

- 1 Design
  - $iML^F$ : an implicit-typed extension of System F
  - Types explained
  - $eML^F$ : an explicitly-typed version of  $iML^F$
- 2 Uses and Implementation
  - Examples
  - Type inference
  - Restrictions and extensions

# ML<sup>F</sup> for Everyone

(Users, Implementers, and Designers)

Didier Rémy

INRIA-Rocquencourt

ML Workshop

(Based on joint work with **Didier Le Botlan** and **Boris Yakobowski**)

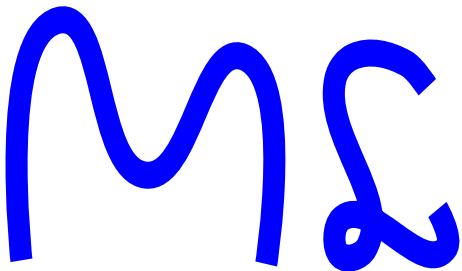
ms

Simple to use

Expressive

Great success

Happy days



M L

F

MS

MS

F

Expressive

~~Simple extension~~ of ML  
Simplification

Even used in full scale languages  
such as Scala.

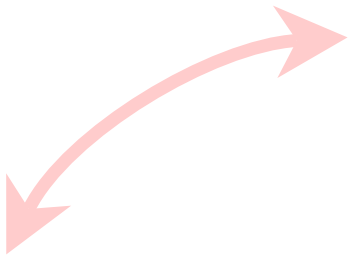
Full type inference  
is undecidable

Full type annotations  
are obfuscating

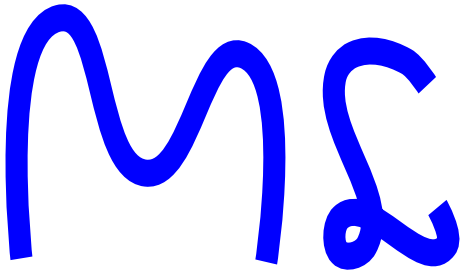
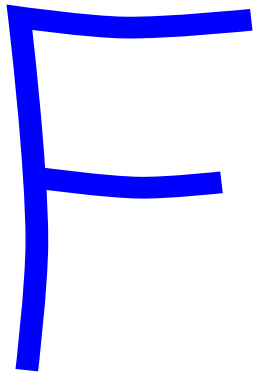
ML

F

ms



F



$F_{M_2}$

$F_{M_2}$

$\frac{L}{M_2}$

$F_{3W}$

$M_2^F$

$3W$

msf

# Outline

## 1 Design

- $iML^F$ : an implicit-typed extension of System F
- Types explained
- $eML^F$ : an explicitly-typed version of  $iML^F$

## 2 Uses and Implementation

- Examples
- Type inference
- Restrictions and extensions

# A universal type system

## Explicit System F:

$$\text{VAR} \quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau}$$

$$\text{APP} \quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 a_2 : \tau_1}$$

$$\text{FUN} \quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x : \tau_0) a : \tau_0 \rightarrow \tau}$$

$$\text{GEN} \quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash \Lambda \alpha. a : \forall(\alpha) \tau_0}$$

$$\text{UNGEN} \quad \frac{\Gamma \vdash a : \forall(\alpha) \tau}{\Gamma \vdash a \tau : \tau_0[\alpha \leftarrow \tau]}$$

# A universal type system

## Implicit System F:

$$\text{VAR} \quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau}$$

$$\text{APP} \quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 a_2 : \tau_1}$$

$$\text{FUN} \quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x \quad ) a : \tau_0 \rightarrow \tau}$$

$$\text{GEN} \quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall(\alpha) \tau_0}$$

$$\text{UNGEN} \quad \frac{\Gamma \vdash a : \forall(\alpha) \tau}{\Gamma \vdash a : \tau_0[\alpha \leftarrow \tau]}$$

# A universal type system

## Implicit System F:

$$\text{VAR} \quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau}$$

$$\text{APP} \quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 a_2 : \tau_1}$$

$$\text{FUN} \quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x \quad ) a : \tau_0 \rightarrow \tau}$$

$$\text{GEN} \quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall(\alpha) \tau_0}$$

$$\text{INST} \quad \frac{}{\forall(\bar{\alpha}) \tau_0 \leq \tau_0[\bar{\alpha} \leftarrow \bar{\tau}]}$$

$$\text{SUB} \quad \frac{\Gamma \vdash a : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash a : \tau_2}$$

# A universal type system

## Implicit System F:

$$\text{VAR} \quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau}$$

$$\text{APP} \quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 a_2 : \tau_1}$$

$$\text{FUN} \quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x \quad ) a : \tau_0 \rightarrow \tau}$$

$$\text{GEN} \quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall(\alpha) \tau_0}$$

$$\text{INST} \quad \frac{\bar{\beta} \notin \text{ftv}(\forall(\bar{\alpha}) \bar{\tau}_0)}{\forall(\bar{\alpha}) \tau_0 \leq \forall(\bar{\beta}) \tau_0[\bar{\alpha} \leftarrow \bar{\tau}]}$$

$$\text{SUB} \quad \frac{\Gamma \vdash a : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash a : \tau_2}$$

# A universal type system

## Implicit System F:

$$\text{VAR} \quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau}$$

$$\text{APP} \quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 a_2 : \tau_1}$$

$$\text{FUN} \quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x \quad ) a : \tau_0 \rightarrow \tau}$$

$$\text{GEN} \quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall(\alpha) \tau_0}$$

$$\text{INST} \quad \frac{\bar{\beta} \notin \text{ftv}(\forall(\bar{\alpha}) \bar{\tau}_0)}{\forall(\bar{\alpha}) \tau_0 \leq \forall(\bar{\beta}) \tau_0[\bar{\alpha} \leftarrow \bar{\tau}]}$$

$$\text{SUB} \quad \frac{\Gamma \vdash a : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash a : \tau_2}$$

Add a construction for local bindings (perhaps derivable):

$$\text{LET} \quad \frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash a_2 : \tau}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau}$$

# A universal type system

**Implicit** System F:

$$\frac{\text{VAR} \quad z : \tau \in \Gamma}{\Gamma \vdash z : \tau}$$

$$\frac{\text{APP} \quad \Gamma \vdash \dots}{\Gamma \vdash \dots}$$

$$\frac{\text{GEN} \quad \Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall(\alpha) \tau_0}$$

Logical, canonical presentation of typing rules

- Covers many variations: F, ML, F<sup>η</sup>, F<sub>≤</sub>, ...
  - Vary the set of types.
  - Vary the instance relation between types.
- For ML, **just restrict** types to ML types.

Add a construction for local bindings (perhaps derivable):

$$\frac{\text{LET} \quad \Gamma \vdash a_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash a_2 : \tau}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau}$$

# A universal type system

**Implicit** System F:

$$\frac{\text{VAR} \quad z : \tau \in \Gamma}{\Gamma \vdash z : \tau}$$

$$\frac{\text{APP} \quad \Gamma \vdash \dots}{\Gamma \vdash \dots}$$

$$\frac{\text{GEN} \quad \Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash a : \forall(\alpha) \tau_0}$$

Logical, canonical presentation of typing rules

- Covers many variations: F, ML, F<sup>η</sup>, F<sub>≤</sub>, ...
  - Vary the set of types.
  - Vary the instance relation between types.
- For ML, **just restrict** types to ML types.

**Do never change the typing rules!**

Add a construction for local bindings (perhaps derivable):

$$\frac{\text{LET} \quad \Gamma \vdash a_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash a_2 : \tau}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau}$$

# Type inference is undecidable — in System F

## Of course, we must

- Use type annotations on function parameters **in some cases**.

## When?

- Always?
  - too many annotations are obfuscating.
- Alleviate some annotations by local type inference?
  - unintuitive and fragile (to program transformations).
- When parameters have polymorphic types?
  - still too many bothersome type annotations.

Not conservative extensions of ML

Are polymorphic types **less important** than monomorphic ones?

# Type inference is undecidable — in System F

## Of course, we must

- Use type annotations on function parameters **in some cases**.

## When?

- Always?
  - too many annotations are obfuscating.
- Alleviate some annotations by local type inference?
  - unintuitive and fragile (to program transformations).
- When parameters have polymorphic types?
  - still too many bothersome type annotations.

Not conservative extensions of ML

Are polymorphic types **less important** than monomorphic ones?

## Our choice

► explained below

- When (and only when) parameters are **used polymorphically**.

# Lack of principal types for applications

## The example of choice

let *choice* =  $\lambda(x) \lambda(y) \text{ if } true \text{ then } x \text{ else } y : \forall \beta . \beta \rightarrow \beta \rightarrow \beta$

let *id* =  $\lambda(z) z : \forall(\alpha) \alpha \rightarrow \alpha$

*choice id* :

# Lack of principal types for applications

## The example of choice

let *choice* =  $\lambda(x) \lambda(y) \text{ if } \textit{true} \text{ then } x \text{ else } y : \forall \beta . \beta \rightarrow \beta \rightarrow \beta$

let *id* =  $\lambda(z) z : \forall(\alpha) \alpha \rightarrow \alpha$

*choice id* :  $\begin{cases} \forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha) \end{cases}$

# Lack of principal types for applications

## The example of choice

let *choice* =  $\lambda(x) \lambda(y) \text{ if } \textit{true} \text{ then } x \text{ else } y : \forall \beta . \beta \rightarrow \beta \rightarrow \beta$

let *id* =  $\lambda(z) z : \forall(\alpha) \alpha \rightarrow \alpha$

*choice id* :  $\begin{cases} \forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha) \end{cases}$

No better choice in F!

# Lack of principal types for applications

## The example of choice

let *choice* =  $\lambda(x) \lambda(y)$  **if** *true* **then** *x* **else** *y* :  $\forall \beta. \beta \rightarrow \beta \rightarrow \beta$

let *id* =  $\lambda(z) z$  :  $\forall(\alpha) \alpha \rightarrow \alpha$

*choice id* :  $\begin{cases} \forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha) \end{cases}$

No better choice in F!

## The problem is serious and inherent

- Follows from rules INST, GEN, and APP.
- Should values be kept as polymorphic or as instantiated as possible?
- A type inference system can do both, but cannot choose.

