A decorative graphic consisting of several thick, light blue lines. A horizontal line is at the top. A vertical line is on the left side. A curved line starts from the left and goes towards the right. Below this, there are two wavy lines that resemble a stylized signature or the letters 'MS'.

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

Didier Rémy  
INRIA-Rocquencourt

ms

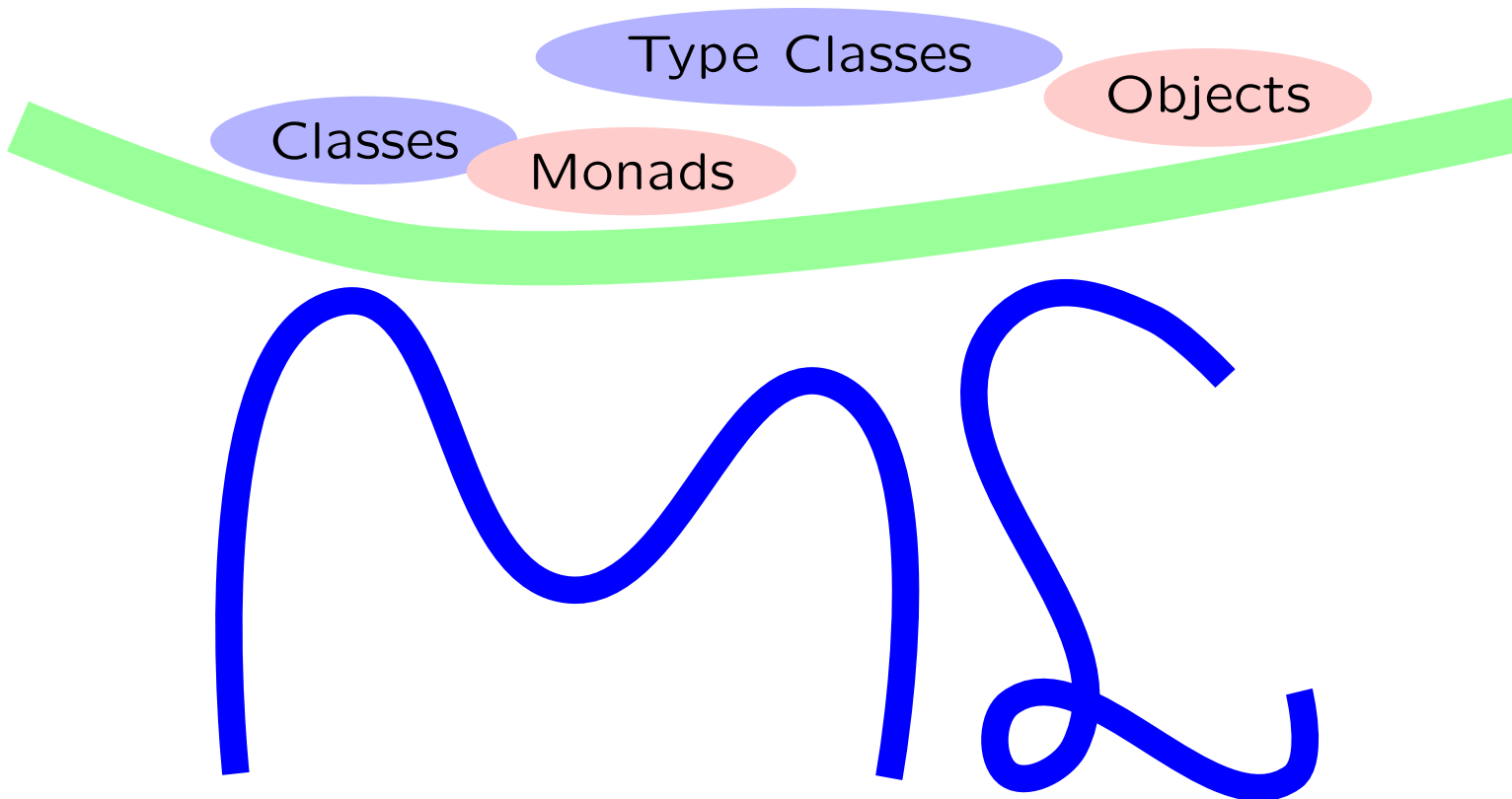


Classes

Objects

