COS320: Compiling Techniques

Zak Kincaid

March 26, 2024
Oat v2

- Specified by a (fairly large) type system
  - \( \sim 20 \) judgements, \( \sim 80 \) inference rules
  - Invest some time in making sure you understand how to read them
- Adds several features to the Oat language:
  - Memory safety
    - *nullable* and *non-null* references. Type system enforces no null pointer dereferences.
    - Run-time array bounds checking (like Java, OCaml)
  - Mutable record types
  - Subtyping
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*

```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 $t$

- Types may overlap.

```ocaml
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>NATINT</th>
<th>SUBSUMPTION</th>
<th>TRANSITIVITY</th>
<th>REFLEXIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash \text{nat} &lt;: \text{int}$</td>
<td>$\Gamma \vdash e : s \quad \vdash s &lt;: t$</td>
<td>$\Gamma \vdash e : t$</td>
<td>$\vdash t_1 &lt;: t_2 \quad \vdash t_2 &lt;: t_3$</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 (e.g., upcast int to float)
  - 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**
\[
\frac{\vdash t_1 <: s_1 \quad \cdots \quad \vdash t_n <: s_n}{\vdash t_1 \cdots t_n <: s_1 \cdots s_n}
\]

**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**
\[\vdash t_1 <: s_1 \quad \ldots 
\quad \vdash t_n <: s_n \quad \vdash t_1 \ast \cdots \ast t_n <: s_1 \ast \cdots \ast s_n\]

**LIST**
\[\vdash s <: t \quad \vdash s \text{ list} <: t \text{ list}\]

**ARRAY**
\[\vdash s <: t \quad \vdash s \text{ array} <: t \text{ array}\]

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

\[\Gamma \vdash x : \text{nat array} \quad \text{VAR} \quad \Gamma, x : \text{nat} \vdash x : \text{int array} \quad \text{ARRAY} \quad n : \text{nat} \vdash n : \text{int} \quad \Gamma, n : \text{nat} \vdash n \vdash 0 : \text{nat} \quad \text{NAT} \quad \Gamma, 0 : \text{nat} \vdash 0 : \text{int} \quad \text{INT} \quad \Gamma \vdash x[0] := -1 \quad \text{ASSN}\]
Width subtyping

```

```
Width subtyping

```kotlin
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 $t$)

```
point3d
x
y
z
```

```
point2d
x
y
```

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

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

- \(\text{nat}_\text{point} <: \text{int}_\text{point}\) or \(\text{int}_\text{point} <: \text{nat}_\text{point}\)?
type nat_point { x : nat, y : nat }

* Liskov: nat_point <: int_point * but only for immutable records!
Depth subtyping

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

\[
\text{RecordDepth} \quad \frac{\vdash s_1 <: t_1 \quad \ldots \quad \vdash s_n <: t_n}{\vdash \{lab_1 : s_1; \ldots; lab_n : s_n\} <: \{lab_1 : t_1; \ldots; lab_n : t_n\}}
\]
Compiling depth subtyping

Easy!

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

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

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)

![Diagram showing subtyping relationship]

Incompatible!
Compiling width+depth subtyping

```
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

\[
\begin{align*}
\text{FUN} & \\
\vdash \tau <: ? & \quad \vdash \theta <: ? \\
\hline
\vdash t_1 \to t_2 <: s_1 \to s_2
\end{align*}
\]
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.,

\[
\text{\textsc{Typ\_CArr}} \\
\Gamma \vdash e_1 : t \quad \ldots \quad \Gamma \vdash e_n : t \\
\Gamma \vdash \text{new } \, t[\{e_1, \ldots, e_n\} : t[][]
\]
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.,

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

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

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

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

- **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** \( t_3 <: t_2 \)
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

- Compiling up:
  - 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