1. Design
   - *i*ML^F: an implicity-typed extension of System F
   - Types explained
   - *e*ML^F: an explicitly-typed version of *i*ML^F

2. Uses and Implementation
   - Examples
   - Type inference
   - Restrictions and extensions

ML^F for Everyone

(Users, Implementers, and Designers)

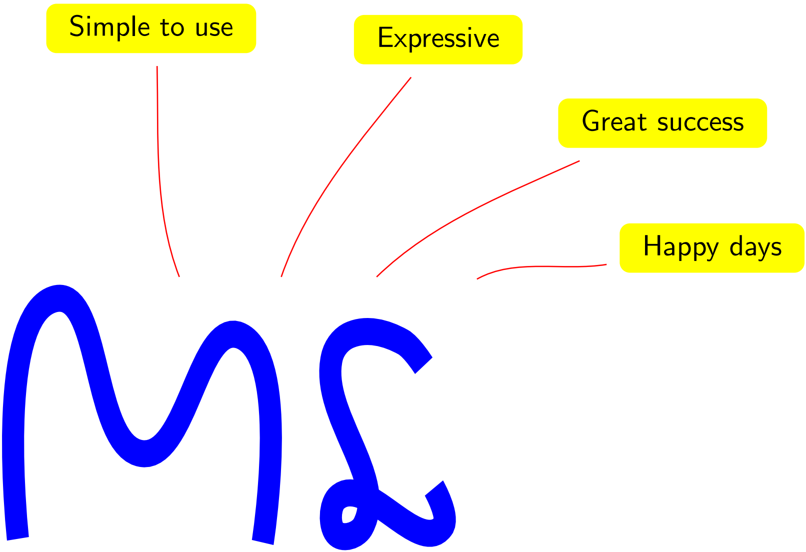
Didier Rémy

INRIA-Rocquencourt

ML Workshop

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

Ms


Simple to use
Expressive
Great success
Happy days

F

ML

F

Ms

Expressive

Simple ~~extension~~ of ML
Simplification

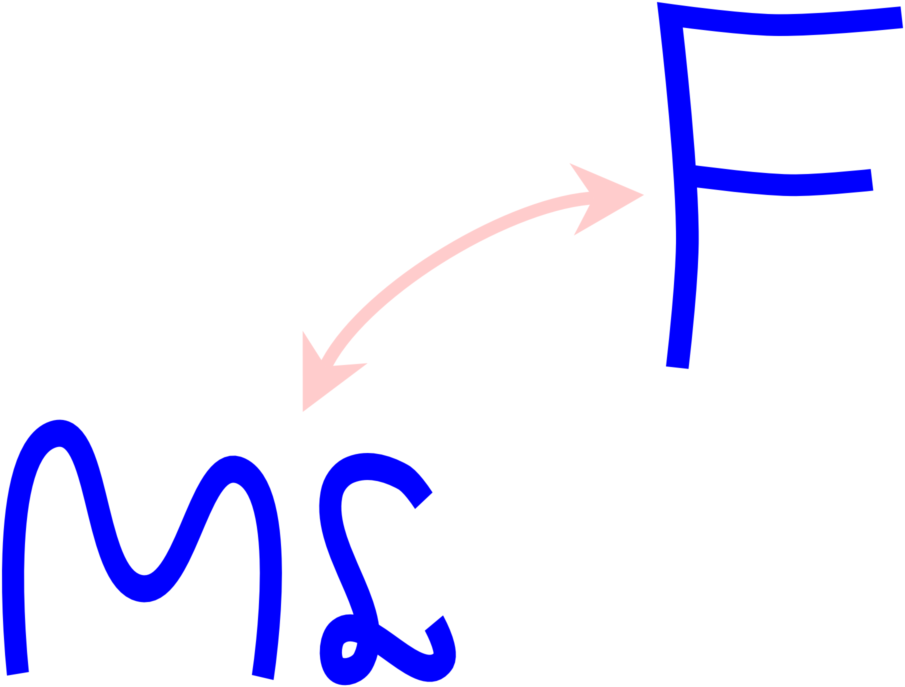
Even used in full scale languages such as Scala.

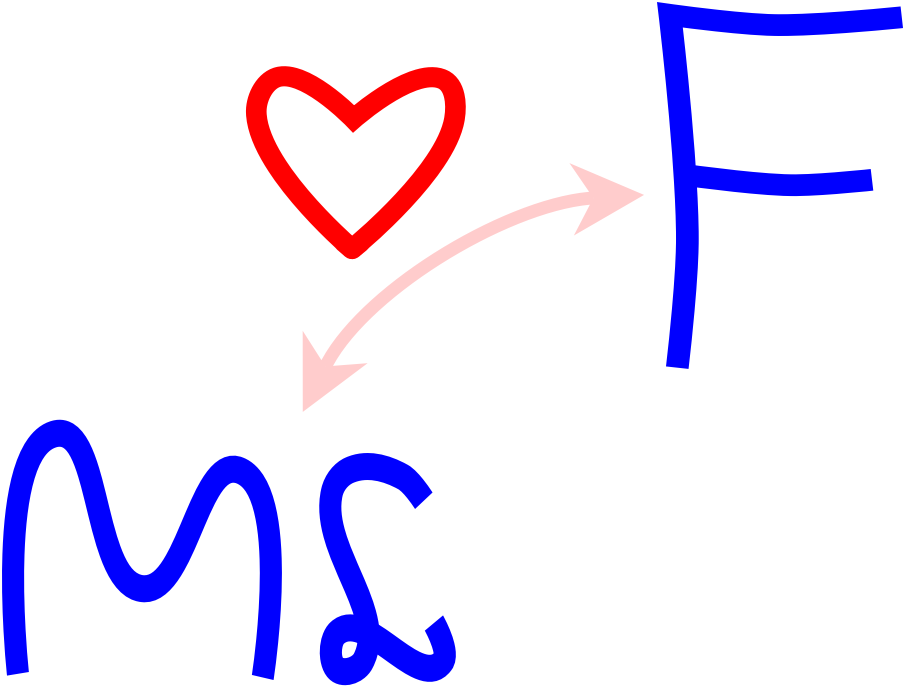
Full type inference is undecidable

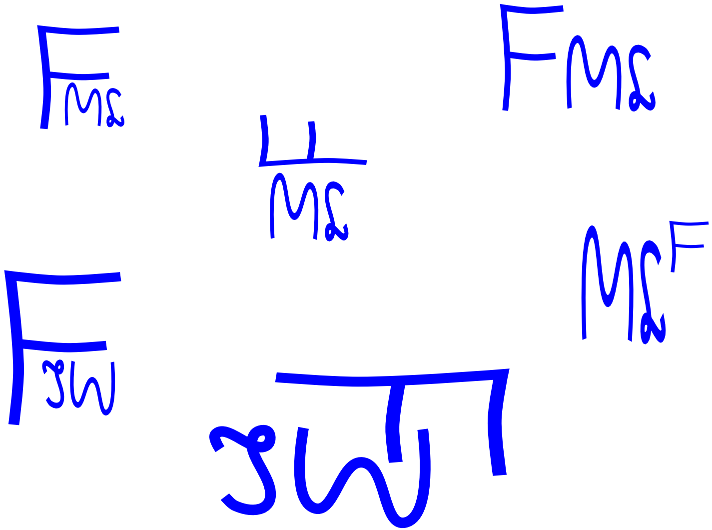
F

Full type annotations are obfuscating

ML


F


F



M&F

Outline

1. Design
   - *i*ML^F: an implicity-typed extension of System F
   - Types explained
   - *e*ML^F: an explicitly-typed version of *i*ML^F
2. Uses and Implementation
   - Examples
   - Type inference
   - Restrictions and extensions

A universal type system

Explicit System F:

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

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

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

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

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

A universal type system

Implicit System F:

$$\textsc{Var}\ \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1} \qquad \textsc{Fun}\ \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x\quad)\ a : \tau_0 \rightarrow \tau}$$

$$\textsc{Gen}\ \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash \qquad a : \forall(\alpha)\ \tau_0} \qquad \textsc{Ungen}\ \frac{\Gamma \vdash a : \forall(\alpha)\ \tau}{\Gamma \vdash a \quad : \tau_0[\alpha \leftarrow \tau]}$$

A universal type system

Implicit System F:

$$\textsc{Var}\quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau} \qquad \textsc{App}\quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1} \qquad \textsc{Fun}\quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x \quad)\ a : \tau_0 \rightarrow \tau}$$

$$\textsc{Gen}\quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash \quad a : \forall(\alpha)\ \tau_0} \qquad \textsc{Inst}\quad \frac{}{\forall(\bar\alpha)\ \tau_0 \leqslant \tau_0[\bar\alpha \leftarrow \bar\tau]} \qquad \textsc{Sub}\quad \frac{\Gamma \vdash a : \tau_1 \qquad \tau_1 \leqslant \tau_2}{\Gamma \vdash a : \tau_2}$$

A universal type system

Implicit System F:

$$\text{VAR}\quad \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau} \qquad \text{APP}\quad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1} \qquad \text{FUN}\quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x\quad)\ a : \tau_0 \to \tau}$$

$$\text{GEN}\quad \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash \quad a : \forall(\alpha)\ \tau_0} \qquad \text{INST}\quad \frac{\bar\beta \notin \mathrm{ftv}(\forall(\bar\alpha)\ \bar\tau_0)}{\forall(\bar\alpha)\ \tau_0 \leqslant \forall(\bar\beta)\ \tau_0[\bar\alpha \leftarrow \bar\tau]} \qquad \text{SUB}\quad \frac{\Gamma \vdash a : \tau_1 \qquad \tau_1 \leqslant \tau_2}{\Gamma \vdash a : \tau_2}$$

A universal type system

Implicit System F:

$$\textsc{Var}\ \frac{z : \tau \in \Gamma}{\Gamma \vdash z : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1} \qquad \textsc{Fun}\ \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda(x \qquad)\ a : \tau_0 \to \tau}$$

$$\textsc{Gen}\ \frac{\Gamma, \alpha \vdash a : \tau_0}{\Gamma \vdash \qquad a : \forall(\alpha)\ \tau_0} \qquad \textsc{Inst}\ \frac{\bar\beta \notin \mathrm{ftv}(\forall(\bar\alpha)\ \bar\tau_0)}{\forall(\bar\alpha)\ \tau_0 \leqslant \forall(\bar\beta)\ \tau_0[\bar\alpha \leftarrow \bar\tau]} \qquad \textsc{Sub}\ \frac{\Gamma \vdash a : \tau_1 \qquad \tau_1 \leqslant \tau_2}{\Gamma \vdash a : \tau_2}$$

Add a construction for local bindings (perhaps derivable):

$$\textsc{Let}\ \frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash a_2 : \tau}{\Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : \tau}$$

A universal type system

Implicit System F:

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

APP
$$\Gamma \vdash$$

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

Logical, canonical presentation of typing rules

- Covers many variations: F, ML, F^η, $F_\leq$, ...
  - 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):

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

A universal type system

Implicit System F:

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

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

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

Logical, canonical presentation of typing rules

- Covers many variations: F, ML, F^η, $F_\leq$, ...
  - 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):

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

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 two many bothersome type annotations.

Not conservative extensions of ML

Are polymorphic types less important than monorphic ones?

Type inference is undecidable — in System F

Of course, we must

- Use type annotations on function parameters in some cases.

When?

Not conservative extensions of ML

- 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 two many bothersome type annotations.

Are polymorphic types less important than monorphic ones?

