Untyped Lambda Calculus (II)

Recall: Call-by-value O.S.

- Basic rule

$$\frac{}{(\backslash x.e)\ \ v \rightarrow e\ [v/x]}$$

- Search rules:

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

Quiz: Write the rules for Right-to-Left call-by-value O.S.?

Call-By-Value Evaluation Example

(\x. x x) (\y. y)

→ x x [\y. y / x]

= (\y. y) (\y. y)

→ y [\y. y / y] Note y is free in the body of \y.y, i.e., y!

= \y. y

Another example

(\x. x x) (\x. x x)

→ x x [\x. x x/x]

= (\x. x x) (\x. x x)

- In other words, it is simple to write non-terminating computations in the lambda calculus
- what else can we do?

We can do everything

- The lambda calculus can be used as an "assembly language"

- We can show how to compile useful, high-level operations and language features into the lambda calculus
  - Result = adding high-level operations is convenient for programmers, but not a computational necessity
    - *Concrete* syntax vs. *abstract* syntax
    - "Syntactic sugar"
  - Result = lambda calculus makes your compiler intermediate language simpler

BOOLEANS

- we can encode booleans
- we will represent "true" and "false" as functions named "tru" and "fls"
- how do we define these functions?
- think about how "true" and "false" can be used
- they can be used by a testing function:
  - "test b then else" returns "then" if b is true and returns "else" if b is false
  - i.e., test tru then else →* then; test fls then else →* else
  - the only thing the implementation of test is going to be able to do with b is to apply it
  - the functions "tru" and "fls" must distinguish themselves when they are applied

BOOLEANS

tru = \t.\f. t   fls = \t.\f. f

test = \x.\then.\else. x then else

Remember applications are left associative: (((test tru) a) b)

- E.g. (underlined are redexes):

  test tru a b

= (\x.\then.\else. x then else) tru a b

→ (\then.\else. tru then else) a b

→ (\else. tru a else) b

→ tru a b

= (\t.\f. t) a b

→ (\f. a) b

→ a

BOOLEANS

tru = \t.\f. t fls = \t.\f. f

and = \b.\c. b c fls

and tru tru

→* tru tru fls

→* tru

(→* stands for multi-step evaluation)

BOOLEANS

tru = \t.\f. t   fls = \t.\f. f

and = \b.\c. b c fls

and fls tru

→* fls tru fls

→* fls

What will be the definition of “or” and “not”?

Booleans

tru = \t.\f. t fls = \t.\f. f

or = \b.\c. b tru c

or fls tru

→* fls tru tru

→* tru

or fls fls

→* fls tru fls

→* fls

Quiz: Step-by-step, evaluate **or tru fls**?

PAIRS

pair = \f.\s.\b. b f s /*pair is a constructor: pair x y*/

fst = \p. p tru /* returns the first of a pair */

snd = \p. p fls /* returns the second of a pair */

fst (pair v w)

= fst ((\f.\s.\b. b f s) v w)

→ fst ((\s.\b. b v s) w)

→ fst (\b. b v w)

= (\p. p tru) (\b. b v w)

→(\b. b v w) tru

→ tru v w /* tru = \t.\f. t */

→* v

AND WE CAN GO ON…

- numbers
- arithmetic expressions (+, -, *,…)
- lists, trees and datatypes
- exceptions, loops, …
- …
- the general trick:
  - values will be functions – construct these functions so that they return the appropriate information when called by an operation （applied by another function)

Quiz:

Suppose the numbers can be encoded in lambda calculus as:

0= \f. \x. x

1 = \f. \x. f x

2 = \f. \x. f (f x)

…

Define succ in lambda calculus such that

succ 0 →* 1

succ 1 →* 2

…

Simply-Typed Lambda Calculus

Simply Typed Lambda-Calculus

- Goal: construct a similar system of language that combines the pure lambda-calculus with the basic types such as bool and num.

- A new type: → (arrow type)
- Set of simple types over the type bool is

$$t ::= \text{bool} \mid t_1 \to t_2$$

- Note: type constructor → is right associative:
  - t1 → t2 → t3 == t1 → (t2 → t3)

Syntax (I)

e ::=		expressions:
	x	(variable)
|	true	(true value)
|	false	(false value)
|	if e1 then e2 else e3	(conditional)
|	\x : t . e	(abstraction)
|	e1 e2	(application)

v ::=		values:
	true	(true value)
|	false	(false value)
|	\x : t . e	(abstraction value)

SYNTAX (II)

t ::=		types:
	bool	(base Boolean type)
|	$t_1 \rightarrow t_2$	(type of functions)
Γ ::=		contexts:
	.	(empty context)
|	Γ, x: t	(variable-type binding)

Γ is a sequence of variable-type binding, which can also be thought of as a functional mapping between x and t.

Typing Rules

- The type system of a language consists of a set of inductive definitions with judgment form:

$$\Gamma \vdash e : t$$

  - "If the current typing context is Γ, then expression *e* has type *t*."
  - This judgment is known as *hypothetical judgment* (Γ is the hypothesis).
  - Γ (also written as "G") is a typing context (type map) which is mapping between *x* and *t* of the form *x*: *t*
  - *x* is the variable name appearing in *e*
  - *t* is a type that's bound to *x*

EVALUATION (O.S.)

