

Boxy types

Inference for higher-rank types and impredicativity

Dimitrios Vytiniotis¹ Stephanie Weirich¹
Simon Peyton Jones²

¹Computer and Information Science Department
University of Pennsylvania

²Microsoft Research

Presentation for ICFP 2006—Portland, Oregon

Trends in functional programming

... generalized algebraic datatypes [e.g. Weirich's talk in ICFP'06],
polymorphic recursion, higher-rank and impredicative polymorphism,
type-level lambdas, existential types, dependent types ...

Trends in functional programming

... generalized algebraic datatypes [e.g. Weirich's talk in ICFP'06], polymorphic recursion, higher-rank and impredicative polymorphism, type-level lambdas, existential types, dependent types ...

- Ideas originating in explicitly typed languages
- Full type inference hard: need mixture of checking and inference

Trends in functional programming

... generalized algebraic datatypes [e.g. Weirich's talk in ICFP'06], polymorphic recursion, higher-rank and impredicative polymorphism, type-level lambdas, existential types, dependent types ...

- Ideas originating in explicitly typed languages
- Full type inference hard: need mixture of checking and inference

Higher-rank types

A type of **higher-rank**: \forall -quantifiers nested to the left of arrows

Higher-rank types can be indispensable [Shan, 2004]:

```
-- generic map from SYB
gmapT :: forall a. Term a => (forall b. Term b => b -> b) -> a -> a

-- encapsulated state monad
runST :: forall a. (forall s. ST s a) -> a
```

Impredicative instantiation

Impredicative instantiation:

Quantified type variables can be instantiated with arbitrary types

Create and use data structures that hold polymorphic values:

```
head :: forall a. [a] -> a
ids  :: [forall c. c -> c]
f = head ids
```

head ids requires instantiation of *a* with $\forall c. c \rightarrow c$

Type inference

There is need for type inference—but not an easy problem

```
f g = (g 3, g True)
```

Which type do we infer for f ?

$\vdash f : (\forall a. a \rightarrow Int) \rightarrow (Int, Int)$

$\vdash f : (\forall a. a \rightarrow a) \rightarrow (Int, Bool)$

$\vdash f : \dots$ other?

- No “best” type for f . No **modularity**
- Worse: Inference for System-F **undecidable** [Wells, 1994]

Modest solutions for higher-rank types

Putting type annotations to work:

- 1 **Annotated** functions take polymorphic args [Odersky and Läufer, 1996]

```
f (g::(forall a.a->a)) = (g 3, g True)
```

- 2 Annotations **not exactly on** the functions; bidirectional propagation [Peyton Jones *et al.*, 2003, 2005][Rémy, 2005]

```
f :: (forall a.a->Int)->(Int,Int)
f = \g -> (g 3, g True)
```

... but no impredicative instantiation

HM + higher-rank types + impredicativity \approx programming in System-F

Boxy types

- ① Hindley-Milner style type inference
 - ▶ **plus** impredicative and higher-rank polymorphism!
- ② Follow “putting type annotations to work” ideas

Design principles

- 1 Only include System-F types; unlike MLF [Le Botlan and Rémy, 2003]
- 2 Simple extension to existing type inference algorithms
 - ▶ Never guess polymorphism; propagate type annotations instead
 - ▶ Single-pass; unlike Rémy's stratified type inference [Rémy, 2005]
- 3 Reach all of System-F via addition of appropriate annotations
- 4 Straightforward semantics-preserving compilation to System-F

Boxy Types

Take System-F types and draw **boxes** around them, e.g.

$$\boxed{\forall a. a \rightarrow a} \rightarrow Int \rightarrow \boxed{Bool}$$

$$\boxed{\forall a. a \rightarrow a \rightarrow a}$$

Judgements of the form:

$$\langle \text{environment} \rangle \vdash \langle \text{expression} \rangle : \langle \text{boxy-type} \rangle$$

- Boxes represent **uncertainty** during inference
 - ▶ Output parameters to the type checker
- Box-free parts represent **known information**
 - ▶ Input parameters to the type checker
 - ▶ Known info originating in type annotations

Boxy types for information propagation

No special semantics for boxes

- Only there to restrict **which typing derivations are valid**

$$\vdash \lambda g.(g\ 3, g\ True) : \underbrace{(\forall a.a \rightarrow a) \rightarrow (Int, Bool)}_{\text{known info}}$$

Boxy types for information propagation

No special semantics for boxes

- Only there to restrict **which typing derivations are valid**

$$\vdash \lambda g.(g\ 3, g\ \text{True}) : \underbrace{(\forall a.a \rightarrow a) \rightarrow (Int, Bool)}_{\text{known info}}$$

$$\vdash \lambda g.(g\ 3, g\ \text{True}) : \underbrace{(\forall a.a \rightarrow a)}_{\text{known info}} \rightarrow \boxed{(Int, Bool)}$$