Our choice

explained below

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

Lack of principal types for applications

The example of choice

let $\mathit{choice} = \lambda(x)\ \lambda(y)$ **if** true **then** x **else** $y : \forall\, \beta \cdot \beta \to \beta \to \beta$
let $\mathit{id} = \lambda(z)\ z : \forall(\alpha)\ \alpha \to \alpha$

$\mathit{choice}\ \mathit{id}$:

Lack of principal types for applications

The example of choice

let $choice = \lambda(x)\ \lambda(y)$ **if** *true* **then** x **else** $y : \forall\, \beta \cdot \beta \to \beta \to \beta$
let $id = \lambda(z)\ z : \forall(\alpha)\ \alpha \to \alpha$

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

Lack of principal types for applications

The example of choice

let $\mathit{choice} = \lambda(x)\ \lambda(y)$ **if** true **then** x **else** $y : \forall\, \beta \cdot \beta \rightarrow \beta \rightarrow \beta$
let $\mathit{id} = \lambda(z)\ z : \forall(\alpha)\ \alpha \rightarrow \alpha$

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

No better choice in F!

Lack of principal types for applications

The example of choice

let $\mathit{choice} = \lambda(x)\ \lambda(y)$ **if** true **then** x **else** $y : \forall\, \beta \cdot \beta \to \beta \to \beta$

let $\mathit{id} = \lambda(z)\ z : \forall(\alpha)\ \alpha \to \alpha$

$$\mathit{choice}\ \mathit{id} : \begin{cases} \forall(\alpha)\ (\alpha \to \alpha) \to (\alpha \to \alpha) \\ (\forall(\alpha)\ \alpha \to \alpha) \to (\forall(\alpha)\ \alpha \to \alpha) \end{cases}$$

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** $y : \forall\, \beta \cdot \beta \to \beta \to \beta$
let *id* $= \lambda(z)\ z : \forall(\alpha)\ \alpha \to \alpha$

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

The solution in iML^F:

$$choice\ id : \quad \forall(\beta \geq \forall(\alpha)\ \alpha \to \alpha)\ \beta \to \beta$$

Lack of principal types for applications

The example of choice

let *choice* $= \lambda(x)\ \lambda(y)$ **if** *true* **then** x **else** $y : \forall\,\beta \cdot \beta \to \beta \to \beta$
let *id* $= \lambda(z)\ z : \forall(\alpha)\ \alpha \to \alpha$

$$\textit{choice id} : \left\{ \begin{array}{l} \forall(\alpha)\ (\alpha \to \alpha) \to (\alpha \to \alpha) \\ (\forall(\alpha)\ \alpha \to \alpha) \to (\forall(\alpha)\ \alpha \to \alpha) \end{array} \right.$$

The solution in iML^F:

$$\textit{choice id} : \quad {\color{red}\forall(\beta \geq} \forall(\alpha)\ \alpha \to \alpha{\color{red})\ \beta \to \beta}$$

$$\leqslant \left\{ \begin{array}{l} {\color{red}(\beta \to \beta)}\ [{\color{red}\beta} \leftarrow \forall(\alpha)\ \alpha \to \alpha] \\ \forall(\alpha)\ {\color{red}(\beta \to \beta)}\ [{\color{red}\beta} \leftarrow \alpha \to \alpha]) \end{array} \right.$$

The definition of *i*ML^F

Types are stratified

$$\begin{array}{rll} \sigma ::= & \tau & \in \mathsf{F} \\ \mid & \forall(\alpha \geq \sigma)\ \sigma & \end{array}$$

We can see and explain types by $\leqslant_{\mathsf{F}}$-closed sets of System-F types:

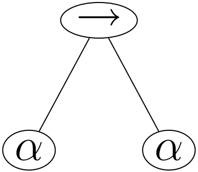
$$\begin{array}{rl} \{\!\{\tau\}\!\} & \triangleq \{\tau' \mid \tau \leqslant_{\mathsf{F}} \tau'\} \\ \{\!\{\forall(\alpha \geq \sigma)\ \sigma'\}\!\} & \triangleq \left\{ \forall(\bar{\beta})\ \tau'[\alpha \leftarrow \tau] \mid \wedge \left(\begin{array}{l} \tau \in \{\!\{\sigma\}\!\} \wedge \tau' \in \{\!\{\sigma'\}\!\} \\ \bar{\beta} \# \mathsf{ftv}(\forall(\alpha \geq \sigma)\ \sigma') \end{array} \right. \right\} \end{array}$$

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

$$\sigma \leqslant_I \sigma' \iff \{\!\{\sigma\}\!\} \supseteq \{\!\{\sigma'\}\!\}$$

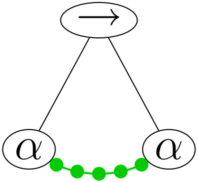
Simple types

$$\alpha \to \alpha$$



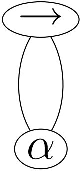
Simple types

$$\alpha \to \alpha$$



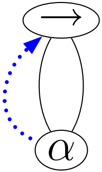
Simple types

$$\alpha \to \alpha$$



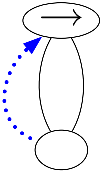
System-F types

$$\forall(\alpha)\ \alpha \to \alpha$$



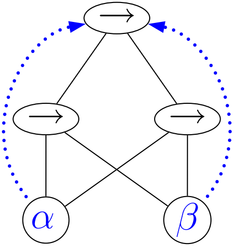
System-F types

$$\forall(\alpha)\ \alpha \to \alpha$$



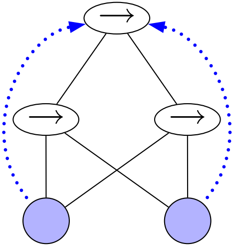
System-F types

$$\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to \alpha \to \beta$$



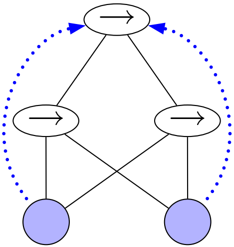
System-F types

$$\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to \alpha \to \beta$$

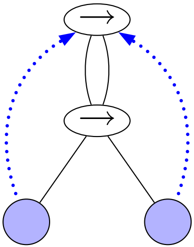


System-F types

$$\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to \alpha \to \beta$$

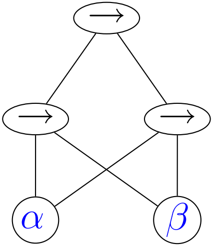


$\lessdot_F$

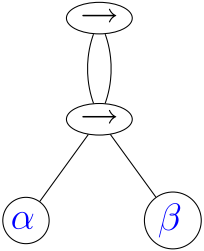


System-F types

$$(\alpha \to \beta) \to \alpha \to \beta$$



$\sqsupseteq ST$

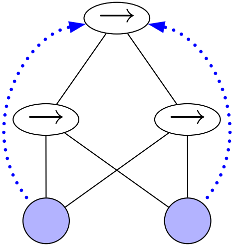


Sharing of inner nodes:

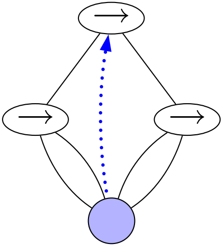
- Coming from the dag-representation of simple types.
- Canonical (unique) representation if disallowed.

System-F types

$$\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to \alpha \to \beta$$



$\leqslant_F$

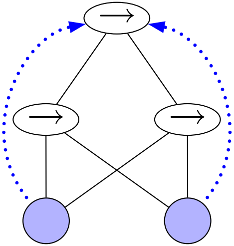


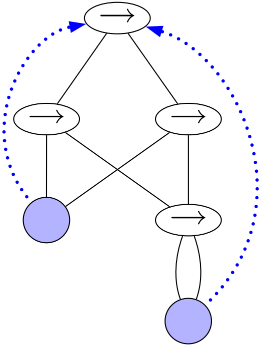
$$\forall(\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha$$

System-F types

$$\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to \alpha \to \beta$$

$\leqslant_F$



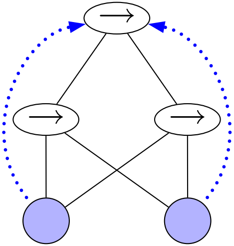


$$\forall(\alpha)\ \forall(\gamma)\ (\alpha \to \gamma \to \gamma) \to \alpha \to \gamma \to \gamma$$

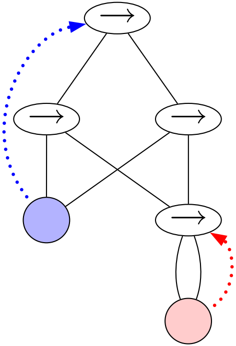
System-F types

more

$$\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to \alpha \to \beta$$



$\leqslant_F$



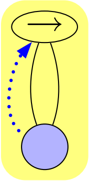
$$\forall(\alpha)(\alpha \to \forall(\gamma)\ \gamma \to \gamma) \to \alpha \to \forall(\gamma)\ \gamma \to \gamma$$

Types in $i\mathsf{ML}^{\mathsf{F}}$

$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha\,) \to \beta \to \beta$$

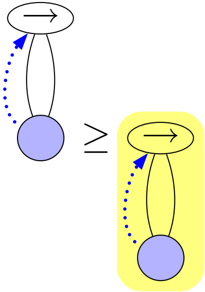
Types in iML^F

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



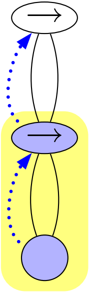
Types in iML^F

$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha) \to \beta \to \beta$$



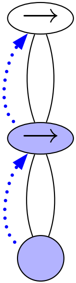
Types in iML^F

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



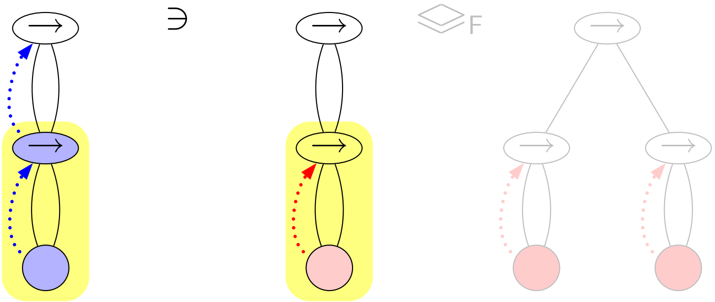
Types in iML^F

$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha\,) \to \beta \to \beta$$



Types in iML^F

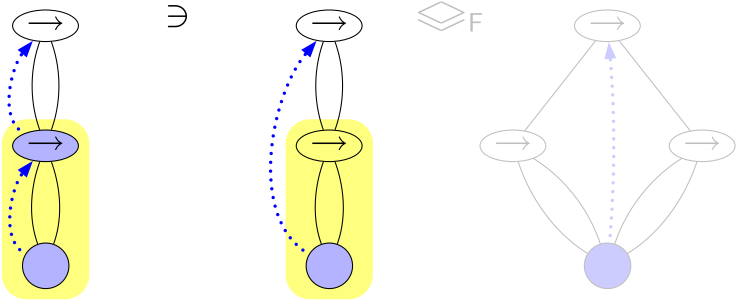
$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha)\to \beta \to \beta$$



$$(\forall(\alpha)\, \alpha \to \alpha) \to \forall(\alpha)\, \alpha \to \alpha$$

Types in iML^F

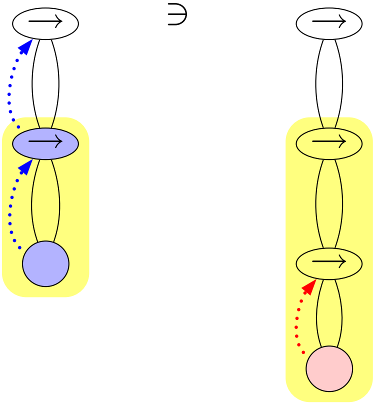
$$\forall(\beta \geq \forall(\alpha)\ \alpha \to \alpha\ ) \to \beta \to \beta$$



$$\forall(\alpha)\ (\alpha \to \alpha) \to (\alpha \to \alpha)$$

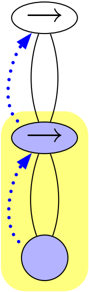
Types in iML^F

$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha) \to \beta \to \beta$$

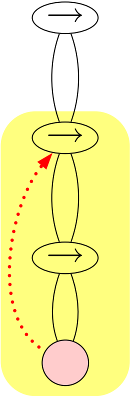


Types in iML^F

$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha\,) \to \beta \to \beta$$

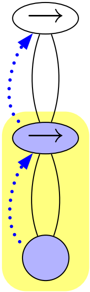


$\ni$

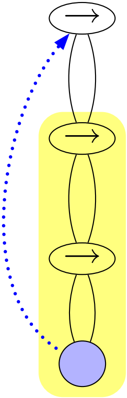


Types in iML^F

$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha\,) \to \beta \to \beta$$

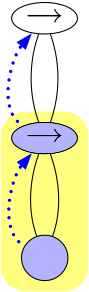


$\ni$



Types in iML^F

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



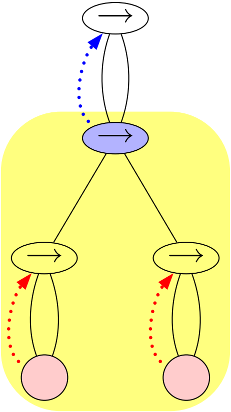
$\ni$ . . .

