Type Systems

CMSC 631 – Program Analysis and Understanding
Spring 2009

Type Systems

The Need for a Type System

- Consider the (untyped) lambda calculus
  - false = \( \lambda x.\lambda y.x \)
  - 0 (Scott) = \( \lambda x.\lambda y.x \)

- 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

- \( t ::= \text{int} \mid t \rightarrow t \)
  - \( t_1 \rightarrow t_2 \) is a the type of a function that, given an argument of type \( t_1 \), returns a result of type \( t_2 \)
    - \( t_1 \) is the domain, and \( t_2 \) 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)
  - \( \emptyset \) is the empty type environment
    - A closed term \( e \) is well-typed if \( \emptyset \vdash e : t \) 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

\[
\begin{align*}
A \vdash n : \text{int} & \quad \text{for } x \in \text{dom}(A) \\
A \vdash x : A(x) & \\
A, x:t \vdash e : t' & \quad A \vdash e_1 : t \rightarrow t' \quad A \vdash e_2 : t \\
A \vdash \lambda x:t.e : t \rightarrow t' & \\
A \vdash e_1\ e_2 : t' & \quad A \vdash e_1\ e_2 : t'
\end{align*}
\]
Example

\[ A = - : \text{int} \rightarrow \text{int} \]

\[- \notin \text{dom}(A) \]

\[ A \vdash - : \text{int} \rightarrow \text{int} \]

\[ A \vdash 3 : \text{int} \]

Another Example

\[ A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ B = A, x : \text{int} \]

\[ + \notin \text{dom}(B) \]

\[ x \notin \text{dom}(B) \]

\[ A \vdash + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ B \vdash + x : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ B \vdash 3 : \text{int} \]

\[ A \vdash 4 : \text{int} \]

\[ A \vdash (\lambda x : \text{int}. + x 3) : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ A \vdash (\lambda x : \text{int}. + x 3) 4 : \text{int} \]

\[ A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ B = A, x : \text{int} \]

\[ B \vdash + x 3 : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

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
  - \( \text{TypeCheck} : \text{type env} \times \text{expression} \rightarrow \text{type} \)

  \[ \text{TypeCheck}(A, n) = \text{int} \]

  \[ \text{TypeCheck}(A, x) = \text{if } x \in \text{dom}(A) \text{ then } A(x) \text{ else fail} \]

  \[ \text{TypeCheck}(A, \lambda x : t. e) = t \rightarrow (\text{TypeCheck}(A, x : t), e) \]

  \[ \text{TypeCheck}(A, e_1 e_2) = \]

  \[ \text{let } t_1 = \text{TypeCheck}(A, e_1) \text{ in} \]

  \[ \text{let } t_2 = \text{TypeCheck}(A, e_2) \text{ in} \]

  \[ \text{if } \text{dom}(t_1) = t_2 \text{ then } \text{range}(t_1) \text{ else fail} \]

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 \textit{stuck}

  \[ e_1 \rightarrow e_1' \]

  \[ (\lambda x : t. e_1) v_2 \rightarrow e_1[v_2/x] \]

  \[ e_1 e_2 \rightarrow e_1' e_2 \]

  \[ e_2 \rightarrow e_2' \]

  \[ v_1 e_2 \rightarrow v_1 e_2' \]

  \[ e ::= v \mid x \mid e e \]

  \[ v ::= n \mid \lambda x : t. e \text{ values – not evaluated} \]
**Progress**

- Suppose \( \vdash e : t \). Then either \( e \) is a value, or there exists \( e' \) such that \( e \rightarrow e' \)
- 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 \( e_1 e_2 \) – If \( e_1 \) is not a value, then by induction we can evaluate it, so we’re done, and similarly for \( e_2 \). Otherwise both \( e_1 \) and \( e_2 \) are values. Inspection of the type rules shows that \( e_1 \) 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 \( e \rightarrow e' \) then \( \vdash e' : t \)
- Proof by induction on \( e \rightarrow e' \)
  - Induction (easier than the base case!). Expression \( e \) must have the form \( \text{el } e_2 \).
  - Assume \( \vdash \text{el } e_2 : t \) and \( \text{el } e_2 \rightarrow e' \). Then we have \( \vdash \text{el } e_2 : t' \rightarrow t \) and \( \vdash e_2 : t' \).
  - Then there are three cases.
    - If \( \text{el } e_2 \rightarrow e'_1 \), then by induction \( \vdash e'_1 : t' \rightarrow t \), so \( e'_1 e_2 \) has type \( t \)
    - If reduction inside \( e_2 \), similar

