1. Design
   - *i*ML$^{\mathsf{F}}$: an implicity-typed extension of System F
   - Types explained
   - *e*ML$^{\mathsf{F}}$: an explicitly-typed version of *i*ML$^{\mathsf{F}}$
2. Results
   - Principal types
   - Robustness to program transformations
   - Practice
3. Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in ML$^{\mathsf{F}}$
4. Concluding remarks

A new look at ML^F

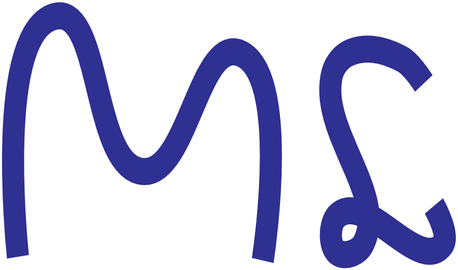
Didier Rémy

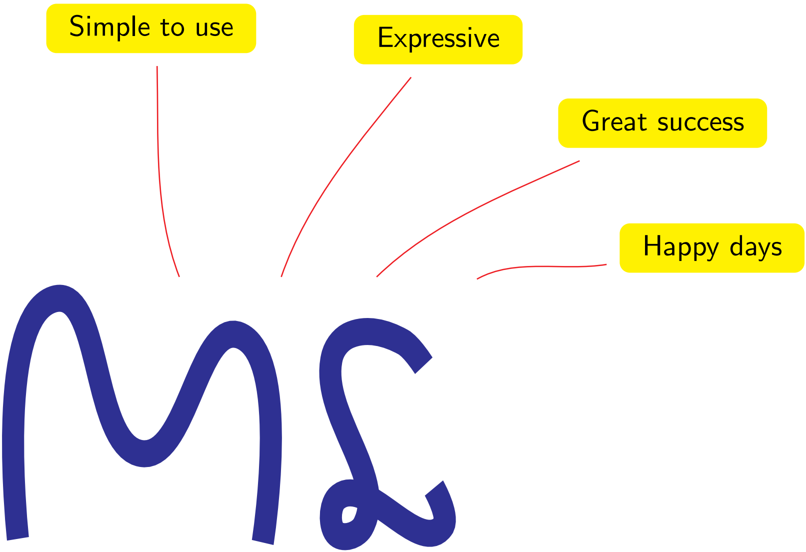
INRIA-Rocquencourt

Portland, June 2008

Based on joint work with

Didier Le Botlan and Boris Yakobowski)




Simple to use
Expressive
Great success
Happy days

F

Ms

F

Mɛ

Expressive

F

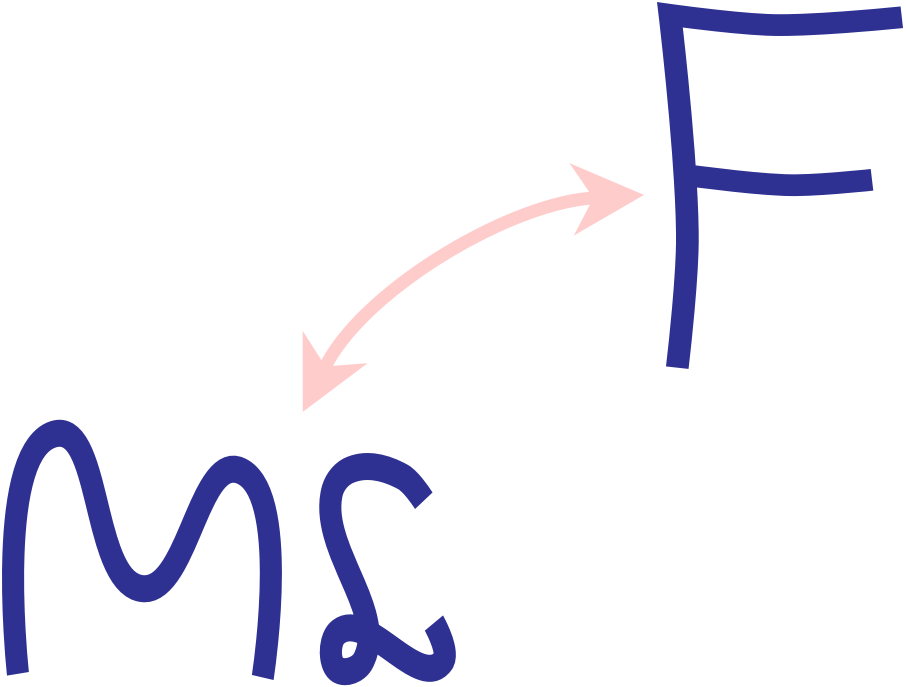
~~Simple extension~~ of ML
Simplification

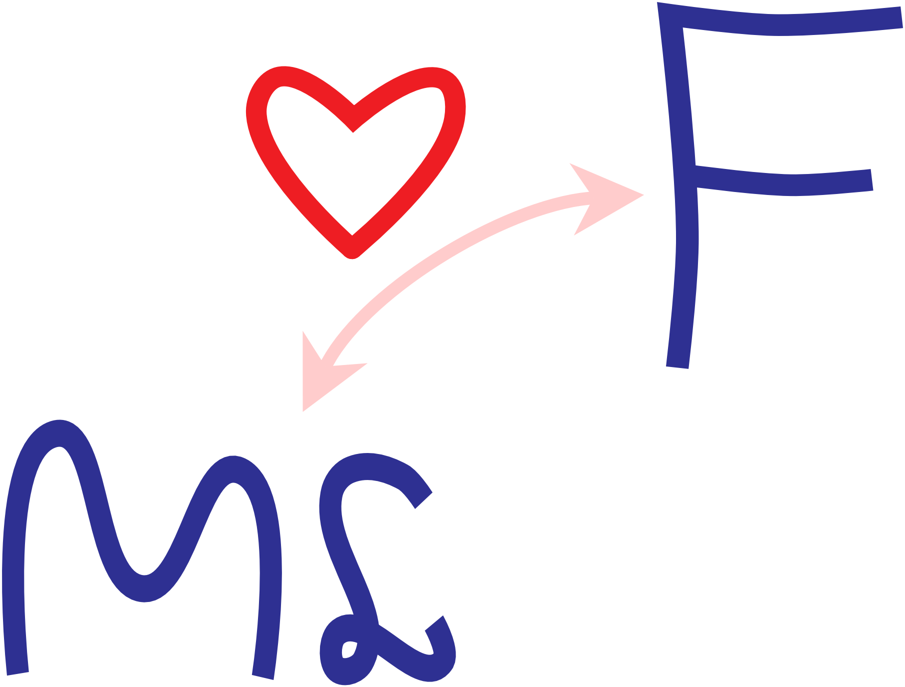
Even used in full scale languages
such as Scala.

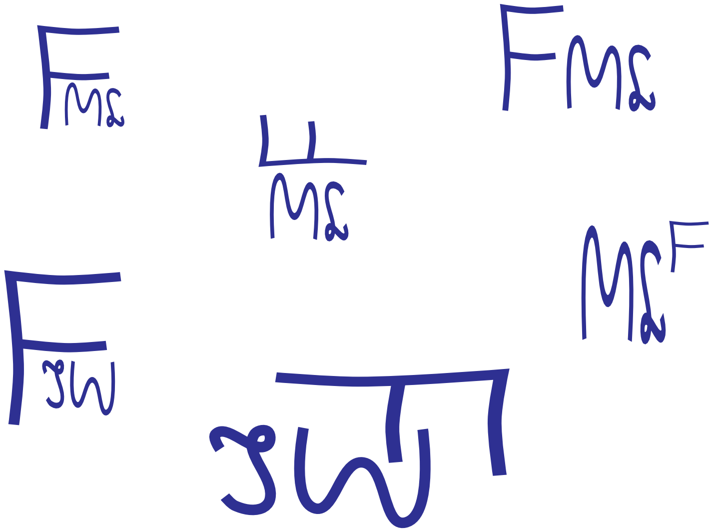
Full type inference
is undecidable

Full type annotations
are obfuscating

ML







MgF

Outline

1. Design
   - iML^F: an implicity-typed extension of System F
   - Types explained
   - eML^F: an explicitly-typed version of iML^F
2. Results
   - Principal types
   - Robustness to program transformations
   - Practice
3. Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in ML^F
4. Concluding remarks

A universal type system

Explicit System F:

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

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

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

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:

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

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

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

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

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

A universal type system

Implicit System F:

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

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

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

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

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

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

A universal type system

Implicit System F:

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

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

A universal type system

Implicit System F:

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

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

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

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:

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

APP
$$\Gamma \vdash$$

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

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):

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

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 monomorphic 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 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* $= \lambda(x)\ \lambda(y)$ **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 :

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

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

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** $y : \forall\,\beta \cdot \beta \to \beta \to \beta$

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

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

No better choice in F!

The problem is serious and inherent

- Follows from rules $\textsc{Inst}$, $\textsc{Gen}$, and $\textsc{App}$.
- 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:

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

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

The definition of $i\mathsf{ML}^\mathsf{F}$

Types are stratified

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

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

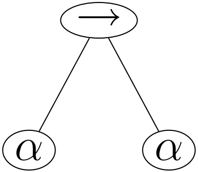
$$\begin{array}{rl} \{\!\{\tau\}\!\} & \triangleq \{\tau' \mid \tau \leqslant_\mathsf{F} \tau'\} \\ \{\!\{\forall(\alpha \geq \sigma)\ \sigma'\}\!\} & \triangleq \left\{ \forall(\bar\beta)\ \tau'[\alpha \leftarrow \tau] \mid \wedge \left(\begin{array}{l} \tau \in \{\!\{\sigma\}\!\} \wedge \tau' \in \{\!\{\sigma'\}\!\}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

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

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

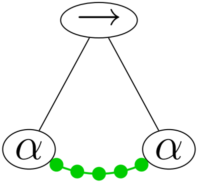
Simple types

$$\alpha \to \alpha$$



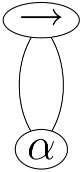
Simple types

$$\alpha \to \alpha$$



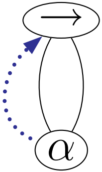
Simple types

$$\alpha \to \alpha$$



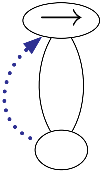
System-F types

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



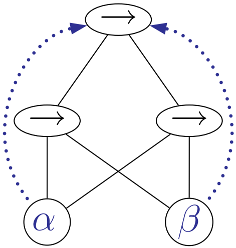
System-F types

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



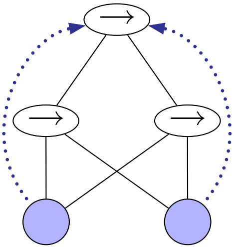
System-F types

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



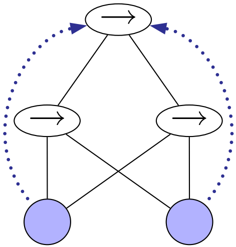
System-F types

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

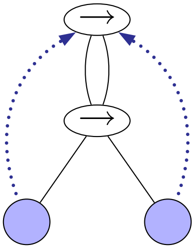


System-F types

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

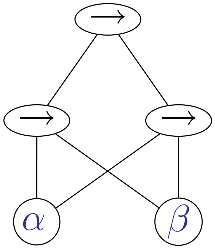


$\sqsupseteq_F$

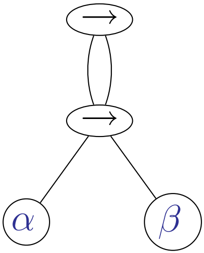


System-F types

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



$\lessapprox_{ST}$

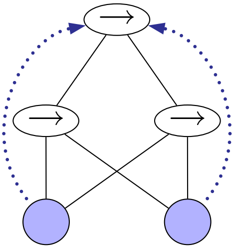


Sharing of inner nodes:

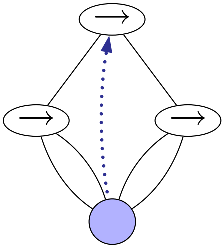
- Coming from the dag-representation of simple types.
- Canonical (unique) representation if disallowed.