Boxy types for information propagation

No special semantics for boxes

- Only there to restrict **which typing derivations are valid**

$$\vdash \lambda g.(g\ 3, g\ True) : \underbrace{(\forall a.a \rightarrow a) \rightarrow (Int, Bool)}_{\text{known info}}$$

$$\vdash \lambda g.(g\ 3, g\ True) : \underbrace{(\forall a.a \rightarrow a)}_{\text{known info}} \rightarrow \boxed{(Int, Bool)}$$

$$\not\vdash \lambda g.(g\ 3, g\ True) : \underbrace{\boxed{\forall a.a \rightarrow a} \rightarrow \boxed{(Int, Bool)}}_{\text{both boxes inferred!}} \Leftarrow \text{Can't do that!}$$

Boxy types for information propagation

No special semantics for boxes

- Only there to restrict **which typing derivations are valid**

$$\vdash \lambda g.(g\ 3, g\ True) : \underbrace{(\forall a.a \rightarrow a)}_{\text{known info}} \rightarrow (Int, Bool)$$

$$\vdash \lambda g.(g\ 3, g\ True) : \underbrace{(\forall a.a \rightarrow a)}_{\text{known info}} \rightarrow \boxed{(Int, Bool)}$$

$$\not\vdash \lambda g.(g\ 3, g\ True) : \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{(Int, Bool)} \quad \Leftarrow \text{Can't do that!}$$

both boxes inferred!

$$\vdash \lambda \underbrace{(g :: \forall a.a \rightarrow a)}_{\text{provides info}}.(g\ 3, g\ True) : \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{(Int, Bool)}$$

Filling in boxes: Never guess polymorphism

- Boxes filled in with info available **locally**

$\vdash \lambda(g :: \forall a.a \rightarrow a).(g\ 3, g\ \text{True}) : \boxed{\forall a.a \rightarrow a} \rightarrow (\text{Int}, \text{Bool})$

Known info not only annotations, but also box-free type parts

$\vdash \lambda x.x : [\forall a.a \rightarrow a] \rightarrow \underbrace{\boxed{[\forall a.a \rightarrow a]}}_{\text{filled by known info}}$

Filling in boxes: Never guess polymorphism

- Boxes filled in with info available **locally**

$$\vdash \lambda(g :: \forall a.a \rightarrow a).(g \ 3, g \ \text{True}) : \boxed{\forall a.a \rightarrow a} \rightarrow (\text{Int}, \text{Bool})$$

Known info not only annotations, but also box-free type parts

$$\vdash \lambda x.x : [\forall a.a \rightarrow a] \rightarrow \underbrace{\boxed{[\forall a.a \rightarrow a]}}_{\text{filled by known info}}$$

- No info available locally: contents **monomorphic**

$$\not\vdash \lambda x.x : \boxed{(\forall a.a \rightarrow a)} \rightarrow \boxed{(\forall a.a \rightarrow a)}$$

Instead:

$$\vdash \lambda x.x : \boxed{\tau} \rightarrow \boxed{\tau} \quad (\text{for any monomorphic } \tau)$$

Impredicative instantiation

Working example $\lambda g.(g\ 3, g\ True)$ requires a higher-rank type

Q: What about **impredicative instantiation**?

A: Same principle! Never “guess” polymorphism!

Impredicative instantiation

Example:

$$f:\forall a.a \rightarrow a \in \Gamma$$

Instantiate a to derive:

$$\Gamma \vdash f : \boxed{Int} \rightarrow \boxed{Int}$$

But **never guess arbitrary polymorphic instantiations**:

$$\Gamma \not\vdash f : \underbrace{\boxed{(\forall a.a \rightarrow a)} \rightarrow \boxed{(\forall a.a \rightarrow a)}}_{\text{no known info to determine instantiation}}$$

Instead, **check arbitrary polymorphic instantiations**:

$$\Gamma \vdash f : \underbrace{(\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)}_{\text{known info determines instantiation!}}$$

Boxes, formally

$$\begin{array}{l|l} \sigma ::= \forall \bar{a}. \rho & \sigma' ::= \forall \bar{a}. \rho' \mid \boxed{\sigma} \\ \rho ::= \sigma \rightarrow \sigma \mid \tau & \rho' ::= \sigma' \rightarrow \sigma' \mid \boxed{\rho} \mid \tau \\ \tau ::= a \mid \tau \rightarrow \tau & \end{array}$$

- Constraints:

- ▶ No nested boxes: $\text{guess}^2 \equiv \text{guess}$
- ▶ No boxes in environments: search-free, single-pass, more in paper
- ▶ No quantified variables free inside boxes: what would that mean?