The semantics cannot be captured by

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

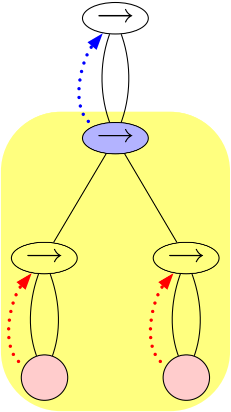
iML^F types

$$\forall(\beta \geq (\forall(\alpha)\, \alpha \to \alpha) \to (\forall(\alpha)\, \alpha \to \alpha)) \to \beta \to \beta$$



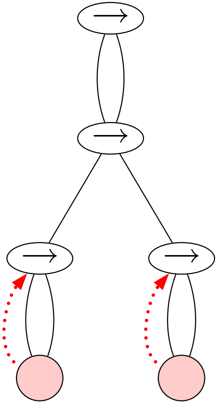
iML^F types

$$\forall(\beta \geq (\forall(\alpha)\, \alpha \to \alpha) \to (\forall(\alpha)\, \alpha \to \alpha)) \to \beta \to \beta$$



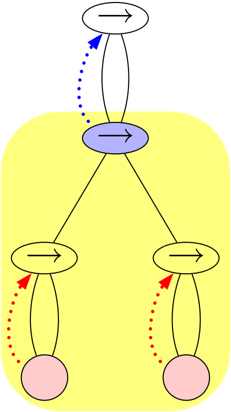
$\ni$

only

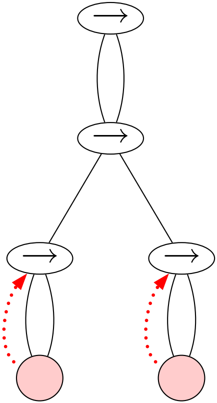


iML^F types

$$\forall(\beta \geq (\forall(\alpha)\, \alpha \to \alpha) \to (\forall(\alpha)\, \alpha \to \alpha)) \to \beta \to \beta$$

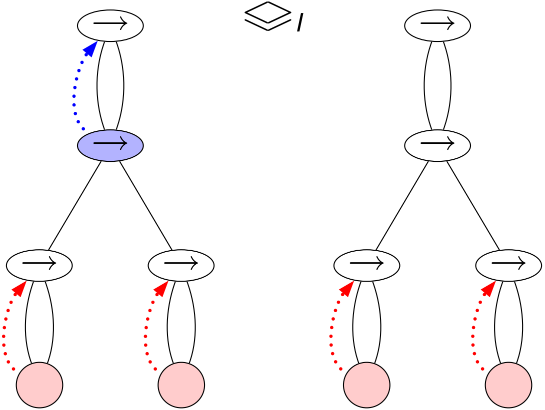


$\sqsubseteq_I$



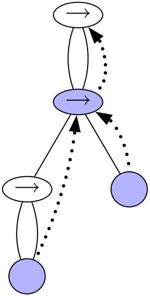
iML^F types

$$\forall(\beta \geq (\forall(\alpha)\,\alpha \to \alpha) \to (\forall(\alpha)\,\alpha \to \alpha))\to \beta \to \beta$$



Type instance $\leqslant$ in iML^F

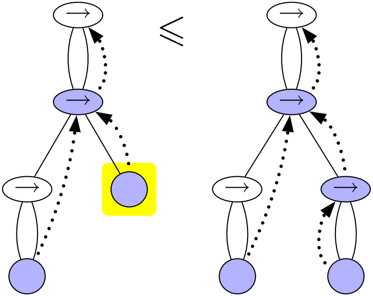
Only four atomic instance operations, and only two new.



Type instance $\leqslant$ in iML^F

Only four atomic instance operations, and only two new.

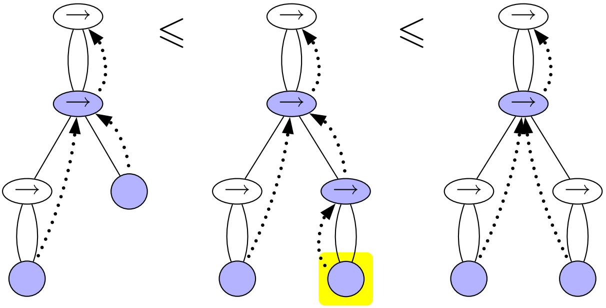
Grafting



Type instance $\leqslant$ in iML^F

Only four atomic instance operations, and only two new.

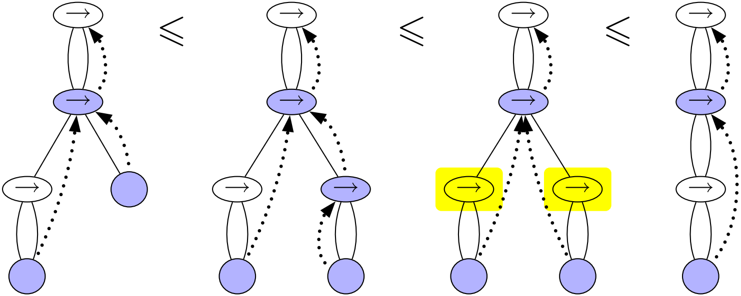
Grafting Raising



Type instance $\leqslant$ in $i\mathsf{ML}^\mathsf{F}$

Only four atomic instance operations, and only two new.

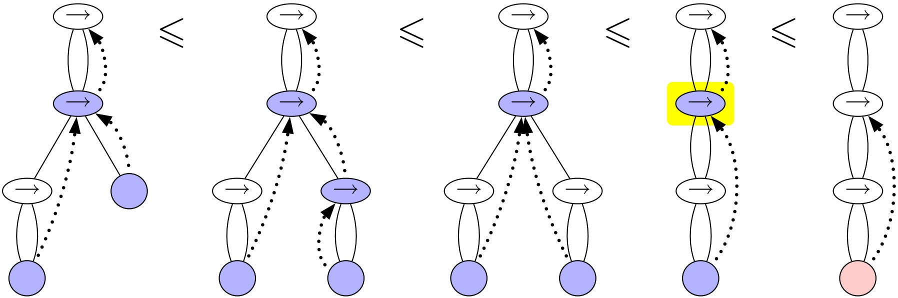
Grafting Raising Merging



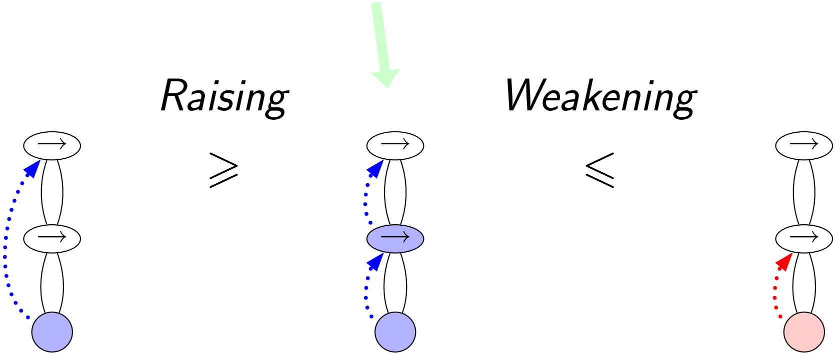
Type instance $\leqslant$ in iML^F

Only four atomic instance operations, and only two new.

Grafting Raising Merging Weakening



Checking the example choice id



Outline

1. Design
   - iML^F: an implicity-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 $e\mathrm{ML}^{\mathrm{F}}$

Goal

Find a restriction $i\mathrm{ML}^{\mathrm{F}}$ 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

$\lambda(z)\ z$	:	$\forall(\alpha)\ \alpha \to \alpha$	as in ML
let $x = \lambda(z)\ z$ in $x\ x$	:	$\forall(\alpha)\ \alpha \to \alpha$	as in ML
$\lambda(x)\ x\ x$	:	ill-typed!	x is used polymorphically
$\lambda(x : \forall(\alpha)\ \alpha \to \alpha)\ x\ x$	:	$(\forall(\alpha)\ \alpha \to \alpha) \to (\forall(\alpha)\ \alpha \to \alpha)$	

First-order inference of second order types

More challenging example

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

First-order inference of second order types

More challenging example

$$(\lambda(z)\ z)\ \ (a : \sigma) \quad \text{where } \sigma \text{ is truly polymorphic}$$

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

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

First-order inference of second order types

More challenging example

$(\lambda(z)\ z)\ \ (a : \sigma)$ where σ 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 n

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

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 $\sigma \to \sigma$?
- 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 $\sigma \to \sigma$?
- 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 $\sigma_{id} \to \sigma_{id}$
2) Instead, allow type variables to stand for polymorphic types:

write	$\forall(\alpha \Rightarrow \sigma_{id})\ \alpha \to \alpha$
read	"$\alpha \to \alpha$ where α abstracts σ_{id}"
means	$\sigma_{id} \to \sigma_{id}$

Mechanism

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

$$\cfrac{\cfrac{\alpha \Rightarrow \sigma_{id}, x : \alpha \vdash x : \alpha}{\alpha \Rightarrow \sigma_{id} \vdash \lambda(x)\ x : \alpha \to \alpha}}{\lambda(x)\ x : \forall(\alpha \Rightarrow \sigma_{id})\ \alpha \to \alpha}$$

Abstracting second-order polymorphism

more

Key point: abstraction is directional

$$\alpha \Rightarrow \sigma \vdash \sigma \leqslant \alpha \qquad\qquad \alpha \Rightarrow \cancel{\sigma \vdash} \alpha \leqslant \sigma$$

Hence,

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash a : \sigma \qquad \cdot}{\alpha \Rightarrow \sigma \vdash a : \alpha} \qquad \alpha \Rightarrow \sigma,\ z : \alpha \to \alpha \vdash z : \alpha \to \alpha}{\alpha \Rightarrow \sigma,\ z : \alpha \to \alpha \vdash z\ a : \alpha}}{\alpha \Rightarrow \sigma \vdash \lambda(z)\ z\ a : (\alpha \to \alpha) \to \alpha}}{\vdash \lambda(z)\ z\ a : \forall(\alpha \Rightarrow \sigma)\ (\alpha \to \alpha) \to \alpha}$$

