Subtyping

• Called subclassing in object-oriented programming

class A { ... }
class B extends A { ... }
A x = new A();  // valid
A x = new B();  // valid, since B is a subclass of A
B x = new A();  // invalid

Definition of Subtyping

• Liskov:
  - If for each object o₁ of type S there is an object o₂ of type T such that for all programs P defined in terms of o₁, the behavior of P is unchanged when o₂ is substituted for o₁, then S is a subtype of T.

• Informal statement
  - If anyone expecting a T can be given an S instead, then S is a subtype of T.

The Subtyping Relation

• Let’s assume that we have
  - A set of primitive objects c (e.g., 0, 1, 0.1, 3.14, ...)
    - e ::= c | x | λx.e | e e
  - A set of primitive types C (e.g., int, float)
    - t ::= C | t → t

• We are also given a partial order ≤ on C
  - This is the subtyping relation
  - E.g., int ≤ float
  - Warning: this is a terrible example; implicitly allowing integer to floating point conversions leads to confusion

Type Checking Rules

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

Example

\[
\begin{align*}
& \vdash + : f \rightarrow f \rightarrow f \\
& \vdash 3 : i \quad \vdash i \mapsto f \\
& \vdash 3 : f \\
& \vdash + : f \rightarrow f \rightarrow f \\
& \vdash 3 : f \\
& \vdash + : f \rightarrow f \\
& \vdash 3 : f \\
& \vdash + : f \\
& \vdash 4.0 : f \\
& \vdash + : f \rightarrow f \\
& \vdash 3.4 : f
\end{align*}
\]
Subtyping and Type Constructors

- We’re given ≤ on primitive types
  - How do we extend to type constructors?
    - A type constructor builds new types from existing types
    - E.g., →, ×, ref
  - Product types are straightforward
    \[ t_1 \times t_2 \leq t_1' \times t_2' \]
  - Example: \( \text{int} \times \text{float} \leq \text{float} \times \text{float} \)

Subtyping and Function Types

- What about function types?
  \[ t_1 \rightarrow t_2 \leq t_1' \rightarrow t_2' \]
- Recall: \( S \) is a subtype of \( T \) if an \( S \) can be used anywhere a \( T \) is expected
  - When can we replace \( R[f x] \) with \( R[g x] \)?

Replacing \( R[f x] \) by \( R[g x] \)

- Assume \( f : t_1 \rightarrow t_1' \) and \( g : t_2 \rightarrow t_2' \)
- Also assume \( R[f x] \) type checks
- ‘When is \( t_2 \rightarrow t_2' \leq t_1 \rightarrow t_1' \)?
- Return type:
  - Every possible result of \( g x \) must be accepted
  - So \( t_2' \leq t_1' \)
- Argument type:
  - Every possible argument to \( f \) must also be accepted by \( g \)
  - So \( t_1 \leq t_2 \)

The Subtype Rule for Functions

- We say that \( \rightarrow \) is
  - \textit{Covariant} in its range (subtyping dir stays the same)
  - \textit{Contravariant} in its domain (subtyping dir flips)

Subtyping and References

- The \textbf{wrong} rule for references
  \[ t \leq t' \]
  \[ \text{ref } t' \leq \text{ref } t' \]

Counterexample:
\[
\begin{align*}
\text{let } x &= \text{ref } 0 \\
\text{let } y &= x \\
y &= 3.14; & \quad \text{// typechecks, since int \leq \text{float}} \\
\text{printInt } (x) & \quad \text{// oops! typechecks, since } y : \text{ref int}
\end{align*}
\]

The Right Rule for References

- Reduce it to the result from functions
  - A reference is like an object with two methods
    - \texttt{get : unit \rightarrow t} reads the value of the ref
    - \texttt{set : t \rightarrow unit} writes the value of the ref
  - Notice that \( t \) occurs both co- and contravariantly
- The right rule:
  \[
  \begin{align*}
  t \leq t' & \quad t' \leq t & \quad \text{or} & \quad t = t' \\
  \text{ref } t \leq \text{ref } t' & \quad \text{ref } t' \leq \text{ref } t
  \end{align*}
  \]
- We say that ref is \textit{nonvariant} or \textit{invariant}
Subtyping Mistakes