Call τ **monotypes**, rest **polytypes**

Typing derivations

The high-level specification of an inference algorithm:

- 1 Rules directed by syntax of terms **and** types
- 2 Close to Haskell syntax-directed rules:
 - ▶ Instantiation occurs at variables
 - ▶ Generalization at (unannotated) `let` expressions
- 3 Intuition: When not sure, use a box!

$$\frac{\Gamma \vdash t : \boxed{\sigma} \rightarrow \rho' \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t u : \rho'} \text{ APP}$$

- ▶ Check t using known info from ρ' and inferring $\boxed{\sigma}$ for argument
- ▶ Check u against σ

- Type annotations result in fully-checkable types

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, x:\sigma \vdash t : \rho'}{\text{let } x :: \sigma = u \text{ in } t : \rho'} \text{SIGLET}$$

More typing

- Type annotations result in fully-checkable types

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, x:\sigma \vdash t : \rho'}{\text{let } x :: \sigma = u \text{ in } t : \rho'} \text{SIGLET}$$

- No information; fully inferred

$$\frac{\Gamma \vdash u : \boxed{\rho} \quad \bar{a} = \text{ftv}(\rho) - \text{ftv}(\Gamma) \quad \Gamma, x:\forall \bar{a}. \rho \vdash t : \rho'}{\Gamma \vdash \text{let } x = u \text{ in } t : \rho'} \text{LET}$$

Shape of types drives typing

$$\frac{\Gamma \vdash (\lambda x.t) : \boxed{\sigma_1} \rightarrow \boxed{\sigma_2}}{\Gamma \vdash (\lambda x.t) : \boxed{\sigma_1 \rightarrow \sigma_2}} \text{ ABS1}$$

$$\frac{\Gamma \vdash t : \rho' \quad \bar{a} \cap \text{ftv}(\Gamma) = \emptyset}{\Gamma \vdash t : \forall \bar{a}. \rho'} \text{ SKOL}$$

- Types $\boxed{\sigma_1} \rightarrow \boxed{\sigma_2}$ and $\boxed{\sigma_1 \rightarrow \sigma_2}$ are effectively interchangeable
- SKOL “skolemizes” quantified variables

Instantiation

As in Haskell:

$$\frac{\vdash \sigma \leq \rho' \quad x:\sigma \in \Gamma}{\Gamma \vdash x : \rho'} \text{VAR}$$

(\leq): **subsumption** relation. Read σ is “more polymorphic than” ρ'

Let $\sigma_{id} = \forall a. a \rightarrow a$. When $f:\sigma_{id} \in \Gamma$ we wish:

- $\Gamma \vdash f : \boxed{Int} \rightarrow \boxed{Int}$ therefore $\vdash \forall a. a \rightarrow a \leq \boxed{Int} \rightarrow \boxed{Int}$
- $\Gamma \not\vdash f : \boxed{\sigma_{id}} \rightarrow \boxed{\sigma_{id}}$ therefore $\not\vdash \forall a. a \rightarrow a \leq \boxed{\sigma_{id}} \rightarrow \boxed{\sigma_{id}}$
- $\Gamma \vdash f : \sigma_{id} \rightarrow \sigma_{id}$ therefore $\vdash \forall a. a \rightarrow a \leq \sigma_{id} \rightarrow \sigma_{id}$

Operation of subsumption

- Instantiation of left-hand side variables with **boxes**

$$\frac{\vdash [\overline{a \mapsto \sigma}] \rho'_1 \leq \rho'_2}{\vdash \forall \overline{a}. \rho'_1 \leq \rho'_2} \text{SPEC}$$

- Filling-in of boxes from both sides
 - ▶ Box-meets-box situation: Force to monotypes!

