Clean-Up and Wrap-Up

CS242

Lecture 18

The Final

- Final exam is 8:30-11:30 next Wednesday (Dec. 11)
  - In B1, Gates

- Open note, and electronic devices are OK
  - But no internet or computation, only use to read your notes

The Untyped and Simply Typed Lambda Calculi

Untyped lambda calculus:

$e \rightarrow x \mid \lambda x.e \mid e\ e$

Simply typed lambda calculus:

$e \rightarrow x \mid \lambda x{:}\ t.e \mid e\ e \mid i$

$t \rightarrow \alpha \mid t \rightarrow t \mid int$

Extension 1: Algebraic Data Types

General form

$\text{DataType } A(var_1,\ldots,var_n):$

$\ldots$

$\text{Constructor}_i: t_1 \to_{\ldots} \to t_k \to A\ (var_1,\ldots,var_n)$

$\ldots$

Each constructor defines a pure lambda term.

Example: Lists

Consider the list data type:

List(A):

nil: List(A)

cons: A -> List(A) -> List(A)

nil: λn.λc.n

cons: λh.λt.λn.λc.c h (t n c)

Other Examples

- Non-negative integers
- Pairs
- Booleans
- Binary trees

- In general, any tree-shaped data structure

Extension 2: Constants

- We can extend the lambda calculus with additional functions and constants

- Example
  - Add all integers …, -1, 0, 1, …
  - And addition. +: int → int → int

- Other typical built-ins:
  - Floating point numbers
  - Booleans
  - Characters
  - Strings
  - Arrays

Control Constructs: If and Recursion

We can also extend the calculus with control constructs

if: Bool $\rightarrow$ t $\rightarrow$ t $\rightarrow$ t

Usage: if e_1 e_2 e_3

Question: Why would if-then-else need to be built in rather than defined within the lambda calculus?

Typing Checking for If

$$\frac{A \vdash e_1 : \text{Bool} \qquad A \vdash e_2 : t \qquad A \vdash e_3 : t}{A \vdash \text{if } e_1\ e_2\ e_3 : t} \quad [\text{If}]$$

Typing Inference for If

$$\frac{\begin{array}{l} A \vdash e_1 : \text{Bool} \\ A \vdash e_2 : t_1 \\ A \vdash e_3 : t_2 \\ t_1 = t_2 \end{array}}{A \vdash \text{if } e_1\ e_2\ e_3 : t_1} \quad [\text{If}]$$

Recursion

Recall

$\text{let } x = e_1 \text{ in } e_2$ is equivalent to $(\lambda x.e_2)\ e_1$

Extend to recursive definitions

$\text{letrec } f = \lambda x.e_1 \text{ in } e_2$ is equivalent to $(\lambda f.e_2)\ (Y\ \lambda f.\lambda x.e_1)$

recall (lecture 4) that $Y = \lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f(x\ x))$

Typing Checking for Recursive Definitions

$$\frac{A, f: t_1 \rightarrow t_2 \vdash \lambda x.e_1 : t_1 \rightarrow t_2 \qquad A, f: t_1 \rightarrow t_2 \vdash e_2 : t}{A \vdash \text{letrec } f = \lambda x.e_1 \text{ in } e_2 : t} \quad \text{[Letrec]}$$

Typing Inference for Recursive Definitions

$$\frac{A, f: \alpha \rightarrow \beta \vdash \lambda x.e_1 : t_1 \rightarrow t_2 \qquad A, f: \alpha \rightarrow \beta \vdash e_2 : t \qquad \alpha = t_1 \quad \beta = t_2}{A \vdash \text{letrec } f = \lambda x.e_1 \text{ in } e_2 : t} \; [\text{Letrec}]$$

Note: This inference rule is more restrictive than necessary – the full rule introduces polymorphic (universally quantified) types ala non-recursive let.

A Question

Why would recursion need to be built in rather than written within the language using the Y combinator?

Extension 3: Polymorphic Types

$e \rightarrow x \mid \lambda x.e \mid e\ e \mid \text{let } f = \lambda x.e \text{ in } e \mid i$

$t \rightarrow \alpha \mid t \rightarrow t \mid \text{int}$

$o \rightarrow \forall \alpha.o \mid t$

Subtyping: A Subtle Topic

$$\frac{A \vdash e_1 : Bool \quad A \vdash e_2 : t_1 \quad A \vdash e_3 : t_2 \quad t_1 = t_2}{A \vdash \text{if } e_1\ e_2\ e_3 : t_1}\ [\text{If}]$$

$$\frac{A \vdash e_1 : Bool \quad A \vdash e_2 : t_1 \quad A \vdash e_3 : t_2 \quad t_1 < t \quad t_2 < t}{A \vdash \text{if } e_1\ e_2\ e_3 : t}\ [\text{If}]$$

Java's Type Rule for ? (Approximately ...)

$$\frac{A \vdash e_1 : \text{Bool} \quad A \vdash e_2 : t_1 \quad A \vdash e_3 : t_2 \quad t_3 = \text{lub}(t_1, t_2)}{A \vdash e_1 ? e_2 : e_3 : t_3} \text{[If]}$$

lub = least upper bound

$\text{lub}(t_1, t_2)$ is the smallest type that is a supertype of both t_1 and t_2

What Else Didn't We Talk About?