- Well-known languages have gotten subtyping wrong
  - Eiffel function types are covariant in the domain
  - The Java array constructor is covariant, not invariant
    - \( S[\ ] \) is a subtype of \( T[\ ] \) if \( S \) is a subtype of \( T \)
- How do they get around the unsoundness?
  - Java adds run-time (dynamic) checks

Type Checking Considerations

- Our type system with subtyping was just our previous type system with the extra rule
  \[
  \frac{A \vdash e : t \quad t \leq t'}{A \vdash e : t'} \quad \text{(Sub)}
  \]
- This rule seems to add non-determinism
  - We can apply it to any term, as often as we like

Type Checking Considerations (cont’d)

- Observation 1: Multiple sequential uses of (Sub)
  can be replaced with a single use
  - Proof: Transitivity of \( \leq \)
- Observation 2: All uses of (Sub) can be pushed down the typing proof to occur just before function application
  - Proof: Omitted
  - Consequence: Can integrate (Sub) into other rules

A Type Checking Algorithm

\[
\begin{align*}
A \vdash c : C(c) & \quad A \vdash x : A(x) \\
A \vdash \lambda x.t : t' & \quad x \in \text{dom}(A)
\end{align*}
\]

- These rules are deterministic
  - Easy to construct an algorithm from them
  - This is sometimes called a syntax-driven system
  - At every step, rule choice determined by syntax

Subtype Polymorphism

- Subtyping (or subclassing from OOP) gives us one kind of polymorphism
  - A polymorphic type represents multiple types
  - For subtyping, we can think of type \( A \) as representing \( A \) and all of \( A \)'s subtypes
  - This is called subtype polymorphism

Limitations of Subtyping

- Suppose \( S \leq T \), and consider the four possible types identity function on \( S \) and \( T \)
  - \( \lambda x : S \rightarrow S \) can’t accept \( T \)'s
  - \( \lambda x : S \rightarrow T \) can’t accept \( T \)'s
  - \( \lambda x : T \rightarrow S \) ill typed
  - \( \lambda x : T \rightarrow T \) can accept \( S \) or \( T \), but returns a \( T \)
- With parametric polymorphism, we can give this type \( \forall \alpha. \alpha \rightarrow \alpha \)
Discussion

- Subtyping is fairly flexible
- Weaker requirements than parametric poly
  - Don’t need to treat parameter with generalizable type as a block box
- Contravariance gets in the way
  - Also a problem with refs

Introduction

- Operational semantics describe program execution
  - E.g., beta reduction $\lambda x.e1 \ e2 \rightarrow e1[e2|x]$
  - But running a program doesn’t make it correct
    - See type checking
- **Axiomatic semantics** describe properties of a program using a logic
  - Examples:
    - This program terminates
    - During the loop, index $i$ is always within bounds of array $a$

Assertions

- **Assertions** describe the state of the program
  - At a particular program point
- We’ll put assertions in { }’s
  - $x := 2 \ \{ x = 2 \}$
    - “After the assignment $x := 2$, $x$ has the value 2”
  - $(x < 0) \ x := x+1 \ \{ x < 1 \}$
    - “If $x$ is less than 0 and we increment $x$, then afterwards $x$ is less than 1”
  - (These and other examples are taken from David Gries, Lecture notes for CS211, Cornell University)

History

- Ultimate goal: Proving programs correct
  - Program verification
- Big names: Turing, Floyd (flow charts), Hoare (imperative programs), Dijskstra, Gries
- Hard to implement for large, realistic systems
  - But see ESC/Java

Hoare Triples

- The notation $\{ Q \} S \{ R \}$ has a special meaning
  - “Execution of statement $S$ begun in a state in which $Q$ is true is guaranteed to terminate, and $R$ is true in the final state.” (Gries)
  - $Q$ is a precondition
  - $R$ is a postcondition
- In practice, we often discard the termination requirement
  - I.e., we only check partial correctness
Another Example