**Preservation, cont’d**

- Otherwise \( (\lambda x : t'.e) v \rightarrow e[v \backslash x] \). Then we have
  \[
  \frac{x : t' \vdash e : t}{\vdash \lambda x : t'.e : t' \rightarrow t}
  \]
  - 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 \( A \vdash \text{el } e_1 : t \) and \( A, x : t \vdash e : t' \), then \( A \vdash e[e[v \backslash x]] : t' \)
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)

\[
e ::= \ldots \mid (e, e) \mid \text{fst } e \mid \text{snd } e
\]

\[
\frac{A \vdash e_1 : t \quad A \vdash e_2 : t'}{A \vdash (e_1, e_2) : t \times t'}
\]

\[
\frac{A \vdash e : t \times t'}{A \vdash \text{fst } e : t}
\]

\[
\frac{A \vdash e : t \times t'}{A \vdash \text{snd } e : t'}
\]

- Or, maybe, just add functions
  - \( \text{pair} : t \rightarrow t' \rightarrow t \times t' \)
  - \( \text{fst} : t \times t' \rightarrow t \)
  - \( \text{snd} : t \times t' \rightarrow t' \)

Sum Types (Tagged Unions)

\[
e ::= \ldots \mid \text{inL}_{t_2} e \mid \text{inR}_{t_1} e
\]

\[
| (\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 \mid x_2 : t_2 \rightarrow e_2)
\]

\[
\frac{A \vdash e : t_1}{A \vdash \text{inL}_{t_2} e : t_1 + t_2}
\]

\[
\frac{A \vdash e : t_2}{A \vdash \text{inR}_{t_1} e : t_1 + t_2}
\]

\[
\frac{A \vdash e : t_1 + t_2}{A, x_1 : t_1 \vdash e_1 : t \quad A, x_2 : t_2 \vdash e_2 : t}
\]

\[
A \vdash (\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 \mid x_2 : t_2 \rightarrow e_2) : t
\]

Self Application and Types

- Self application is not checkable in our system
  - \( A, x? : \rightarrow x : t' \rightarrow t \rightarrow t' \)
    - \( A \vdash x : t \)
    - \( A \vdash \lambda x :?. x x : \ldots \)
  - 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 $t = t \rightarrow t'$
  - We define the type $\mu \alpha. t$ to be the solution to the (recursive) equation $\alpha = t$
  - Example: $\mu \alpha. \text{int} \rightarrow \alpha$

**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
  - $\text{unfold } \mu \alpha. t = t[\mu \alpha. t \alpha]$  
  - $\text{fold } t[\mu \alpha. t \alpha] = \mu \alpha. t$

**Iso-recursive Types**

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

**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 (\text{inR}_{\text{int}} (2, \text{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 (\text{unfold} \ e)`
    - `x : \text{int} \rightarrow e1`
    - `| x : \text{int} \times (\mu \beta. \text{int}^+(\text{int} \times \beta)) \rightarrow 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 \( \alpha \), 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:
  - \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \forall \alpha.t \)
  - \( e ::= n \mid x \mid \lambda x.e \mid e e \mid \land \alpha.e \mid e [t] \)

- 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

\[
\begin{align*}
A, \alpha & \vdash e : t \\
A & \vdash \lambda \alpha. e : \forall \alpha. t \\
A & \vdash e[t'] : t[t' \setminus \alpha]
\end{align*}
\]

- 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 \( \alpha \) to \( A \); we could (should?) use this to ensure types are well-formed

Small-step Semantics Rules

\[
\begin{align*}
\text{(type-app)} & : e \rightarrow e' \\
\text{(tapp-cong)} & : e[t] \rightarrow e'[t]
\end{align*}
\]

- 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
  - \( \text{FV}(\alpha) = \{\alpha\} \)
  - \( \text{FV}(c) = \emptyset \)
  - \( \text{FV}(t \rightarrow t') = \text{FV}(t) \cup \text{FV}(t') \)
  - \( \text{FV}(\forall \alpha. t) = \text{FV}(t) - \{\alpha\} \)

