CS 565

Programming Languages (graduate)
Spring 2024

Week 8

Type Systems

Types

Today

- Identify key concepts in type systems:
- Type systems as inductive relations
- Type safety

Ill-Typed Imp$^+$

- Let’s weaken IMP’s expression language slightly:

```
e  ::= B | N | e * e | e + e
     | true | false | ¬ e | e ∧ e
     | Id | e = e | e < e | e ? e : e
```

- Looks good, we can now write (and evaluate):

```
x * ((y > 3) ? 3 : y)
```

- But we can also write:

```
x * ((3 + (6 ∧ 5)) ? 3 : y)
```

– How do we evaluate this? What’s the problem?

Bad Behaviors

- What constitutes a "bad" expression in our IMP variant?
  - One that adds two booleans: `true + 3` ⟶ ?
  - One with a non-boolean conditional: `3 ? x : y` ⟶ ?
  - A use of an unassigned variable: `x + y` ⟶ ?
- What about Coq?
  - Bad pattern match discriminees: `match 0 with [ ] -> …`
  - Function applied to wrong argument types: `plus 9 minus`
  - Application of non-function: `9 minus`

What about other languages?

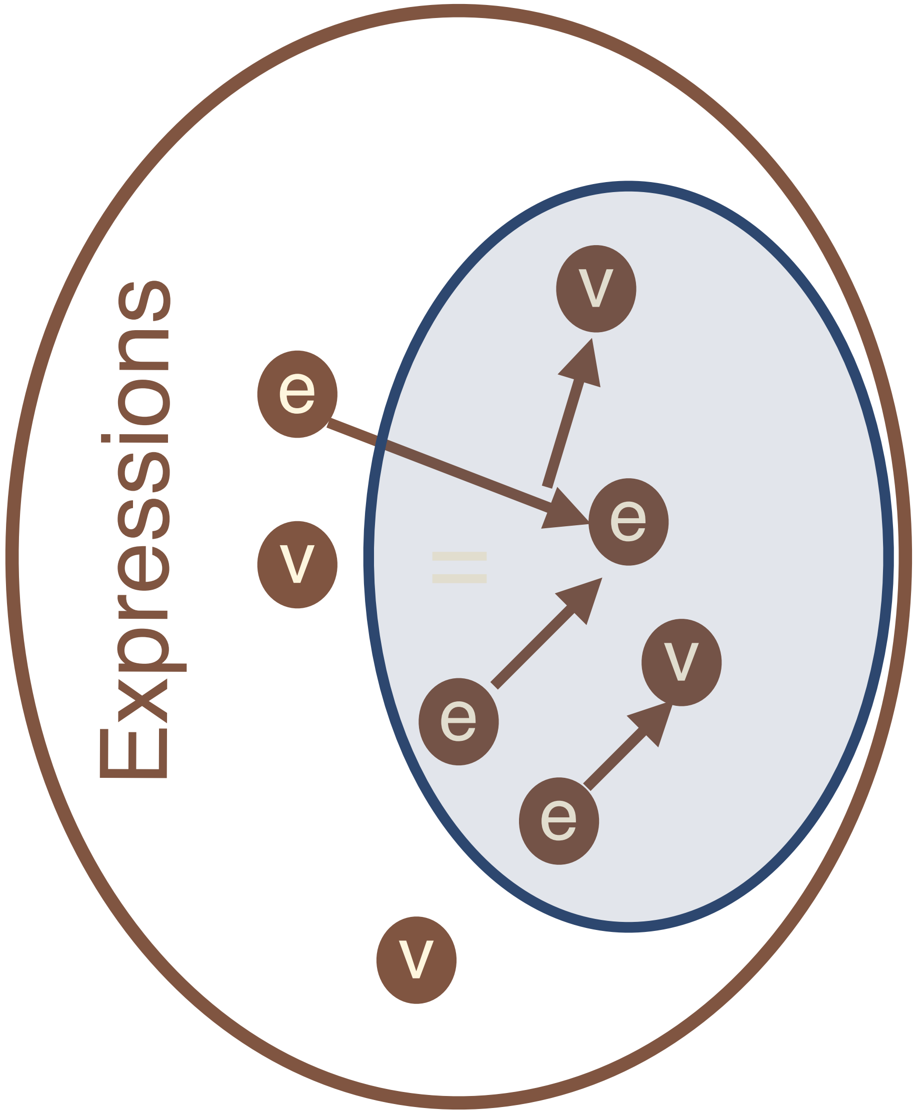
Badness is language specific!

