {MakeRecord d []} end

fun {DictMember D F} {HasFeature D F} end

fun {DictAccess D F} D.F end

fun {DictAdjoin D F X}
    {AdjoinAt D F X}
end

■ Example: telephone book

d(name1:62565243 name2:67893421
taxi1:65521111...)
Example: Frequency Word Counting

```haskell
local

fun {Inc D X}
  if {DictMember D X} then
    {DictAdjoin D X {DictAccess D X}+1}
  else {DictAdjoin D X 1}
end

in

fun {Cnt Xs}
  % returns dictionary
  {FoldL Xs Inc {MakeDict}}
end
end

{Inc mr(a:3 b:2 c:1) b} → mr(a:3 b:3 c:1)
```
Example: Frequency Word Counting

```ml
local
  fun {Inc D X}
    if {DictMember D X} then
      {DictAdjoin D X {DictAccess D X} + 1}
    else {DictAdjoin D X 1}
  end
end

in
  fun {Cnt Xs}
    % returns dictionary
    {FoldL Xs Inc {MakeDict}}
  end
end

{Browse {Cnt [a b c a b a]}} \rightarrow \text{mr}(a:3 \ b:2 \ c:1)
```

homework: understand and try this example!
Evolution of ADTs

- Important aspect of developing ADTs
  - start with simple (possibly inefficient) implementation
  - refine to better (more efficient) implementation
  - refine to carefully chosen implementation
    - hash table
    - search tree

- Evolution is local to ADT
  - no change to external programs needed!
Theoretically

- Polymorphic type is related to Universal Type

  ```
  fun {Id X} X end
  Id :: A \rightarrow A
  Universal type : \forall A. A \rightarrow A
  ```

- ADT can be implemented using existential type.
  - \exists A. type
  - where A is considered to be hidden/abstracted
Example

- Say we want to Peano-number ADT

\[
\text{Expr} = (\text{fun } \{\text{MakeSucc } N: \text{Nat}\} \{\text{Succ } N\} \text{ end}, \text{fun } \{\text{MakeZero}\} 0: \text{Nat end})
\]

This implementation currently has type:
\[(\text{Nat } \rightarrow \text{Nat}, \text{Nat})\]

- Can make into existential type using:

\[
\text{pack Nat as } N \text{ in Expr}
\]

which will now have a more abstract type:
\[\exists N. (N \rightarrow N, N)\]
Haskell

Typeful and Lazy Functional Language
Typeful Programs

- Every expression has a statically determined type that can be declared or inferred
- Equations defined by pattern-matching equations

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer} \n\]
\[
\begin{align*}
\text{fact 0} & = 1 \\
\text{fact } n \mid n > 0 & = n \times \text{fact } (n-1)
\end{align*}
\]
Lazy Evaluation

- Each argument is not evaluated before the call but evaluated when needed (e.g. when matched against patterns)

```haskell
andThen :: Bool -> Bool -> Bool
andThen True x = x
andThen False x = False
```
Type Declaration

- Data types have to be declared/enumerated.

```
data Bool = True | False
data ListInt = Nil | Cons Integer ListInt
type PairInt = (Integer, Integer)
```
Polymorphic Types

- Generic types can be defined with type variables.

```haskell
data BTree a = Empty
             | Node a (BTree a) (BTree a)

type BTreeInt = BTree Int

size :: BTree a -> Integer
size Empty = 0
size (Node v l t) = 1 + (size l) + (size t)
```
Currying

Functions with multiple parameters may be partially applied.

$$\text{add} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$$
$$\text{add } x \ y = x+y$$

$$\text{addT} :: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}$$
$$\text{addT}(x,y) = x+y$$

Valid Expressions:

$$(\text{add } 1 \ 2) \quad = \quad \text{addT}(1,2)$$
$$(\text{add } 1) \quad = \quad \lambda \ y \rightarrow \text{addT}(1,y)$$
Type Classes

- Some functions work on a set of types. For example, sorting works on data values that are comparable.

- Wrong to use polymorphic types!

  \[
  \text{sort} :: (\text{List } a) \rightarrow (\text{List } a)
  \]

- Use type class \texttt{Ord a} instead.

  \[
  \text{sort} :: \text{Ord } a \Rightarrow (\text{List } a) \rightarrow (\text{List } a)
  \]
Type Classes

- Class is characterized by a set of methods

```haskell
class Eq a
    ==  :: a -> a -> Bool

class Eq a => Ord a
    >,  >=  :: a -> a -> Bool
    a>=b = (a>b) or (a==b)
```
Type Classes

- Need to define instances of given class

\[
\text{instance Ord Int}
\]
\[
a > b = a > \text{Int} b
\]

\[
\text{instance Ord } a \Rightarrow \text{Ord } [a]
\]
\[
[] > ys = \text{False}
\]
\[
x:xs > [] = \text{True}
\]
\[
x:xs > y:ys = x > y \text{ or } (x == y \text{ and } xs > ys)
\]

