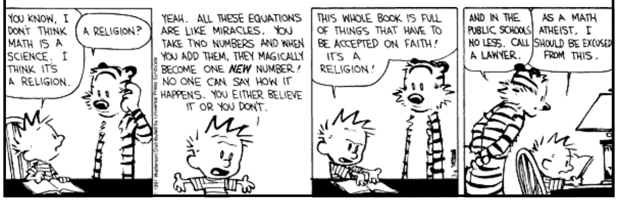
More Lambda Calculus
and
Intro to Type Systems



The Reading

- Explain the Xavier Leroy article to me …

The correctness of the translation follows from a simulation argument between the executions of the Cminor source and the RTL translation, proved by induction on the Cminor evaluation derivation. In the case of expressions, the simulation property is summarized by the following diagram:

$$\begin{tikzcd} sp, L, a, E, M \arrow[r, dash, "I \wedge P"] \arrow[d, Rightarrow] & sp, n_s, R, M \arrow[d, dotted, "*"] \\ sp, L, v, E', M' \arrow[r, dotted, dash, "I \wedge Q"] & sp, n_d, R', M' \end{tikzcd}$$

On the choice of semantics We used big-step semantics for the source language, "mixed-step" semantics for the intermediate languages, and small-step semantics for the target language. A consequence of this choice is that our semantic preservation theorems hold only for terminating source programs: they all have premises of the form "if the source program evaluates to result r", which do not hold for non-terminating programs. This is unfortunate for

- How did he do register allocation?

Plan

- Heavy Class Participation
  - Thus, wake up! *(not actually kidding)*
- Lambda Calculus
  - How is it related to real life?
  - Encodings
  - Fixed points
- Type Systems
  - Overview
  - Static, Dyamic
  - Safety, Judgments, Derivations, Soundness

Lambda Review

- λ-calculus is a calculus of functions

  $e ::= x \mid \lambda x.\ e \mid e_1\ e_2$

- Several evaluation strategies exist based on β-reduction

  $(\lambda x.e)\ e' \to_\beta [e'/x]\ e$

- How does this simple calculus relate to real programming languages?

Functional Programming

- The λ-calculus is a prototypical functional language with:
  - no side effects
  - several evaluation strategies
  - lots of functions
  - nothing but functions (pure λ-calculus does not have any other data type)

- How can we program with functions?
- How can we program with *only* functions?

Programming With Functions

- Functional programming is a programming style that relies on lots of functions
- A typical functional paradigm is *using functions as arguments or results of other functions*
  - Called "higher-order programming"
- Some "impure" functional languages permit side-effects (e.g., Lisp, Scheme, ML, Python)
  - references (pointers), in-place update, arrays, exceptions
  - Others (and by "others" we mean "Haskell") use monads to model state updates

Variables in Functional Languages

- We can introduce new variables:

  let $x = e_1$ in e_2
  - x is bound by let
  - x is statically scoped in (a subset of) e_2
- This is pretty much like $(\lambda x.\ e_2)\ e_1$
- In a functional language, variables are never updated
  - they are just *names for expressions or values*
  - e.g., x is a name for the value denoted by e_1 in e_2
- This models the meaning of "let" in math (proofs)

#7

Referential Transparency

- In "pure" functional programs, we can reason equationally, by substitution
  - Called "referential transparency"

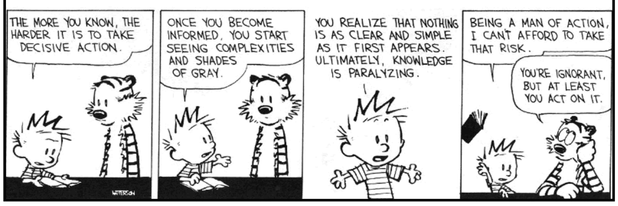
  let $x = e_1$ in e_2 === $[e_1/x]e_2$
- In an imperative language a side-effect in e_1 might invalidate the above equation
- The behavior of a function in a "pure" functional language depends only on the actual arguments
  - Just like a function in math
  - This makes it easier to understand and to reason about functional programs

#8

How Tough Is Lambda?