{ true }
if x \leq y → z := x
[] x \geq y → z := y
fi
{ z = \min(x,y) }

- This is a guarded command
  - Non-deterministically pick one guard that is true, and execute the corresponding command
  - Notice if x = y, we may execute either command

Considerations

- { false } S { R } is always valid
- { Q } S { R } says nothing about executing S in a state in which Q is not true
- In general, predicates can contain
  - Program variables (and values before/after procedure)
  - Arithmetic
  - Logic connectives and quantifiers (for all, exists)
  - Other uninterpreted predicates (e.g., facts about arrays)

Weakest Preconditions

- A common reasoning technique
  - We have a postcondition R we want to reach
  - We have a statement S we think will get us there
  - What is the weakest precondition \( \wp(S, R) \) that must be true before S in order to establish R?

  Weakest means makes fewest assumptions about environment
  - \{ true \} x := 2 { x = 2 } vs.
  - \{ y = 55 \} x := 2 { x = 2 }

Assignment Statement

- \( \wp(x := e, R) = R[x/e] \)
  - I.e., \( \{R[x/e]\} x := e \{ R \} \) is always a valid triple
  - Example: compute \( \wp(x := x + 1, x = y) \)

  What happens if e contains
  - Side effects?
  - Pointers?

Conditional Statement

- When does \{ Q \} if B then S1 else S2 { R } hold?
  - \( \{ Q \land B \} S1 \{ R \} \) and \( \{ Q \land \neg B \} S2 \{ R \} \)
  - So, \( \wp(\text{if B then S1 else S2, R}) = (B \Rightarrow \wp(S1, R)) \land (\neg B \Rightarrow \wp(S2, R)) \)

  Example:
  - Compute \( \wp(P; z = \max(x, y)) \) where P is
    if x \geq y → z := x
    [] x \leq y → z := y
    fi

Loops

- When does \{ Q \} while B do S { R } hold?
  - This is hard! We need...
    - \( Q \land \neg B \Rightarrow R \) (0 iterations), or
    - \( Q \land B \) S \( \{ R \land \neg B \} \) (1 iteration), or
    - \( Q \land B \) S \( Q \land B \) S \( \{ R \land \neg B \} \) (2 iterations), or
    - ...
  - There’s no mechanical way to compute Q from R
    - And it’s hard to do even non-mechanically
**Loop Invariants**

- What we need is a predicate $I$ that is true just before, during, and just after the loop executes
  - This is called a loop invariant
- So if we can show $\{I \land B\} S \{I\}$
  - If $I$ is true just before the loop and the loop executes once, then $I$ is true after
- Then we can derive $\{I\}$ while $B$ do $S \{I \land \neg B\}$
  - If $I$ is true before the loop, then $I$ is still true after any number of iterations (and $B$ is false when the loop terminates)

**Example**

- Let $I$ be
  - $0 \leq i \leq 10$ and
  - $s$ is the sum of the first $i$ elements of $a$

```plaintext
{ true }
i := 0; s := 0;
{ I }
while i ≠ 10 do
  { I 㱸i ≠ 10 } s := s + a[i]; i := i + 1; { I }
{ I 㱸i = 10 }
```

**Termination**

- If we’re also concerned about termination, need to prove that loop will eventually exit

- Do this using a *bound function* $t$
  - An expression whose value decreases with each iteration of the loop
  - For our example, could choose $t = 10 - i$

**Proving a Loop (Fully) Correct**

- Want to show $\{Q\}$ while $B$ do $S \{R\}$
  - First, invent a loop invariant $I$ and bound function $t$
  - Then...
  - Prove $Q \Rightarrow I$ ($I$ is true before the loop)
  - Prove $\{I \land B\} S \{I\}$ ($I$ is a loop invariant)
  - Prove $I \land \neg B \Rightarrow R$ (postcondition holds)
  - Prove $t$ decreases with each iteration
  - Prove that $I \land B$ implies $t > 0$ ($t$ is a bound)