# ML is simple, yet expressive

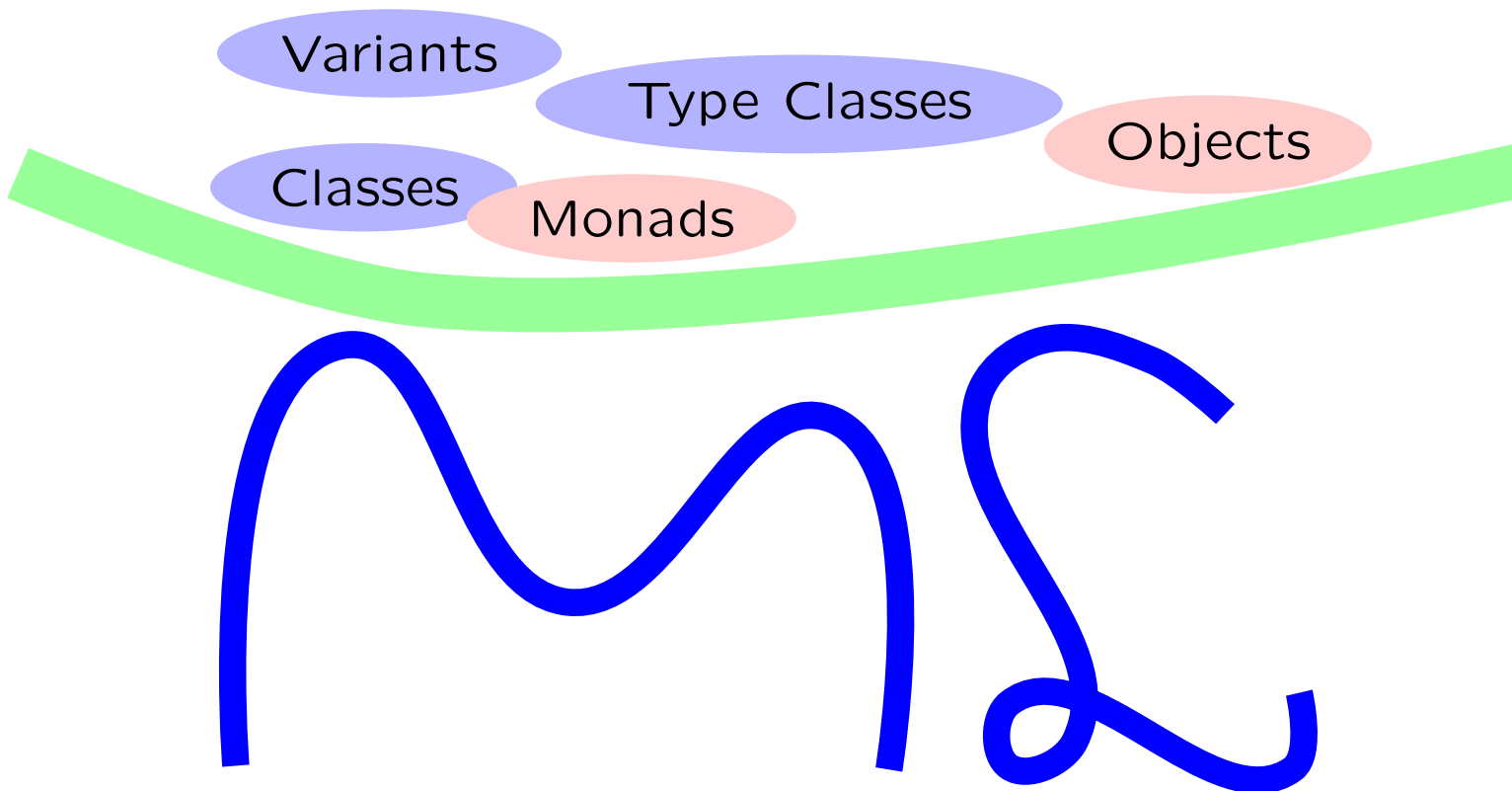
---

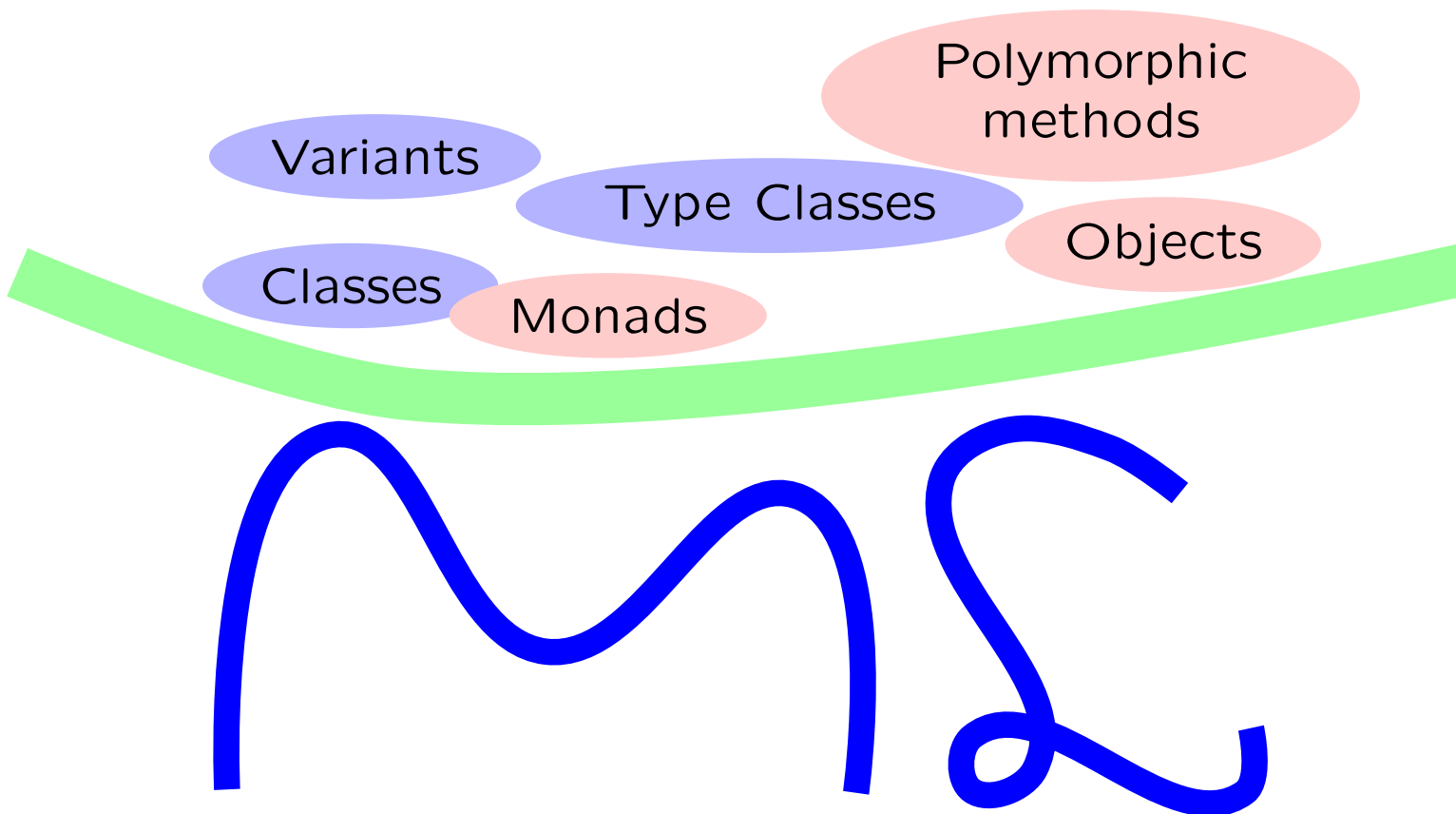


# ML is simple, yet expressive and robust