- Given e_1 and e_2, how complex (a la CS theory) is it to determine if:

$$e_1 \rightarrow_\beta^* e \quad \textit{and} \quad e_2 \rightarrow_\beta^* e$$



Expressiveness of λ-Calculus

- The λ-calculus is a minimal system but can express
  - data types (integers, booleans, lists, trees, etc.)
  - branching
  - recursion
- This is enough to encode Turing machines
  - We say the lambda calculus is Turing-complete
- Corollary: $e_1 =_\beta e_2$ is undecidable
- Still, how do we encode all these constructs using only functions?
- Idea: *encode the **"behavior"** of values* and not their structure

#10

Encoding Booleans in λ-Calculus

- What can we *do* with a boolean?
  - we can make a binary choice (= "if" statement)
- A boolean is a function that, given two choices, selects one of them:
  - **true** $=_{def}$ $\lambda x.\ \lambda y.\ x$
  - **false** $=_{def}$ $\lambda x.\ \lambda y.\ y$
  - **if E_1 then E_2 else E_3** $=_{def}$ $E_1\ E_2\ E_3$
- Example: "if true then u else v" is

  $(\lambda x.\ \lambda y.\ x)\ u\ v \rightarrow_\beta (\lambda y.\ u)\ v \rightarrow_\beta u$

#11

More Boolean Encodings

- Let's try to do boolean or together
- Recall:
  - true $=_{def}$ $\lambda x.\ \lambda y.\ x$
  - false $=_{def}$ $\lambda x.\ \lambda y.\ y$
  - if E_1 then E_2 else E_3 $=_{def}$ $E_1\ E_2\ E_3$
- We want or to take in two booleans and yield a result that is a boolean
- How can we do this?

#12

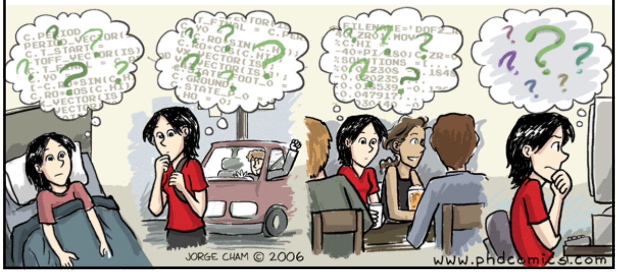
A Trying Ordeal

- Recall:
  - true $=_{def}$ $\lambda x.\ \lambda y.\ x$
  - false $=_{def}$ $\lambda x.\ \lambda y.\ y$
  - if E_1 then E_2 else E_3 $=_{def}$ $E_1\ E_2\ E_3$
- Intution:
  - or a b = if a then true else b
- Either of these will work:
  - or $=_{def}$ $\lambda a.\ \lambda b.\ a$ true b
  - or $=_{def}$ $\lambda a.\ \lambda b.\ \lambda x.\ \lambda y.\ a\ x\ (b\ x\ y)$

#13

Final Boolean Encodings

- Think about how to do and and not
- Without peeking!



#14

Another Demand

- How to do and and not
- and a b = if a then b else false
  - and $=_{def}$ $\lambda a.\ \lambda b.\ a\ b$ false
  - and $=_{def}$ $\lambda a.\ \lambda b.\ \lambda x.\ \lambda y.\ a\ (b\ x\ y)\ y$
- not a = if a then false else true
  - not $=_{def}$ $\lambda a.\ a$ false true
  - not $=_{def}$ $\lambda a.\ \lambda x.\ \lambda y.\ a\ y\ x$

#15

Encoding Pairs in λ-Calculus

- What can we *do* with a pair?
  - we can access one of its elements (= ".field access")
- A pair is a function that, given a boolean, returns the first or second element

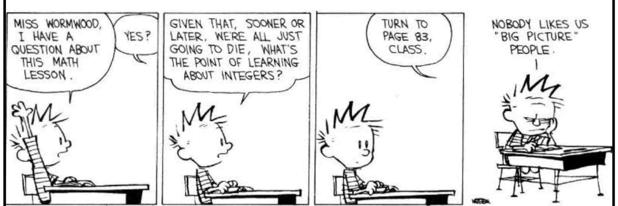
  mkpair x y $=_{def}$ $\lambda b.\ b\ x\ y$
  fst p $=_{def}$ p true
  snd p $=_{def}$ p false

- fst (mkpair x y) $\rightarrow_\beta$ (mkpair x y) true $\rightarrow_\beta$ true x y $\rightarrow_\beta$ x

#16

Encoding Numbers in λ–Calculus

- What can we *do* with a natural number?
  - What do you, the viewers at home, think?



