Overview

- Organization
- Course overview
- Introduction to programming concepts
Organization

Labs and Tutorials

- Do get Emacs and Mozart installed
  - Did you do tutorial 1
  - You must finish tutorial 2 before next lecture
  - Start with Assignment 1 now (compulsory)
  - Mozart is also available on tembusu (Linux cluster)
Lectures

- Lecture 02
  - More Introduction to programming concepts
  - Why Programming Model?
- Lecture 03 (next week)
  - The Declarative Programming Model I
  - (most of the week I’ll be away)
- Lecture 04 (after one week break)
  - The Declarative Programming Model II
  - Declarative Programming Techniques

Reading Suggestions

- As for Lecture 02
  - Browse as you like
  - Abstract and Preface [casual]
  - Introduction 1.1 – 1.15 [careful]
    [try examples]
  - The rest of Chapter 1 [read through]
  - Appendix A.1 for Oz Development Environment
  - Appendix B.1 and B.2 [as you need]
  - Look at the Oz base environment
Last Lecture

- Variable
  - variable declaration
  - store variables
  - assignment
- Data structures
  - numbers and atoms
  - tuples and record
- Functions
  - definition
  - call (application)
- Recursion

This Lecture

- More Data structures (compound data types)
  - tuples, lists, records
- More on variables
  - bound and unbound variables
  - partial values
  - dataflow variables and dataflow synchronization
- More on computing
  - pattern matching
- Why a computation model
  - procedures as opposed to functions
Data Structures

- Already seen:
  - number: integers 1, 2, ~1, 0
  - floating point (floats) 1.0, ~1.21

- This lecture: compound data structures
  - tuple combining several values
  - list special case of tuple
  - record generalization of tuple

Tuples
Tuples

- Combine several values (variables)
  - here: 1, a, 2
  - position is significant!
- Have a label
  - here: state

\[ x = \text{state}(1 \ a \ 2) \]

Tuple Operations

- \{Label \ x\} returns label of tuple \( x \)
  - here: state
  - is an atom
- \{width \ x\} returns the width (number of fields)
  - here: 3
  - is a positive integer
Feature Access
(Dot Access)

x=state(1 a 2)

- Fields are numbered from 1 to \{\text{width } x}\)
- \(x.N\) returns \(N\)-th field of tuple
  - here, \(x.1\) returns 1
  - here, \(x.3\) returns 2
- In \(x.N\), \(N\) is called feature

Tuples for Trees

\(x=m(Y Z)\)

- Trees can be constructed with tuples:
  declare
  \(Y=l(1 2)\) \(Z=r(3 4)\)
  \(X=m(Y Z)\)
Equality Operator (==)

- Testing equality with an atom or number
  - simple, must be the same number or atom
  - okay to use
  - we will see pattern matching as something much nicer in many cases

- Testing equality among trees
  - not so straightforward
  - don’t do it, we don’t need it (yet)

Summary: Tuples

- Tuple
  - label
  - width
  - field
  - feature
Records

- Records are generalizations of tuples
  - features can be atoms
  - features can be arbitrary integers
    - not restricted to start with 1
    - not restricted to be consecutive

- Records also have label and width
- Needed for assignment 01, will be discussed again
Records

\[ x = \text{state}(a:1\ b:2) \]

- Position is insignificant
- Field access is as with tuples
  \[ x.a \text{ is } 1 \]

Tuples are Records

- Constructing
  ```
  declare
  x = \text{state}(1:a\ 2:b\ 3:c)
  ```
  is equivalent to
  ```
  x = \text{state}(a\ b\ c)
  ```
Partial Values

Trees With Variables

- Z can be assigned later
- Declare Z without assigning:

\[
\text{declare} \\
Y = l(1 2) \ Z \ X = m(Y \ Z)
\]
Partial Values

- Assigning a value to a variable
  - we also say: binding a variable
  - unbound variable: no value assigned yet
  - bound variable: value already assigned

- Values can be described partially
  - data structure can contain unbound variables
  - often called partial values

Unbound Variables

- In Java, when declaring a variable it is initialized
- In C/C++, when declaring a variable it refers to a memory location containing garbage
- Here, the variable is left unassigned
  - can be used before bound!