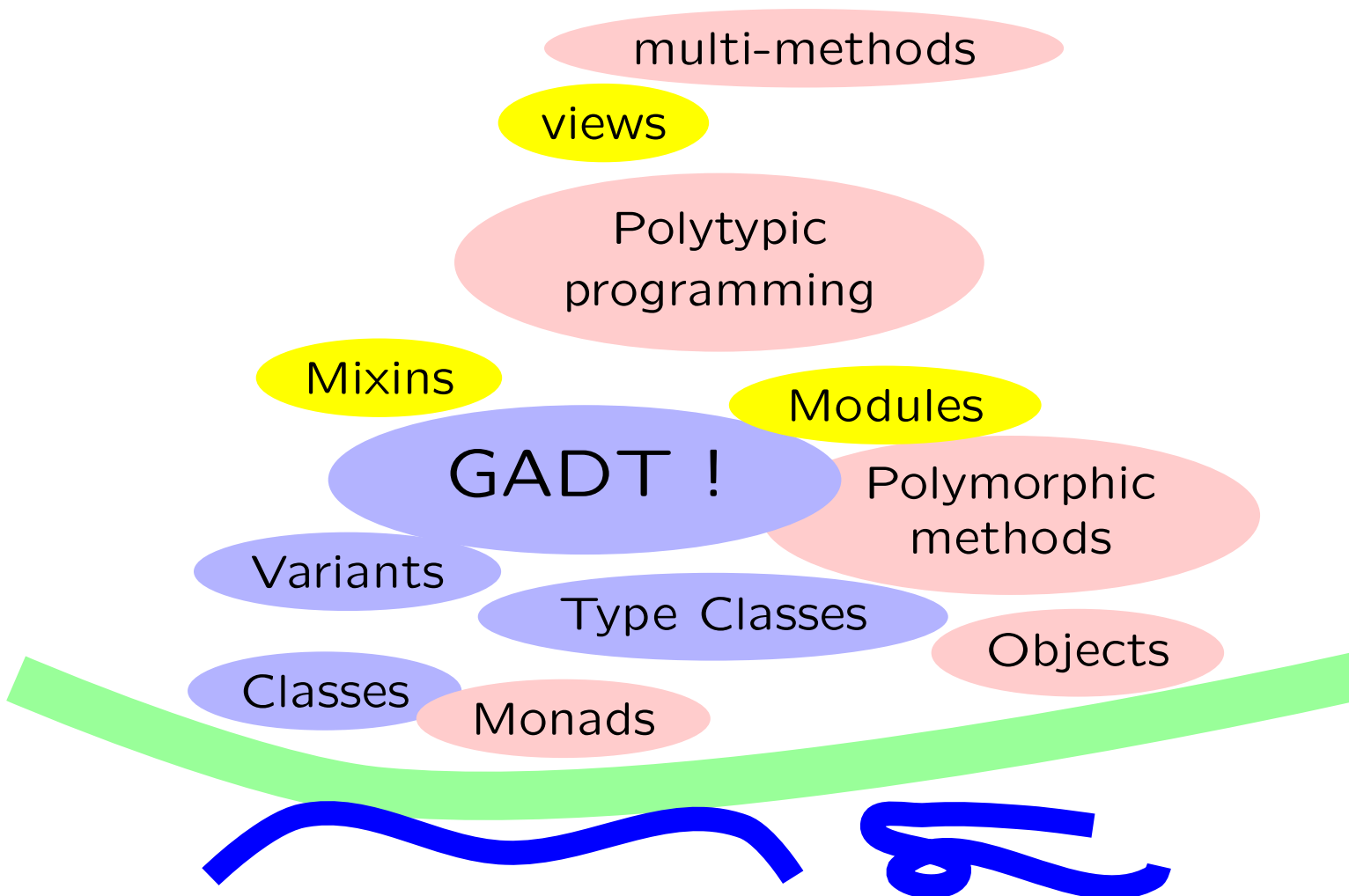
---

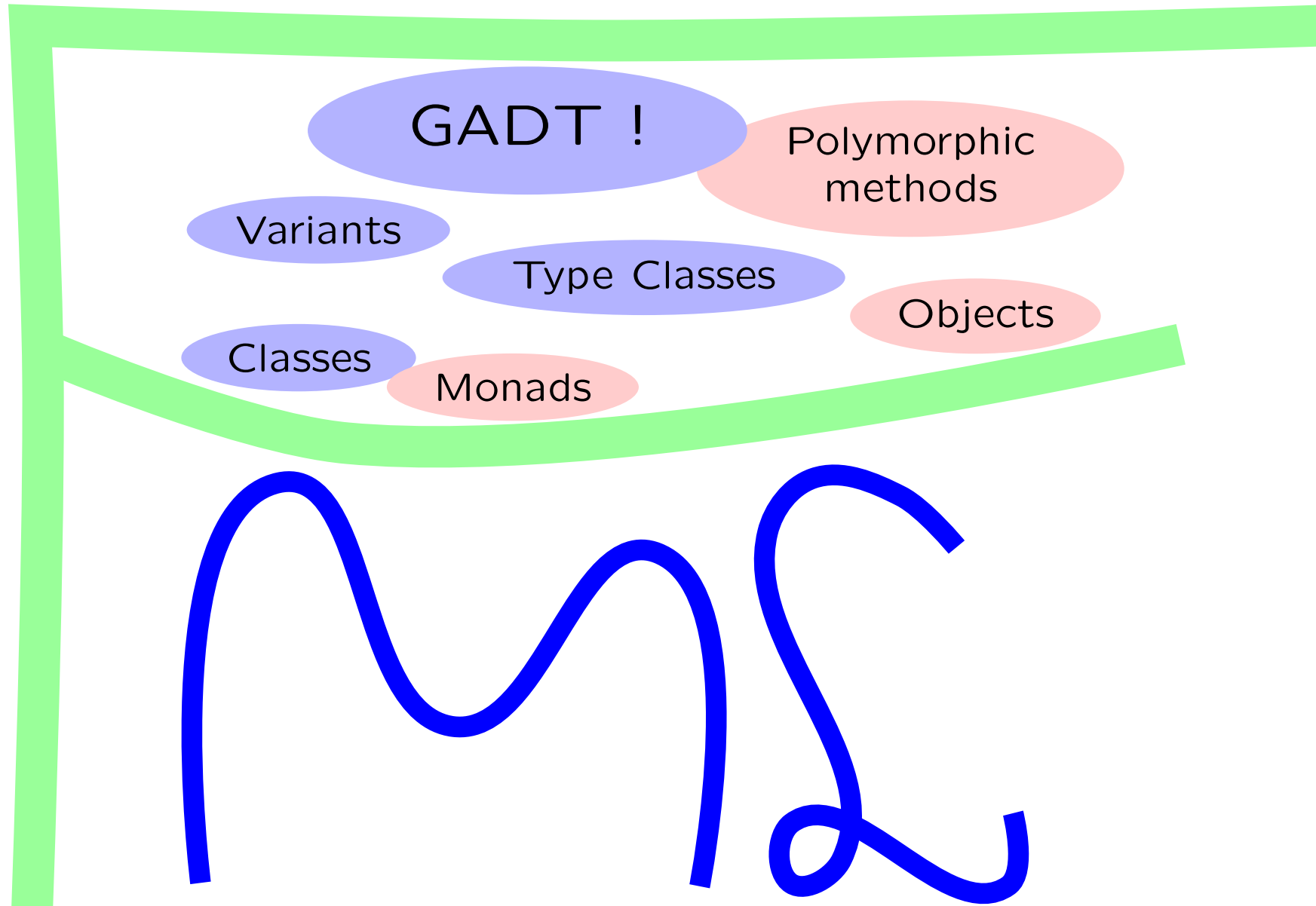
▷ 2(4)/23





# ML is cracking under extensions





Second-order types are good!