Operation of subsumption, continued

$$\not\vdash \forall a. a \rightarrow a \leq \boxed{\sigma_{id}} \rightarrow \boxed{\sigma_{id}}$$

$$\leftrightarrow \vdash \boxed{?} \rightarrow \boxed{?} \leq \boxed{\sigma_{id}} \rightarrow \boxed{\sigma_{id}}$$

$$\leftrightarrow \vdash \underbrace{\boxed{?} \leq \boxed{\sigma_{id}}}_{\text{result types}} \wedge \underbrace{\vdash \boxed{?} \sim \boxed{\sigma_{id}}}_{\text{invariant argument types}}$$

$$\leftrightarrow \text{Fail: } \sigma_{id} \text{ not monotype!}$$

Theorem: $\not\vdash \boxed{\forall a. \sigma} \leq \boxed{\forall a. \sigma}$ and $\vdash \boxed{\tau} \leq \boxed{\tau}$

(\implies hence also $\vdash \forall a. a \rightarrow a \leq \boxed{Int} \rightarrow \boxed{Int}$)

Operation of subsumption, continued

On the other hand:

$$\vdash \forall a. a \rightarrow a \leq \sigma_{id} \rightarrow \sigma_{id}$$

$$\leftrightarrow \vdash \boxed{?} \rightarrow \boxed{?} \leq \sigma_{id} \rightarrow \sigma_{id}$$

$$\leftrightarrow \underbrace{\vdash \boxed{?} \leq \sigma_{id}}_{\text{result types}} \wedge \underbrace{\vdash \boxed{?} \sim \sigma_{id}}_{\text{argument types}}$$

$$\leftrightarrow ? = \sigma_{id}$$

Theorem: $\vdash \forall a. \sigma \leq [a \mapsto \sigma_a] \sigma$

- One more abstraction rule:

$$\frac{\vdash \sigma'_1 \sim \boxed{\sigma_1} \quad \Gamma, x:\sigma_1 \vdash t : \sigma'_2}{\Gamma \vdash (\lambda x.t) : \sigma'_1 \rightarrow \sigma'_2} \text{ ABS2}$$

- $\vdash \sigma'_1 \sim \boxed{\sigma_1}$ effectively forces boxes of σ'_1 to monotypes

Technical properties

- 1 Type-safety via **straightforward** translation to System-F
- 2 Inference algorithm can be read-off from typing rules
- 3 “Principal” types for a given amount of “checked” information
- 4 Type system conservatively extends Hindley-Milner
- 5 All of System-F typeable with addition of annotations
 - ▶ placed on **polymorphic** type instantiation sites
 - ▶ but many heuristics to improve this, see paper

- ① Boxy-style propagation for dependent and higher-order types
- ② Practical evaluation
 - ▶ Implementation already released in GHC
- ③ Bridge gap between MLF and Boxy Types
 - ▶ Delaying of instantiation decisions, logical specification, etc.
- ④ Increase robustness under program transformations

Thank you for your attention!

Boxes are **GOOD FUN!**

Thank you for your attention!

Boxes are **GOOD FUN!**

Please read the paper for details:

www.cis.upenn.edu/~dimitriv/boxy

MLF [Le Botlan and Rémy,2003]:

- **Different type structure** than that of System-F
- Regain principal types and decidable type inference
- Keep expressions generalized, make use of type annotations for FCP
- Instantiation constraints on quantified type variables
- Trick: Constraints permit delaying of instantiation decisions
- Evidence translation to System-F [Leijen and Löh] but not proven computationally light

Subsumption, instance rules

$$\frac{\vdash \boxed{\sigma} \sim \sigma'}{\vdash \boxed{\sigma} \leq \sigma'} \text{ SBOXY} \qquad \frac{\vdash \forall \bar{a}. \rho'_1 \leq \rho'_2 \quad \bar{b} \notin \text{ftv}(\forall \bar{a}. \rho'_1)}{\vdash \forall \bar{a}^+. \rho'_1 \leq \forall \bar{b}. \rho'_2} \text{ SKOL}$$

$$\frac{\vdash [a \mapsto \boxed{\sigma}] \rho'_1 \leq \rho'_2}{\vdash \forall \bar{a}. \rho'_1 \leq \rho'_2} \text{ SPEC}$$

Subumption, congruence rules

$$\frac{\vdash \sigma'_1 \rightarrow \sigma'_2 \leq \boxed{\sigma_3} \rightarrow \boxed{\sigma_4}}{\vdash \sigma'_1 \rightarrow \sigma'_2 \leq \boxed{\sigma_3 \rightarrow \sigma_4}} \text{ F1}$$

$$\frac{\vdash \sigma'_3 \sim \sigma'_1 \quad \vdash \sigma'_2 \leq \sigma'_4}{\vdash \sigma'_1 \rightarrow \sigma'_2 \leq \sigma'_3 \rightarrow \sigma'_4} \text{ F2}$$

$$\frac{}{\vdash \tau \leq \tau} \text{ MONO}$$

- Notice function argument **invariance**. Completeness reasons.

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

- **a box meets a sans-box type** \implies fill in the box!

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

- **a box meets a sans-box type** \implies fill in the box!
- **a box meets a box** \implies force both to be monotypes!