*lexicographic ordering*
Classes in Standard Library

- **Eq**: All except IO, (->)
- **Show**: All Prelude types
- **Read**: All except IO, (->)
- **Ord**: All except (->)
  - IO, IOError
- **Num**: Int, Integer, Float, Double
- **Bounded**: Int, Char, Bool, Ordering, Tuple
- **Enum**: 0, Bool, Char, Ordering, Int, Integer, Float, Double
- **Real**: Int, Integer, Float, Double
- **Fractional**: Float, Double
- **Integral**: Int, Integer
- **RealFrac**: Float, Double
- **Floating**: Float, Double
- **Monad**: IO, [], Maybe
- **RealFloat**: Float, Double
- **MonadPlus**: IO, [], Maybe
- **Functor**: IO, [], Maybe
Multi-Parameter Type Classes

- Can support generic type constructors

```haskell
class Functor f where
    fmap :: (a \to b) \to f a \to f b

instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node l r)
        = Node (fmap f l) (fmap f r)
```
Design methodology

Standalone applications
Design methodology

- “Programming in the large”
  - Written by more than one person, over a long period of time

- “Programming in the small”
  - Written by one person, over a short period of time
Design methodology. Recommendations

- **Informal specification**: inputs, outputs, relation between them
- **Exploration**: determine the programming technique; split the problem into smaller problems
- **Structure and coding**: determine the program’s structure; group related operations into one module
- **Testing and reasoning**: test cases/formal semantics
- **Judging the quality**: Is the design correct, efficient, maintainable, extensible, simple?
Software components

- Split the program into **modules** (also called **logical units, components**)

- A module has two parts:
  - An **interface** = the visible part of the logical unit. It is a record that groups together related languages entities: procedures, classes, objects, etc.
  - An **implementation** = a set of languages entities that are accessible by the interface operations but hidden from the outside.
Module

declare MyList in
local
  proc {Append ... } ... end
  proc {Sort ... } ... end
...

in
  MyList = `export'( append:Append
                   sort : Sort
                   ... )

end
Modules and module specifications

- A module specification (e.g. functor) is a template that creates a module (component instance) each time it is instantiated.

- In Oz, a functor is a function whose arguments are the modules it needs and whose result is a new module.
  - Actually, the functor takes module interfaces as arguments, creates a new module, and returns that module’s interface!
Functor

fun {MyListFunctor}
   proc {Append ... } ... end
   proc {Sort ... } ... end
   ...
in
   'export'( append : Append
      sort : Sort
      ... )
end
Modules and module specifications

- A **software component** is a unit of independent deployment, and has no persistent state.
- A **module** is the result of installing a **functor** in a particular **module environment**.
- The **module environment** consists of a set of modules, each of which may have an execution state.
Functors

- A functor has three parts:
  - an `import` part = what other modules it needs
  - an `export` part = the module interface
  - a `define` part = the module implementation including initialization code.

- Functors in the Mozart system are compilation units.
  - source code (i.e., human-readable text, `.oz`)
  - object code (i.e., compiled form, `.ozf`).
Standalone applications (1)

- It can be run without the interactive interface.
- It has a main functor, evaluated when the program starts.
- Imports the modules it needs, which causes other functors to be evaluated.
- Evaluating (or “installing”) a functor creates a new module:
  - The modules it needs are identified.
  - The initialization code is executed.
  - The module is loaded the first time it is needed during execution.
Standalone applications (2)

- This technique is called **dynamic linking**, as opposed to **static linking**, in which the modules are already loaded when execution starts.
- At any time, the set of currently installed modules is called the **module environment**.
- Any functor can be compiled to make a standalone program.
Functors. Example \((\text{GenericFunctor.oz})\)

functor
export generic:Generic
define

define
  fun \{\text{Generic Op InitVal N}\}
    if N == 0 then InitVal
    else \{\text{Op N \{Generic Op InitVal (N-1)\}}\}
  end
end
end

- The compiled functor \text{GenericFunctor.ozf} is created:
  - \text{ozc -c GenericFunctor.oz}
Functors (Standalone Application)

functor
import

  GenericFunctor
  Browser

define

  fun {Mul X Y} X*Y end
  fun {FactUsingGeneric N}
    {GenericFunctor.generic Mul 1 N}
  end

  {Browser.browse {FactUsingGeneric 5}}
end

- The executable functor GenericFact.exe is created:
  - `ozc -x GenericFact.oz`
Functors. Interactive Example

declare
[GF]={Module.link ['GenericFunctor.ozf']} fun {Add X Y} X+Y end fun {GenGaussSum N} {GF.generic Add 0 N} end {Browse {GenGaussSum 5}}

- **Function** Module.link is defined in the system module Module.

- It takes a list of functors, load them from the file system, links them together
  - (i.e., evaluates them together, so that each module sees its imported modules),
- and returns a corresponding list of modules.
Summary

- Type Notation
  - Constructing programs by following the type
- Haskell
- Design methodology
  - modules/functors
Reading suggestions

- From [van Roy, Haridi; 2004]
  - Chapter 3, Sections 3.2-3.4, 3.9
  - Exercises 2.9.8, 3.10.6-3.10.10
Future

- 12Oct : Declarative Concurrency
- 19Oct : Message Passing Concurrency
- 26Oct : Stateful Programming
- 2Nov  : Quiz 2 (1.5 hr and open book)
- 9Nov  : Relational Programming
- 16Nov : Revision