System-F types

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



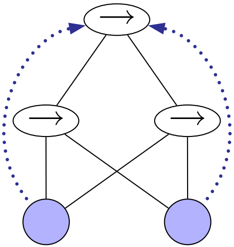
$\leqslant_F$

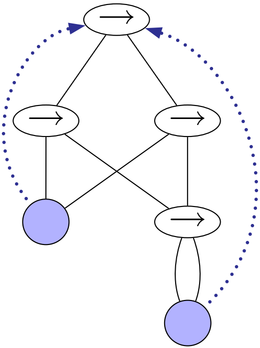


$$\forall(\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha$$

System-F types

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



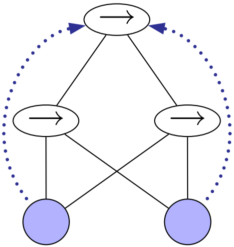


$$\forall(\alpha)\ \forall(\gamma)\ (\alpha \to \gamma \to \gamma) \to \alpha \to \gamma \to \gamma$$

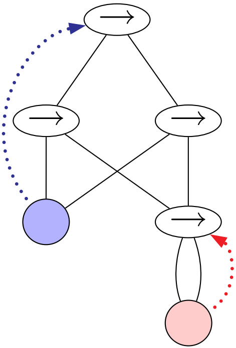
System-F types

▸ more

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



$\leqslant_{\mathsf{F}}$



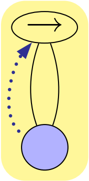
$$\forall(\alpha)(\alpha \to \forall(\gamma)\ \gamma \to \gamma) \to \alpha \to \forall(\gamma)\ \gamma \to \gamma$$

Types in iML^F

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

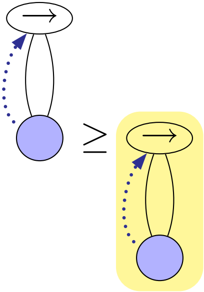
Types in iML^F

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



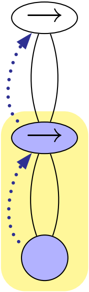
Types in *i*ML^F

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



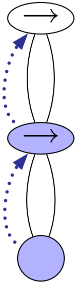
Types in $i\mathsf{ML}^{\mathsf{F}}$

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



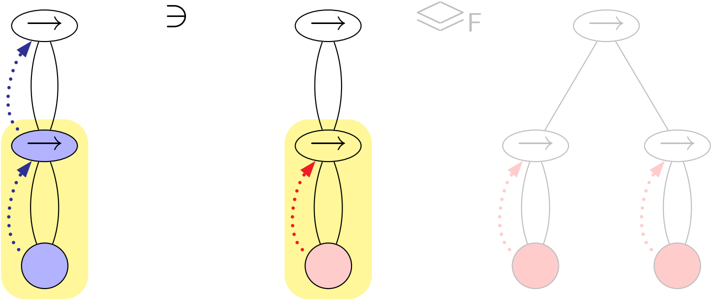
Types in $i\mathrm{ML}^{\mathrm{F}}$

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



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

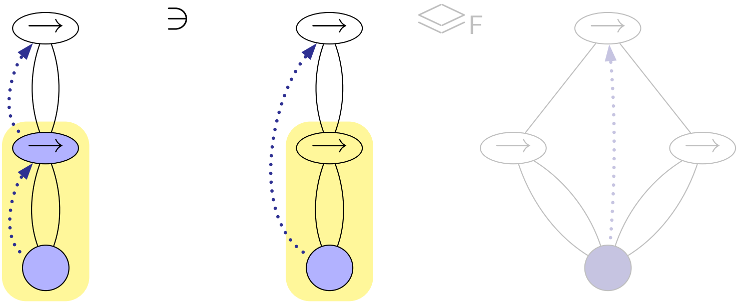
$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha\,)\, \beta \to \beta$$



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

Types in iML^F

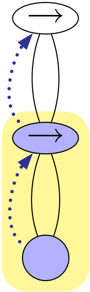
$$\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha\,)\ \beta \to \beta$$



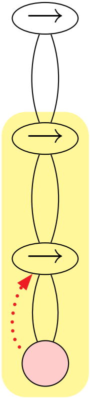
$$\forall(\alpha)\ (\alpha \to \alpha) \to (\alpha \to \alpha)$$

Types in iML^F

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

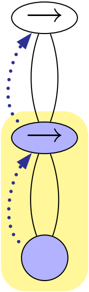


$\ni$

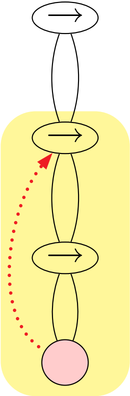


Types in iML^F

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

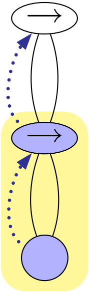


$\ni$

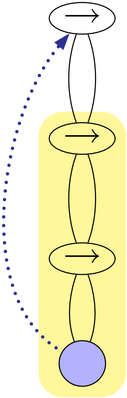


Types in iML^F

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

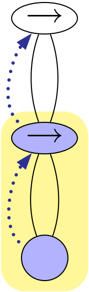


$\ni$



Types in iML^F

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



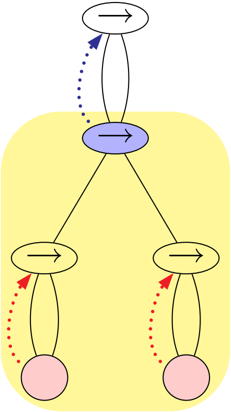
$\ni$ $\quad \dots$

The semantics cannot be captured by

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

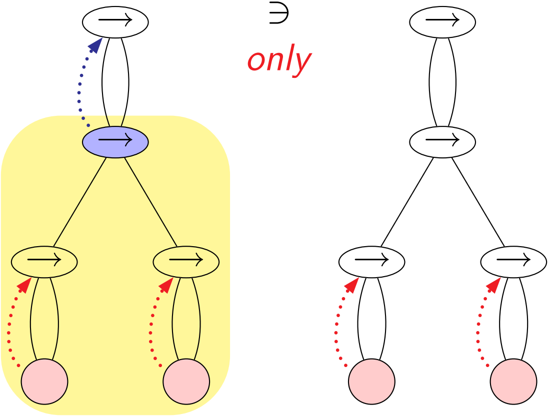
*i*ML^F types

$$\forall(\beta \geq \boxed{(\forall(\alpha)\ \alpha \to \alpha) \to (\forall(\alpha)\ \alpha \to \alpha)}) \to \beta \to \beta$$



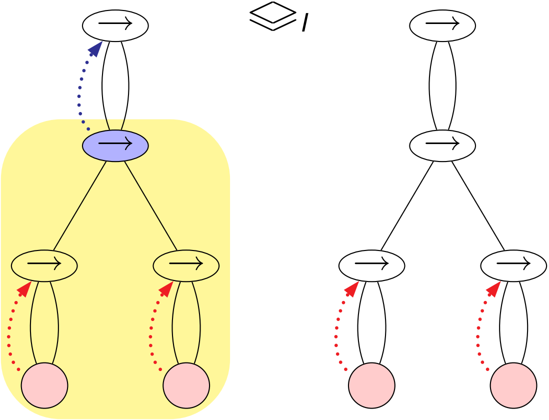
iML^F types

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



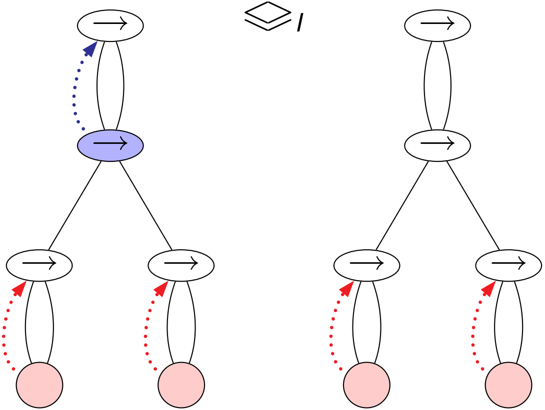
iML^F types

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



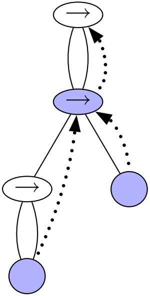
iML^F types

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



Type instance $\leqslant$ in iML^F

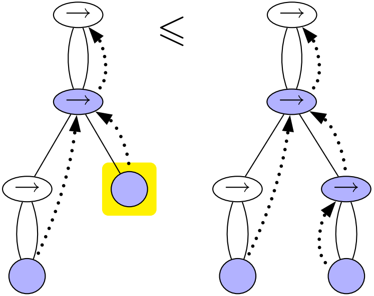
Only four atomic instance operations, and only two new.



Type instance $\leqslant$ in iML^F

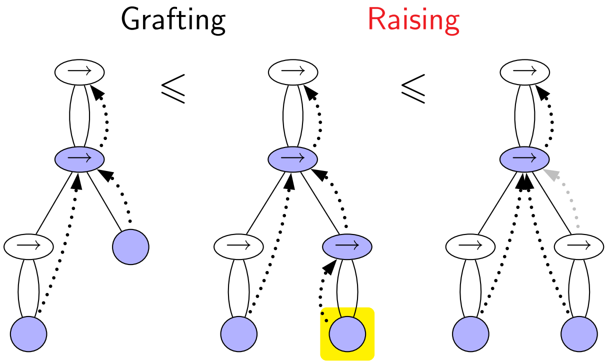
Only four atomic instance operations, and only two new.

Grafting



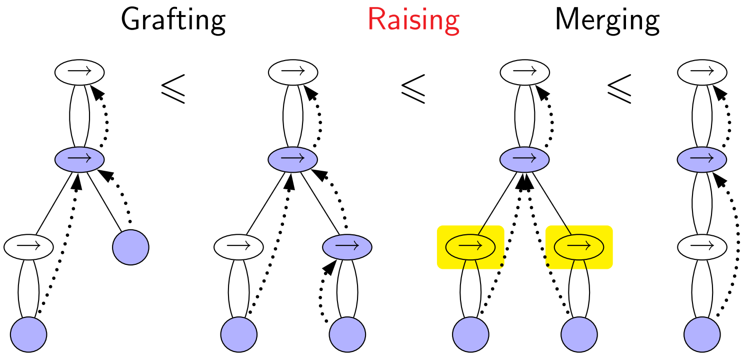
Type instance $\leqslant$ in iML^F

Only four atomic instance operations, and only two new.



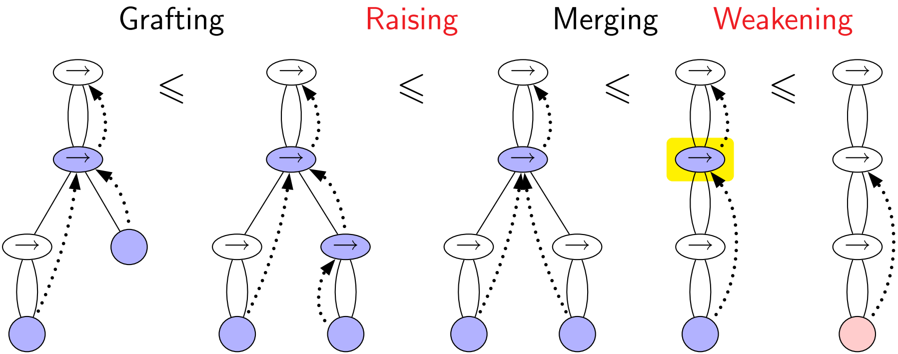
Type instance $\leqslant$ in iML^F

Only four atomic instance operations, and only two new.



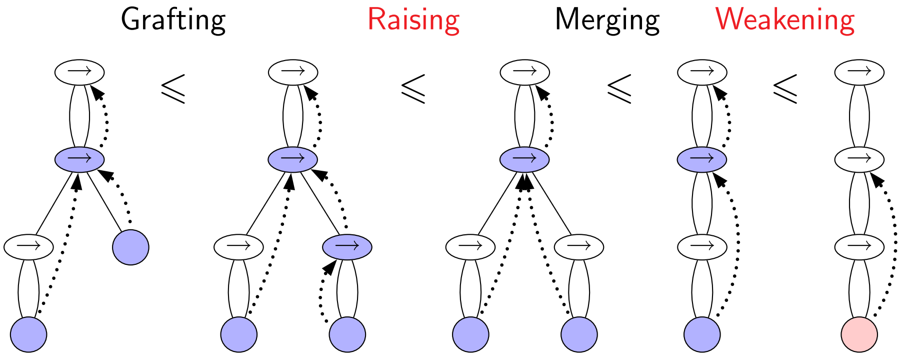
Type instance $\leqslant$ in iML^F

Only four atomic instance operations, and only two new.



Type instance $\leqslant$ in iML^F

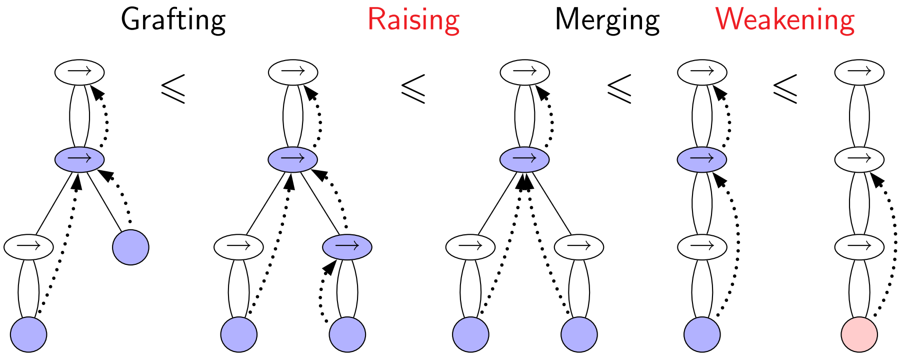
Only four atomic instance operations, and only two new.



