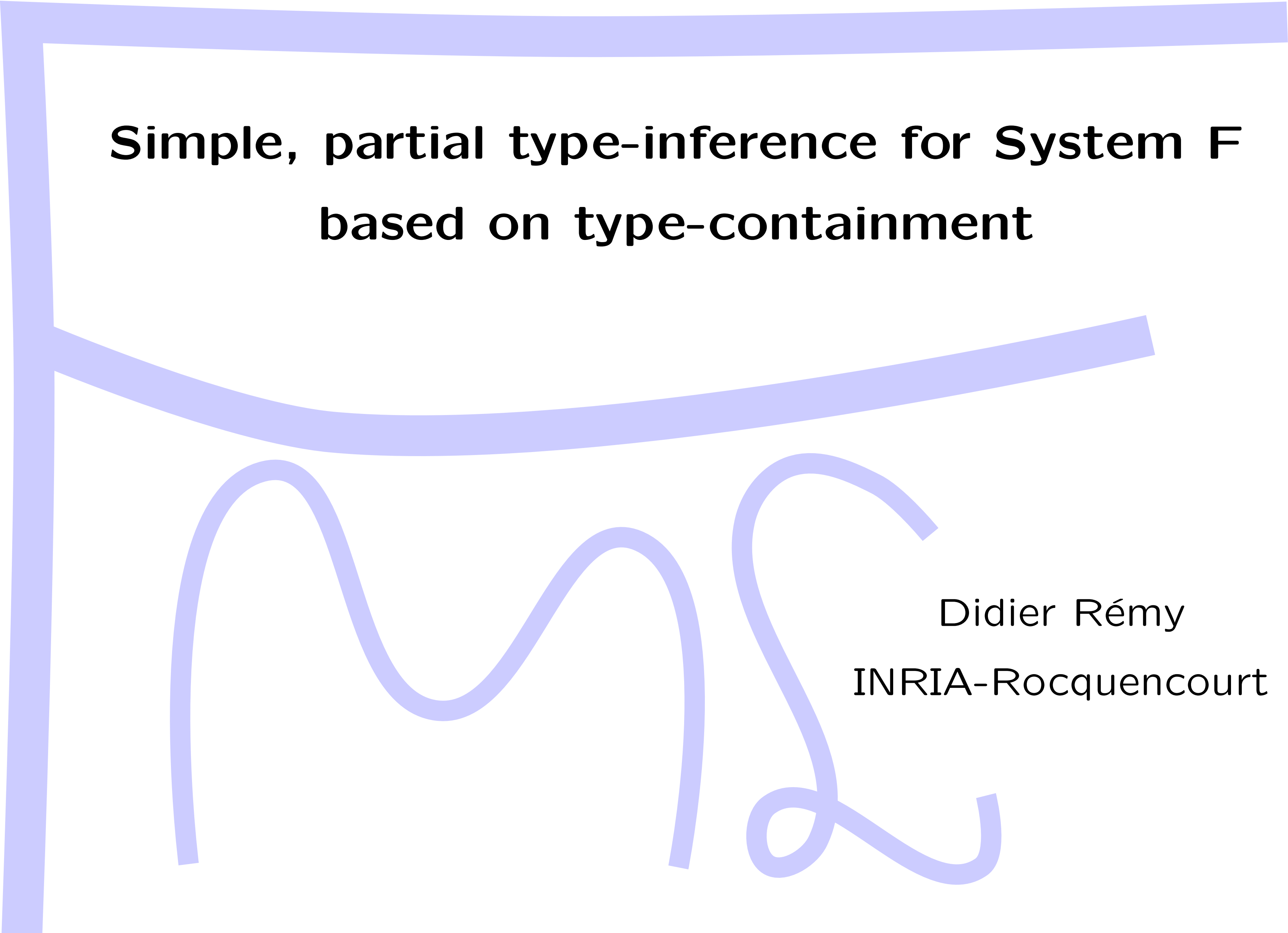
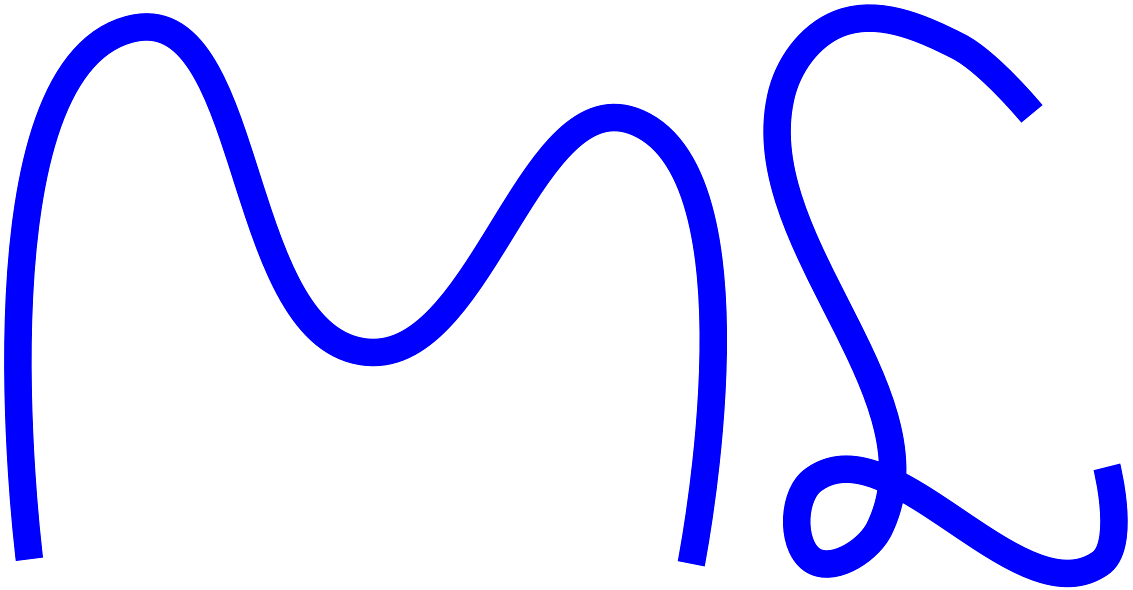


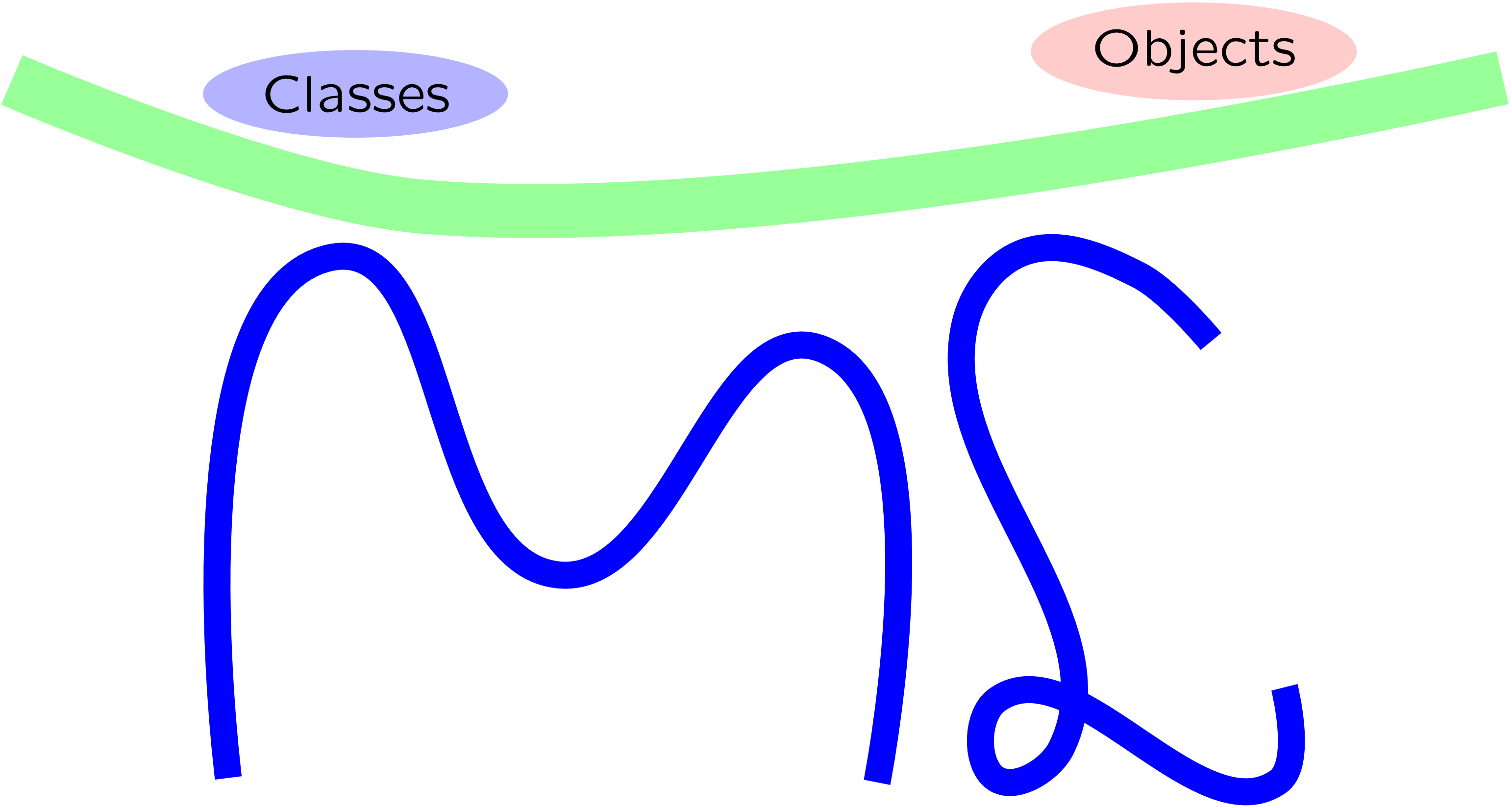
Simple, partial type-inference for System F based on type-containment