- Traditional overloading

- Having multiple functions of different types with the same name

+: int → int → int
+: float → float → float
+: string → string → string

Overloading rules in languages with subtyping are complicated.

Functional Languages

- Lambda calculus + primitive functions + algebraic data types

- These features are the core of all functional languages
  - Lisp, Scheme, Racket

- Plus polymorphic types for typed functional languages
  - ML, OCaml, Haskell

Monads

- Plumbs generalized “state” through a computation
  - Makes implicit arguments (like global variables and state) explicit
  - Does the sequencing through higher-order functions
- Many language features can be expressed as monads
  - State
  - Continuations
  - Exceptions
  - (Some kinds of) threads
- All except pure functional languages have some built-in monads
  - Typically state and exceptions, continuations and threads are less common
  - Haskell exposes monads to the programmer – define your own language features!

Objects

- Objects are something different
  - Typed object-oriented languages are not easily translated into typed functional languages

- Unrestricted method override is difficult to deal with in typed systems

- Solutions
  - Restrict method override: Java, C++ limit it to inheritance between classes
  - Use core functional language + records to get most of OO: OCaml, Haskell
  - Go to an untyped language: Python, Javascript
  - Use traits, mixins: Rust, Scala

Big Picture

- All mainstream languages have converged on supporting
  - Objects
  - First-class functions
  - Means typed languages must deal with parametric polymorphism and subtyping in some way

- The details vary
  - Because the theory suggests there is no one best design

- But why did this happen?

Object Oriented vs Functional Languages

- Functional language example:

f cons(a,b) = a

f nil = nil

Adding a new function is a local change.

Adding a new kind of data, such as a new constructor to a data type, requires updating every function that uses that type.

Object Oriented vs Functional Languages

- Object-oriented language example:

```
Class List of
   method cons(x,y) …
   method nil …
end
```

Adding a new kind of data type is a local change.

Adding a new function (method) may require updating many classes with a definition of that method (modulo inheritance).

Adding Objects to Functional Languages

- *Type classes* are Haskell's way of providing object-like features
  - But really much closer to Java's interfaces than objects

- Examples

(==) :: Eq a => a -> a -> bool

Any type a that supports equality should be part of the Eq class

(<) :: Ord a => a -> a -> bool

Any type a that supports ordering should be part of the Ord class

Type Classes

(<) :: Ord a => a -> a -> bool

Idea: Code that requires certain functionality can require a value of the appropriate type class, without saying how it is implemented.

Example: A generic sorting function can take a comparison function < in the Ord type class as an argument.

Adding Functions to OO Languages

- C++ has had lambdas since C++14
  - Involves explicitly naming captured variables
  - And whether they are captured by value or reference

- Java has had lambdas since Java 8

- And both have polymorphic types
  - C++ has templates
  - Java has generics

Bottom Line

- There is no single best way to combine functional and object-oriented features.

- Emphasizing some features requires restricting other features.

Approaches to Proving Properties of Programs

Automatic, Low complexity	Automatic, High complexity	Automatic or Semi-automatic Often undecidable	Manual, Undecidable
Simply Typed Lambda Calculus	Static Analysis	Invariant Inference	Dependent Types
CS242	CS143, CS243		CS242

Inductive (Loop) Invariants

Pre

while (B)

I

{

... code ...

Post

}

$Pre \rightarrow I$

$I \wedge B$

$\{ code \}$

I

$I \wedge \neg B \rightarrow Post$

A Loop Invariant Example

```
int A[10];
i = 1
// i = 1
while i < 11 {
    // ∀1 ≤ j < i.  A[j] = 0
    A[i] = 0;
    i += 1
}
// ∀1 ≤ j ≤ 10.  A[j] = 0
```

Three conditions:

$i = 1 \rightarrow \forall 1 \leq j < i.\ \ A[j] = 0$

$\forall 1 \leq j < i.\ \ A[j] = 0$
$\{ A[i] = 0; i = i + 1\}$
$\forall 1 \leq j < i.\ \ A[j] = 0$

$((\forall 1 \leq j < i.\ \ A[j] = 0) \wedge\ i \geq 11) \rightarrow$
$\forall 1 \leq j \leq 10.\ \ A[j] = 0$

Types As Propositions

$$\frac{A \vdash e_1 : t \to t' \qquad A \vdash e_2 : t}{A \vdash e_1\, e_2 : t'} \; [\text{App}]$$

$$\frac{A, x : t \vdash e : t'}{A \vdash \lambda x.e : t \to t'} \; [\text{Abs}]$$

From a proof of $t \to t'$ and and a proof of t, we can prove t'.

If assuming t we can prove t', then we can prove $t \to t'$.

Here we regard the types as propositions: If we can prove certain propositions are true, then we can prove that other propositions are true.

Approaches to Proving Properties of Programs

Automatic,
Low complexity

Automatic,
High complexity

Automatic or Semi-automatic
Often undecidable

Manual,
Undecidable

Simply Typed
Lambda Calculus

Every typed
language

Gradual Types

Static Analysis

Every optimizing
compiler

Invariant Inference

Still figuring this
part out …

Dependent Types

Emerging from
the lab …

Other topics …

- Concurrency and parallelism

- Very different from sequential languages
  - Not well-modeled by lambda calculus, object calculus etc.
  - Requires entirely different approaches that makes concurrency primitive

- Will be an increasingly important aspect of programming languages
  - And unfortunately something we did not have time to get into in this course!

The End … and Thanks!