- Merging only allowed on nodes transitively bound at the root (blue).
- Other operations only disallowed on variable nodes that are not transitively bound at the root (red).

Type instance $\leqslant$ in iML^F

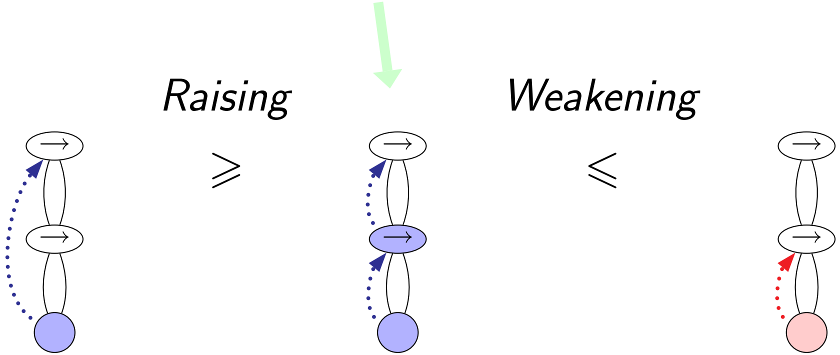
Only four atomic instance operations, and only two new.



These operations are sound and complete for the definition of $\leqslant$.

Can always be ordered as $\leqslant^G$; $\leqslant^R$; $\leqslant^{MW}$.

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. Results
   - Principal types
   - Robustness to program transformations
   - Practice
3. Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in ML^F
4. Concluding remarks

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 examples

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

- z carries values of a polymorphic type.

First-order inference of second order types

More challenging examples

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

- z carries 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 examples

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

- z must have a polymorphic type $\sigma \to \tau$.

First-order inference of second order types

More challenging examples

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

ACCEPT

- z must have a polymorphic type $\sigma \to \tau$.
- z need not be used polymorphically:
  it may just carry polymorphism without using it.
- Indeed, it is the reduct of

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

which can be typed in ML^F, exactly as the previous example.

Annotations need not be introduced during reduction!

Abstracting second-order polymorphism as first-order types

Solution

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

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

Mechanism

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

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

Abstracting second-order polymorphism

more

Key point: abstraction is directional

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

Hence,

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

Abstracting second-order polymorphism

▸ more

Key point: abstraction is directional

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

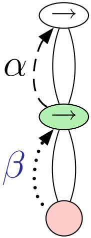
But,

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

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

Introduce a new binder for abstraction

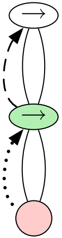
$$\forall(\alpha \Rightarrow \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha$$



Types in eML^F

Introduce a new binder for abstraction

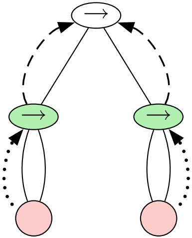
$$\forall(\alpha \Rightarrow \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha$$



Types in eML^F

Introduce a new binder for abstraction

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



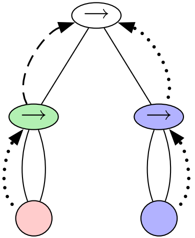
More general

sharing of $\Rightarrow$ matters

Types in eML^F

Introduce a new binder for abstraction

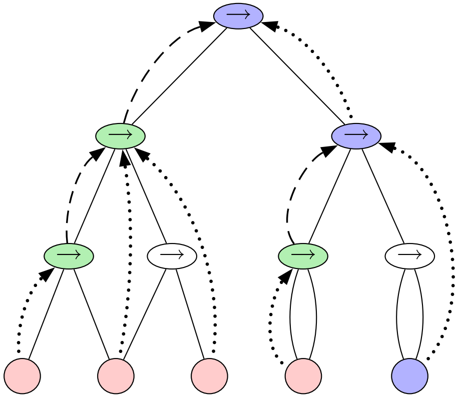
$$\forall(\alpha \Rightarrow \forall(\beta)\ \beta \to \beta)\ \forall(\alpha' \geq \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha'$$



Even more general $\geq$ better than $\Rightarrow$

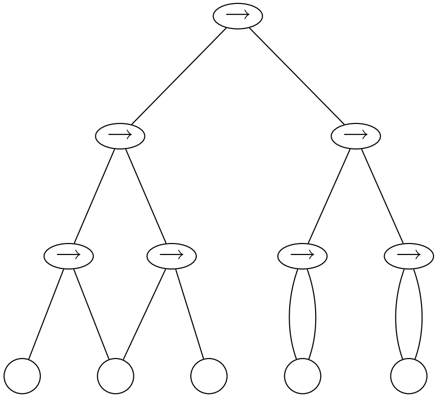
Types, graphically

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



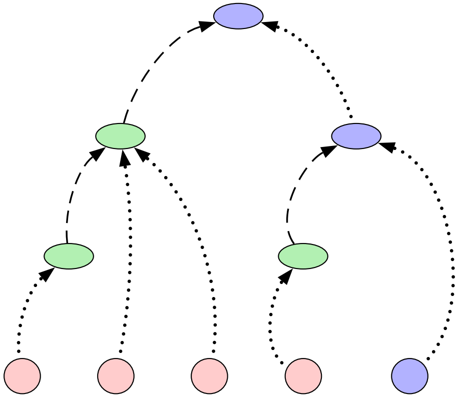
Types, graphically

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



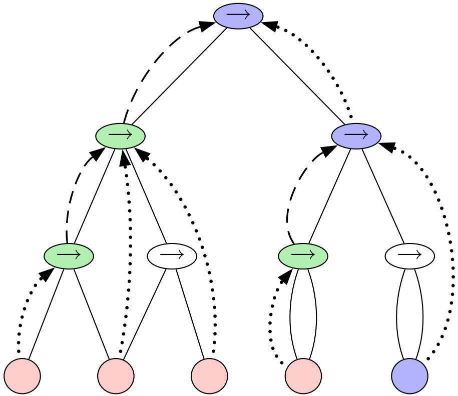
Types, graphically

= first-order term-dag + a binding tree



Types, graphically

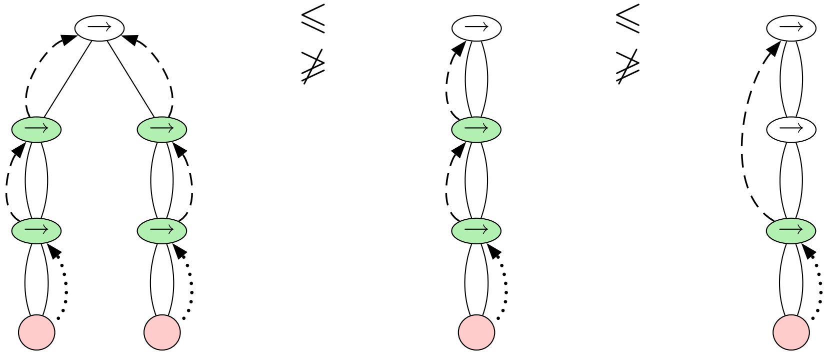
= first-order term-dag + a binding tree



+ well-formedness conditions relating the two

Type instance $\leqslant$ in eML^F

Sharing and binding of abstract nodes now 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

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

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

  Notice that the larger $\leqslant$, the fewer type annotations.

Type annotations

Recovering the missing power

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

Technically

- Intuitively,

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

- Actually, use coercion functions:

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

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

Type annotations

Recovering the missing power

$$(\leqslant) \subset (\leqslant_I) = (\leqslant \cup \sqsupseteq_I)^*$$

Technically

- Intuitively,

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

- 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 $\lambda(x : \sigma)\ a \quad \triangleq \quad \lambda(x)\ \text{let } x = (x : \sigma) \text{ in } a$
  $\qquad \equiv \quad \lambda(x)\ a[x \leftarrow (x : \sigma)]$

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
   - *i*ML$^{\mathsf{F}}$: an implicity-typed extension of System F
   - Types explained
   - *e*ML$^{\mathsf{F}}$: an explicitly-typed version of *i*ML$^{\mathsf{F}}$
2. Results
   - Principal types
   - Robustness to program transformations
   - Practice
3. Type inference
   - Type constraints for simple types
   - Type constraints for ML
   - Type inference in ML$^{\mathsf{F}}$
4. Concluding remarks

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 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 program transformations

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

Let-conversion $(x \in a_2)$ $\quad \text{let } x = a_1 \text{ in } a_2 \subseteq a_2[x \leftarrow a_1]$

Common subexpression can be factored out.

η-conversion of functional expressions $\quad a \subseteq \lambda(x)\, a\, x$

Delay the evaluation.

Redefinable application $\quad a_1\, a_2 \subseteq (\lambda(f)\, \lambda(x)\, f\, x)\, a_1\, a_2$

Many functionals, such as maps are typed as applications.

Reordering of arguments $\quad a\, a_1\, a_2 \subseteq (\lambda(x)\, \lambda(y)\, a\, y\, x)\, a_2\, a_1$

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

All valid in ML^F

Robustness to program transformations

Reduction

- Transforms existing type annotations
- Does not introduce new type annotations

Printing types

▶ more

Problem

- Types are graphs.
- They can be represented syntactically with prefix notation,
- Not very readable: compare $(\forall(\gamma)\,\gamma \to \gamma) \to (\forall(\gamma)\,\gamma \to \gamma)$
  with $\forall(\alpha \Rightarrow \forall(\gamma)\,\gamma \to \gamma)\,\forall(\beta \geq \forall(\gamma)\,\gamma \to \gamma)\,\alpha \to \beta$

Solution

- Inline linear bindings that are
  - flexible at covariant positions, or
  - rigid at contravariant positions.
- Very effective in practice: types look often as in System F.

Examples

▶ more

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.

Outline

Type inference for ML

Based on first-order unification

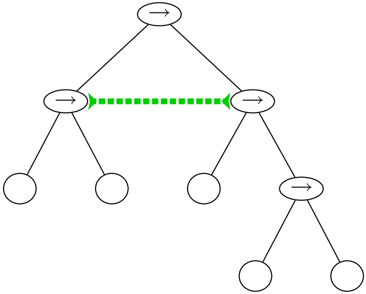
- Best implemented *and* formalized using graphs (Huet) or, equivalently, multi-equations.

Type inference

- Best formalized by type constrains
- See *The essence of ML, in ATTAPL*.

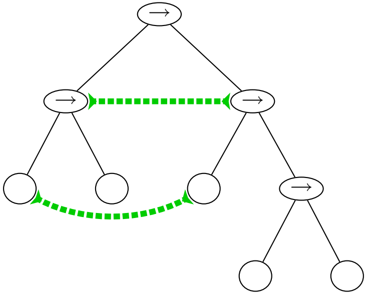
First-order unification

Unification problems may be represented on a term using unification edges



First-order unification

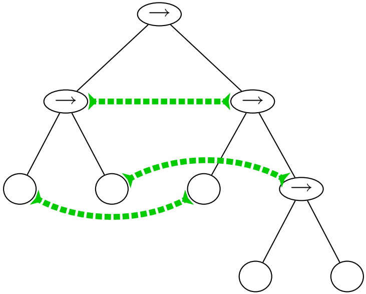
Unification problems may be represented on a term using unification edges



Congruence closure

First-order unification

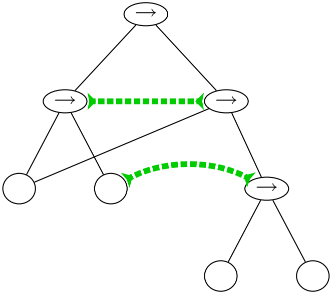
Unification problems may be represented on a term using unification edges



Merging

First-order unification

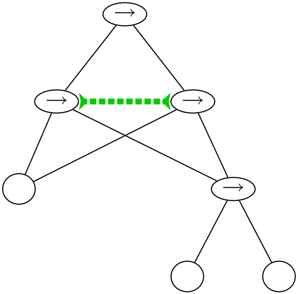
Unification problems may be represented on a term using unification edges



Grafting

First-order unification

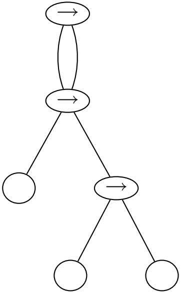
Unification problems may be represented on a term using unification edges



Merging

First-order unification

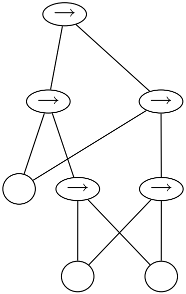
Unification problems may be represented on a term using unification edges



- Unification builds a dag, by merging variabes or inner nodes,

First-order unification

Unification problems may be represented on a term using unification edges



- Unification builds a dag, by merging variabes or inner nodes,
- However, the dag may be read back up to sharing of inner nodes.
  - Because extra sharing will never block further simplications
  - Thus, inner nodes could always be maximally shared (hash-consing).
  - Hence, sharing of inner nodes does not matter.