Didier Rémy
INRIA-Rocquencourt

ML is simple



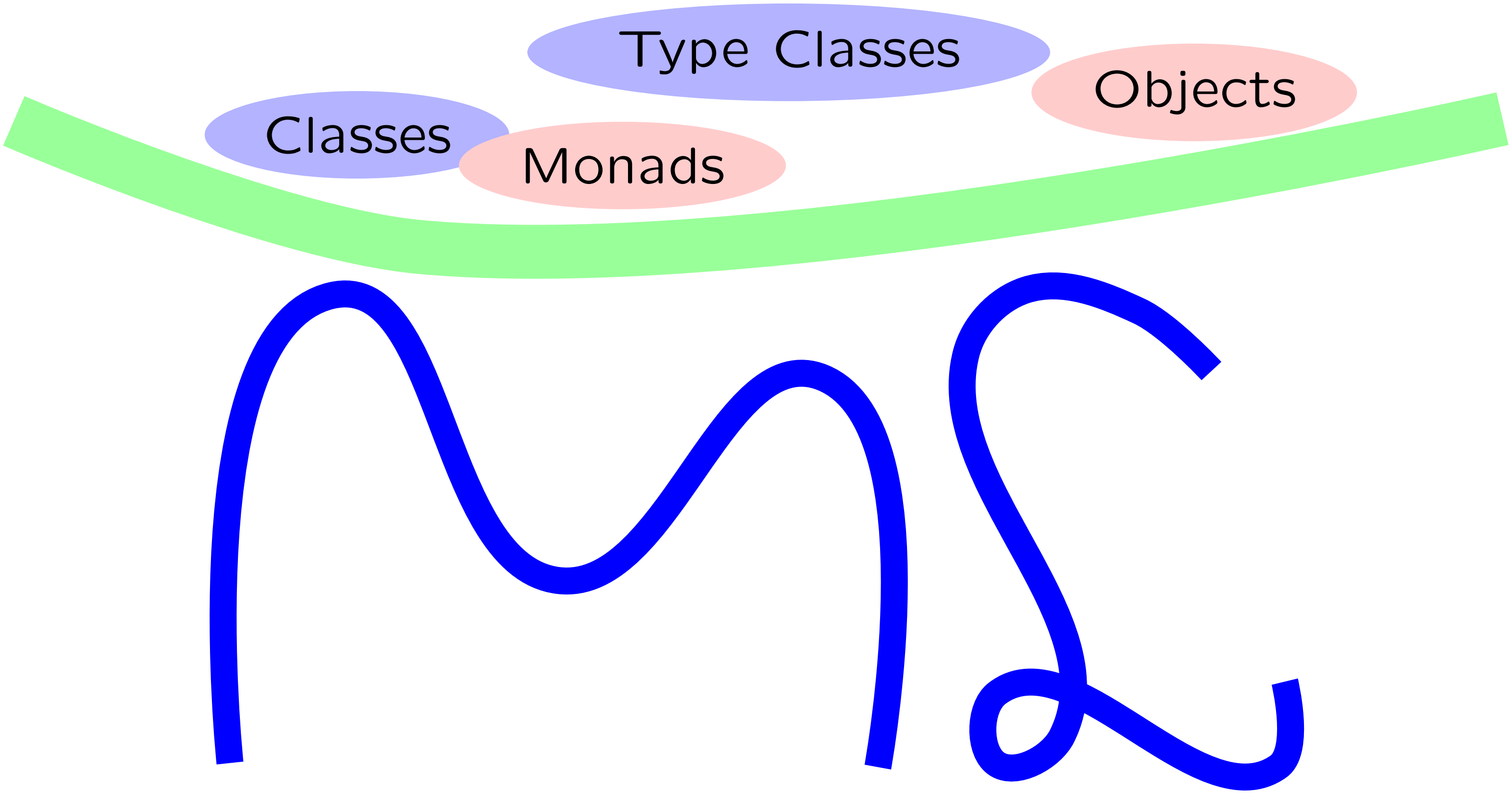
ML is simple



Classes

Objects

ML is simple, yet expressive



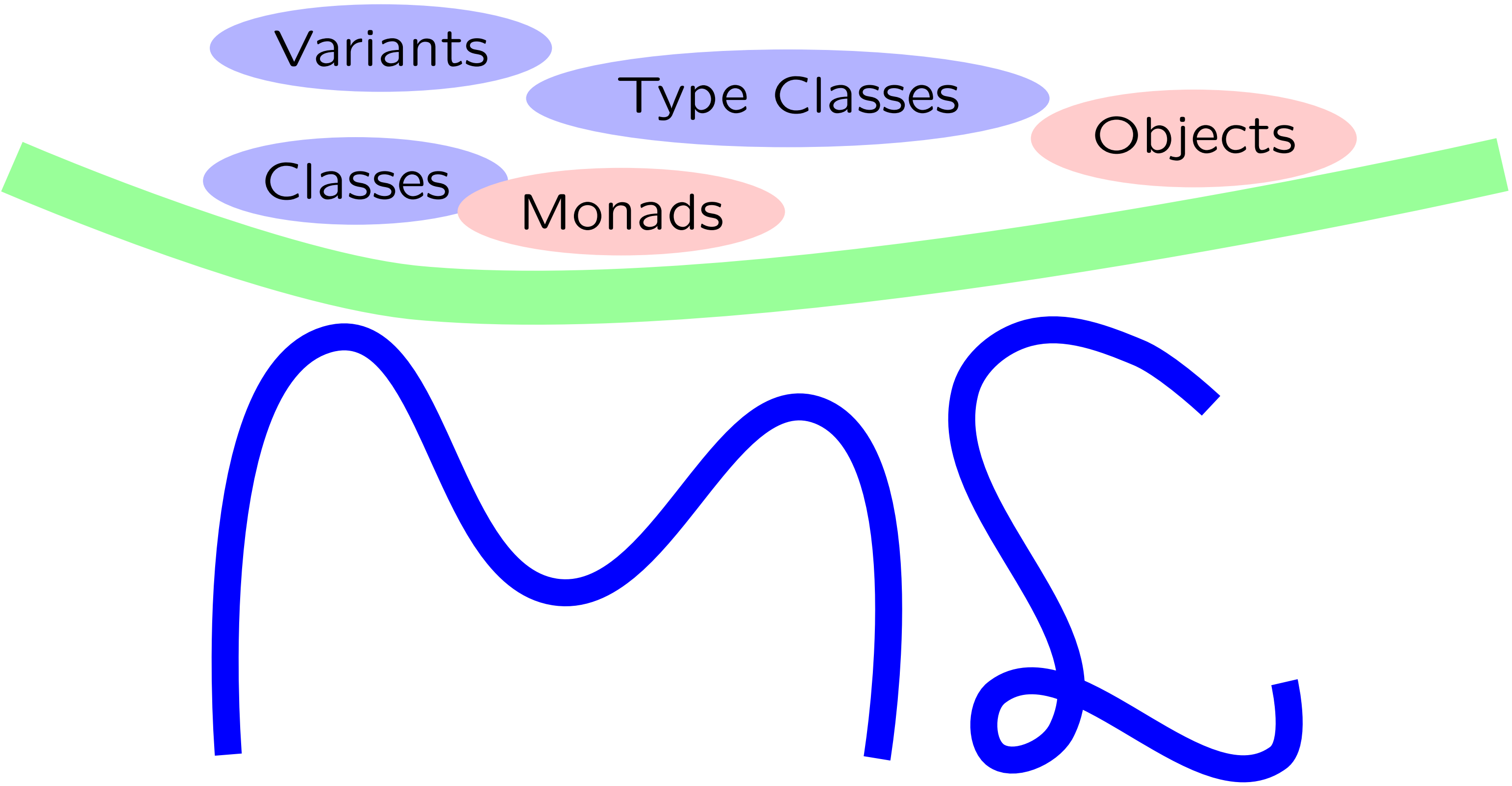
Type Classes

Objects

Classes

Monads

ML is simple, yet expressive and robust



Variants

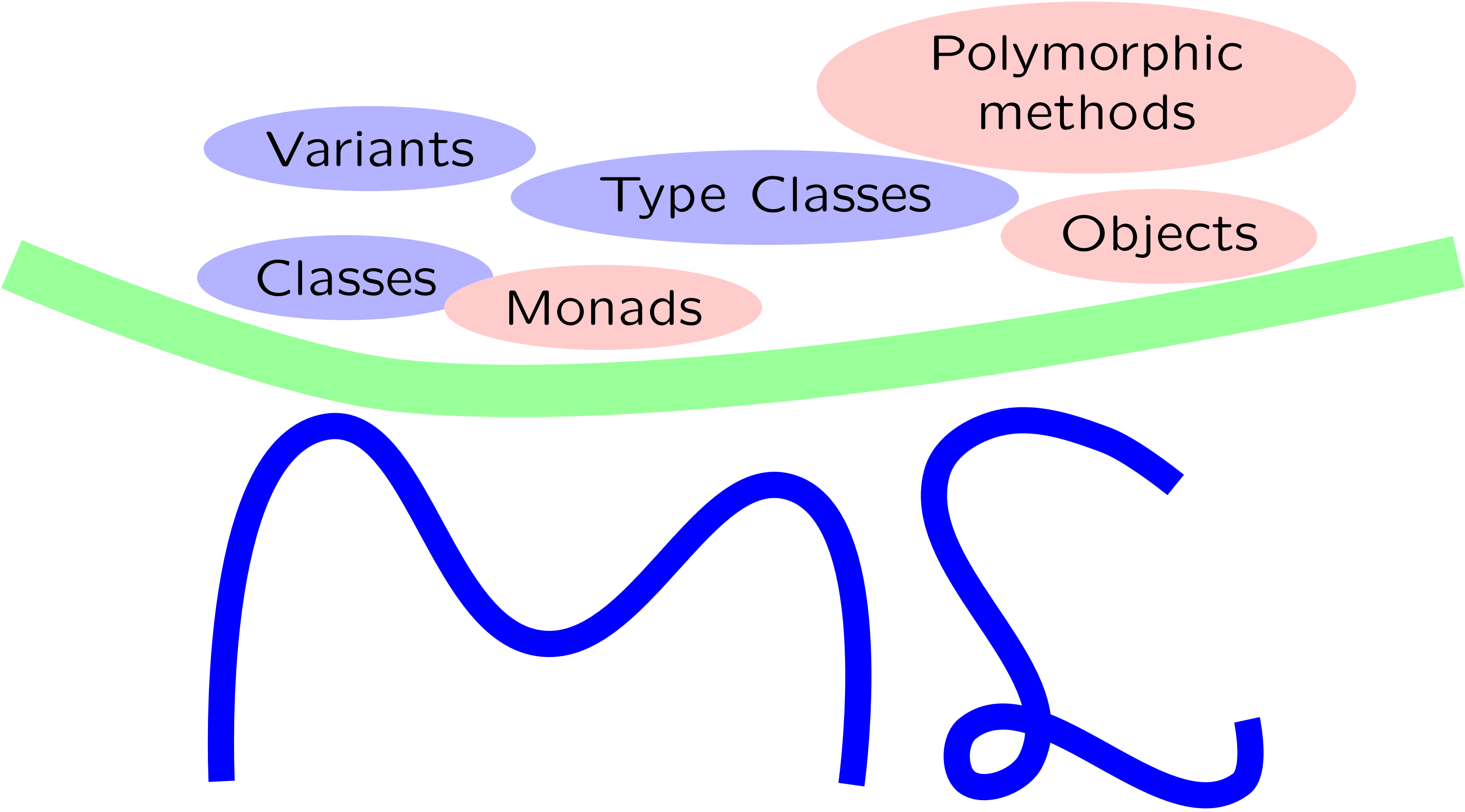
Type Classes

Objects

Classes

Monads

ML is aging...



Polymorphic methods

Variants

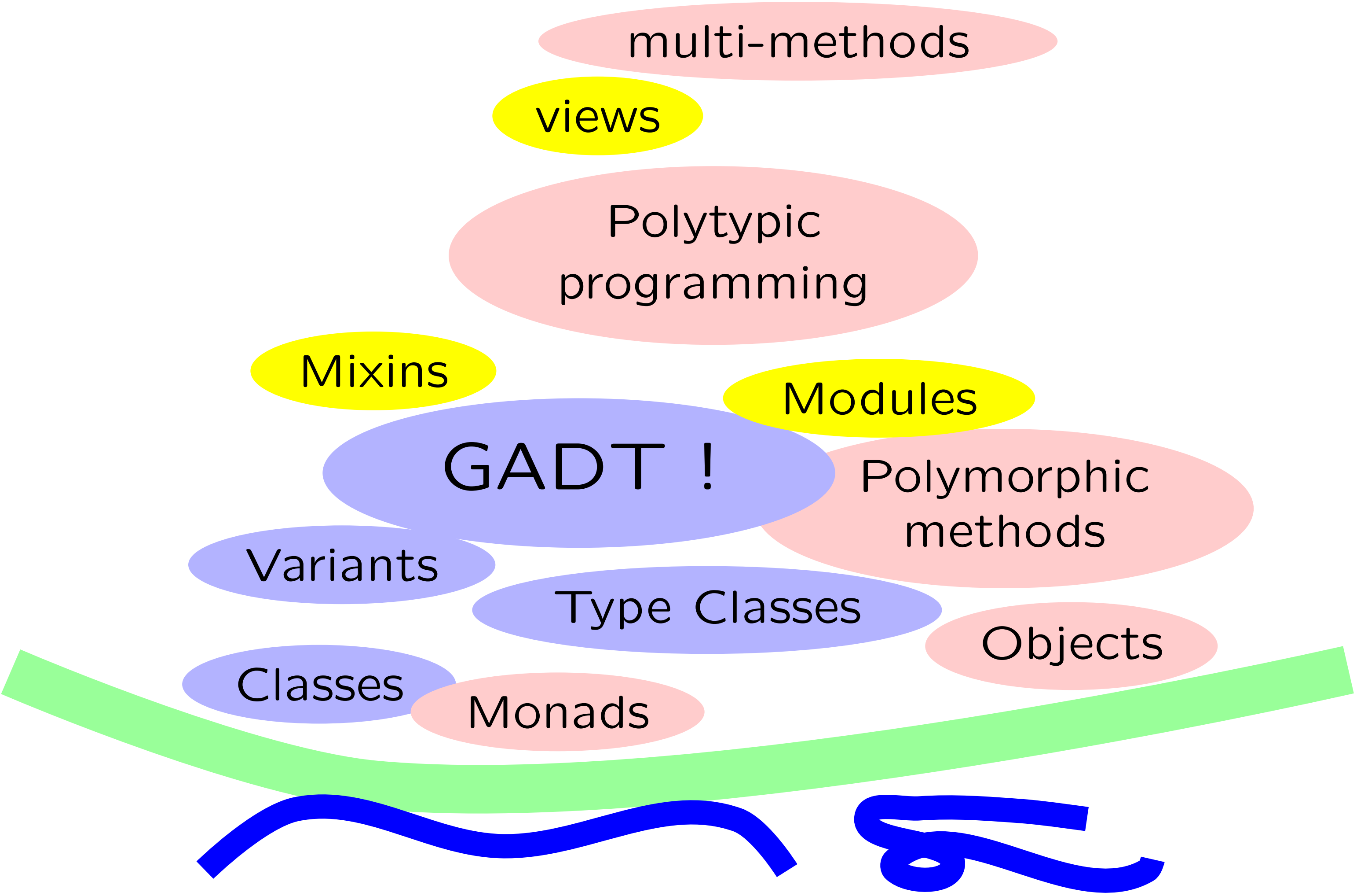
Type Classes

Objects

Classes

Monads

ML is cracking under extensions



multi-methods

views

Polytypic programming

Mixins

Modules

GADT !

Polymorphic methods

Variants

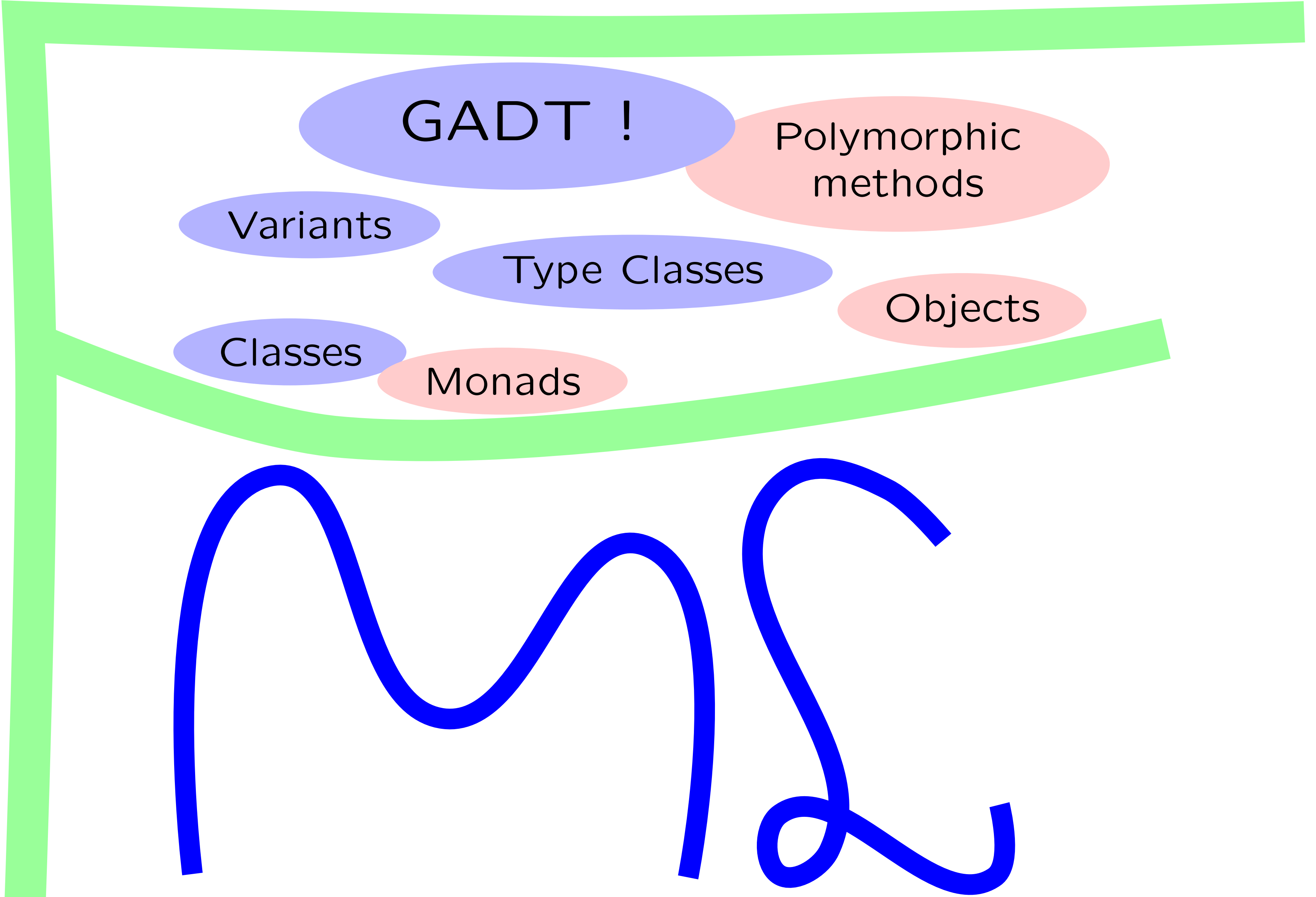
Type Classes

Objects

Classes

Monads

ML needs a lifting...



GADT !