- Unbound variables and…
  - ... computing?
  - ... constructing data structures?
Unbound Variables
Computing With Them

● Option A: don’t care
  ● undefined behavior, do whatever implementation likes
  ● unpredictable

● Option B: error on access
  ● a little better, because more predictable

● Option C: utilize expressive power!
  ● of course, that is what we do!

Computing With Partial Values

● Example
  \[ X = Y + Z \]
  where \( Y \) and \( Z \) are still unbound

● Computation will stop automatically
  ● we say: computation suspends

● Computation resumes, if both \( Y \) and \( Z \) become bound
  ● we say: computation resumes

● Also
  ● automatic synchronization
  ● dataflow behavior
  ● variables also called dataflow variables
Why Dataflow Variables

- Assume multiple *concurrent* computations
  - communicate with dataflow variables
  - one can be producer: bind variable
  - one can be consumer: access variable
- Consumer automatically synchronizes on producer
  - data flows from producer to consumer
- Will be discussed in detail

Concurrent Programming Model

Remark: Variables

- Two properties
  - single assignment: can be assigned at most once
  - dataflow: automatic synchronization
- Deep connection between them
  - regardless of when a suspended computation resumes, the values available will be the same!
- This makes concurrent programming simple
Programming Errors

- Bad news
  - we only make use of dataflow later
  - you’ll make errors by having computations that just suspend already now (bastard: ==)
  - most likely: forget to bind a variable

- Good news
  - you don’t have to restart Oz
  - bind the variable later (the interactive session can do that)
  - just correct the bug and retry (can do that as well)

Unbound Variables

Constructing Data Structures

- Leave unbound variables to be filled later
- Common techniques
  - construct skeleton of data structure, fill in details later
  - split computations into parts that do this
- Questions to be answered
  - can variables be bound to variables? Yes!
Skeleton Construction

- Skeleton construction
  
  \[ \text{declare} \]
  
  \[ A \ B \ C \ X = a(A \ B \ C) \]

Fill in Values

- By binding variables
  
  \[ B = b \]
Fill in Values with “dot”

- By binding variables via “dot” access: $X, 3 = c$
- Dot returns the (store) variable

Constructing Tuple Skeletons

- \{\texttt{MakeTuple} \textit{Label} \textit{Width}\}
  - creates new tuple with label \textit{Label} and width \textit{Width}
  - fields are initialized to variables
- Access to fields then by “dot”
Example Tuple Construction

- Created by execution of
  
  `declare`
  
  `X = {MakeTuple a 3}`

Example Tuple Construction

- After execution of
  
  `X.2 = b`
  
  `X.3 = c`
Variable-Variable Binding

- Two store variables after executing declare $X \ Y$

Variable-Variable Binding

- Two store variables after executing declare $X \ Y$
- Do variable-variable binding $X=Y$
Variable-Variable Binding

- Binding variables together makes them equal
  - binding a variable (or another variables) affects all variables that are bound together
- In our previous example:
  - Executing $X=a$ also binds $Y$ to $a$

```
 X  a  Y
```

- Now links are not longer needed

```
 X  a  Y
```

Constructing Graphs

- Example
  
  ```
  declare
  Y Z X = a(Y Z)
  ```

- Now bind Z to X
  
  ```
  Z = X
  ```

- Possible due to deferred assignment
  - we won’t make use of it right now
Remark: Other Cases

- Suppose in $X=Y$ both $X$ and $Y$ are bound

- Case one: no variables involved
  - test whether data structures are the “same”
  - is already involved due to graphs

- Case two: $X$ or $Y$ refer to partial values
  - occurring variables are bound to make $X$ and $Y$ the “same”
  - this process is called unification

```
delclare
X=f(a b) Y=f(a b)
X=Y % passes silently
```

```
delclare
X=f(a b) Y=f(a c)
X=Y % raises an error
```
Remark: Other Cases

- Suppose in $X=Y$ both $X$ and $Y$ are bound
- Case one: no variables involved
  - test whether data structures are the “same”
  - is already involved due to graphs
- declare
  - $X=f(a\ X)\ Y=f(a\ f(a\ Y))$
  - $X=Y\ %\ this\ is\ fine$
- $X=Y=f(a\ f(a\ f(a\ \ldots)))\ %\ ad\ infinitum$