Unification formally

Term view

A solution to a unification problem is an instance in which subterms connected by unification edges are equal.

Graph view (simpler)

A solution to a unification problem is an instance in which nodes connected by unification edges are identical.

The algorithm (with simple proof of correctness)

- Each transformation preserves the set of solutions.
- Applying transformations terminates, with either:
  - an obvious conflict, thus, their is no solution; or
  - a graph without constraints, hence
    a solution of which all others are instances, *i.e.* a principal solution.

Type inference for simply typed λ-calculus

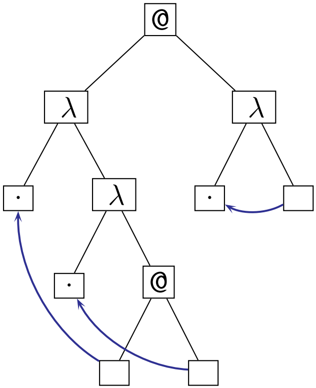
It is well-known that it reduces to unification problems:

Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ \big(\lambda(y)\ y\big)$$

Graphically: the λ-term

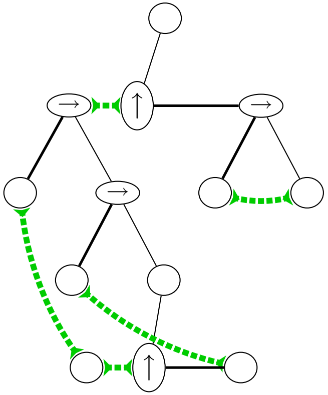


Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ \big(\lambda(y)\ y\big)$$

Graphically: its type constraint

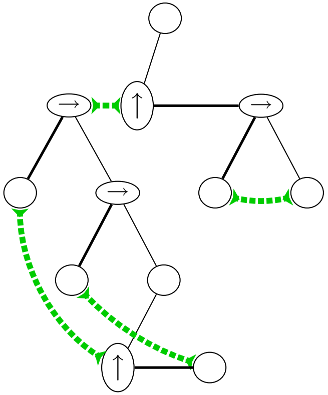


Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ \big(\lambda(y)\ y\big)$$

Graphically: its type constraint

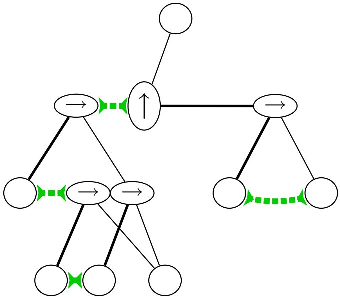


Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ (\lambda(y)\ y)$$

Graphically: its type constraint

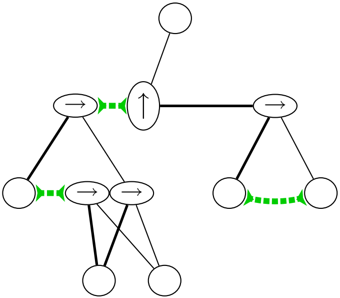


Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ \big(\lambda(y)\ y\big)$$

Graphically: its type constraint

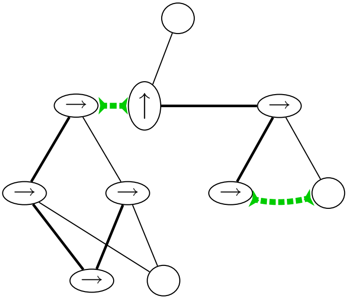


Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ \big(\lambda(y)\ y\big)$$

Graphically: its type constraint

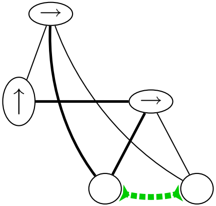


Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ \big(\lambda(y)\ y\big)$$

Graphically: its type constraint

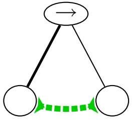


Type inference for simply typed λ-calculus

Example:

$$\Big(\lambda(f)\ \lambda(x)\ f\ x\Big)\ \big(\lambda(y)\ y\big)$$

Graphically: its solved form

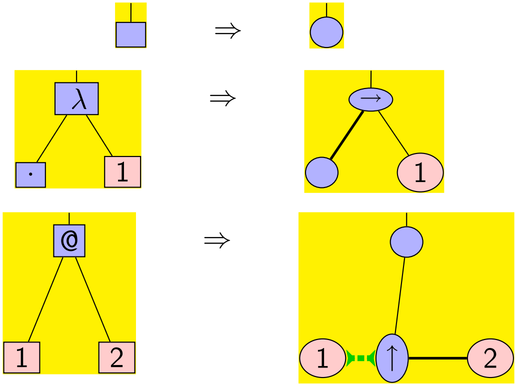


Constraint generation (more details)

Variables

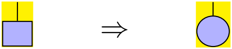
Functions

Applications

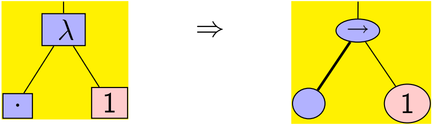


Constraint generation (more details)

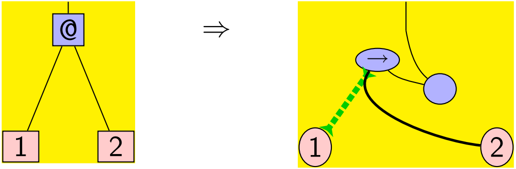
Variables



Functions

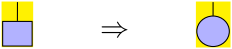


Applications

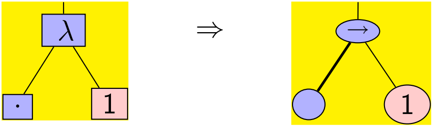


Constraint generation (more details)

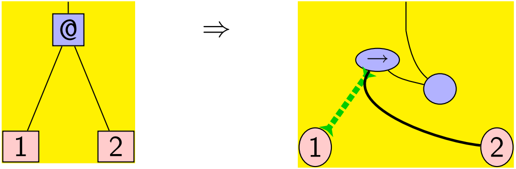
Variables



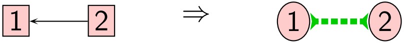
Functions



Applications



Bindings



Type inference for let-bindings

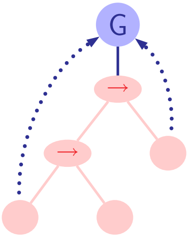
Can we extend the previous schema?

- The question is usually eluded in books.
- The solution is type inference with let-constraints.
- Can be better explained graphically.

Type inference for let-bindings

Introduce G-nodes (Generalization points)

to represent type schemes and distinguish them from types



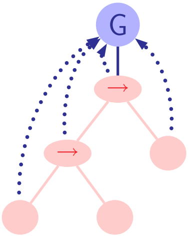
$$\boxed{\forall(\alpha\beta)\ \boxed{(\alpha \to \beta) \to \gamma}}$$

Generalized variables are drawn as binding edges to G.

Type inference for let-bindings

Introduce G-nodes (Generalization points)

to represent type schemes and distinguish them from types



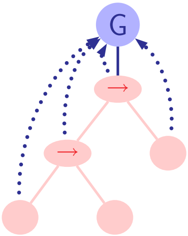
$$\forall(\alpha\beta)\ \boxed{(\alpha \to \beta) \to \gamma}$$

Generalized variables are drawn as binding edges to G.
Inner nodes may also be bound to G-nodes.

Type inference for let-bindings

Introduce G-nodes (Generalization points)

to represent type schemes and distinguish them from types



$$\forall(\alpha\beta)\ \boxed{(\alpha \to \beta) \to \gamma}$$

Generalized variables are drawn as binding edges to G.

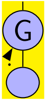
Constraint generation

Expressions now represent G-nodes., *i.e.* type scheme constraints.

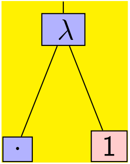
Constraint generation for ML (revisited)

Variables

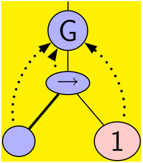
$\Rightarrow$



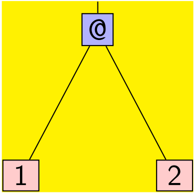
Functions



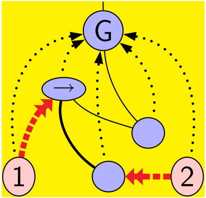
$\Rightarrow$



Applications

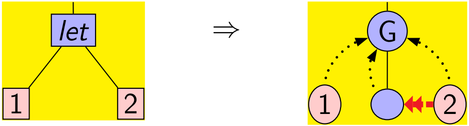


$\Rightarrow$

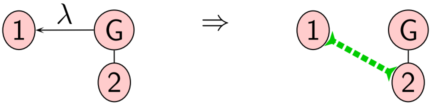


Constraint generation for ML (revisited)

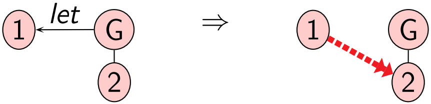
Let-bindings



λ-bindings



let-bindings

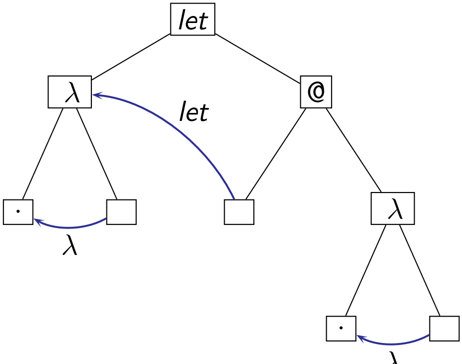


Constraint generation for ML (example)

Example:

let $g = \lambda(x)\ x$ in
$g\ \left(\lambda(y)\ y\right)$

Graphically (on the right):



the λ-term

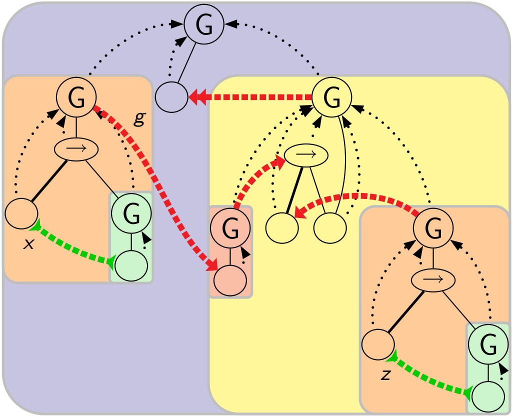
Constraint generation for ML (example)

Example:

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

Graphically (on the right):

its type constraint



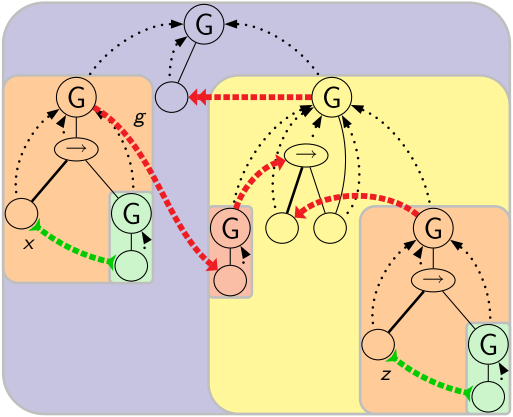
Constraint generation for ML (example)

Example:

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

Graphically (on the right):

its type constraint



Superfluous generalization points

- As in ML generalization is only needed at let-bindings.
- Useless G-nodes may be simplified after/during constraint generation.

Well-formedness of constraints

Well-formedness

- Arities: all nodes have a fixed number of outgoing structure edges
- Kinds: we distinguish G-nodes from other, regular nodes.
  - Instantiation edges are from G-nodes to regular nodes.
  - Unification edges are between from regular nodes.
- All nodes are bound to some G-node.
- The binding of a node is one of its dominators
  for mixed structure and binding edges.

Existential nodes

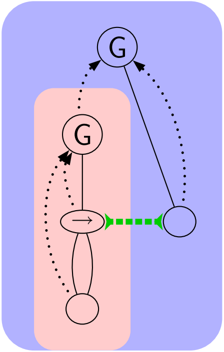
- Nodes that do not have an incoming structure edge.

Projection

- Remove all existential nodes and constraints.