Polymorphic methods

Variants

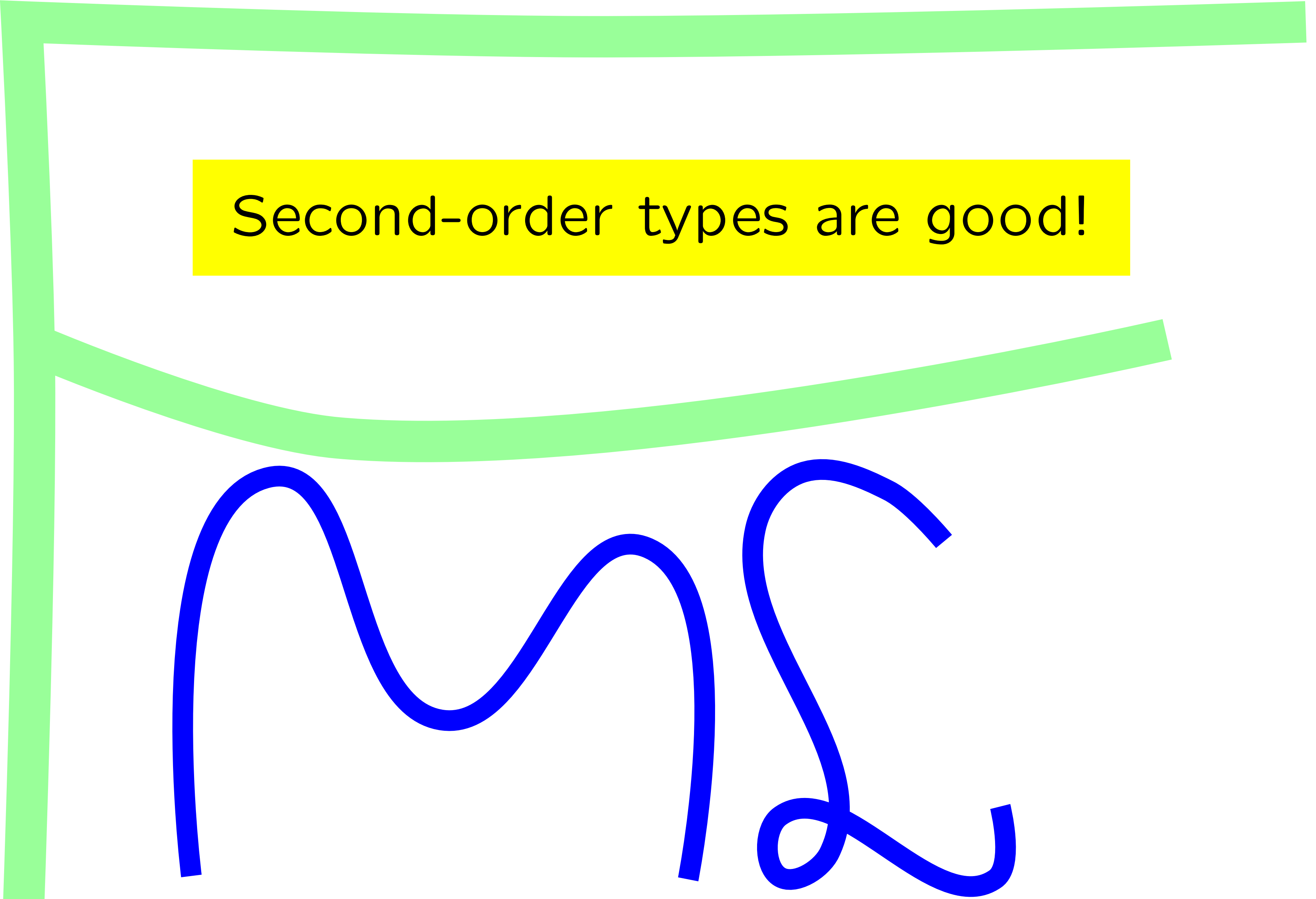
Type Classes

Objects

Classes

Monads

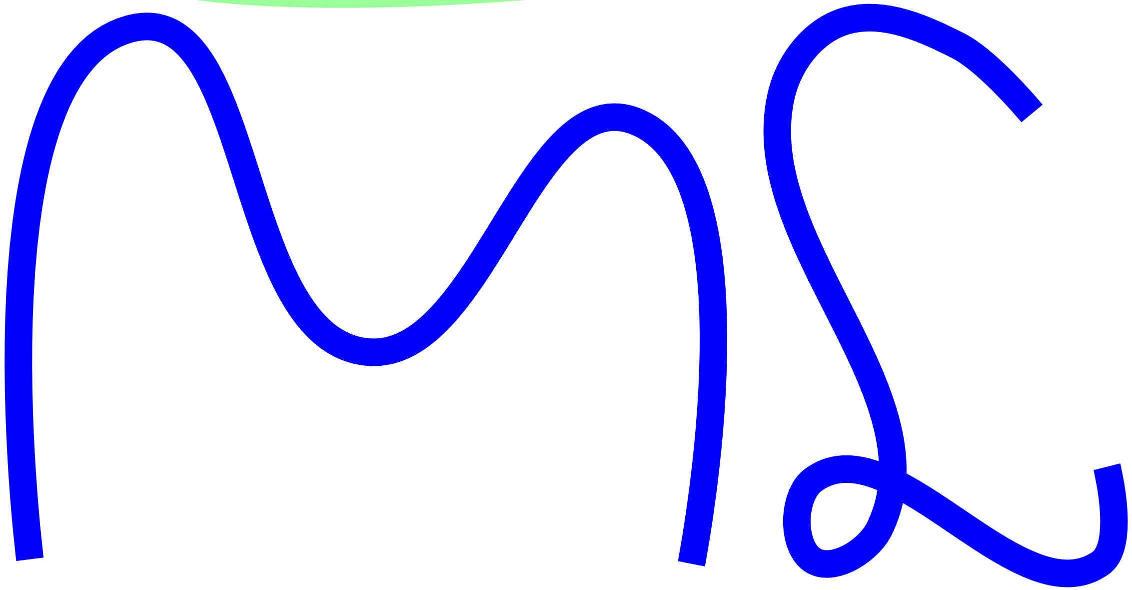
ML needs a lifting...



Second-order types are good!

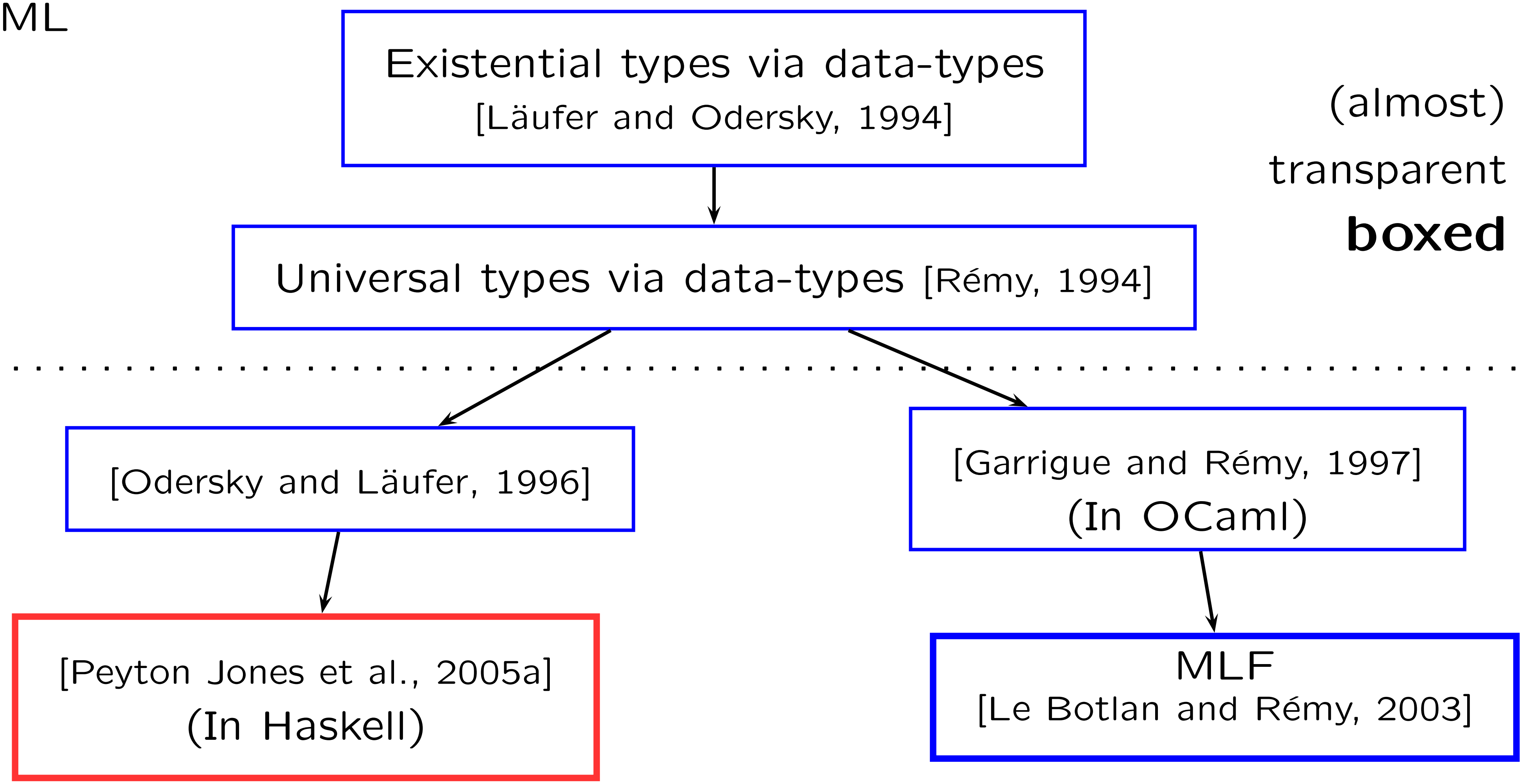
However...

Type inference is also good!



From ML toward system F

ML



Existential types via data-types
[Läufer and Odersky, 1994]

(almost)
transparent
boxed

Universal types via data-types [Rémy, 1994]

[Odersky and Läufer, 1996]

[Garrigue and Rémy, 1997]
(In OCaml)

[Peyton Jones et al., 2005a]
(In Haskell)

MLF
[Le Botlan and Rémy, 2003]

