Lecture 9: More Lambda Calculus / Types

CSC 131
Spring, 2019

Kim Bruce

Pure Lambda Calculus

- Terms of pure lambda calculus
  - $M ::= v \mid (M\ M) \mid \lambda v.\ M$
  - Impure versions add constants, but not necessary!
  - Turing-complete
- Left associative: $M\ N\ P = (M\ N)\ P$.
- Computation based on substituting actual parameter for formal parameters

Computation Rules

- Reduction rules for lambda calculus:

  (α) $\lambda x.\ M \rightarrow \lambda y.\ ([y/x]\ M)$, if $y \notin FV(M)$.

  change name of parameters if new not capture old

  (β) $(\lambda x.\ M)\ N \rightarrow [N/x]\ M$.

  computation by subst function argument for formal parameter

  (η) $\lambda x.\ (M\ x) \rightarrow M$.

  Optional rule to get rid of excess λ's

Computability

- Can encode all computable functions in pure untyped lambda calculus.
  - true = λ u. λ v. u
    - true a b = a
  - false = λ u. λ v. v
    - false a b = b
  - cond = λ u. λ v. λ w. u v w
    - cond true a b = ? cond false a b = ?

Encoding Natural Numbers

- Natural numbers:
  - $\underline{0} = \lambda\ s.\ \lambda\ z.\ z.$
  - $\underline{1} = \lambda\ s.\ \lambda\ z.\ s\ z.$
  - $\underline{2} = \lambda\ s.\ \lambda\ z.\ s\ (s\ z).$
- Integers encode repetition:
  - $\underline{2}\ f\ x = f\ (f\ x)$
  - $\underline{3}\ f\ x = f(\ f\ (f\ x))$
  - $\underline{n}\ f\ x = f^{(n)}\ (x)$

Arithmetic

- Succ = λ n. λ s. λ z. s (n s z)
  - Succ n = λ s. λ z. s (n s z) = λ s. λ z. s ($s^{(n)}$ z) = n+1
- Plus = λ n. λ m. λ s. λ z. m s (n s z).
- Mult = λ n. λ m. (m (Plus n) 0).
- isZero = λ n. n (λ x. false) true
- Subtraction is hard!!

Recursion

- A different perspective: Start with
  - fact = λn. cond (isZero n) 1 (Mult n (fact (Pred n)))
- Let F stand for the closed term:
  - λf. λn. cond (isZero n) 1 (Mult n (f (Pred n)))
  - Notice F(fact) = fact.
  - fact is a *fixed point* of F
  - To find fact, need only find fixed point of F!
- Easy w/ g(x) = x * x, but F????

Fixed Points

- Several fixed point operators:
  - Ex: $\underline{Y} = \lambda f\,.\,(\lambda x.\ f\,(xx))(\lambda x.\ f\,(xx))$

Invented by Haskell Curry

- Claim for all g, $\underline{Y}\,g = g\,(\underline{Y}\,g)$

$$\underline{Y}\,g = (\lambda f\,.\,(\lambda x.\ f\,(xx))(\lambda x.\ f\,(xx)))\ g$$

$$= (\lambda x.\ g(xx))\underline{(\lambda x.\ g(xx))}$$

$$= g(\underline{(\lambda x.\ g(xx))}\ \underline{(\lambda x.\ g(xx))})$$

$$= g\,(\underline{Y}\,g)$$

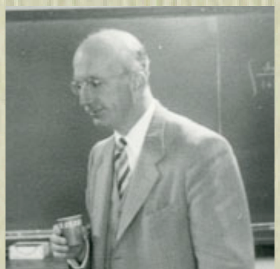
- If let $x_0 = \underline{Y}\,g$, then $g\,(x_0) = x_0$.

Lambda Calculus

- λ-calculus invented in 1928 by Church in Princeton & first published in 1932.
- Goal to provide a foundation for logic
- First to state explicit conversion rules.
- Original version inconsistent, but corrected
  - "If this sentence is true then 1 = 2" *problematic!!*
- 1933, definition of natural numbers

Collaborators

- 1931-1934: Grad students:



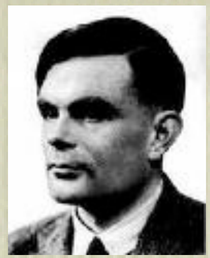
  - J. Barkley Rosser and Stephen Kleene
  - Church-Rosser confluence theorem ensured consistency (earlier version inconsistent)
  - Kleene showed λ-definable functions very rich
    - Equivalent to Herbrand-Gödel recursive functions
    - Equivalent to Turing-computable functions.
    - Founder of recursion theory, invented regular expressions

- Church's thesis:
  - λ-definability ≡ effectively computable

Undecidability

- Convertibility problem for λ-calculus undecidable.
- Validity in first-order predicate logic undecidable.
- Proved independently year later by Turing.
  - First showed halting problem undecidable

Alan Turing



- Turing
  - 1936, in Cambridge, England, definition of Turing machine
  - 1936-38, in Princeton to get Ph.D. under Church.
  - 1937, first published fixed point combinator
    - (λx. λy. (y (x x y))) (λx. λy. (y (x x y)))
  - *Kleene did not use fixed-point operator in defining functions on natural numbers!*
  - Broke German enigma code in WW2, Turing test AI
  - Persecuted as homosexual, committed suicide in 1954

