Announcements

- Exam 2 graded
  - Rainbow grades: HW1-5, Exam 1-2, Quiz 1-6
- Part 1: Problem 1, Problem 2, will go over 2
- Part 2: Problem 1, Problem 2
- Part 3: Problem 1, Problem 2, Problem 3, will go over 2 and 3
- Part 4: Problem 1, Problem 2, will go over 1 later in lecture
- Part 5

Today's Lecture Outline

- From pure lambda calculus to a functional programming language
  - Applied lambda calculus
  - Typed lambda calculus
- Types
- Type systems
  - Type checking
  - Type safety
- Type equivalence

From Pure Lambda Calculus to a Programming Language¹

- Applied Lambda Calculus
- Typed Lambda Calculus

¹ This material will only appear on extra credit HWs 😊

Pure Lambda Calculus

- M → x | (λx. M₁) | (M₁ M₂)

- Amazingly, we can define boolean logic (true and false constants, and, or and not functions)
  - Church booleans
- Amazingly, we can define arithmetic (numerals, predecessor, successor, plus, times functions, etc.) just in terms of the pure lambda calculus!
  - Church numerals
- We can define control flow (if-then-else and recursion) in terms of the pure lambda calculus!
Applied Lambda Calculus (from Sethi)

- \( M \rightarrow c \mid x \mid (\lambda x. M) \mid (M_1 M_2) \)

Augments the pure lambda calculus with constants. Each applied lambda calculus defines its set of constants and reduction rules. For example:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>if, true, false</td>
<td>if true M N ( \Rightarrow \delta M )</td>
</tr>
<tr>
<td>if false M N ( \Rightarrow \delta N )</td>
<td></td>
</tr>
<tr>
<td>0, iszero, pred, succ</td>
<td>iszero 0 ( \Rightarrow \delta ) true</td>
</tr>
<tr>
<td>iszero (succ 0) ( \Rightarrow \delta ) false, k&gt;0</td>
<td></td>
</tr>
<tr>
<td>iszero (pred 0) ( \Rightarrow \delta ) false, k&gt;0</td>
<td></td>
</tr>
<tr>
<td>succ (pred M) ( \Rightarrow \delta M )</td>
<td></td>
</tr>
<tr>
<td>pred (succ M) ( \Rightarrow \delta M )</td>
<td></td>
</tr>
</tbody>
</table>

A Functional Language

<table>
<thead>
<tr>
<th>Construct</th>
<th>Applied ( \lambda )-Calculus</th>
<th>Our Language (ML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>Constant</td>
<td>( c )</td>
<td>( c )</td>
</tr>
<tr>
<td>Application</td>
<td>( M N )</td>
<td>( M N )</td>
</tr>
<tr>
<td>Abstraction</td>
<td>( \lambda x. M )</td>
<td>( \text{fun } x \Rightarrow M )</td>
</tr>
<tr>
<td>Integer</td>
<td>( \text{succ}#0, k&gt;0 ) k</td>
<td>( \text{pred}#0, k&gt;0 -k )</td>
</tr>
<tr>
<td>Conditional</td>
<td>if P M N</td>
<td>if P then M else N</td>
</tr>
<tr>
<td>Let</td>
<td>( (\lambda x. M) N )</td>
<td>let val ( x = N ) in M end</td>
</tr>
</tbody>
</table>

Example

- Applied lambda calculus expression 
  \( (\lambda x. \text{if } x \text{ true } y) \) false

  - In our language (ML), it becomes:
    \[ \text{let val } x = \text{false in if } x \text{ then true else } y \text{ end} \]

  - In Scheme, it becomes:
    \[ \text{(let ((x #f)) (if x #t y)) or} \]
    \[ \text{((lambda (x) (if x #t y)) #f)} \]

The Fixed-Point Combinator

- One more constant, and one more rule:
  \[ \text{fix } M \Rightarrow \delta M (\text{fix } M) \]

  - Needed to define recursive functions:
    \[ \text{plus } x y = \begin{cases} y & \text{if } x = 0 \\ \text{plus} (\text{pred } x) (\text{succ } y) & \text{otherwise} \end{cases} \]

  - Therefore, we must have:
    \[ \text{plus} = \lambda x y. \text{if } (\text{iszero } x) y (\text{plus} (\text{pred } x) (\text{succ } y)) \]

The Fixed-Point Combinator

- But how do we define plus?

  Define plus = fix M, where
  \[ M = \lambda f. \lambda x. \text{if } (\text{iszero } x) y (f (\text{pred } x) (\text{succ } y)) \]

  We must show that
  \[ \text{fix } M = \delta M \]
  \[ \lambda x y. \text{if } (\text{iszero } x) y (\text{fix } M (\text{pred } x) (\text{succ } y)) \]

Exercise: define factorial = ?
The Y Combinator

- \text{fix} is a pure lambda expression!

\[ Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

This is the famous Y-combinator

Show that \( Y M \) beta-reduces to \( M (Y M) \)

Think of \text{fix} (e.g., \( Y \)) as the function that takes a function \( M \) and returns \( M(M(M(\ldots))) \)

**Typed Lambda Calculus**

- Constants add power
- But they raise problems because they permit erroneous expressions such as
  - if \((\lambda x. x) y z\) - function values not permitted as predicate, only \text{true}/\text{false}
  - \((0 \times)\) - 0 is not a function
  - \text{succ} \text{true} - undefined in our language
  - plus \text{true} \((\lambda x. x) x\) etc.