Substitution, Again

- Define \( t[u\setminus\alpha] \) as
  - \( \alpha[u\setminus\alpha] = u \)
  - \( \beta[u\setminus\alpha] = \beta \quad \text{where} \ \beta \neq \alpha \)
  - \( (t \rightarrow t')[u\setminus\alpha] = t[u\setminus\alpha] \rightarrow t'[u\setminus\alpha] \)
  - \( (\forall \beta. t)[u\setminus\alpha] = \forall \beta. (t[u\setminus\alpha]) \quad \text{where} \ \beta \neq \alpha \ \text{and} \ \beta \notin \text{FV}(u) \)
- Define \( e[u\setminus\alpha] \) as
  - \( (\lambda x : t. e)[u\setminus\alpha] = \lambda x : t[u\setminus\alpha]. e[u\setminus\alpha] \)
  - \( (\lambda \beta. e)[u\setminus\alpha] = \lambda \beta. e[u\setminus\alpha] \quad \text{where} \ \beta \neq \alpha \ \text{and} \ \beta \notin \text{FV}(u) \)
  - \( (e_1 e_2)[u\setminus\alpha] = e_1[u\setminus\alpha] \ e_2[u\setminus\alpha] \)
  - \( x[u\setminus\alpha] = x \quad \text{and} \quad n[u\setminus\alpha] = n \)
An Imperative Language

\[ e ::= x \mid \lambda x.e \mid e \mid \text{ref } e \mid !e \mid e := e \mid e; e \]

- 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

\begin{align*}
!(\text{ref } 0) \\
\text{let } x = \text{ref } 0 \text{ in} \\
& x := !x + 1 \\
\text{let } x = \text{ref } 0 \text{ in} \\
& \lambda y. x := !x + 1; !x
\end{align*}

Type Checking Rules

- \[ t ::= \ldots \mid \text{ref } t \]
  - Note: in ML this type is written \[ t \text{ ref} \]

\[ \begin{array}{c}
A \vdash e : t \\
A \vdash \text{ref } e : \text{ref } t \\
A \vdash !e : t \\
A \vdash e_1 : \text{ref } t \quad A \vdash e_2 : t \\
A \vdash e_1 := e_2 : t
\end{array} \]

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:
  - \[ e ::= \ldots \mid () \]
  - \[ t ::= \ldots \mid \text{unit} \]

\[ \begin{array}{c}
A \vdash () : \text{unit} \\
A \vdash e_1 : \text{ref } t \quad A \vdash e_2 : t \\
A \vdash e_1 := e_2 : \text{unit}
\end{array} \]
Operational Semantics

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

- As before, evaluation will yield a fully-evaluated term, also called a value
  - \( v ::= x | \lambda x.e \)
  - \( e ::= v | e \; e | \text{ref} \; e | !e | e := e \)

Operational Semantics (cont’d)

\[
\begin{align*}
\langle S, (\lambda x.e1) \rangle & \rightarrow \langle S, (\lambda x.e1) \rangle \\
\langle S, e1 \rangle & \rightarrow \langle S', v1 \rangle \quad \langle S', e2 \rangle \rightarrow \langle S'', v2 \rangle \\
\langle S, e1 ; e2 \rangle & \rightarrow \langle S'', v2 \rangle \\
\langle S, e \rangle & \rightarrow \langle S', v \rangle \quad \text{loc fresh} \\
\langle S, \text{ref} \; e \rangle & \rightarrow \langle S[v \text{loc}], \text{loc} \rangle
\end{align*}
\]

Operational Semantics (cont’d)

\[
\begin{align*}
\langle S, e \rangle & \rightarrow \langle S', \text{loc} \rangle \\
\langle S, !e \rangle & \rightarrow \langle S', S'(\text{loc}) \rangle \\
\langle S, e1 \rangle & \rightarrow \langle S', \text{loc} \rangle \quad \langle S', e2 \rangle \rightarrow \langle S'', v \rangle \\
\langle S, e1 := e2 \rangle & \rightarrow \langle S''[v \text{loc}], v \rangle \\
\langle S, e1 \rangle & \rightarrow \langle S', \lambda x.e \rangle \quad \langle S', e2 \rangle \rightarrow \langle S'', v \rangle \\
\langle S'', e[v \lambda x] \rangle & \rightarrow \langle S''', v' \rangle \\
\langle S, e1 \; e2 \rangle & \rightarrow \langle S'''', v' \rangle
\end{align*}
\]