# Lack of principal types for applications

## The example of choice

let *choice* =  $\lambda(x) \lambda(y)$  **if** *true* **then** *x* **else** *y* :  $\forall \beta. \beta \rightarrow \beta \rightarrow \beta$

let *id* =  $\lambda(z) z$  :  $\forall(\alpha) \alpha \rightarrow \alpha$

*choice id* :  $\begin{cases} \forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha) \end{cases}$

## The solution in iML<sup>F</sup>:

*choice id* :  $\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta$

# Lack of principal types for applications

## The example of choice

let *choice* =  $\lambda(x) \lambda(y)$  **if true then x else y** :  $\forall \beta \cdot \beta \rightarrow \beta \rightarrow \beta$

let *id* =  $\lambda(z) z$  :  $\forall(\alpha) \alpha \rightarrow \alpha$

$$\text{choice id} : \begin{cases} \forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha) \end{cases}$$

## The solution in iML<sup>F</sup>:

*choice id* :  $\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \beta \rightarrow \beta$

$$\leq \begin{cases} (\beta \rightarrow \beta) [\beta \leftarrow \forall(\alpha) \alpha \rightarrow \alpha] \\ \forall(\alpha) (\beta \rightarrow \beta) [\beta \leftarrow \alpha \rightarrow \alpha] \end{cases}$$

# The definition of iML<sup>F</sup>

## Types are stratified

$$\sigma ::= \begin{array}{l} \tau \quad \in F \\ | \quad \forall(\alpha \geq \sigma) \sigma \end{array}$$

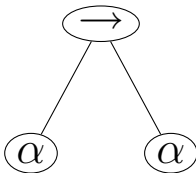
We can see and explain types by  $\leq_F$ -closed sets of System-F types:

$$\begin{aligned} \{\tau\} &\triangleq \{\tau' \mid \tau \leq_F \tau'\} \\ \{\forall(\alpha \geq \sigma) \sigma'\} &\triangleq \left\{ \forall(\bar{\beta}) \tau'[\alpha \leftarrow \tau] \mid \wedge \left( \begin{array}{l} \tau \in \{\sigma\} \wedge \tau' \in \{\sigma'\} \\ \bar{\beta} \# \text{ftv}(\forall(\alpha \geq \sigma) \sigma') \end{array} \right) \right\} \end{aligned}$$

Type instance  $\leq_I$  is set inclusion on the translations

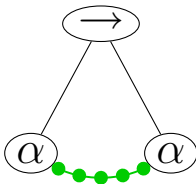
$$\sigma \leq_I \sigma' \iff \{\sigma\} \supseteq \{\sigma'\}$$

# Simple types

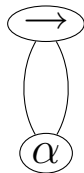
$$\alpha \rightarrow \alpha$$


# Simple types

$$\alpha \rightarrow \alpha$$

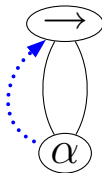


# Simple types

$$\alpha \rightarrow \alpha$$


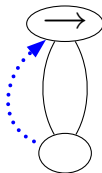
# System-F types

$$\forall(\alpha) \alpha \rightarrow \alpha$$



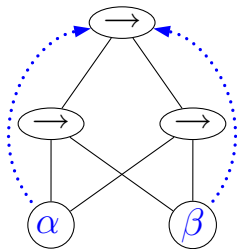
# System-F types

$$\forall(\alpha) \alpha \rightarrow \alpha$$



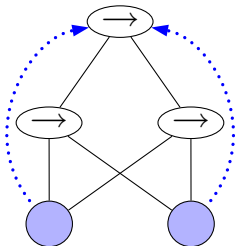
# System-F types

$$\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$



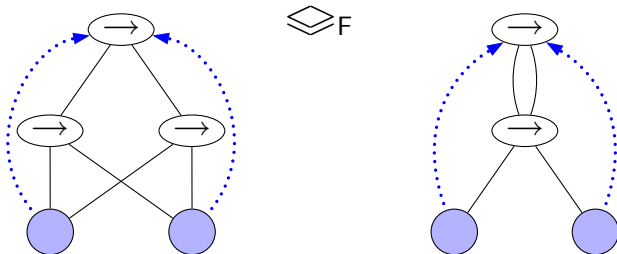
# System-F types

$$\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$



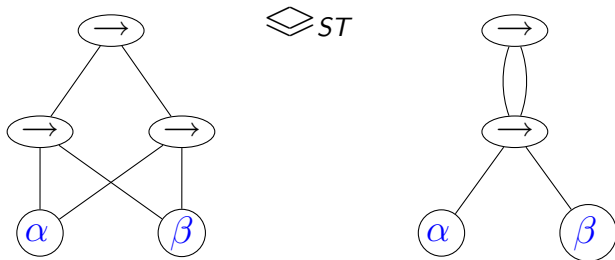
# System-F types

$$\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$



# System-F types

$$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

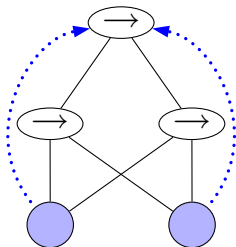
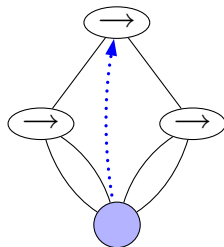


Sharing of inner nodes:

- Coming from the dag-representation of simple types.
- Canonical (unique) representation if disallowed.

# System-F types

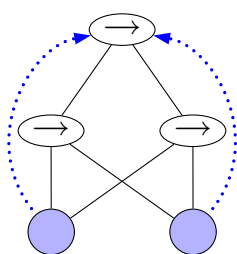
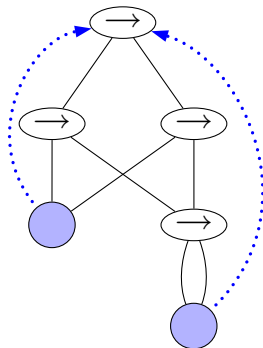
$$\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$


 $\leq_F$ 


$$\forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$

# System-F types

$$\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

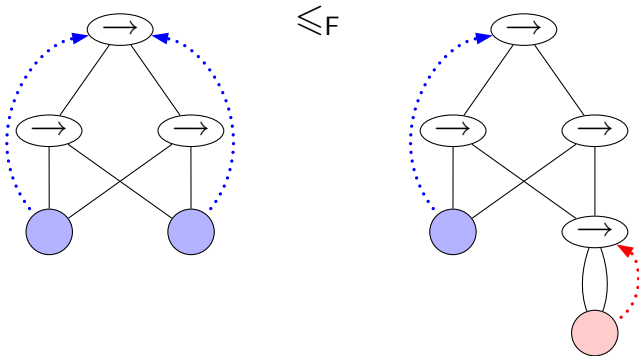

 $\leq_F$ 


$$\forall(\alpha) \forall(\gamma) (\alpha \rightarrow \gamma \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma \rightarrow \gamma$$

## System-F types

▶ more

$$\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$



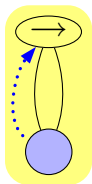
$$\forall(\alpha)(\alpha \rightarrow \forall(\gamma) \gamma \rightarrow \gamma) \rightarrow \alpha \rightarrow \forall(\gamma) \gamma \rightarrow \gamma$$

Types in *iML<sup>F</sup>*

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$

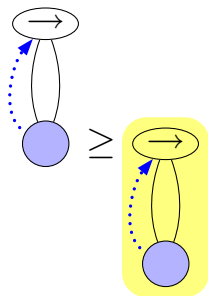
Types in *iML<sup>F</sup>*

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



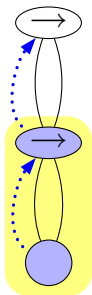
Types in iML<sup>F</sup>

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



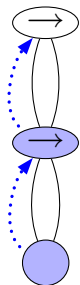
Types in iML<sup>F</sup>

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



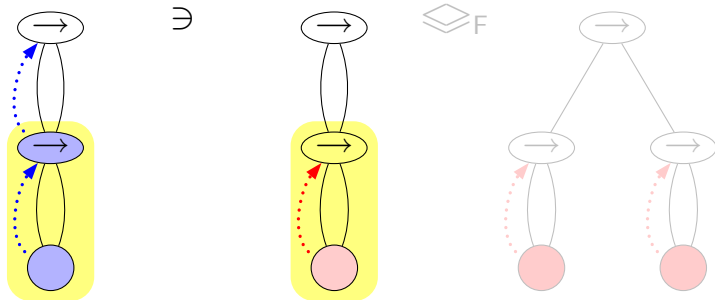
Types in iML<sup>F</sup>

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



Types in iML<sup>F</sup>

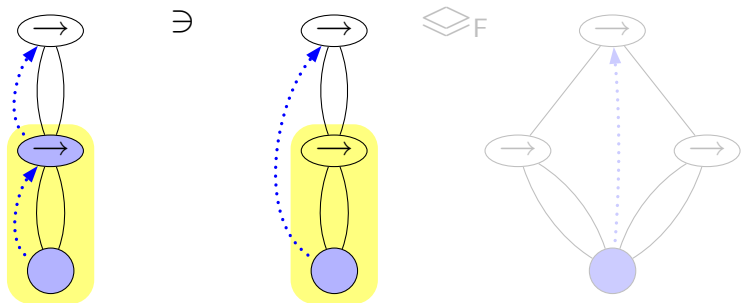
$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



$$(\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \forall(\alpha) \alpha \rightarrow \alpha$$

Types in iML<sup>F</sup>

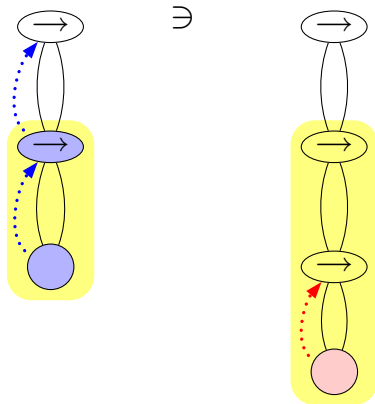
$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



$$\forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$$

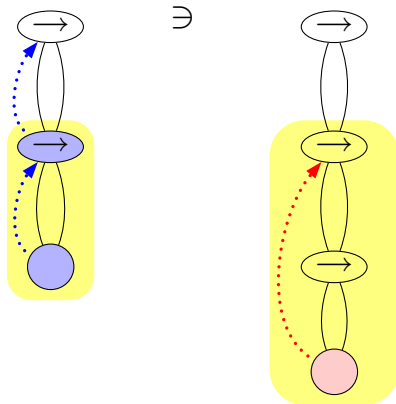
Types in iML<sup>F</sup>

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



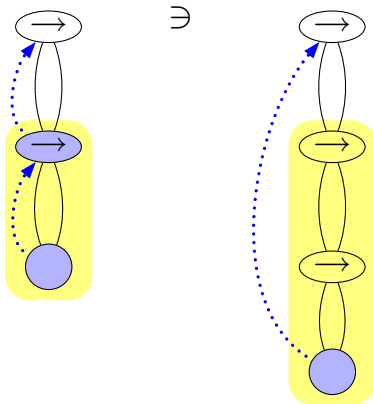
Types in iML<sup>F</sup>

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



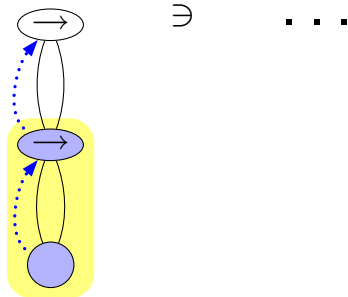
Types in iML<sup>F</sup>

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$



Types in iML<sup>F</sup>

$$\forall(\beta \geq \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$

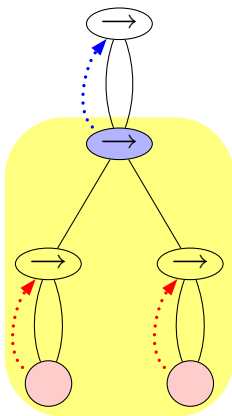


The semantics cannot be captured by

- a finite set of System-F types up to  $\leq$
- a finite intersection type.

iML<sup>F</sup> types

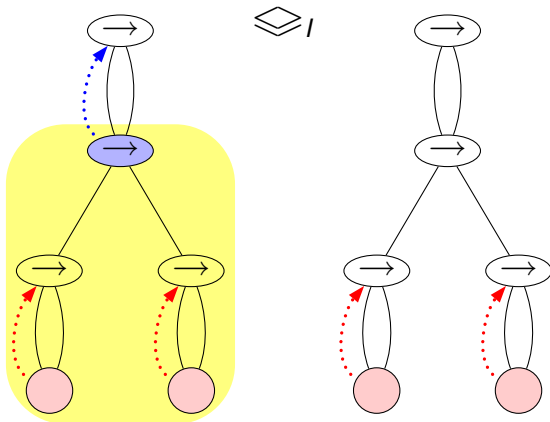
$$\forall(\beta \geq (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)) \rightarrow \beta \rightarrow \beta$$





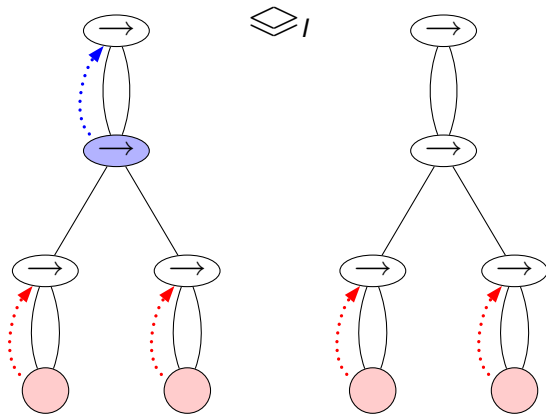
iML<sup>F</sup> types

$$\forall(\beta \geq (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)) \rightarrow \beta \rightarrow \beta$$



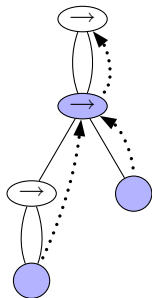
iML<sup>F</sup> types

$$\forall(\beta \geq (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)) \rightarrow \beta \rightarrow \beta$$



# Type instance $\leq$ in iML<sup>F</sup>

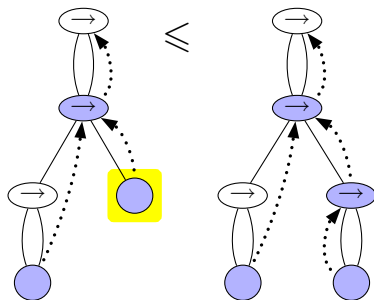
Only four atomic instance operations, and **only two new**.