#17

Encoding Numbers λ-Calculus

- What can we *do* with a natural number?
  - we can iterate a number of times over some function (= "for loop")
- A natural number is a function that given an operation f and a starting value s, applies f a number of times to s:

  0 $=_{def}$ $\lambda f.\ \lambda s.\ s$
  1 $=_{def}$ $\lambda f.\ \lambda s.\ f\ s$
  2 $=_{def}$ $\lambda f.\ \lambda s.\ f\ (f\ s)$

  - Very similar to List.fold_left and friends
- These are numerals in a unary representation
- Called Church numerals

#18

Test Time!

- How would you encode the successor function (succ x = x+1)?
- How would you encode more general addition?
- Recall: $4 =_{def} \lambda f.\ \lambda s.\ f\ f\ f\ (f\ s)$

Computing with Natural Numbers

- The successor function
  - succ n $=_{def} \lambda f.\ \lambda s.\ f\ (n\ f\ s)$
  - *or* succ n $=_{def} \lambda f.\ \lambda s.\ n\ f\ (f\ s)$
- Addition
  - add n_1 n_2 $=_{def} n_1$ succ n_2
- Multiplication
  - mult n_1 n_2 $=_{def} n_1$ (add n_2) 0
- Testing equality with 0
  - iszero n $=_{def}$ n (λb. false) true
- Subtraction
  - Is not instructive, but makes a fun exercise …

Computation Example

- What is the result of the application add 0?
  - $(\lambda n_1.\ \lambda n_2.\ n_1\ \text{succ}\ n_2)\ 0 \ \to_\beta$
  - $\lambda n_2.\ 0\ \text{succ}\ n_2 =$
  - $\lambda n_2.\ (\lambda f.\ \lambda s.\ s)\ \text{succ}\ n_2 \to_\beta$
  - $\lambda n_2.\ n_2 =$
  - $\lambda x.\ x$
- By computing with functions we can express some optimizations
  - But we need to reduce under the lambda
  - Thus this "never" happens in practice

Toward Recursion

- Given a predicate P, encode the function "find" such that "find P n" is the smallest natural number which is larger than n and satisfies P
- Claim: with find we can encode all recursion

Intuitively, why is this true?

Encoding Recursion

- Given a predicate P encode the function "find" such that "find P n" is the smallest natural number which is larger than n and satisfies P
- find satisfies the equation

  find p n = if p n then n else find p (succ n)
- Define

  F = λf.λp.λn.(p n) n (f p (succ n))
- We need a fixed point of F

  find = F find

  or

  find p n = F find p n

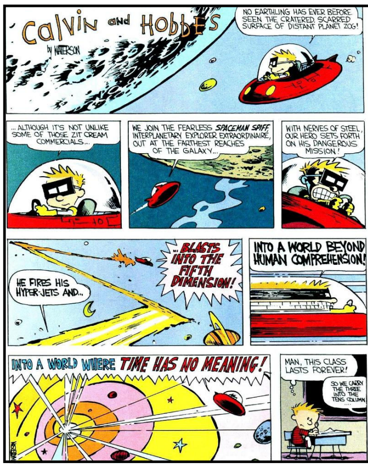
The Fixed-Point Combinator Y

- Let $Y = \lambda F.\ (\lambda y.F(y\ y))\ (\lambda x.\ F(x\ x))$
  - This is called the fixed-point combinator
  - Verify that Y F is a fixed point of F

    $Y\ F \to_\beta (\lambda y.F\ (y\ y))\ (\lambda x.\ F\ (x\ x)) \to_\beta F\ (Y\ F)$
  - Thus $Y\ F =_\beta F\ (Y\ F)$
- Given any function in λ-calculus we can compute its fixed-point (w00t! why do we not win here?)
- Thus we can define "find" as the fixed-point of the function F from the previous slide
- Essence of recursion is the self-application "y y"

Expressiveness of Lambda Calculus

- Encodings are fun
  - Yes! Yes they are!
- But programming in pure λ-calculus is painful
- So we will add constants (0, 1, 2, …, true, false, if-then-else, etc.)
- Next we will add *types*



Still Going!

