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:
  \[ e :: \text{type} \]

- 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
  - \(<\text{Value}>, <\text{Record}>, \ldots\>
  - \(<\text{Record}> \subset <\text{Value}>\)
    - The Record type is a subtype of the Value type
  - List is either \(\text{nil}\) or \(X|X_r\)
    where \(X_r\) is a list and \(X\) is an arbitrary value
  - \(<\text{List}> ::= \text{nil} \mid <\text{Value}>\mid '<\text{List}>\)

Polymorphic List

- Usually all elements of the same type
- Polymorphic list with elements of T type
  \(<\text{List T}> ::= \text{nil} \mid <T> \mid '<\text{List T}>\)
  - \(T\) is a type variable
  - \(<\text{List T}>\) is a type constructor
  - \(<\text{List }\langle\text{Int}\rangle>:\) a list whose elements are integers
  - \(<\text{List }\langle\text{Value}\rangle>:\) is equal to \(<\text{List}>\)

Polymorphic Binary Tree

- Binary trees
  \(<\text{BTree T}> ::= \text{leaf} \mid \text{tree}(\text{key: }<\text{Literal}> \mid \text{value: }T \mid \text{left: }<\text{BTree T}> \mid \text{right: }<\text{BTree T}>\)
  - Binary tree representing a dictionary mapping keys to values
  - Binary tree is:
    - either a \text{leaf} (atom leaf), or
    - an \text{internal node} with label \text{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 $T_1 \ldots T_n$ 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:

```haskell
case Xs of
    nil then (expr1) % base case
    [] X|Xr then (expr2) % recursive call
end
```

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”

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

```haskell
declare
Xs1=[[1 2] 4 nil]
{Browse Xs1} \rightarrow [[1 2] 4 nil]
Xs2=[[1 2] 4]|nil
{Browse Xs2} \rightarrow [[[1 2] 4]]
```
Nested lists type declaration

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

General structure:

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

Length :: \{\langle \text{NList } T \rangle \rightarrow \langle \text{Int} \rangle\}

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

\[
\text{fun } \{\text{IsList } L\} \\
L == \text{nil} \text{ orelse} \\
\{\text{Label } L\} == '|' \text{ andthen } \{\text{Width } L\} == 2 \\
\text{end}
\]

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 \( \text{stack } T \) 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)
  - \( \{ \text{IsEmpty } \{ \text{NewStack} \} \} = \text{true} \)
  - For any stack \( S \), \( \{ \text{IsEmpty } \{ \text{Push } S \} \} = \text{false} \)
  - For any \( E \) and \( S \), \( \{ \text{Pop } \{ \text{Push } S \ E \} \ E \ S \} \) holds
  - For any stack \( S \), \( \{ \text{Pop } \{ \text{NewStack} \} \ S \} \) raises error

stack (implementation I) using lists

\[
\begin{align*}
\text{fun} & \{ \text{NewStack} \} \ \text{nil} \ \text{end} \\
\text{fun} & \{ \text{Push } S \ E \} \ E \text{|}\ S \ \text{end} \\
\text{proc} & \{ \text{Pop } E \text{|}\ S \ ?E1 \ ?S1 \} \\
& \quad \ E1 = E \\
& \quad \ S1 = S \\
\text{end} \\
\text{fun} & \{ \text{IsEmpty } S \} \ S\text{==nil} \ \text{end}
\end{align*}
\]

stack (implementation II) using tuples

\[
\begin{align*}
\text{fun} & \{ \text{NewStack} \} \ \text{emptyStack} \ \text{end} \\
\text{fun} & \{ \text{Push } S \ E \} \ \text{stack}(E \ S) \ \text{end} \\
\text{proc} & \{ \text{Pop } \text{stack}(E \ S) \ E1 \ S1 \} \\
& \quad \ E1 = E \\
& \quad \ S1 = S \\
\text{end} \\
\text{fun} & \{ \text{IsEmpty } S \} \ S\text{==emptyStack} \ \text{end}
\end{align*}
\]
**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}  →  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#65521111...]

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

Example: Frequency Word Counting

local
  fun (Inc D X)
    if {DictMember D X} then
      {DictAdjoin D X}
    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 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
  
  \[
  \text{fun } \{\text{Id } X\} X \text{ end} \\
  \text{Id } :: A \to A \\
  \text{Universal type : } \forall A. A \to A
  \]

- ADT can be implemented using existential type.
  - \(\exists A. \text{type} \)
  - where \(A\) is considered to be hidden/abstracted

Example

- Say we want to Peano-number ADT
  
  \[
  \text{Expr } = \{\text{fun } \{\text{MakeSucc } N:Nat\} \{\text{Succ } N\} \text{ end} \\
  \text{,fun } \{\text{MakeZero}\} 0:Nat \text{ end}\}
  \]
  
  This implementation currently has type :
  
  \(\text{(Nat } \to \text{ Nat, Nat)}\)

- Can make into existential type using:
  
  \[
  \text{pack Nat as } N \text{ in Expr} \\
  \text{ which will now have a more abstract type :} \\
  \exists N. (N \to N, N)
  \]

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

\[
\text{fact} :: \text{Integer} \to \text{Integer} \\
\text{fact} \ 0 \quad = \ 1 \\
\text{fact} \ n \ | \ n > 0 \quad = \ n \times \text{fact} \ (n-1)
\]

Lazy Evaluation

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

\[
\text{andThen} :: \text{Bool} \to \text{Bool} \to \text{Bool} \\
\text{andThen} \ True \ x \ = \ x \\
\text{andThen} \ False \ x \ = \ False
\]

Type Declaration

- Data types have to be declared/enumerated.

\[
\text{data} \ \text{Bool} \ = \ True \ | \ False \\
\text{data} \ \text{ListInt} \ = \ Nil \ | \ \text{Cons} \ \text{Integer} \ \text{ListInt} \\
\text{type} \ \text{PairInt} \ = \ (\text{Integer}, \ \text{Integer})
\]

Polymorphic Types

- Generic types can be defined with type variables.

\[
\text{data} \ \text{BTree} a \ = \ \text{Empty} \\
\quad \quad \quad \ | \ \text{Node} \ a \ (\text{BTree} \ a) \ (\text{BTree} \ a) \\
\text{type} \ \text{BTreeInt} \ = \ \text{BTree} \ \text{Int} \\
\text{size} :: \text{BTree} a \to \text{Integer} \\
\text{size} \ \text{Empty} \quad = \ 0 \\
\text{size} \ (\text{Node} \ v \ l \ t) \quad = \ 1+\text{size} \ l+\text{size} \ t
\]
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:

\[
(\text{add} \ 1 \ 2) = \text{addT}(1,2) \\
(\text{add} \ 1) = \ \lambda \ y \rightarrow \text{addT}(1,y)
\]

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

\[
\text{class Eq a} \\
\text{==} :: a \rightarrow a \rightarrow \text{Bool} \\
\text{class Eq a} \Rightarrow \text{Ord} a \\
> , >= :: a \rightarrow a \rightarrow \text{Bool} \\
a>=b = (a>b) \text{ or } (a==b)
\]

- Need to define instances of given class

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

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

\text{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

- “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

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).

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.

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)

```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)

```oz
functor
  import
    GenericFunctor
    Browser
  define
    fun {Mul X Y} X*Y end
    fun {FactUsingGeneric N} {GenericFunctor.generic Mul 1 N} end
  end
end
```

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

Functors. Interactive Example

```oz
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}}
end
```

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