CS 131

Typed Lambda Calculus

Types and Terms

- Types:
  - t ::= bool | t1 -> t2
- Terms:
  - e ::= x | e1 e2 | lambda x:t. e | true | false | if e1 e2 e3
- Values ⊆ Terms
  - lambda x:t. e is a value
  - true is a value
  - false is a value

Computation Rules

for call-by-value lambda calculus

```
--------------------
if true e2 e3 --> e2
```

```
---------------------
if false e2 e3 --> e3
```

```
           e1 --> e1'
----------------------------
if e1 e2 e3 --> if e1' e2 e3
```

```
          v2 is a value
--------------------------------
(lambda x:t. e1) v2 --> e1[v2/x]
```

```
   e1 --> e1'
----------------
e1 e2 --> e1' e2
```

```
v1 is a value    e2 --> e2'
---------------------------
     v1 e2 --> v1 e2'
```

Stuck terms

- Can’t be reduced to a value
  - Example: true (lambda x:bool. x)
  - Normal forms that are not values!
  - Should be illegal!

Typing-Context

Provides context for determining types of expressions!

Γ ::= empty | Γ; x: t

lookup(Γ; x:t, x) = t
lookup(Γ; y:t, x) = lookup(Γ, x)

Type-Checking Rules

```
lookup(Γ,x) = t
---------------
Γ |- x : t
```

```
----------------
Γ |- true : bool
```

```
-----------------
Γ |- false : bool
```

```
Γ |- e1 : bool    Γ |- e2 : t    Γ |- e3 : t
---------------------------------------------
Γ |- if e1 e2 e3 : t
```

```
Γ; x:t1 |- e : t2
------------------------------
Γ |- lambda x:t1. e : t1 -> t2
```

```
Γ |- e1 : t1 -> t2    Γ |- e2 : t1
-----------------------------------
Γ |- e1 e2 : t2
```

Example: *type check* (lambda x:bool. if x false true) true

Automate

- Represent typed lambda-calculus expressions in Haskell
- Convert type-checking rules to function that
  - Given Γ, expression, returns type
- Can do same with interpreter
  - Given expression, return normal form
  - See Haskell code

Copy & Paste

:load /Users/kim/typeChecker.hs

let myMap = Data.Map.empty

typeOf myMap (Lam "x" TBool (Var “x"))

typeOf myMap (App (Lam "x" TBool (If (Var "x") (Bool False) (Bool True))) (Bool True))

What's the Connection?

- What do computation rules and type-checking rules have to do with each other?
  - Rule out programs that don't type check.
    - What does that tell us about computation?
    - If e has type τ and e —> e' then what about type of e'?

Soundness

- Theorem (Type soundness). If $\vdash e: \tau$ and $e \rightarrow^* e'$ and e cannot be further reduced, then e' is a value and $\vdash e' : \tau$
- Follows from two lemmas:
  - Lemma (Preservation). If $\vdash e: \tau$ and $e \rightarrow e'$ then $\vdash e' : \tau$.
    - Proved by induction on length of computation.
  - Lemma (Progress). If $\vdash e: \tau$ then either e is a value or there exists an e' such that $e \rightarrow e'$

Even Better!

- Normalization
  - Theorem. If $\vdash e: \tau$ then there exists a value v such that $e \to_* v$.
  - Every program in typed lambda calculus terminates!
- Is that good or bad?
  - Remember halting problem!

Recursion?

- Can’t write non-terminating computations!
  - Ω = (λx. x x) (λx. x x) is not typeable
    - What could type of λx. x x be?
  - Recursion has to be explicitly added (and then get non-termination).
  - But type-checking is decidable, even w/recursion!

Adding Recursion

- New term: fix e — *represents fixed point of function e.*
  - *FACT = fix λf :int → int. λn:int. if n = 0 then 0 else n × (f (n – 1))*
- Computation rule:
  - fix λx: τ. e → e{(fix λx: τ. e)/x}
- Type checking:

$$\frac{\Gamma \vdash e : t \rightarrow t}{\Gamma \vdash \text{fix } e : t}$$

-