COS320: Compiling Techniques

Zak Kincaid

March 16, 2022
Subtyping
Extrinsic (sub)types

- **Extrinsic view** (Curry-style): a type is a *property* of a term. Think:
  - There is some set of *values*

    ```
    type value =
      | VInt of int
      | VBool of bool
    ```

  - Each type corresponds to a subset of values

    ```
    let typ_int = function
      | VInt _ -> true
      | _ -> false
    let typ_bool = function
      | VBool _ -> true
      | _ -> false
    ```

  - A term has type \( t \) if it evaluates to a value of type \( t \)
Extrinsic (sub)types

- **Extrinsic view** (Curry-style): a type is a *property* of a term. Think:
  - There is some set of *values*

```
type value =
  | VInt of int
  | VBool of bool
```

- Each type corresponds to a subset of values

```
let typ_int = function
  | VInt _ -> true
  | _ -> false
let typ_bool = function
  | VBool _ -> true
  | _ -> false
```

- A term has type $t$ if it evaluates to a value of type $t$

```
let typ_nat = function
  | VInt x -> x >= 0
  | _ -> false
```
Subtyping

- Call \( s \) a **subtype** of type \( t \) if the values of type \( s \) is a subset of values of type \( t \)
- A subtyping judgement takes the form \( \vdash s <: t \)
  - “The type \( s \) is a subtype of \( t \)”
  - Liskov substitution principle: if \( s \) is a subtype of \( t \), then terms of type \( t \) can be replaced with terms of type \( s \) without breaking type safety.
Subtyping

- Call $s$ a **subtype** of type $t$ if the values of type $s$ is a subset of values of type $t$.
- A subtyping judgement takes the form $\vdash s <: t$:
  - “The type $s$ is a subtype of $t$”
  - Liskov substitution principle: if $s$ is a subtype of $t$, then terms of type $t$ can be replaced with terms of type $s$ without breaking type safety.

<table>
<thead>
<tr>
<th>Subsumption</th>
<th>Transitivity</th>
<th>Reflexivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash e : s$</td>
<td>$\vdash t_1 &lt;: t_2$</td>
<td>$\vdash t &lt;: t$</td>
</tr>
<tr>
<td>$\vdash s &lt;: t$</td>
<td>$\vdash t_2 &lt;: t_3$</td>
<td></td>
</tr>
<tr>
<td>$\vdash n a t &lt;: i n t$</td>
<td>$\vdash t_1 &lt;: t_3$</td>
<td></td>
</tr>
</tbody>
</table>

- Subsumption: if $s$ is a subtype of $t$, then terms of type $s$ can be used as if they were terms of type $t$. 
Casting

- **Upcasting**: Suppose $s <: t$ and $e$ has type $s$. May safety cast $e$ to type $t$.
  - Subsumption rule: upcast implicitly (C, C++, Java, ...)
    - Not necessarily a no-op
  - In OCaml: upcast $e$ to $t$ with $(e :> t)$ (important for type inference!)
- **Downcasting**: Suppose $s <: t$ and $e$ has type $t$. May **not** safety cast $e$ to type $s$.
  - **Checked downcasting**: check that downcasts are safe at runtime (Java, dynamic_cast in C++)
    - Type safe – throwing an exception is not the same as a type error
  - **Unchecked downcasting**: static_cast in C++
  - **No downcasting**: OCaml
Extending the subtype relation

**Tuple**

\[ \frac{\vdash t_1 <: s_1 \quad \ldots \quad \vdash t_n <: s_n}{\vdash t_1 \cdots t_n <: s_1 \cdots s_n} \]

**List**

\[ \frac{\vdash s <: t}{\vdash s \text{ list} <: t \text{ list}} \]

**Array**

\[ \frac{\vdash s <: t}{\vdash s \text{ array} <: t \text{ array}} \]
Extending the subtype relation

**Tuple**

\[ \frac{}{t_1 <: s_1 \ldots t_n <: s_n} \]

\[ \frac{}{t_1 \ast \cdots \ast t_n <: s_1 \ast \cdots \ast s_n} \]

**List**

\[ \frac{}{s <: t} \]

\[ \frac{}{s \text{ list} <: t \text{ list}} \]

**Array**

\[ \frac{}{s <: t} \]

\[ \frac{}{s \text{ array} <: t \text{ array}} \]

- Array subtyping rule is **unsound** (Java!)
  
