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

Simply-Typed Lambda Calculus

- e ::= n | x | λx:t.e | e e
  - Functions include the type of their argument
  - We don’t really need this, but it will come in handy
- t ::= int | t → t
  - t1 → 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 ⊢ 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 \( \vdash e : t \)
- \( A, x : t \) just like \( A \), except \( x \) now has type \( t \)
  - The type of \( x \) in \( A, x \) is \( t \)
  - The type of \( x \) in \( A, x \) 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

Example

\[
\begin{align*}
A & = \ : \text{int} \to \text{int} \\
A & \vdash - : \text{int} \\
A & \vdash 3 : \text{int} \\
A & \vdash (\lambda x : \text{int}. x + 3) : \text{int} \\
\end{align*}
\]

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 \times expression \rightarrow type
      \[
      \begin{align*}
      \text{TypeCheck}(\emptyset, n) &= \text{int} \\
      \text{TypeCheck}(\emptyset, x) &= \text{int} \times \text{dom}(A) \text{ then } A(x) \text{ else fail} \\
      \text{TypeCheck}(A, \lambda x : e) &= \text{TypeCheck}(A, \lambda x.e) \\
      \text{TypeCheck}(A, e1 e2) &= \\
      &\text{let } t1 = \text{TypeCheck}(A, e1) \text{ in} \\
      &\text{let } t2 = \text{TypeCheck}(A, e2) \text{ in} \\
      &\text{if } \text{dom}(t1) = t2 \text{ then range}(t1) \text{ else fail}
      \end{align*}
      \]

Semantics

- Here is our semantics, with integers
  - Notice that the last rule requires that \( e1 \) is a function
    - The other cases are undefined (i.e., an error)
  - We'd usually use infix \( x + 3 \)
Semantics with Error

- If we want to make this precise, can add a new term error
  - Invalid programs reduce to error

\[
e_1 \to \text{error} \\
\text{e}_1 \text{e}_2 \to \text{error}
\]

Soundness (cont’d)

- Induction: Suppose \( e = e_1 \ e_2 \)
  - Then by assumption this type checks by the rule
    - Since \( \text{e}_1 : u \to u' \) the reduction rule
  - \( \text{xu} i \to e : u' \)

Sum Types (Tagged Unions)

\[
e ::= ... | \text{in}_{\text{L}1} \ e | \text{in}_{\text{R}1} \ e \\
| (\text{case e of x1:t1} \to e1| x2:t2 \to e2)
\]

- Also, since \( \text{xu} i \to e : u' \), we must have applied the type rule
  - \( \text{e}_1 \to \lambda x.e \text{e}[\text{e}_2 x] \to l e' \)

Product Types (Tuples)

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

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

Soundness (cont’d)

- Theorem: If \( e \) is a closed term and \( \vdash e : t \) for some \( t \), then \( e \to l e' \) where \( e' \) is not error
  - Note: I will omit the argument types in the proof
- Lemma: If \( \vdash e : t \) and \( e \to l e' \), then \( \vdash e' : t \)
  - Notice \( e' \) cannot be \text{error}, because no type rules for it
  - Proof: By induction on the reduction \( e \to l e' \)
  - Base case \( e = n \) trivial
  - Base case \( e = \lambda x.e \) trivial

Sound (cont’d)

- Induction: Suppose \( e = e_1 \ e_2 \)
  - Then by assumption this type checks by the rule
    - Since \( \vdash e_1 : u \to u' \) the reduction rule
  - \( \text{e}_1 \to l \lambda x.e \text{e}[\text{e}_2 x] \to l e' \)
  - \( \text{e}_1 \text{e}_2 \to l e' \)

- Also, since \( \vdash \lambda x.e : u \to u' \), we must have applied the type rule
  - \( \text{e}_1 \to l \lambda x.e \text{e}[\text{e}_2 x] \to l e' \)

- Then since \( \text{e}_1 \to l \lambda x.e \text{e}[\text{e}_2 x] \to l e' \) and \( \text{e}_1 \to l \lambda x.e \text{e}[\text{e}_2 x] \to l e' \), by the substitution lemma (omitted), we know \( \vdash \text{e}_2 : u \) by the reduction \( e \to l e' \)
  - Then by induction again, we know that \( \vdash e' : u' \), and so we’re done.
Self Application and Types

- Self application is not checkable in our system
  \[ A, x :? \vdash x : t \quad \vdash A, x :? \vdash x : t \quad \vdash A, x :? \vdash x : \_ \]

- 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 \( \alpha \cdot t \) to be the solution to the (recursive) equation \( \alpha \equiv t \)
  - Example: \( \alpha \cdot \text{int} \rightarrow \alpha \)