[e → e']

$$\frac{e_1 \to e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \to \text{if } e_1' \text{ then } e_2 \text{ else } e_3} \text{ (E−if0)}$$

$$\frac{}{\text{if } true \text{ then } e_2 \text{ else } e_3 \to e_2} \text{ (E−if1)}$$

$$\frac{}{\text{if } false \text{ then } e_2 \text{ else } e_3 \to e_3} \text{ (E−if2)}$$

$$\frac{e_1 \to e_1'}{e_1 \;\; e_2 \to e_1' \;\; e_2} \text{ (E−App1)}$$

$$\frac{e_2 \to e_2'}{v_1 \;\; e_2 \to v_1 \;\; e_2'} \text{ (E−App2)}$$

$$\frac{}{(\lambda x{:}\,t.\;\; e) \;\; v \to e[v/x]} \text{ (E−AppAbs)}$$

Typing

$[\Gamma \vdash e : t]$

$$\frac{x{:}t \in \Gamma}{\Gamma |\!- x{:}t}$$ (T-Var)

$$\frac{}{\Gamma |\!- true : bool}$$ (T-True)

$$\frac{}{\Gamma |\!- false : bool}$$ (T-False)

This is the only place Γ can be extended: may need to alpha rename x so that x is distinct from any vars bound in Γ

$$\frac{\Gamma |\!- \ e_1 : bool \quad \Gamma |\!- \ e_2 : t \quad \Gamma |\!- \ e_3 : t}{\Gamma |\!- \ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$ (T-If)

$$\frac{\Gamma, x : t_1 |\!- e_2 : t_2}{\Gamma |\!- \lambda x : t_1. \ \ e_2 : t_1 \to t_2}$$ (T-Abs)

$$\frac{\Gamma |\!- e_1 : t_{11} \to t_{12} \qquad \Gamma |\!- e_2 : t_{11}}{\Gamma |\!- e_1 \ \ e_2 : t_{12}}$$ (T-App)

Properties of Simply-Typed Lambda Calculus

Lemma 1 (Uniqueness of Typing). For every typing context Γ and expression e, there exists ***at most*** one *t* such that Γ |-- *e : t.*

(note: we don't consider sub-typing here)

Proof:

By induction on the derivation of Γ |- e : t.

Case t-var: since there's at most one binding for x in Γ, x has either no type or one type t. Case proved

Case t-true and t-false: obviously true.

Case t-if: $$\frac{\Gamma| -\ e_1 : bool \quad \Gamma| -\ e_2 : t \quad \Gamma| -\ e_3 : t}{\Gamma| -\ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$

(1) t is unique (By I.H.)

Case proved.

Properties of Simply-Typed Lambda Calculus

Case t-abs: $\dfrac{\Gamma, x: t_1 | - e_2: t_2}{\Gamma | - \lambda x: t_1.\ e_2: t_1 \to t_2}$

(1) t_2 is unique (By I.H.)

(2) Γ contains just one (x, t) pair so t_1 is unique (By (1) and assumption of t-abs)

(3) t1 → t2 is unique (By (2) and t-abs)

Case t-app: $\dfrac{\Gamma | - e_1: t_{11} \to t_{12} \qquad \Gamma | - e_2: t_{11}}{\Gamma | - e_1\ \ e_2: t_{12}}$

(1) e_1 and e_2 satisfies Lemma 1 (By I.H.)

(2) There's at most one instance of t_{11} (By (1))

(3) t_{12} is unique, too (By (2) & I.H.)

Properties of Simply-Typed Lambda Calculus

Lemma 2 (Inversion for Typing).

- If $\Gamma \vdash x : t$ then $x : t \in \Gamma$
- If $\Gamma \vdash (\lambda x : t_1.e) : t$ then there is a t_2 such that
  $$t = t_1 \rightarrow t_2 \text{ and } \Gamma, x : t_1 \vdash e : t_2$$
- If $\Gamma \vdash e_1\ e_2 : t$ then there is a t' such that
  $$\Gamma \vdash e_1 : t' \rightarrow t \text{ and } \Gamma \vdash e_2 : t'$$

Proof:

From the definition of the typing rules, there is only one rule for each type of expression, hence the result.

- **Well-typedness**: An expression e in the language L is said to be *well-typed*, if there exists some type t, such that $e : t$.

Properties of Simply-Typed Lambda Calculus

Canonical Forms Lemma

(Idea: Given a type, want to know something about the shape of the value)

If . |- v: t then

If t = bool then v = true or v = false;

If $t = t_1 \rightarrow t_2$ then $v = \backslash x{:}\ t_1.\ e$

Proof:

By inspection of the typing rules.

Properties of Simply-Typed Lambda Calculus

Exchange Lemma

If G, $x:t_1$, $y:t_2$, G' |- e:t,

then G, $y:t_2$, $x:t_1$, G' |- e:t.

