Subtyping
Extrinsic (sub)types

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

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

- Each type corresponds to a subset of values

```ocaml
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 $t$

- *Types may overlap.*

```
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.
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.

### Subsumption

<table>
<thead>
<tr>
<th>NATINT</th>
<th>SUBSUMPTION</th>
<th>TRANSITIVITY</th>
<th>REFLEXIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash e : s$</td>
<td>$\vdash s &lt;: t$</td>
<td>$\vdash t_1 &lt;: t_2$</td>
<td>$\vdash t_2 &lt;: t_3$</td>
</tr>
<tr>
<td>$\vdash \text{nat} &lt;: \text{int}$</td>
<td>$\Gamma \vdash e : t$</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, Java, C++, ...)
    - 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**

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

**LIST**

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

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

**ARRAY**

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

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

\[ \text{TUPLE} \]
\[ \frac{\Gamma \vdash t_1 <: s_1 \quad \ldots \quad \Gamma \vdash t_n <: s_n}{\Gamma \vdash t_1 \times \ldots \times t_n <: s_1 \times \ldots \times s_n} \]

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

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

- Array subtyping rule is **unsound** (Java!)

Let \( \Gamma = [x \mapsto \text{nat array}] \)

\[ \text{VAR} \]
\[ \frac{\Gamma \vdash x : \text{nat array}}{\Gamma \vdash x : \text{int array}} \]

\[ \text{ARRAY} \]
\[ \frac{\Gamma \vdash \text{nat array} <: \text{int array}}{\Gamma \vdash x : \text{int array}} \]

\[ \text{ASSN} \]
\[ \frac{\Gamma \vdash x[0] := -1}{\Gamma \vdash x[0] := -1} \]
Width subtyping

```plaintext

**Width subtyping**

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

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

```plaintext
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`)
Width subtyping

```plaintext

- `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`)

  \[
  \text{RECORDWIDTH}
  \]

  \[
  \vdash \{ \text{lab}_1 : s_1; \ldots; \text{lab}_m : s_m \} <: \{ \text{lab}_1 : s_1; \ldots; \text{lab}_n : s_n \} \quad n < m
  \]
Compiling width subtyping

Easy!

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

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

- \texttt{type nat\_point} { \texttt{x : nat, y : nat} }
- \texttt{type int\_point} { \texttt{x : int, y : int} }

- \texttt{nat\_point <: int\_point or int\_point <: nat\_point?}
Depth subtyping

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

\text{RECORDDEPTH}
\begin{align*}
\vdash s_1 <: t_1 & \quad \ldots \quad \vdash s_n <: t_n \\
\vdash \{ \text{lab}_1 : s_n; \ldots; \text{lab}_m : 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.

- **pt is a nat_point**: $\text{pt->y} \text{ is } *(\text{pt } + \text{sizeof(nat)})$

- **pt is an int_point**: $\text{pt->y } \text{is } *(\text{pt } + \text{sizeof(int)})$

- $\text{sizeof(int)} = \text{sizeof(nat)}$
Compiling width+depth subtyping

- **Width + depth**: \( \text{pyramid} <: \text{rectangle} \) (with immutable records)

```
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 }
```
Compiling width+depth subtyping

```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 }
```

- **Width + depth:** \( \text{pyramid} <: \text{rectangle} \) (with immutable records)

- Add an indirection layer!
Function subtyping

\[
\text{FUN} \\
\vdash s_1 <: t_1 \quad \vdash t_2 <: s_2 \\
\vdash t_1 \rightarrow t_2 <: s_1 \rightarrow s_2
\]

- 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.,

\[
\text{TYP\_CARR} \\
\begin{array}{c}
\Gamma \vdash e_1 : t \\
\vdots \\
\Gamma \vdash e_n : t \\
\hline
\Gamma \vdash \text{new } t[]\{e_1, \ldots, e_n\} : t[]
\end{array}
\]
• 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.,

\[
\text{TYP\_CARR} \\
\begin{array}{ccc}
\Gamma \vdash e_1 : t_1 & \ldots & \Gamma \vdash e_n : t_n & \vdash t_1 <: t & \ldots & \vdash t_n <: t \\
\hline
\Gamma \vdash \text{new } t[]\{e_1, \ldots, e_n\} : t[]
\end{array}
\]
\[
\begin{align*}
\text{IF} & \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t \\
\quad & \quad \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\end{align*}
\]
\[
\text{if } \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t \\
\hline \\
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\]
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 \( t \)?
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)
\[
\begin{align*}
\text{If} & \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t \\
\hline
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\end{align*}
\]

- 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.
\[
\begin{align*}
\text{If} & \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \\
& \quad \VDash t_2 <: t \quad \VDash t_3 <: t \\
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\end{align*}
\]

- **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
  - **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)
- 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 \( t_3 <: t_2 \)
\[
\text{If } \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad \vdash t_2 <: t \quad \vdash t_3 <: t
\]
\[
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\]

- **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.