Polymorphism

Gimme, gimme yer homework!

What is it?

Ad-hoc (OOP) and Parametric (Functional)

Higher-order and anonymous functions

Unification of Horn clauses and types
Logistics

Hand in your homework

Last week’s graded homework is available at the end of class

Reminder: Instructor’s office hours are TW 5-6 in WWH 401
What Is Polymorphism?

Literally, “many shapes”

It’s all Greek to me....

Code that can be applied to different types

Type: set of values

A class is a type

Ad-hoc: Defined from the base type

Start from type and work \textit{out} to usage

Parametric: Proscribed by how the type is used

Start from usage and work \textit{in} to type
Ad-hoc (OOP)

Define a type (a class)

Methods define minimum requirements

Capabilities

Subclasses must meet those capabilities

Adding new capabilities further restricts the type (i.e., subsets of the set of values represented by supertype)

Others use the type but don’t know subtype

Ad-hoc: New subclasses mean new code
Example of Ad-Hoc Polymorphism

Sending message with run-time method look-up

Self, Smalltalk, Objective-C, CLOS

Aka virtual functions

Java, C++

Tell every Shape in a list to Draw; Each Draws itself differently
Parametric (Functional)

Instead of explicitly defining polymorphic type, define polymorphic usage of (implicit) type

In Haskell, add anything that can ”+” (numbers, strings, etc)
add \( a \ b = a + b \)

Sort anything that sort returns that can merge
sort \( a \) = let front = firsthalf \( a \)
back = secondhalf \( a \) in
merge (sort \( a \)) (sort \( b \))

Any value that can perform capabilities defined by usage can be used

How to determine what values can be used?
Parametric Polymorphism with Dynamic Typing

Aka “Untyped”

Any value passed around; runtime checks validity of operations

Example languages:

- Lisp, Scheme
- Perl, many scripting languages
- Self, Smalltalk

Exception/Error if operation unsupported by value
Parametric Polymorphism and Static Typing

Declaration of type variables

- Possibly recompile polymorphic code for new types
- C++, CLU, Eiffel
- Haskell when types are explicitly declared
- Cf. interfaces in Java

Type inferencing

- Compiler determines most general type from usage
- ML, Haskell
Example: Static Typing and C++

Add

template <class A, class B, class C>
C add(A a, B b) { return a+b; }

Merge sort

template <class A>
List<A> mergesort (List<A> a) {
    List<A> front = firsthalf(a);
    List<A> back = secondhalf(a);
    return
        merge (mergesort(front, mergesort(back));
}
Mixing Ad-Hoc and Parametric

Haskell allows definition of a “class”