  Let \( \Gamma = [x \mapsto \text{nat array}] \)

\[
\begin{array}{c}
\Gamma \vdash x : \text{nat array} \\
\hline
\text{VAR} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash x : \text{nat array} \\
\hline
\text{ARRAY} \\
\end{array}
\]

\[
\begin{array}{c}
\text{nat array} <: \text{int array} \\
\hline
\text{NATINT} \\
\end{array}
\]

\[
\begin{array}{c}
	ext{nat} <: \text{int} \\
\hline
\text{NATINT} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash 0 : \text{nat} \\
\hline
\text{NAT} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash -1 : \text{int} \\
\hline
\text{INT} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash x[0] := -1 \\
\hline
\text{ASSN} \\
\end{array}
\]
Width subtying

```cpp
type point2d { x : int, y : int }
type point3d { x : int, y : int, z : int }

• point2d <: point3d or point3d <: point2d?
```
Width subtyping

```python
type point2d { x : int, y : int }
```

```python
type point3d { x : int, y : int, z : int }
```

- `point2d <: point3d` or `point3d <: point2d`?
  - Liskov: Every 3-dimensional point can be used as a 2-dimensional point (`point3d <: point2d`)
Width subtyping

```verbatim
 type point2d { x : int, y : int }
 type point3d { x : int, y : int, z : int }
```

- `point2d <: point3d` or `point3d <: point2d`?
  - Liskov: Every 3-dimensional point can be used as a 2-dimensional point (`point3d <: point2d`)

  \[
  \text{RECORDWIDTH} \\
  \vdash \{ \text{lab}_1 : s_1; \ldots; \text{lab}_m : s_m \} <: \{ \text{lab}_1 : s_1; \ldots; \text{lab}_n : s_n \} \quad n < m
  \]
Easy!

- \( s <: t \) means \( \text{sizeof}(t) \leq \text{sizeof}(s) \), but field positions are the same (\textit{e.lab} compiled the same way, whether \( e \) has type \( s \) or type \( t \))

- e.g., \( \text{pt} -> y \) is \( \ast(\text{pt} + \text{sizeof(int)}) \), regardless of whether \( \text{pt} \) is 2d or 3d
Depth subtyping

\[
\begin{align*}
\text{type } & \text{nat\_point } \{ x : \text{nat}, y : \text{nat} \} \\
\text{type } & \text{int\_point } \{ x : \text{int}, y : \text{int} \}
\end{align*}
\]

- nat\_point <: int\_point or int\_point <: nat\_point?
Depth subtyping

```plaintext
type nat_point { x : nat, y : nat }
type int_point { x : int, y : int }

- nat_point <: int_point or int_point <: nat_point?
  - Liskov: nat_point <: int_point but only for immutable records!
```
Depth subtyping

\[
\begin{align*}
\text{type } \text{nat\_point} & \{ x : \text{nat}, y : \text{nat} \} \\
\text{type } \text{int\_point} & \{ x : \text{int}, y : \text{int} \}
\end{align*}
\]

- \text{nat\_point} <: \text{int\_point} or \text{int\_point} <: \text{nat\_point}?
- Liskov: \text{nat\_point} <: \text{int\_point} \text{ but only for immutable records!}

**RecordDepth**

\[
\begin{align*}
\vdash s_1 <: t_1 & \quad \ldots \quad \vdash s_n <: t_n \\
\vdash \{ \text{lab}_1 : s_1; \ldots; \text{lab}_n : s_n \} <: \{ \text{lab}_1 : t_1; \ldots; \text{lab}_n : t_n \}
\end{align*}
\]
Compiling depth subtyping

Easy!

- \( s <: t \) means \( \text{sizeof}(s) = \text{sizeof}(t) \), so field positions are the same.

- \( \text{pt is a nat_point: pt->y} = *(pt + \text{sizeof(nat)}) \)
- \( \text{pt is an int_point: pt->y} = *(pt + \text{sizeof(int)}) \)
- \( \text{sizeof(int)} = \text{sizeof(nat)} \)
Compiling width+depth subtyping

- Width + depth: `pyramid <: rectangle` (with immutable records)

```plaintext
type point2d { x : int, y : int }
type point3d { x : int, y : int, z : int }
type rectangle = { tl : point2d, br : point2d }
type pyramid = { tl : point3d, br : point3d, top : point3d }
```

```
<table>
<thead>
<tr>
<th>pyramid</th>
<th>rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl.x</td>
<td>tl.x</td>
</tr>
<tr>
<td>tl.y</td>
<td>tl.y</td>
</tr>
<tr>
<td>tl.z</td>
<td>br.x</td>
</tr>
<tr>
<td>br.x</td>
<td>br.x</td>
</tr>
<tr>
<td>br.y</td>
<td>br.y</td>
</tr>
<tr>
<td>br.z</td>
<td></td>
</tr>
<tr>
<td>top.x</td>
<td></td>
</tr>
<tr>
<td>top.y</td>
<td></td>
</tr>
<tr>
<td>top.z</td>
<td></td>
</tr>
</tbody>
</table>
```

incompatible!
Compiling width+depth subtyping

```haskell
type point2d { x : int, y : int }
type point3d { x : int, y : int, z : int }
type rectangle = { tl : point2d, br : point2d }
type pyramid = { tl : point3d, br : point3d, top : point3d }
```