Abstracting second-order polymorphism

more

Key point: abstraction is directional

$$\alpha \Rightarrow \sigma \vdash \sigma \leqslant \alpha \qquad\qquad \cancel{\alpha \Rightarrow \sigma \vdash \alpha \leqslant \sigma}$$

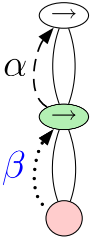
But,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha \Rightarrow \sigma_{\mathsf{id}},\ z : \alpha \vdash z : \alpha}{\alpha \Rightarrow \sigma_{\mathsf{id}},\ z : \alpha \vdash z : \sigma_{\mathsf{id}}}}{\alpha \Rightarrow \sigma_{\mathsf{id}},\ z : \alpha \vdash z : \alpha \to \alpha} \qquad \alpha \Rightarrow \sigma_{\mathsf{id}},\ z : \alpha \vdash z : \alpha}{\alpha \Rightarrow \sigma_{\mathsf{id}},\ z : \alpha \vdash z\ z : \alpha}}{\alpha \Rightarrow \sigma_{\mathsf{id}} \vdash \lambda(z)\ z\ z : \alpha \to \alpha}}{\vdash \lambda(z)\ z\ z : \forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha}$$

Types in eML^F

Introduce a new binder for abstraction

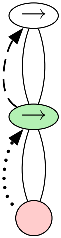
$$\forall(\alpha \Rightarrow \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha$$



Types in eML^F

Introduce a new binder for abstraction

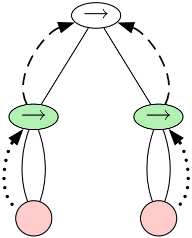
$$\forall(\alpha \Rightarrow \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha$$



Types in eML^F

Introduce a new binder for abstraction

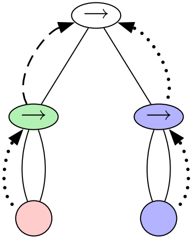
$$\forall(\alpha \Rightarrow \forall(\beta)\ \beta \to \beta)\ \forall(\alpha' \Rightarrow \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha'$$



Types in eML^F

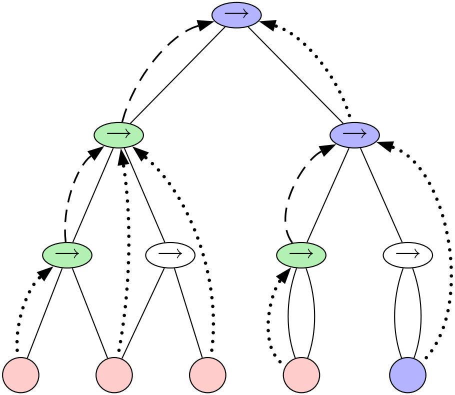
Introduce a new binder for abstraction

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



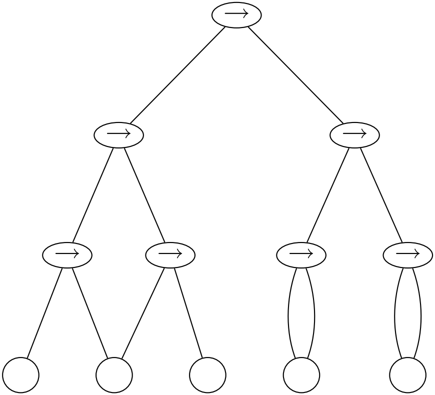
Types, graphically

= first-order term-dag + a binding tree



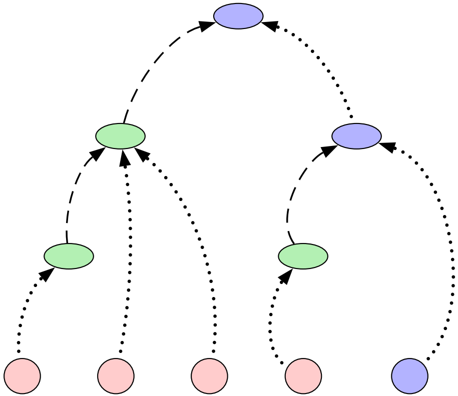
Types, graphically

$=$ first-order term-dag $+$ a binding tree



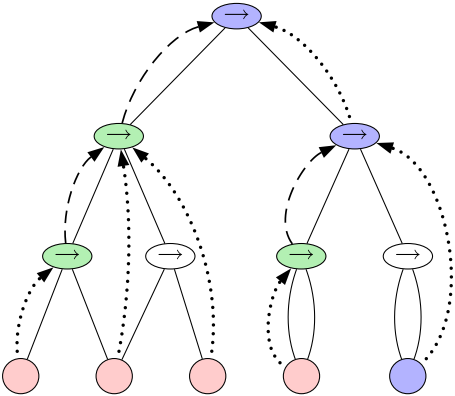
Types, graphically

= first-order term-dag + a binding tree



Types, graphically

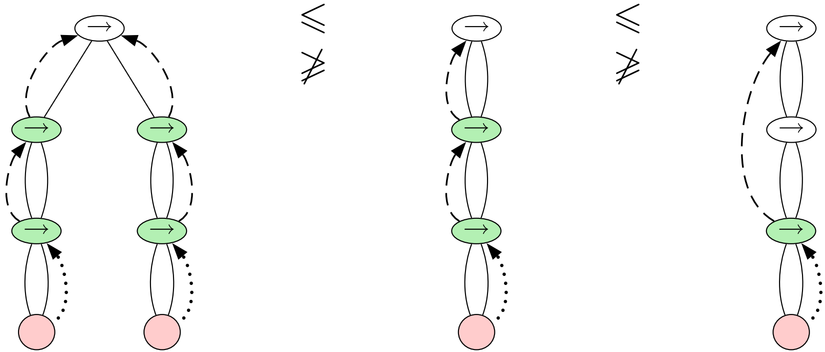
= first-order term-dag + a binding tree



+ well-formedness conditions relating the two

Type instance $\leqslant$ in eML^F

Sharing and binding of abstract nodes matter



Grafting, Merging, Raising, Weakening

Unchanged.

Type annotations

Recovering the missing power

$$(\leqslant) \subset (\leqslant_I)$$

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

Type annotations

Recovering the missing power

$$(\leqslant) \subset (\leqslant_I) = (\leqslant \cup \mathrel{\diamond}_I)^* = (\leqslant ; \mathrel{\diamond}_I)$$

- $\leqslant$ is weaker than $\leqslant_I$, as sharing and binding of abstract nodes matters.
- Use explicit type annotations to recover $(\mathrel{\diamond}_I \setminus \leqslant)$.

  Notice that the weaker $\leqslant$, the more annotations will be required.

Type annotations

Recovering the missing power

$$(\leqslant) \subset (\leqslant_I) = (\leqslant \cup \otimes_I)^* = (\leqslant ; \otimes_I)$$

- Intuitively,

$$\frac{\Gamma \vdash a : \tau \qquad \tau \otimes_I \tau'}{\Gamma \vdash (a : \tau') : \tau'}$$

- Actually, use coercion functions:

$$(\_ : \quad \sigma) \;:\; \forall(\alpha \Rightarrow \sigma)\, \forall(\alpha' \Rightarrow \sigma)\, \alpha \to \alpha'$$

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

Type annotations

Recovering the missing power

$$(\leqslant) \subset (\leqslant_I) = (\leqslant \cup \mathrel{\diamondsuit}_I)^* = (\leqslant ; \mathrel{\diamondsuit}_I)$$

- Intuitively,

$$\frac{\Gamma \vdash a : \tau \qquad \tau \mathrel{\diamondsuit}_I \tau'}{\Gamma \vdash (a : \tau') : \tau'}$$

- Actually, use coercion functions:

$$(\_ : \exists\,(\bar\beta)\;\sigma) \;:\; \forall(\bar\beta)\,\forall(\alpha \Rightarrow \sigma)\,\forall(\alpha' \Rightarrow \sigma)\;\alpha \to \alpha'$$

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

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 $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$

Outline

1. Design
   - iML^F: an implicity-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

- Programmmers must be free of choising 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

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

Common subexpression can be factored out.

Redefine application

$a_1\ a_2 \;\subseteq\!\!\supseteq\; (\lambda(f)\ \lambda(x)\ f\ x)\ a_1\ a_2$

Many functionals, such as maps are typed as applications.

η-conversion of functional expressions

$a \;\subseteq\!\!\supseteq\; \lambda(x)\ a\ x$

Delay the evaluation.

Reordering of arguments

$a\ a_1\ a_2 \;\subseteq\!\!\supseteq\; (\lambda(x)\ \lambda(y)\ a\ y\ x)\ a_2\ a_1$

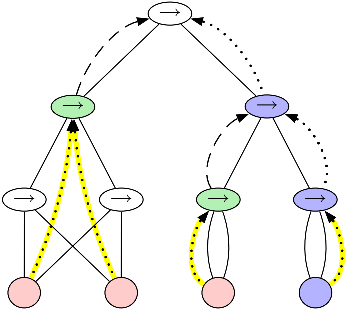
Curryfication

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

All valid in ML^F

Printing types

Only overlined bindings need to be drawn



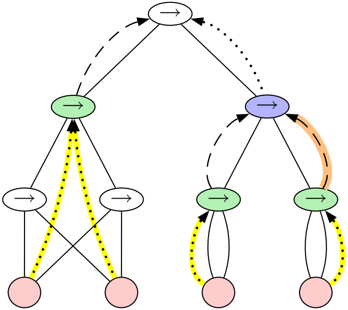
Leave implicit bindings that are

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

$$(\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)) \to (\forall(\alpha)\ \alpha \to \alpha) \to (\forall(\alpha)\ \alpha \to \alpha)$$

Printing types

Only overlined bindings need to be drawn



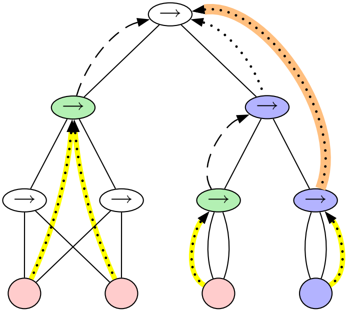
Leave implicit bindings that are

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

$$(\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)) \to \forall(\gamma \Rightarrow \forall(\alpha)\ \alpha \to \alpha)\ (\forall(\alpha)\ \alpha \to \alpha) \to \gamma$$

Printing types

Only overlined bindings need to be drawn



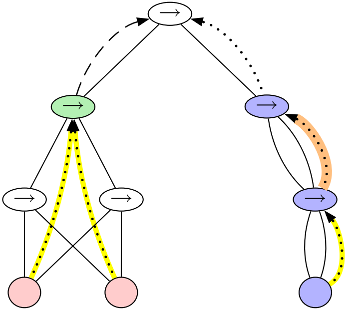
Leave implicit bindings that are

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

$$\forall(\gamma \Rightarrow \forall(\alpha)\ \alpha \to \alpha)\ (\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)) \to (\forall(\alpha)\ \alpha \to \alpha) \to \gamma$$

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.

$$(\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)) \to \forall(\gamma \geq \sigma_{\mathsf{id}})\ \gamma \to \gamma$$

Examples

Library functions

```
let rec fold f v = function
  | Nil → v
  | Cons (h, t) → fold f (f h t) t ;;
```

val fold : $\forall(\alpha)\,\forall(\beta)\,(\alpha \to \alpha\ \mathit{list} \to \beta) \to \beta \to \alpha\ \mathit{list} \to \beta$

Few type annotations are needed in practice

- No dummmy/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

skip

Church's numerals

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

With type annotations on the iterator

let succ (n : nat) = fun f x $\to$ n f (f x);;
val succ : nat $\to (\forall\ (\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha)$

let add (n : nat) m = n succ m;;
val add : nat $\to (\forall\ (\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha)$

let mul n (m : nat) = m (add n) zero;;
mul : nat $\to$ nat $\to (\forall(\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha)$

More examples

skip

Church's numerals

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

Without type annotations

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

In ML:

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

More examples

skip

Church's numerals

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

Mandatory type annotations

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

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 maximun 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.

