1 Subtyping

Principle of safe substitution: \( S <: T \) (S is a subtype of T) means that any term of type S can be safely used in a context where a term of type T is expected.

\[
\begin{align*}
\Gamma \vdash t : S \quad & \Rightarrow \quad \Gamma \vdash t : T \\
S <: T & \quad \Rightarrow \quad \Gamma \vdash t : T
\end{align*}
\]

\text{T-SUB}

Ex: \( \{ x: \text{Int}, y: \text{Bool} \} <: \{ x: \text{Int} \} \).

\text{S-ARROW}

Ex: \( \{ a: \text{Apple} \} \rightarrow \{ j: \text{Juice}, r: \text{Residue} \} <: \{ a: \text{Apple}, p: \text{Pineapple} \} \rightarrow \{ j: \text{Juice} \} \)

\text{S-RCD}

Also, the permutation of fields are allowed. Ex: \( \{ x: \text{Int}, y: \text{Bool}, z: \text{Unit} \} <: \{ z: \text{Unit}, y: \text{Bool}, x: \text{Int} \} \).

The following example, which is beyond the scope of this class, shows peculiar behaviors of subtyping in Java. Type system should preserve type safety.

```java
// wrong subtyping
Integer ary_i[] = new Integer[1];
ary_i[0] = 9;
Object ary_o[] = ary_i;
ary_o[0] = new Object();  // run-time error
for (Object o: ary_o) System.out.println(o);

// correct subtyping
Vector<Integer> vec_i = new Vector<Integer>();
vec_i.add(new Integer(9));
Vector<Object> vec_o = vec_i;  // compile-time error
Vector<? extends Object> vec_so = vec_i;
vec_so.add(new Object());  // compile-time error
for (Object o: vec_so) System.out.println(o);
```

2 The Whole Picture


2. The lexer can use regular expressions to translate texts into tokens. Ex: Num(54) ‘*’ Num(32) ‘+’ Num(1).

3. The parser converts concrete syntax to abstract syntax tree according to context-free grammar.

   - ambiguous grammar like
     \[ A \rightarrow N|A + A | A * A \]
     Ex: either (54*32)+1 or 54*(32+1) is possible.

   - unambiguous grammar like
     \[ A \rightarrow A + B \\
     B \rightarrow N | B * N \]
     Ex: only (54*32)+1 is possible. Multiplication has higher precedence. Multiplication and addition are left associative.

   abstract syntax: \( exp := N|exp + exp|exp * exp \)

4. type checking:

   type safety = progress + preservation
   - progress: a well-typed term is either a value or can take a step of evaluation.
   - preservation: if a well-typed term takes a step of evaluation, the resulting term is also well-typed.
record and union types are similar to \( \land \) and \( \lor \) in logics.
(Curry-Howard correspondence, which is also beyond the scope of this class, gives rise to the correspondence between propositions in logics and types in programming languages.)

5. Semantics is defined over abstract syntax. That is why operators with higher precedence evaluates first in general. Operational semantics is one way to formally define program behaviors.

Ex: recursion can be specified concisely. 
\[
\begin{align*}
v' &= (x : \tau) \Rightarrow [(\text{def } g(x : \tau) : \tau' = e_1 \text{ in } g/g)]e_1 \\
\text{def } g(x : \tau) : \tau' = e_1 \text{ in } e_2 &\Rightarrow [v'/g]e_2
\end{align*}
\]
Ex: The following example, which is beyond the scope of this class, shows changing the value of a variable more than once between two sequence points is an undefined behavior in C/C++. Without formally defined semantics, this code may or may not work as expected.

\[
\texttt{\#define SWAP}(x, y) \ ( (x) \ = \ (y) \ = \ (x) \ = \ (y))
\]

3 Structural Induction

Please do simple parts first: explicitly state induction variable, base cases, induction hypothesis. Then, prove inductive cases.

4 Functional/Imperative Programming

Higher-order functions are convenient! Modern imperative programming languages like Python adopts them.