*m s*

Type inference is also good!



ms

ML

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

(almost)  
transparent  
**boxed**

Universal types via data-types [Rémy, 1994]

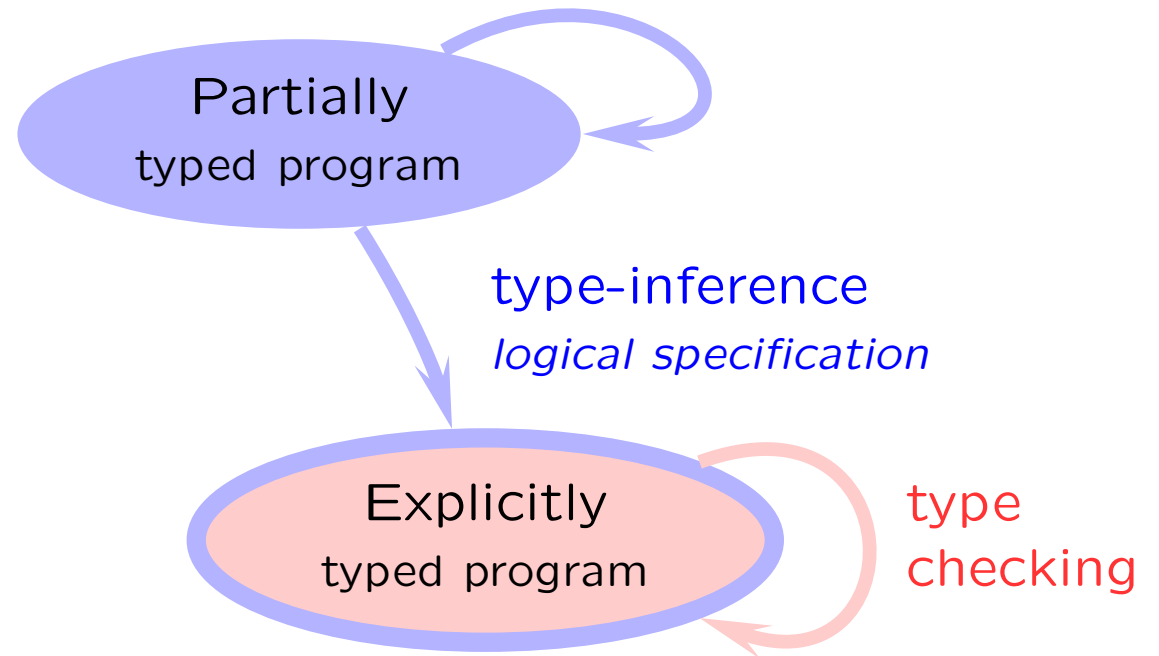
[Odersky and Läufer, 1996]