# Type instance $\leq$ in iML<sup>F</sup>

Only four atomic instance operations, and **only two new**.

Grafting

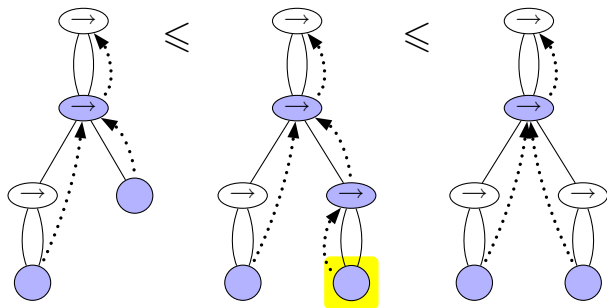


# Type instance $\leq$ in iML<sup>F</sup>

Only four atomic instance operations, and **only two new**.

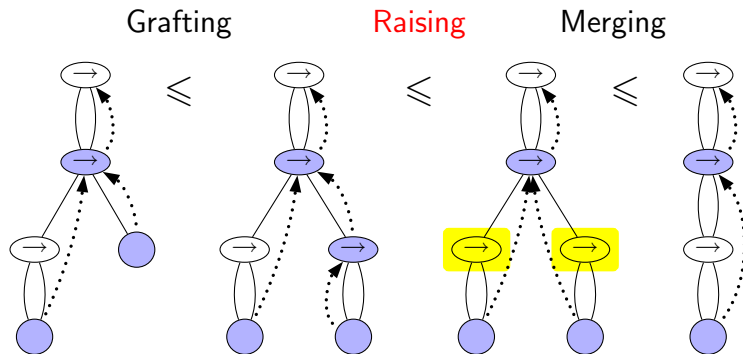
Grafting

Raising



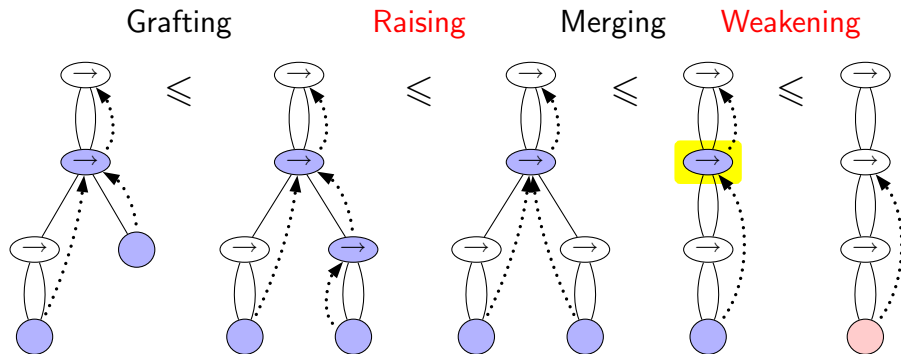
# Type instance $\leq$ in iML<sup>F</sup>

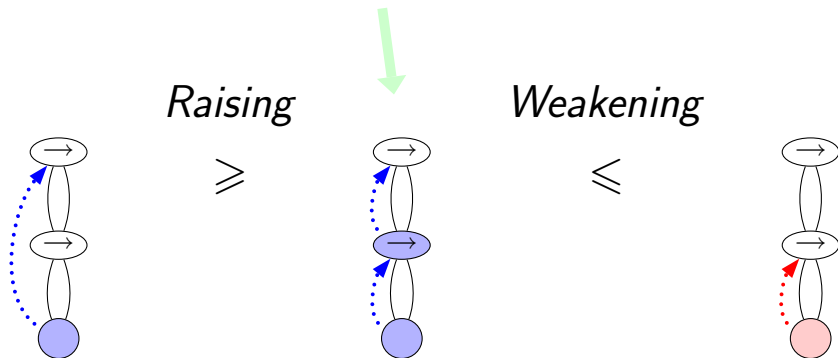
Only four atomic instance operations, and **only two new**.



# Type instance $\leq$ in iML<sup>F</sup>

Only four atomic instance operations, and **only two new**.



Checking the example **choice id**

# Outline

## 1 Design

- $iML^F$ : an implicit-typed extension of System F
- Types explained
- $eML^F$ : an explicitly-typed version of  $iML^F$

## 2 Uses and Implementation

- Examples
- Type inference
- Restrictions and extensions

# Design of $eML^F$

## Goal

Find a restriction  $iML^F$  where programs that would require **guessing** polymorphism are ill-typed.

## Guideline

[← design](#)

Function parameters that are used **polymorphically** (and only those) need an annotation.

# First-order inference with second-order types

## Easy examples

$\lambda(z) z$  :  $\forall(\alpha) \alpha \rightarrow \alpha$  as in ML

let  $x = \lambda(z) z$  in  $x x$  :  $\forall(\alpha) \alpha \rightarrow \alpha$  as in ML

$\lambda(x) x x$  : ill-typed!  $x$  is used polymorphically

$\lambda(x : \forall(\alpha) \alpha \rightarrow \alpha) x x$  :  $(\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)$

# First-order inference of second order types

## More challenging example

$(\lambda(z) z) (a : \sigma)$  where  $\sigma$  is truly polymorphic

# First-order inference of second order types

## More challenging example

$(\lambda(z) z) (a : \sigma)$  where  $\sigma$  is truly polymorphic

- $z$  must carry values of a **polymorphic type**.
- but  $z$  is not **used polymorphically**.
- Indeed, it can be typed in System F as

$(\Lambda\alpha. \lambda(z : \alpha) z) [\sigma] (a : \sigma)$

# First-order inference of second order types

## More challenging example

$(\lambda(z) z) (a : \sigma)$  where  $\sigma$  is truly polymorphic

**ACCEPT**

- $z$  must carry values of a **polymorphic type**.
- but  $z$  is not **used polymorphically**.
- Indeed, it can be typed in System F as

$(\Lambda\alpha. \lambda(z : \alpha) z) [\sigma] (a : \sigma)$

# First-order inference of second order types

## More challenging example

$$\lambda(z) (z (a : \sigma))$$

# First-order inference of second order types

## More challenging example

$$\lambda(z) (z (a : \sigma))$$

- $z$  have the polymorphic type  $\sigma \rightarrow \sigma$  ?
- $z$  is node **used polymorphically**:  
polymorphism is only carried out from the argument to the result.

# First-order inference of second order types

## More challenging example

$$\lambda(z) (z (a : \sigma))$$
**ACCEPT**

- $z$  have the polymorphic type  $\sigma \rightarrow \sigma$  ?
- $z$  is node **used polymorphically**:  
polymorphism is only carried out from the argument to the result.

# Abstracting second-order polymorphism as first-order types

## Solution

- 1) Disallow second-order types under arrows, e.g. such as  $\sigma_{\text{id}} \rightarrow \sigma_{\text{id}}$
- 2) Instead, allow type variables to stand for polymorphic types:

write  $\forall(\alpha \Rightarrow \sigma_{\text{id}}) \alpha \rightarrow \alpha$

read “ $\alpha \rightarrow \alpha$  where  $\alpha$  **abstracts**  $\sigma_{\text{id}}$ ”

means  $\sigma_{\text{id}} \rightarrow \sigma_{\text{id}}$

## Mechanism

- 1) function parameters must be monomorphic (but may be abstract).
- 2) forces all polymorphism to be abstracted away in the context.

$$\frac{\alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash x : \alpha}{\alpha \Rightarrow \sigma_{\text{id}} \vdash \lambda(x) x : \alpha \rightarrow \alpha}$$

$$\lambda(x) x : \forall(\alpha \Rightarrow \sigma_{\text{id}}) \alpha \rightarrow \alpha$$

# Abstracting second-order polymorphism

[▶ more](#)

Key point: abstraction is directional

$$\alpha \Rightarrow \sigma \vdash \sigma \leq \alpha$$

~~$$\alpha \Rightarrow \sigma \vdash \alpha \leq \sigma$$~~

Hence,

$$\frac{\vdash a : \sigma}{\alpha \Rightarrow \sigma \vdash a : \alpha} \quad \alpha \Rightarrow \sigma, z : \alpha \rightarrow \alpha \vdash z : \alpha \rightarrow \alpha$$


---


$$\alpha \Rightarrow \sigma, z : \alpha \rightarrow \alpha \vdash z a : \alpha$$


---


$$\alpha \Rightarrow \sigma \vdash \lambda(z) z a : (\alpha \rightarrow \alpha) \rightarrow \alpha$$


---


$$\vdash \lambda(z) z a : \forall (\alpha \Rightarrow \sigma) (\alpha \rightarrow \alpha) \rightarrow \alpha$$

# Abstracting second-order polymorphism

[▶ more](#)

Key point: abstraction is directional

$$\alpha \Rightarrow \sigma \vdash \sigma \leq \alpha$$

~~$$\alpha \Rightarrow \sigma \vdash \alpha \leq \sigma$$~~

But,

~~$$\alpha \Rightarrow \sigma_{\text{id}}, z : \alpha \vdash z : \alpha$$~~

$$\alpha \Rightarrow \sigma_{\text{id}}, z : \alpha \vdash z : \sigma_{\text{id}}$$

$$\alpha \Rightarrow \sigma_{\text{id}}, z : \alpha \vdash z : \alpha \rightarrow \alpha$$

$$\alpha \Rightarrow \sigma_{\text{id}}, z : \alpha \vdash z : \alpha$$

$$\alpha \Rightarrow \sigma_{\text{id}}, z : \alpha \vdash z z : \alpha$$

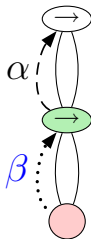
$$\alpha \Rightarrow \sigma_{\text{id}} \vdash \lambda(z) z z : \alpha \rightarrow \alpha$$

$$\vdash \lambda(z) z z : \forall(\alpha \geq \sigma_{\text{id}}) \alpha \rightarrow \alpha$$

# Types in eML<sup>F</sup>

Introduce a new binder for abstraction

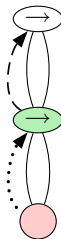
$$\forall(\alpha \Rightarrow \forall(\beta) \beta \rightarrow \beta) \alpha \rightarrow \alpha$$