Static Semantics

A recipe for defining a language:

1. Syntax:
   - What are the valid expressions?
2. Semantics (Dynamic Semantics):
   - How do I evaluate valid expressions?
3. Sanity Checks (Static Semantics):
   - What expressions are "good", i.e have meaningful evaluations?

Type systems identify a subset of good expressions



Typing Imp$^+$

A recipe for type systems:

1. Define bad programs
2. Define typing rules for classifying programs
3. Show that the type system is sound, i.e. that it only identifies good programs

Typing Imp$^+$

A recipe for type systems:

1. **Define bad programs**
2. Define typing rules for classifying programs
3. Show that the type system is sound, i.e. that it only identifies good programs

Typing Imp$^+$

- First step is to define badness:
  - Needs to be broad, program-independent properties
  - Some user-provided specification is okay (type annotations)
- What are bad Imp expressions?

```
3 ? true : 4

true + 3

x * ((y > 3) ? 3 : y)
```

- Those that evaluate to a stuck expression: a normal form that isn't a value

Typing Imp$^+$

- First step is to define badness:
  - Needs to be broad, program … erties
  - Some user-pr…

    annot…
- What ar…

$$x * ((y > 3) ? 3 : y)$$

- Those that evaluate to a stuck expression: a normal form that isn't a value

“Well-typed programs cannot go wrong”

A Theory of Type Polymorphism in Programming (Milner 78)

Typing Imp$^+$

A recipe for type systems:

1. Define bad programs
2. **Define typing rules for classifying programs**
3. Show that the type system is sound, i.e. that it only identifies good programs

Typing Rules

Next, define a classifier for good, well-formed programs:

$$\vdash e : T$$

Goal is to classify good uses of each type of expression:

$$\frac{n \in N}{\vdash \mathbf{n} : \text{nat}}\ \text{T}\textsc{Num} \qquad \frac{\vdash e_1 : \text{nat} \qquad \vdash e_2 : \text{nat}}{\vdash e_1 + e_2 : \text{nat}}\ \text{T}\textsc{Add}$$

$$\frac{}{\vdash x : \text{nat}}\ \text{T}\textsc{Var}$$

$$\frac{\vdash e1 : \text{nat} \qquad \vdash e2 : \text{nat}}{\vdash e_1 * e_2 : \text{nat}}\ \text{T}\textsc{Mult}$$

Typing Rules

Goal is to classify good uses of each type of expression:

$$\frac{}{\vdash \textbf{true} : \text{bool}}\ \text{T}\textsc{True}$$

$$\frac{\vdash e : \text{bool}}{\vdash \neg\, e : \text{bool}}\ \text{T}\textsc{Not}$$

$$\frac{}{\vdash \textbf{false} : \text{bool}}\ \text{T}\textsc{False}$$

$$\frac{\vdash e_1 : \text{bool} \quad \vdash e_2 : \text{bool}}{\vdash e_1 \wedge e_2 : \text{bool}}\ \text{T}\textsc{And}$$

Typing Rules

Goal is to classify good uses of each type of expression:

$$\frac{\vdash e_1 : \mathrm{nat} \qquad \vdash e_2 : \mathrm{nat}}{\vdash e_1 < e_2 : \mathrm{bool}}\ \text{TL}\small{\text{E}}$$

$$\frac{\vdash e_1 : T \qquad \vdash e_2 : T}{\vdash e_1 = e_2 : \mathrm{bool}}\ \text{TE}\small{\text{Q}}$$

$$\frac{\vdash e_1 : \mathrm{bool} \qquad \vdash e_2 : T \qquad \vdash e_3 : T}{\vdash e_1\ ?\ e_2 : e_3 : T}\ \text{TC}\small{\text{OND}}$$

Typing Rules

Goal is to classify good uses of each type of expression:

$$\frac{\vdash e_1 : \text{bool} \qquad \vdash e_2 : T \qquad \vdash e_3 : T}{\vdash e_1 \mathbin{?} e_2 : e_3 : T}\ \textsc{TCond}$$

$$\frac{\vdash e_1 : \text{nat} \qquad \vdash e_2 : \text{nat}}{\vdash e_1 + e_2 : \text{nat}}\ \textsc{TAdd}$$

```
3 ? true : 4
true + 3
⊢ x + ((y > 3) ? true : y)
```