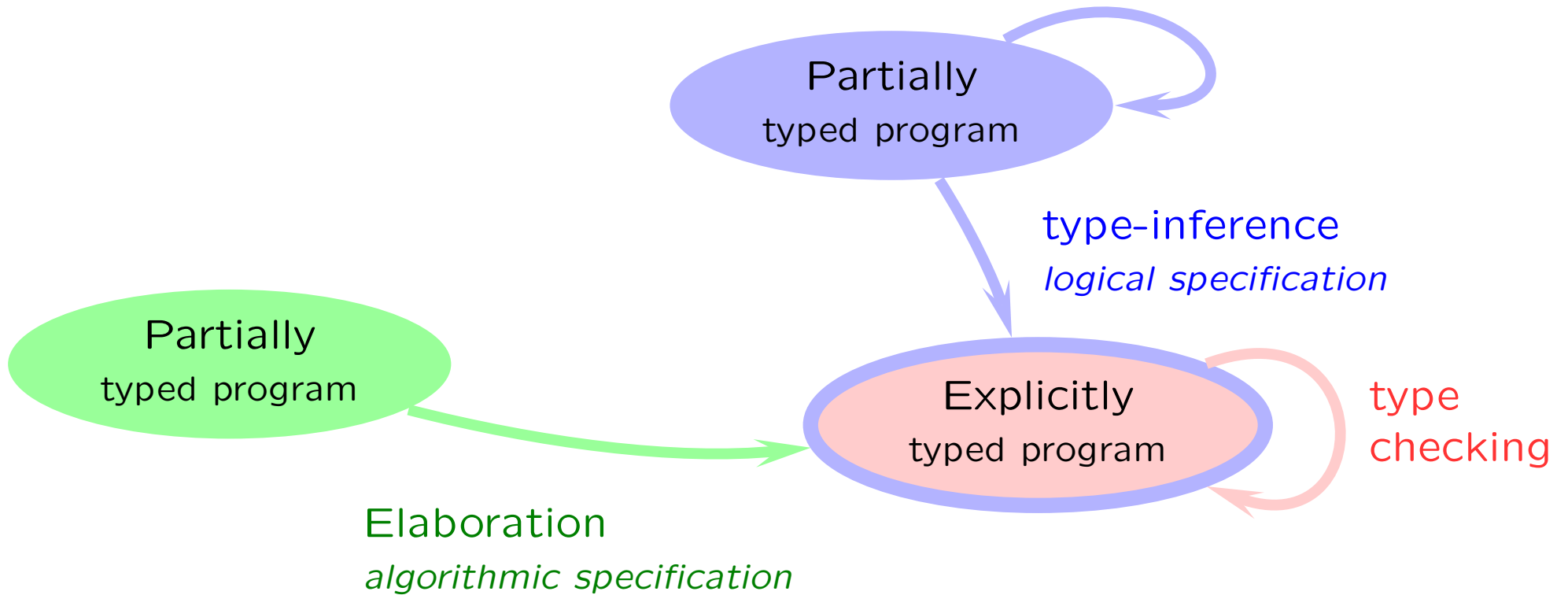
[Garrigue and Rémy, 1997]  
(In OCaml)

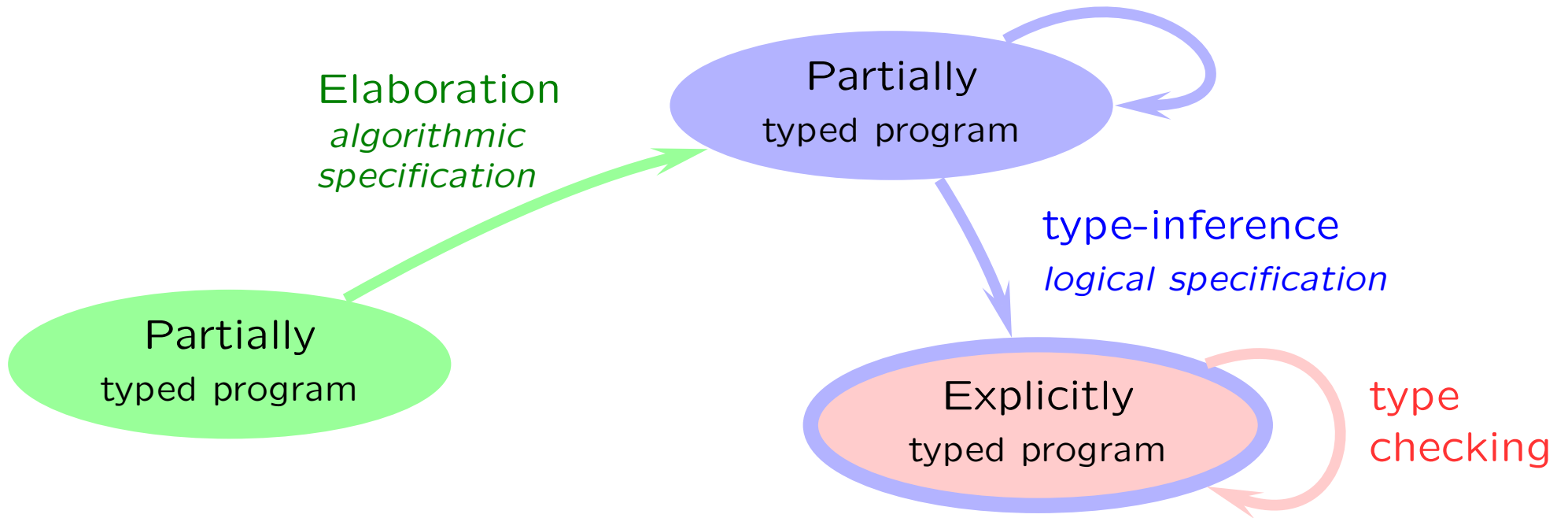
[Peyton Jones et al., 2005a]  
(In Haskell)