# Types in eML<sup>F</sup>

Introduce a new binder for abstraction

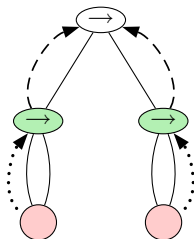
$$\forall(\alpha \Rightarrow \forall(\beta) \beta \rightarrow \beta) \alpha \rightarrow \alpha$$



Types in eML<sup>F</sup>

Introduce a new binder for abstraction

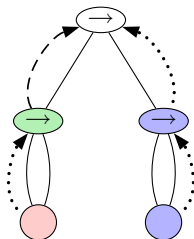
$$\forall(\alpha \Rightarrow \forall(\beta) \beta \rightarrow \beta) \forall(\alpha' \Rightarrow \forall(\beta) \beta \rightarrow \beta) \alpha \rightarrow \alpha'$$



Types in eML<sup>F</sup>

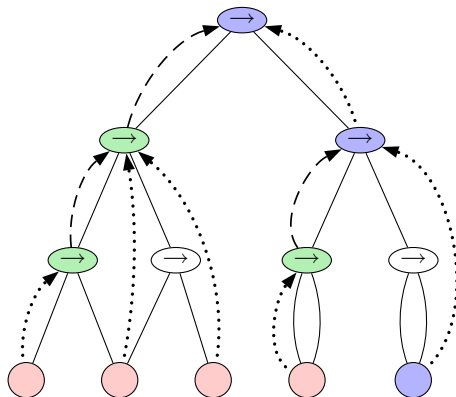
Introduce a new binder for abstraction

$$\forall(\alpha \Rightarrow \forall(\beta) \beta \rightarrow \beta) \forall(\alpha' \geq \forall(\beta) \beta \rightarrow \beta) \alpha \rightarrow \alpha'$$



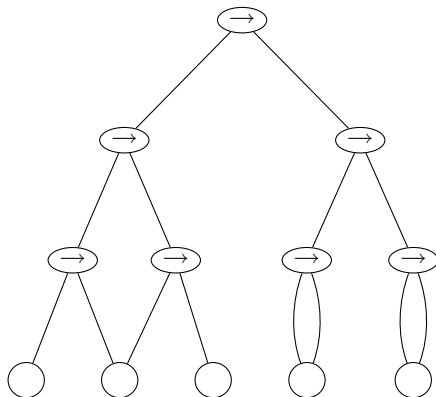
# Types, graphically

= first-order term-dag + a binding tree



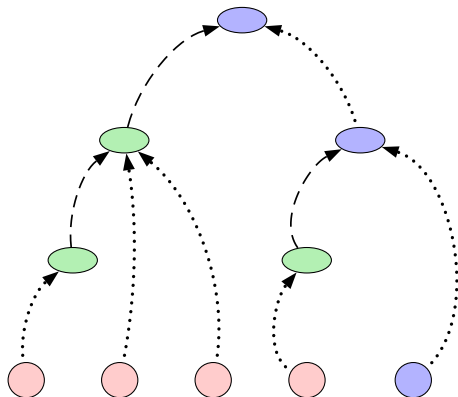
# Types, graphically

= first-order term-dag + a binding tree



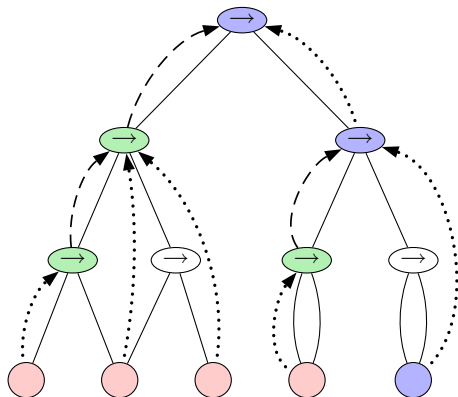
# Types, graphically

= first-order term-dag + a binding tree



# Types, graphically

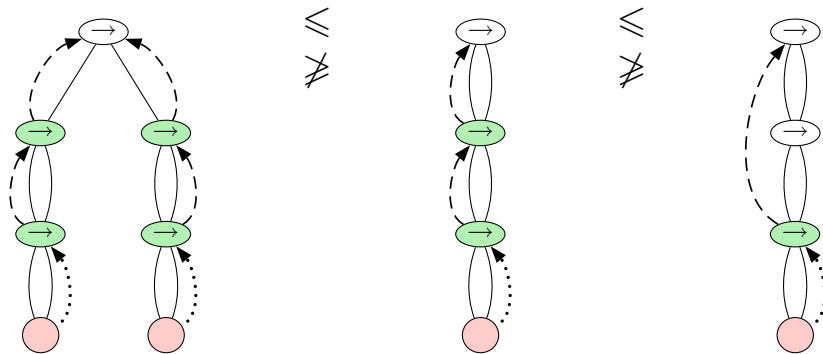
= first-order term-dag + a binding tree



+ well-formedness conditions relating the two

# Type instance $\leq$ in eML<sup>F</sup>

Sharing and binding of abstract nodes matter



Grafting, Merging, Raising, Weakening  
Unchanged.

# Type annotations

## Recovering the missing power

$$(\leq) \subset (\leq_I)$$

- $\leq$  is weaker than  $\leq_I$ , as sharing and binding of abstract nodes matters.

# Type annotations

## Recovering the missing power

$$(\leq) \subset (\leq_I) = (\leq \cup \diamond_I)^* = (\leq; \diamond_I)$$

- $\leq$  is weaker than  $\leq_I$ , as sharing and binding of abstract nodes matters.
- Use **explicit type annotations** to recover  $(\diamond_I \setminus \leq)$ .  
Notice that the weaker  $\leq$ , the more annotations will be required.

# Type annotations

## Recovering the missing power

$$(\leq) \subset (\leq_I) = (\leq \cup \diamond_I)^* = (\leq; \diamond_I)$$

- Intuitively,

$$\frac{\Gamma \vdash a : \tau \quad \tau \diamond_I \tau'}{\Gamma \vdash (a : \tau') : \tau'}$$

- Actually, use coercion functions:

$$(- : \sigma) : \quad \forall(\alpha \Rightarrow \sigma) \forall(\alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha'$$

- Add syntactic sugar  $\lambda(x : \sigma) a \stackrel{\Delta}{=} \lambda(x) \text{ let } x = (x : \sigma) \text{ in } a$   
 $\equiv \lambda(x) a[x \leftarrow (x : \sigma)]$

# Type annotations

## Recovering the missing power

$$(\leq) \subset (\leq_I) = (\leq \cup \diamond_I)^* = (\leq; \diamond_I)$$

- Intuitively,

$$\frac{\Gamma \vdash a : \tau \quad \tau \diamond_I \tau'}{\Gamma \vdash (a : \tau') : \tau'}$$

- Actually, use coercion functions:

$$(- : \exists(\bar{\beta}) \sigma) : \forall(\bar{\beta}) \forall(\alpha \Rightarrow \sigma) \forall(\alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha'$$

- Add syntactic sugar  $\lambda(x : \sigma) a \quad \begin{array}{l} \triangleq \\ \equiv \end{array} \begin{array}{l} \lambda(x) \text{ let } x = (x : \sigma) \text{ in } a \\ \lambda(x) a[x \leftarrow (x : \sigma)] \end{array}$

# Type annotations

Remember  $\alpha \Rightarrow \sigma, x : \alpha \vdash x : \sigma$

- Prevents typing  $\lambda(x) x x$

With an annotation  $\alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \sigma$

[▶ more](#)

- Allows typing  $\lambda(x : \sigma_{\text{id}}) x x$

# Outline

- 1 Design
  - $iML^F$ : an implicit-typed extension of System F
  - Types explained
  - $eML^F$ : an explicitly-typed version of  $iML^F$
- 2 Uses and Implementation
  - Examples
  - Type inference
  - Restrictions and extensions

# About principal types

## Fact

- Programs have principal types, **given with their type annotations**.

## Programs with type annotations

- Two versions of the same program, but with different type annotations, usually have different principal types.

## Programs typable without type annotations

- Exactly ML programs.
- But usually have a more general type than in ML (*e.g. choice id*)
- Annotations may still be useful to get more polymorphism.

# Robustness to small program transformations

## Agreed

- Programmers must be free of choosing their programming patterns/styles.
- Programs should be maintainable.

## Therefore

- Programs should be stable under some small, but **important** program transformations.

# Robustness to small program transformations

$a \subseteq a'$  means *all typings of  $a$  are typings of  $a'$*

## Let-conversion

$$\text{let } x = a_1 \text{ in } a_2 \subseteq a_2[x \leftarrow a_1]$$

Common subexpression can be factored out.

## Redefine application

$$a_1 a_2 \subseteq (\lambda(f) \lambda(x) f x) a_1 a_2$$

Many functionals, such as **maps** are typed as applications.

## $\eta$ -conversion of functional expressions

$$a \subseteq \lambda(x) a x$$

Delay the evaluation.

## Reordering of arguments

$$a a_1 a_2 \subseteq (\lambda(x) \lambda(y) a y x) a_2 a_1$$

## Curryfication

$$a (a_1, a_2) \subseteq (\lambda(x) \lambda(y) a (x, y)) a_1 a_2$$

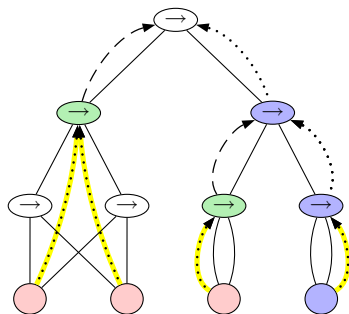
All valid in ML<sup>F</sup>

# Printing types

Only overlined bindings need to be drawn

Leave implicit bindings that are

- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.



$$(\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha) \alpha \rightarrow \alpha)$$



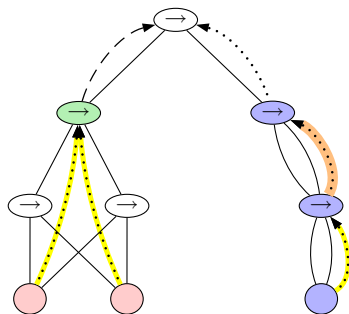


# Printing types

Only overlined bindings need to be drawn

Leave implicit bindings that are

- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.



$$(\forall(\alpha) \forall(\beta) (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \rightarrow \forall(\gamma \geq \sigma_{\text{id}}) \gamma \rightarrow \gamma$$

# Examples

## Library functions

```
let rec fold f v = function
```

```
| Nil → v
```

```
| Cons (h, t) → fold f (f h t) t;;
```

```
val fold :  $\forall(\alpha) \forall(\beta) (\alpha \rightarrow \alpha \text{ list} \rightarrow \beta) \rightarrow \beta \rightarrow \alpha \text{ list} \rightarrow \beta$ 
```

## Few type annotations are needed in practice

- No dummy/annoying/unpredictable annotations.

## Output types are usually readable

- Most inner binding edges may be left implicit.
- Many library functions libraries keep their ML type in  $ML^F$ , modulo the syntactic sugar.