Folding and Unfolding

- We can check type equivalence with the previous definition
  - Standard unification, omit occurs checks

- Alternative solution:
  - The programmer puts in explicit fold and unfold operations to expand/contract one “level” of the type trees
    - \( \text{fold } \alpha \cdot t = \tau \equiv \alpha \cdot (\text{int } \text{t}) \)
    - \( \text{unfold } (\alpha \cdot t) = \alpha \cdot \text{int } \text{t} \)

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; }

Examples

\[ !(\text{ref } 0) \]

\[ \text{let } x = \text{ref } 0 \text{ in } \]
\[ x := !(x + 1) \]

\[ \text{let } x = \text{ref } 0 \text{ in } \]
\[ \lambda y. x := !(x + 1); !x \]

An Imperative Language

\[ e ::= x | \lambda x. e | e e \]
\[ | \; \text{ref } e \quad \text{allocation} \]
\[ | !e \quad \text{dereference} \]
\[ | e := e \quad \text{assignment} \]
\[ | e; e \quad \text{sequencing} \]

- 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
### Type Checking Rules

- \( t ::= \ldots | \text{ref } t \)
  - Note: in ML this type is written \( t \text{ ref} \)

\[
\begin{align*}
A \rightarrow e : t & \quad \quad A \rightarrow e : \text{ref } t \\
A \rightarrow \text{ref } e : \text{ref } t & \quad \quad A \rightarrow ! e : t \\
A \rightarrow e_1 : \text{ref } t & \quad \quad A \rightarrow e_2 : t \\
A \rightarrow e_1 := e_2 : \text{ref } t & \quad \quad A \rightarrow e_1 := e_2 : t
\end{align*}
\]

### 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 \textit{unit}:
  - \( e ::= \ldots | () \)
  - \( t ::= \ldots | \text{unit} \)

\[
\begin{align*}
A \rightarrow e_1 : \text{ref } t & \quad \quad A \rightarrow e_2 : \text{unit} \\
A \rightarrow e_1 := e_2 : \text{unit} & \quad \quad A \rightarrow () : \text{unit}
\end{align*}
\]

### 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
  - \( v ::= x | \lambda x. e \)
  - \( e ::= v | e_1 e_2 | \text{ref } e | ! e | e_1 := e_2 \)

\[
\begin{align*}
\langle S, (\lambda x. e) \rangle & \rightarrow \langle S, (\lambda x. e) \rangle \\
\langle S, e \rangle & \rightarrow \langle S', \text{loc} \rangle \\
\langle S, e \rangle & \rightarrow \langle S', (\text{loc}) \rangle \\
\langle S, e \rangle & \rightarrow \langle S', \text{loc} \rangle \\
\langle S, e_1 e_2 \rangle & \rightarrow \langle S''', v \rangle \\
\langle S, e_1 := e_2 \rangle & \rightarrow \langle S''', [\text{loc} \mapsto v] \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S', \lambda x. e \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S', \text{ref } v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle
\end{align*}
\]

### Operational Semantics (cont’d)

\[
\begin{align*}
\langle S, e \rangle & \rightarrow \langle S', \text{loc} \rangle \\
\langle S, e \rangle & \rightarrow \langle S', (\text{loc}) \rangle \\
\langle S, e \rangle & \rightarrow \langle S', \text{loc} \rangle \\
\langle S, e \rangle & \rightarrow \langle S', \text{loc} \rangle \\
\langle S, e_1 e_2 \rangle & \rightarrow \langle S''', v \rangle \\
\langle S, e_1 := e_2 \rangle & \rightarrow \langle S''', [\text{loc} \mapsto v] \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S', \lambda x. e \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S', \text{ref } v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S'', v \rangle
\end{align*}
\]

### 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

Type Language

• Problem: Consider the rule for functions

\[ A, x : t \vdash e : t' \]

\[ A \vdash \lambda x : t. e : t \rightarrow t' \]

• Without type annotations, where do we get \( t \)?
  ■ We’ll use type variables to stand for as-yet-unknown types
    - \( t ::= \alpha | \text{int} | t \rightarrow t \)
  ■ We’ll generate equality constraints \( t = t \) among the types and type variables
    - And then we’ll solve the constraints to compute a typing

Type Inference

• Let’s consider the simply typed lambda calculus with integers
  ■ \( e ::= n | x | \lambda x : t. e | e \ e \)
  ■ (No parametric polymorphism)

• Type inference: Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?

