COS320: Compiling Techniques

Zak Kincaid

March 26, 2020

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
- *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 priciple: 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 priciple: if s is a subtype of t, then terms of type t can be replaced with terms of type s without breaking type safety.

NatInt
$$\frac{}{\vdash \texttt{nat} <: \texttt{int}}$$

Subsumption
$$\frac{\Gamma \vdash e : s \qquad \vdash s <: t}{\Gamma \vdash e : t}$$

Transitivity
$$\frac{\vdash t_1 <: t_2 \qquad \vdash t_2 <: t_3}{\vdash t_1 <: t_3}$$

Reflexivity
$$\frac{}{\vdash t <: t}$$

- 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

$$\frac{\vdash t_1 <: s_1 \qquad \ldots \qquad \vdash t_n <: s_n}{\vdash t_1 * \ldots * t_n <: s_1 * \ldots * s_n}$$

List

$$\frac{\vdash s <: t}{\vdash s\ \texttt{list} <: t\ \texttt{list}}$$

Array

$$\frac{\vdash s <: t}{\vdash s\ \texttt{array} <: t\ \texttt{array}}$$

Extending the subtype relation

Tuple

$$\frac{\vdash t_1 <: s_1 \quad \dots \quad \vdash t_n <: s_n}{\vdash t_1 * \dots * t_n <: s_1 * \dots * s_n}$$

List

$$\frac{\vdash s <: t}{\vdash s \texttt{ list} <: t \texttt{ list}}$$

Array

$$\frac{\vdash s <: t}{\vdash s \texttt{ array} <: t \texttt{ array}}$$

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

$$\textsc{Assn}\ \frac{\textsc{Sub}\ \dfrac{\textsc{Var}\ \dfrac{}{\Gamma \vdash x : \texttt{nat array}} \quad \textsc{Array}\ \dfrac{\textsc{NatInt}\ \dfrac{}{\texttt{nat} <: \texttt{int}}}{\texttt{nat array} <: \texttt{int array}}}{\Gamma \vdash x : \texttt{int array}} \quad \textsc{Nat}\ \dfrac{}{\Gamma \vdash 0 : \texttt{nat}} \quad \textsc{Int}\ \dfrac{}{\Gamma \vdash -1 : \texttt{int}}}{\Gamma \vdash x[0] := -1}$$

Width subtying

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

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

Width subtying

```
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 subtying

```
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`)

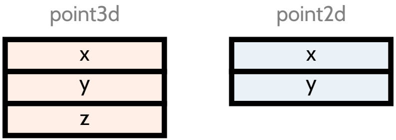
RecordWidth

$$\frac{}{\vdash \{\mathit{lab}_1 : s_1; \ldots; \mathit{lab}_m : s_m\} <: \{\mathit{lab}_1 : s_1; \ldots; \mathit{lab}_n : s_n\}}\ n < m$$

Compiling width subtyping

Easy!

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



- e.g., `pt->y` is `*(pt + sizeof(int))`, regardless of whether `pt` is 2d or 3d

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`?

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

```
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!*

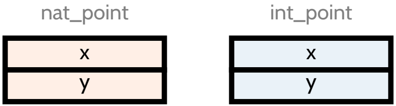
RECORDDEPTH

$$\frac{\vdash s_1 <: t_1 \qquad \ldots \qquad \vdash s_n <: t_n}{\vdash \{\mathit{lab}_1 : s_n; \ldots; \mathit{lab}_m : s_n\} <: \{\mathit{lab}_1 : t_1; \ldots; \mathit{lab}_n : t_n\}}$$

Compiling depth subtyping

Easy!

- $s <: t$ means `sizeof`$(s) =$ `sizeof`(t), so field positions are the same.

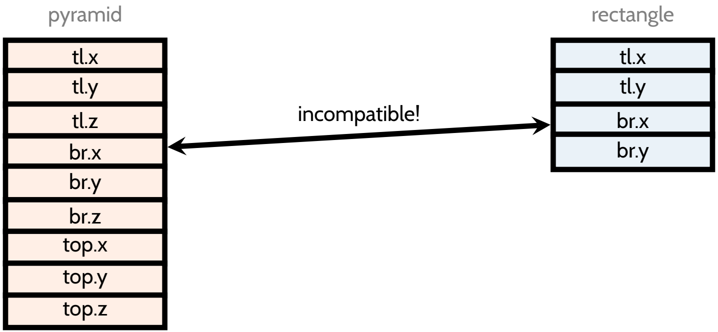


- `pt` is a `nat_point`: `pt->y` is `*(pt + sizeof(nat))`
- `pt` is an `int_point`: `pt->y` is `*(pt + sizeof(int))`
- `sizeof(int) = sizeof(nat)`

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 }
```

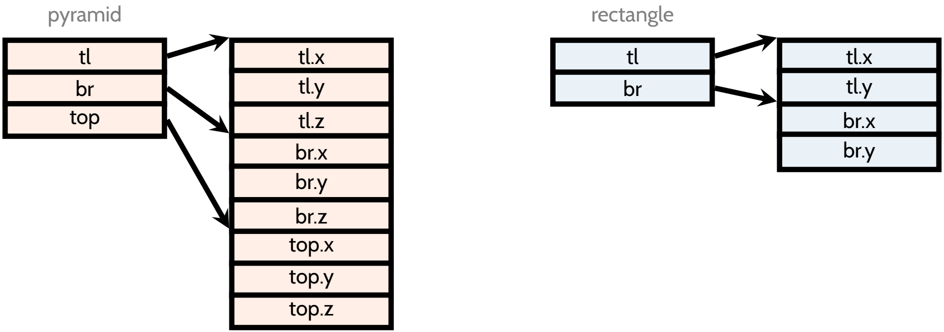
- Width + depth: `pyramid <: rectangle` (with immutable records)



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

$$\textsc{Fun}\ \frac{\vdash s_1 <: t_1 \qquad \vdash t_2 <: s_2}{\vdash t_1 \to t_2 <: s_1 \to s_2}$$

- 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

SUBSUMPTION

$$\frac{\Gamma \vdash e : s \qquad \vdash s <: t}{\Gamma \vdash e : t}$$

- 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)

SUBSUMPTION

$$\frac{\Gamma \vdash e : s \qquad \vdash s <: t}{\Gamma \vdash e : t}$$

- 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 Γ 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.,

TYP_CARR

$$\frac{\Gamma \vdash e_1 : t \qquad \ldots \qquad \Gamma \vdash e_n : t}{\Gamma \vdash \texttt{new t[]\{}e_1, \ldots, e_n\texttt{\}} \ \texttt{:} \ \texttt{t[]}}$$

- 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.,

TYP_CARR

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \ldots \qquad \Gamma \vdash e_n : t_n \qquad \vdash t_1 <: t \qquad \ldots \qquad \vdash t_n <: t}{\Gamma \vdash \texttt{new t[]\{}e_1, \ldots, e_n\texttt{\} : t[]}}$$

IF

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

If

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad \vdash t_2 <: t \qquad \vdash t_3 <: t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

$$\frac{\textsc{If} \quad \Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad \vdash t_2 <: t \qquad \vdash t_3 <: t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

- Problem: what is t?

If

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad \vdash t_2 <: t \qquad \vdash t_3 <: t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

- 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

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad \vdash t_2 <: t \qquad \vdash t_3 <: t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

- 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

$$\textsc{If}\quad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad \vdash t_2 <: t \qquad \vdash t_3 <: t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

- 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

If

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad \vdash t_2 <: t \qquad \vdash t_3 <: t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

- 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

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad \vdash t_2 <: t \qquad \vdash t_3 <: t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

- 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.