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

- **a box meets a sans-box type** \implies fill in the box!
- **a box meets a box** \implies force both to be monotypes!

Examples:

$$\vdash \text{Int} \rightarrow \boxed{\text{Int}} \sim \boxed{\text{Int} \rightarrow \text{Int}}$$

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

- **a box meets a sans-box type** \implies fill in the box!
- **a box meets a box** \implies force both to be monotypes!

Examples:

$$\begin{aligned} &\vdash \text{Int} \rightarrow \boxed{\text{Int}} \sim \boxed{\text{Int} \rightarrow \text{Int}} \\ \not\vdash \text{Int} \rightarrow \boxed{\forall a. a \rightarrow a} &\sim \boxed{\text{Int} \rightarrow \forall a. a \rightarrow a} \end{aligned}$$

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

- **a box meets a sans-box type** \implies fill in the box!
- **a box meets a box** \implies force both to be monotypes!

Examples:

$$\begin{aligned} &\vdash \text{Int} \rightarrow \boxed{\text{Int}} \sim \boxed{\text{Int} \rightarrow \text{Int}} \\ \not\vdash \text{Int} \rightarrow \boxed{\forall a. a \rightarrow a} &\sim \boxed{\text{Int} \rightarrow \forall a. a \rightarrow a} \\ \vdash \text{Int} \rightarrow \forall a. a \rightarrow a &\sim \boxed{\text{Int} \rightarrow \forall a. a \rightarrow a} \end{aligned}$$

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

- a **box** meets a **sans-box type** \implies fill in the box!
- a **box** meets a **box** \implies force both to be monotypes!

Examples:

$$\begin{aligned} & \vdash \text{Int} \rightarrow \boxed{\text{Int}} \sim \boxed{\text{Int} \rightarrow \text{Int}} \\ \not\vdash \text{Int} \rightarrow \boxed{\forall a. a \rightarrow a} & \sim \boxed{\text{Int} \rightarrow \forall a. a \rightarrow a} \\ \vdash \text{Int} \rightarrow \forall a. a \rightarrow a & \sim \boxed{\text{Int} \rightarrow \forall a. a \rightarrow a} \\ & \vdash \boxed{a} \sim \boxed{a} \end{aligned}$$

Boxy matching (a.k.a. super-unification)

$$\vdash \sigma'_1 \sim \sigma'_2$$

Intuition: Walk down the type structure and when:

- a **box** meets a **sans-box type** \implies fill in the box!
- a **box** meets a **box** \implies force both to be monotypes!

Examples:

$$\begin{aligned} & \vdash \text{Int} \rightarrow \boxed{\text{Int}} \sim \boxed{\text{Int} \rightarrow \text{Int}} \\ \not\vdash \text{Int} \rightarrow \boxed{\forall a. a \rightarrow a} & \sim \boxed{\text{Int} \rightarrow \forall a. a \rightarrow a} \\ \vdash \text{Int} \rightarrow \forall a. a \rightarrow a & \sim \boxed{\text{Int} \rightarrow \forall a. a \rightarrow a} \\ & \vdash \boxed{a} \sim \boxed{a} \\ \not\vdash \boxed{\forall a. a \rightarrow a} & \sim \boxed{\forall a. a \rightarrow a} \end{aligned}$$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a.a \rightarrow a \leq \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{\forall a.a \rightarrow a}$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a.a \rightarrow a \leq \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{\forall a.a \rightarrow a}$
- $\vdash \forall a.a \rightarrow a \leq (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a.a \rightarrow a \leq \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{\forall a.a \rightarrow a}$
- $\vdash \forall a.a \rightarrow a \leq (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)$
- $\vdash \forall a.a \rightarrow a \leq \boxed{b} \rightarrow \boxed{b}$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a.a \rightarrow a \leq \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{\forall a.a \rightarrow a}$
- $\vdash \forall a.a \rightarrow a \leq (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)$
- $\vdash \forall a.a \rightarrow a \leq \boxed{b} \rightarrow \boxed{b}$
- $\not\vdash \boxed{\forall a.a \rightarrow a} \leq \boxed{\forall a.a \rightarrow a}$