Typing Imp$^+$

A recipe for type systems:

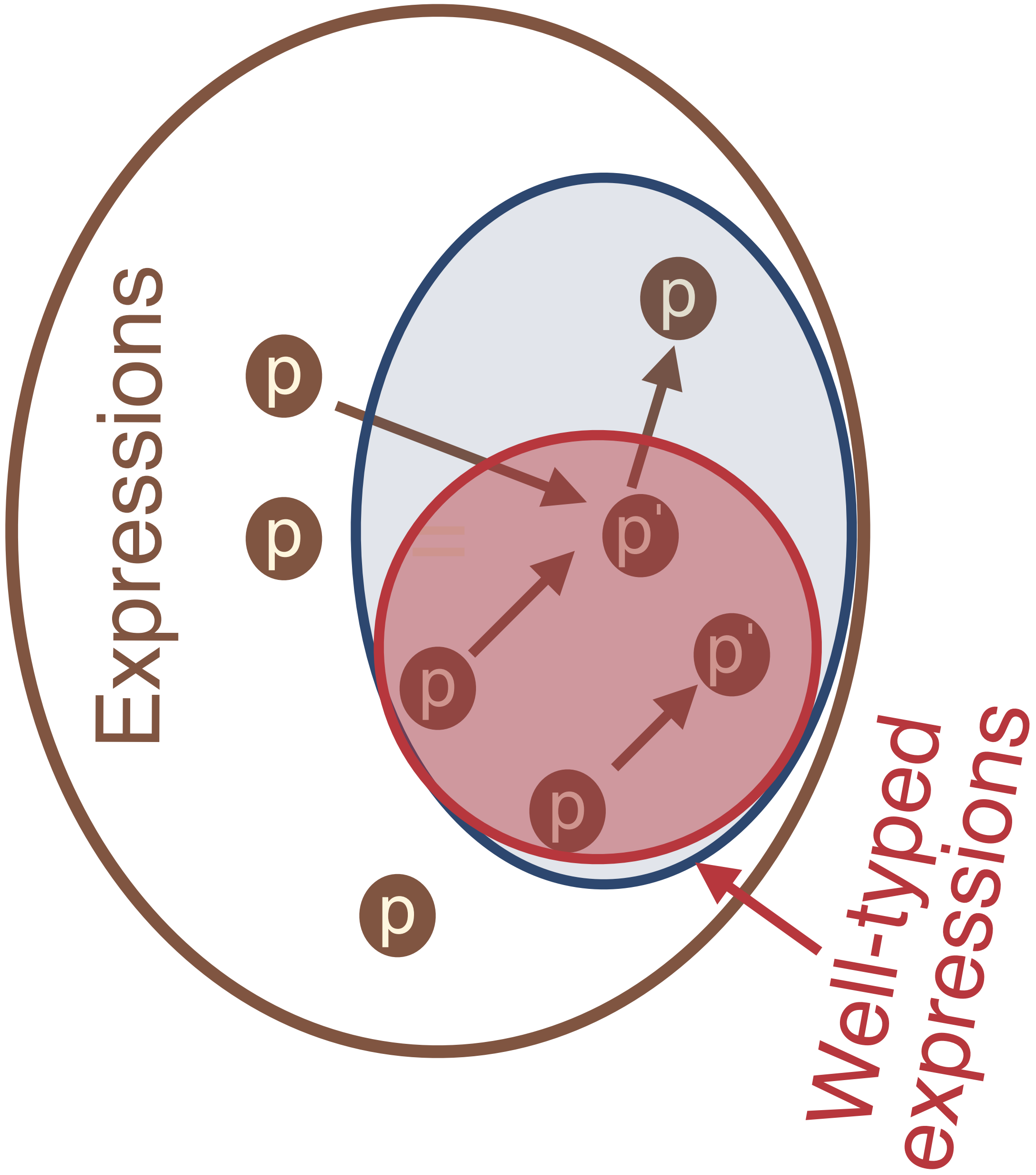
1. Define bad programs
2. Define a typing rules for classifying programs
3. **Show that the type system is sound, i.e. that it only identifies good programs**

