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
  - ⟨Value⟩, ⟨Record⟩,…
  - ⟨Record⟩ ⊂ ⟨Value⟩
    - The `Record` type is a subtype of the `Value` type
  - List is either `nil` or `X|Xr`

    where `Xr` is a list and `X` is an arbitrary value
  - ⟨List⟩ ::= `nil` | ⟨Value⟩'|'⟨List⟩

Polymorphic List

- Usually all elements of the same type
- Polymorphic list with elements of T type

  ⟨List T⟩ ::= `nil` | ⟨T⟩'|'⟨List T⟩
  - T is a type variable
  - ⟨List ?⟩ is a type constructor
  - ⟨List ⟨Int⟩⟩ : a list whose elements are integers
  - ⟨List ⟨Value⟩⟩ is equal to ⟨List⟩

Polymorphic Binary Tree

- Binary trees

  ⟨BTree T⟩ ::= `leaf` |
  `tree(key:`⟨Literal⟩ `value:`T
  `left:` ⟨BTree T⟩
  `right:` ⟨BTree T⟩)
  - 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 sub-trees, 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 … T_n are the types of its arguments can be represented by:

  ⟨proc {$ T_1 … T_n}⟩

  or

  {T_1 … T_n} → ()

On Types: procedures and functions

- The type of a function where T_1 … T_n are the types of the arguments, and T is the type of the result is:

  ⟨fun {$ T_1 … T_n}:T⟩

  or

  {T_1 … T_n} → T
- Append ::{⟨List⟩ ⟨List⟩}→⟨List⟩

  or precisely ::{⟨List A⟩ ⟨List A⟩}→⟨List A⟩

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= [[1 2]  4  nil  [[5] 10]]
{Length Xs} = 5
```

```
declare
Xs1=[[1 2] 4 nil]
{Browse Xs1}          → [[1 2] 4 nil]
Xs2=[[1 2] 4]|nil
{Browse Xs2}          → [[[1 2] 4]]
```

Constructing Programs from Type

- Nested lists type declaration
- ⟨NList T⟩ ::= `nil`
  | ⟨NList T⟩ '|' ⟨NList T⟩
  | T '|' ⟨NList T⟩ (T is not `nil` nor a cons)
- General structure:

```
case Xs
  of nil then ⟨expr1⟩ % base case
  [] X|Xr andthen {IsList X} then
     ⟨expr2⟩ % recursive calls for X and Xr
  [] X|Xr then
     ⟨expr3⟩ % recursive call for Xr
end
```


Constructing Programs from Type

- `Length ::` {⟨NList T⟩} → ⟨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

Preview

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 ⟨stack T⟩ whose elements are of any type T
- We define the following operations (with type definitions)

```
⟨fun {NewStack}: ⟨stack T⟩⟩
⟨fun {Push ⟨stack T⟩ ⟨T⟩ }: ⟨stack T⟩⟩
⟨proc {Pop ⟨stack T⟩ ?⟨T⟩ ?⟨stack T⟩}⟩
⟨fun {IsEmpty ⟨stack T⟩}: ⟨Bool⟩⟩
```


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)

```
declare Top S4
% ... either implementation
S1={NewStack}
S2={Push S1 2}
S3={Push S2 5}
{Pop S3 Top S4}
{Browse Top}        → 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

```
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
```

```
{Inc mr(a:3 b:2 c:1) b} → mr(a:3 b:3 c:1)
```


Example: Frequency Word Counting

```
local
   fun {Inc D X}
      if {DictMember D X} then
         {DictAdjoin D X
      else {DictAdjoin
      end
   end
in
   fun {Cnt Xs}
      % returns dictio
      {FoldL Xs Inc {MakeDict}}
   end
end
{Browse {Cnt [a b c a b a]}} → 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 → A
```

Universal type : ∀ A. A → A

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

Example

- Say we want to Peano-number ADT

```
Expr=(fun {MakeSucc N:Nat} {Succ N} end
     ,fun {MakeZero} 0:Nat end)
```

This implementation currently has type :

(Nat → Nat, Nat)

- Can make into existential type using:

pack Nat **as** N **in** Expr

which will now have a more abtract type :

∃ N. (N → 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

```
fact :: Integer -> Integer
fact 0          = 1
fact n | n>0    = n * fact (n-1)
```


Lazy Evaluation

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

```
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.

```
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.

```
add :: Integer -> Integer -> Integer
add x y = x+y
addT :: (Integer, Integer) -> Integer
addT(x,y) = x+y
```

Valid Expressions:

```
  (add 1 2)     =    addT(1,2)
  (add 1)       =    \ y -> 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!

```
sort :: (List a) -> (List a)
```

- Use type class `Ord a` instead.

```
sort :: Ord a => (List a) -> (List a)
```


Type Classes

- Class is characterized by a set of methods

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


Type Classes

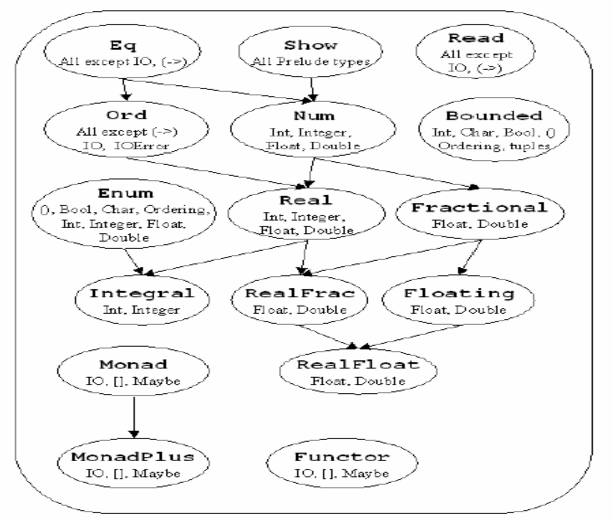
- Need to define instances of given class

```
instance Ord Int
 a>b  = a >Int b

instance Ord a => Ord [a]
 [] > ys       = False
 x:xs > []     = True
 x:xs > y:ys   = x>y or (x==y & xs>ys)
```

lexicographic ordering

Classes in Standard Library



Multi-Parameter Type Classes

- Can support generic type constructors

```
class Functor f where
  fmap :: (a → b) → f a → 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)

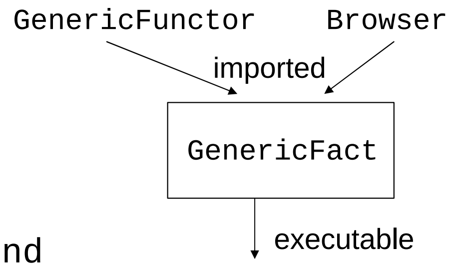
- This technique is called **dynamic linking**, as opposed to **static linking**, in which the modules are already loaded when execution starts.
- At any time, the set of currently installed modules is called the **module environment**.
- Any functor can be compiled to make a standalone program.

Functors. Example (`GenericFunctor.oz`)

```
functor
export generic:Generic
define
   fun {Generic Op InitVal N}
      if N == 0 then InitVal
      else {Op N {Generic Op InitVal (N-1)}}
      end
   end
end
```

- The compiled functor `GenericFunctor.ozf` is created:
  - `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