Subsumption examples

- $\vdash (\forall ab. a \rightarrow b) \rightarrow (\forall a. a \rightarrow a) \leq (\forall ab. a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a. a \rightarrow a \leq \forall a. Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a. a \rightarrow a \leq \boxed{\forall a. a \rightarrow a} \rightarrow \boxed{\forall a. a \rightarrow a}$
- $\vdash \forall a. a \rightarrow a \leq (\forall a. a \rightarrow a) \rightarrow (\forall a. a \rightarrow a)$
- $\vdash \forall a. a \rightarrow a \leq \boxed{b} \rightarrow \boxed{b}$
- $\not\vdash \boxed{\forall a. a \rightarrow a} \leq \boxed{\forall a. a \rightarrow a}$
- $\vdash \boxed{\top} \leq \boxed{\top}$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a.a \rightarrow a \leq \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{\forall a.a \rightarrow a}$
- $\vdash \forall a.a \rightarrow a \leq (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)$
- $\vdash \forall a.a \rightarrow a \leq \boxed{b} \rightarrow \boxed{b}$
- $\not\vdash \boxed{\forall a.a \rightarrow a} \leq \boxed{\forall a.a \rightarrow a}$
- $\vdash \boxed{\top} \leq \boxed{\top}$
- $\vdash \forall a.a \rightarrow a \leq \boxed{Int \rightarrow Int}$

Subsumption examples

- $\vdash (\forall ab.a \rightarrow b) \rightarrow (\forall a.a \rightarrow a) \leq (\forall ab.a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a.a \rightarrow a \leq \forall a.Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a.a \rightarrow a \leq \boxed{\forall a.a \rightarrow a} \rightarrow \boxed{\forall a.a \rightarrow a}$
- $\vdash \forall a.a \rightarrow a \leq (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)$
- $\vdash \forall a.a \rightarrow a \leq \boxed{b} \rightarrow \boxed{b}$
- $\not\vdash \boxed{\forall a.a \rightarrow a} \leq \boxed{\forall a.a \rightarrow a}$
- $\vdash \boxed{\top} \leq \boxed{\top}$
- $\vdash \forall a.a \rightarrow a \leq \boxed{Int \rightarrow Int}$
- $\vdash \boxed{(\forall a.a \rightarrow a) \rightarrow \forall ab.a \rightarrow b} \leq (\forall a.a \rightarrow a) \rightarrow \forall ab.a \rightarrow b$

Subsumption examples

- $\vdash (\forall ab. a \rightarrow b) \rightarrow (\forall a. a \rightarrow a) \leq (\forall ab. a \rightarrow b) \rightarrow (Int \rightarrow Int)$
- $\vdash Int \rightarrow \forall a. a \rightarrow a \leq \forall a. Int \rightarrow a \rightarrow a$
- $\not\vdash \forall a. a \rightarrow a \leq \boxed{\forall a. a \rightarrow a} \rightarrow \boxed{\forall a. a \rightarrow a}$
- $\vdash \forall a. a \rightarrow a \leq (\forall a. a \rightarrow a) \rightarrow (\forall a. a \rightarrow a)$
- $\vdash \forall a. a \rightarrow a \leq \boxed{b} \rightarrow \boxed{b}$
- $\not\vdash \boxed{\forall a. a \rightarrow a} \leq \boxed{\forall a. a \rightarrow a}$
- $\vdash \boxed{\top} \leq \boxed{\top}$
- $\vdash \forall a. a \rightarrow a \leq \boxed{Int \rightarrow Int}$
- $\vdash \boxed{(\forall a. a \rightarrow a) \rightarrow \forall ab. a \rightarrow b} \leq (\forall a. a \rightarrow a) \rightarrow \forall ab. a \rightarrow b$
- $\vdash (\forall ab. a \rightarrow b) \rightarrow \forall a. a \rightarrow a \leq \boxed{\forall ab. a \rightarrow b} \rightarrow (Int \rightarrow Int)$

Program example

```
let  f x = x
     z :: [∀a.a → a] = [(f :: (∀a.a → a) → (∀a.a → a)) (λx.x)]
in   ((f :: [∀a.a → a] → [∀a.a → a]) z)
```

The safe game:

- Annotations on polymorphic instantiations will always work
- But raises questions about flow of information (look at z's type)

Note: Invariant list constructor

Program example

Infer $\boxed{a \rightarrow a}$ then unbox and generalize $f:\forall a.a \rightarrow a$

```
let  f x = x
     z :: [ $\forall a.a \rightarrow a$ ] = [(f :: ( $\forall a.a \rightarrow a$ )  $\rightarrow$  ( $\forall a.a \rightarrow a$ )) ( $\lambda x.x$ )]
in   ((f :: [ $\forall a.a \rightarrow a$ ]  $\rightarrow$  [ $\forall a.a \rightarrow a$ ]) z)
```

The safe game:

- Annotations on polymorphic instantiations will always work
- But raises questions about flow of information (look at z's type)

