CMSC 631 – Program Analysis and Understanding
Spring 2009

Type Systems

The Need for a Type System

- Consider the (untyped) lambda calculus
  - false = λx.λy.x
  - 0 (Scott) = λx.λy.x
- Everything is encoded as a function
  - So we can easily misuse combinators
    - false 0 if 0 then ... etc...
  - This is no better than assembly language!

What is a Type System?

- A *type system* is some mechanism for distinguishing good programs from bad
  - Good programs = well typed
  - Bad programs = ill typed or not typable
- Examples:
  - 0 + 1 // well typed
  - false 0 // ill-typed: can't apply a boolean
  - 1 + (if true then 0 else false) // ill-typed: can't add boolean to integer

---

A Definition of Type Systems

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."
>
> – Benjamin Pierce, *Types and Programming Languages*

Simply-Typed Lambda Calculus

- $e ::= n \mid x \mid \lambda x{:}t.e \mid e\ e$
  - Functions include the type of their argument
  - We don’t really need this, but it will come in handy
- $t ::= \text{int} \mid t \to t$
  - $t1 \to t2$ is a the type of a function that, given an argument of type $t1$, returns a result of type $t2$
    - $t1$ is the *domain*, and $t2$ is the *range*

Type Judgments

- Our type system will prove *judgments* of the form
  - $A \vdash e : t$
  - “In type environment A, expression e has type t”

Type Environments

- A *type environment* is a map from variables to types (a kind of symbol table)
  - $\varnothing$ is the empty type environment
    - A closed term e is *well-typed* if $\varnothing \vdash e : t$ for some t
    - We’ll abbreviate this as $\vdash e : t$
  - $A, x{:}t$ is just like A, except x now has type t
    - The type of x in $A, x{:}t$ is t
    - The type of $z{\neq}x$ in $A, x{:}t$ in the type of z in A
- When we see a variable in a program, we look in the type environment to find its type

---

Type Rules

$$\frac{}{A \vdash n : \text{int}} \qquad \frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$$

$$\frac{A, x{:}t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e1 : t \to t' \quad A \vdash e2 : t}{A \vdash e1\ e2 : t'}$$

Example

A = - : int→int

$$\dfrac{\dfrac{-\in \text{dom}(A)}{A \vdash - : \text{int}\to\text{int}} \qquad A \vdash 3 : \text{int}}{A \vdash -\ 3 : \text{int}}$$

Another Example

A = + : int→int→int
B = A, x : int

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{+\in \text{dom}(B)}{B \vdash + :} \qquad \dfrac{x\in \text{dom}(B)}{B \vdash x : \text{i}}}{B \vdash +\ x : \text{int}\to\text{int}} \qquad B \vdash 3 : \text{int}}{B \vdash +\ x\ 3 : \text{int}}}{A \vdash (\lambda x{:}\text{int}.+\ x\ 3) : \text{int}\to\text{int}} \qquad A \vdash 4 : \text{int}}{A \vdash (\lambda x{:}\text{int}.+\ x\ 3)\ 4 : \text{int}}$$

We'd usually use infix x + 3

An Algorithm for Type Checking

- Our type rules are deterministic
  - For each syntactic form, only one possible rule
- They define a natural type checking algorithm
  - TypeCheck : type env × expression → type

```
TypeCheck(A, n) = int
TypeCheck(A, x) = if x in dom(A) then A(x) else fail
TypeCheck(A, λx:t.e) = t → (TypeCheck((A, x:t), e))
TypeCheck(A, e1 e2) =
    let t1 = TypeCheck(A, e1) in
    let t2 = TypeCheck(A, e2) in
        if dom(t1) = t2 then range(t1) else fail
```