Data type can be declared as an “instance” of class

Class serves same “start inside, work out” purpose as OOP class

Example

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimal complete definition: (==) or (/=)
  x == y     = not (x/=y)
  x /= y     = not (x==y)
```
Example: Type Extension

\[
\text{class (Eq a) => Ord a where}
\]
\[
\begin{align*}
\text{compare} & : : a \rightarrow a \rightarrow \text{Ordering} \\
(\langle\rangle, (\langle\le\rangle), (\langle\ge\rangle), (\langle\rangle)) & : : a \rightarrow a \rightarrow \text{Bool} \\
\text{max, min} & : : a \rightarrow a \rightarrow a
\end{align*}
\]

\[
\text{-- Minimal complete definition: (\langle\le\rangle) or compare}
\]
\[
\text{compare } x \ y \ | \ x\equiv y \ = \text{ EQ} \\
| x\le y \ = \text{ LT} \\
| \text{otherwise} \ = \text{ GT}
\]
\[
\begin{align*}
\text{x }\leq \text{ y} & = \text{ compare x y } /= \text{ GT} \\
\text{x }< \text{ y} & = \text{ compare x y } == \text{ LT} \\
\text{x }\geq \text{ y} & = \text{ compare x y } /= \text{ LT} \\
\text{x }> \text{ y} & = \text{ compare x y } == \text{ GT}
\end{align*}
\]
Example (cont). and Instance

\[
\begin{align*}
\text{max } x \ y & \quad | \quad x \geq y \quad = \quad x \\
& \quad | \quad \text{otherwise} \quad = \quad y \\
\text{min } x \ y & \quad | \quad x \leq y \quad = \quad x \\
& \quad | \quad \text{otherwise} \quad = \quad y \\
\end{align*}
\]

\[
\text{data } \text{Bool} \quad = \quad \text{False} \mid \text{True} \\
\text{deriving} \quad (\text{Eq}, \quad \text{Ord}, \quad \text{Ix}, \quad \text{Enum}, \quad \text{Read}, \quad \text{Show}, \quad \text{Bounded})
\]

\[
(\&\&), \quad (\mid \mid) \quad :: \quad \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{False} \quad \&\& \quad x \quad = \quad \text{False} \\
\text{True} \quad \&\& \quad x \quad = \quad x \\
\text{False} \quad \mid \mid \quad x \quad = \quad x \\
\text{True} \quad \mid \mid \quad x \quad = \quad \text{True}
\]

Recap: Polymorphism

One code base, many interpretations

Code reuse

Can be loose and intuitive — Dynamic typing
  Faster prototyping
  Used often for scripting

Can be precise and rigorous — Static typing
  More provably correct code
  Better for solid engineering
Break time!

Come back in 5 minutes....
Higher-Order Functions

Recall: Not Data-Driven vs Control-Driven

Dead issue: Real algorithms, real systems are both
May apply to a single mechanism, but not language as whole
Some languages have little or no distinction

If functions return data, but there is no distinction between data and control....

Functions can return functions
Function Pointers: Not What We Mean

C, C++, etc. allow for *function pointers*

C++ syntactic extension to member function pointers

Effectively, address of where to continue execution

Must point to pre-existing function

Pointing to something new would require:

Generate code and write to disk

Compile

Link in, all at run-time
Consider Lambda Abstraction

Unnamed or anonymous function

Given Greek letter \( \lambda \), resulting in “lambda calculus”, due to Church

Parameter, followed by an expression

Scheme
(lambda (x) x + x)

Haskell
\( \lambda x \rightarrow x + x \)

Give it a name to use it as a regular function

double = \( \lambda x \rightarrow x + x \)
double2 x = x + x
Functions as Values

Evaluating a lambda abstraction replaces free variable with another lambda expression

Formally,
\[ \text{exp : name | abstraction | application | parenthesized exp} \]

Therefore lambda abstraction can contain a lambda expression

Evaluating (or applying) abstraction replaces variable with another expression

I.e., abstraction is applied to argument, argument replaces variable in lambda abstraction
Returning a Lambda Abstraction

\( \text{example} = \lambda x \rightarrow \lambda y \rightarrow y \times x \)

\( \text{user} > \text{example} \; 665 \)

\( \text{example} \; 665 = \lambda x \rightarrow 665 \times x \)

\( \lambda x \rightarrow 665 \times x \)

\( \text{map} \; x \; \text{alist} = x \; (\text{head} \; \text{alist}) : \text{map} \; x \; (\text{tail} \; \text{alist}) \)

\( \text{map} \; _{[]} = [] \)

\( \text{mapAList} = \lambda x \rightarrow \lambda y \rightarrow \text{map} \; y \; x \)

\( \text{doubleAList} = \text{mapAList} \; \lambda z \rightarrow z + z \)

\( \lambda x \rightarrow \text{map} \; (\lambda z \rightarrow z + z) \; y \)
Operations on Functions

Application: \( F \ n \)

If \( F \) is a lambda exp, substitute \( n \) for the variable

Serial composition: \( F \) of \( G \) of \( n \)

Substitute \( (G \ n) \) for variable in \( F \)

Parallel composition: \( F \ n, G \ n \)

Produce a pair of values

Currying: Applying one parameter

\[
\text{curryExample} = \lambda x \rightarrow \lambda y \rightarrow x + y
\]

\[
\text{curryExample2} \ x \ y = x + y
\]

\[
\text{curried} = \text{curryExample} \ 7
\]

\[
\text{curried2} = \lambda x \rightarrow x + 7
\]
First-Order Functions

Lambda expression adds a variable

Currying removes one

Build new values by successively adding, removing variables, composing, etc.

Difference between number and no variable abst?

\[
\begin{align*}
    \text{aNumber} &= 17 \\
    \text{aAbstNumber} &= \lambda x \rightarrow 17
\end{align*}
\]
Next Week, Unification and More on Types

Reread 2nd half of Ch 11

Begin Ch 7