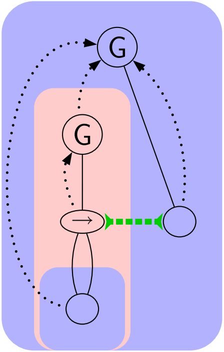
A new instance operation

Raising a binding edge along another one.
This amounts to treating a polymorphic as locally monomorphic



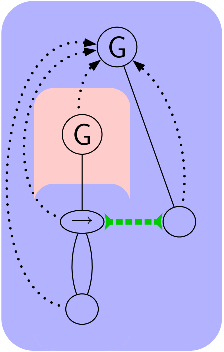
A new instance operation

Raising a binding edge along another one.
This amounts to treating a polymorphic as locally monomorphic



A new instance operation

Raising a binding edge along another one.
This amounts to treating a polymorphic as locally monomorphic

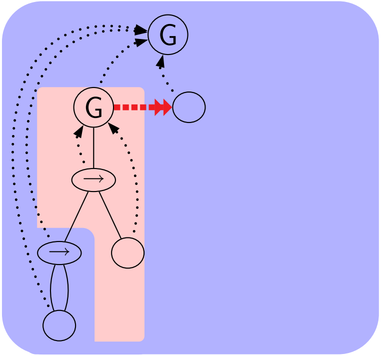


Solved instantiation edge

Informally

An instantiation edge is solved if its target is an instance of its origin.

Expansion

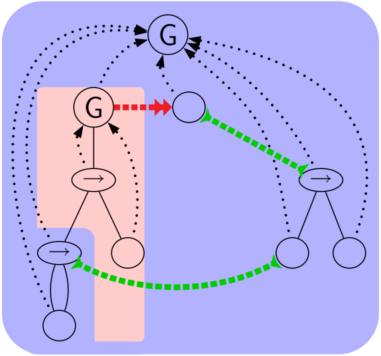


Solved instantiation edge

Informally

An instantiation edge is solved if its target is an instance of its origin.

Expansion



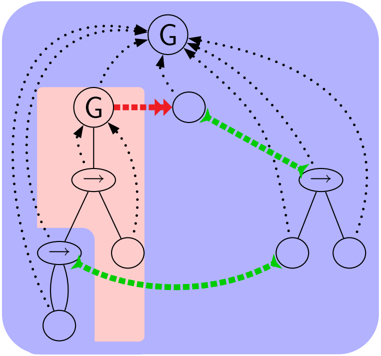
Expansion does not copy constraint edges nor existential nodes.

Solved instantiation edge

Informally

An instantiation edge is solved if its target is an instance of its origin.

Expansion



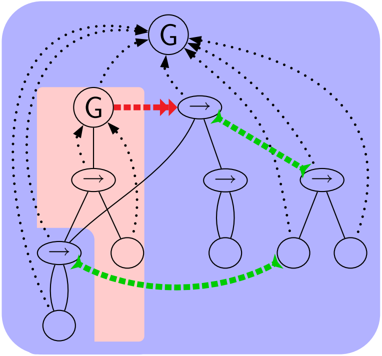
Can we come back to the original term by instantiation? —No

Solved instantiation edge

Informally

An instantiation edge is solved if its target is an instance of its origin.

Expansion



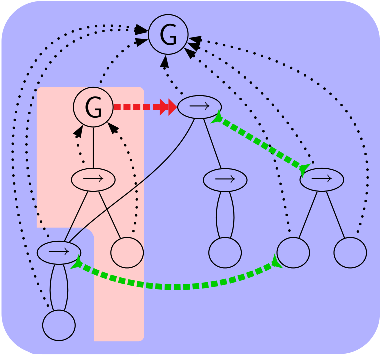
Can we come back to the original term by instantiation? —Yes

Solved instantiation edge

Informally

An instantiation edge is solved if its target is an instance of its origin.

Expansion



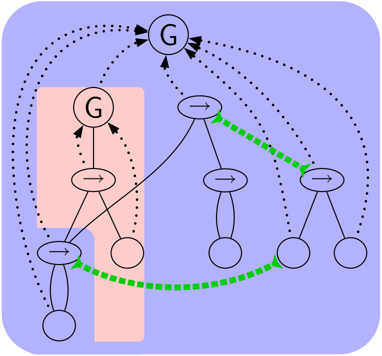
Expansion can be used as a test

Solved instantiation edge

Informally

An instantiation edge is solved if its target is an instance of its origin.

Expansion



Expansion can also be used as a simplification: the instantiation edge can be removed, if the origin is solved type scheme were solved.

Semantics of constraints

The set of its instances in which all constained edges are solved.

Constraint simplifications (preserve the semantics)

- Solving a unification edge by unification (as before).
- Expansion-elimination of an instantiation edge whose origin is solved.
- Garbage collection of unconstrained existential nodes.
- Elimination of superfluous G-nodes.

Algorithm

1. Eliminate superfluous G-nodes first, for efficiency.
2. Solve unification edges eagerly.
3. Solve instantiation constraints, innermost first.
4. Garbage collect at any time (no efficiency impact).

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

- k is the maximal size of types (usually not too large)
- d is the maximal nesting of type schemes
  i.e. after simplificatin of useless generalizations, let-nesting of let-bindings (reasonably below 5).

Explains why ML type inference works well in practice

- Large programs mainly increase right nesting of let-bindings.

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).

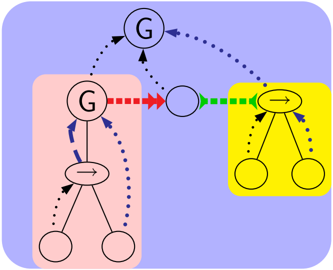
Note

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

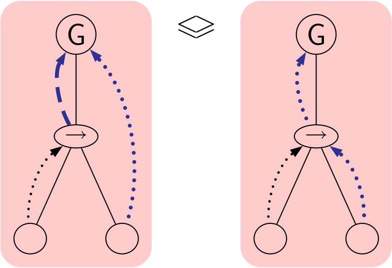
Type inference

Proceeds much as in ML, except that

- Generalize as much as possible at every step (not just at every let).
- Nodes may be bound to G-nodes or other nodes.
- Existential nodes only bound to G-nodes.
- Expansion is modified to reset topmost bindings:



i.e.



In particular, constraint generation is unchanged.

Type inference

▶ Demo?

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

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).

Outline

Variations on ML^F

skip

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}

skip

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

skip

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).

Variations on ML^F

skip

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 $\Rightarrow$.

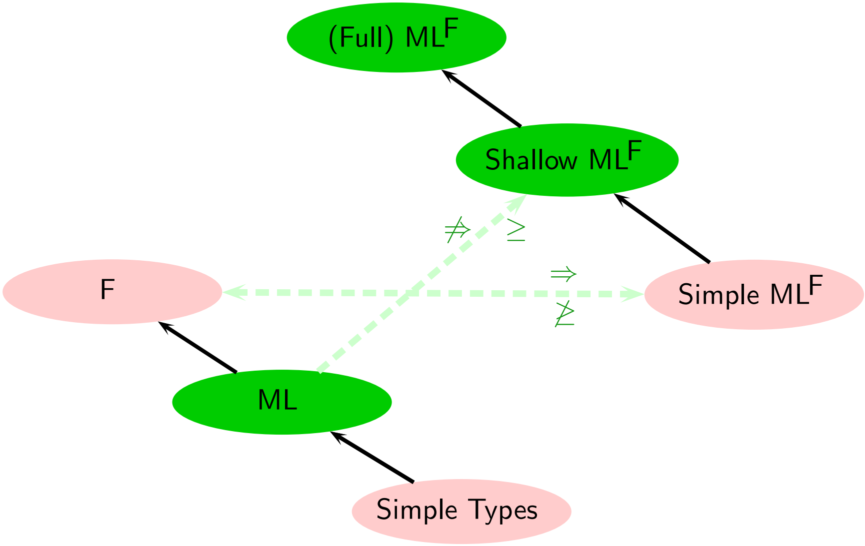
- Equivalent to System F.
- Principal types are lost (no type inference).

Is there an interesting variant in between?

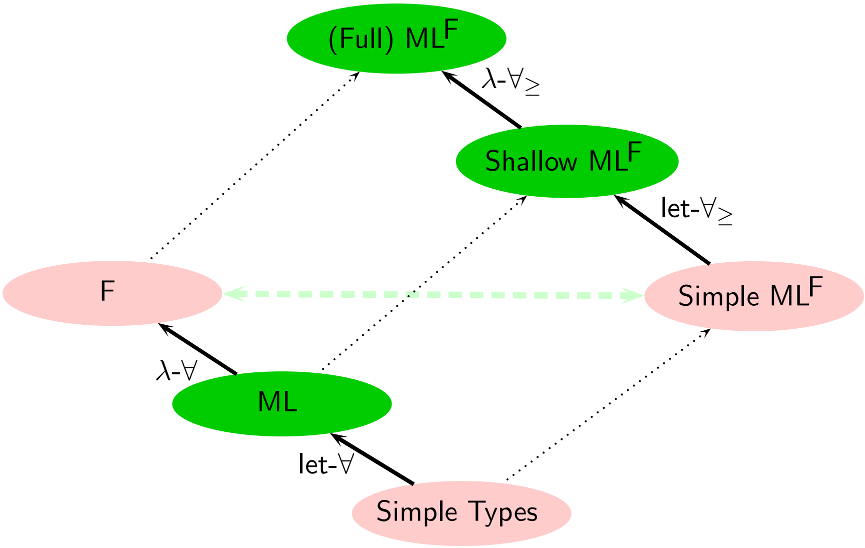
- As expressive as System F.
- With type inference and principal types.

Yes! Leijen's HML

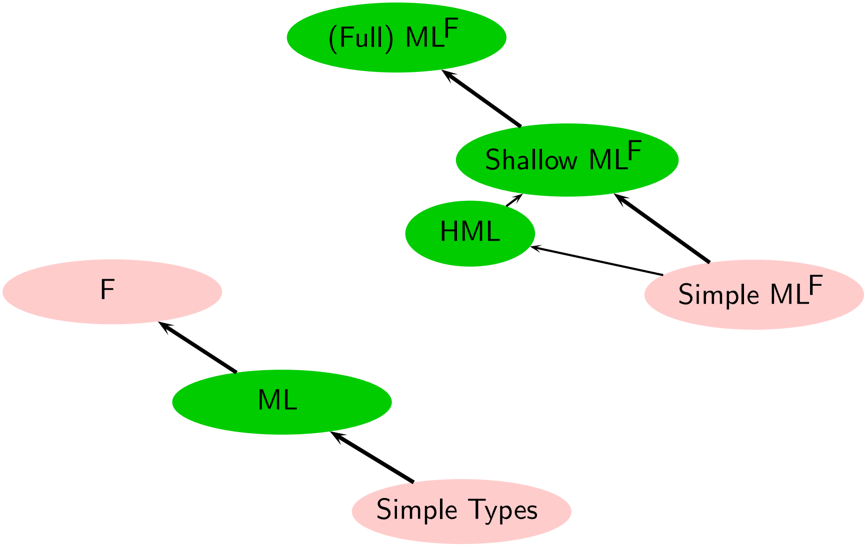
A hierarchy of languages



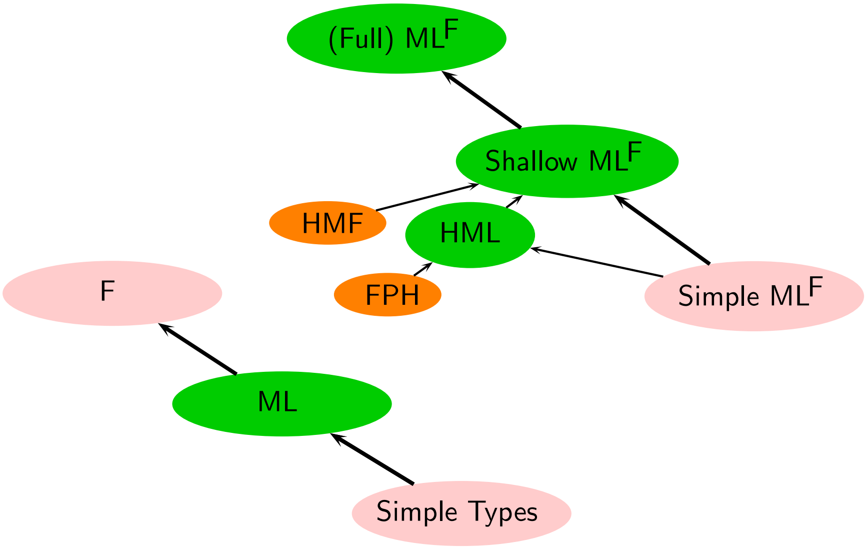
A hierarchy of languages



A hierarchy of languages



A hierarchy of languages



An internal language for ML^{F} (on going work)

Problem

- $i\mathsf{ML}^{\mathsf{F}}$ is in curry style.
- $e\mathsf{ML}^{\mathsf{F}}$ is not quite in church style:
  - type reconstruction is non local
  - type annotations must be transformed during reduction, but $e\mathsf{ML}^{\mathsf{F}}$ does not describe how to do so.
- Need for a church-style ML^{F} (*e.g.* compiling Haskell)

Solution

- Make type abstaction and type application fully explicit,
- Annotate all parameters of functions,
- Use a more general form of type application that witness the correct type-instantiation.