Remark: Other Cases

- Suppose in $X=Y$ both $X$ and $Y$ are bound
- Case two: $X$ or $Y$ refer to partial values
  - occurring variables are bound to make $X$ and $Y$ the “same”
  - this process is called *unification*
- declare
  - $U\ Z\ X=f(a\ U)\ Y=f(Z\ b)$
- $U$ is bound to $b$, $Z$ is bound to $a$
Summary: Partial Values

- Bound and unbound variables
- Variable-variable binding
- Skeleton construction
- Automatic synchronization
- Variable-variable equality (unification)
Lists

- A list contains a sequence of elements
- A list
  - is the empty list, or
  - consists of a cons (or list pair) with head and tail
    - head contains an element
    - tail contains a list

Encoding Lists with Tuples

- Lists are encoded with atoms and tuples
  - empty list: the atom nil
  - cons: tuple of width 2 with label ‘|’

- Special syntax for cons
  \[ X = Y | Z \]
  instead of
  \[ X = ‘|’ (Y Z) \]
  But of course: both are equivalent
An Example List

- After execution of declare
  \[ X_1 = a | X_2 \quad X_2 = b | X_3 \quad X_3 = c | \text{nil} \]

Simple List Construction

- One can also write
  \[ X_1 = a | b | c | \text{nil} \]
  which abbreviates
  \[ X_1 = a | (b | (c | \text{nil})\) ) \]
  which abbreviates
  \[ X_1 = ' | (a ' | (b ' | (c nil))) \]
- Even shorter
  \[ X_1 = [a \ b \ c] \]
Computing With Lists

- Remember: a cons is a tuple!
- Access head of cons
  \[ X.1 \]
- Access tail of cons
  \[ X.2 \]
- Test whether list \( X \) is empty:
  
  \[
  \text{if } X = \text{nil} \text{ then } ... \text{ else } ... \text{ end}
  \]

Head And Tail

- Define abstractions for lists
  
  \[
  \text{fun } \{ \text{Head} \ \text{xs} \}
  \begin{align*}
  \text{xs.1} \\
  \text{end}
  \end{align*}
  \]

  \[
  \text{fun } \{ \text{Tail} \ \text{xs} \}
  \begin{align*}
  \text{xs.2} \\
  \text{end}
  \end{align*}
  \]
Example of Head and Tail

- \{\text{Head \ [a \ b \ c]}\}
  returns a
- \{\text{Tail \ [a \ b \ c]}\}
  returns \[b \ c]\n- \{\text{Head \ {Tail \ {Tail \ [a \ b \ c]}}}\}\}
  returns c

- Draw the trees!

How to Process Lists

- Given: list of integers
- Wanted: sum of its elements
  - implement function \text{Sum}

- Inductive definition over list structure
  - Sum of empty list is 0
  - Sum of non-empty list \text{L} is
    \{\text{Head \ L} + \{\text{Sum \ {Tail \ L}}\}\}
Sum of a List

fun \{Sum L\} 
  \text{if } L==\text{nil} \text{ then } 
  \text{0} 
  \text{else} 
  \text{\{Head L\} + \{Sum \{Tail L\}\}} 
  \text{end} 
\text{end} 

General Method

- Lists are processed recursively
  - base case: list is empty (nil)
  - inductive case: list is cons
    access head, access tail

- Powerful and convenient technique
  - \textit{pattern matching}
  - matches patterns of values and provides access to fields of compound data structures
Sum with Pattern Matching

fun {Sum L}
    case L
        of nil then 0
        [] H|T then H+{Sum T}
    end
end

Clause

• nil is the pattern of the clause
Sum with Pattern Matching

fun {Sum L}
  case L
  of nil then 0
  [H|T then H+{Sum T}
  end
end

- H | T is the pattern of the clause

Pattern Matching

- The first clause uses of, all other []
- Clauses are tried in textual order
- A clause matches, if its pattern matches
- A pattern matches, if the width, label and features agree
  - then, the variables in the pattern are assigned to the respective fields
- Case-statement executes with first matching clause
Length of a List