Type Safety

- When is a type system correct?
  - Need to show this classification is sound. i.e. no false positives:

$$\vdash e : T \quad \rightarrow \quad \sim e \text{ is bad!}$$

- If the a language's type system is sound, it is said to be type-safe.
- Soundness relates provable claims to semantic property



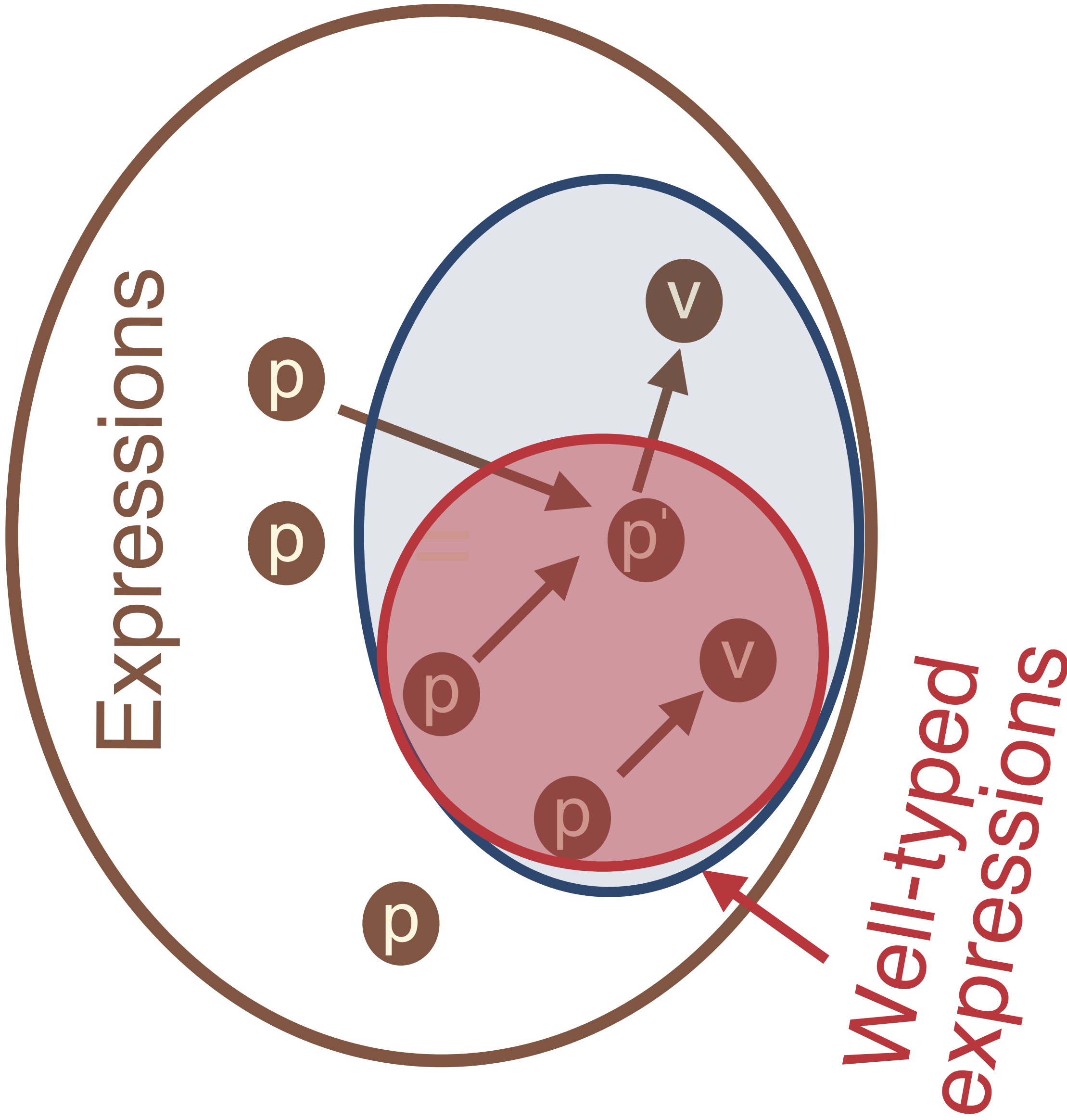
Progress

Theorem [PROGRESS]: Suppose e is a well-typed expression ($\vdash$e:T). Then either e is a value or there exists some e' such that e evaluates to e' (σ, e $\longrightarrow$ e').