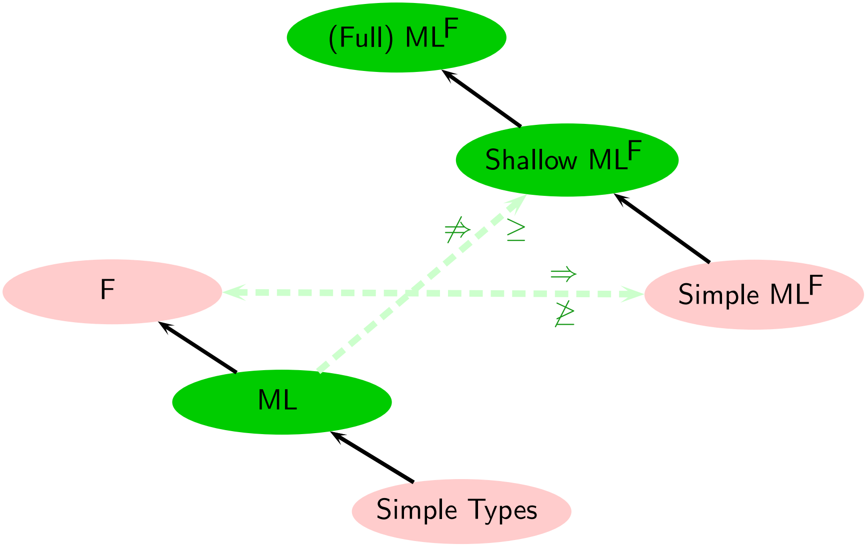
- Types are stratified.
- Instance bounded types cannot appear in bounds of abstract variables.
- In particular, type annotations must be F types.

Simple ML^F

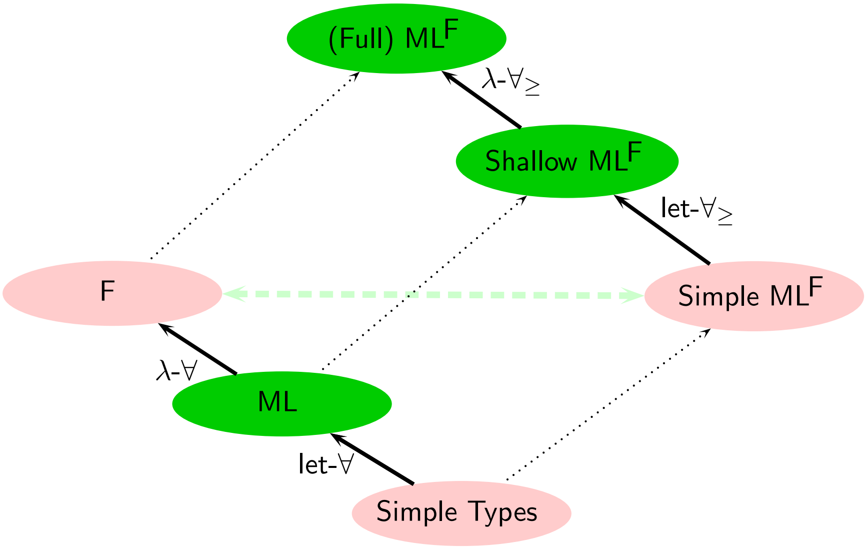
Remove instance bindings $\geq$, keep abstract bindings $\Rightarrow$.

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

A hierarchy of languages

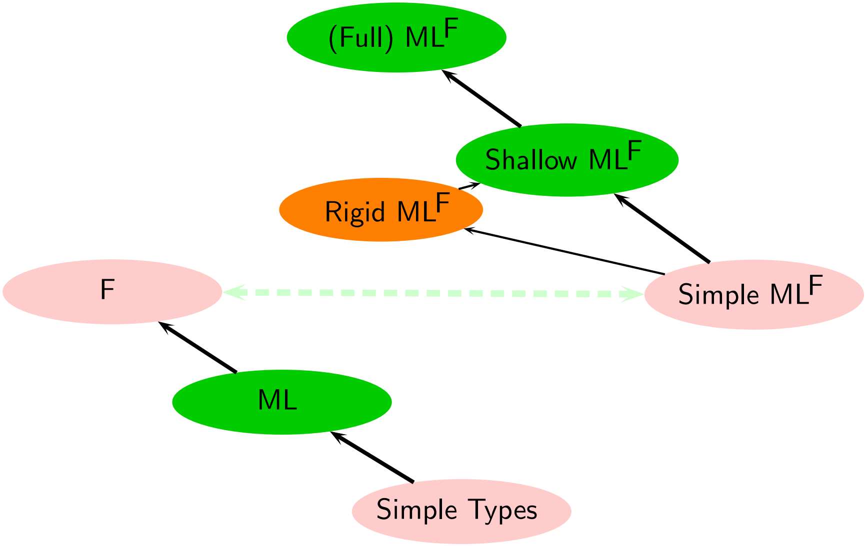


A hierarchy of languages



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 croses quantifiers are difficult.

Higher-order types

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

Papers and prototypes

Talk mainly based on

- Recasting-ML^F *with Didier Le Boltan*.
- A Graphical Presentation of ML^F 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 polymophically.

ML^{F} design, use, and implementation are close to ML

- ML^{F} piggy-backs on ML type-shemes 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.

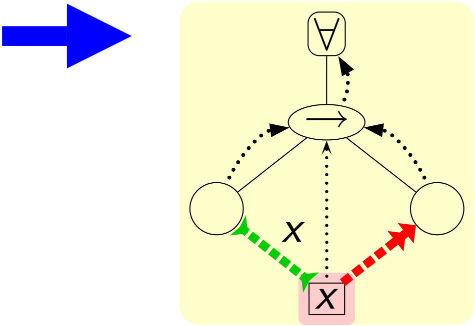
Appendix

Type inference with typing constraints (demo)

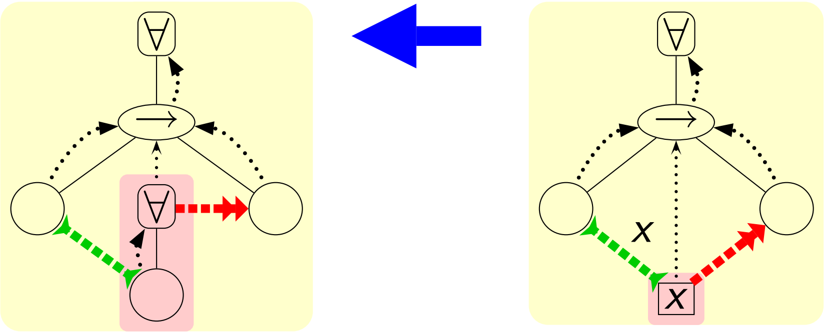
$$\boxed{\lambda(x)\ x}$$

Type inference with typing constraints (demo)

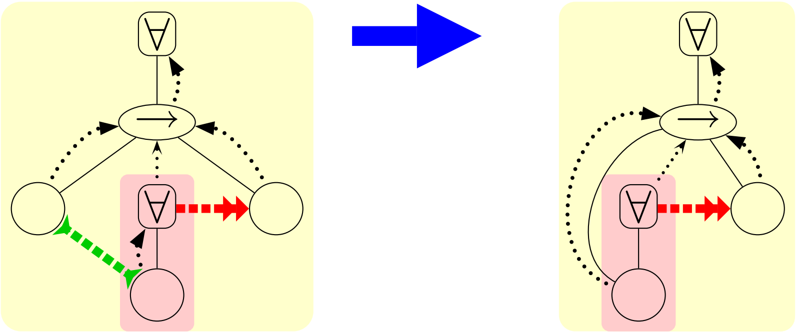
$\boxed{\lambda(x)\ x}$



Type inference with typing constraints (demo)

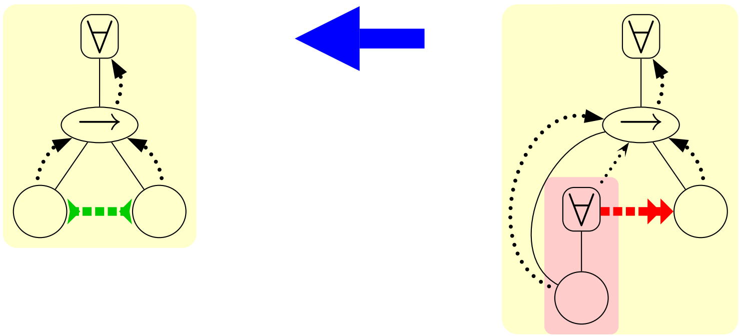


Type inference with typing constraints (demo)



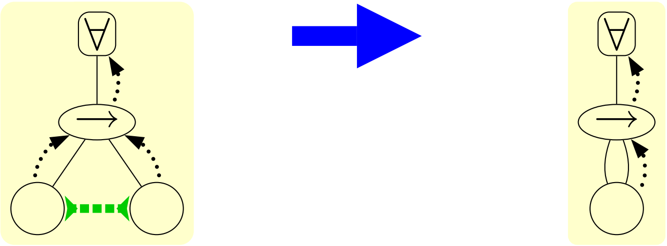
Type inference with typing constraints (demo)

skip back



Type inference with typing constraints (demo)

skip back



Type inference with typing constraints (demo)

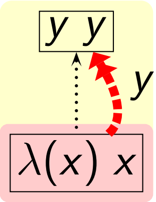
skip back

$$\boxed{\begin{array}{c}\text{let } y = \lambda(x)\ x \\ \text{in } y\ y\end{array}}$$

Type inference with typing constraints (demo)

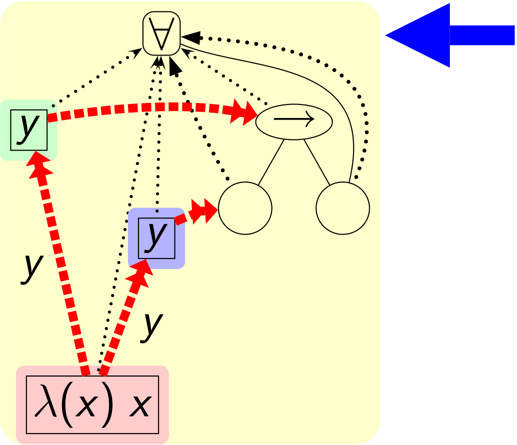
skip back

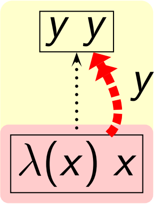
$$\boxed{\begin{array}{c}\text{let } y = \lambda(x)\ x \\ \text{in } y\ y\end{array}}$$



Type inference with typing constraints (demo)

