Type Systems (continued)
Recap: Type Systems

• 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
    - Notice that the type system may be conservative — it may report programs as erroneous if they could run without type errors
Recap: Type System Benefits

• Help catch bugs early
  ▪ Before ever running code (it’s a kind of *static analysis*)

• Provides useful documentation
  ▪ Automatically checked
  ▪ Especially useful as a program grows, evolves
Recap: Simply-Typed Lambda Calculus

• $e ::= n \mid x \mid \lambda x: t. e \mid e \ e$
  - Functions include the type of their argument
  - We’ve added integers, so we can have (obvious) type errs

• $t ::= \text{int} \mid t \rightarrow t$
  - $t_1 \rightarrow t_2$ 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
Recap: Type Judgments

- Type system proves judgments of the form
  - $A \vdash e : t$
  - “In type environment $A$, expression $e$ has type $t$”

- A type environment is a map from variables to types
Recap: Type Rules

- Need one rule for every kind of expression
A = - : int \rightarrow int

\[
\frac{- \in \text{dom}(A)}{A \vdash - : \text{int} \rightarrow \text{int}}
\]

\[
\frac{A \vdash - : \text{int} \rightarrow \text{int}}{A \vdash 3 : \text{int}}
\]

A \vdash -3 : \text{int}
Recap: Type Soundness

- We prove soundness of a type system by proving two subtheorems: *progress* and *preservation*

- **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 \( \vdash e' : t \)

- **Soundness**: If \( \vdash e : t \) then either there exists a value \( v \) such that \( e \rightarrow^* v \), or \( e \) diverges (doesn’t terminate).
  - This relates the type system to runtime guarantees!
Today

• Types for more advanced language features
  ▪ Conditionals
  ▪ Pairs/tuples
  ▪ Sum types
  ▪ refs

• Polymorphism
  ▪ Subtyping
Conditionals

• Add booleans, conditionals to our expressions:

\[ e ::= \ldots | \text{true} | \text{false} | \text{if } e \text{ then } e \text{ else } e \]

• Add boolean type to our types:

\[ t ::= \ldots | \text{bool} \]
Conditionals: Op. Semantics

if true then e2 else e3 → e2

if false then e2 else e3 → e3

e1 → e1’

if e1 then e2 else e3
if e1’ then e2 else e3

• What are the type rules?
**Conditionals: Type Rules**

- \( A \vdash \text{true} : \text{bool} \)
- \( A \vdash \text{false} : \text{bool} \)
- \( A \vdash e_1 : \text{bool} \)
- \( A \vdash e_2 : t \)
- \( A \vdash e_3 : t \)
- \( A \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \)
Product Types

• Let’s add *pairs* to our language
  ▪ Like a restricted version of SML’s tuples
• New expressions:
  
  \[
  e ::= \ldots \mid (e, e) \mid \text{fst } e \mid \text{snd } e
  \]
• A new *product* type:
  
  \[
  t ::= \ldots \mid t \times t
  \]
• What are the type rules?
Product Types: Type Rules

\[
\begin{align*}
A \vdash e_1 : t & \quad A \vdash e_2 : t' \\
\hline
A \vdash (e_1, e_2) : t \times t' \\
\end{align*}
\]

\[
\begin{align*}
A \vdash e : t \times t' & \\
\hline
A \vdash \text{fst } e : t \\
\end{align*}
\]

\[
\begin{align*}
A \vdash e : t \times t' & \\
\hline
A \vdash \text{snd } e : t' \\
\end{align*}
\]
Sum Types (Tagged Unions)

• Programs often deal with heterogeneous collections of values

• Think: many datatypes in SML, e.g.,
  datatype intlist = inil | cons of int * intlist;

• We can perform a case analysis on an intlist
  ▪ When \textit{inil}, list is empty, can’t get any values from it
  ▪ When \textit{cons}, we can split it into an int and the rest of the list

• How can type system represent this?
Sum Types (Tagged Unions)

- We introduce *sum types*, a set of values drawn from exactly two given types

\[ t ::= \ldots \mid t + t \]

- e.g., if `Cat` and `Dog` are types, we can create a new type `Pet = Cat + Dog`
Sum Types: Injecting

- \( \text{Pet} = \text{Cat} + \text{Dog} \)
- We create elements of type \( \text{Pet} \) by tagging elements of type \( \text{Cat} \) or type \( \text{Dog} \)
  - If \( c \) is a \( \text{Cat} \), then \( \text{inl} \ c \) is a \( \text{Pet} \)
  - If \( d \) is a \( \text{Dog} \), then \( \text{inr} \ d \) is a \( \text{Pet} \)
- e.g., for a function \( f: (\text{Pet}) \rightarrow \text{int} \)
  - \( f(c) \) is ill-typed
  - \( f(\text{inl} \ c) \) is okay!
- We are \textit{injecting} elements of type \( \text{Cat} \) and \( \text{Dog} \) into the left/right components of sum type \( \text{Pet} \)
Sum Types: New Expressions