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?

Conclusions

To bring back home

- 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-schemes and generalization mechanism.
- Part of the credits should be returned to the great designer of ML.

Hopefully

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

Papers and prototypes

Talk mainly based on

- Recasting-ML^F *with Didier Le Boltan*.
- Graphic Type Constraints and Efficient Type Inference: from ML to ML^F, *with Boris Yakobowski*.

Other papers and online prototype at

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

See also Daan Leijen's papers and prototypes (HMF, HML)

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

and works by Vytinoitis *et al.* (Boxy types, FPH)

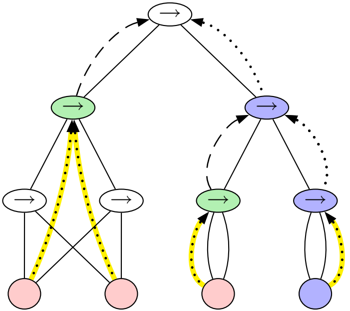
- http://research.microsoft.com/users/daan/pubs.html

Appendix

Printing types

◀ next

Only overlined bindings need to be drawn



Leave implicit bindings that are

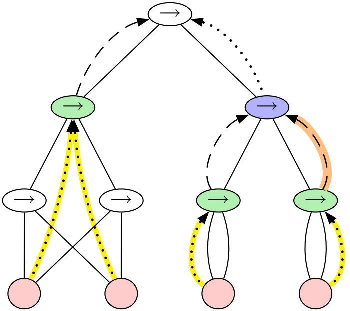
- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.

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

Printing types

◀ next

Only overlined bindings need to be drawn



Leave implicit bindings that are

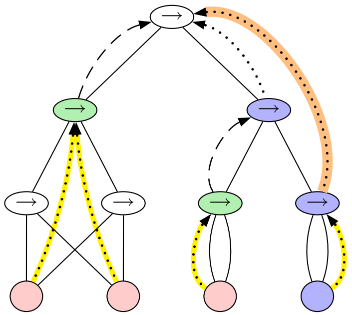
- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.

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

Printing types

◀ next

Only overlined bindings need to be drawn



Leave implicit bindings that are

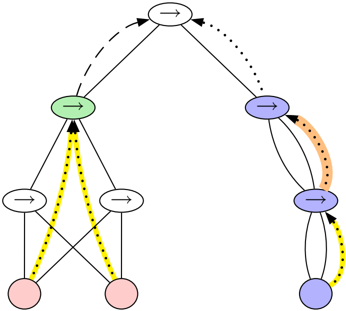
- at unshared, inner nodes,
- bound just above,
- abstractions on the left of arrows,
- instances on the right arrows.

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

Printing types

◀ next

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.

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

◀ back

More examples

◀ next

Church numerals

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

◀ next

Church 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

◀ next

Church numerals

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

Mandatory type annotations

let succ n = fun f x → n f (f x);;
let succ' = (succ : nat → nat);;
fails

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

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 → x+1);;
let packed_pair = pack (1, fun x → (x, x));;