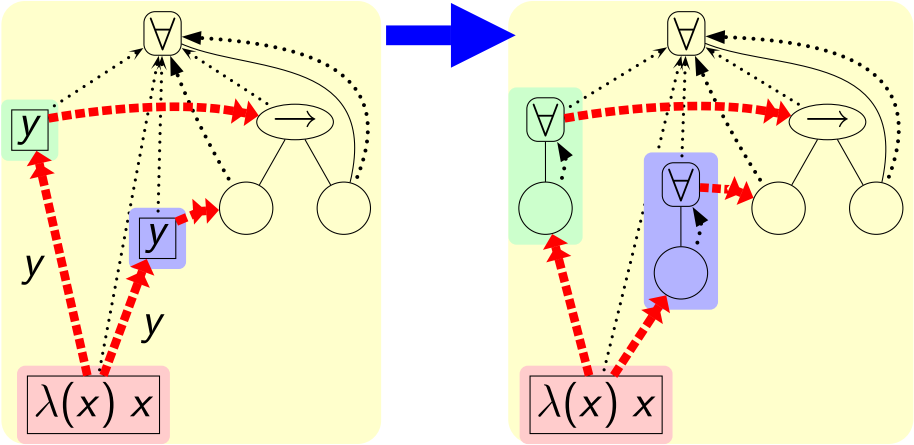
skip back



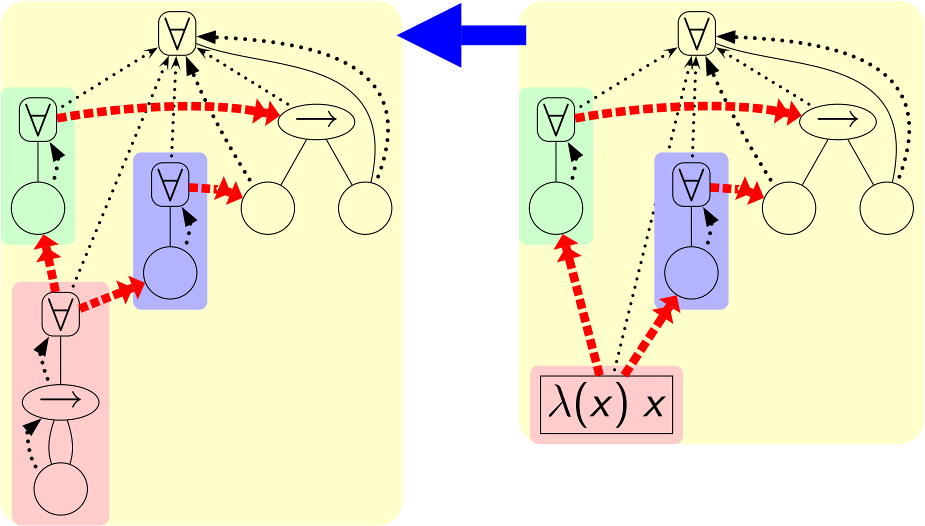


Type inference with typing constraints (demo)

skip back

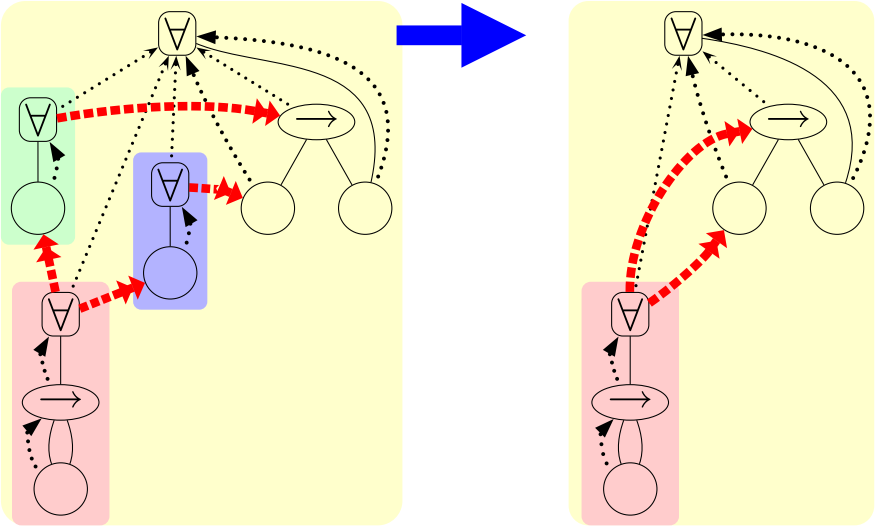


Type inference with typing constraints (demo)



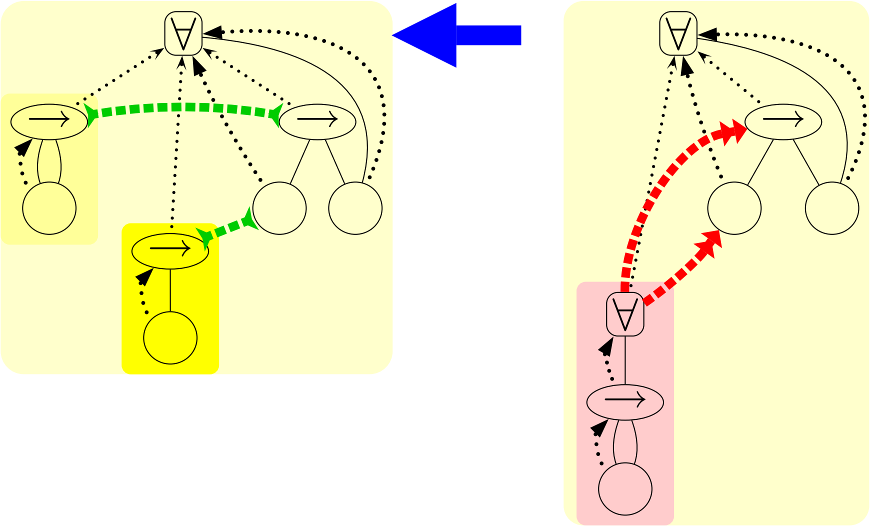
Type inference with typing constraints (demo)

skip back

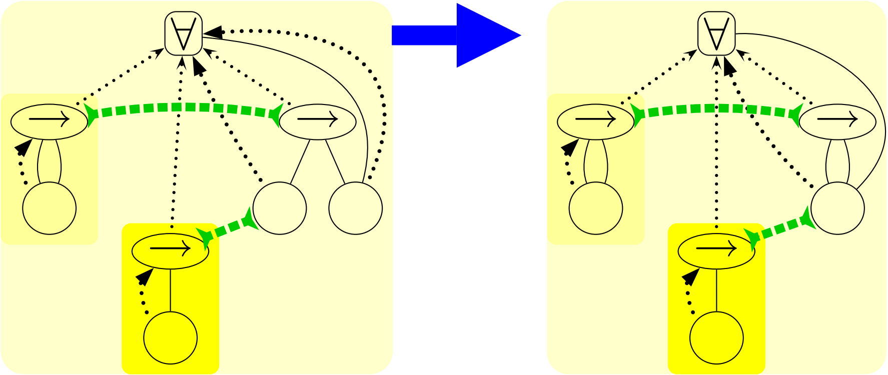


Type inference with typing constraints (demo)

skip back

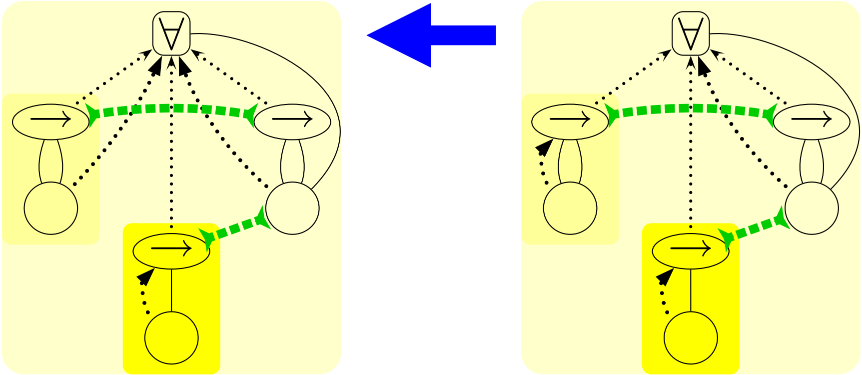


Type inference with typing constraints (demo)



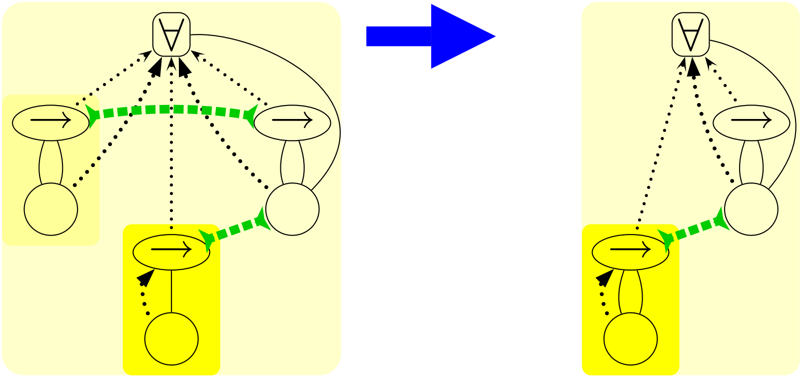
Type inference with typing constraints (demo)

skip back



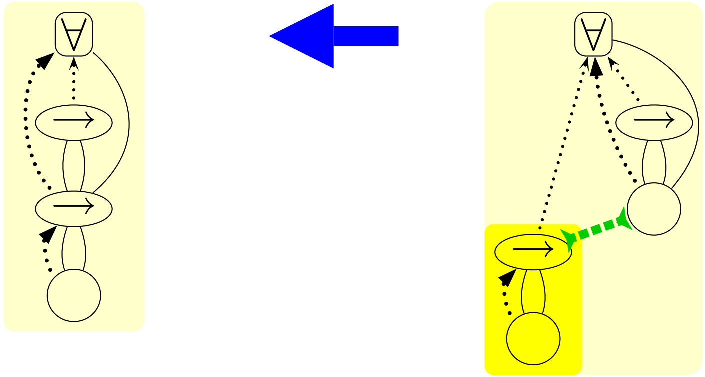
Type inference with typing constraints (demo)

skip back



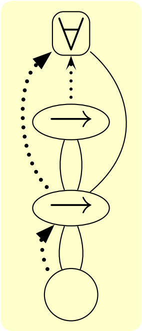
Type inference with typing constraints (demo)

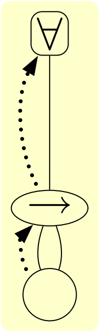
skip back



Type inference with typing constraints (demo)

skip back





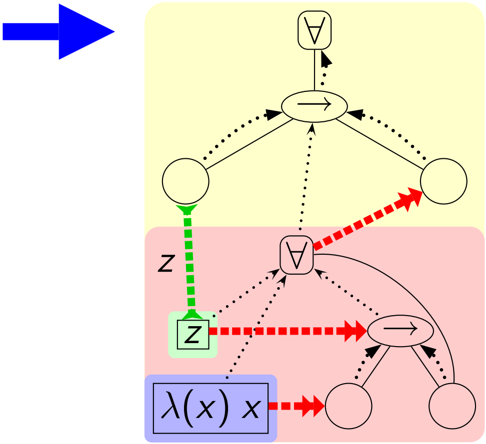
Type inference with typing constraints (demo)

skip back

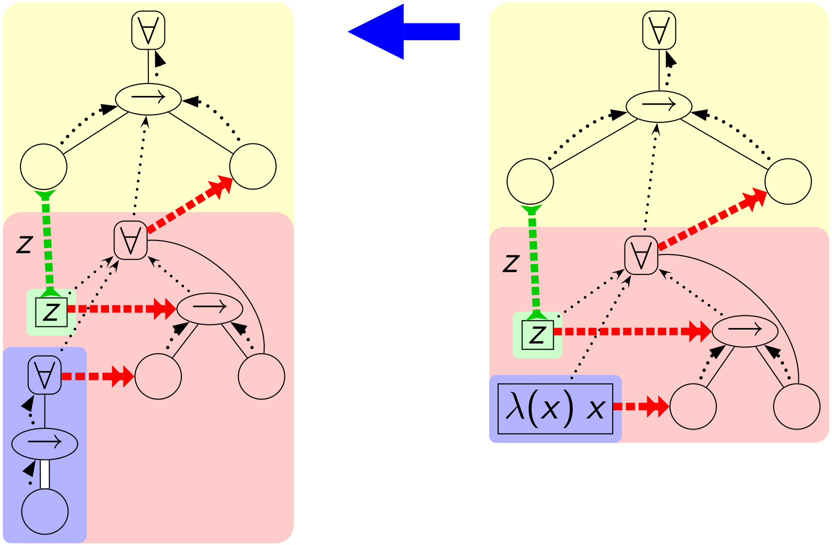
$$\boxed{\lambda(z)\ z\ (\lambda(x)\ x)}$$