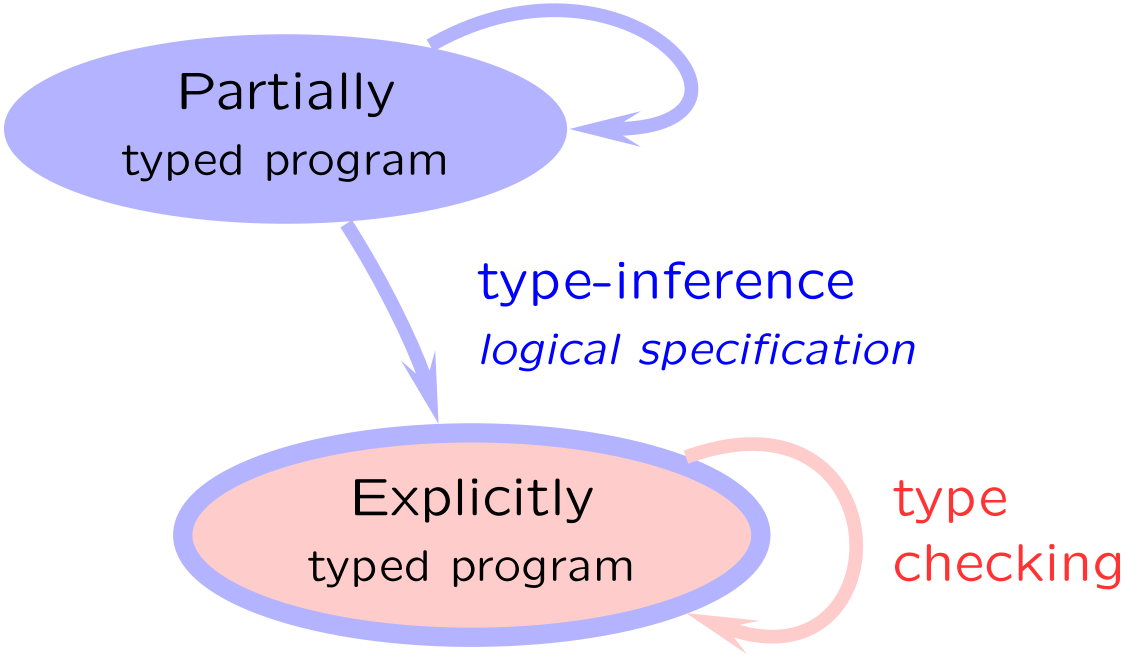
Predicative... *v.s.* ...Impredicative

Checking, Inference, and Elaboration

Explicitly
typed program

type
checking

Checking, Inference, and Elaboration



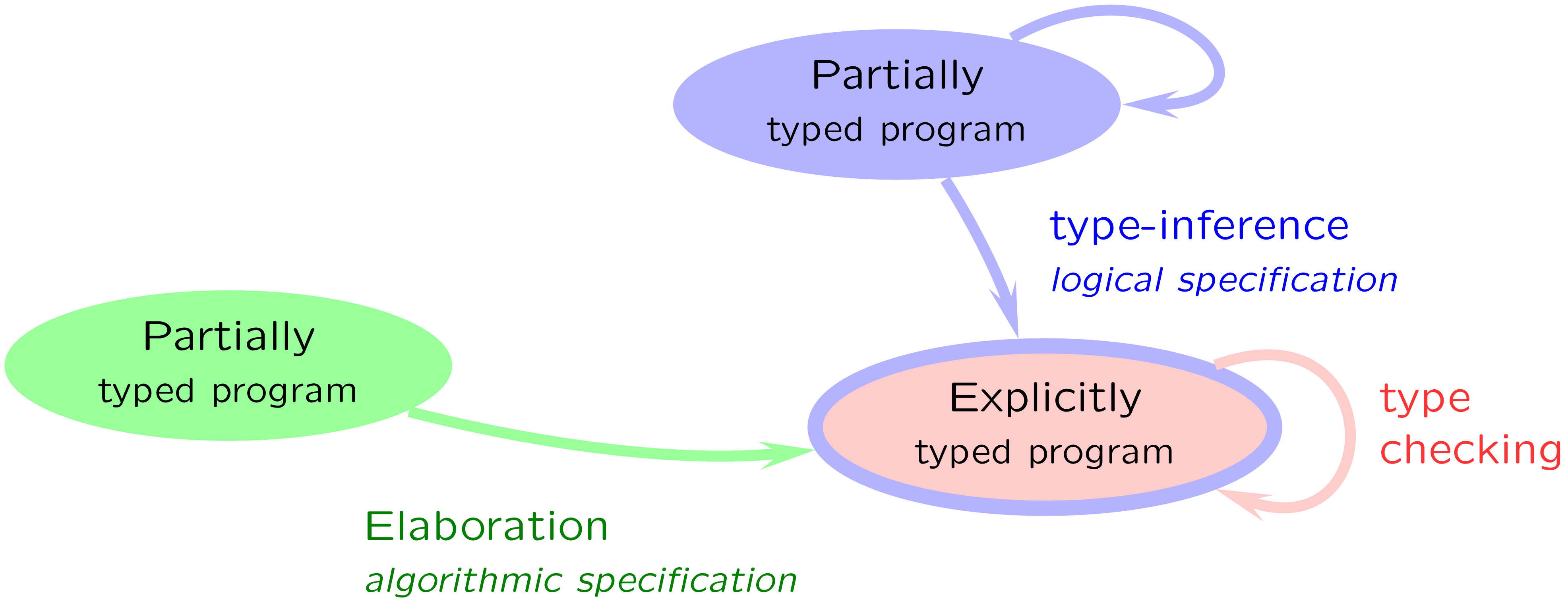
Partially
typed program

type-inference
logical specification

Explicitly
typed program

type
checking

Checking, Inference, and Elaboration



Partially
typed program

type-inference
logical specification

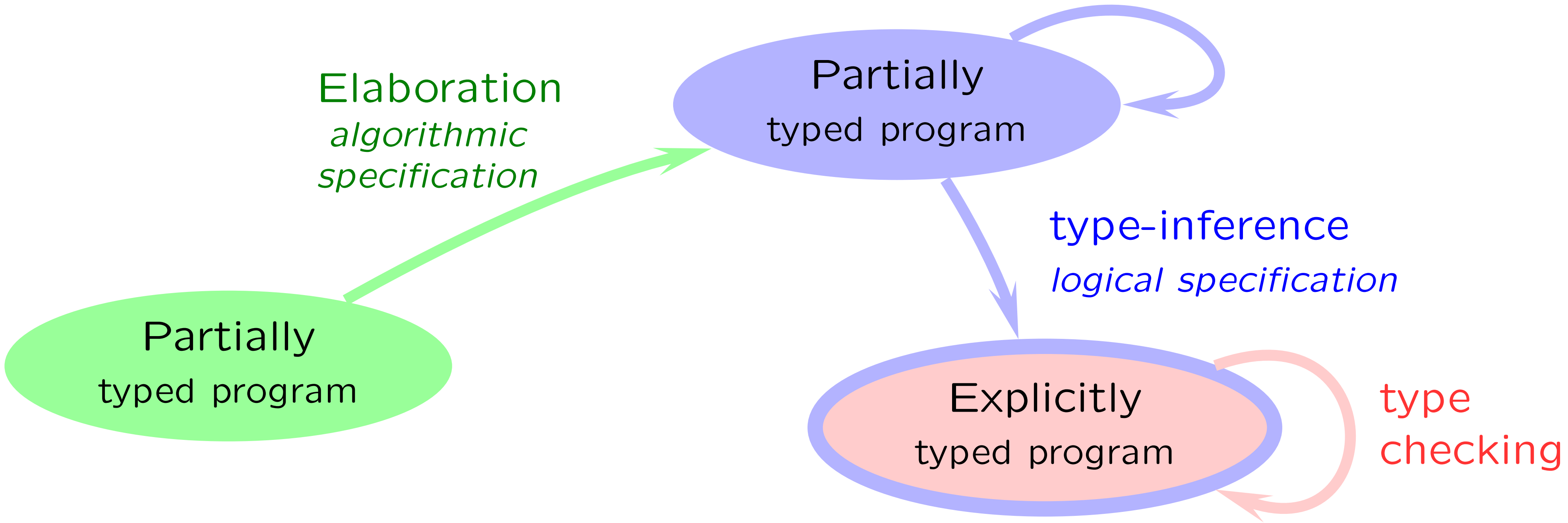
Partially
typed program

Explicitly
typed program

type
checking

Elaboration
algorithmic specification

Checking, Inference, and Elaboration



Elaboration
algorithmic specification

Partially
typed program

Partially
typed program

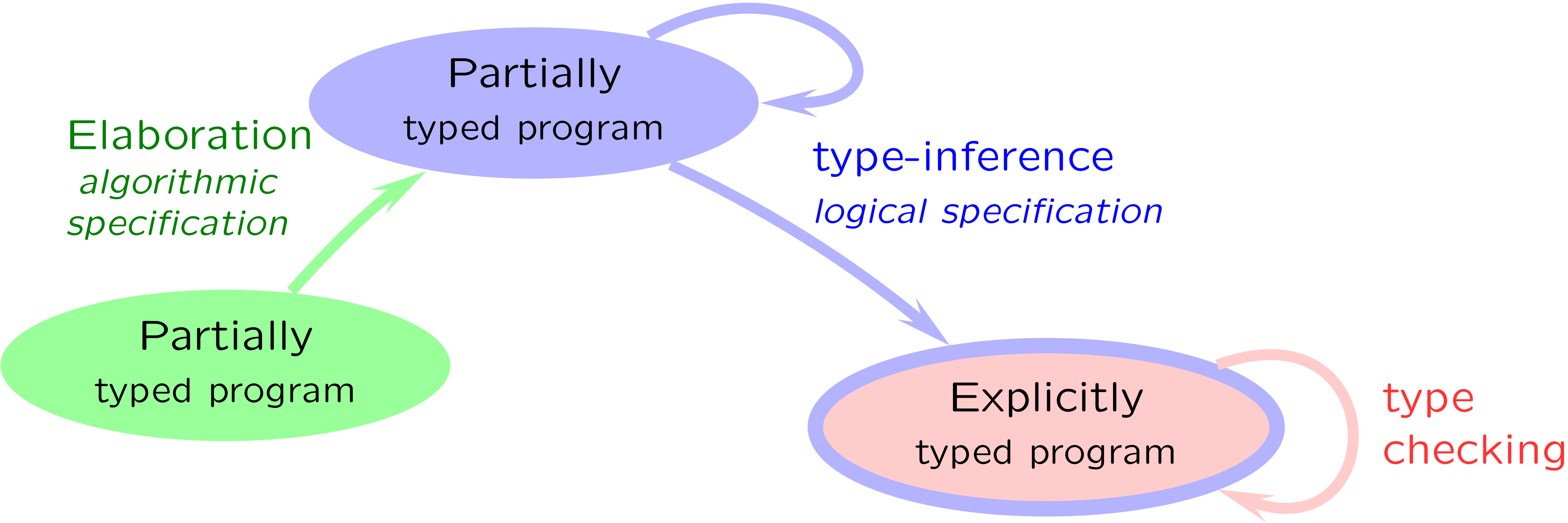
type-inference
logical specification

Explicitly
typed program

type
checking

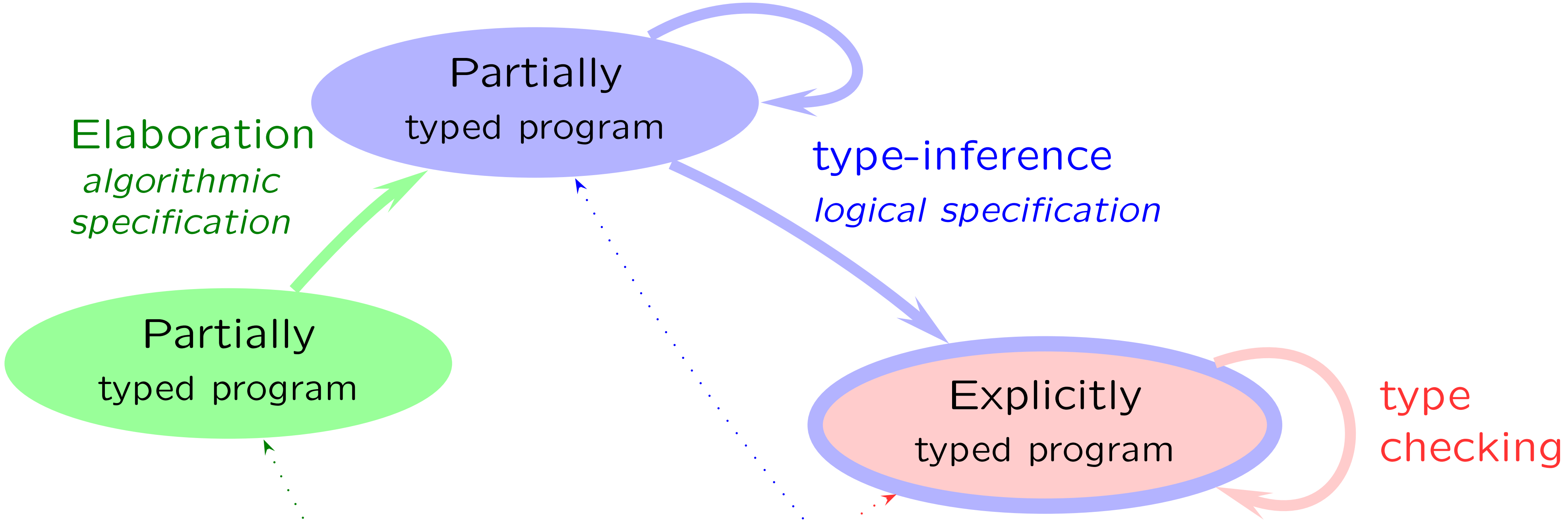
Stratified type inference

Checking, Inference, and Elaboration



Stratified type inference

Checking, Inference, and Elaboration



Partially typed program

Elaboration
algorithmic specification

Partially typed program

type-inference
logical specification

Explicitly typed program

type checking

This work

- A reference target language F($\leqslant$) (for some instance relation $\leqslant$).
- A robust, partially typed core language $\mathrm{CF}_{\mathrm{ML}}$ with complete type inference.
- A surface language $\mathrm{SF}_{\mathrm{ML}}$ with its elaboration into $\mathrm{CF}_{\mathrm{ML}}$.

Type inference in System F

Assume

let $choose = \lambda$z y . if ... then z else y	$\forall\,\alpha.\,\alpha \to \alpha \to \alpha$
let $id = \lambda$z . z	$\forall\,\beta.\,\beta \to \beta$

The instantiate-generalize problem

$choose\ id$	?
▷ instantiate id, generalize the result?	$\forall\,\beta.\,\big((\beta \to \beta) \to (\beta \to \beta)\big)$
▷ keep id polymorphic?	$(\forall\,\beta.\,\beta \to \beta) \to (\forall\,\beta.\,\beta \to \beta)$

No principal type! (no one is better than the other)

▷ concisely describe all possible types?	**difficult...**
▷ make such situations illegal?	
▷ make one case ill-typed?	
▷ pick one type and be incomplete?	**unsatisfactory**

Typing rules for implicit System F(X)

Terms: $\mathsf{z} \mid \lambda \mathsf{z}.\,\mathsf{t} \mid \mathsf{t}\ \mathsf{t}$ Types: $\alpha \mid \sigma \to \sigma \mid \forall\,\alpha.\,\sigma$

Var

$$\frac{\mathsf{z} : \sigma \in \Gamma}{\Gamma \vdash \mathsf{z} : \sigma}$$

App

$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_2 \to \sigma_1 \qquad \Gamma \vdash \mathsf{t}_2 : \sigma_2}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \sigma_1}$$

Fun

$$\frac{\Gamma, \mathsf{z} : \sigma \vdash \mathsf{t} : \sigma'}{\Gamma \vdash \lambda \mathsf{z}.\,\mathsf{t} : \sigma \to \sigma'}$$

Gen

$$\frac{\Gamma \vdash \mathsf{t} : \sigma \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \mathsf{t} : \forall\,\alpha.\,\sigma}$$

Inst

$$\frac{\Gamma \vdash \mathsf{t} : \sigma \qquad \sigma \le_X \sigma'}{\Gamma \vdash \mathsf{t} : \sigma'}$$

- Simple specification,
- Parameterized by an instance relation $\le_X$, called type containment [Mitchell, 1988] .

Type-containment $\leq$

$\leq$ for System F

The smallest relation $\leqslant$ that satisfies the rules:

Sub

$$\frac{\bar\beta \notin \mathsf{ftv}(\forall\,\bar\alpha.\,\sigma)}{\forall\,\bar\alpha.\,\sigma \leqslant \forall\,\bar\beta.\,\sigma[\bar\sigma/\bar\alpha]}$$

Type-containment $\leq^\eta$ [Mitchell, 1988]

$\leq^\eta$ for System F$^\eta$ $\quad = \quad$ System F modulo η-expansion.

The smallest relation $\leqslant$ that satisfies the rules:

Sub

$$\frac{\bar\beta \notin \mathsf{ftv}(\forall\,\bar\alpha.\,\sigma)}{\forall\,\bar\alpha.\,\sigma \leqslant \forall\,\bar\beta.\,\sigma[\bar\sigma/\bar\alpha]}$$

Trans

$$\frac{\sigma \leqslant \sigma' \qquad \sigma' \leqslant \sigma''}{\sigma \leqslant \sigma''}$$

Arrow

$$\frac{\sigma'_1 \leqslant \sigma_1 \qquad \sigma_2 \leqslant \sigma'_2}{\sigma_1 \to \sigma_2 \leqslant \sigma'_1 \to \sigma'_2}$$

All

$$\frac{\sigma \leqslant \sigma'}{\forall\,\alpha.\,\sigma \leqslant \forall\,\alpha.\,\sigma'}$$

Distrib

$$\forall\,\alpha.\,\sigma \to \sigma' \leqslant (\forall\,\alpha.\,\sigma) \to \forall\,\alpha.\,\sigma'$$

Type-containment $\leq^\eta$ [Mitchell, 1988]

$\leq^\eta$ for System F$^\eta$ = System F modulo η-expansion.

The smallest relation $\leqslant$ that satisfies the rules:

Sub

$$\frac{\bar\beta \notin \mathsf{ftv}(\forall\,\bar\alpha.\,\sigma)}{\forall\,\bar\alpha.\,\sigma \leqslant \forall\,\bar\beta.\,\sigma[\bar\sigma/\bar\alpha]}$$

Trans

$$\frac{\sigma \leqslant \sigma' \qquad \sigma' \leqslant \sigma''}{\sigma \leqslant \sigma''}$$

Arrow

$$\frac{\sigma_1' \leqslant \sigma_1 \qquad \sigma_2 \leqslant \sigma_2'}{\sigma_1 \to \sigma_2 \leqslant \sigma_1' \to \sigma_2'}$$

All

$$\frac{\sigma \leqslant \sigma'}{\forall\,\alpha.\,\sigma \leqslant \forall\,\alpha.\,\sigma'}$$

Distrib

$$\forall\,\alpha.\,\sigma \to \sigma' \leqslant (\forall\,\alpha.\,\sigma) \to \forall\,\alpha.\,\sigma'$$

Distrib-Left

$$\frac{\alpha \notin \mathsf{ftv}(\sigma')}{\forall\,\alpha.\,\sigma \to \sigma' \leqslant (\forall\,\alpha.\,\sigma) \to \sigma'}$$

(derivable)

Distrib-Right

$$\frac{\alpha \notin \mathsf{ftv}(\sigma)}{\forall\,\alpha.\,\sigma \to \sigma' \leqslant \sigma \to \forall\,\alpha.\,\sigma'}$$

(reversible, hence $\equiv$)

Why our interest in type-containment?

F$(\leq^\eta)$ is better suited for type inference

Distrib

$$\forall\,\beta.(\beta \to \beta) \to (\beta \to \beta) \quad \leq^\eta \quad (\forall\,\beta.\,\beta \to \beta) \to (\forall\,\beta.\,\beta \to \beta)$$

There are more principal types (noticed by [Mitchell, 1988])

Still, many programs do not have principal types.

However

- Neither F nor F$(\leq^\eta)$ allows for type inference.
- Type-containment $\leq^\eta$ is itself undecidable.
- The problem lies in Rule Sub, which implies guessing polytypes.

Predicative type-containment $\leq_p^\eta$

Requires that type variables may only be instantiated by monotypes.

monotypes	$\tau \mid \alpha \mid \tau \to \tau$
(poly)types	$\sigma \mid \tau \mid \sigma \to \sigma \mid \forall\, \alpha.\, \sigma$

Rule Sub is replaced by its predicative version:

$$\text{Sub}_p \quad \frac{\bar\beta \notin \mathsf{ftv}(\forall\, \bar\alpha.\, \sigma)}{\forall\, \bar\alpha.\, \sigma \leqslant \forall\, \bar\beta.\, \sigma[\,\bar\tau\,/\bar\alpha]}$$

Defines $\leq_p^{\mathsf{F}}$ and $\leq_p^\eta$ (leaving all other rules unchanged).

Properties

- $\leq_p^\eta$ is decidable.
- So is solving first-order $\leq_p^\eta$ type-containment constraints.

Predicative type-containment (continued)

Checking $\leq_p^{\eta -}$

Take $\leq_p^{\eta -}$ to be $\leq_p^{\eta}$ without Rule Distrib.

The relation $\leq_p^{\eta -}$ has a simple syntax-directed presentation:

Refl
$$\sigma \leqslant \sigma$$

Arrow
$$\frac{\sigma_1 \leqslant \sigma'_1 \qquad \sigma'_2 \leqslant \sigma_2}{\sigma_2 \to \sigma_1 \leqslant \sigma'_2 \to \sigma'_1}$$

All-I
$$\frac{\sigma \leqslant \sigma' \qquad \alpha \notin \mathsf{ftv}(\sigma)}{\sigma \leqslant \forall\, \alpha.\, \sigma'}$$

All-E
$$\frac{\sigma[\tau/\alpha] \leqslant \sigma'}{\forall\, \alpha.\, \sigma \leqslant \sigma'}$$

Checking $\leq_p^{\eta}$

Let $\mathsf{prf}\,(\sigma)$ computes the prenex form of σ applying the rewrite rule:

$$\sigma' \to \forall\, \alpha.\, \sigma \;\rightsquigarrow\; \forall\, \alpha.\, \sigma' \to \sigma$$

Theorem [Peyton Jones et al., 2005b]

The relation $\sigma \leq_p^{\eta} \sigma'$ may be computed as $\mathsf{prf}\,(\sigma) \leq_p^{\eta -} \mathsf{prf}\,(\sigma')$.

BTW, Rule Distrib-Right, hence $\leq_p^{\eta}$, is not sound with side effects.

Partial type inference in $\mathsf{CF}_{\mathsf{ML}}$

Goal

Use type inference *a la ML* with predicative instantiation $\leq_p^{\eta-}$, and annotations whenever necessary, to reach all programs of $\mathsf{F}(\leq_p^{\eta-})$.

Expressions

$$\mathsf{t} ::= \mathsf{x} \mid \lambda \mathsf{z} \,.\, \mathsf{t} \mid \mathsf{t}_1 \quad \mathsf{t}_2 \qquad \mid \mathsf{let}\ \mathsf{z} = \ \mathsf{t}_1 \qquad \mathsf{in}\ \mathsf{t}_2$$

Partial type inference in $\mathsf{CF}_{\mathsf{ML}}$

Goal

Use type inference *a la ML* with predicative instantiation $\leq_p^{\eta-}$, and annotations whenever necessary, to reach all programs of $\mathsf{F}(\leq_p^{\eta-})$.

Expressions

$$\mathsf{t} ::= \mathsf{x} \mid \lambda \mathsf{z} \,.\, \mathsf{t} \mid \mathsf{t}_1\ (\mathsf{t}_2 : \theta) \mid \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \theta)\ \mathsf{in}\ \mathsf{t}_2$$

Annotations

$$\theta ::= \exists\, \bar{\beta}.\, \sigma$$

- σ explicitly specify the polymorphic shape of types.
- $\bar{\beta}$ let type inference guess the monomorphic parts.
- The monomorphic structure is always hanging off under some (possibly empty) polymorphic structure.
- Annotations are mandatory, but may be the empty annotation $\exists\, \beta.\, \beta$.