Semantics

- Here is a small-step, call-by-value semantics
  - If an expression can't be evaluated any more and is not a value, then it is *stuck*

$$\dfrac{}{(\lambda x{:}t.e1)\ v2 \to e1[v2\backslash x]} \qquad \dfrac{e1 \to e1'}{e1\ e2 \to e1'\ e2}$$

$$\dfrac{e2 \to e2'}{v1\ e2 \to v1\ e2'}$$

$$e ::= v \mid x \mid e\ e$$

$$v ::= n \mid \lambda x{:}t.e$$ *values* – not evaluated

Progress

- Suppose $\vdash e : t$. Then either e is a value, or there exists e’ such that $e \to e'$
- Proof by induction on e
  - Base cases n, $\lambda x.e$ – these are values, so we’re done
  - Base case x – can’t happen (empty type environment)
  - Inductive case $e1\ e2$ – If $e1$ is not a value, then by induction we can evaluate it, so we’re done, and similarly for $e2$. Otherwise both $e1$ and $e2$ are values. Inspection of the type rules shows that $e1$ must have a function type, and therefore must be a lambda since it’s a value. Therefore we can make progress.

Preservation

- If $\vdash e : t$ and $e \to e'$ then $\vdash e' : t$
- Proof by induction on $e \to e'$
  - Induction (easier than the base case!). Expression e must have the form $e1\ e2$.
  - Assume $\vdash e1\ e2 : t$ and $e1\ e2 \to e'$. Then we have $\vdash e1 : t' \to t$ and $\vdash e2 : t'$.
  - Then there are three cases.
    - If $e1 \to e1'$, then by induction $\vdash e1' : t' \to t$, so $e1'\ e2$ has type t
    - If reduction inside $e2$, similar

Preservation, cont’d

- Otherwise $(\lambda x{:}t'.e)\ v \to e[v\backslash x]$. Then we have

$$\frac{x{:}t' \vdash e : t}{\vdash \lambda x{:}t'.e : t' \to t}$$

  - Thus we have
    - $x : t' \vdash e : t$
    - $\vdash v : t'$
  - Then by the substitution lemma (not shown) we have
    - $\vdash e[v\backslash x] : t$
  - And so we have preservation

Substitution Lemma

- If $A \vdash v : t$ and $A, x{:}t \vdash e : t'$, then $A \vdash e[v\backslash x] : t'$
- Proof: Induction on the structure of e
- For lazy semantics, we’d prove
  - If $A \vdash e1 : t$ and $A, x{:}t \vdash e : t'$, then $A \vdash e[e1\backslash x] : t'$

Soundness

- So we have
  - Progress: Suppose $\vdash e : t$. Then either e is a value, or there exists e' such that $e \rightarrow e'$
  - Preservation: If $\vdash e : t$ and $e \rightarrow e'$ then $\vdash e' : t$
- Putting these together, we get soundness
  - If $\vdash e : t$ then either there exists a value v such that $e \rightarrow^* v$, or e diverges (doesn't terminate).
- What does this mean?
  - Evaluation getting stuck is bad, so
  - "Well-typed programs don't go wrong"

Product Types (Tuples)

$e ::= \ldots \mid (e, e) \mid \mathrm{fst}\ e \mid \mathrm{snd}\ e$

$$\frac{A \vdash e1 : t \qquad A \vdash e2 : t'}{A \vdash (e1, e2) : t \times t'}$$

$$\frac{A \vdash e : t \times t'}{A \vdash \mathrm{fst}\ e : t} \qquad \frac{A \vdash e : t \times t'}{A \vdash \mathrm{snd}\ e : t'}$$

- Or, maybe, just add functions
  - $\mathrm{pair} : t \rightarrow t' \rightarrow t \times t'$
  - $\mathrm{fst} : t \times t' \rightarrow t$
  - $\mathrm{snd} : t \times t' \rightarrow t'$

Sum Types (Tagged Unions)