Values:

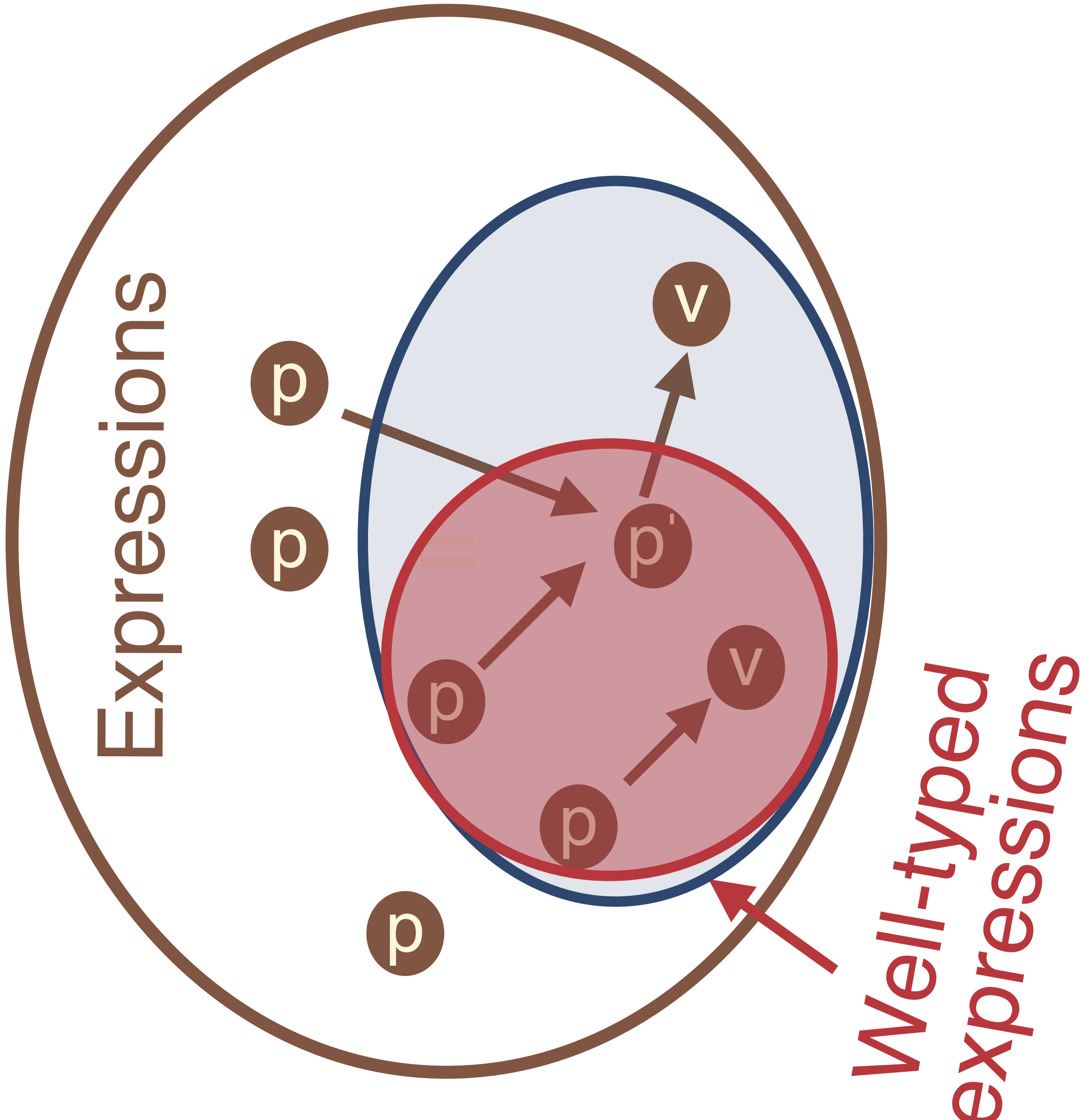
$$\frac{}{\text{value true}}\ \textsc{TValue}$$

$$\frac{n \in N}{\text{value } n}\ \textsc{NumValue}$$



Preservation

- ★ **Theorem** [PRESERVATION]: Suppose e is a well-typed term (⊢ e :T). Then, if e evaluates to e', e' is also a well-typed term under the empty context, with the same type as e (⊢ e' :T).