- Add three new expressions:
  
  \[
  e ::= \ldots | \text{inL}_{t_2} e | \text{inR}_{t_1} e \\
  | (\text{case } e \text{ of } x_1:t_1 \rightarrow e_1 | x_2:t_2 \rightarrow e_2)
  \]

- **case** statement:
  - if \( e \) has type \( t_1 \), bind \( x_1 \) to \( e \) and evaluate \( e_1 \)
  - Similar for other case

- Why is \( t_2 \) necessary in \( \text{inL}_{t_2} \)?
  - Tells us the “other” type in the sum
  - See typing rules…
Sum Types: Type Rules

\[
\begin{align*}
A \vdash e : t_1 & \quad A \vdash e : t_2 \\
\frac{}{A \vdash \text{inL}_{t_2} \ e : t_1 + t_2} & \quad \frac{}{A \vdash \text{inR}_{t_1} \ e : t_1 + t_2} \\
\frac{}{A \vdash (\text{case} \ e \ \text{of} \ x_1 : t_1 \ \to \ e_1 \mid x_2 : t_2 \ \to \ e_2) : t}
\end{align*}
\]
Lambda Calc with Updatable Refs

- $e ::= \ldots \mid \text{ref } e \mid !e \mid e := e$
  - ML-style updatable references
    - $\text{ref } e$ — allocate memory and set its contents to $e$; return pointer
    - $!e$ — dereference pointer and return contents
    - $e_1 := e_2$ — update contents pointed to by $e_1$ with $e_2$

- $t ::= \ldots \mid t \text{ ref}$
  - A $t \text{ ref}$ is a pointer to contents of type $t$
Type Rules for Refs

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

• “Polymorphism refers to a range of language mechanisms that allow a single part of a program to be used with different types in different contexts.” (Pierce, Types and Programming Languages)

• Three most common kinds of polymorphism:
  - Subtyping
    - Typically, subclasses in OOP
  - Parametric polymorphism
    - Generics in Java, type variables ‘a in SML
  - Ad-hoc polymorphism
    - e.g., overloaded function + that works for ints and floats
Subtyping

• The Liskov Substitution Principle (paraphrased):

Let \( q(x) \) be a property provable about objects \( x \) of type \( T \). If \( S \) is a subtype of \( T \), then \( q(y) \) should be provable for objects \( y \) of type \( S \).

• In other words

If \( S \) is a subtype of \( T \), then an \( S \) can be used anywhere a \( T \) is expected.

• Commonly found in object-oriented programming
  ▪ Subclasses can be used where superclasses expected
Lambda Calc with Subtyping

• To demonstrate subtyping, let’s add floats f

• \( e ::= n \mid f \mid x \mid \lambda x:t.e \mid e \ e \)
  ▪ We now have both floating point numbers and integers
  ▪ We want to be able to implicitly use an integer wherever a floating point number is expected
  ▪ Warning: This is a bad design! Don’t do this in real life

• \( t ::= \text{int} \mid \text{float} \mid t \rightarrow t \)
  ▪ We want \( \text{int} \) to be a subtype of \( \text{float} \)
Subtyping

• We’ll write \( t_1 \leq t_2 \) if \( t_1 \) is a subtype of \( t_2 \)
• Define subtyping by more inference rules
• Base case

\[
\text{int} \leq \text{float}
\]

- (notice reverse is not allowed)

• What about function types?

\[
???
\]

\[
t_1 \rightarrow t_1' \leq t_2 \rightarrow t_2'
\]
Replacing “f x” by “g x”

- Suppose \( g : t_1 \rightarrow t_1' \) and \( f : t_2 \rightarrow t_2' \)
- When is \( t_1 \rightarrow t_1' \leq t_2 \rightarrow t_2' \)?

- Return type:
  - We are expecting \( t_2' \) (f’s return type)
  - So we can return \textit{at most} \( t_2' \)
  - So need \( t_1' \leq t_2' \)

- Examples
  - If we’re expecting \texttt{float}, can return \texttt{int} or \texttt{float}
  - If we’re expecting \texttt{int}, can only return \texttt{int}
Replacing “f x” by “g x”

• Suppose $g : t_1 \rightarrow t_1'$ and $f : t_2 \rightarrow t_2'$

• When is $t_1 \rightarrow t_1' \leq t_2 \rightarrow t_2'$?