## More examples



### Church's numerals

```
type nat =  $\forall (\alpha) (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ ;;  
let zero = fun f x  $\rightarrow x$ ;;  
val zero :  $\forall(\alpha) \alpha \rightarrow (\forall(\beta) \beta \rightarrow \beta)$ 
```

### With type annotations on the iterator

```
let succ (n : nat) = fun f x  $\rightarrow n f (f x)$ ;;  
val succ : nat  $\rightarrow (\forall (\alpha) (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha)$   
let add (n : nat) m = n succ m;;  
val add : nat  $\rightarrow (\forall (\alpha) (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha)$   
let mul n (m : nat) = m (add n) zero;;  
mul : nat  $\rightarrow nat \rightarrow (\forall(\alpha) (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha)$ 
```

# More examples

[▶ skip](#)

## Church's numerals

```

type nat =  $\forall (\alpha) (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ ;
let zero = fun f x  $\rightarrow x$ ;
val zero :  $\forall(\alpha) \alpha \rightarrow (\forall(\beta) \beta \rightarrow \beta)$ 

```

## Without type annotations

```

let succ n = fun f x  $\rightarrow n f (f x)$ ;
val succ :  $\forall (\alpha, \beta, \gamma) ((\alpha \rightarrow \beta) \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$ 
let add n m = n succ m;
val add :  $\forall(\delta \geq \forall(\alpha, \beta, \gamma) ((\alpha \rightarrow \beta) \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma)$ 
            $\forall(\varepsilon, \varphi) (\delta \rightarrow \varepsilon \rightarrow \varphi) \rightarrow \varepsilon \rightarrow \varphi$ 

```

In ML:

```

val add :  $\forall (\alpha, \beta, \gamma, \varepsilon, \varphi) (((\alpha \rightarrow \beta) \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma)$ 
            $\rightarrow \varepsilon \rightarrow \varphi) \rightarrow \varepsilon \rightarrow \varphi$ 

```

## More examples

[▶ skip](#)

### Church's numerals

```
type nat =  $\forall (\alpha) (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ ;;  
let zero = fun f x  $\rightarrow$  x;;  
val zero :  $\forall(\alpha) \alpha \rightarrow (\forall(\beta) \beta \rightarrow \beta)$ 
```

### Mandatory type annotations

```
let mul n m = m (add n) zero;;  
let mul' = (mul : : nat  $\rightarrow$  nat  $\rightarrow$  nat);;  
fails
```

$\text{ML}^F$  without any type annotation at all does not do better than ML!

# Unification algorithm

## Computes principal unifiers, in three steps

- Computes the underlying dag-structure by first-order unification.
- Computes the binding structure
  - by raising binding edges
  - as little as possible to maintain well-formedness.
- Checks that no locked binding edge (in red) has been raised or merged.

## Complexity

- Same as first-order unification. Other passes are in linear time.
- $O(n)$  (or  $O(n\alpha(n))$  if incremental).

## Note

- The algorithm performs “*first-order unification of second-order types*”.

# Type inference

## Proceeds much as in ML

- Implement type-instantiation by copying the polymorphic part.
- Use unification to solve typing constraints.
- **Generalize as much as possible at every step (not just at every let).**

## Type inference with typing constraints

[▶ Demo](#)

## Complexity in $O(kn(\alpha(kn) + d)) \approx O(kdn)$

- As for ML (see McAllester).
  - $k$  is the maximal size of types (usually not too large)
  - $d$  is the maximal nesting of type schemes.
- However, ML and  $ML^F$  differs on  $d$ , which is:
  - the left-nesting of let-bindings in ML
  - the maximum height of an expression in  $ML^F$   
(Still, does not grow on the right of let-bindings).

# Variations on $ML^F$

## Shallow $ML^F$

The version we presented is a “downgraded” version of  $ML^F$ .

- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be F types.

# Variations on $ML^F$

## Shallow $ML^F$

The version we presented is a “downgraded” version of  $ML^F$ .

- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be F types.

## Full $ML^F$

- No stratification, more expressive.
- All interesting properties are preserved.
- Algorithms are mostly unchanged.
- We loose the interpretation of types as sets of System-F types.

# Variations on $ML^F$

## Shallow $ML^F$

The version we presented is a “downgraded” version of  $ML^F$ .

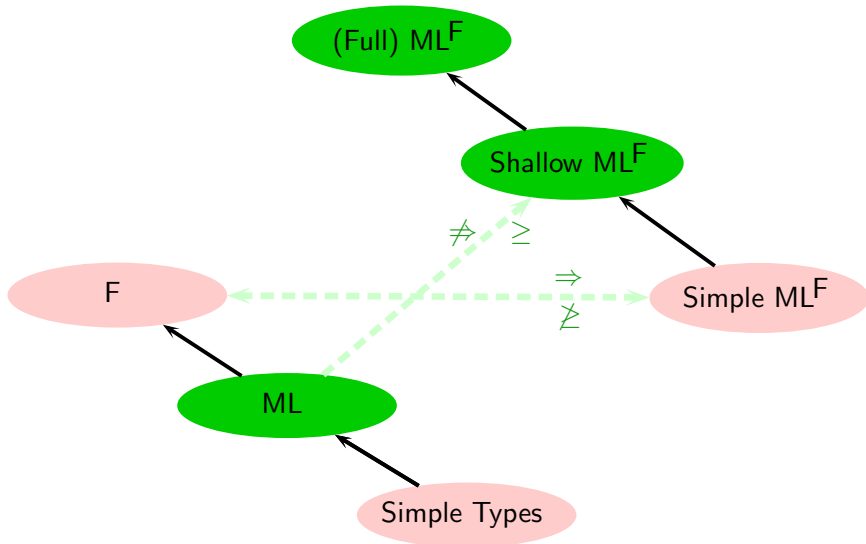
- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be F types.

## Simple $ML^F$

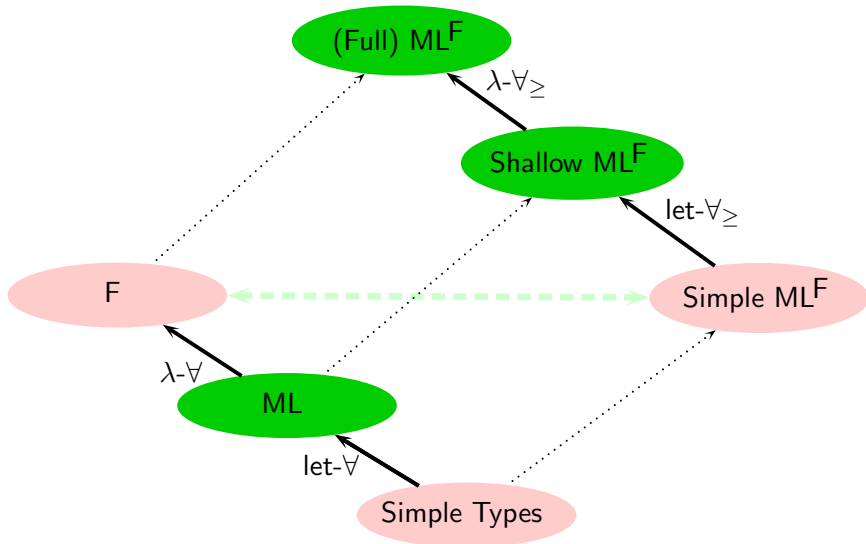
Remove instance bindings  $\geq$ , keep abstract bindings  $\Rightarrow$ .

- Equivalent to System F.
- Principal types are lost (no type inference).

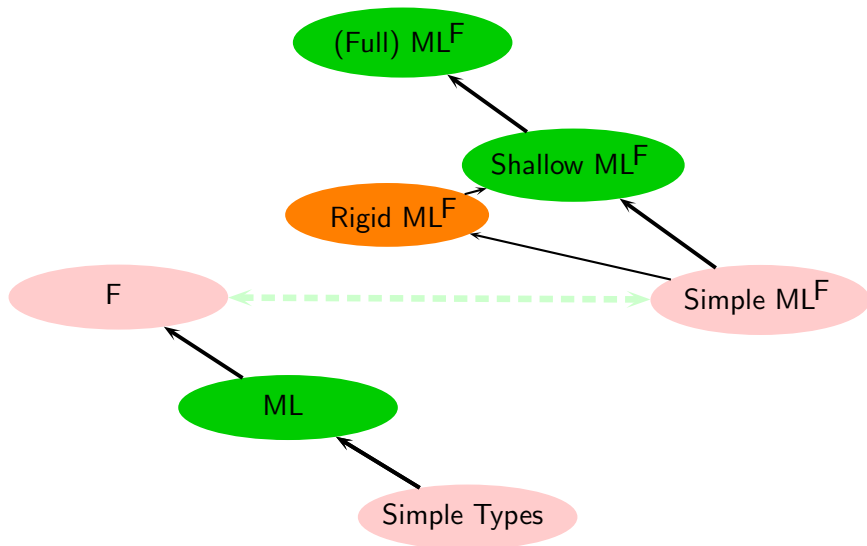
# A hierarchy of languages



# A hierarchy of languages



# A hierarchy of languages

▶ Rigid  $ML^F$ 

# Extensions

## Primitive Existential types

- Encoding with existential types works well (only annotate at creation).
- Can more be done with primitive existential ?

## (Equi-) recursive types

- Easy when cycles do not contain quantifiers.
- Cycles that crosses quantifiers are difficult.

## Higher-order types

- Use two quantifiers (explicit coercions between the two permitted)
  - $\forall^F$  for fully explicit type abstractions and
  - $\forall^{ML^F}$  for implicit  $ML^F$  polymorphism.
- Restrict  $\forall^{ML^F}$  to the first-order type variables.
- Can  $\forall^{ML^F}$  also be used at higher-order kinds?

# Papers and prototypes

## Talk mainly based on

- Recasting-ML<sup>F</sup> *with Didier Le Boltan.*
- A Graphical Presentation of ML<sup>F</sup> Types, *with Boris Yakobowski.*

## Other papers and online prototype at

- <http://gallium.inria.fr/~remy/mlf/>

## See also Daan Leijen's papers and prototypes

- <http://research.microsoft.com/users/daan/pubs.html>

# Conclusions

## Just two things to remember

- $ML^F$  allows function parameters to implicitly carry polymorphic values that are used **monomorphically**.
- Type annotations are required only to allow function parameters to carry (polymorphic) values that are used **polymorphically**.

## $ML^F$ design, use, and implementation are close to ML

- $ML^F$  piggy-backs on ML type-schemes and generalization mechanism.
- Part of the credits should be returned to the great designers of ML.

## Hopefully

- ML users will feel “*at home*”.
- Other users will also appreciate the convenience of type inference.