Typing rules for **ML**

Var

$$\frac{z : \sigma \in \Gamma}{\Gamma \vdash z : \sigma}$$

Inst

$$\frac{\Gamma \vdash t : \sigma' \qquad \sigma' \leq_{ML} \sigma}{\Gamma \vdash t : \sigma}$$

Gen

$$\frac{\Gamma \vdash t : \sigma \qquad \bar{\alpha} \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash t : \forall \bar{\alpha}.\, \sigma}$$

Fun

$$\frac{\Gamma, z : \tau_2 \vdash t : \tau_1}{\Gamma \vdash \lambda z . t : \tau_2 \to \tau_1}$$

App

$$\frac{\Gamma \vdash t_1 : \tau_2 \qquad \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$$

Let

$$\frac{\Gamma \vdash t_1 : \forall \bar{\alpha}.\, \tau_1 \qquad \Gamma, z : \forall \bar{\alpha}.\, \tau_1 \qquad \vdash t_2 : \tau_2}{\Gamma \vdash \mathrm{let}\ z = \ t_1 \qquad \mathrm{in}\ t_2 : \tau_2}$$

Typing rules for ML with annotations

Var

$$\frac{z : \sigma \in \Gamma}{\Gamma \vdash z : \sigma}$$

Inst

$$\frac{\Gamma \vdash t : \sigma' \qquad \sigma' \leq_{\mathsf{ML}} \sigma}{\Gamma \vdash t : \sigma}$$

Gen

$$\frac{\Gamma \vdash t : \sigma \qquad \bar{\alpha} \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash t : \forall \bar{\alpha}. \sigma}$$

Fun

$$\frac{\Gamma, z : \tau_2 \vdash t : \tau_1}{\Gamma \vdash \lambda z. t : \tau_2 \to \tau_1}$$

App

$$\frac{\Gamma \vdash t_1 : \tau_2[\bar{\tau}/\bar{\beta}] \to \tau \qquad \Gamma \vdash t_2 : \tau_2[\bar{\tau}/\bar{\beta}]}{\Gamma \vdash t_1\ (t_2 : \exists \bar{\beta}. \tau_2) : \tau}$$

Let

$$\frac{\Gamma \vdash t_1 : \forall \bar{\alpha}. \tau_1[\bar{\tau}/\bar{\beta}] \qquad \Gamma, z : \forall \bar{\alpha}. \tau_1[\bar{\tau}/\bar{\beta}] \vdash t_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ z = (t_1 : \exists \bar{\beta}. \tau_1)\ \mathsf{in}\ t_2 : \tau_2}$$

Typing rules for $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})$

Var

$$\frac{\mathsf{z} : \sigma \in \Gamma}{\Gamma \vdash \mathsf{z} : \sigma}$$

Inst

$$\frac{\Gamma \vdash \mathsf{t} : \sigma' \qquad \sigma' \leq_p^{\eta-} \sigma}{\Gamma \vdash \mathsf{t} : \sigma}$$

Gen

$$\frac{\Gamma \vdash \mathsf{t} : \sigma \qquad \bar{\alpha} \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \mathsf{t} : \forall \bar{\alpha}.\, \sigma}$$

Fun

$$\frac{\Gamma, \mathsf{z} : \sigma_2 \vdash \mathsf{t} : \sigma_1}{\Gamma \vdash \lambda \mathsf{z}.\, \mathsf{t} : \sigma_2 \to \sigma_1}$$

App

$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_2[\bar{\tau}/\bar{\beta}] \to \sigma \qquad \Gamma \vdash \mathsf{t}_2 : \sigma_2[\bar{\tau}/\bar{\beta}]}{\Gamma \vdash \mathsf{t}_1\ (\mathsf{t}_2 : \exists \bar{\beta}.\, \sigma_2) : \sigma}$$

Let

$$\frac{\Gamma \vdash \mathsf{t}_1 : \forall \bar{\alpha}.\, \sigma_1[\bar{\tau}/\bar{\beta}] \qquad \Gamma, \mathsf{z} : \forall \bar{\alpha}.\, \sigma_1[\bar{\tau}/\bar{\beta}] \vdash \mathsf{t}_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists \bar{\beta}.\, \sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \sigma_2}$$

Typing rules for $\mathbf{F}_{\mathbf{ML}}(\leq_p^{\eta-})$

Var-Inst

$$\frac{\mathsf{z} : \sigma' \in \Gamma \qquad \sigma' \leq_p^{\eta-} \rho}{\Gamma \vdash \mathsf{z} : \rho}$$

Gen

$$\frac{\Gamma \vdash \mathsf{t} : \sigma \qquad \bar{\alpha} \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \mathsf{t} : \forall\,\bar{\alpha}.\,\sigma}$$

Fun

$$\frac{\Gamma, \mathsf{z} : \sigma_2 \vdash \mathsf{t} : \sigma_1}{\Gamma \vdash \lambda \mathsf{z}\,.\,\mathsf{t} : \sigma_2 \to \sigma_1}$$

App-Rho

$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_2[\bar{\tau}/\bar{\beta}] \to \rho \qquad \Gamma \vdash \mathsf{t}_2 : \sigma_2[\bar{\tau}/\bar{\beta}]}{\Gamma \vdash \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar{\beta}.\,\sigma_2) : \rho}$$

Let-Gen

$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_1[\bar{\tau}/\bar{\beta}] \qquad \Gamma, \mathsf{z} : \forall \setminus \Gamma.\,\sigma_1[\bar{\tau}/\bar{\beta}] \vdash \mathsf{t}_2 : \rho_2}{\Gamma \vdash \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\bar{\beta}.\,\sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \rho_2}$$

(quasi) syntax-directed presentation.

Type inference via type constraints

See the paper

Examples in CF$_{\text{ML}}$

As in ML

$\text{let } \mathit{id} = \lambda \mathsf{z}\,.\,\mathsf{z}$

$\mathit{id}\ \ \mathit{id}$

$\text{let } \xcancel{\mathit{auto} = \lambda}\mathsf{f}\,.\,\mathsf{f}\ \mathsf{f}$

Plus...

$\text{let } \mathit{auto} = (\lambda \mathsf{f}\,.\,\mathsf{f}\ \mathsf{f} : \exists\,\beta.\,(\forall\,\alpha.\,\alpha \to \alpha) \to \beta)$

$\mathit{auto}\ (\lambda \mathsf{z}\,.\,\mathsf{z} : \forall\,\alpha.\,\alpha \to \alpha)$

$(\lambda \mathsf{f}\,.\,\mathsf{f}\ \mathsf{f})\ (\mathit{id} : \forall\,\alpha.\,\alpha \to \alpha)$

$(\lambda \mathsf{z}\,.\,\mathsf{z})\ (\mathit{auto} : \exists\,\beta.\,(\forall\,\alpha.\,\alpha \to \alpha) \to \beta)$

Elaboration

- can we get rid of the overlined annotations?
- propagate source annotations *before* typechecking

Shapes

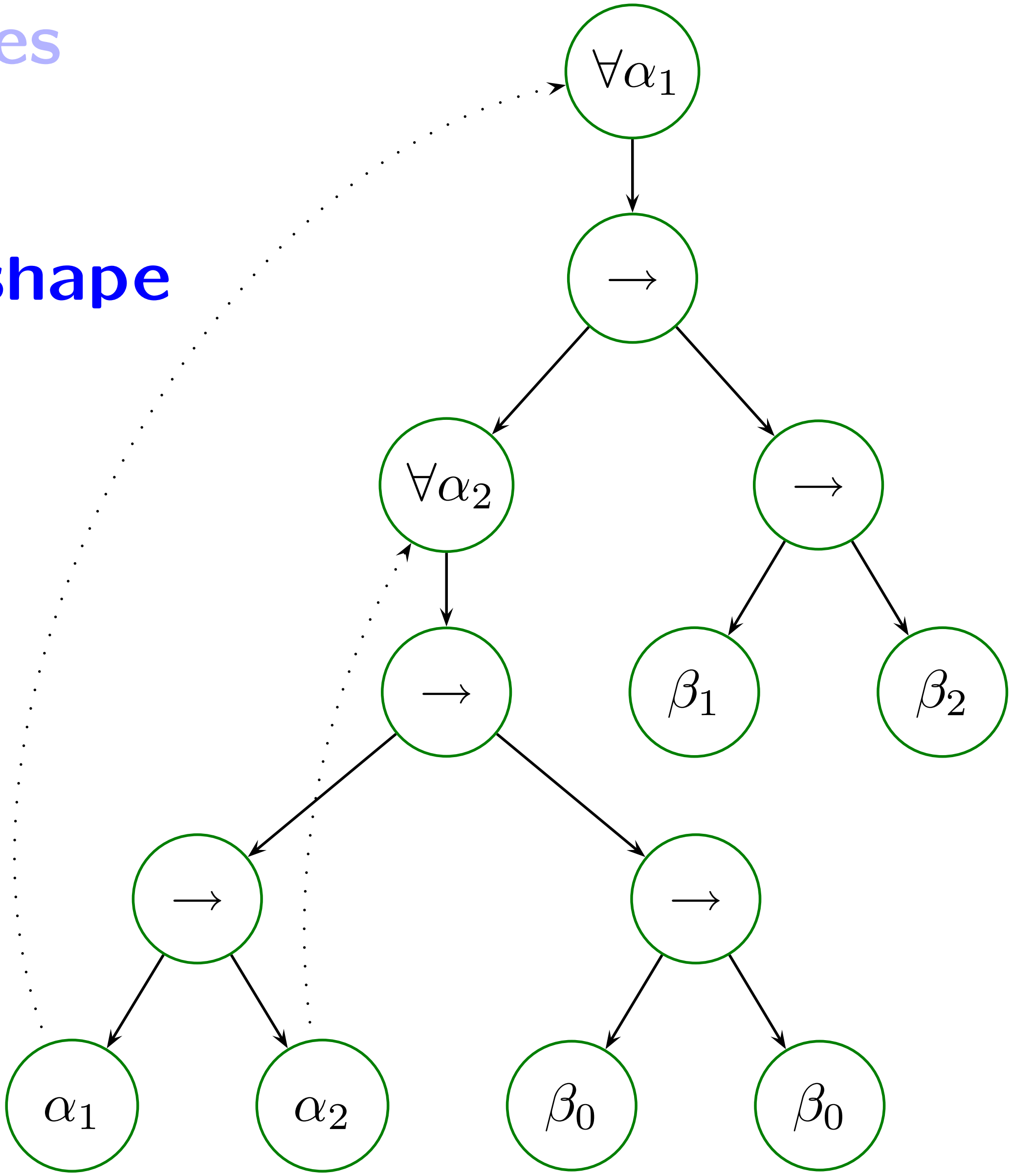
- In CF_{ML} monotypes are inferred, polytypes are checked.
- Shapes abstract away monomorphic parts of polytypes.

Definition

- We extend polytypes with a constant $\sharp$ to represent monotypes.
- Shapes are closed polytypes
  (free variables become monotypes represented by $\sharp$)
- Shapes are taken modulo the absorbing equation $\sharp \rightarrow \sharp = \sharp$
  (we ignore the structure of monotypes)