- Inductive definition
  - length of empty list is 0
  - length of cons is 1 + length of tail

```haskell
fun {Length Xs}
  case Xs
  of nil then 0
  [] X|Xr then 1+{Length Xr}
  end
  end
```

General Pattern Matching

- Pattern matching can be used not only for lists!
- Any value, including numbers, atoms, tuples, records
- Will be practiced in tutorial 2
Lists: Summary

- List is either empty or cons with head and tail
- List processing is recursive processing
- Useful for this is pattern matching

Functions over lists

- Compute the function \( \{\text{Pascal N}\} \)
- Takes an integer \( N \), and returns the \( N \)th row of a Pascal triangle as a list

\[
\begin{array}{c}
1 \\
1 1 \\
1 2 1 \\
1 3 3 1 \\
1 4 6 4 1 \\
\end{array}
\]
Functions over lists

- Compute the function \( \{ \text{Pascal } N \} \)
- 1. For row 1, the result is [1]
- 2. For row N, shift to left row N-1 and shift to the right row N-1
- 3. Align and add the shifted rows element-wise to get row N

\[
\begin{array}{cccc}
1 & 1 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccc}
(0) & 1 & 3 & 3 & 1
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccc}
1 & 4 & 6 & 4 & 1
\end{array}
\]

Shift right [0 1 3 3 1]

Shift left [1 3 3 1 0]
Functions over lists (2)

\[
\text{declare}
\text{fun } \{\text{Pascal } N\}
\begin{align*}
\text{if } N = 1 & \text{ then } [1] \\
\text{else} & \\
\{\text{AddList} \\
\{\text{ShiftLeft } \{\text{Pascal } N-1\}\} \\
\{\text{ShiftRight } \{\text{Pascal } N-1\}\}\} \\
\text{end} \\
\text{end}
\end{align*}
\]

Functions over lists (3)

\[
\text{fun } \{\text{ShiftLeft } L\}
\begin{align*}
\text{case } L \text{ of } H|T & \text{ then } \\
H & \{\text{ShiftLeft } T\} \\
\text{else} & [0] \\
\text{end} \\
\text{end}
\end{align*}
\]

\[
\text{fun } \{\text{ShiftRight } L\} \ 0|L \text{ end}
\]

2003-08-22 S. Haridi, CS2104, Lecture 03 (slides: C. Schulte, S. Haridi)
Functions over lists (3)

fun {AddList L1 L2}
  case L1 of H1|T1 then
      case L2 of H2|T2 then
        H1+H2|{AddList T1 T2}
      end
    else
        nil
    end
end

Top-down program development

- Understand how to solve the problem by hand
- Try to solve the task by decomposing it to simpler tasks
- Devise the main function (main task) in terms of suitable auxiliary functions (subtasks) that simplifies the solution (ShiftLeft, ShiftRight and AddList)
- Complete the solution by writing the auxiliary functions
Is your program correct?

- "A program is correct when it does what we would like it to do"
- In general we need to reason about the program:
  - **Semantics for the language**: a precise model of the operations of the programming language
  - **Program specification**: a definition of the output in terms of the input (usually a mathematical function or relation)
  - Use mathematical techniques to reason about the program, using programming language semantics

Mathematical induction

- Select one or more input to the function
- Show the program is correct for the *simple cases* (base case)
- Show that if the program is correct for a *given case*, it is then correct for the *next case*.
- For integers base case is either 0 or 1, and for any integer n the next case is n+1
- For lists the base case is nil, or a list with one or few elements, and for any list T the next case H|T
Correctness of factorial

fun \{Fact N\}
  if \(N=0\) then 1 else \(N \times \{Fact\ N-1\}\) end
end

- Base Case: \{Fact 0\} returns 1
- \((N>1), \ N \times \{Fact\ N-1\}\) assume \{Fact N-1\} is correct, from the spec we see the \{Fact N\} is \(N \times \{Fact\ N-1\}\)
- More techniques to come!