- One minute break
- Stretch!

Types

- A program variable can assume a *range of values* during the execution of a program
- An upper bound of such a range is called a type of the variable
  - A variable of type “bool” is supposed to assume only boolean values
  - If x has type “bool” then the boolean expression “not(x)” has a sensible meaning during every run of the program

Typed and Untyped Languages

- Untyped languages
  - Do *not* restrict the range of values for a given variable
  - Operations might be applied to inappropriate arguments. The behavior in such cases might be unspecified
  - The pure λ-calculus is an extreme case of an untyped language (however, its behavior is completely specified)
- (Statically) Typed languages
  - Variables are assigned (non-trivial) types
  - A type system keeps track of types
  - Types might or might not appear in the program itself
  - Languages can be explicitly typed or implicitly typed

The Purpose Of Types

- The foremost purpose of types is *to prevent certain types of run-time execution errors*
- Traditional trapped execution errors
  - Cause the computation to stop immediately
  - And are thus well-specified behavior
  - Usually enforced by hardware
  - e.g., Division by zero, floating point op with a NaN
  - e.g., Dereferencing the address 0 (on most systems)
- Untrapped execution errors
  - Behavior is unspecified (depends on the state of the machine = this is very bad!)
  - e.g., accessing past the end of an array
  - e.g., jumping to an address in the data segment

Execution Errors

- A program is deemed safe if it does *not* cause untrapped errors
  - Languages in which all programs are safe are safe languages
- For a given language we can designate a set of forbidden errors
  - A superset of the untrapped errors, usually including some trapped errors as well
    - e.g., null pointer dereference
- Modern Type System Powers:
  - prevent race conditions (e.g., Flanagan TLDI ‘05)
  - prevent insecure information flow (e.g., Li POPL ’05)
  - prevent resource leaks (e.g., Vault, Weimer)
  - help with generic programming, probabilistic languages, …
  - … are often combined with dynamic analyses (e.g., CCured)

Preventing Forbidden Errors - Static Checking

- Forbidden errors can be caught by a combination of static and run-time checking
- Static checking
  - Detects errors early, *before testing*
  - Types provide the necessary static information for static checking
  - e.g., ML, Modula-3, Java
  - Detecting certain errors statically is undecidable in most languages

#31

Preventing Forbidden Errors - Dynamic Checking

- Required when static checking is undecidable
  - e.g., array-bounds checking
- Run-time encodings of types are still used (e.g. Lisp)
- Should be limited since it delays the manifestation of errors
- Can be done in hardware (e.g. null-pointer)

#32

Safe Languages

- There are typed languages that are not safe ("weakly typed languages")
- *All safe languages use types* (static or dynamic)

	Typed		Untyped
	Static	Dynamic	
Safe	ML, Java, Ada, C#, Haskell, ...	Lisp, Scheme, Ruby, Perl, Smalltalk, PHP, Python, …	λ-calculus
Unsafe	C, C++, Pascal, ...	?	Assembly

- We focus on statically typed languages

#33

Why Typed Languages?

- Development
  - *Type checking catches early many mistakes*
  - Reduced debugging time
  - Typed signatures are a powerful basis for design
  - Typed signatures enable separate compilation
- Maintenance
  - Types act as checked specifications
  - Types can enforce abstraction
- Execution
  - Static checking reduces the need for dynamic checking
  - Safe languages are easier to analyze statically
    - the compiler can generate better code

#34

Homework

- Read Cardelli article
- Read great works of literature
- Homework 5 Due In A Fortnight

#35