Type Soundness

Theorem [Type Soundness]: If an expression e has type T, and e reduces to e' in zero or more steps, then e' is not a stuck term.

Proof.

By induction on $\sigma, e \longrightarrow^{*} e'$…

Qed.

- ★ Corollary [Normalization]: If an expression e has type T, e reduces to a value in zero or more steps.

Example

$$\frac{\vdash e_1 : \mathrm{bool} \qquad \vdash e_2 : \mathrm{nat}}{\vdash e_1 + e_2 : \mathrm{nat}}\ \textsc{TBAdd}$$

Example

$$\frac{}{0 \; ? \; e_1 : e_2 \longrightarrow e_1}$$

Example

$$\frac{}{0 \; ? \; e_1 : e_2 \longrightarrow e_1}$$

$$\frac{\vdash e_1 : \text{nat} \qquad \vdash e_2 : T \qquad \vdash e_3 : T}{\vdash e_1 \; ? \; e_2 : e_3 : T} \; \textsc{TCond}_2$$

Recap

- Type systems classify semantically meaningful expressions
- Our recipe for defining a type system
  1. Define bad states (irreducible, non-value expressions)
  2. Define a typing judgement and rules classifying good expressions $(\vdash e : T)$
  3. Show that the type system is sound, i.e. that good expressions don't reduce to bad states

The Limitations of F_1 (simply-typed λ-calculus)

- In F_1 each function works exactly for one type
- Example: the identity function
  - id = $\lambda x{:}\tau.\, x : \tau \to \tau$
  - We need to write one version for each type
  - Even more important: sort : $(\tau \to \tau \to \text{bool}) \to \tau\ \text{array} \to \text{unit}$
- The various sorting functions differ only in typing
  - At runtime they perform exactly the same operations
  - We need different versions only to keep the type checker happy
- Two alternatives:
  - Circumvent the type system (see C, Java, ...), or
  - Use a more flexible type system that lets us write only one sorting function

Polymorphism

- Informal definition

  A function is polymorphic if it can be applied to "many" types of arguments

- Various kinds of polymorphism depending on the definition of "many"
  - subtype (or bounded) polymorphism

    "many" = all subtypes of a given type
  - ad-hoc polymorphism

    "many" = depends on the function

    choose behavior at runtime (depending on types, e.g. sizeof)
  - parametric predicative polymorphism

    "many" = all monomorphic types
  - parametric impredicative polymorphism

    "many" = all types

System F

> The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.
>
> —Mark Manasse.

- System F is a calculus in which polymorphic functions can be written.
  - Name was coined by Jean-Yves Girad, was originally a logic
- A "core" calculus for parametric polymorphism.
  - Used to formalize module systems, approaches to data abstraction
  - Enough for type safe 'pure' OO programming (w/o inheritance)

System F

Here is the syntax , with new bits highlighted.

$t ::= x \mid \lambda x{:}T.t \mid t\ t$
$\quad \mid \Lambda X.t$ ⇐ Type Abstraction
$\quad \mid t\ [T]$ ⇐ Type Application

$v ::= \lambda x{:}T.t \mid \Lambda X.t$

$T ::= T \rightarrow T$
$\quad \mid \forall X.T$ ⇐ Universal Type
$\quad \mid X$ ⇐ Type Variable

Type variables have a different interpretation than before.

Examples

- Examples:
  - $\text{id} = \Lambda X.\lambda x{:}X.\ x \quad : \quad \forall X.X \to X$
  - $\text{id}[\text{int}] = \lambda x{:}\text{int}.\ x \quad : \quad \text{int} \to \text{int}$
  - $\text{id}[\text{bool}] = \lambda x{:}\text{bool}.\ x \quad : \quad \text{bool} \to \text{bool}$
  - "id 5" is invalid. Use "id [int] 5" instead

  - $\text{double} = \Lambda X.\ \lambda f{:}X \to X.\ f\ (f\ x)$
  - $\text{polyf} = \lambda f{:}(\ \forall X.X \to X).\ (f\ [\text{int}]\ 1, f\ \ [\text{bool}]\ \text{True})$

    polyf id

System F

Here are the new bits of the operational semantics

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \textsc{EApp}_1 \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}\ \textsc{EApp}_2$$

$$\frac{}{(\lambda x{:}T.e)\ v \longrightarrow e_1\ [x \mapsto v]}\ \textsc{EAppAbs}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ [T_2] \longrightarrow e_1'\ [T_2]}\ \textsc{ETApp}$$

$$\frac{}{(\Lambda X.e_1)\ [T] \longrightarrow e_1\ [X := T]}\ \textsc{ETAppTAbs}$$