Typed Lambda Calculus

Types

- Can specify types of identifiers
- Start with base type e and build up types and terms:
  - Type ::= e | Type → Type
  - M ::= v | (M M) | λv : Type. M
- Examples:
  - Types: e, e → e, e → e → e, (e → e) → e, ...
  - Terms: λx: e. x, λf: e → e. λz: e. f(f(z))

Definitions

- Earlier definitions generalize over types t:
  - $\text{true}^t = \lambda x{:}t.\ \lambda y{:}u.\ x$
  - $\underline{n}^t = \lambda s{:}\ t \to t.\ \lambda z{:}\ t.\ s^{(n)}(z)$
- Some untyped terms can't be typed:
  - $\Omega = (\lambda x.\ (x\ x))(\lambda x.\ (x\ x))$
  - $Y = \lambda f\ .\ (\lambda x.\ f\ (x\ x))(\lambda x.\ f\ (x\ x))$

Totally Awesome!!

- Theorem: If M is a term of the typed lambda calculus, then M has a unique normal form. *I.e., every term of the typed lambda calculus is total.*
- Corollary: The typed lambda calculus does *not* include all computable functions.

Types

Why (Static) Types?

- Increase readability, esp. for libraries
- Hide representation
- Detection of errors.
- Help disambiguate operators
- Compiler optimization. E.g. know where fields of record/struct are.
- Help ensure different components in separately compiled units will interoperate properly
- Provide basis for code completion in editors

Types & Constructors

- Built-in types - *primitive types (incl. size)*
- Aggregate types - *records/structs*
- Mapping types - *arrays/functions*
- Recursive types - *lists/trees*
- Sequence types - *files and strings (primitive?)*
- User-defined types

Aggregate Types

- Cartesian products (tuples)
- Records / Structs
- Union Types
  - C: typedef union {int i; float r;} utype
  - unsafe
  - Discriminated union safer
  - Haskell type defs safe

Discriminated Union: Ada

```
type geometric (Kind: (Triangle, Square) := Square) is
   record
      color : ColorType := Red ;
      case Kind of
         when Triangle =>
              pt1,pt2,pt3:Point;
         when Square =>
              upperleft : Point;
              length : INTEGER range 1..100;
      end case;
   end record;

ob1 : geometric -- default is Square
ob2 : geometric(Triangle) -- frozen, can't be changed
```

Kind is tag

Mappings

- Arrays
  - Static - *location & size frozen at compile time* (FORTRAN)
  - Semi-static - *size bound at compile time, location at invocation* (Pascal, C)
  - Dynamic - *size and location bound at creation* (ALGOL 60, Ada, Java)
  - Flex - *size and location can be changed any time* (Java vectors)
- Function Types - *update less efficient*
  - update f arg nuVal = fn x => if x = arg then nuVal else f x

Recursive Types

- In Haskell: data List = Nil | Cons (Integer, List)
- In C: struct list { int x; list *next; };
- Solutions to: list = { Nil } ∪ (int × list)
  1. finite seqs of ints followed by Nil: e.g., (2,(5,Nil))
  2. finite or infinite seqs: if finite then end w/ Nil
- Recursive eqn's always have a least solution
  - least fixed point!

Least Recursive Solutions

$$\begin{aligned}
list_0 &= \{Nil\} \\
list_1 &= \{Nil\} \cup (int \times list_0) \\
&= \{Nil\} \cup \{(n, Nil) | n \in int\} \\
list_2 &= \{Nil\} \cup (int \times list_1) \\
&= \{Nil\} \cup \{(n, Nil) | n \in int\} \cup \{(m, (n, Nil)) | m, n \in int\} \\
&\ldots \\
list &= \bigcup_n list_n
\end{aligned}$$

Some solutions inconsistent w/classical math!

User-Defined Types

- Named types
  - More readable
  - Easy to modify if localized
  - Factorization (why repeat same def?)
  - Added consistency checking if generative
- Enumeration types added to Java 5

What does it mean for a language to be type-safe?

Safe Languages

- Two kinds of execution errors
  - Trapped errors: cause computation to halt immediately.
    - Divide by zero, null pointer exception
  - Untrapped errors: go unnoticed and later cause problems.
    - Access an illegal address, e.g., array bounds error.
- Program fragment is *safe* if it causes no untrapped errors.
  - Language is safe if all program fragments are safe.

See "Type Systems" by Luca Cardelli
http://lucacardelli.name/Papers/TypeSystems%201st%20Edition.US.pdf

Strongly Typed Languages

- Language designates *forbidden* errors
  - those that are not allowed to happen.
  - should include all untrapped errors
- Program fragment is *well behaved* if it generates no forbidden errors.
- Language where all legal programs are well behaved is *strongly typed*

Static vs. Dynamic Typing

- Most use static typing
  - including C/Java/ML/Haskell
  - binding of types to variables done at translation time.
  - Find errors earlier, but conservative.
- dynamic typing
  - LISP/Scheme/Racket/Python/Javascript/Grace
  - binding of type to value, not variable.
    - thus binding of type to variable changes dynamically
  - Dynamic more flexible, but more overhead.

(Static) Type Checking

Static Type Checking

- Static type-checkers for strongly-typed languages (i.e., rule out all "bad" programs) must be conservative:
  - Rule out some programs without errors.
- if (*program-that-could-run-forever*) {
     expression-w-type-error;
  } else {
     expression-w-type-error;
  }