# Appendix

- 3 Type inference demo
- 4 More examples: encoding of existential types
- 5 About Rigid  $ML^F$
- 6 Questions
  - What is an Intermediate language for  $ML^F$
  - Sharing of abstract nodes is irreversible (implicitly)
- 7 Details of slides
  - Another example of System F types
  - Abstraction in action

# Type inference with typing constraints (demo)

▶ skip

◀ back

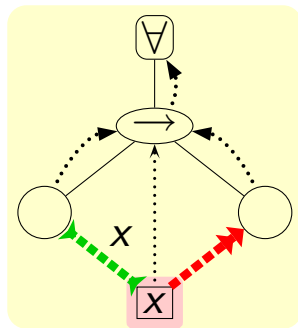
$\lambda(x) x$

## Type inference with typing constraints (demo)

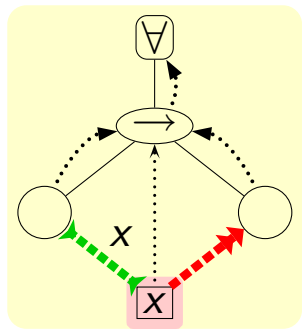
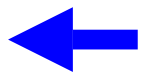
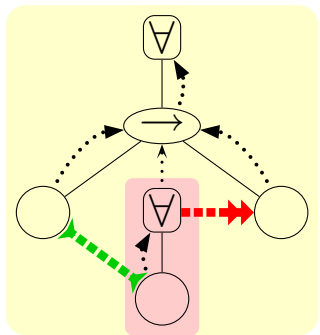
▶ skip

◀ back

$$\lambda(x) x$$



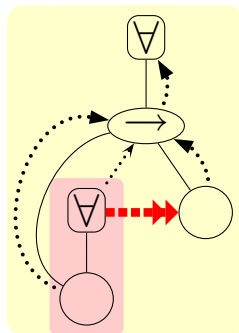
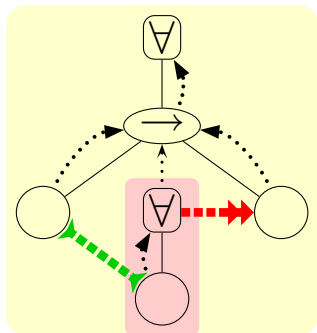
# Type inference with typing constraints (demo)



# Type inference with typing constraints (demo)

▶ skip

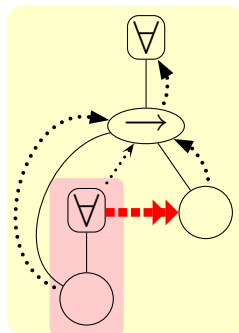
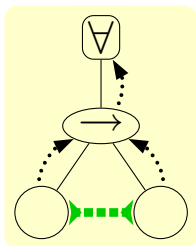
◀ back



# Type inference with typing constraints (demo)

▶ skip

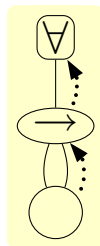
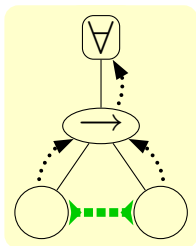
◀ back



# Type inference with typing constraints (demo)

▶ skip

◀ back



## Type inference with typing constraints (demo)

▶ skip

◀ back

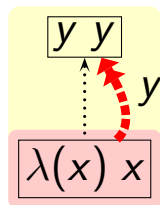
```
let  $y = \lambda(x) x$   
  in  $y y$ 
```

## Type inference with typing constraints (demo)

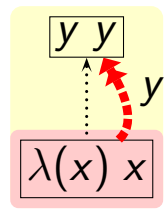
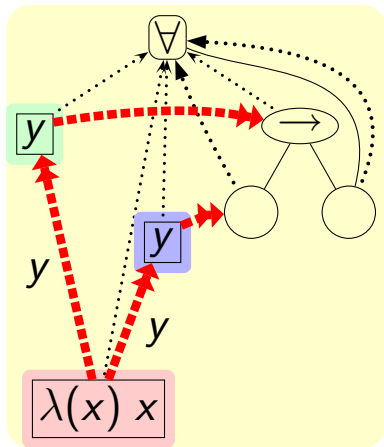
▶ skip

◀ back

```
let y = λ(x) x
    in y y
```



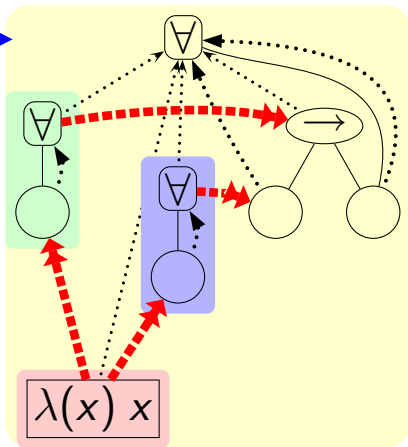
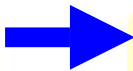
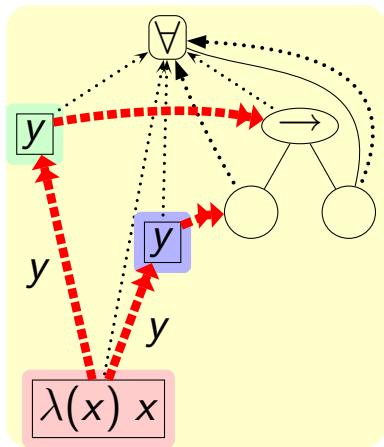
# Type inference with typing constraints (demo)



# Type inference with typing constraints (demo)

▶ skip

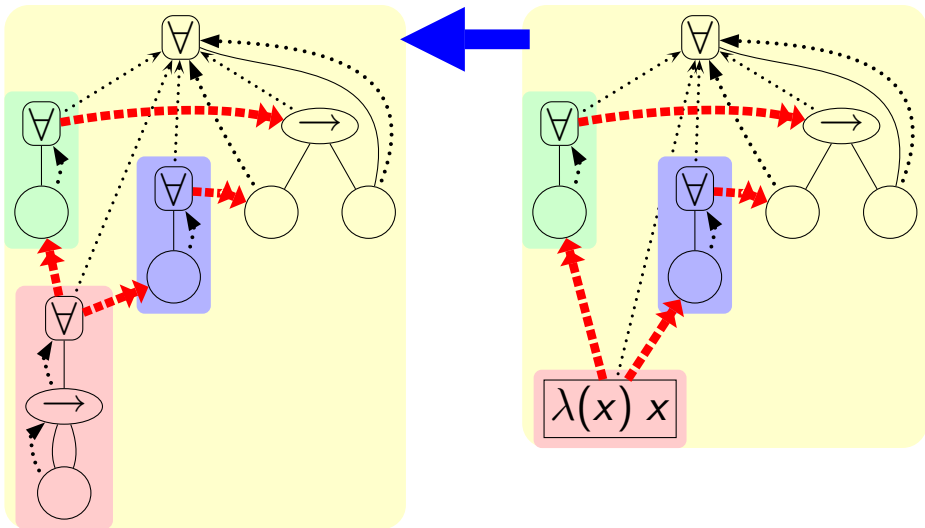
◀ back



# Type inference with typing constraints (demo)

▶ skip

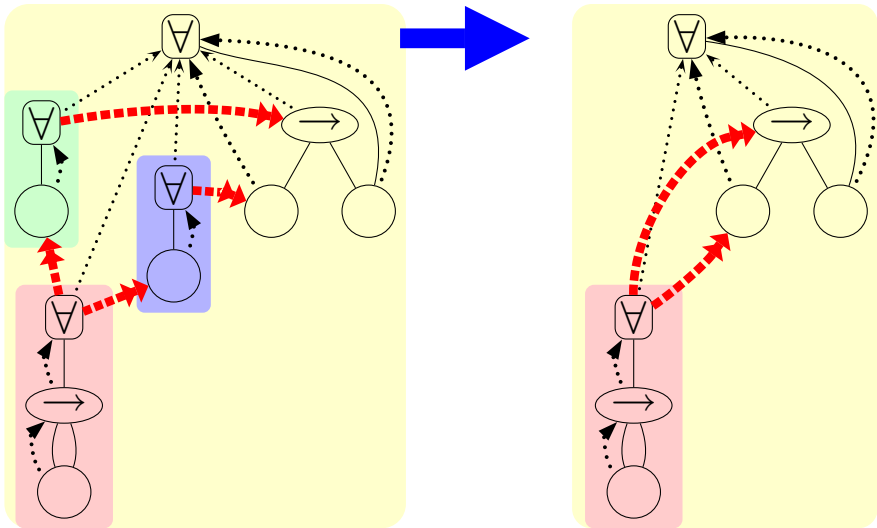
◀ back



# Type inference with typing constraints (demo)

▶ skip

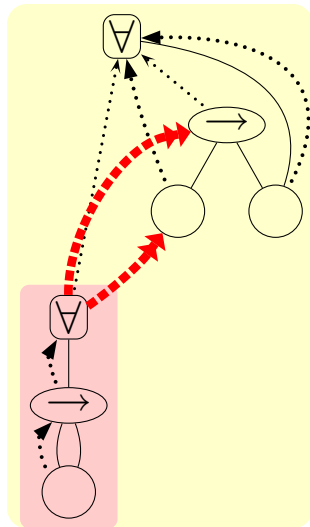
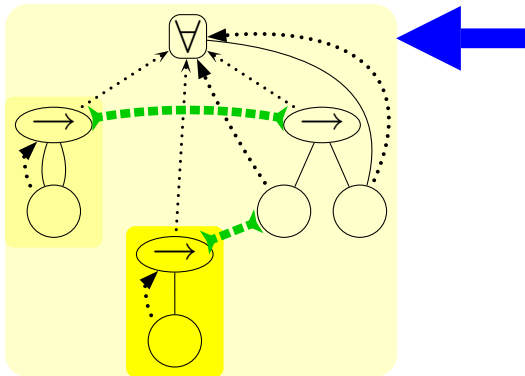
◀ back



# Type inference with typing constraints (demo)

▶ skip

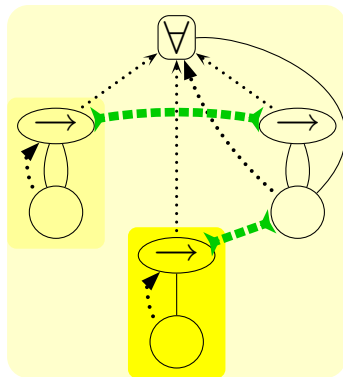
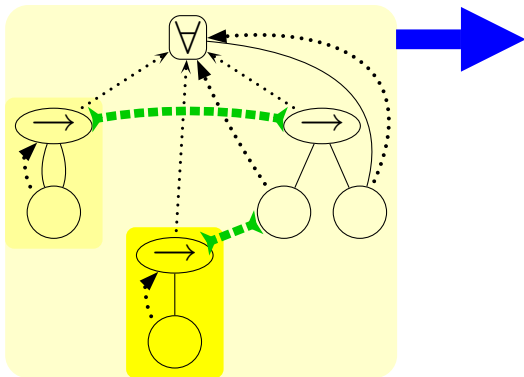
◀ back



# Type inference with typing constraints (demo)

▶ skip

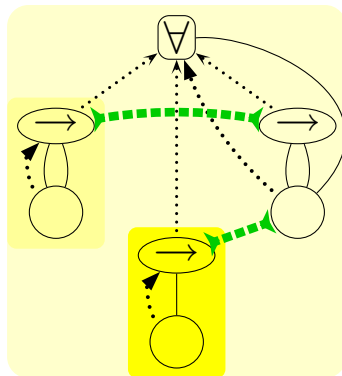
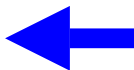
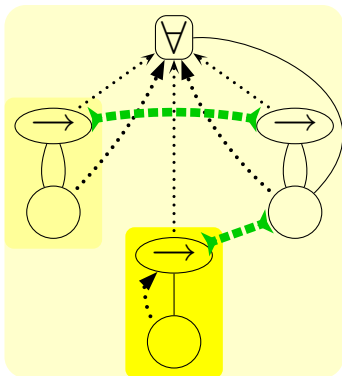
◀ back



# Type inference with typing constraints (demo)

▶ skip

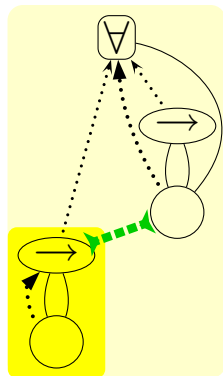
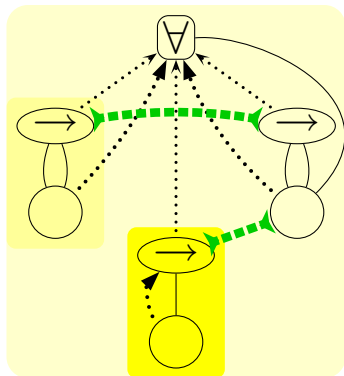
◀ back



