

Types

- Type systems
- Simply-typed lambda calculus
- Type judgments



Type systems

- Types are used to specify properties of a program
 - *Normally, types specify classes of normal forms*
 - *Numbers, functions, pairs, etc.*
 - *If a program has type int, and it has a normal form, the normal form is a number*
- Type systems contain:
 - *A set of types*
 - *A set a programs*
 - *A type judgment*
 - *Typing may or may not be decidable*



Simply-typed lambda-calculus

- Types are “simple,” meaning not polymorphic
- We start with a set of base types, and a type of functions

$$t ::= \bullet \quad (\text{or } \mathbb{N}, \mathbb{Z}, \mathbb{B}, \dots)$$
$$| t \rightarrow t \quad (\text{functions})$$


Programs with types

- Next, we augment the programs so that they include types
- There are two ways to do this
 - *Define the types completely, everywhere*
 - *Define only as much as necessary*
- Verbose typing:

$$\begin{array}{l} e ::= v : t \\ \quad | (e_1 e_2) : t \\ \quad | (\lambda x : t_1. e) : (t_1 \rightarrow t_2) \end{array}$$



Programs with minimal typing info

- The programs contain just enough type information so that verbose typing can be inferred
- Concise types
 - *Add a type constraint to function parameters*

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


Evaluation

- Ignore the types

$$(\lambda x:t.e_1) e_2 \rightarrow_{\beta} e_1[e_2/x]$$

Define multi-step reduction \rightarrow_{β}^* in the normal way (**not** symmetric).



Judgments

- What programs have which types?
- To determine this, we need to define a typing *judgment*



Judgments

- The basic judgment is a **sequent**, “the program e has type t in context Γ ”:

$$\Gamma \vdash e : t$$

- The “ \vdash ” symbol is called a **turnstile**.
- Γ is a **type assignment**, $x_1 : t_1, \dots, x_n : t_n$



Typing rules

- Type judgments are defined by inference rules (just like we used for specifying natural semantics)
- We need three rules
 - *Variables*
 - *Applications*
 - *Abstractions (functions)*



Typing rules

$$\frac{}{\Gamma, x:t \vdash x : t} \text{ var}$$

$$\frac{\Gamma \vdash e_1 : (s \rightarrow t) \quad \Gamma \vdash e_2 : s}{\Gamma \vdash (e_1 e_2) : t} \text{ app}$$

$$\frac{\Gamma, x:s \vdash t}{\Gamma \vdash (\lambda x:s.e) : (s \rightarrow t)} \text{ abs}$$



Notes

- These rules are *templates*, Γ stands for an arbitrary context, x is an arbitrary variable, e_1, e_2 are arbitrary programs, and s, t are arbitrary types.
- A program e has type t if it has that type in an empty context.

$$\vdash e : t$$



Some examples

Identity combinator I :

$$\frac{\overline{x:t \vdash x:t}^2}{\vdash (\lambda x:t.x) : (t \rightarrow t)}^1$$



K combinator

K combinator:

$$\frac{\frac{\frac{\overline{x:s, y:t \vdash x:s}^3}{x:s \vdash (\lambda y:t.x) : (t \rightarrow s)}^2}{\vdash (\lambda x:s.\lambda y:t.x) : (s \rightarrow t \rightarrow s)}^1$$



Another example

A bogus term $\lambda x:t.xx$:

$$\frac{\frac{\frac{x:(s_1 \rightarrow s_2) \vdash xx : ???}{x:t \vdash xx : ???} \quad 2}{\vdash (\lambda x:t.xx) : (t \rightarrow ???)} \quad 1}{x:(s_1 \rightarrow s_2) \vdash x : (s_1 \rightarrow s_2) \quad x:(s_1 \rightarrow s_2) \vdash x : s_1} \quad 3$$



Typed programming

- Type systems are usually used for sanity
 - *Normally, a well-typed program will never core dump*
 - *The type-system for C is not sound*
- Type systems are designed for static checking
- Type checking is computable, usually in linear time
- The price:
 - *Some sensible programs are rejected*
- Limitations:
 - *Doesn't prove termination*
 - *Doesn't guarantee absence of divide-by-zero, array bounds violations, etc.*
 - *These are usually caught as runtime exceptions*



What this buys us

- “Well-typed programs do not get stuck”
- Two parts:
 - *Preservation: the type of a program does not change as it is evaluated*
 - *Progress: if a program has a type, and it is not a value, it can be reduced further*
- We’ll need some additional properties