Type inference with typing constraints (demo)

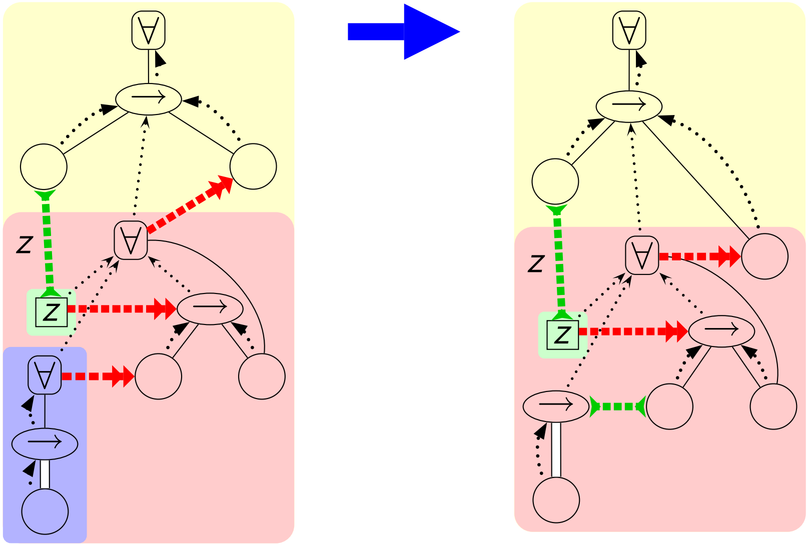
$\boxed{\lambda(z)\ z\ (\lambda(x)\ x)}$



Type inference with typing constraints (demo)

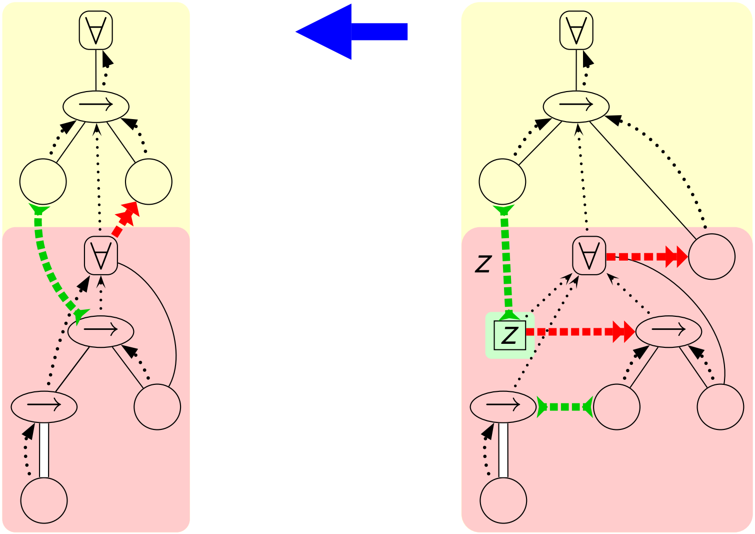


Type inference with typing constraints (demo)

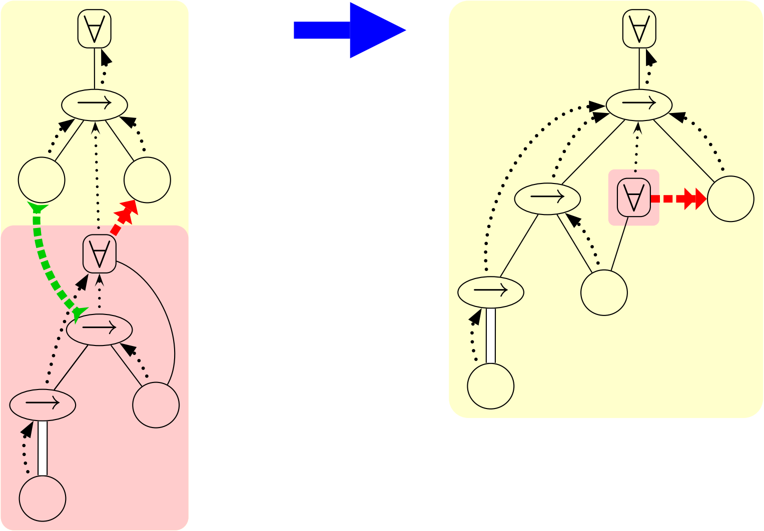


Type inference with typing constraints (demo)

skip back

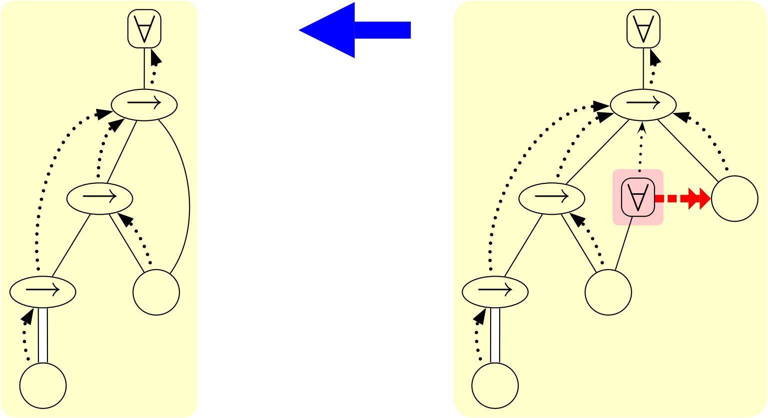


Type inference with typing constraints (demo)



Type inference with typing constraints (demo)

skip back



Type inference with typing constraints (demo)

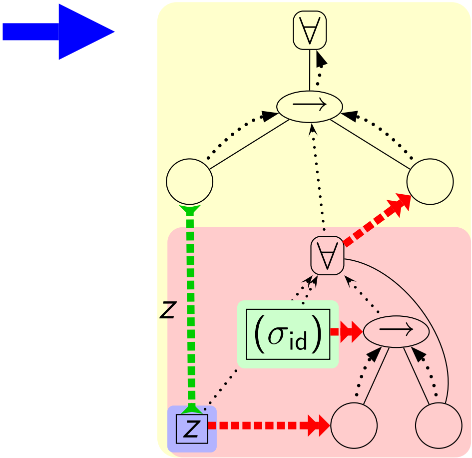
skip back

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

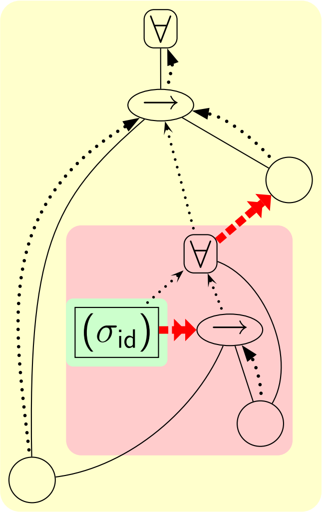
Type inference with typing constraints (demo)

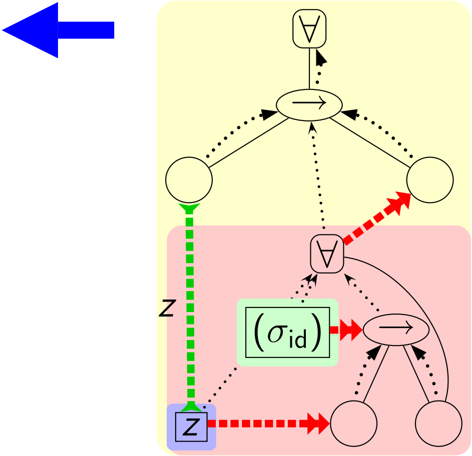
skip back

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



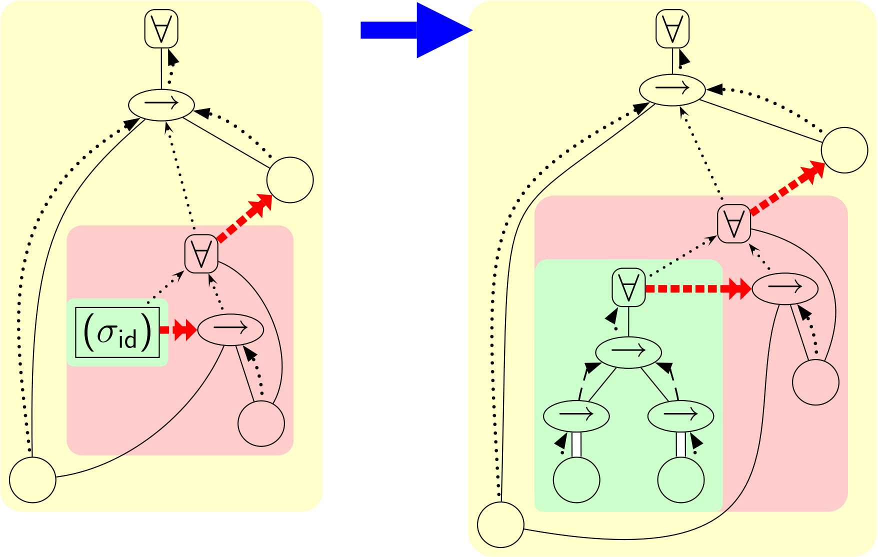
Type inference with typing constraints (demo)



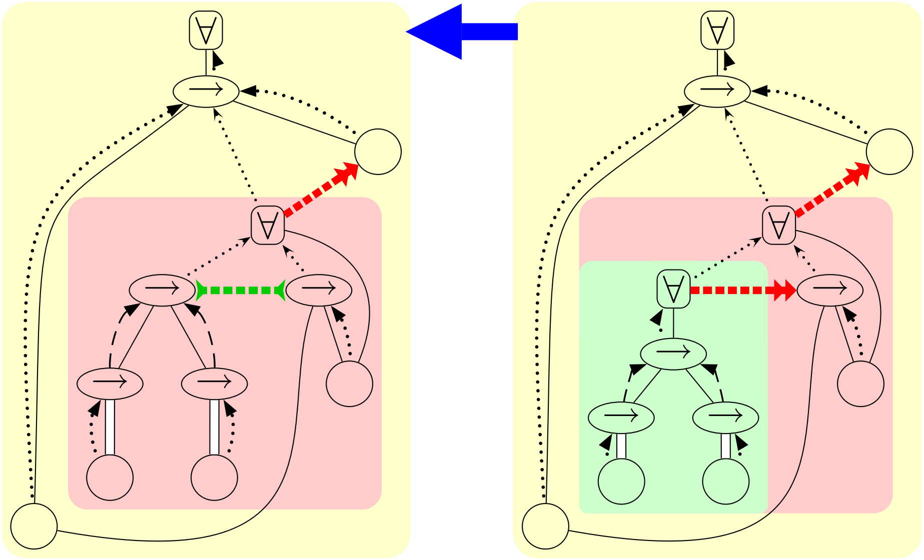


Type inference with typing constraints (demo)