• Argument type:
  - We are supposed to accept expecting $t_2$ (f’s arg type)
  - So $g$ must accept at least $t_2$
  - So need $t_2 \leq t_1$

• Examples
  - A function that accepts an int can be replaced by one that accepts int, or one that accepts float
  - A function that accepts a float can only be replaced by one that accepts float
Subtyping on Function Types

- We say that subtyping is
  - **Covariant** in the range (subtyping dir the same)
  - **Contravariant** in the domain (subtyping dir flips)

- Some languages have gotten this wrong
  - Eiffel allows covariant parameter types
Contravariance: Second Example

• Let Pug, Dog, Animal be types such that Pug ≤ Dog ≤ Animal
• Let \( f \) be a function of type \((\text{Dog} \to \text{Dog}) \to \text{Int}\)
• Let \( g \) be a function of type \( \text{Animal} \to \text{Dog} \)
• Let \( h \) be a function of type \( \text{Pug} \to \text{Dog} \)

• Is \( f(g) \) type safe?
  • Yes! \( f \) expects to apply \( g \) to a Dog, and since \( g \) accepts Animals, it accepts Dogs

• Is \( f(h) \) type safe?
  • No! \( f \) expects to apply \( h \) to any Dog, but \( h \) only accepts Pugs, not other kinds of Dogs
Type Rules, with Subtyping

\[
\begin{align*}
A & \vdash n : \text{int} \\
A & \vdash x : A(x) \\
A & \vdash x : \text{dom}(A) \\
A & \vdash \lambda x : t. e : t \to t' \\
A & \vdash f : \text{float} \\
A & \vdash \lambda x : t. e : t \to t' \\
A & \vdash e_1 : t_1 \to t_1' \\
A & \vdash e_2 : t_2 \\
t_2 & \leq t_1 \\
A & \vdash e_1 \ e_2 : t_1'
\end{align*}
\]
Soundness

• Progress and preservation still hold
  - Slight tweak: as evaluation proceeds, expression’s type may “decrease” in the subtyping sense
  - Example:
    - (if true then n else f) : float
    - But after taking one step, will have type int ≤ float
Subtyping, again

• Instead of having $t_2 \leq t_1$ in premise of application rule, we could have:

$$
\begin{align*}
A \vdash e_1 : t_1 \rightarrow t_1' & \quad A \vdash e_2 : t_1 \\
\hline
A \vdash e_1 \; e_2 : t_1' \\
\end{align*}
$$

$$
\begin{align*}
A \vdash e : t & \quad t \leq t' \\
\hline
A \vdash e : t' \\
\end{align*}
$$
Subtyping, again (cont’d)

• Rule with subtyping is called *subsumption*
  ▪ Very clearly captures subtyping property
• But system is no longer *syntax driven*
  ▪ Given an expression e, there are two rules that apply to e
    (“regular” type rule, and subsumption rule)
• Can prove that the two systems are equivalent
  ▪ Exercise left to the reader
Subtyping Refs

• The wrong rule for subtyping refs is

\[
\begin{align*}
\text{t1} \leq \text{t2} \\
\hline
\text{t1} \text{ ref} \leq \text{t2} \text{ ref}
\end{align*}
\]

• Counterexample

\[
\begin{align*}
\text{let } x = \text{ref 3 in} & \quad (* x : \text{int ref} *) \\
\text{let } y = x \text{ in} & \quad (* y : \text{float ref} *) \\
y := 3.14 & \quad (* \text{oops! !x is now a float} *)
\end{align*}
\]
Aliasing

• We have multiple names for the same memory location
  ▪ But they have different types
  ▪ This we can **write** different types into the same memory
Solution #1: Java’s Approach

• Java uses this subtyping rule
  - If $S$ is a subclass of $T$, then $S[]$ is a subclass of $T[]$

• Counterexample:
  - Foo[] a = new Foo[5];
  - Object[] b = a;
  - b[0] = new Object(); // forbidden at runtime
  - a[0].foo(); // …so this can’t happen
Solution #2: Purely Static

- Reason from rules for functions
  - A reference is like an object with two methods:
    - \( \text{get} : \text{unit} \rightarrow t \)
    - \( \text{set} : t \rightarrow \text{unit} \)
  - Notice that \( t \) occurs both co- and contravariantly
  - Thus it is \textit{non-variant}

- The right rule:

\[
\begin{align*}
\text{if } \quad & t_1 \leq t_2 \quad \text{and} \quad t_2 \leq t_1 \\
\text{then } \quad & t_1 \text{ ref} \leq t_2 \text{ ref} \quad \text{or} \quad \text{if } \quad & t_1 = t_2 \\
\text{then } \quad & t_1 \text{ ref} \leq t_2 \text{ ref}
\end{align*}
\]