Note: Invariant list constructor

Program example

- 1 $\vdash (f :: (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)) (\lambda x.x) : \forall a.a \rightarrow a$
- 2 $\vdash (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a) \leq \boxed{\forall a.a \rightarrow a} \rightarrow (a \rightarrow a)$
- 3 $\vdash \forall a.a \rightarrow a \leq (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)$
- 4 $\vdash \lambda x.x : \forall a.a \rightarrow a$

```
let  f x = x
     z :: [\forall a.a \rightarrow a] = [(f :: (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)) (\lambda x.x)]
in   ((f :: [\forall a.a \rightarrow a] \rightarrow [\forall a.a \rightarrow a]) z)
```

The safe game:

- Annotations on polymorphic instantiations will always work
- But raises questions about flow of information (look at z's type)

Note: Invariant list constructor

Program example

Check

$$\vdash \forall a. a \rightarrow a \leq [\forall a. a \rightarrow a] \rightarrow [\forall a. a \rightarrow a]$$

and infer type $[\forall a. a \rightarrow a]$

```
let  f x = x
     z :: [\forall a. a \to a] = [(f :: (\forall a. a \to a) \to (\forall a. a \to a)) (\lambda x. x)]
in   ((f :: [\forall a. a \to a] \to [\forall a. a \to a]) z)
```

The safe game:

- Annotations on polymorphic instantiations will always work
- But raises questions about flow of information (look at z's type)

Note: Invariant list constructor

Flow of information in application nodes

Consider again the application rule

$$\frac{\Gamma \vdash t : \boxed{\sigma} \rightarrow \rho' \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t u : \rho'} \text{ APP}$$

Information flows from function to argument.

Flow of information in application nodes

Consider again the application rule

$$\frac{\Gamma \vdash t : \boxed{\sigma} \rightarrow \rho' \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t u : \rho'} \text{ APP}$$

Information flows from function to argument.

But one could imagine a different rule:

$$\frac{\Gamma \vdash u : \boxed{\sigma} \quad \Gamma \vdash t : \sigma \rightarrow \rho'}{\Gamma \vdash t u : \rho'} \text{ APP}'$$

This would type expressions like the *head ids* example of the introduction.

... or even go one step further: Smart-application.

$$\frac{\begin{array}{l} x : \forall \bar{a} \bar{b}. \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \sigma_r \in \Gamma \\ \bar{a} \in \text{ftv}(\sigma_1, \dots, \sigma_n) \quad \bar{b} \notin \text{ftv}(\sigma_1, \dots, \sigma_n) \\ \Gamma \vdash u_i : [\bar{a} \mapsto \boxed{\sigma_a}] \sigma_i \quad \vdash [\bar{a} \mapsto \sigma_a, \bar{b} \mapsto \boxed{\sigma_b}] \sigma_r \leq \rho' \end{array}}{\Gamma \vdash x u_1 \dots u_n : \rho'}$$

... or even go one step further: Smart-application.

$$\frac{\begin{array}{l} x : \forall \bar{a} \bar{b}. \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \sigma_r \in \Gamma \\ \bar{a} \in \text{ftv}(\sigma_1, \dots, \sigma_n) \quad \bar{b} \notin \text{ftv}(\sigma_1, \dots, \sigma_n) \\ \Gamma \vdash u_i : [\bar{a} \mapsto \boxed{\sigma_a}] \sigma_i \quad \vdash [\bar{a} \mapsto \sigma_a, \bar{b} \mapsto \boxed{\sigma_b}] \sigma_r \leq \rho' \end{array}}{\Gamma \vdash x u_1 \dots u_n : \rho'}$$

(But such rules threaten completeness!)

Other heuristics

Modification of subsumption (rule SPEC) and smart-application to two phases:

- 1 First, the invariant parts of the types are matched and a substitution is created for the quantified variables.
- 2 Next, that substitution is used for normal subsumption.

For example

$$\not\vdash \forall a. a \rightarrow a \leq (\forall a. a \rightarrow a) \rightarrow Int \rightarrow Int$$

With the heuristic however we would:

- 1 First, create a substitution $[a \mapsto (\forall a. a \rightarrow a)]$
- 2 Next, use the substitution to check that

$$\vdash (\forall a. a \rightarrow a) \rightarrow (\forall a. a \rightarrow a) \leq (\forall a. a \rightarrow a) \rightarrow Int \rightarrow Int$$

Other heuristics

Modification of subsumption (rule SPEC) and smart-application to two phases:

- 1 First, the invariant parts of the types are matched and a substitution is created for the quantified variables.
- 2 Next, that substitution is used for normal subsumption.

