Compiling Techniques : Subtyping

Extrinsic subtypes

type is property of term.

- types as subsets of values
- arises when types overlap
  (e.g.) Int ⊇ Nat, 2ℤ ∩ ℕ.

typ_nat = function |

- type S is subtype of t iff
  {val : S} are subset of {val : t}

Judgement: $\vdash S <: t$

"S subtype of t"

Liskov substitution principle

if $S <: t$ then terms of type S can be used as terms of type t w.o. breaking type safety.

"terms of type t can be replaced w/ terms of type S"

System:

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

$$\star\ \text{Subsumption}\ \frac{\Gamma \vdash e : s \qquad \vdash s <: t}{\Gamma \vdash e : t}$$ (Impl. of Liskov substitution)

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

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

Casting Suppose $s <: t$ and e has type s.

May safely cast e to type t.

- Use subsumption to upcast implicitly.

* Ocaml doesn't. Upcast $e \to t$ $(e :> t)$, important for type inference.

Downcasting. Suppose $s <: t$ and $e : t$. May not safely cast e to $\overset{\text{type}}{\wedge}{}^{s}$. (Not safe)

- Checked downcasting: at runtime check when safe (dynamic_cast in C++)
  - typesafe, throw exception when type mismatch
- Unchecked downcasting: static_cast in C++.
- No downcasting in Ocaml in general.

Extending subtype relation for data structures ③

Tuple $\dfrac{\vdash t_1 <: s_1 \quad \ldots \quad \vdash t_n <: s_n}{\vdash t_1 * \ldots * t_n <: s_1 * \ldots * s_n}$

List $\dfrac{\vdash s <: t}{\vdash s\ list <: t\ list}$

*: array $\dfrac{\vdash s <: t}{\vdash s\ array <: t\ array}$ (Not necessarily right... arrays are mutable).

there's a thing we can do w/ a t array: write t to some cell.

- Array subtyping is unsound.

let $\Gamma = [x \mapsto nat\ array]$

$$\text{Sub}\ \dfrac{\text{Var}\ \dfrac{}{\Gamma \vdash x : nat\ array} \quad \text{array}\ \dfrac{\text{Nat}\ \dfrac{}{nat <: int}}{nat\ array <: int\ array}}{\Gamma \vdash x : int\ array} \quad \dfrac{\text{Nat}}{\Gamma \vdash 0 : int}$$

$$\Gamma \vdash x[0] := -1$$

(Java has this subtyping rule).

(requires runtime checks of array writes

Subtyping records : width subtyping

ex

type p2d { x:int, y:int }

type p3d { x:int, y:int, z:int }

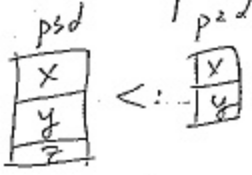
* p3d <: p2d (Use subsumption rule).

"p3d includes everything p2d has"

$$\Rightarrow \quad \text{RecWidth} \ \frac{}{\{lab_1 : S_1; \ldots; lab_m : S_m\} <: \{lab_1 : S_1; \ldots; lab_n : S_n\}} \ n<m$$

Compiling this is easy.

$S <: t \triangleq sizeof(t) \leq sizeof(S)$ ∧ field positions are the same.



depth subtyping

type nat_point { x : nat , y : nat }

type int_point { x : int , y : int }

* : nat_point <: int_point

Only for immutable records!

↓

Just like tuple rule vs. array rule!

$$\text{RecDepth}\ \frac{\vdash S_1 <: t_1 \quad \cdots \quad \vdash S_n <: t_n}{\vdash \{lab_1 : S_1; \ldots; lab_n : S_n\} <: \{lab_1 : t_1; \ldots lab_n : t_n\}}$$

- easy to compile:

$S <: t \triangleq$ sizeof(S) = sizeof(t)

+ field positions are the same.

Compiling w/ depth + width

type p2d ...

type p3d ...

type rect { t_1: p2d, br: p3d }

type py { t_1: p3d, br: p3d, top: p3d }

width + depth: py <: rect (w/ immutable recs)

→ but doesn't work if we "flatten" the recs.

→ Use pointer structure

Function subtyping

$$\text{Fun}\ \frac{\vdash (S_1 <: T_1) \quad \vdash (T_2 <: S_2)}{\vdash (T_1 \to T_2) <: (S_1 \to S_2)}$$

↑

Interesting ... "flipped"

- argument type is contravariant
- output type is covariant

Use Liskov substitution principle.

- (also because of subsumption. (eg. Eiffel, Dart)
- Some languages have covariant argument subtyping, not type safe.

Type inference with subtyping

recall
Subsumption: $\dfrac{\Gamma \vdash e : s \quad \vdash s <: t}{\Gamma \vdash e : t}$

- No longer have unique typing

- think of typeinf as partial mapping maps term $\longrightarrow$ type.

  $\Rightarrow$ Now, a term can have multiple types. When a term has

- Subtyping is (typically) a partial order. (Subtyping forms a preorder relation, reflexive and transitive).

  ✱ Preorder not partial order ex:

  $u \leq v$ iff $\exists$ path from $u \to v$.

- Goal given context Γ and expr e,

  find least type t such that

  $\Gamma \vdash e : t$ is derivable.

Implementation

Subsumption isn't syntax-directed

i.e. we can't use program syntax to determine when to use subsumption rule.

* Don't use subsumption! Integrate it into other inference rules:

ex: Type_CArr

$$\frac{\Gamma \vdash e_1 : t \quad \ldots \quad \Gamma \vdash e_n : t}{\Gamma \vdash \text{new } t[]\{e_1; \ldots; e_n\} : t[]}$$

"weaken" these premises:

Instead of proving that $e_1 \ldots e_n : t[]$

we first prove that $e_1 : t \ \ldots \ e_n : t$. and "defer" subsumption into conclusion.

Interesting case: Conditionals.

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

Problem: What is t? (In type_(ter, new "t" is supplied.)

Condition: — t is least upper bound of t_2 and t_3

① $t_2 <: t \ \wedge \ t_3 <: t$

② For all t' s.t. $t_2 <: t' \wedge t_3 <: t'$, $t <: t'$.

if $<:$ is partial order then least upper bound is unique.