Some notes

- We'll use both colon and epsilon notation; they mean the same thing (its just convention)
- A sequent has two parts, a type assignment “ $x_1:t_1, x_2:t_2, \dots, x_n:t_n$ ” before the turnstile, and a type judgment “ $e \in t$ ” after
- For now, the order in the type assignment doesn't matter, but we'll still consider it to be a list

$$x_1:t_1, x_2:t_2, \dots, x_n:t_n \vdash e \in t$$



Outline

- We'll prove:
 - *All well-typed programs are closed*
 - *Every program has at most one type*
 - *Substitution lemma*
 - *Preservation*
 - *Progress*



Structural induction

Structural induction is a method for proving properties of all λ terms.

The induction principle is: assume that the induction principle holds for all smaller terms, and prove it for the term.

There are three cases to prove a property $P(e)$:

var Prove $P(x)$ (base case)

abs Prove $P(\lambda x.e)$ from $P(e)$

app Prove $P(e_1 e_2)$ from $P(e_1)$ and $P(e_2)$



All well-typed programs are closed

If $\Gamma \vdash e \in t$ and x does not appear in Γ , then x is not free in e .

var if e is a var, trivial.

abs if $e = \lambda y:t_1.e'$, then $t = t_1 \rightarrow t_2$, and

$$\frac{\Gamma, y:t_1 \vdash e' \in t_2}{\Gamma \vdash (\lambda y:t_1.e') \in (t_1 \rightarrow t_2)} \text{ abs}$$

If $x \neq y$ then then x is not free in e' by induction.

app if $e = e_1 e_2$, then $\Gamma \vdash e_1 \in t' \rightarrow t$ and $\Gamma \vdash e_2 \in t'$, for some t' . By induction x is not free in e_1 or e_2 .



Every program at most one type

If $\Gamma \vdash e \in s$ and $\Gamma \vdash e \in t$ then $s \equiv t$.

var if e is a var, trivial.

abs if $e = \lambda y:u.e'$, then $s = u \rightarrow s'$ and $t = u \rightarrow t'$,
and $\Gamma, y:u \vdash e' \in s'$, and $\Gamma, y:u \vdash e' \in t'$. By
induction $s' \equiv t'$.

app if $e = e_1 e_2$, then e_1 and e_2 have unique types by
induction, so e does too.



Substitution

If $\Gamma, x:s, \Delta \vdash e_1 \in t$, and $\Gamma, \Delta \vdash e_2 \in s$, then $\Gamma, \Delta \vdash e_1[e_2/x] \in t$.

var if e is a var, trivial.

app if $e = e_3 e_4$, result follows by induction.

abs if $e = \lambda y:u.e_3$, then $t = u \rightarrow v$, and $\Gamma, x:s, \Delta, y:u \vdash e_3 \in v$, so $\Gamma, \Delta, y:u \vdash e_3[e_2/x]:v$ by induction.



Subject-reduction (preservation)

Theorem (Preservation) If $e_1 \rightarrow_{\beta}^* e_2$ and $\Gamma \vdash e_1 \in t$ then $\Gamma \vdash e_2 \in t$.

Proof

- Substitution is the key lemma
- Reduction
 - $(\lambda x:t.e_1) e_2 \rightarrow_{\beta} e_1[e_2/x]$
 - ignores types
 - can be used in any context



Subject-reduction proof

- Use induction to reduce to a single β step.
- Show if $\Gamma \vdash (\lambda x:t_1.e_1) e_2 \in t_2$, then $\Gamma \vdash e_1[e_2/x] \in t_2$.
- By typing rules, $\Gamma \vdash (\lambda x:t_1.e_1) \in (t_1 \rightarrow t_2)$, and $\Gamma \vdash e_2 \in t_1$.
- Then $\Gamma, x:t_1 \vdash e_1 \in t_2$,
- So $\Gamma \vdash e_1[e_2/x] \in t_2$ by substitution lemma.



Progress

Theorem (Progress) If $e : t$ and e is not a value (it is not in normal form), then there is an e' such that $e \rightarrow_{\beta} e'$.

Proof If e is not a value, it must either:

- Contain a redex, but then it can be reduced.
- Have weak head normal form $\lambda e_1 e_2 \cdots e_n$, where e_1, \dots, e_n are values. Not possible, because this program is not closed.