System F

Here are the new bits of the typing rules

$$\frac{\Gamma, [x \mapsto T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2} \quad \text{TABS}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{TVAR}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \quad \text{TAPP}$$

$$\frac{\Gamma, X \vdash t : T_2}{\Gamma \vdash \Lambda X.t : \forall X.T_2} \quad \text{TTABS}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_2}{\Gamma \vdash t_1\ [T_1] : T_2[X \coloneqq T_1]} \quad \text{TTAPP}$$

Observations

- Based on the type of a term we can prove properties of that term
- There is only one value of type $\forall X.X \rightarrow X$

  The polymorphic identity function
- There is no value of type $\forall X.X$
- Take the function: reverse : $\forall X.\ X\ List \rightarrow t\ List$
  * This function cannot inspect the elements of the list
  * It can only produce a permutation of the original list
  * If L_1 and L_2 have the same length and let "match" be a function that compares two lists element-wise according to an arbitrary predicate then

    "match L_1 L_2" $\equiv$ "match (reverse L_1) (reverse L_2)" !

Encoding Base Types in F2

- Booleans
  - $\mathrm{bool} = \forall X.X \to X \to X$ (given any two things, select one)
  - There are exactly two values of this type !
    - $\mathrm{true} = \Lambda X.\ \lambda x{:}X.\lambda y{:}t.\ X$
    - $\mathrm{false} = \Lambda X.\ \lambda x{:}X.\lambda y{:}t.\ X$
  - $\mathrm{not} = \lambda b{:}\mathrm{bool}.\ \Lambda X.\lambda x{:}X.\lambda y{:}X.\ b\ [X]\ y\ x$
- Naturals
  - $\mathrm{nat} = \forall X.\ (X \to X) \to X \to X$ (given a successor and zero element compute a natural number)
  - $0 = \Lambda X.\ \lambda x{:}X \to X.\ \ \lambda z{:}X.\ z$
  - $\mathrm{succ}(e) = \Lambda X.\ \lambda s{:}X \to X.\ \lambda z{:}X.\ s\ (e\ [X]\ s\ z)$
  - $\mathrm{add} = \lambda n{:}\mathrm{nat}.\ \lambda m{:}\mathrm{nat}.\ \Lambda X.\ \lambda s{:}X \to X.\ \lambda z{:}X.\ n\ [X]\ s\ (m\ [X]\ s\ z)$
  - $\mathrm{mul} = \lambda n{:}\mathrm{nat}.\ \lambda m{:}\mathrm{nat}.\ \Lambda X.\ \lambda s{:}X \to X.\ \lambda z{:}X.\ n\ [X]\ (m\ [X]\ s)\ z$

System F Metatheory

System F shares many of STLC's meta-theoretic properties:

Theorem [Progress]: Suppose t is a closed, well-typed term (i.e. $\vdash$ p : T). Then either t is a value or there exists some t' such that t evaluates to t'.

Theorem [Preservation]: Suppose t is a well-typed term under context Γ (i.e. Γ $\vdash$ p : T). Then, if t evaluates to t', t'is also a well-typed term under context Γ, with the same type as t.

Theorem [Normalization]: Suppose t is a closed, well-typed term (i.e. $\vdash$ p : T). Then, t halts, that is there must exist some value v, such that t evaluates to v.

System F Meta-theory

OTOH, the metatheory System F diverges from STLC in key ways with respect to type inference:

$\lceil x \rceil = x$

$\lceil \lambda x{:}T.M \rceil = \lambda x.\lceil M \rceil$

$\lceil M1\ M2 \rceil = \lceil M1 \rceil\ \lceil M2 \rceil$

$\lceil \Lambda X.t \rceil = \lceil t \rceil$

$\lceil t1\ [T2] \rceil = \lceil t1 \rceil$

Theorem [Type Inference is Undecidable]: Suppose m is a closed term in the untyped lambda calculus. Then it is undecidable if there exists some well-typed term system F term, t , such that $\lceil t \rceil = m$.