

Explicit Casts

The basis for this extension is the simply typed lambda calculus with support for subtyping and exceptions.

Syntax:

$$e ::= \dots \mid e \text{ as } T$$

Type rule:

$$\frac{\Gamma \vdash e : S}{\Gamma \vdash e \text{ as } T : T}$$

Evaluation contexts:

$$E ::= \dots \mid E \text{ as } T$$

Reduction rule:

$$\frac{\vdash v : T}{v \text{ as } T \longrightarrow v}$$

$$\frac{\not\vdash v : T}{v \text{ as } T \longrightarrow \text{raise "cast error"}}$$

Type Tests (alternative to using exceptions)

Syntax:

$$e ::= \dots \mid \text{if } e \text{ in } T \text{ then } x. e \text{ else } e$$

Type rule:

$$\frac{\Gamma \vdash e_1 : S \quad \Gamma, x : T_1 \vdash e_2 : T_2 \quad \Gamma \vdash e_3 : T_2}{\Gamma \vdash \text{if } e_1 \text{ in } T_1 \text{ then } x. e_2 \text{ else } e_3 : T_2}$$

Evaluation contexts:

$$E ::= \dots \mid \text{if } E \text{ in } T_1 \text{ then } x. e_2 \text{ else } e_3$$

Reduction rules:

$$\frac{\vdash v : T_1}{\text{if } v \text{ in } T_1 \text{ then } x. e_2 \text{ else } e_3 \longrightarrow [x \mapsto v]e_2}$$

$$\frac{\not\vdash v : T_1}{\text{if } v \text{ in } T_1 \text{ then } x. e_2 \text{ else } e_3 \longrightarrow e_3}$$

Coercion Semantics for Subtyping

- ▶ Suppose we want $int <: float$.
- ▶ With the semantics given for subtyping so far, this would mean integers would have to be represented in the same way as floats.
- ▶ For efficiency reasons, we probably want integers and floats to have different representations.
- ▶ To allow for this, we can associate run-time coercions with uses of the subsumption rule, for example, to convert an integer into a float.

Coercion Semantics for Subtyping

For each subtype derivation, we'll associate a function that coerces a value of the left-hand type to a value of the right-hand type. Given a derivation D , we write $\llbracket D \rrbracket$ for the coercion function.

$$\begin{array}{l} \left[\frac{}{Bool <: Bool} \right] \\ \left[\frac{}{Int <: Float} \right] \\ \left[\frac{D_1 :: T_1 <: S_1 \quad D_2 :: S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \right] \\ \left[\frac{\begin{array}{l} dom(R_2) \subseteq dom(R_1) \\ \forall l \in dom(R_2). R_1(l) = R_2(l) \end{array}}{R_1 <: R_2} \right] \end{array}$$

$$= \lambda x : Bool. x$$

$$= intToFloat$$

$$= \lambda f : S_1 \rightarrow S_2. \lambda x : T_1. \\ ([D_2] (f ([D_1] x)))$$

$$= \lambda r : R_1. \{l = r.l \mid l \in dom(R_2)\}$$

⋮

Coercion Semantics for Subtyping

Define a translation from the language with subtyping to the language with explicit coercions:

$$\boxed{\Gamma \vdash e \Rightarrow e : T}$$

The translation rule corresponding to subsumption inserts a coercion.

$$\frac{\Gamma \vdash e \Rightarrow e' : S \quad D :: S <: T}{\Gamma \vdash e \Rightarrow ([D] e') : T}$$

Implementing a Type Checker for Subtyping

- ▶ Recall the subsumption rule:

$$\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}$$

- ▶ This rule can be applied anywhere, anytime. So how do we know when to use it in the type checker?
- ▶ The problem with this rule is that it is not syntax directed.
- ▶ Can we get rid of the rule by using $S <: T$ in the other rules?

Syntax Directed Type System

- ▶ The basic idea is to sprinkle uses of $<:$ in all the places that you would use type equality.
- ▶ For example, in function application:

$$\frac{\Gamma \vdash e_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash e_2 : T_2 \quad T_2 <: T_{11}}{\Gamma \vdash (e_1 \ e_2) : T_{12}}$$

Algorithmic Subtyping

- ▶ There's also a problem with the definition of subtyping: we don't know when, or in what order, to apply the record width and depth rules.

$$\frac{\text{dom}(R_2) \subseteq \text{dom}(R_1) \quad \forall l \in \text{dom}(R_2). R_1(l) = R_2(l)}{R_1 <: R_2}$$

$$\frac{\text{dom}(R_1) = \text{dom}(R_2) \quad \text{for } l \in \text{dom}(R_1). R_1(l) <: R_2(l)}{R_1 <: R_2}$$

- ▶ These two rules can be replaced by the following rule:

$$\frac{\text{dom}(R_2) \subseteq \text{dom}(R_1) \quad \text{for } l \in \text{dom}(R_2). R_1(l) <: R_2(l)}{R_1 <: R_2}$$

Transitivity of Subtyping

Suppose for the moment we have defined subtyping by the following syntax-directed rules:

$$\frac{}{Bool <: Bool} \quad \frac{}{Int <: Int} \quad \frac{}{T <: Top}$$
$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

Can we prove that this subtype relation is transitive?

Transitivity of Subtyping

Proposition

If $R <: S$ and $S <: T$ then $R <: T$.

Proof. by induction on S .

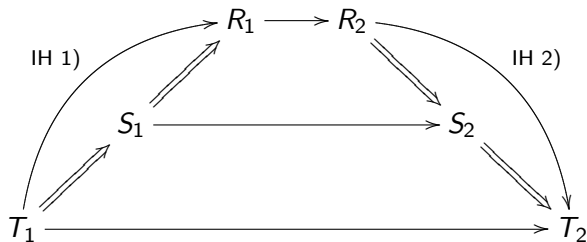
Case $S = Int$. Then $R = Int$ and $T = Int$ or $T = Top$. In either case, $R <: T$.

Case $S = Bool$. Similar.

Case $S = Top$. So $T = Top$ and then $R <: T$.

Transitivity of Subtyping, continued

Case $S = S_1 \rightarrow S_2$. Then $R = R_1 \rightarrow R_2$ with $S_1 <: R_1$ and $R_2 <: S_2$, and either (a) $T = T_1 \rightarrow T_2$ with $T_1 <: S_1$ and $S_2 <: T_2$ or else (b) $T = Top$. The following diagram shows the proof for case (a):



For case (b), because $T = Top$ we immediately have $R <: T$.

Handling Conditional Expressions

- ▶ What type should the following expression have?

if true then {x=true,y=false} **else** {x=false,z=true}

- ▶ Should it be $\{x : Top\}$, $\{x : Bool\}$, or just $\{\}$? In some sense, any type that is supertype of both branches will work. But which is best?
- ▶ We don't want to throw information away, so the best type is the **least upper bound** of the types of the two branches.
- ▶ I.e., if the two branches have type S and T , we want the type R such that $S <: R$ and $T <: R$, and for any other type R' , $R <: R'$.
- ▶ Does such a type always exist?