# Type inference with typing constraints (demo)

▶ skip

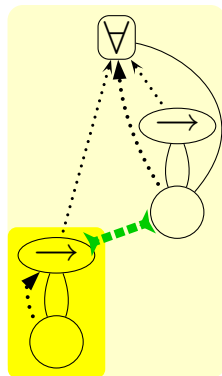
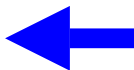
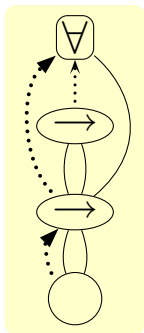
◀ back



# Type inference with typing constraints (demo)

▶ skip

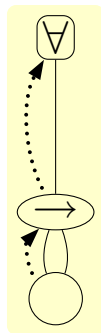
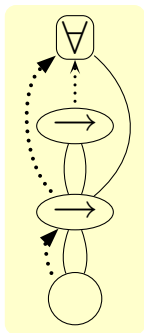
◀ back



# Type inference with typing constraints (demo)

▶ skip

◀ back



## Type inference with typing constraints (demo)

▶ skip

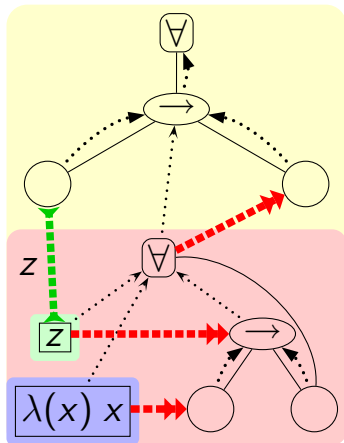
◀ back

 $\lambda(z) z (\lambda(x) x)$

# Type inference with typing constraints (demo)

▶ skip

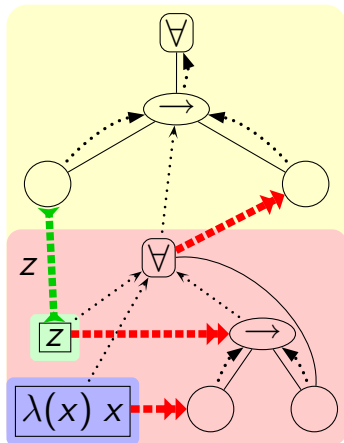
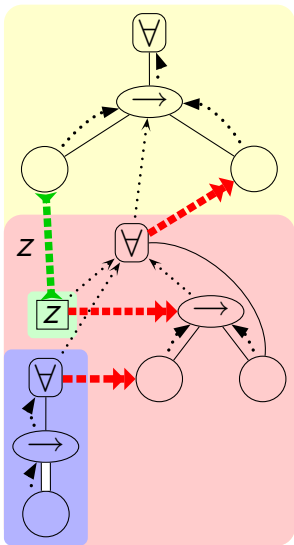
◀ back

$$\lambda(z) z (\lambda(x) x)$$


# Type inference with typing constraints (demo)

▶ skip

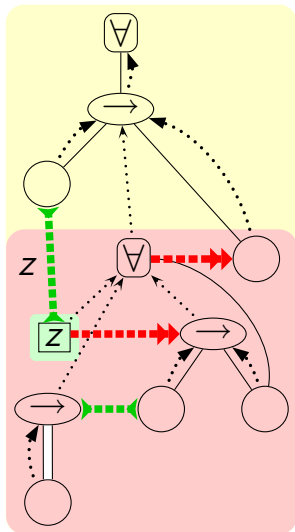
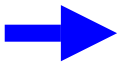
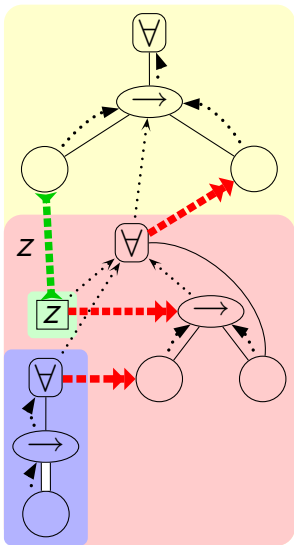
◀ back



# Type inference with typing constraints (demo)

▶ skip

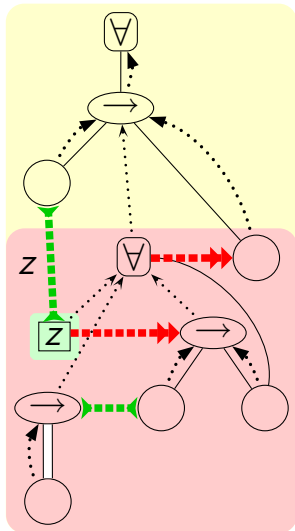
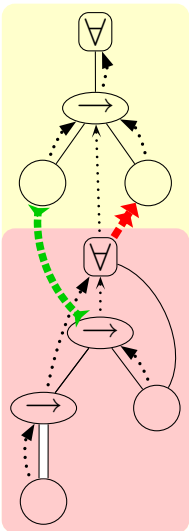
◀ back



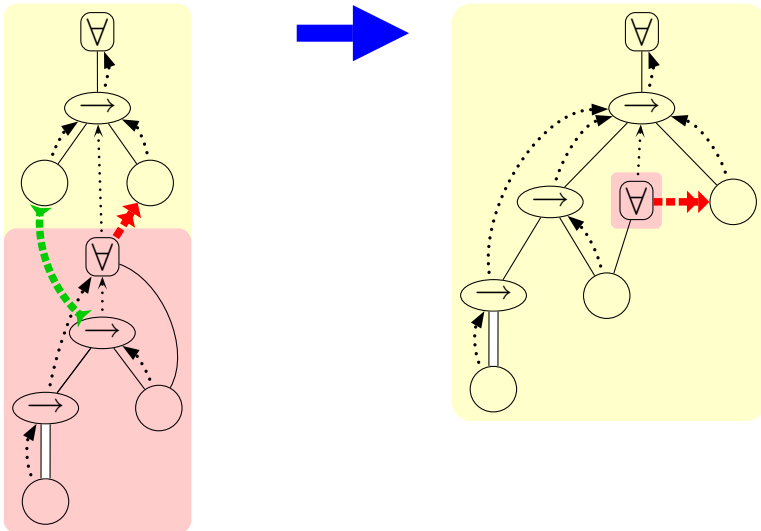
# Type inference with typing constraints (demo)

▶ skip

◀ back



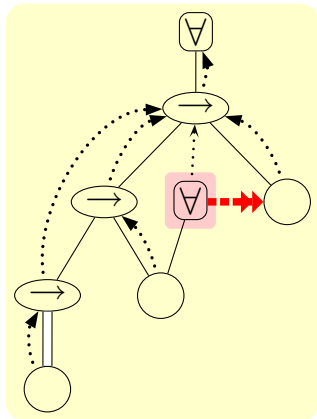
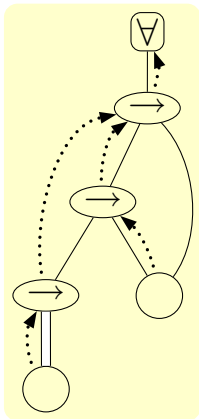
# Type inference with typing constraints (demo)



# Type inference with typing constraints (demo)

▶ skip

◀ back



## Type inference with typing constraints (demo)

▶ skip

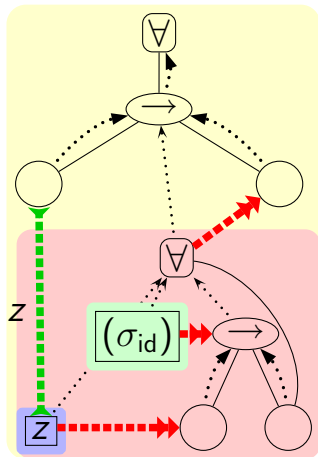
◀ back

 $\lambda(z) (z : \sigma_{\text{id}})$

# Type inference with typing constraints (demo)

▶ skip

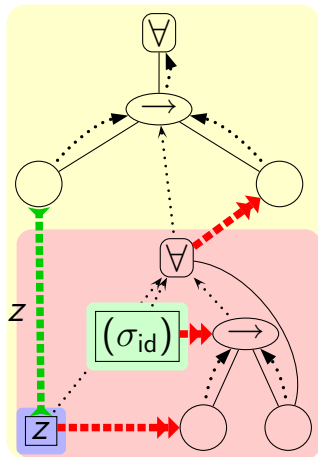
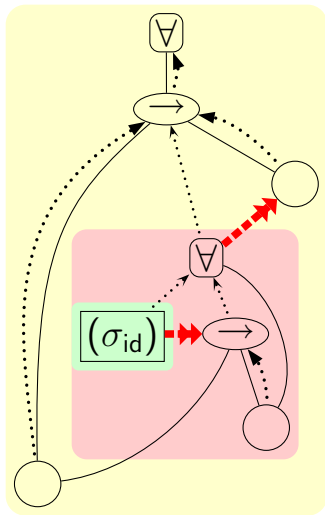
◀ back

$$\lambda(z) (z : \sigma_{id})$$


# Type inference with typing constraints (demo)

▶ skip

◀ back

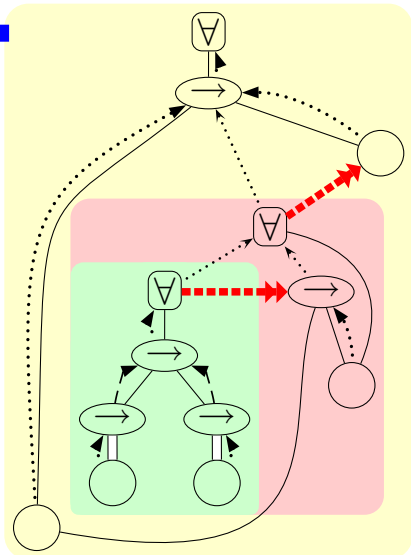
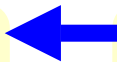
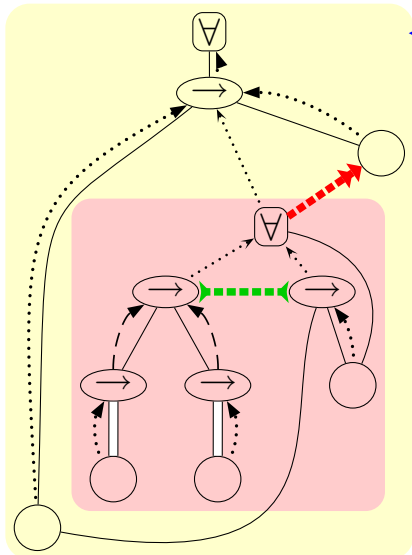




# Type inference with typing constraints (demo)

▶ skip

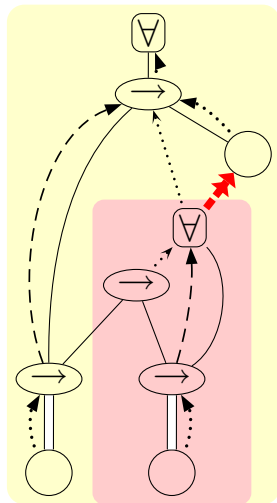
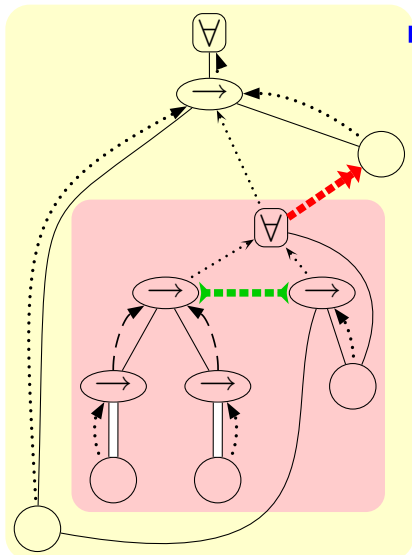
◀ back