Ex: simple quick sort can be implemented easily.

\[
\begin{align*}
def qsort\ ((l : \text{List[Int]}) : \text{List[Int]} = 1 \text{ match } \\
\text{case Nil} &\Rightarrow \text{Nil} \\
\text{case x :: xs} &\Rightarrow qsort\ (xs . \text{filter} (\_ <= x)) :: (x :: qsort\ (xs . \text{filter} (\_ > x)))
\end{align*}
\]

The concept of map and reduce (fold) is used by Google.

The memory state is important for imperative programming language, so the evaluation becomes \( \langle M, e \rangle \rightarrow \langle M', e' \rangle \) where memory M is a mapping from address a to value v.

5 Parameter Passing

- call-by-value

\[
\langle M, ((x : \tau) => e)(v) \rangle \rightarrow \langle M, [v/x]e \rangle
\]

- call-by-reference

\[
a \in \text{dom}(M) \\
\langle M, ((\text{ref } x : \tau) => e)(*a) \rangle \rightarrow \langle M, [*a/x]e \rangle
\]

- call-by-name (lazy evaluation):

\[
\langle M, ((\text{name } x : \tau) => e_1)(e_2) \rangle \rightarrow \langle M, [e_2/x]e_1 \rangle
\]

- call-by-value-result

\[
\texttt{\void p}(\texttt{int value_result} x, \texttt{int value_result} y)\{
  \ x = x + 1;
  \ y = y + 1;
\}
\texttt{int} z = 0;
\texttt{p}(z, z);
\texttt{cout} \ll z; \ // \texttt{prints} \ 1
\]
6 Continuation

A simple example

```scala
def fib(n: Int): Int = {
  def helper(n: Int, k: Int => Int): Int = n match {
    case 0 => k(0)
    case 1 => k(1)
    case _ => helper(n-1, (x: Int) => helper(n-2, (y: Int) => k(x+y)))
  }
  helper(n, (x: Int) => x)
}
```

An example of failure continuation

```scala
def substr(sub: String, str: String): Boolean = {
  def helper(subH: String, strH: String, fc: (()) => Boolean): Boolean = {
    if (subH == "") true
    else if (strH == "") false
    else if (subH.charAt(0) == strH.charAt(0))
      helper(subH.drop(1), strH.drop(1), fc)
    else fc()
  }
  helper(sub, str, () => substr(sub, str.drop(1)))
}
```

7 Others

- higher-order abstract syntax: \( (x: \text{Int}) \Rightarrow x + 1 \equiv (y: \text{Int}) \Rightarrow y + 1 \)
- capture-avoiding substitution: \( [(x + 1)/y](x: \text{Int}) \Rightarrow x + y) \equiv (z: \text{Int}) \Rightarrow z + (x + 1) \)
- type equality:

  TYPE Student = RECORD id: INTEGER; name: String; END;
  TYPE Course = RECORD id: INTEGER; name: String; END;

  These two types are different when name type equality is used. ("RECORD" creates a new type name.) These two types are equal when structural type equality is used.

- short circuit evaluation: \( \text{true} \mid (\text{whatever} \&\& \text{it} \mid \text{is}) \Rightarrow \text{true} \)
- dynamic typing: no type checking is performed before evaluation. A runtime type error is raised when evaluation encounters an operations that cannot be applied to the argument values.

\[
\begin{align*}
  v_1 & \neq n_1 \\
  v_1 + e_2 & \Rightarrow \text{typeerror} \\
  v_1 & = (x: \tau) \Rightarrow e \\
  v_1 == e_2 & \Rightarrow \text{typeerror}
\end{align*}
\]

- dynamic scoping:

  The following program prints 3.

  ```java
  int x = 1;
  int f() { return x+1; }
  int g() { int x = 2; print f(); }
  g();
  ```

  The following program is a similar functional version and also print 3.
((x: Int) =>
  ((f: Unit => Int) =>
    ((x: Int) =>
      print(f())
    )(2)
  )() => x+1
)(1)