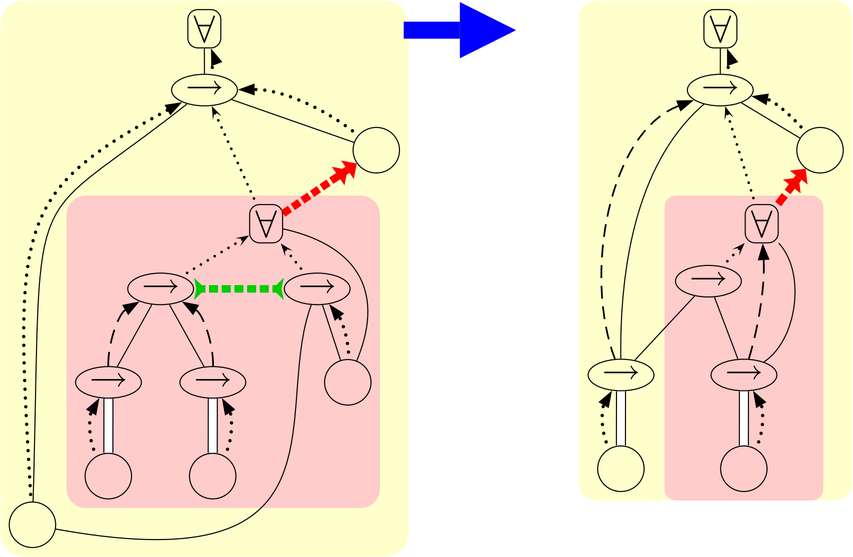
skip back



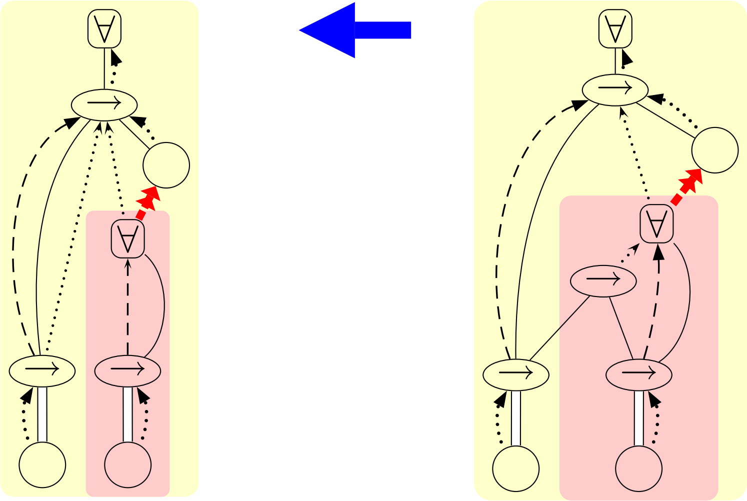
Type inference with typing constraints (demo)



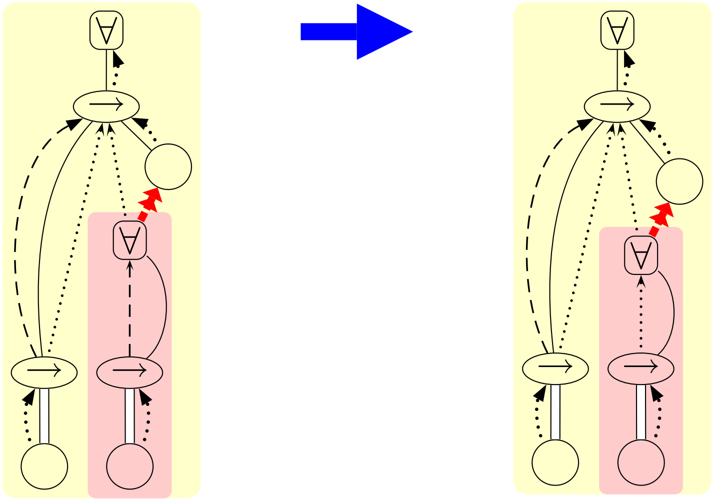
Type inference with typing constraints (demo)



Type inference with typing constraints (demo)

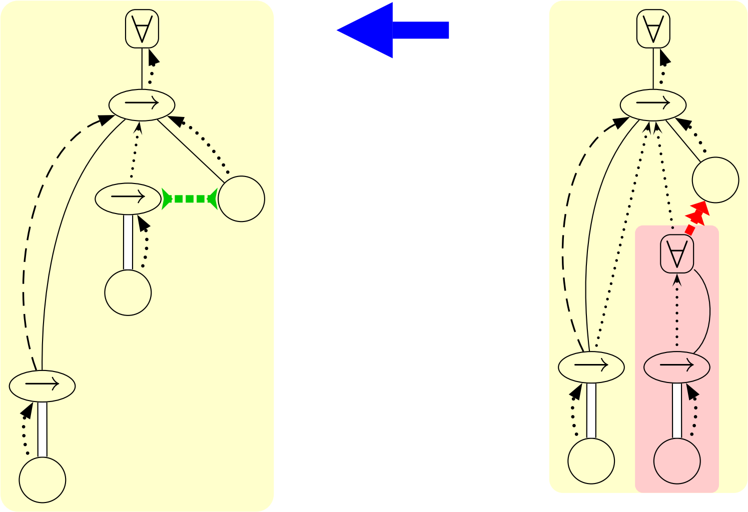


Type inference with typing constraints (demo)

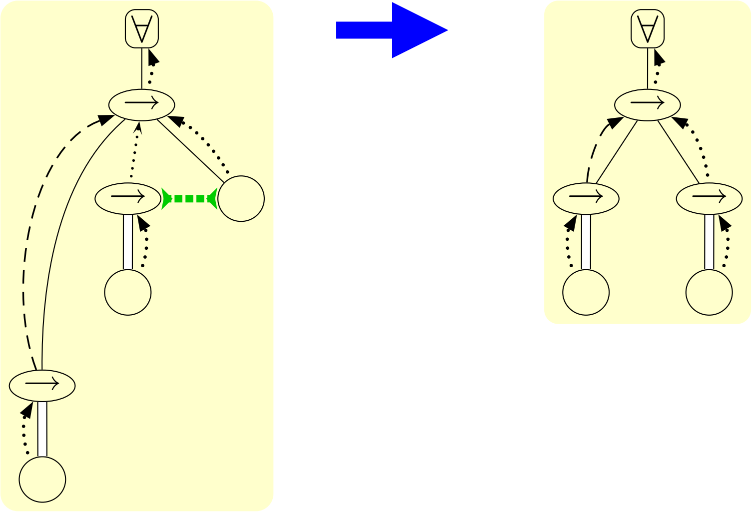


Type inference with typing constraints (demo)

skip back



Type inference with typing constraints (demo)



◀ back

More examples

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

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

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

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

let packed_int = pack (1, fun x $\to$ x+1);;
let packed_pair = pack (1, fun x $\to$ (x, x));;

let v = packed_int (fun p $\to$ (snd p) (fst p));;

About Rigid ML^F

back

Rigid ML^F lies very close to ML^F

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

Rigid ML^F looses important properties of ML^F

- There are no principal types *per se*.
  - Rigid ML^F 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^F is not invariant by let-expansion (which signs the lost of truly principal types).

About Rigid ML^{F}

back

Rigid ML^{F} lies very close to ML^{F}

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

Rigid ML^{F} looses important properties of ML^{F}

- There are no principal types *per se*.
- Rigid ML^{F} is not invariant by let-expansion (which signs the lost of truly principal types).

Rigid ML^{F} is a subset of System F

- This is both its interest and its problem.

What would be an intermediate language for ML^F?

Problem

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

Solution

- Introduce a fully explicit version of $x\mathrm{ML}^F$ (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: $\forall(\alpha \Rightarrow \sigma)\ \alpha \to \alpha \not\leq \forall(\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma)\ \alpha \to \alpha'$

Observe that:

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

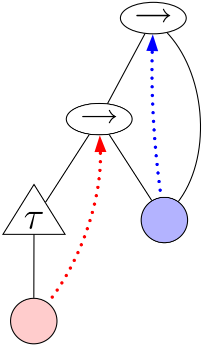
Then, the context $a \triangleq \lambda(x)\ []\ x\ x$ distinguishes those two expressions.

- $a[\lambda(z)\ z]$ 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

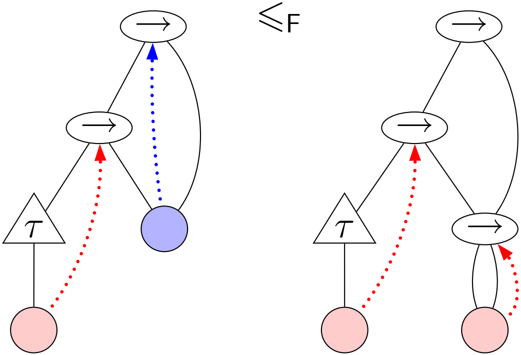
$$\forall(\alpha)\ (\forall(\beta)\ \tau_\beta \to \alpha) \to \alpha$$



System-F types (encoding of existential types)

back

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

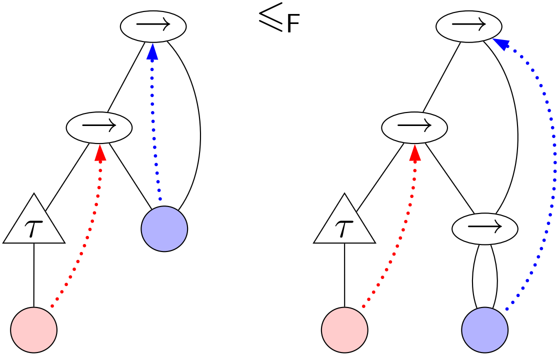


$$(\forall(\beta)\ \tau_\beta \to \forall(\alpha)\ \alpha \to \alpha) \to \forall(\alpha)\ \alpha \to \alpha$$

System-F types (encoding of existential types)

back

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



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

◀ back

Type annotations

back

$$\frac{\alpha \Rightarrow \sigma, \beta \Rightarrow \sigma \vdash \sigma \leqslant \alpha \text{ and } \sigma \leqslant \beta}{\alpha \Rightarrow \sigma, \beta \Rightarrow \sigma \vdash \begin{array}{ll} & \forall(\alpha' \Rightarrow \sigma)\,\forall(\beta' \Rightarrow \sigma)\,\alpha' \to \beta' \\ \leqslant & \forall(\alpha' \Rightarrow \alpha)\,\forall(\beta' \Rightarrow \beta)\,\alpha' \to \beta' \\ & \diamond \\ & \alpha \to \beta \end{array}}$$

$$\dfrac{\dfrac{\dfrac{\alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash (\_ : \sigma) : \alpha \to \beta \qquad \alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash x : \alpha}{\alpha \Rightarrow \sigma, x : \alpha, \beta \Rightarrow \sigma \vdash (x : \sigma) : \beta}}{\alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \forall(\beta \Rightarrow \sigma)\,\beta}}{\boxed{\alpha \Rightarrow \sigma, x : \alpha \vdash (x : \sigma) : \sigma}}$$

Type annotations

back

$$
\dfrac{\dfrac{\dfrac{\alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash (x : \sigma_{\text{id}}) : \sigma_{\text{id}}}{\alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash (x : \sigma_{\text{id}}) : \alpha \to \alpha} \qquad \alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash x : \alpha}{\dfrac{\alpha \Rightarrow \sigma_{\text{id}}, x : \alpha \vdash (x : \sigma_{\text{id}})\ x : \alpha}{\alpha \Rightarrow \sigma_{\text{id}} \vdash \lambda(x)\ (x : \sigma_{\text{id}})\ x : \alpha \to \alpha}}}{\vdash \lambda(x)\ (x : \sigma_{\text{id}})\ x : \forall(\alpha \Rightarrow \sigma_{\text{id}})\ \alpha \to \alpha}
$$