Operation on shapes

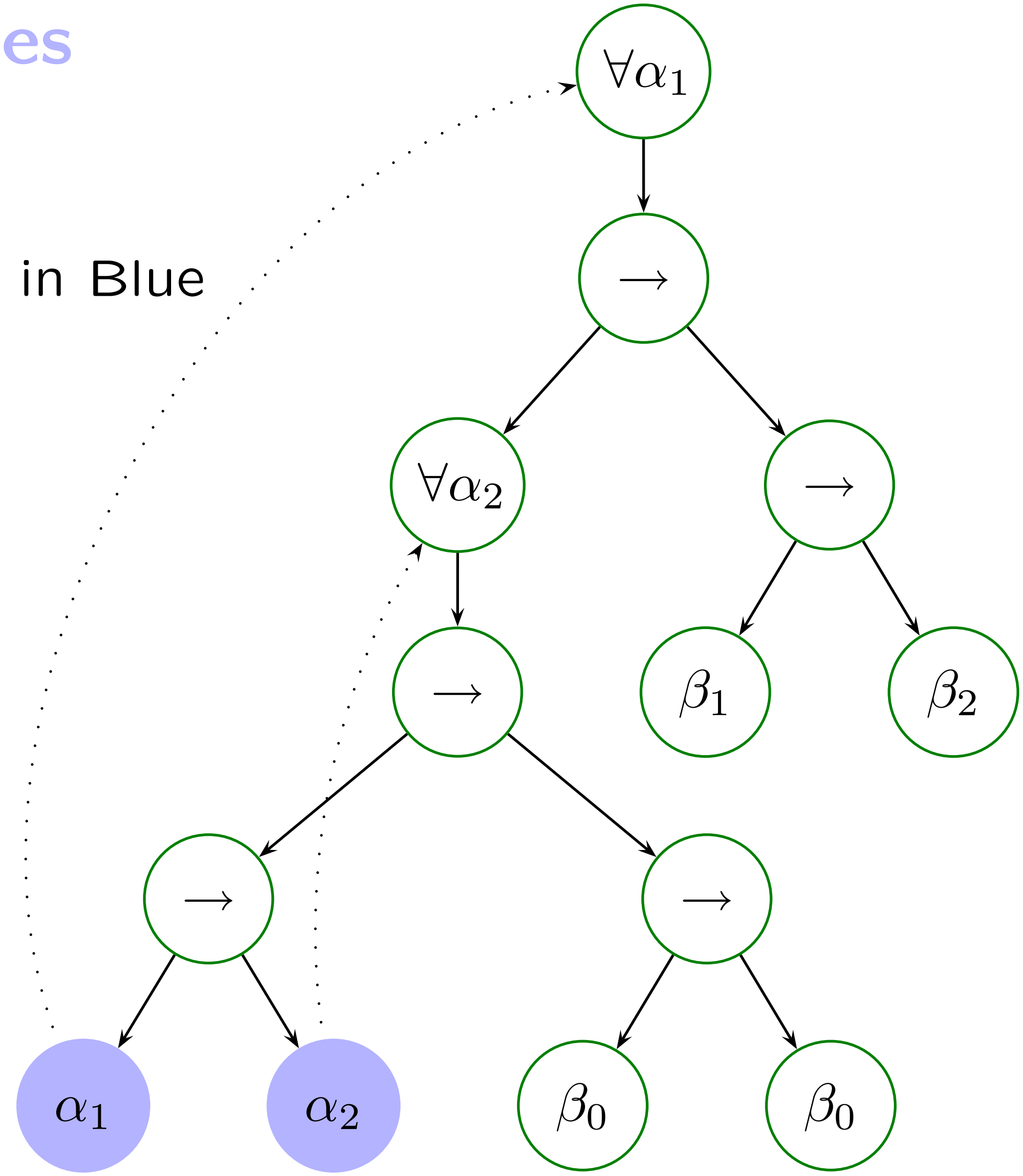
- **Computing the shape**



σ

Operation on shapes

- ▷ Mark bound variables in Blue

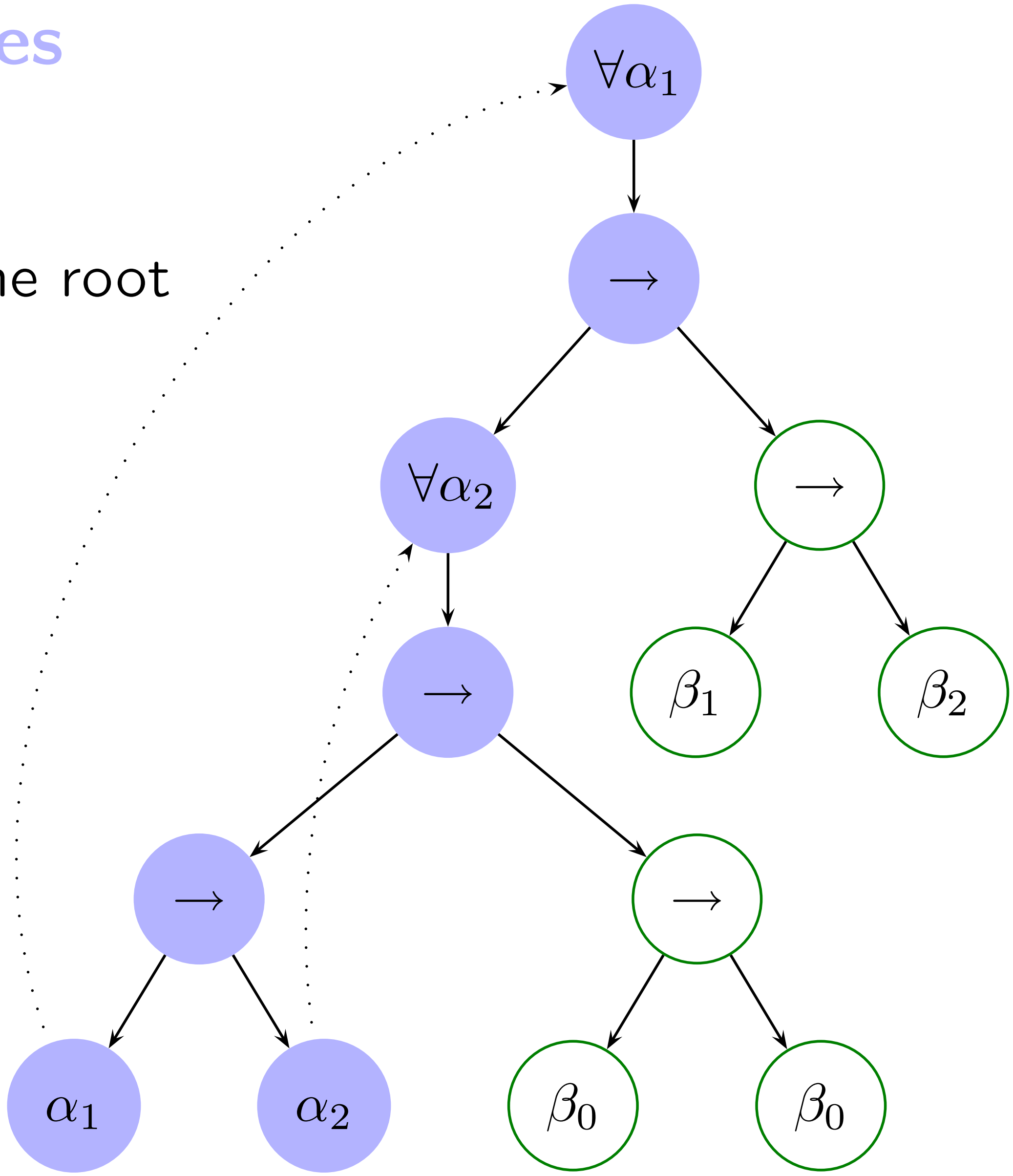


$$\lceil \sigma \rceil$$

Operation on shapes

- ▷ Spread blue toward the root

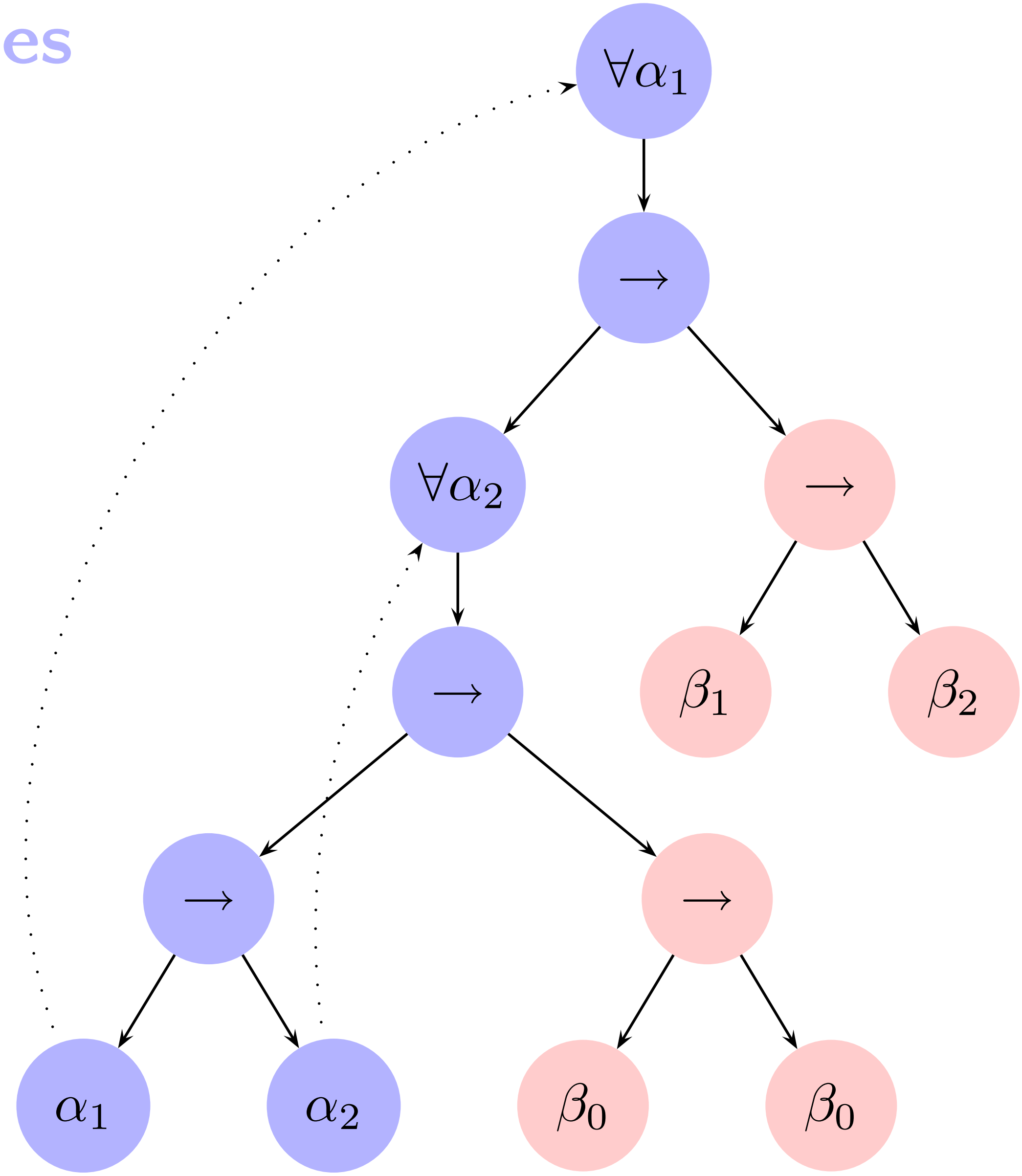
$$\lceil \sigma \rceil$$



Operation on shapes

- Mark the rest in red

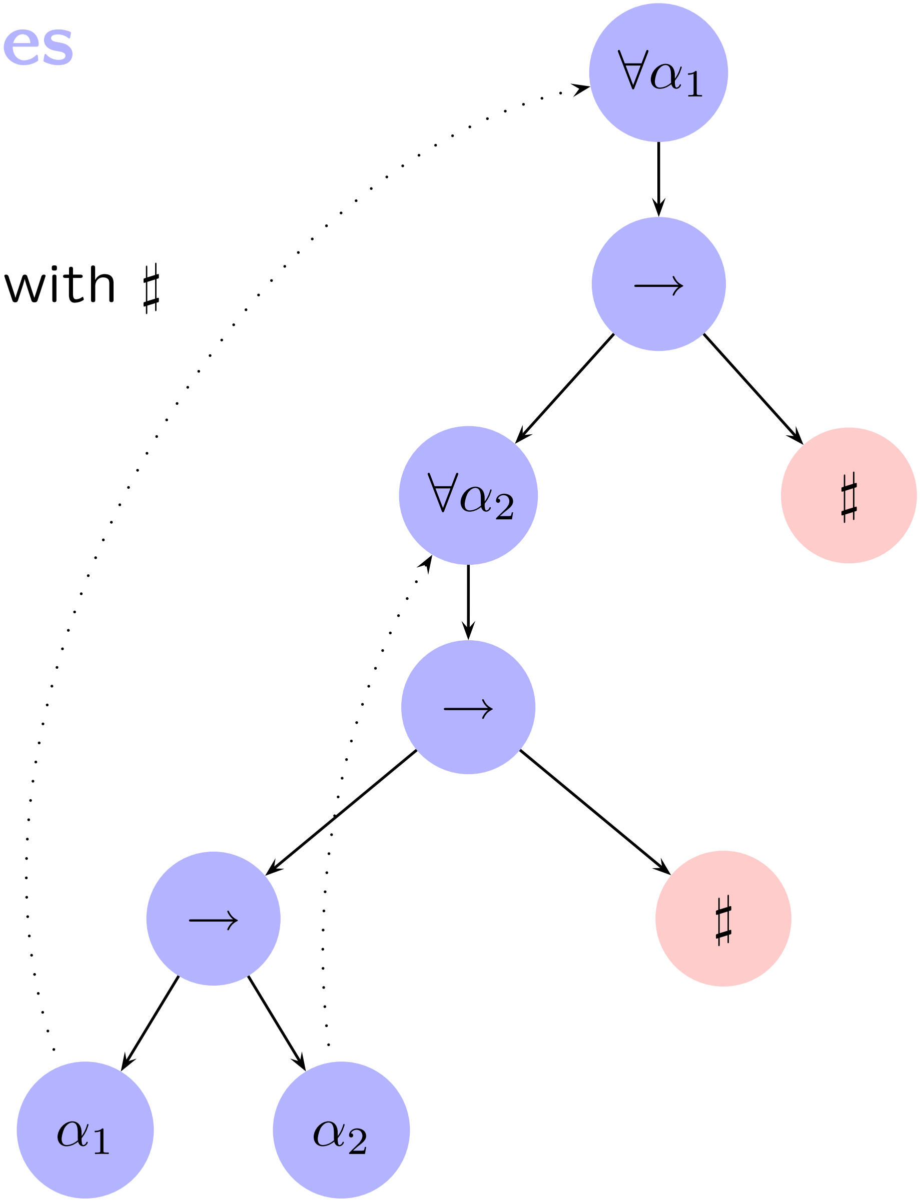
$\boxed{\lceil \sigma \rceil}$



Operation on shapes

- ▷ Replace red subtrees with ♯

$$\lceil \sigma \rceil$$

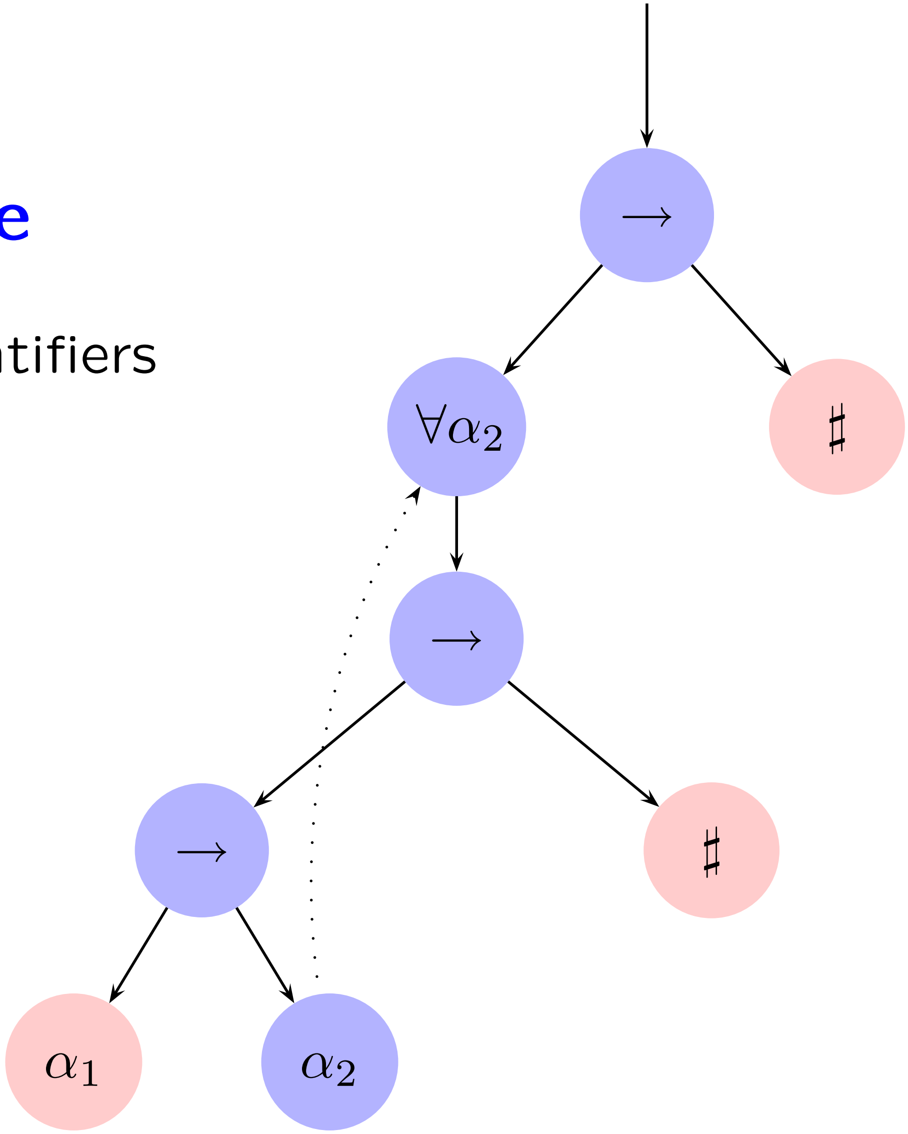


peration on shapes

- **Stripping a shape**

  Remove toplevel quantifiers

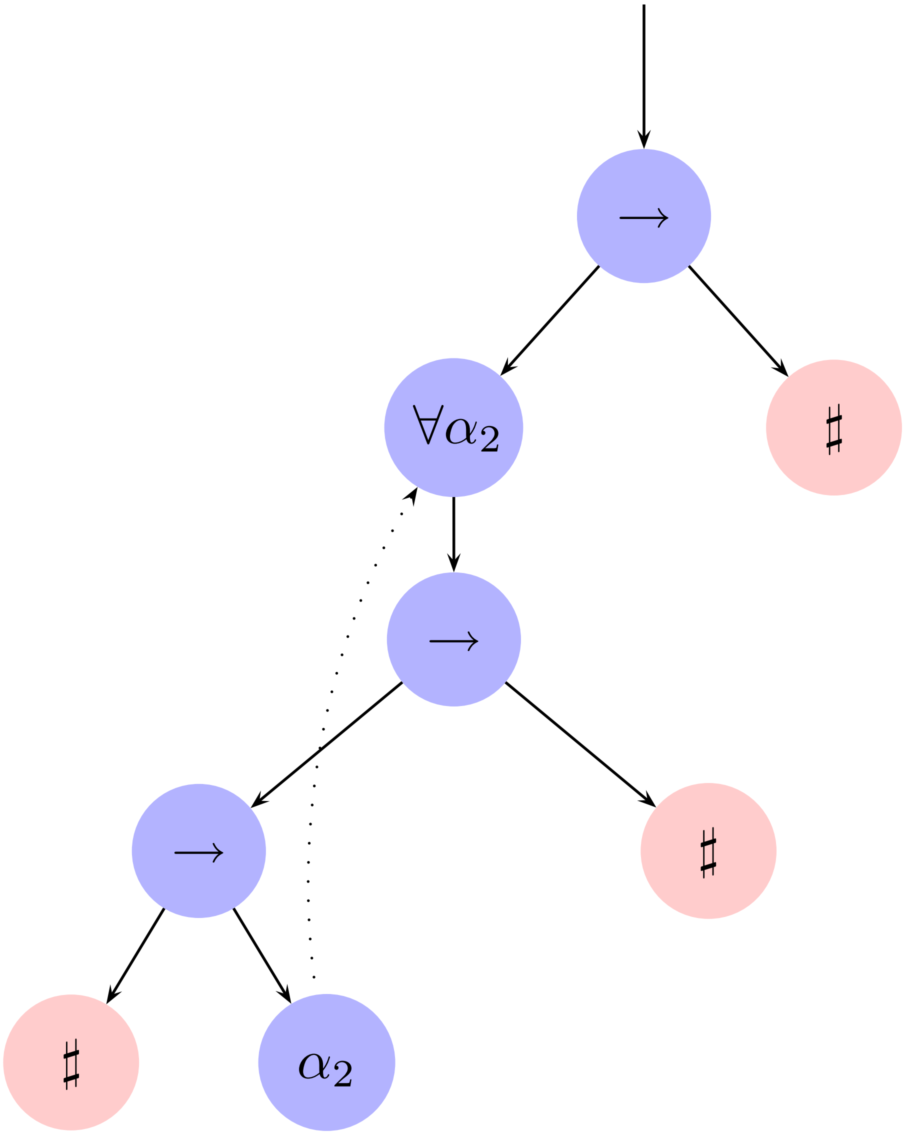
$$\boxed{\mathcal{S} = \sigma^\flat}$$



Operation on shapes

- Reshape

$$\mathcal{S} = \sigma^\flat$$

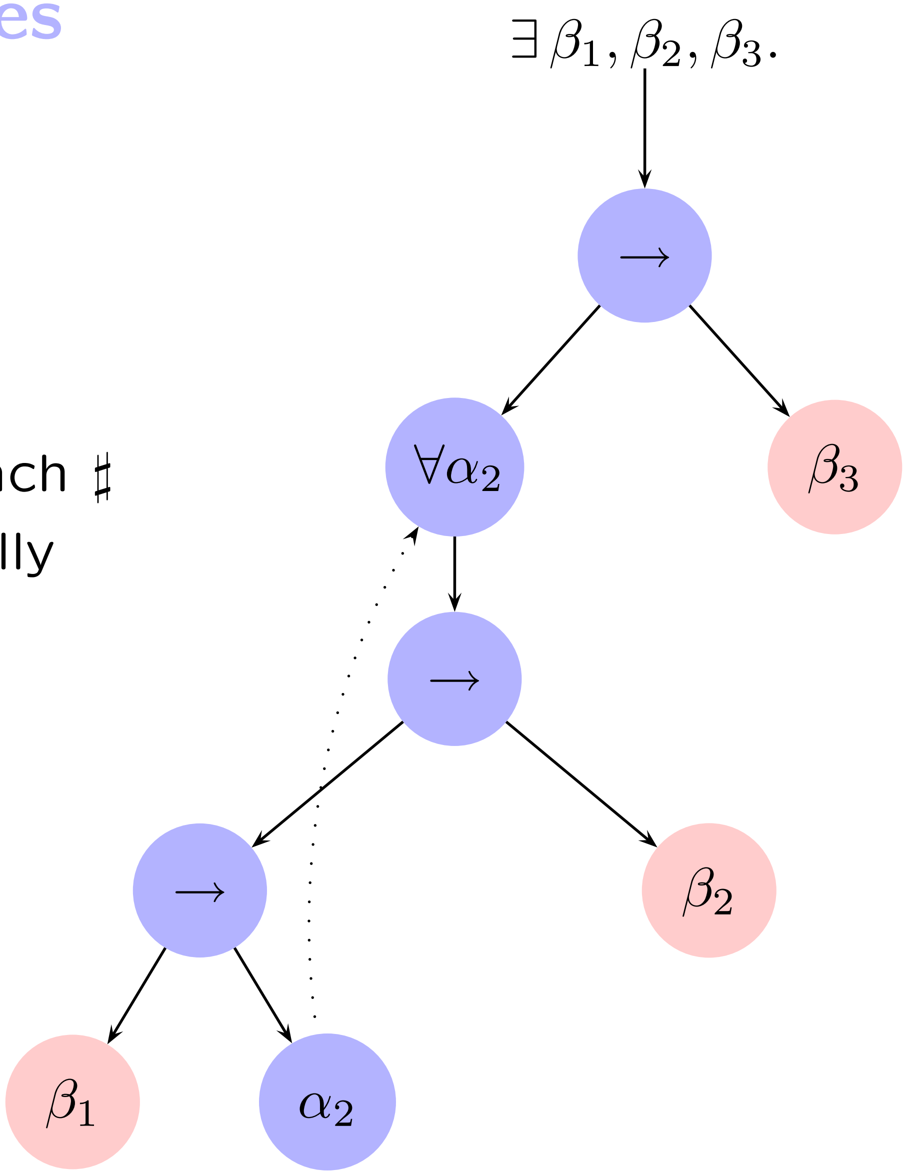


Operation on shapes

- **Building an annotation**

from a shape: turn each ♯ into a fresh existentially quantified variable.

$\lfloor \mathcal{S} \rfloor$



Only shapes of annotations matter

Lemma 1 *If* $\Gamma \vdash \mathsf{t} : \sigma$ *then* $\Gamma \vdash \lfloor\lceil \mathsf{t} \rceil\rfloor : \sigma$.

The monomorphic part of annotations can be inferred!
So only the expected shape must be given.

Can we infer shapes?

- Shape inference must be incomplete
  - For instance, $\vdash \lambda \mathsf{z} \,.\, \mathsf{z}\,\mathsf{z} : \sigma \to \sigma$ for types σ with incomparable shapes.
- So it should be simple and intuitive!
  - Allow some annotations to be omitted.
  - Propagate given shapes to rebuild missing annotations.
  - Fall back to $\sharp$ for unknown shapes.

Elaboration

Allow missing annotations on source terms

$$\mathsf{t} ::= \ldots \mid \mathsf{let}\ \mathsf{z} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 \mid \mathsf{t}_1\ \mathsf{t}_2 \mid \lambda \mathsf{z} : \theta\,.\,\mathsf{t}$$

and, as a counterpart, annotations on function parameters.

Return a term with missing annotations filled in.

$$\Gamma \vdash_{\uparrow} \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'$$

inference mode

$$\Gamma \vdash_{\downarrow} \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'$$

checking mode

Typechecking $\mathsf{SF}_{\mathsf{ML}}$ by elaboration into $\mathsf{CF}_{\mathsf{ML}}$

$$\Gamma \vdash_{\mathsf{SF}_{\mathsf{ML}}} \mathsf{t} : \sigma \iff \Gamma \vdash_{\uparrow} \mathsf{t} : \lceil \sigma \rceil \Rightarrow \mathsf{t}' \;\wedge\; \Gamma \vdash_{\mathsf{CF}_{\mathsf{ML}}} \mathsf{t}' : \sigma$$

Elaboration rules

Var-I Var-C Gen-C Let$_a$-I Let$_a$-C Let-I Let-C Fun$_a$-C Fun$_a$-I

Example of easy rules

Fun-C

$$\frac{\Gamma, z : \mathcal{S}_2 \vdash_{\downarrow} t : \mathcal{S}_1 \Rightarrow t'}{\Gamma \vdash_{\downarrow} \lambda z \,.\, t : \mathcal{S}_2 \rightarrow \mathcal{S}_1 \Rightarrow \lambda z \,.\, t'}$$

Fun-I

$$\frac{\Gamma, z : \sharp \vdash_{\uparrow} t : \mathcal{S} \Rightarrow t'}{\Gamma \vdash_{\uparrow} \lambda z \,.\, t : \sharp \rightarrow \mathcal{S} \Rightarrow \lambda z \,.\, t'}$$

More challenging rules: applications

App-C

$$\frac{\begin{array}{ll} \Gamma \vdash_{\uparrow} t_1 : \mathcal{S} \Rightarrow t'_1 & \mathcal{S}^{\flat} = \mathcal{S}_2 \rightarrow \mathcal{S}'_1 \\ \Gamma \vdash_{\downarrow} t_2 : \mathcal{S}_2 \Rightarrow t'_2 & \mathcal{S}'_1 \lesssim \mathcal{S}_1 \end{array}}{\Gamma \vdash_{\downarrow} t_1 \; t_2 : \mathcal{S}_1 \;\Rightarrow t'_1 \; (t'_2 : \lfloor \mathcal{S}_2 \rfloor)}$$

App$_a$-C App$_a$-I App-I

Elaboration rules

Var-I Var-C Gen-C Let$_a$-I Let$_a$-C Let-I Let-C Fun$_a$-C Fun$_a$-I

Example of easy rules

Fun-C

$$\frac{\Gamma, z : \mathcal{S}_2 \vdash_{\downarrow} t : \mathcal{S}_1 \Rightarrow t'}{\Gamma \vdash_{\downarrow} \lambda z\,.\,t : \mathcal{S}_2 \to \mathcal{S}_1 \Rightarrow \lambda z\,.\,t'}$$

Fun-I

$$\frac{\Gamma, z : \sharp \vdash_{\uparrow} t : \mathcal{S} \Rightarrow t'}{\Gamma \vdash_{\uparrow} \lambda z\,.\,t : \sharp \to \mathcal{S} \Rightarrow \lambda z\,.\,t'}$$

More challenging rules: applications

App-C-Variant

$$\frac{\begin{array}{ccc} \Gamma \vdash_{\uparrow} t_1 : \mathcal{S} \Rightarrow t_1' & & \mathcal{S}^{\flat} = \mathcal{S}_2 \to \mathcal{S}_1' \\ \Gamma \vdash_{\uparrow} t_2 : \mathcal{S}_2' \Rightarrow t_2' & \mathcal{S}_1' \lesssim \mathcal{S}_1 & \mathcal{S}_2' \lesssim \mathcal{S}_2 \end{array}}{\Gamma \vdash_{\downarrow} t_1\ t_2 : \mathcal{S}_1 \Rightarrow t_1'\ (t_2' : \lfloor \mathcal{S}_2 \rfloor)}$$