Type Inference Rules

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

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

\[ A \vdash x : A(x) \]

\[ A, x : \alpha \vdash e : t' \quad \alpha \text{ fresh} \]

\[ \alpha \rightarrow \alpha \]

\[ A \vdash \lambda x : \alpha. e : \alpha \rightarrow t' \]

\[ A \vdash e_1 : t_1 \]

\[ A \vdash e_2 : t_2 \]

\[ t_1 = t_2 \rightarrow \beta \quad \beta \text{ fresh} \]

\[ A \vdash e_1 e_2 : \beta \]

Example

\[ A, x : \alpha \vdash x : \alpha \]

\[ A \vdash (\lambda x : \alpha). \alpha : \alpha \rightarrow \alpha \]

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

\[ \alpha \rightarrow \alpha = \text{int} \rightarrow \beta \]

\[ A \vdash (\lambda x : \alpha). 3 : \beta \]

• We can solve the constraint \( \alpha \rightarrow \alpha = \text{int} \rightarrow \beta \)
  ■ \( \alpha = \text{int} = \beta \)

• Thus this program is typable, and we can derive a typing by replacing \( \alpha \) and \( \beta \) by \( \text{int} \) in the proof tree

Solving Equality Constraints

• We can solve the equality constraints using the following rewrite rules, which reduce a larger set of constraints to a smaller set

  \[ C \cup \{ \text{int=\text{int}} \} \Rightarrow C \]

  \[ C \cup \{ \alpha = e \} \Rightarrow C[t] \]

  \[ C \cup \{ e = \alpha \} \Rightarrow C[t] \]

  \[ C \cup \{ t_1 \rightarrow t_2 = t_1' \rightarrow t_2' \} \Rightarrow C \cup \{ t_1 = t_1' \} \cup \{ t_2 = t_2' \} \]

  \[ C \cup \{ \text{int=t} \rightarrow \text{t} \} \Rightarrow \text{unsatisfiable} \]

  \[ C \cup \{ t_1 \rightarrow \text{t} = \text{int} \} \Rightarrow \text{unsatisfiable} \]
**Occurs Check**

- We don’t have recursive types, so we shouldn’t infer them
- So in the operation $C[t\alpha]$, require that $\alpha \not\in \text{FV}(t)$
- In practice, it may better to allow $\alpha \in \text{FV}(t)$ and do the occurs check at the end
  - But that can be awkward to implement

**Unifying a Variable and a Type**

- Computing $C[t\alpha]$ by substitution is inefficient
- Instead, use a union-find data structure to represent equal types
  - The terms are in a union-find forest
  - When a variable and a term are equated, we union them so they have the same ECR
  - Note: Only need to maintain ECR of variables, not of all terms, though doing terms as well has some potential advantages

**Example**

\[
\begin{align*}
\alpha &= \text{int} \rightarrow \beta \\
\gamma &= \text{int} \rightarrow \text{int} \\
\alpha &= \gamma
\end{align*}
\]

**Discussion**

- The algorithm we’ve given finds the most general type of a term
  - Any other valid type is “more specific,” e.g., $\lambda x : \text{int} \rightarrow \text{int}$
  - Formally, any other valid type can be gotten from the most general type by applying a substitution to the type variables
- This is still a monomorphic type system
  - $\alpha$ stands for “some particular type, but it doesn’t matter exactly which type it is”

**Unification**

- The process of finding a solution to a set of equality constraints is called unification
  - Original algorithm due to Robinson
    - But his algorithm was inefficient
  - Often written out in different form
    - See Algorithm W
  - Constraints usually solved on-line
    - As type inference rules applied

**Parametric Polymorphism**

- Observation: $\lambda x.x$ returns its argument exactly and does not place any 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
When we use a parametric polymorphic type, we instantiate it with a particular type.

- For now, the programmer specifies this by hand.
- \((\lambda x.x)[S] : S \rightarrow S\)
- \((\lambda x.x)[T] : T \rightarrow T\)

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.
- Sometimes this type is written \(\alpha \rightarrow \alpha\)

When is it safe to generalize (quantify) a type variable \(\alpha\) in the type of expression \(e\)?

- Answer: Whenever we can redo the typing proof for \(e\), choosing \(\alpha\) to be anything we want, and still have a valid typing proof.

Define \(t[u[\alpha]]\) as

- \(\alpha[u[\alpha]] = u\)
- \(\beta[u[\alpha]] = \beta\) where \(\beta \neq \alpha\)
- \((t \rightarrow t')[u[\alpha]] = t[u[\alpha]] \rightarrow t'[u[\alpha]]\)
- \((\forall \beta. t)[u[\alpha]] = \forall \beta.(t[u[\alpha]])\) where \(\beta \neq \alpha\) and \(\beta \in \text{FV}(u)\)

Look familiar?

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

Notice we may need to use alpha conversion on the quantified type to avoid capture.

Notice we can substitute in any type.

That’s what the for all means!

The choice of the type of \(x\) is purely local to type checking \(\lambda x.x\).