$e ::= \ldots \mid \mathrm{inL}_{t2}\ e \mid \mathrm{inR}_{t1}\ e$

$\quad \mid (\mathrm{case}\ e\ \mathrm{of}\ x1{:}t1 \rightarrow e1 \mid x2{:}t2 \rightarrow e2)$

$$\frac{A \vdash e : t1}{A \vdash \mathrm{inL}_{t2}\ e : t1 + t2} \qquad \frac{A \vdash e : t2}{A \vdash \mathrm{inR}_{t1}\ e : t1 + t2}$$

$$\frac{\begin{array}{c} A \vdash e : t1 + t2 \\ A, x1{:}t1 \vdash e1 : t \quad A, x2{:}t2 \vdash e2 : t \end{array}}{A \vdash (\mathrm{case}\ e\ \mathrm{of}\ x1{:}t1 \rightarrow e1 \mid x2{:}t2 \rightarrow e2) : t}$$

Self Application and Types

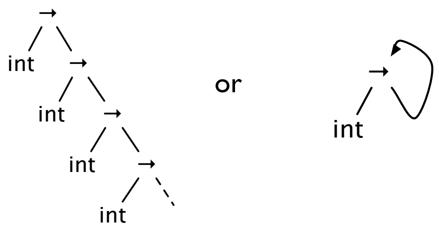
- Self application is not checkable in our system

$$\frac{\dfrac{A, x{:}? \vdash x : t \rightarrow t' \qquad A, x{:}? \vdash x : t}{A, x{:}? \vdash x\ x : \ldots}}{A \vdash \lambda x{:}?.x\ x : \ldots}$$

  - It would require a type t such that $t = t \rightarrow t'$
    - (We'll see this next, but so far...)
- The simply-typed lambda calculus is *strongly normalizing*
  - Every program has a normal form
  - I.e., every program halts!

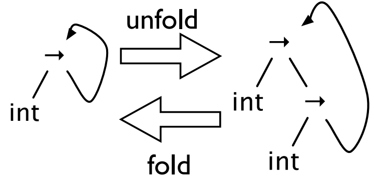
Recursive Types

- We can type self application if we have a type to represent the solution to equations like $t = t \to t'$
  - We define the type $\mu\alpha.t$ to be the solution to the (recursive) equation $\alpha = t$
  - Example: $\mu\alpha.\text{int} \to \alpha$



Folding and Unfolding

- We can check type equivalence with the previous definition (*equi-recursive types*)
  - Standard unification, omit occurs checks
- Alternative solution (*iso-recursive types*):
  - The programmer puts in explicit *fold* and *unfold* operations to expand/contract one "level" of the type trees
    - $\text{unfold } \mu\alpha.t = t[\mu\alpha.t\backslash\alpha]$
    - $\text{fold } t[\mu\alpha.t\backslash\alpha] = \mu\alpha.t$



Iso-recursive Types

- $e ::= \ldots \mid \text{fold } e \mid \text{unfold } e$

$$\frac{A \vdash e : t[\mu\alpha.t\backslash\alpha]}{A \vdash \text{fold } e : \mu\alpha.t} \qquad \frac{A \vdash e : \mu\alpha.t}{A \vdash \text{unfold } e : t[\mu\alpha.t\backslash\alpha]}$$

ML Datatypes

- Combines iso-recursive and sum types
  - Each occurrence of a type constructor when producing a value corresponds to occurrences of inL/inR and, when recursion is involved, fold
  - Each occurrence of a type constructor in a pattern match corresponds to a case and, when recursion is involved, (at least one) unfold

ML Datatypes Example

- type intlist = Int of int | Cons of int * intlist
  - Equivalent to $\mu\alpha.\text{int}+(\text{int} \times \alpha)$
- (Int 3) equivalent to
  - fold ($\text{inL}_{\text{int}\times\mu\beta.\text{int}+(\text{int}\times\beta)}$ 3)
- (Cons (2,(Int 3)) equivalent to
  - fold (inR_{int} (2, fold ($\text{inL}_{\text{int}\times\mu\beta.\text{int}+(\text{int}\times\beta)}$ 3)))
- match e with Int x -> e1 | Cons x -> e2 same as
  - case (unfold e)
    - x:int → e1
    - | x: $\text{int}\times(\mu\beta.\text{int}+(\text{int}\times\beta))$ → e2