Complexity

- Pascal runs very slow, try \{Pascal 24\}
- \{Pascal 20\} calls: \{Pascal 19\} twice, \{Pascal 18\} four times, \{Pascal 17\} eight times, ..., \{Pascal 1\} \(2^{19}\) times
- Execution time of a program up to a constant factor is called program’s time complexity.
- Time complexity of \{Pascal N\} is proportional to \(2^N\) (exponential)
- Programs with exponential time complexity are impractical
Faster Pascal

- Introduce a local variable L
- Compute \( \text{FastPascal N-1} \) only once
- Try with 30 rows.
- \text{FastPascal} is called N times, each time a list on the average of size N/2 is processed
- The time complexity is proportional to \( N^2 \) (polynomial)
- Low order polynomial programs are practical.

```pascal
fun \{\text{FastPascal N}\}
if N==1 then [1]
else
  local L in
  L=\{\text{FastPascal N-1}\}
  \{\text{AddList} \{\text{ShiftLeft} L\} \{\text{ShiftRight} L\}\}
end
end
```

Higher-order programming

- Assume we want to write another Pascal function which instead of adding numbers performs exclusive-or on them
- It calculates for each number whether it is odd or even (parity)
- Either write a new function each time we need a new operation, or write one generic function that takes an operation (another function) as argument
- The ability to pass functions as argument, or return a function as result is called higher-order programming
- Higher-order programming is an aid to build generic abstractions
Variations of Pascal

- Compute the parity Pascal triangle
  ```
  fun {Xor X Y} if X==Y then 0 else 1 end end
  ```

  
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Higher-order programming

```java
fun {GenericPascal Op N}
  if N==1 then [1]
  else L in L = {GenericPascal Op N-1}
    {OpList Op {ShiftLeft L} {ShiftRight L}}
  end
end
```
Higher-order programming

fun {OpList Op L1 L2}
  case L1 of H1|T1 then
    case L2 of H2|T2 then
      {Op H1 H2}|{OpList Op T1 T2}
    end
  else nil end
end

fun {Add N1 N2} N1+N2 end
fun {Xor N1 N2}
  if N1==N2 then 0 else 1 end
end

fun {Pascal N} {GenericPascal Add N} end
fun {ParityPascal N}
  {GenericPascal Xor N}
end
This Lecture

- Data structures
  - tuples, lists, records
- More on variables
  - bound and unbound variables
  - partial values
  - dataflow variables and dataflow synchronization
- More on computing
  - pattern matching
- Higher order programs (functions)
- Why a computation model
  - procedures as opposed to functions

Towards the Model

This is the outlook section
Confusion

- By now you should feel uneasy and slightly embarrassed (maybe even confused)
- We haven’t explained how computation actually proceeds
- No, you are fine? Wait and see…

Another Length

fun {L xs N}
  case xs
    of nil then N
    [] X|Xr then {L Xr N+1}
  end
end
fun {Length xs}
  {L xs 0}
end
Comparison

- This length is six-times faster then our first one!
  - hey, it has one argument more!
  - so what
  - what could be the difference
  - and what is more: it takes considerable less memory!
  - actually, it runs in constant memory!

- Our model will answer
  - intuition: even though recursive it executes like a loop

There Is No Free Lunch!

- Before we can answer the questions we have to make the language small
  - sort out what is primitive: kernel language
  - what can be expressed

- Kernel language
  - based on procedures
  - no functions
What Is a Procedure?

- It does not return a value
  - Java: methods with `void` as return type

- But how to return a value anyway?
  - Idea: use an unbound variable
  - Why: we can supply value later (before return)
  - Aha: so that’s why we have been dwelling on this!

Our First Procedure: Sum

```plaintext
proc {Sum Xs N}
  case Xs
  of nil then N=0
  [] X|Xr then N=X+{Sum Xr}
  end
end
```

- Hey, we call `Sum` as if it was a function
  - that’s okay. It is just syntax
  - we’ll sort that out next week
Being More Primitive

```plaintext
proc {Sum Xs N}
    case Xs
    of nil then N=0
    [] X|Xr then
        local M in {Sum Xr M} N=X+M end
    end
end
```

- Local declaration of variables
- Needed to fully base kernel language on procedures

Summary

- Tuples and records
- Variables
- Lists
- Pattern matching
- Towards the model
Have a Nice Weekend!