Proof by induction on derivation of

G, $y:t_1$, $x:t_2$, G' |- e:t

(Homework!)

Weakening Lemma

If G |- e:t then G, x:t' |- e:t (provided x not in Dom(G))

(Homework!)

Type Safety of a Language

- Safety of a language = Progress + Preservation

- Progress: A well-type term is not stuck (either it is a value or it can take a step according to the evaluation rules)

- Preservation: If a well-typed term (with type *t*) takes a step of evaluation, then the resulting term is also well typed with type *t*.

- **Type-checking**: the process of verifying *well-typedness* of a program (or a term).

PROGRESS THEOREM

- Suppose e is a closed and well-typed term (that is e : t for some t). Then either e is a value or else there is some e' for which e → e'.

Proof: By induction on the derivation of typing: [$\Gamma \vdash e : t$]

Case T-Var: doesn't occur because e is closed.

Case T-True, T-False, T-Abs: immediate since these are values.

Case T-App:

(1) e_1 is a value or can take one step evaluation. Likewise for e_2. (By I.H.)

(2) If e_1 can take a step, then E-App1 can apply to (e_1 e_2). (By (1))

(3) If e_2 can take a step, then E-App2 can apply to (e_1 e_2) (By (1))

(4) If both e_1 and e_2 are values, then e1 must be an abstraction, therefore E-AppAbs can apply to (e_1 e_2) (By (1) and canonical forms v)

(5) Hence (e1 e2) can always take a step forward. (By (2,3,4))

Progress Theorem (Cont'd)

Case T-if:

1. e1 can either take a step or is a value (By I.H.)
2. Subcase 1: e1 can take a step (By I.H.)
   1. if e1 then e2 else e3 can take a step (By E-if0)
3. Subcase 2: e1 is a value (By I.H.)
   1. If e1 = true, if e1 then e2 else e3 → e2 (By E-if1)
   2. If e1 = false, if e1 then e2 else e3 → e3 (By E-if2)
4. In both subcases, e can take a step. Case proved.

Preservation Theorem

- If G |- e : t and e → e', then G |- e' : t.

Proof: By induction on the derivation of G|- e : t.

Case T-Var, T-Abs, T-True, T-False:

Case doesn't apply because variable or values can't take one step evaluation.

Case T-If: e = if e1 then e2 else e3.

If e → e' there are two subcases cases:

Subcase 1: e1 is not a value.

(1) e1 : bool (By assumption and invesion of T-if)

(2) e1 → e1' and e1' : bool (By IH)

(3) G |- e' : t (By T-If and (2))

Subcase 2: e1 is a value, i.e. either true or false.

(4) e → e2 or e → e3 and e' : t (e'=e2 or e3) (By E-If1, E-If2 and IH)

Case proved.

Preservation Theorem (Cont'd)

Case T-App: $e = e_1\ e_2$. Need to prove, $G \mid\!-\ e' : t_{12}$

If e_1 is not a value then:

(5) $e_1 \rightarrow e_1'$, and $e_1' : t_{11} \rightarrow t_{12}$. (By IH)

(6) $e_1'\ e_2 : t_{12}$ (By T-App)

If e_1 is a value then:

(7) e_1 is an abstraction. (By assumption and T-Abs)

There are two subcases for e_2.

Subcase 1: e_2 is a value. Let's call it v.

(8) $e = \backslash x\ .\ e''\ \ v$, and

$G \mid\!-\ \backslash x.e'' : t_{11} \rightarrow t_{12},$ (By assumption of T-App)

$G, x: t_{11} \mid\!-\ e'' : t_{12},$

$G \mid\!-\ v : t_{11}$ (By (7) and inversion of T-Abs)

(9) $\backslash x.\ e''\ v \rightarrow e''\ [v / x]$ (By E-AppAbs)

(10) $G \mid\!-\ e''[v / x] : t_{12}$. (By (8), (9) and **substitution lemma**)

(11) $G \mid\!-\ e' : t_{12}$ (By (10) & assumption)

Subcase 2: e_2 is not a value.

(12) Suppose $e_2 \rightarrow e_2'$. Then $e \rightarrow e_1\ e_2'$, i.e., $e' = e_1\ e_2'$. (By E-App2)

(13) $G \mid\!- e_2' : t_{11}$ (By I.H., T-App)

(14) $G \mid\!- e_1\ e_2' : t_{12}$. (By (13))

(15) $G \mid\!- e' : t_{12}$. (By (12) & (14))

Case proved.

QED.

Substitution Lemma

If G, x : t' |- e : t, and G |- v : t', then G |- e [v / x] : t.

Proof left as an exercise.

Curry-Howard Correspondence

- A.k.a *Curry-Howard Isomorphism*
- Connection between type theory and logic

Logic	Programming Languages
Propositions	Types
Proposition $P \supset Q$	Type P→ Q
Proposition $P \wedge Q$	Type P × Q (product/pair type)
Proof of proposition P	Expression e of type P
Proposition P is provable	Type P is inhabited (by some expression)