let v = packed_int (fun p → (snd p) (fst p));;
```

HML: no rigid bindings

◀ back

HML, proposed by Daan Leijen

Very interesting!

- the specification uses the same types as $i\mathrm{ML}^{\mathsf{F}}$.

A strict subset of ML^{F}

- annotate exactly arguments that are used polymorphically.
- can be explained as follows;
  - Disable rigid bindings in prefixes.
  - Then, abstraction commutes with type inference
  - Hence, types may be treated up to abstraction. bindings.

Gains and losses

$\oplus$ Simpler, more intuitive types.

$\odot$ Keep most essential properties (pincipal types, robustness)

$\ominus$ Lost of some robustness. Polymorphism is not quite first-class.
e.g., primitive integers can't be replaced by church numerals.

FPH: only System-F like types *in the specification* ◀ back

HML can be further restricted

Less interesting...

- The specification uses only System-F types.

Many losses

- ⊙ Inference algorithm is kept (using ML^F internally...)
- ⊖ Bigger lost of some robustness.
- ⊖ No longer principal types *per se*.

Two variants to recover principal derivations

- HML: imposes minimal rank of polymorphism when ambiguous.
  which may require type annotations to get deeper polymorphism.
- FPH: requires no ambiguity at let-bindings,
  which may require type annotations to disambiguate.

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).

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.

◀ back

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.

back

Stability by linear beta-expansion

back

Linear β-conversion? $(\lambda^1(x)\ a_1)\ a_2 \overset{?}{\supseteq} a_1[x \leftarrow a_2]$

- No! otherwise, for $x \in a_1$:

$$
\begin{aligned}
(\lambda(x)\ a_1)\ a_2 &\supseteq (\lambda^1(x)\ \text{let } x = x \text{ in } a_1)\ a_2 \\
&\supseteq (\text{let } x = x \text{ in } a_1)[x \leftarrow a_2] \supseteq (\text{let } x = a_2 \text{ in } a_1)
\end{aligned}
$$

- Linearity is misleading:

$$\lambda^1(x)\ \text{let } y = x \text{ in } y\ y$$

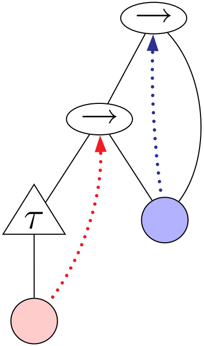
is not typable! Indeed, x must be used polymorphically via y.

back

System-F types (encoding of existential types)

back

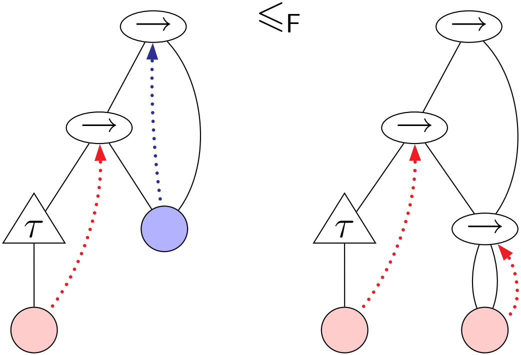
$$\forall(\alpha)\ (\forall(\beta)\ \tau_\beta \to \alpha) \to \alpha$$



System-F types (encoding of existential types)

back

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

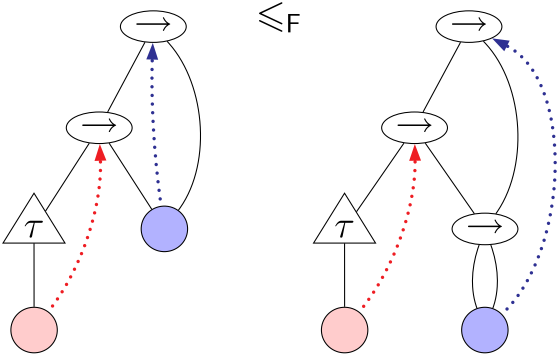


$$(\forall(\beta)\ \tau_\beta \to \forall(\alpha)\ \alpha \to \alpha) \to \forall(\alpha)\ \alpha \to \alpha$$

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

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

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

Type annotations

back

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

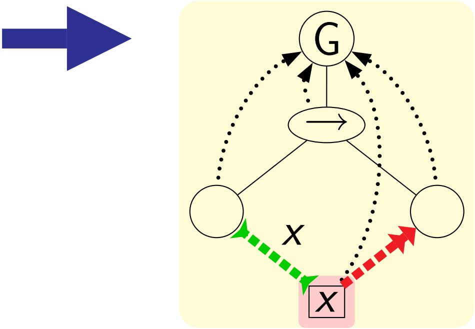
Type inference with typing constraints (demo)

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

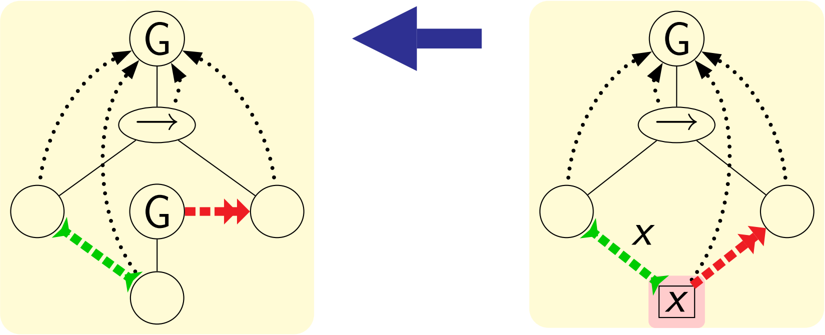
Type inference with typing constraints (demo)

▸▸ skip ◂ back

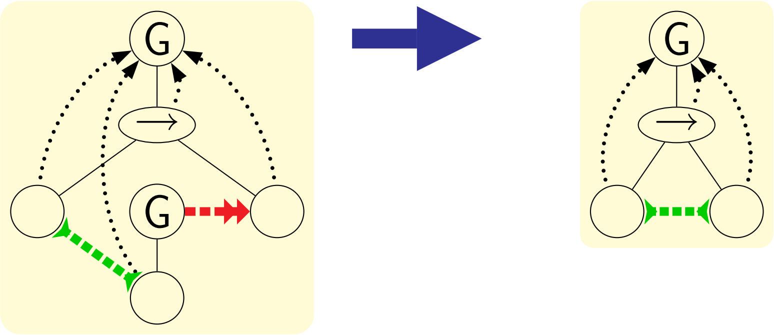
$\boxed{\lambda(x)\ x}$



Type inference with typing constraints (demo)

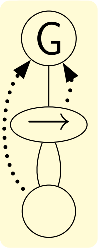


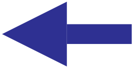
Type inference with typing constraints (demo)

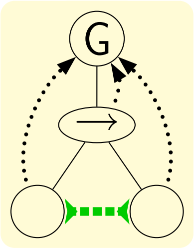