- **Width + depth**: `pyramid <: rectangle` (with immutable records)

- Add an indirection layer!
Function subtyping

\[
\text{\textbf{FUN}}
\]

\[
\begin{align*}
\vdash & \ ? < : \ ? & \vdash & \ ? < : \ ? \\
\vdash & t_1 \to t_2 < : s_1 \to s_2 
\end{align*}
\]
In the function subtyping rule, we say that the argument type is **contravariant**, and the output type is **covariant**

Some languages (Eiffel, Dart) have **covariant** argument subtyping. Not type-safe!
Type inference with subtyping
In the presence of the subsumption rule, a term may have more than one type. Which type should we infer?

- Subtyping forms a preorder relation (REFLEXIVITY and TRANSITIVITY)
- Typically (but not necessarily), subtyping is a partial order
  - A partial order is a binary relation that is reflexive, transitive, and antisymmetric
    - If \(a <: b\) and \(b <: a\), then \(a = b\)
  - A preorder that is not a partial order: graph reachability (\(u \leq v\) iff there is a path from \(u\) to \(v\))
In the presence of the subsumption rule, a term may have more than one type. Which type should we infer?

- Subtyping forms a preorder relation (REFLEXIVITY and TRANSITIVITY)
- Typically (but not necessarily), subtyping is a partial order
  - A partial order is a binary relation that is reflexive, transitive, and antisymmetric
    If $a <: b$ and $b <: a$, then $a = b$
  - A preorder that is not a partial order: graph reachability ($u \leq v$ iff there is a path from $u$ to $v$)

- Given a context $\Gamma$ and expression $e$, goal is to infer least type $t$ such that $\Gamma \vdash e : t$ is derivable.
Subsumption is not syntax-directed
  
  Type inference can’t use program syntax to determine when to use subsumption rule
Subsumption is not syntax-directed
  Type inference can’t use program syntax to determine when to use subsumption rule
Do not use subsumption! Integrate subsumption into other inference rules. E.g.,

\[
\begin{align*}
\text{TYP\_CArr} \\
\Gamma \vdash e_1 : t & \quad \ldots \quad & \Gamma \vdash e_n : t \\
\Gamma \vdash \text{new } t[]\{e_1, \ldots, e_n\} : t[]
\end{align*}
\]
Subsumption is not syntax-directed
- Type inference can't use program syntax to determine when to use subsumption rule
- Do not use subsumption! Integrate subsumption into other inference rules. E.g.,

\[
\text{Typ}_\text{CARR}
\]
\[
\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_n : t_n \quad \vdash t_1 <: t \quad \ldots \quad \vdash t_n <: t
\]
\[
\Gamma \vdash \text{new } t[]\{e_1, \ldots, e_n\} : t[]
\]
\[ \text{IF} \]
\vspace{-2ex}
\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{bool} \\
\Gamma \vdash e_2 : t \\
\Gamma \vdash e_3 : t \\
\end{array}
\]
\[
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\]

Problem: what is $t$?

Say that $t$ is a least upper bound of $t_2$ and $t_3$ if $t < t_2$ and $t < t_3$:

For any type $t'$ such that $t_2 < t'$ and $t_3 < t'$, we have $t < t'$ (If $<$ is a partial order, least upper bound is unique).

Take $t$ to be the least upper bound of $t_2$ and $t_3$.

Java: every pair of types has a least upper bound • Least upper bound is the least common ancestor in class hierarchy

C++: with multiple inheritance, classes can have multiple upper bounds, none of which is least • Require $t_2 < t_3$ or $t_3 < t_2$.

OCaml: no subsumption rule. Must explicitly upcast each side of the branch.
\[
\begin{align*}
\text{if} & \quad (\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t) \\
\hline 
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\end{align*}
\]
If
\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t \]
\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]

- Problem: what is \( t \)?