**Type Expressions**

- Introducing type expressions
  - \( \tau ::= b \mid \tau \rightarrow \tau \)
  - A type is a basic type (bool, int) or a function type
- Examples
  - int
  - bool
  - int \(\rightarrow\) (int \(\rightarrow\) int)

- Syntax of typed lambda calculus:
  - \( M \rightarrow x \mid (\lambda x : \tau. M_1) \mid (M_1 M_2) \)

**Type Judgments and Environment**

- An expression in the typed lambda calculus is
  - Type correct, or
  - Type incorrect
- The rules of type correctness are given in the form of type judgments in an environment
  - Environment \( \text{env} \vdash M : \tau \)
  - Read: environment \( \text{env} \) entails that \( M \) has type \( \tau \)
  - Type judgment

**Example**

- Deduce the type for \( \lambda x : \text{int}. \lambda y : \text{bool}. x \) in the nil environment
Types

Read: Scott, Chapter 7.1 – 7.4

What is a type?
- A set of values and the valid operations on those values
  - Integers: + - * / < <= == > ... 
  - Arrays: lookUp(array, index), assign(array, index, value), initialize(array), setBounds(array)
  - User-defined types: Java interfaces

What is the role of types?
- What is the role of types in programming languages?
  - Semantic correctness
  - Data abstraction
    - ADTs
  - Documentation (static types only)

3 Views of Types

Denotational (or set) point of view:
- A type is simply a set of values. A value has a given type if it belongs to the set. E.g.
  - int = {1, 2, ...}
  - char = {'a', 'b', ...}
  - bool = {true, false}

Abstraction-based point of view:
- A type is an interface consisting of a set of operations with well-defined meaning

Constructive point of view:
- Primitive/simple/built-in types: e.g., int, char, bool
- Composite/constructed types:
  - Constructed by applying type constructors
    - pointer e.g., pointerTo(int)
    - array e.g., arrayOf(char) or arrayOf(char, 20) or ...
    - record/struct e.g., record(age: int, name: string)
  - union e.g., union(rest, pointerTo(char))
  - list e.g., list(...) 
  - function e.g., float → int
  - CAN BE NESTED! pointerTo(arrayOf(pointerTo(char)))

For most of us, types are a mixture of these 3 views

What is a Type System?
- A mechanism to define types and associate them with programming language constructs

- Additional rules for type equivalence, type compatibility
  - Important from pragmatic point of view
What is Type Checking?
- The process of ensuring that the program obeys the type rules of the language

Type checking can be done statically
- At compile-time, i.e., before execution
  - Statically typed (or statically checked) language

Type checking can be done dynamically
- At runtime, i.e., during execution
  - Dynamically typed (or dynamically checked) language

Type safety
- Textbook defines term prohibited application (also known as forbidden error): intuitively, a prohibited application is an application of an operation on values of the wrong type
- Type safety is the property that no operation ever applies to values of the wrong type at runtime, i.e., no prohibited application (forbidden error) ever occurs

Forbidden Errors
- Example: indexing an array out of bounds
  - \( a[i] \), \( a \) is of size \( \text{Bound} \), \( i < 0 \) or \( \text{Bound} \leq i \)
  - In C, C++, this is not a forbidden error
    - \( \text{Bound} \) and \( i \) is not checked (bounds are not part of type)
    - What are the tradeoffs here?
  - In Pascal, this is a forbidden error. Prevented with static checks
    - \( \text{Bound} \) and \( i \) must be checked at compile time
    - What are the tradeoffs here?
  - In Java, this is a forbidden error. It is prevented with dynamic checks
    - \( \text{Bound} \) and \( i \) must be checked at runtime
    - What are the tradeoffs here?

Language Design Choices
- Design choice: what is the set of forbidden errors?
  - Obviously, we cannot forbid all possible semantic errors...
  - Define a set of forbidden errors
- Design choice: Once we've chosen the set of forbidden errors, how does the type system prevent them?
  - Static checks only? Dynamic checks only? A combination of both?
  - Furthermore, are we going to absolutely disallow forbidden errors (be type safe), or are we going to allow for programs to circumvent the system and exhibit forbidden errors (i.e., be type unsafe)?

Type Safety
- Java vs. C++:
  - Java: Duck q; ...; q.quack() class Duck has quack
  - C++: Duck *q; ...; *q->quack() class Duck has quack

  Can we write code that calls `quack()` on an object that isn't a `Duck`?
  - In Java?
  - In C++?

Java is said to be type safe while C++ is said to be type unsafe
C++ is type unsafe

```cpp
#include <iostream>
using namespace std;

int main()
{
    // Example 1
    void* x = (void*) new A;
    B* q = (B*) x; // a safe downcast?
    int case1 = q->foo(); // what happens?

    // Example 2
    void* x = (void*) new A;
    B* q = (B*) x; // a safe downcast?
    int case2 = q->foo(66); // what happens?

    return 0;
}
```

What is Type Checking?

- Static typing vs. dynamic typing
  - What are the advantages of static typing?
  - What are the advantages of dynamic typing?

One more thing…

- What is strong typing?
  - One often hears “Java is strongly typed” while “C++ is weakly typed”...
  - The term is often used in nonsensical way
    - “The language has a type checker”
    - “The language is sound”
    - To most it seems to mean: “A language like C or Java related in a way I can’t make quite precise”
- Correct (I think): the language has some element of static typing and is type safe

Lecture Outline

- Types
- Type systems
  - Type checking
  - Type safety
- Type equivalence