- There is no interaction with the outside environment.
- Thus we can generalize the type of \(x\).
Examples (cont’d)

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

- The function restricts the type of \( x \), so we cannot introduce a type variable
  - Thus we cannot generalize the type of \( x \)
  - We can only generalize when the function doesn’t “look at” its parameter

Discussion

- We’ve seen two forms of polymorphism
  - Subtype polymorphism (see OOP)
  - Parametric polymorphism
    - A more restrictive variant is also called Hindley-Milner style polymorphism

- Some languages also have ad-hoc polymorphism
  - E.g., + operator that works on ints and floats
  - E.g., overloading in Java

Another Justification

- Suppose we have
  - \( A \vdash e : t \) and \( \alpha \in \text{FV}(A) \)
  - \( A \vdash e : \forall \alpha . t \)
  - Then let \( u \) be any type. By induction, can show
    - \( A[u/\alpha] \vdash e : t[u/\alpha] \)
    - But then since \( \alpha \in \text{FV}(A) \), that’s equivalent to
      - \( A \vdash e : t[u/\alpha] \)

Polymorphic Type Inference

- We’d like to extend our algorithm to polymorphic type inference

  - Major problem: Our system for polymorphism is too expressive
    - In fact, type inference is undecidable
Hindley-Milner Polymorphism

- Restrict polymorphism to only the “top level”
- Only introduce polymorphism at `let`
- Always fully instantiate when we use a variable with a polymorphic type
  - `e ::= n | x | λx.e | e e | let x = e in e`
  - These are type schemes
  - `t ::= α | int | t → t`

Old Type Inference Rules

- A type inference algorithm that explicitly solves the equality constraints on-line
- Instead of implicit global substitution (like we used before), threads the substitution through the inference
- In practice, use previously algorithm, plus generalize at `let` and instantiate at variable uses

New Type Inference Rules

- At `let`, generalize over all possible variables
  \[
  \frac{A ⊢ e : t \quad A, x : \forall α. t ⊢ e_2 : t_2 \quad \alpha = FV(t)(FV(A))}{A ⊢ \text{let } x = e_1 \text{ in } e_2 : t_2}
  \]

- At variable uses, instantiate to all fresh types
  \[
  \frac{A(x) = \forall \alpha. t \quad \beta \text{ fresh}}{A ⊢ x : t[\beta[\alpha]]}
  \]

Example

- Parametric polymorphic type inference
  \[
  \begin{align*}
  \text{let } x = \lambda x. x \text{ in} & \quad \text{// } x : \forall \alpha. \alpha \to \alpha \\
  x 3; & \quad \text{// } x : \beta \to \beta, \beta = \text{int} \\
  x (\lambda y. y) & \quad \text{// } x : \gamma \to \gamma, \gamma = \delta \to \delta
  \end{align*}
  \]
  - This would be untypable in a monomorphic type system

Algorithm W

- A type inference algorithm that explicitly solves the equality constraints on-line

Polymorphism and References

- Suppose we want polymorphism in our imperative language
  - `e ::= x | n | λx.e | e e | !e | e := e`
  - `s ::= t | \forall α. s`
  - `t ::= α | int | t → t`

- What if we try our standard rule?
  \[
  \frac{A ⊢ e : t \quad A, x : \forall α. t ⊢ e_2 : t_2 \quad \alpha = FV(t)(FV(A))}{A ⊢ \text{let } x = e_1 \text{ in } e_2 : t_2}
  \]
Naive Generalization is Unsound

- Example (due to Tofte)
  ```
  let r = ref (\lambda x.x) in    // r : \forall \alpha. ref (\alpha \rightarrow \alpha)
  r := \lambda x.x+1;           // checks; use r at ref (int \rightarrow int)
  (r) true                     // oops! checks; use r at ref(bool \rightarrow bool)
  ```

- \(\alpha\) should not be generalized, because later uses of \(r\) may place constraints on it

- Nobody realized there was a problem for a long time

Solution: The Value Restriction

- Only allow values to be generalized
  ```
  v ::= x | n | \lambda x.e
  e ::= v | e e | ref e | !e | e := e
  ```

- Intuition: Values cannot later be updated
  - This solution due to Wright and Felleisen
  - Tofte found a much more complicated solution

Benefits of Type Inference

- Handles higher-order functions
- Handles data structures smoothly
- Works in infinite domains
  - Set of types is unlimited
- No forward/backward distinction
- Polymorphism provides context-sensitivity

Drawbacks to Type Inference

- Flow-insensitive
  - Types are the same at all program points
  - May produce coarse results
  - Type inference failure can be hard to understand

- Polymorphism may not scale
  - Exponential in worst case
  - Seems fine in practice (witness ML)