Say that \( t \) is a least upper bound of \( t_2 \) and \( t_3 \) if
\[ t_2 <: t \quad \text{and} \quad t_3 <: t \]
For any type \( t' \) such that \( t_2 <: t' \) and \( t_3 <: t' \), we have \( t <: t' \) (If \(<: \) is a partial order, least upper bound is unique)

Take \( t \) to be the least upper bound of \( t_2 \) and \( t_3 \)

Java: every pair of types has a least upper bound
- Least upper bound is the least common ancestor in class hierarchy

C++: with multiple inheritance, classes can have multiple upper bounds, none of which is least
- Require \( t_2 <: t_3 \) or \( t_3 <: t_2 \)

OCaml: no subsumption rule. Must explicitly upcast each side of the branch.
If
\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t \]
\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]

- Problem: what is \( t \)?
- Say that \( t \) is a **least upper bound** of \( t_2 \) and \( t_3 \) if
  1. \( t_2 <: t \) and \( t_3 <: t \)
  2. For any type \( t' \) such that \( t_2 <: t' \) and \( t_3 <: t' \), we have \( t <: t' \)

(If \(<: \) is a partial order, least upper bound is unique)
If
\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t \]
then
\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]

- Problem: what is \( t \)?
- Say that \( t \) is a **least upper bound** of \( t_2 \) and \( t_3 \) if
  1. \( t_2 <: t \) and \( t_3 <: t \)
  2. For any type \( t' \) such that \( t_2 <: t' \) and \( t_3 <: t' \), we have \( t <: t' \)

(If \( <: \) is a partial order, least upper bound is unique)
- Take \( t \) to be the least upper bound of \( t_2 \) and \( t_3 \)

Java: every pair of types has a least upper bound
- Least upper bound is the least common ancestor in class hierarchy

C++: with multiple inheritance, classes can have multiple upper bounds, none of which is least
- Require \( t_2 <: t \) or \( t_3 <: t \)

OCaml: no subsumption rule. Must explicitly upcast each side of the branch.
\[
\begin{align*}
\text{If} & \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t \\
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\end{align*}
\]

- **Problem**: what is \( t \)?

- **Say that** \( t \) is a *least upper bound* of \( t_2 \) and \( t_3 \) if
  1. \( t_2 <: t \) and \( t_3 <: t \)
  2. For any type \( t' \) such that \( t_2 <: t' \) and \( t_3 <: t' \), we have \( t <: t' \)

  (If \(<: \) is a partial order, least upper bound is unique)

- **Take** \( t \) to be the least upper bound of \( t_2 \) and \( t_3 \)
  - **Java**: every pair of types has a least upper bound
    - Least upper bound is the least common ancestor in class hierarchy
  - **C++**: with multiple inheritance, classes can have multiple upper bounds, none of which is least
  - **OCaml**: no subsumption rule. Must explicitly upcast each side of the branch.

- **Java**: every pair of types has a least upper bound
  - Least upper bound is the least common ancestor in class hierarchy
\[ \text{If} \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 \mathrel{<:} t \quad \vdash t_3 \mathrel{<:} t \]

\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]

- **Problem**: what is \( t \)?
- **Say that** \( t \) is a *least upper bound* of \( t_2 \) and \( t_3 \) if
  1. \( t_2 \mathrel{<:} t \) and \( t_3 \mathrel{<:} t \)
  2. For any type \( t' \) such that \( t_2 \mathrel{<:} t' \) and \( t_3 \mathrel{<:} t' \), we have \( t \mathrel{<:} t' \)

  (If \( \mathrel{<:} \) is a partial order, least upper bound is unique)
- **Take** \( t \) to be the least upper bound of \( t_2 \) and \( t_3 \)
  - *Java*: every pair of types has a least upper bound
    - Least upper bound is the least common ancestor in class hierarchy
  - *C++*: with multiple inheritance, classes can have multiple upper bounds, none if which is *least*
    - Require \( t_2 \mathrel{<:} t_3 \) or \( t_3 \mathrel{<:} t_2 \)
\[
\text{If } \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t
\]
\[
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\]

- **Problem**: what is \( t \)?
- **Say that** \( t \) is a **least upper bound** of \( t_2 \) and \( t_3 \) if
  1. \( t_2 <: t \) and \( t_3 <: t \)
  2. For any type \( t' \) such that \( t_2 <: t' \) and \( t_3 <: t' \), we have \( t <: t' \)

(If \(<: \) is a partial order, least upper bound is unique)
- **Take** \( t \) to be the least upper bound of \( t_2 \) and \( t_3 \)
  - **Java**: every pair of types has a least upper bound
    - Least upper bound is the least common ancestor in class hierarchy
  - **C++**: with multiple inheritance, classes can have multiple upper bounds, none if which is least
    - Require \( t_2 <: t_3 \) or \( t_3 <: t_2 \)
  - **OCaml**: no subsumption rule. Must explicitly upcast each side of the branch.
Looking ahead

- Next week:
  - Compiling with types, start on optimization
  - HW4: Oat v2
    - Need to implement a type-checker (among other things)
    - (Oat v2 has subtyping)

- A few weeks later: compiling object-oriented languages
  - Subtyping plays a prominent role