For example

$$\not\vdash \forall a. a \rightarrow a \leq (\forall a. a \rightarrow a) \rightarrow Int \rightarrow Int$$

With the heuristic however we would:

- 1 First, create a substitution $[a \mapsto (\forall a. a \rightarrow a)]$
- 2 Next, use the substitution to check that

$$\vdash (\forall a. a \rightarrow a) \rightarrow (\forall a. a \rightarrow a) \leq (\forall a. a \rightarrow a) \rightarrow Int \rightarrow Int$$

(Heuristics: useful but not robust)

Examples

```
let  f x = x
     z :: [∀a.a → a] = [f (λx.x)]
in   (f z)
```

Let $\vdash \text{choose} : \forall a.a \rightarrow a \rightarrow a$ and $\vdash \text{id} : \forall a.a \rightarrow a$. Then

```
let auto           = λ(x :: (∀a.a → a)).x x
let f              = choose id
let f1            = choose id succ
let f2            = choose id auto
let f3            = choose (id :: ∀a.(∀a.a → a) → (a → a)) auto
let f4 :: σid → σid = choose id auto
```

Examples

```
let  f x = x
     z :: [∀a.a → a] = [f (λx.x)]
in   (f z)
```

Let $\vdash \text{choose} : \forall a.a \rightarrow a \rightarrow a$ and $\vdash \text{id} : \forall a.a \rightarrow a$. Then

```
let auto           = λ(x :: (∀a.a → a)).x x
let f              = choose id
let f1            = choose id succ
let f2            = choose id auto
let f3            = choose (id :: ∀a.(∀a.a → a) → (a → a)) auto
let f4 :: σid → σid = choose id auto
```

$\forall a.(\forall a.a \rightarrow a) \rightarrow (a \rightarrow a)$

Examples

```
let  f x = x
     z :: [∀a.a → a] = [f (λx.x)]
in   (f z)
```

Let $\vdash \text{choose} : \forall a.a \rightarrow a \rightarrow a$ and $\vdash \text{id} : \forall a.a \rightarrow a$. Then

```
let auto           = λ(x :: (∀a.a → a)).x x
let f              = choose id
let f1            = choose id succ
let f2            = choose id auto
let f3            = choose (id :: ∀a.(∀a.a → a) → (a → a)) auto
let f4 :: σid → σid = choose id auto
```

$\forall a.(a \rightarrow a) \rightarrow (a \rightarrow a)$

Examples

```
let  f x = x
     z :: [∀a.a → a] = [f (λx.x)]
in   (f z)
```

Let $\vdash \text{choose} : \forall a.a \rightarrow a \rightarrow a$ and $\vdash \text{id} : \forall a.a \rightarrow a$. Then

```
let auto           = λ(x :: (∀a.a → a)).x x
let f              = choose id
let f1            = choose id succ
let f2            = choose id auto
let f3            = choose (id :: ∀a.(∀a.a → a) → (a → a)) auto
let f4 :: σid → σid = choose id auto
```

$\text{Int} \rightarrow \text{Int}$

Examples

```
let  f x = x
     z :: [∀a.a → a] = [f (λx.x)]
in   (f z)
```

Let $\vdash \text{choose} : \forall a. a \rightarrow a \rightarrow a$ and $\vdash \text{id} : \forall a. a \rightarrow a$. Then

```
let auto           = λ(x :: (∀a.a → a)).x x
let f              = choose id
let f1            = choose id succ
let f2            = choose id auto
let f3            = choose (id :: ∀a.(∀a.a → a) → (a → a)) auto
let f4 :: σid → σid = choose id auto
```

fail!

Examples

```
let  f x = x
     z :: [∀a.a → a] = [f (λx.x)]
in   (f z)
```

Let $\vdash \text{choose} : \forall a.a \rightarrow a \rightarrow a$ and $\vdash \text{id} : \forall a.a \rightarrow a$. Then

```
let auto           = λ(x :: (∀a.a → a)).x x
let f              = choose id
let f1             = choose id succ
let f2             = choose id auto
let f3             = choose (id :: ∀a.(∀a.a → a) → (a → a)) auto
let f4 :: σid → σid = choose id auto
```

$\forall a.(\forall a.a \rightarrow a) \rightarrow (a \rightarrow a)$

Examples

```
let  f x = x
     z :: [∀a.a → a] = [f (λx.x)]
in   (f z)
```

Let $\vdash \text{choose} : \forall a.a \rightarrow a \rightarrow a$ and $\vdash \text{id} : \forall a.a \rightarrow a$. Then

```
let auto           = λ(x :: (∀a.a → a)).x x
let f              = choose id
let f1            = choose id succ
let f2            = choose id auto
let f3            = choose (id :: ∀a.(∀a.a → a) → (a → a)) auto
let f4 :: σid → σid = choose id auto
```

works with last heuristic method!