First-Order Functions and Unification

Gimme, gimme yer homework!

What is a first-order function?

What is unification?

Relationship to types and polymorphism

But first, homework....
Q1, Support of Polymorphism

Self

Delegation → ad hoc
Dynamic “typing” → parametric

Haskell: Both

Scheme

Dynamic typing → parametric
No native ad hoc support, can build as extensions

Java: Ad hoc only, Parametric faked with interfaces

Eiffel: Both
Q2, Using Polymorphism

Dictionary: Parametric (Dict of any type)

Registrar: Ad hoc (Student base class)

Factory: Either

  Parametric over type of manufactured obj, types of machine, process of manufacture
  Ad hoc plug-in process and machines, get manufactured result

Graphics System: Both

  Collections of shapes: Parametric
  Categories of shapes and views: Ad hoc
  Multiple argument ad hoc polymorphism?
Multi-Methods

Most OOPLs tie messages to a single receiver

    Different method chosen based on receiver

Some allow choice of method based on several objects; This is called “multi-methods” or “generic functions”

First appeared in CLOS (Lisp), now in a few others (Cecil, Dylan)
Multi-Method Example: Draw

(draw aShape aWindow) — the message

Methods list types for arguments:

(draw aShape:Sphere aWindow:OpenGLWin
  -- make OpenGL calls to draw a sphere)
(draw aShape:Cube aWindow:OpenGLWin
  -- make OpenGL calls for a cube)
(draw aShape:Sphere aWindow:DirectXWin
  -- make DirectX calls for a sphere)
(draw aShape:Sphere aWindow:PDFWin
  -- generate PDF code to render a sphere)

Most specific method is used

Impl: n-dimensional table of methods
Impl: graph lookup; BFS along multiple inheritance hierarchies
Functional “vs” OOP

Generic functions/multi-methods as generalization of OOP “virtuals”

Ties together ad hoc and parametric

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 no distinction between code and data

  Approach pioneered by Lisp
  Cleaned up in Scheme
Higher-Order Functions

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

Functions can return functions

This is a higher-order function

\[ f \to g \text{ is zeroth order} \]
\[ f \to g \to h \text{ is first order} \]
\[ f \to g \to h \to i \text{ is second order} \]
Etc.
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
- Or, write native machine code into memory
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
\x -> x + x -- backslash meant to look like $\lambda$

Give it a name to use it as a regular function

double = \x -> x + x
double2 x = x + x -- syntactic sugar for double
Functions as Values

A lambda expression is either a name, a function, an invocation, or a parenthesized expression

Formally,
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

\[ \text{example} = \lambda x \rightarrow \lambda y \rightarrow y \times x \]

user> example 665

\[ \text{example} \ 665 = \lambda x \rightarrow 665 \times x \]

\[ \lambda x \rightarrow 665 \times x \]

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

\[ \text{map} _ \ [] = [] \]

\[ \text{doubleAList} = \text{map} \ \lambda z \rightarrow z + z \]

-- evaluates to:

\[ \lambda x \rightarrow \text{map} \ (\lambda z \rightarrow z + z) \ x \]

-- or:

\[ \text{doubleAList} \ \text{alist} = (\text{head} \ \text{alist}) + (\text{head} \ \text{alist}) : \]

\[ \text{map} \ (\lambda x \rightarrow x+x) \ (\text{tail} \ \text{alist}) \]

\[ \text{doubleAList} \ [] = [] \]
Operations on Functions

Application: F n

If F is a lambda abstraction, 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

```haskell
curryExample = \x -> \y -> x + y
curryExample2 x y = x + y
curried = curryExample 7
curried2 = \x -> x + 7
```
First-Order Functions

Lambda abstraction adds a variable

Currying removes one

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

Difference between number and a λ abstraction with no use of variable?

\[ \text{aNumber} = 17 \]
\[ \text{aAbstNumber} = \lambda x \to 17 \]

Eta reduction eliminates lambda
Unification

Recursively matching predicates and facts

Pioneered as PL mechanism in Prolog
Theory present in 1st-order predicate calculus

Process called Unification

Comparison: Parsing
Does a string unify with a grammar?

Comparison: Pattern-matching types
Does a value unify with a type?
Set-up: Horn Clauses

Either atom, variable, or predicate

Predicate: Consequent implied by a body

Body is logical conjunction of clauses (or terms)

Empty body is a “fact” or “rule”

parent(Brook, Zooey).
parent(Chris, Brook).
grandparent(X, Y) :- parent(X,Z),parent(Z,Y).
?- parent(Brook, X). → Zooey.
?- parent(Zooey, X). → no.
?- grandparent(X, Zooey). → Chris.
Resolution

Given two Horn clauses C and D:

\[ c(X) :- \text{foo}(X), \text{bar}(X). \]
\[ d(Z) :- c(Z), \text{baz}(Z), \text{dingle}(Z). \]

If the consequent of C is a term in D, then consequent C in D can be replaced by the body of C

\[ d(Z) :- \text{foo}(Z), \text{bar}(Z), \text{baz}(Z), \text{dingle}(Z). \]

Add *instantiation* of variables

\[ \text{foo}(\text{ape}). \]

Can variable be tied to atom everywhere in clause?

\[ d(\text{ape}) :- \text{foo}(\text{ape}), \text{bar}(\text{ape}), \text{baz}(\text{ape}), \text{dingle}(\text{ape}). \]
Unification

A constant unifies with itself

Two terms unify if:
- they have the same name (functor)
- they have the same number of args
- their args unify recursively

A variable unifies with anything

- Unify with a non-variable: instantiation
- Unifying with another variable: ties variables together; unifying either will unify the other

Resolution explores possible unifications
Compare: Parsing

Horn clauses as rules in a grammar

Query consists of unifying string with top production

\[
\text{program} \rightarrow \text{functions}, \text{mainline}
\]
\[
\text{functions} \rightarrow \text{function,functions} | \text{nothing}
\]

Insert implicit variables for substrings; then query

\[
\text{program}(S_1, S_2) :-
\]
\[
\text{functions}(\text{split}(S_1)), \text{mainline}(S_2).
\]
\[
\text{functions}(S_1, S_2) :-
\]
\[
\text{function}(S_1), \text{functions}(\text{split}(S_2)).
\]
\[
\text{functions}(S_1, S_2) :- \text{nothing}.
\]

?- program("", "printf(\"Hello, World\")");")
Compare: Pattern Matching

Types

Unify run-time values with type expressions

\[
\text{map } x \ a : \text{list} = x \ a : \text{map } x \ \text{list} \\
\text{map } _ [ ] = []
\]

\[
\text{map } \ \lambda x \to x^*2 \ [ ]
\]

first fails; empty list does not match with cons of something and a list
second succeeds; _ matches anything, [] is a constant and matches with itself

Commonality of mechanisms, again
Next Week

Types: Ch 7

Homework assignment up on website