Discussion

- In the pure lambda calculus, every term is typable with recursive types
  - (Pure = variables, functions, applications only)
- Most languages have some kind of "recursive" type
  - E.g., for data structures like lists, tree, etc.
- However, usually two recursive types that define the same structure but use a different name are considered different
  - E.g., struct foo { int x; struct foo *next; } is different from struct bar { int x; struct bar *next; }

Recap

- We've discussed simple types so far
  - Integers, functions, pairs, unions
  - Extensions for recursive types and updatable refs
- Type systems have nice properties
  - Type checking is straightforward (needs annotations)
  - Well typed programs don't go "wrong"
    - They don't get stuck in the operational semantics
- But...We can't type check all good programs

Up Next: Improving Types

- How can we build more flexible type systems?
  - More programs type check
  - Type checking is still tractable
- How can reduce the annotation burden?
  - Type inference

Parametric Polymorphism

- Observation: $\lambda x.x$ returns its argument exactly and places no constraints on the type of x
  - The identity function works for any argument type
- We can express this with *universal quantification:*
  - $\lambda x.x : \forall\alpha.\alpha\rightarrow\alpha$
  - For any type α, the identity function has type $\alpha\rightarrow\alpha$
  - This is also known as *parametric polymorphism*

System F: annotated polymorphism

- Let's extend our system as follows:
  - $t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \forall\alpha.t$
  - $e ::= n \mid x \mid \lambda x.e \mid e\ e \mid \Lambda\alpha.e \mid e\ [t]$
- That is, we add polymorphic types, and we add explicit *type abstraction (generalization)* …
  - Annotated code locations at which a value of polymorphic type is created
- … and *type application (instantiation)*
  - Explicitly annotated code locations at which a value of polymorphic type is used
- This system due to Girard, concurrently Reynolds

Defining Polymorphic Functions

- Polymorphic functions map types to terms
  - Normal functions map terms to terms
- Examples
  - $\Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha\rightarrow\alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha\rightarrow\beta\rightarrow\alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.y : \forall\alpha.\forall\beta.\alpha\rightarrow\beta\rightarrow\beta$

Instantiation

- When we use a parametric polymorphic type, we *apply* (or *instantiate*) it with a particular type
  - In System F this is done by hand:
  - $(\Lambda\alpha.\lambda x{:}\alpha.x)[t1] : t1 \rightarrow t1$
  - $(\Lambda\alpha.\lambda x{:}\alpha.x)[t2] : t2 \rightarrow t2$
- This is where the term *parametric* comes from
  - The type $\forall\alpha.\alpha\rightarrow\alpha$ is a "function" in the domain of types, and it is passed a parameter at instantiation time

Type Rules

$$\frac{A,\alpha \vdash e : t}{A \vdash \Lambda\alpha.e : \forall\alpha.t} \qquad \frac{A \vdash e : \forall\alpha.t}{A \vdash e[t'] : t[t'\backslash\alpha]}$$

- Notice that there are no constructs for manipulating values of polymorphic type
  - This justifies instantiation with *any* type—that's what the forall means!
- Note also that we are adding α to A; we could (should?) use this to ensure types are well-formed

---

Small-step Semantics Rules

$$\frac{}{(\Lambda\alpha.e)[t] \to e[t\backslash\alpha]}\ \text{(type-app)} \qquad \frac{e \to e'}{e[t] \to e'[t]}\ \text{(tapp-cong)}$$

- We have to extend substitution to include types; that's up next … !

Free Variables, Again

- We're going to need to perform substitutions on quantified types
  - So just like with lambda calculus, we need to worry about free variables and capture-free substitution
- Define the free variables of a type
  - $FV(\alpha) = \{\alpha\}$
  - $FV(c) = \varnothing$
  - $FV(t \to t') = FV(t) \cup FV(t')$
  - $FV(\forall\alpha.t) = FV(t) - \{\alpha\}$