# Type inference with typing constraints (demo)

▶ skip

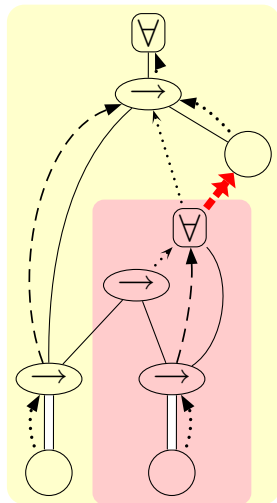
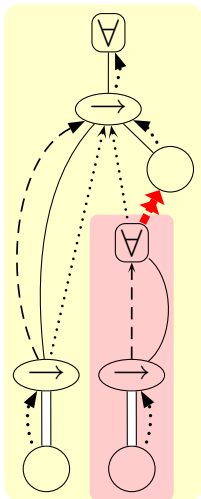
◀ back



# Type inference with typing constraints (demo)

▶ skip

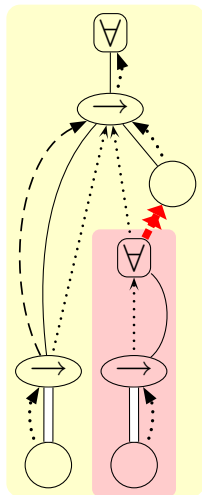
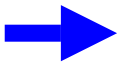
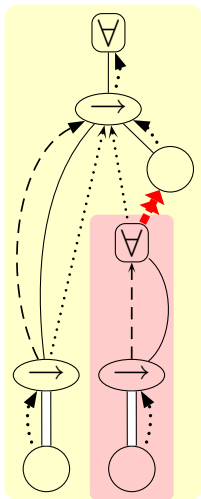
◀ back



# Type inference with typing constraints (demo)

▶ skip

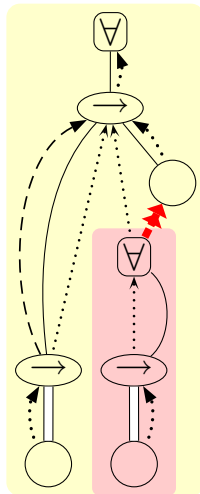
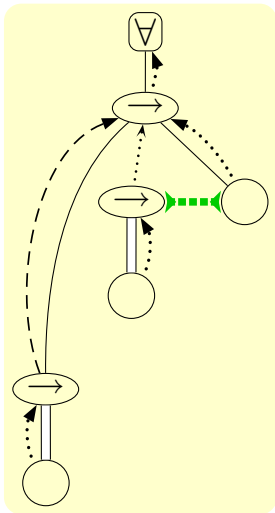
◀ back



# Type inference with typing constraints (demo)

▶ skip

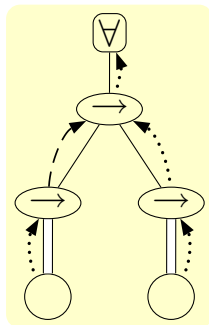
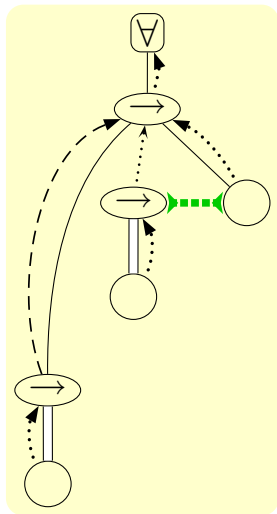
◀ back



# Type inference with typing constraints (demo)

▶ skip

◀ back





## More examples

Encoding of existential types, e.g.  $\exists\beta.\beta \times \beta \rightarrow \alpha$

**type**  $\alpha$  **func** =  $\forall(\gamma) \forall (\delta = \forall(\beta) \beta * (\beta \rightarrow \alpha) \rightarrow \gamma) \delta \rightarrow \gamma$

**val** **pack** **z** = **fun** (f :  $\exists(\gamma) \forall(\beta) \beta * (\beta \rightarrow \alpha) \rightarrow \gamma$ )  $\rightarrow$  f **z**;;

**val** **pack** :  $\forall(\alpha) \forall(\beta) \alpha * (\alpha \rightarrow \beta) \rightarrow (\forall(\gamma) (\forall(\delta) \delta * (\delta \rightarrow \beta) \rightarrow \gamma) \rightarrow \gamma)$

**let** **packed\_int** = **pack** (1, **fun** x  $\rightarrow$  x+1);;

**let** **packed\_pair** = **pack** (1, **fun** x  $\rightarrow$  (x, x));;

**let** **v** = **packed\_int** (**fun** p  $\rightarrow$  (snd p) (fst p));;

# About Rigid ML<sup>F</sup>

## Rigid ML<sup>F</sup> lies very close to ML<sup>F</sup>

- It uses and relies on (Shallow) ML<sup>F</sup> internally.
- It projects ML<sup>F</sup> principal types into System-F types at let-bindings, by raising variable bindings as much as possible.

## Rigid ML<sup>F</sup> loses important properties of ML<sup>F</sup>

- There are no principal types *per se*.
  - Rigid ML<sup>F</sup> pretends to have principal types, but this is in an ad hoc manner, using a non logical typing rule for Let-bindings with a premise that blocks free uses of type-instantiation.
- let  $x = \lambda(z : \sigma) z$  in  $a_2$  may be accepted while let  $x = \lambda(z) z$  in  $a_2$  would be rejected.
- Rigid ML<sup>F</sup> is not invariant by let-expansion (which signs the lost of truly principal types).

# About Rigid ML<sup>F</sup>

## Rigid ML<sup>F</sup> lies very close to ML<sup>F</sup>

- It uses and relies on (Shallow) ML<sup>F</sup> **internally**.
- It projects ML<sup>F</sup> principal types into System-F types at let-bindings, by raising variable bindings as much as possible.

## Rigid ML<sup>F</sup> loses important properties of ML<sup>F</sup>

- There are no principal types *per se*.
- Rigid ML<sup>F</sup> is not invariant by let-expansion (which signs the loss of truly principal types).

## Rigid ML<sup>F</sup> is a subset of System F

- This is both its **interest** and its **problem**.



# What would be an intermediate language for ML<sup>F</sup>?

## Problem

- Subject reduction is only proved in  $i\text{ML}^F$ , which has the same type erasure as  $e\text{ML}^F$ .
  - This ensures correctness of  $i\text{ML}^F$
  - But does not help to propagate annotations during reduction (or other program transformations)
- Even so,  $e\text{ML}^F$  requires type inference, which is not a local process.

## Solution

- Introduce a fully explicit version of  $x\text{ML}^F$  (easy)
- Instrument reduction rules to keep track of types during reduction (not entirely trivial)
- **This has to be investigated.**

# Sharing of abstract nodes is irreversible (implicitly)

[← back](#)

Can you show an example illustrating the difference?

**Fact:**  $\forall(\alpha \Rightarrow \sigma) \alpha \rightarrow \alpha \not\equiv \forall(\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha'$

Observe that:

- $\lambda(z) z : \forall(\alpha \Rightarrow \sigma) \alpha \rightarrow \alpha$
- $(\_ : \sigma) : \forall(\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma) \alpha \rightarrow \alpha'$

Then, the context  $a \triangleq \lambda(x) [] x x$  distinguishes those two expressions.

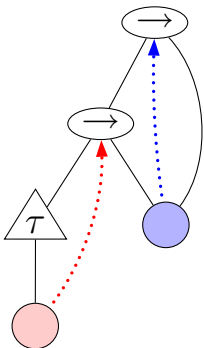
- $a[\lambda(z) z]$  is ill-typed.  
(As it uses no type annotation and it is ill-typed in ML)
- $a[(\_ : \sigma)]$  is well-typed.



## System-F types (encoding of existential types)

[← back](#)

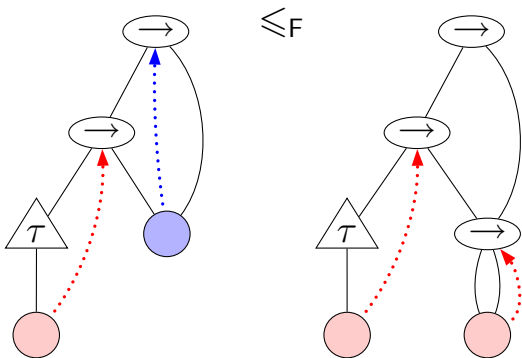
$$\forall(\alpha) (\forall(\beta) \tau_\beta \rightarrow \alpha) \rightarrow \alpha$$



# System-F types (encoding of existential types)

◀ back

$$\forall(\alpha) (\forall(\beta) \tau_\beta \rightarrow \alpha) \rightarrow \alpha$$

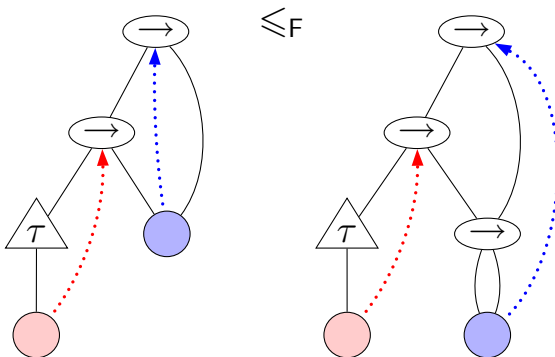


$$(\forall(\beta) \tau_\beta \rightarrow \forall(\alpha) \alpha \rightarrow \alpha) \rightarrow \forall(\alpha) \alpha \rightarrow \alpha$$

# System-F types (encoding of existential types)

◀ back

$$\forall(\alpha) (\forall(\beta) \tau_\beta \rightarrow \alpha) \rightarrow \alpha$$



$$\forall(\alpha) (\forall(\beta) \tau_\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$



## Type annotations

◀ back

$$\frac{\alpha \Rightarrow \sigma, \beta \Rightarrow \sigma \vdash \sigma \leq \alpha \text{ and } \sigma \leq \beta}{\alpha \Rightarrow \sigma, \beta \Rightarrow \sigma \vdash \begin{array}{l} \forall(\alpha' \Rightarrow \sigma) \forall(\beta' \Rightarrow \sigma) \alpha' \rightarrow \beta' \\ \leq \quad \forall(\alpha' \Rightarrow \alpha) \forall(\beta' \Rightarrow \beta) \alpha' \rightarrow \beta' \\ \diamond \\ \alpha \rightarrow \beta \end{array}}$$

$$\frac{\alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash (- : \sigma) : \alpha \rightarrow \beta \quad \alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash x : \alpha}{\frac{\alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash (x : \sigma) : \beta}{\alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \forall(\beta \Rightarrow \sigma) \beta}}$$

$$\alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \sigma$$

## Type annotations

◀ back

$$\frac{\alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash (x : \sigma_{\text{id}}) : \sigma_{\text{id}}}{\alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash (x : \sigma_{\text{id}}) : \alpha \rightarrow \alpha} \quad \alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash x : \alpha$$


---


$$\frac{\alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash (x : \sigma_{\text{id}}) x : \alpha}{\alpha \Rightarrow \sigma_{\text{id}} \vdash \lambda(x) (x : \sigma_{\text{id}}) x : \alpha \rightarrow \alpha}$$


---


$$\vdash \lambda(x) (x : \sigma_{\text{id}}) x : \forall(\alpha \Rightarrow \sigma_{\text{id}}) \alpha \rightarrow \alpha$$