App$_a$-C App$_a$-I App-I

Limitation of predicative polymorphism

A significant restriction...

- apply $(\lambda \mathsf{f} \,.\, \lambda \mathsf{z} \,.\, \mathsf{f}\ \mathsf{z})$ (or map, iter, *etc.*) will only accept monomorphic arguments...

  Need one version per polymorphic *shape* of the type of f . . .
  including one version per polymorphic shape of the type of z,

  which is really unacceptable!

- same limitation for constructors: a new definition for each shape of arguments.
- More theoretical arguments in the paper.

Indirect limitation

- indirect limitation: $(\lambda \mathsf{z} \,.\, \mathsf{z})\ auto$ need to be annotated, even though the argument z is not used polymorphically.

Recovering impredicativity

Steal existing solutions for ML...

Embed the impredicative fragment within data-types (as in Haskell)

A better available solution is semi-explicit polymorphism
[Garrigue and Rémy, 1997] (as in OCaml)

It simplifies in CF_{ML} since polytypes are already in the host language.

Use coercion/retyping functions

$(\_ : (\forall\,\alpha.\,\sigma \triangleright \sigma[\sigma'/\alpha]))$ of type $\forall\,\alpha.\,\sigma \rightarrow \sigma[\sigma'/\alpha]$ that behaves as the identity.

No extension needed to the theory, use a denumerable set of primitives.

- $(\lambda \mathsf{z}\,.\,\mathsf{z} : (\forall\,\alpha.\,\alpha \rightarrow \alpha \triangleright \sigma \rightarrow \sigma))\ (auto : \sigma)$
  where σ is $(\forall\,\alpha.\,\alpha \rightarrow \alpha) \rightarrow \forall\,\alpha.\,\alpha \rightarrow \alpha$

Incomplete coercions or even coercions themselves may be elaborated.

What about side-effects?

The value restriction may be adapted (See the paper):

Only a small change in the specification but a more significant change in the syntax-directed system:

Syntax-directed presentations are fragile.

Results

Predicative system, effect-free semantics

About as expressive as [Peyton Jones and Shields, 2003] (which is implemented in the Haskell compiler).

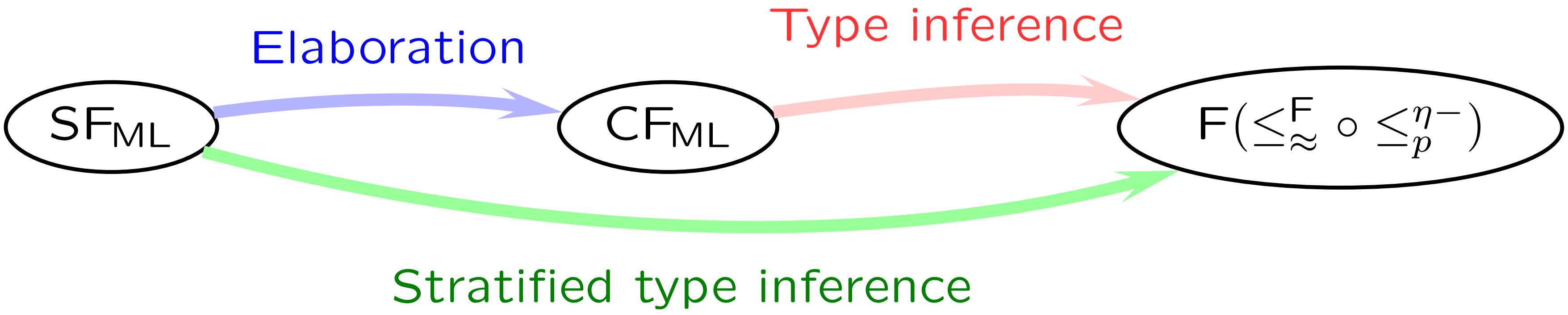
- simpler specification.
- all choices (sources of incompleteness) occur in the elaboration.

Still, some annotations remain unpleasant.

With predicative polymorphism

- naive elaboration of coercions is not very satisfactory.
- tension between elaboration of predicative polymorphism and impredicative polymorphism at application nodes.
- more *ad hoc* rules may be used, *e.g.* “ smart applications ” that treats multi-applications of the form $z\, t_1 \, \ldots \, t_n$ all at once.

Conclusions



Elaboration

Type inference

$\mathsf{SF}_{\mathsf{ML}}$

$\mathsf{CF}_{\mathsf{ML}}$

$\mathsf{F}(\leq^{\mathsf{F}}_{\approx} \circ \leq^{\eta-}_{p})$

Stratified type inference

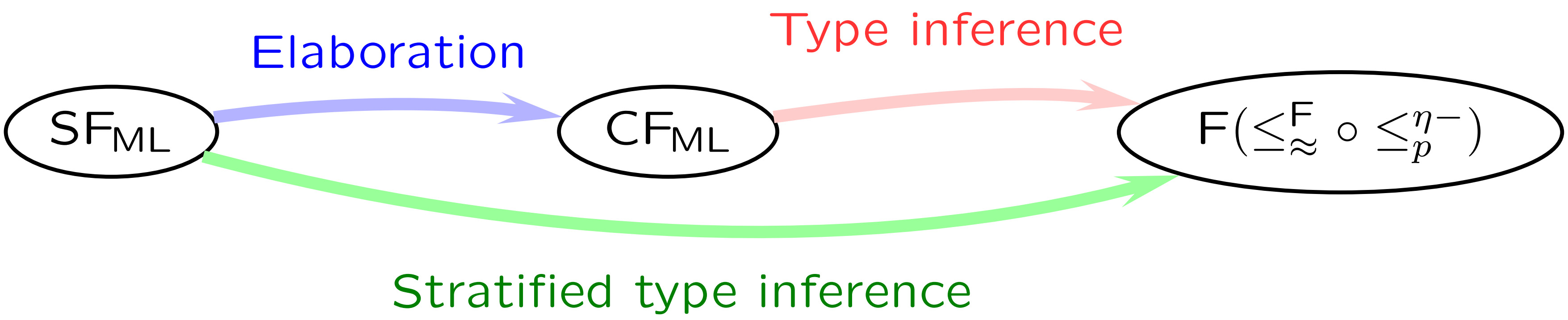
Stratified type inference is good

- Each side is easier to specify and understand, separately.
- $\mathsf{SF}_{\mathsf{ML}}$ is simple, but not entirely satisfactory.
- While $\mathsf{CF}_{\mathsf{ML}}$ is a tiny robust extension to ML with good properties,
- Elaboration is more *ad hoc* and fragile
  (with respect to small program changes, or language variations).

  Small variants as well as other less naive elaboration methods may be explored.

  Does [Vytiniotis et al., 2005] fit in this framework?

Conclusions



Elaboration

Type inference

SF$_{\mathsf{ML}}$

CF$_{\mathsf{ML}}$

$\mathsf{F}(\leq^{\mathsf{F}}_{\approx} \circ \leq^{\eta-}_{p})$

Stratified type inference

Stratified type inference is good

However,

- Elaboration must remain a simple recipe —it should not be too spicy.
- The goal remains to find cleverer *complete* type inference algorithms with respect to *logical* specifications.
- ML$^{\mathsf{F}}$ is theoretically more involved but also significantly more powerful than SF$_{\mathsf{ML}}$ and maybe a more promising direction, because it really addresses the instantiate/generalize dilemma.

Thank you.

Poly-ML [Garrigue and Rémy, 1997]

Explicit introduction of polymorphism

let *id* = *poly* (fun x -> x : $\forall\alpha.\alpha \to \alpha$)

let *auto* (f : $\forall\alpha.\alpha \to \alpha$) = (*mono* f) f

auto id

Available in OCaml !