Type inference with typing constraints (demo)

▸▸ skip ◂ back







Type inference with typing constraints (demo)

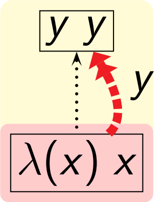
skip back

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

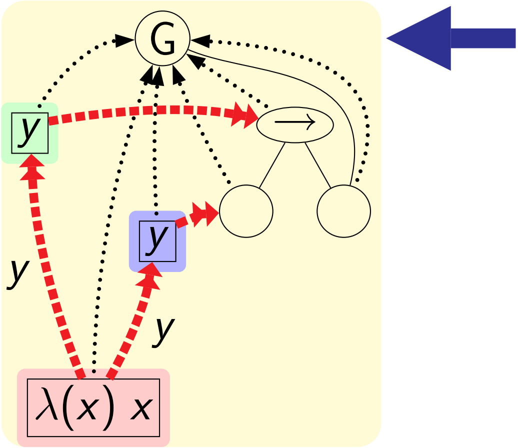
Type inference with typing constraints (demo)

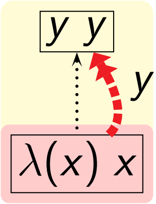
skip back

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



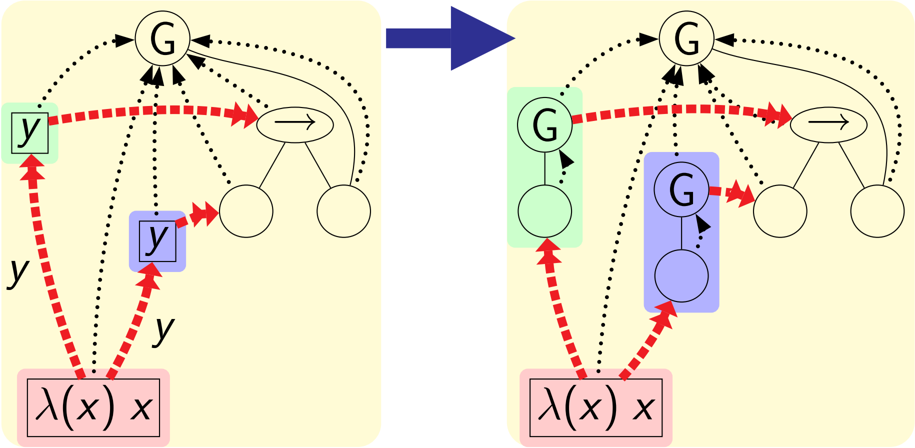
Type inference with typing constraints (demo)





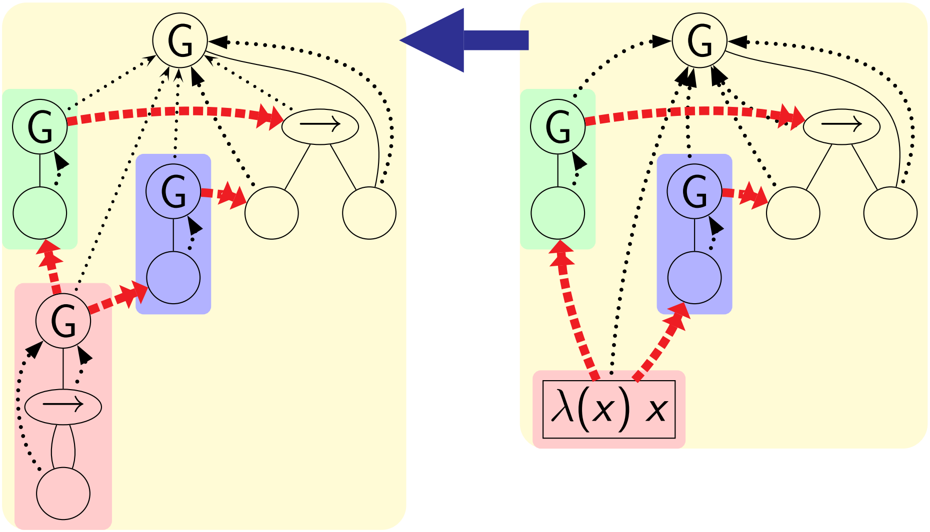
Type inference with typing constraints (demo)

skip back

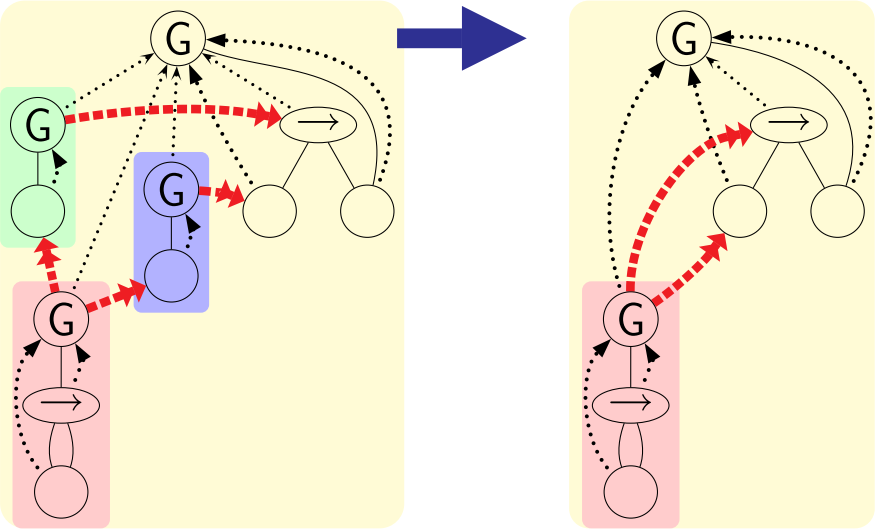


Type inference with typing constraints (demo)

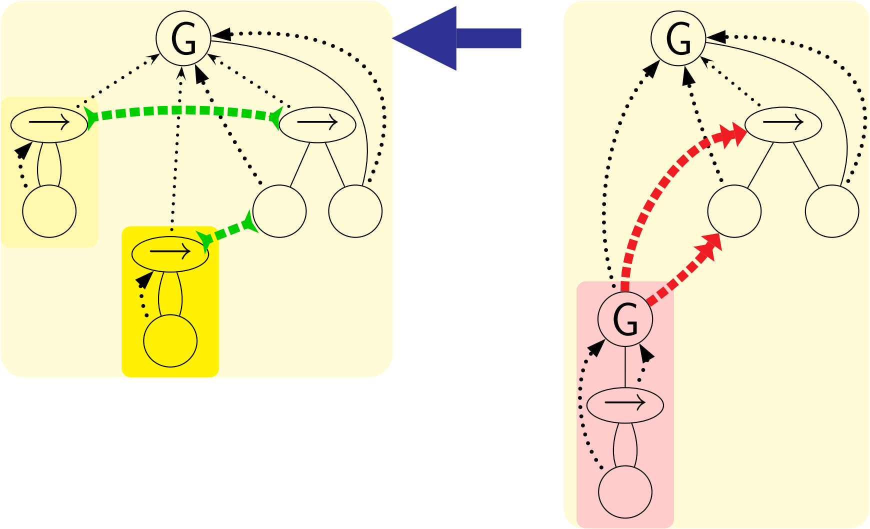
skip back



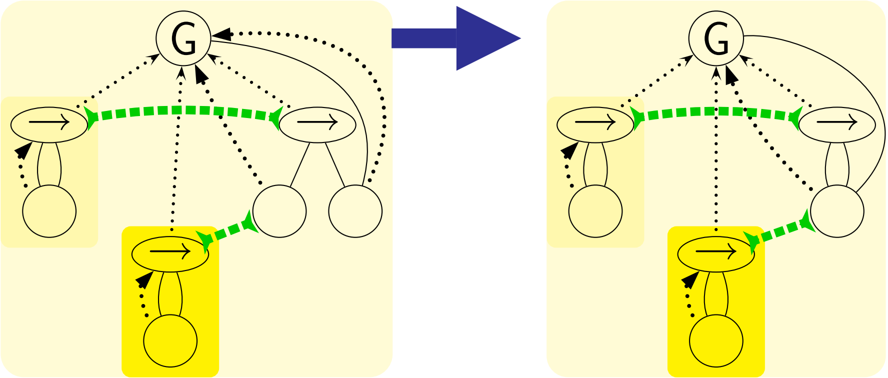
Type inference with typing constraints (demo)



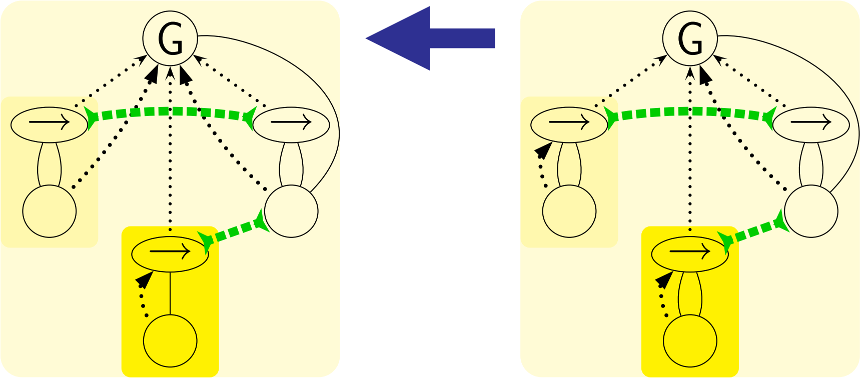
Type inference with typing constraints (demo)



Type inference with typing constraints (demo)

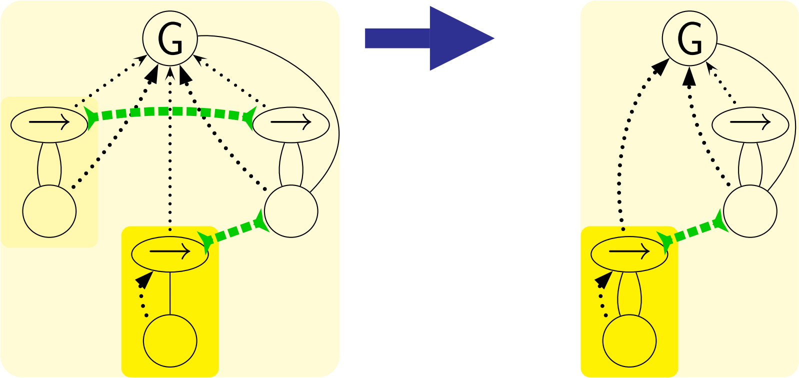


Type inference with typing constraints (demo)

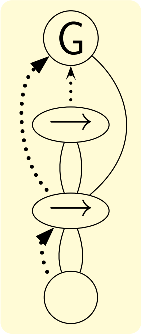


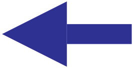
Type inference with typing constraints (demo)

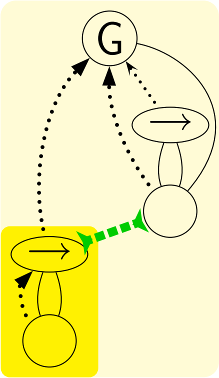
▸▸ skip ◂ back



Type inference with typing constraints (demo)

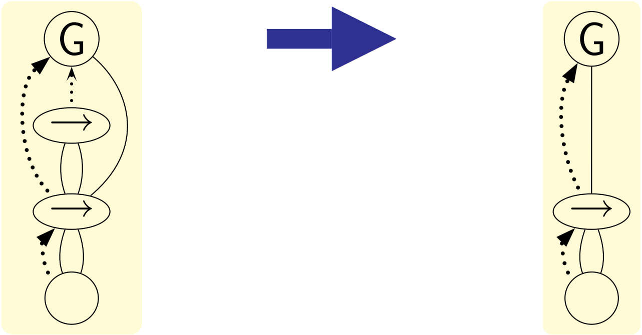






Type inference with typing constraints (demo)

skip back



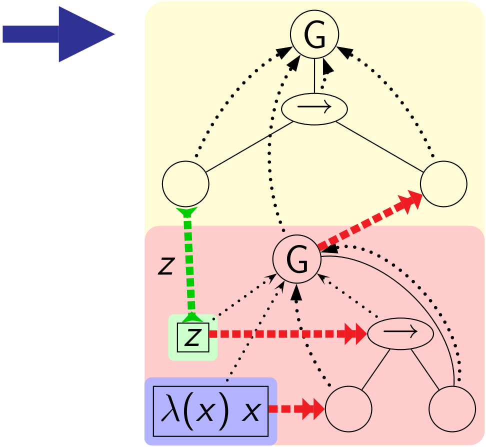
Type inference with typing constraints (demo)

skip back

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

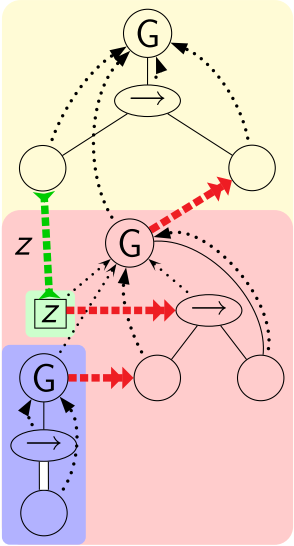
Type inference with typing constraints (demo)

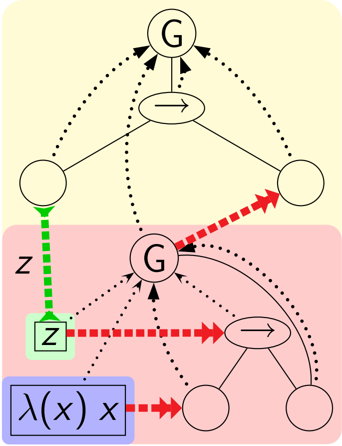
$\boxed{\lambda(z)\ z\ (\lambda(x)\ x)}$



Type inference with typing constraints (demo)

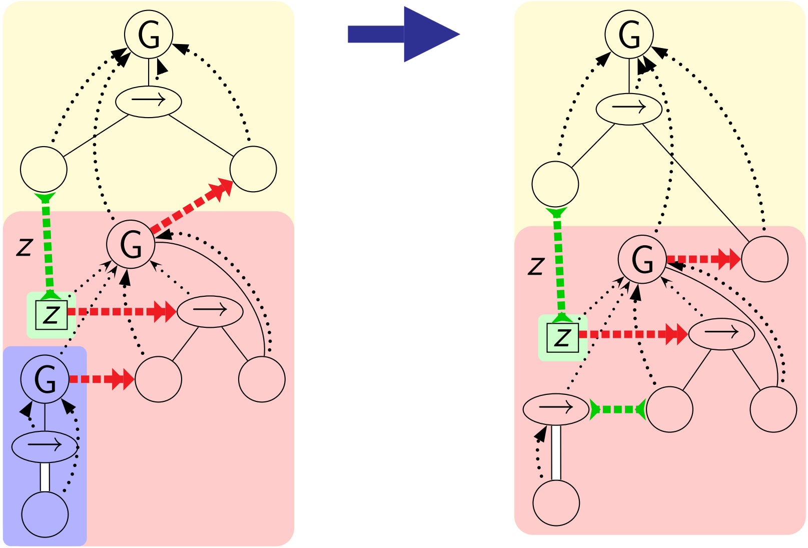
skip back



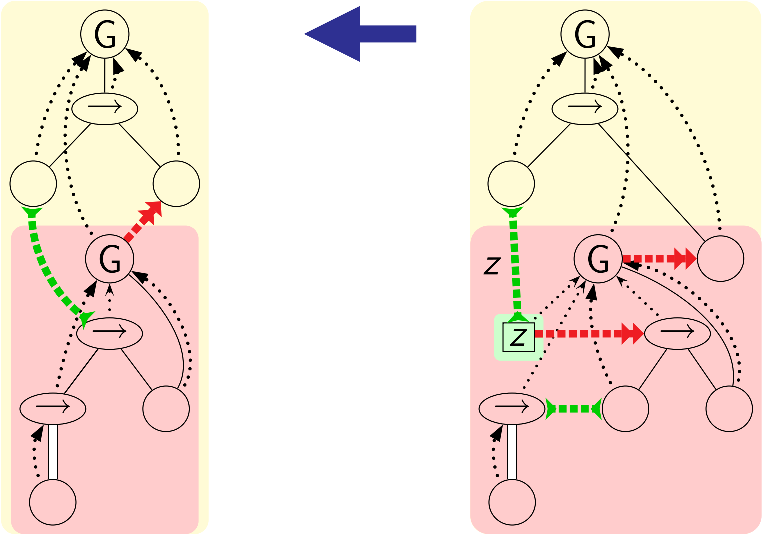


Type inference with typing constraints (demo)

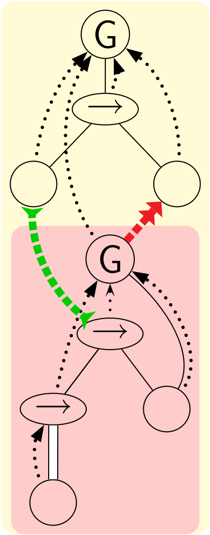
▸▸ skip ◂ back

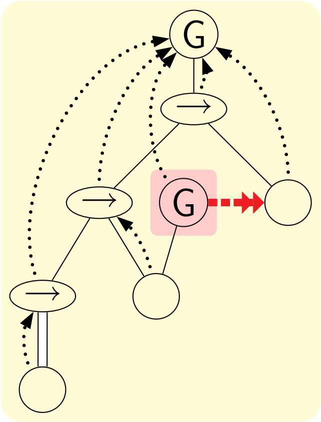


Type inference with typing constraints (demo)

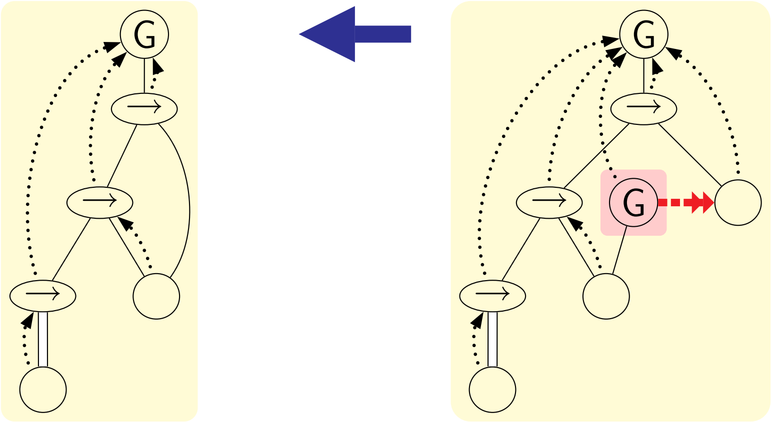


Type inference with typing constraints (demo)





Type inference with typing constraints (demo)



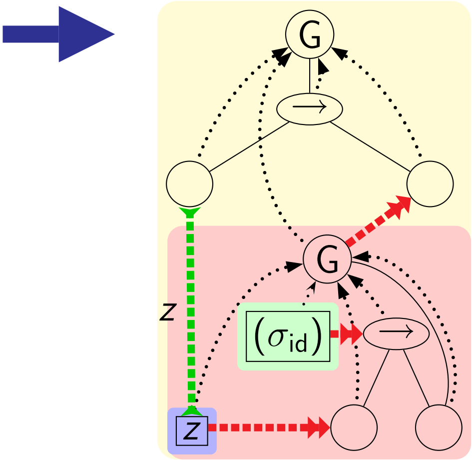
Type inference with typing constraints (demo)

skip back

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

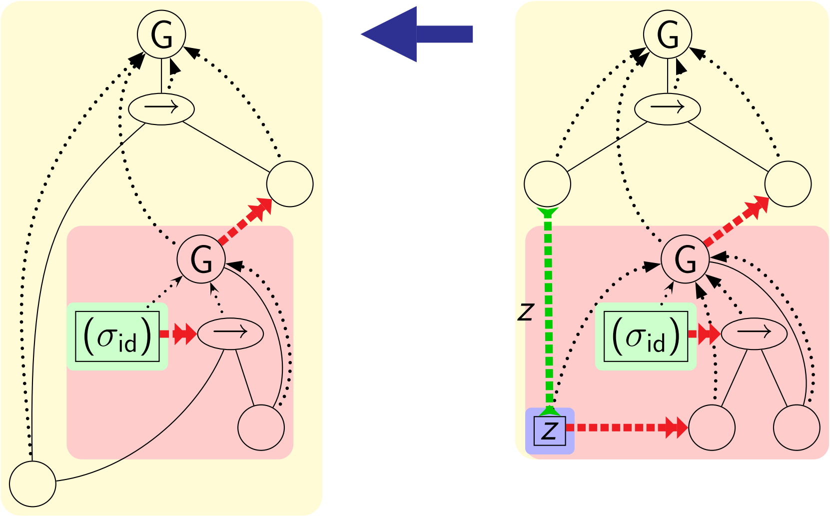
Type inference with typing constraints (demo)

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



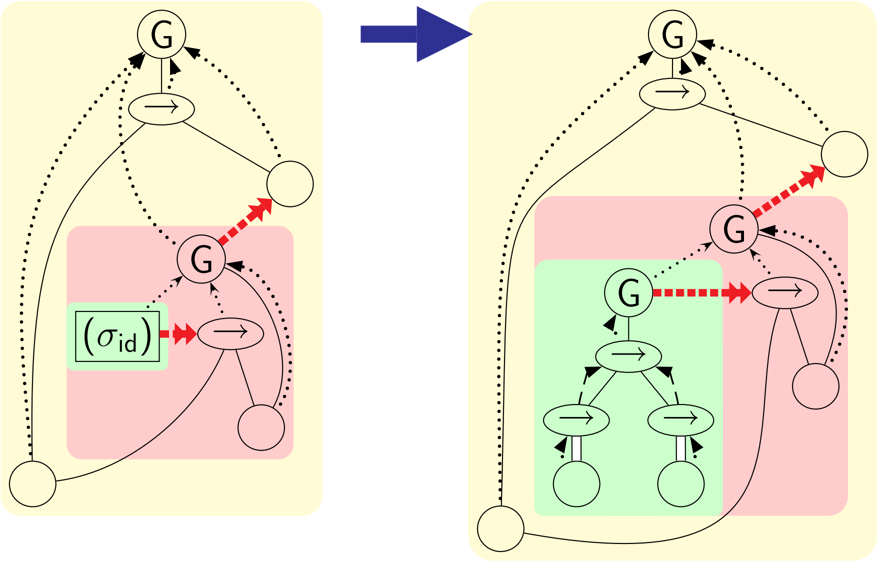
Type inference with typing constraints (demo)

skip back



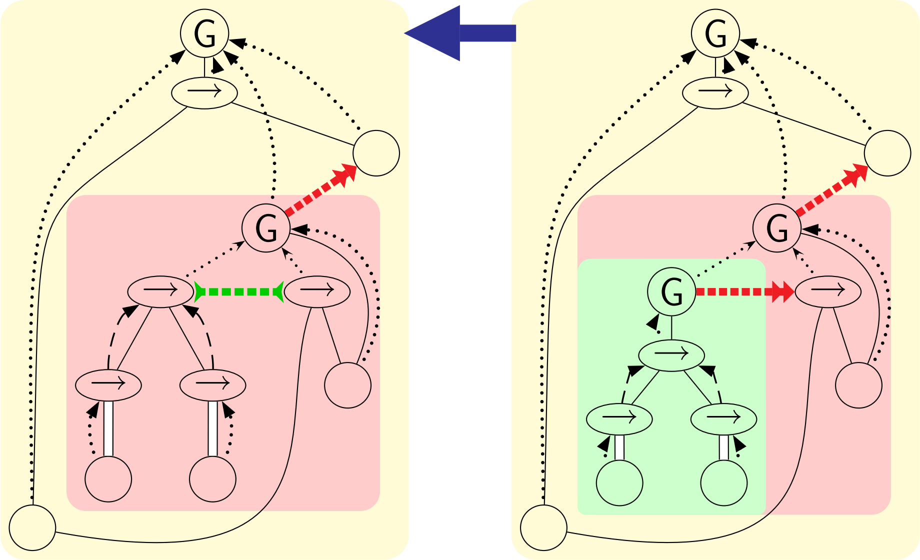
Type inference with typing constraints (demo)

skip back

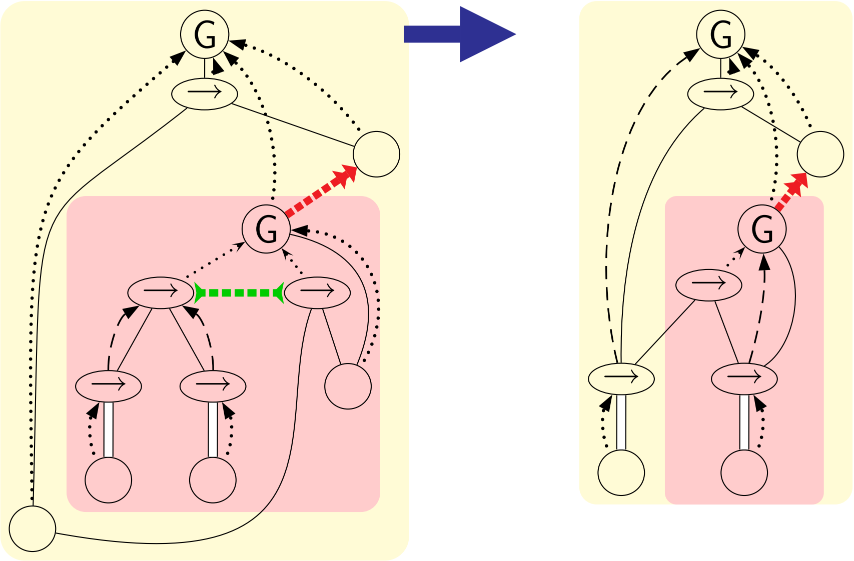


Type inference with typing constraints (demo)

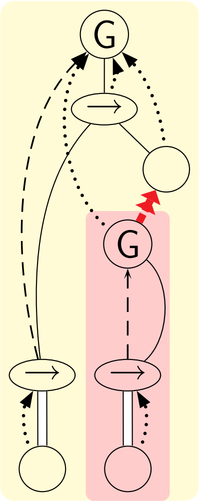
skip back

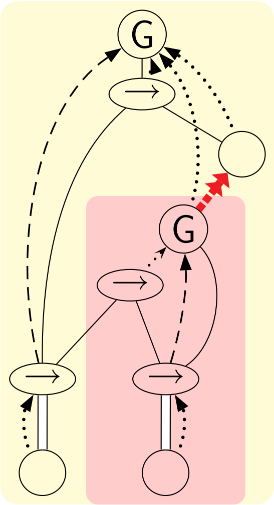


Type inference with typing constraints (demo)



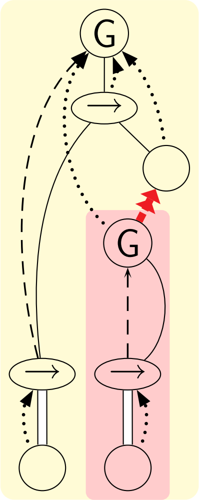
Type inference with typing constraints (demo)

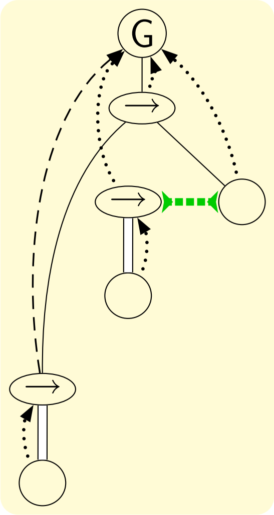




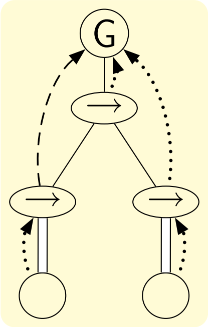
Type inference with typing constraints (demo)

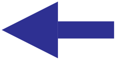
▸▸ skip ◂ back

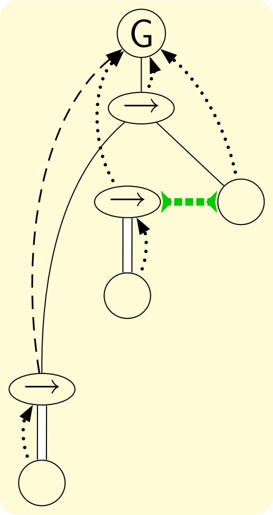




Type inference with typing constraints (demo)







◀ back