Substitution, Again

- Define $t[u\backslash\alpha]$ as
  - $\alpha[u\backslash\alpha] = u$
  - $\beta[u\backslash\alpha] = \beta$ where $\beta \neq \alpha$
  - $(t \to t')[u\backslash\alpha] = t[u\backslash\alpha] \to t'[u\backslash\alpha]$
  - $(\forall\beta.t)[u\backslash\alpha] = \forall\beta.(t[u\backslash\alpha])$ where $\beta \neq \alpha$ and $\beta \notin FV(u)$
- Define $e[u\backslash\alpha]$ as
  - $(\lambda x{:}t.e)[u\backslash\alpha] = \lambda x{:}t[u\backslash\alpha].e[u\backslash\alpha]$
  - $(\Lambda\beta.e)[u\backslash\alpha] = \Lambda\beta.e[u\backslash\alpha]$ where $\beta \neq \alpha$ and $\beta \notin FV(u)$
  - $(e1\ e2)[u\backslash\alpha] = e1[u\backslash\alpha]\ e2[u\backslash\alpha]$
  - $x[u\backslash\alpha] = x$ and $n[u\backslash\alpha] = n$

An Imperative Language

e ::= x | λx.e | e e
| ref e allocation
| !e dereference
| e := e assignment
| e; e sequencing

- Notice that this is not C
  - Variables cannot be updated; only references can
  - I.e., there are no l-values or r-values
- This is a language with *updatable references*

Examples

!(ref 0)

let x = ref 0 in
x := !x + 1

let x = ref 0 in
λy. x := !x + 1; !x

Type Checking Rules

- t ::= ... | ref t
  - Note: in ML this type is written t ref

$$\frac{A \vdash e : t}{A \vdash \text{ref } e : \text{ref } t} \qquad \frac{A \vdash e : \text{ref } t}{A \vdash\ !e : t}$$

$$\frac{A \vdash e1 : \text{ref } t \quad A \vdash e2 : t}{A \vdash e1 := e2 : t}$$

Unit and the Unit Type

- Sometimes in imperative programs we write expressions that have some *side effect* but no interesting result
- To represent this directly, use *unit:*
  - e ::= ... | ()
  - t ::= ... | unit

$$\frac{}{A \vdash () : \text{unit}} \qquad \frac{A \vdash e1 : \text{ref } t \quad A \vdash e2 : t}{A \vdash e1 := e2 : \text{unit}}$$

Operational Semantics

- Now we need to keep track of memory
  - State is a map from locations to values
  - Our redexes will be tuples $\langle \text{State, expression} \rangle$
  - As a consequence, order of evaluation matters

- As before, evaluation will yield a fully-evaluated term, also called a *value*
  - $v ::= x \mid \lambda x.e$
  - $e ::= v \mid e\ e \mid \text{ref } e \mid\ !e \mid e := e$

Operational Semantics (cont’d)

$$\frac{}{\langle S, (\lambda x.e1)\rangle \to \langle S, (\lambda x.e1)\rangle}$$

$$\frac{\langle S, e1\rangle \to \langle S', v1\rangle \quad \langle S', e2\rangle \to \langle S'', v2\rangle}{\langle S, e1; e2\rangle \to \langle S'', v2\rangle}$$

$$\frac{\langle S, e\rangle \to \langle S', v\rangle \qquad \text{loc fresh}}{\langle S, \text{ref } e\rangle \to \langle S[v\backslash loc], loc\rangle}$$

Operational Semantics (cont’d)

$$\frac{\langle S, e\rangle \to \langle S', loc\rangle}{\langle S, !e\rangle \to \langle S', S'(loc)\rangle}$$

$$\frac{\langle S, e1\rangle \to \langle S', loc\rangle \quad \langle S', e2\rangle \to \langle S'', v\rangle}{\langle S, e1 := e2\rangle \to \langle S''[v\backslash loc], v\rangle}$$

$$\frac{\langle S, e1\rangle \to \langle S', \lambda x.e\rangle \quad \langle S', e2\rangle \to \langle S'', v\rangle \quad \langle S'', e[v\backslash x]\rangle \to \langle S''', v'\rangle}{\langle S, e1\ e2\rangle \to \langle S''', v'\rangle}$$