Boxed Polymorphism

This is the poor man's polymorphism: use a datatype constructor to automatically *encapsulate* a polytype into an ML type, and its destructor to *project* it back into a polymorphic polytype.

$$\text{type } \mathit{id}\ \alpha = \mathit{Id} \text{ of } \forall\ \alpha\,.\ (\alpha \to \alpha)$$

Using the constructor at both introduction and elimination points is then sufficient:

$$\text{let } \mathit{id} = \mathit{Id}\ (\text{fun } x \to x)$$

$$\text{let } \mathit{auto}\ (\mathit{Id}\ f) = f\ f$$

$$\mathit{auto}\ \mathit{id}$$

Boxed Polymorphism

This is the poor man's polymorphism: use a datatype constructor to automatically *encapsulate* a polytype into an ML type, and its destructor to *project* it back into a polymorphic polytype.

$$\text{type } \mathit{id}\ \alpha = \mathit{Id} \text{ of } \forall\ \alpha.\ (\alpha \to \alpha)$$

Limitations

- Simple cases are easy, but examples become tricky when quantifiers appear under other quantifiers.
- A polytype can often be embedded in several (incompatible) ways.
- Heavy for an intensive usage:
  - require type declarations before use, even for a single use.
  - types are less readable—each (group of) quantifiers must be named.

Some reading...

[Garrigue and Rémy, 1997] Garrigue, J. and Rémy, D. (1997). Extending ML with semi-explicit higher-order polymorphism. In Ito, T. and Abadi, M., editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 20–46. Springer-Verlag.

[Läufer and Odersky, 1994] Läufer, K. and Odersky, M. (1994). Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430.

[Le Botlan and Rémy, 2003] Le Botlan, D. and Rémy, D. (2003). MLF: Raising ML to the power of system-F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38.

[Mitchell, 1988] Mitchell, J. C. (1988). Polymorphic type inference and containment. *Information and Computation*, 76:211–249.

[Odersky and Läufer, 1996] Odersky, M. and Läufer, K. (1996). Putting type annotations to work. In *ACM Symposium on Principles of Programming*, pages 54–67, St. Petersburg, Florida. ACM Press.

[Odersky et al., 2001] Odersky, M., Zenger, C., and Zenger, M. (2001). Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53.

[Peyton Jones and Shields, 2003] Peyton Jones, S. and Shields, M. (2003). Lexically-scoped type variables. Submitted to ICFP 2004.

[Peyton Jones et al., 2005a] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. (2005a). Practical type inference for arbitrary-rank types. Submitted to the Journal of Functional Programming.

[Peyton Jones et al., 2005b] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. (2005b). Practical type inference for arbitrary-rank types. technical appendix. Private communication.

[Pfenning, 1988] Pfenning, F. (1988). Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 153–163. ACM Press.

[Pierce and Turner, 2000] Pierce, B. C. and Turner, D. N. (2000). Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44.

[Rémy, 1994] Rémy, D. (1994). Programming objects with ML-ART: An extension to ML with abstract and record types. In Hagiya, M. and Mitchell, J. C., editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag.

[Vytiniotis et al., 2005] Vytiniotis, D., Weirich, S., and Peyton Jones, S. (2005).

Boxy type inference for higher-rank types and impredicativity. Available electronically at `http://www.cis.upenn.edu/~dimitriv/boxy/boxy.ps`.

Elaboration rules

Var-I

$$\frac{\mathsf{x} : \mathcal{S} \in \Gamma}{\Gamma \vdash_{\uparrow} \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Var-C

$$\frac{\mathsf{x} : \mathcal{S}' \in \Gamma}{\Gamma \vdash_{\downarrow} \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Elaboration rules

Var-I

$$\frac{x : \mathcal{S} \in \Gamma}{\Gamma \vdash_{\uparrow} x : \mathcal{S} \Rightarrow x}$$

Var-C

$$\frac{x : \mathcal{S}' \in \Gamma}{\Gamma \vdash_{\downarrow} x : \mathcal{S} \Rightarrow x}$$

Gen-C

$$\frac{\Gamma \vdash_{\downarrow} t : \mathcal{S}^{\flat} \Rightarrow t'}{\Gamma \vdash_{\downarrow} t : \mathcal{S} \Rightarrow t'}$$

Elaboration rules

Var-I

$$\frac{x : \mathcal{S} \in \Gamma}{\Gamma \vdash_{\uparrow} x : \mathcal{S} \Rightarrow x}$$

Var-C

$$\frac{x : \mathcal{S}' \in \Gamma}{\Gamma \vdash_{\downarrow} x : \mathcal{S} \Rightarrow x}$$

Gen-C

$$\frac{\Gamma \vdash_{\downarrow} t : \mathcal{S}^{\flat} \Rightarrow t'}{\Gamma \vdash_{\downarrow} t : \mathcal{S} \Rightarrow t'}$$

Let$_a$-I

$$\frac{\Gamma \vdash_{\downarrow} t_1 : \lceil \sigma \rceil \Rightarrow t_1' \qquad \Gamma, z : \lceil \sigma \rceil \vdash_{\uparrow} t_2 : \mathcal{S}_2 \Rightarrow t_2'}{\Gamma \vdash_{\uparrow} \text{let } z = (t_1 : \exists \bar{\beta}. \sigma) \text{ in } t_2 : \mathcal{S}_2 \Rightarrow \text{let } z = (t_1' : \exists \bar{\beta}. \sigma) \text{ in } t_2'}$$

Elaboration rules

Var-I

$$\frac{\mathsf{x} : \mathcal{S} \in \Gamma}{\Gamma \vdash_{\uparrow} \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Var-C

$$\frac{\mathsf{x} : \mathcal{S}' \in \Gamma}{\Gamma \vdash_{\downarrow} \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Gen-C

$$\frac{\Gamma \vdash_{\downarrow} \mathsf{t} : \mathcal{S}^{\flat} \Rightarrow \mathsf{t}'}{\Gamma \vdash_{\downarrow} \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'}$$

Let$_a$-C

$$\frac{\Gamma \vdash_{\downarrow} \mathsf{t}_1 : \lceil \sigma \rceil \Rightarrow \mathsf{t}_1' \qquad \Gamma, \mathsf{z} : \lceil \sigma \rceil \vdash_{\downarrow} \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_{\downarrow} \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\, \bar{\beta}.\, \sigma)\ \mathsf{in}\ \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1' : \exists\, \bar{\beta}.\, \sigma)\ \mathsf{in}\ \mathsf{t}_2'}$$

Elaboration rules

Var-I

$$\frac{\mathsf{x} : \mathcal{S} \in \Gamma}{\Gamma \vdash_{\uparrow} \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Var-C

$$\frac{\mathsf{x} : \mathcal{S}' \in \Gamma}{\Gamma \vdash_{\downarrow} \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Gen-C

$$\frac{\Gamma \vdash_{\downarrow} \mathsf{t} : \mathcal{S}^{\flat} \Rightarrow \mathsf{t}'}{\Gamma \vdash_{\downarrow} \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'}$$

Let-C

$$\frac{\Gamma \vdash_{\uparrow} \mathsf{t}_1 : \mathcal{S}_1 \Rightarrow \mathsf{t}'_1 \qquad \Gamma, \mathsf{z} : \mathcal{S}_1 \vdash_{\downarrow} \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{t}'_2}{\Gamma \vdash_{\downarrow} \mathsf{let}\ \mathsf{z} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{let}\ \mathsf{z} = (\mathsf{t}'_1 : \lfloor \mathcal{S}_1 \rfloor)\ \mathsf{in}\ \mathsf{t}'_2}$$

Elaboration rules

Var-I

$$\frac{x : \mathcal{S} \in \Gamma}{\Gamma \vdash_{\uparrow} x : \mathcal{S} \Rightarrow x}$$

Var-C

$$\frac{x : \mathcal{S}' \in \Gamma}{\Gamma \vdash_{\downarrow} x : \mathcal{S} \Rightarrow x}$$

Gen-C

$$\frac{\Gamma \vdash_{\downarrow} t : \mathcal{S}^{\flat} \Rightarrow t'}{\Gamma \vdash_{\downarrow} t : \mathcal{S} \Rightarrow t'}$$

Let-I

$$\frac{\Gamma \vdash_{\uparrow} t_1 : \mathcal{S}_1 \Rightarrow t'_1 \qquad \Gamma, z : \mathcal{S}_1 \vdash_{\uparrow} t_2 : \mathcal{S}_2 \Rightarrow t'_2}{\Gamma \vdash_{\uparrow} \text{let } z = t_1 \text{ in } t_2 : \mathcal{S}_2 \Rightarrow \text{let } z = (t'_1 : \lfloor \mathcal{S}_1 \rfloor) \text{ in } t'_2}$$

Elaboration rules

Fun-C

$$\frac{\Gamma, \mathsf{z} : \mathcal{S}_2 \vdash_{\downarrow} \mathsf{t} : \mathcal{S}_1 \Rightarrow \mathsf{t}'}{\Gamma \vdash_{\downarrow} \lambda \mathsf{z} \,.\, \mathsf{t} : \mathcal{S}_2 \to \mathcal{S}_1 \Rightarrow \lambda \mathsf{z} \,.\, \mathsf{t}'}$$

Fun$_{a}$-C

$$\frac{\Gamma, \mathsf{z} : \lceil \sigma \rceil \vdash_{\downarrow} \mathsf{t} : \mathcal{S}_1 \Rightarrow \mathsf{t}'}{\begin{array}{c}\Gamma \vdash_{\downarrow} \lambda \mathsf{z} : \exists\, \beta . \sigma \,.\, \mathsf{t} : \mathcal{S}_2 \to \mathcal{S}_1 \\ \Rightarrow \lambda \mathsf{z} \,.\, \mathsf{let}\ \mathsf{z} = (\mathsf{z} : \exists\, \bar{\beta} . \sigma)\ \mathsf{in}\ \mathsf{t}'\end{array}}$$

Fun$_{a}$-I

$$\frac{\Gamma, \mathsf{z} : \lceil \sigma \rceil \vdash_{\uparrow} \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'}{\begin{array}{c}\Gamma \vdash_{\uparrow} \lambda \mathsf{z} : \exists\, \bar{\beta} . \sigma \,.\, \mathsf{t} : \lceil \sigma \rceil \to \mathcal{S} \\ \Rightarrow \lambda \mathsf{z} \,.\, \mathsf{let}\ \mathsf{z} = (\mathsf{z} : \exists\, \bar{\beta} . \sigma)\ \mathsf{in}\ \mathsf{t}'\end{array}}$$

Fun-I

$$\frac{\Gamma, \mathsf{z} : \sharp \vdash_{\uparrow} \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'}{\Gamma \vdash_{\uparrow} \lambda \mathsf{z} \,.\, \mathsf{t} : \sharp \to \mathcal{S} \Rightarrow \lambda \mathsf{z} \,.\, \mathsf{t}'}$$

Elaboration rules

App$_a$-C

$$\frac{\Gamma \vdash_{\downarrow} \mathsf{t}_1 : \lceil \sigma \rceil \to \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \Gamma \vdash_{\downarrow} \mathsf{t}_2 : \lceil \sigma \rceil \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_{\downarrow} \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar{\alpha}.\,\sigma) : \mathcal{S} \Rightarrow \mathsf{t}_1'\ (\mathsf{t}_2' : \exists\,\bar{\alpha}.\,\sigma)}$$

Elaboration rules

App$_a$-C

$$\frac{\Gamma \vdash_{\downarrow} \mathsf{t}_1 : \lceil\sigma\rceil \to \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \Gamma \vdash_{\downarrow} \mathsf{t}_2 : \lceil\sigma\rceil \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_{\downarrow} \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar{\alpha}.\,\sigma) : \mathcal{S} \Rightarrow \mathsf{t}_1'\ (\mathsf{t}_2' : \exists\,\bar{\alpha}.\,\sigma)}$$

App$_a$-I

$$\frac{\Gamma \vdash_{\uparrow} \mathsf{t}_1 : \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \mathcal{S}^{\flat} = \mathcal{S}_2 \to \mathcal{S}_1 \qquad \Gamma \vdash_{\downarrow} \mathsf{t}_2 : \lceil\sigma\rceil \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_{\uparrow} \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar{\alpha}.\,\sigma) : \mathcal{S}_1 \Rightarrow \mathsf{t}_1'\ (\mathsf{t}_2' : \exists\,\bar{\alpha}.\,\sigma)}$$

App-C

$$\frac{\Gamma \vdash_{\uparrow} \mathsf{t}_1 : \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \mathcal{S}^{\flat} = \mathcal{S}_2 \to \mathcal{S}_1 \qquad \Gamma \vdash_{\downarrow} \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_{\downarrow} \mathsf{t}_1\ \mathsf{t}_2 : \mathcal{R}_1 \Rightarrow \mathsf{t}_1\ (\mathsf{t}_2 : \lfloor\mathcal{S}_2\rfloor)}$$

App-I

$$\frac{\Gamma \vdash_{\uparrow} \mathsf{t}_1 : \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \mathcal{S}^{\flat} = \mathcal{S}_2 \to \mathcal{S}_1 \qquad \Gamma \vdash_{\downarrow} \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_{\uparrow} \mathsf{t}_1\ \mathsf{t}_2 : \mathcal{S}_1 \Rightarrow \mathsf{t}_1\ (\mathsf{t}_2 : \lfloor\mathcal{S}_2\rfloor)}$$

From System F toward ML

- Second-order unification [Pfenning, 1988].

  Expressive, but undecidable. Places for type abstraction and type application must still be explicit.

- Local type inference [Pierce and Turner, 2000] [Odersky et al., 2001]

  to remove the most dummy type annotations. Not conservative over ML.

Type inference via constraints

$$\begin{aligned}
\llbracket \mathsf{x} : \rho \rrbracket &\longrightarrow \mathsf{x} \preceq \rho \\
\llbracket \lambda \mathsf{z} \,.\, \mathsf{t} : \alpha \rrbracket &\longrightarrow \exists\, \beta_1 \beta_2 . (\llbracket \lambda \mathsf{z} \,.\, \mathsf{t} : \beta_1 \to \beta_2 \rrbracket \wedge \beta_1 \to \beta_2 \leqslant \alpha) \\
\llbracket \lambda \mathsf{z} \,.\, \mathsf{t} : \sigma_2 \to \sigma_1 \rrbracket &\longrightarrow \mathsf{let}\ \mathsf{z} : \sigma_2\ \mathsf{in}\ \llbracket \mathsf{t} : \sigma_1 \rrbracket \\
\llbracket \mathsf{t}_1\ (\mathsf{t}_2 : \exists\, \bar\beta . \sigma_2) : \rho_1 \rrbracket &\longrightarrow \exists\, \bar\beta . (\llbracket \mathsf{t}_1 : \sigma_2 \to \rho_1 \rrbracket \wedge \llbracket \mathsf{t}_2 : \sigma_2 \rrbracket) \\
\llbracket \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\, \bar\beta . \sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \rho_2 \rrbracket &\longrightarrow \mathsf{let}\ \mathsf{z} : \forall \bar\beta [\llbracket \mathsf{t}_1 : \sigma_1 \rrbracket] . \sigma_1\ \mathsf{in}\ \llbracket \mathsf{t}_2 : \rho_2 \rrbracket \\
\llbracket \mathsf{t} : \forall\, \bar\alpha . \rho \rrbracket &\longrightarrow \forall\, \bar\alpha . \llbracket \mathsf{t}_2 : \rho \rrbracket
\end{aligned}$$

Logical interpretation of constraints

1. Standard interpretation of $\exists$, $\forall$, $\wedge$.
2. let constraints can be understood by macro expansion.
3. $(\forall \bar\beta [C] . \sigma) \preceq \sigma'$ then means $\exists\, \bar\beta . (C \wedge \sigma \leqslant \sigma')$
4. $\leqslant$ constraints are interpreted by $\leq_p^{\eta-}$

Type inference via constraints

$$\begin{array}{rcll} \tau \leqslant \tau' & \longrightarrow & \tau = \tau' & \text{Refl} \\ \sigma_1 \to \sigma_2 \leqslant \sigma_1' \to \sigma_2' & \longrightarrow & \sigma_1' \leqslant \sigma_1 \wedge \sigma_2 \to \sigma_2' & \text{Arrow} \\ \forall\,\alpha.\,\sigma \leqslant \rho & \longrightarrow & \exists\,\alpha.(\sigma \leqslant \rho) & \text{All-E} \\ \sigma \leqslant \forall\,\alpha.\,\sigma' & \longrightarrow & \forall\,\alpha.(\sigma \leqslant \sigma') & \text{All-I} \end{array}$$

- Follows syntax-directed rules for $\leq_p^{\eta-}$
- Reduces instantiation constraints into equality constraints

Inference is first order

- No meta variables for σ or ρ, only α for τ.
- Polymorphic shapes are only checked.