MLF  
[Le Botlan and Rémy, 2003]

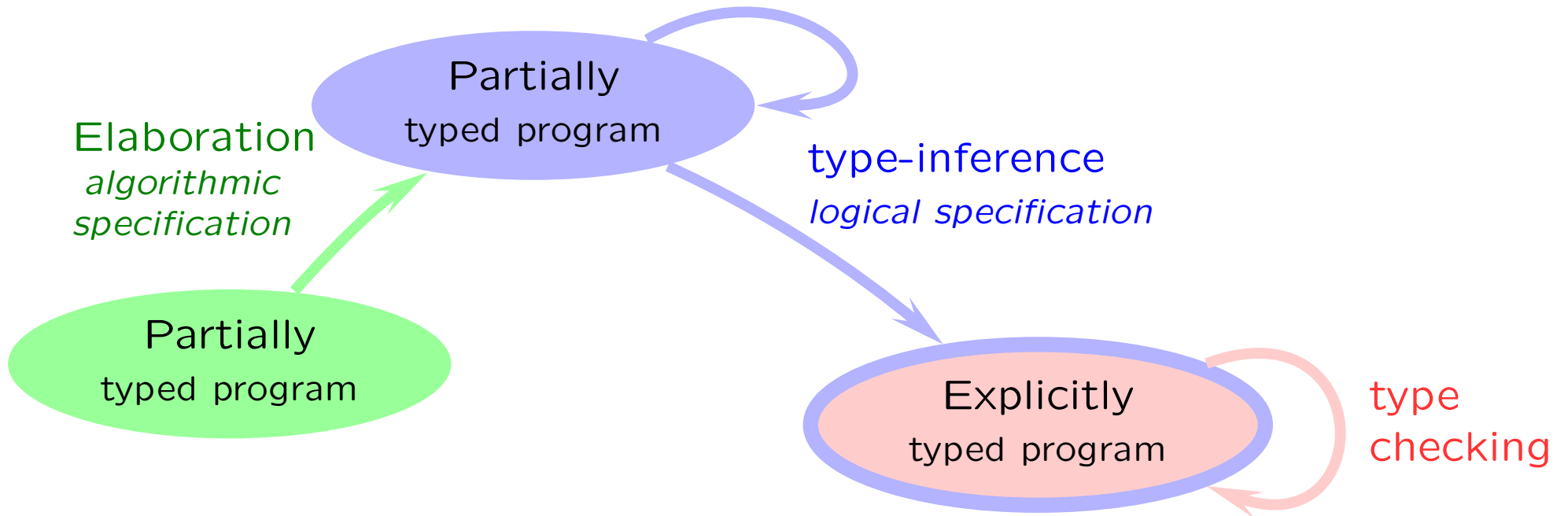




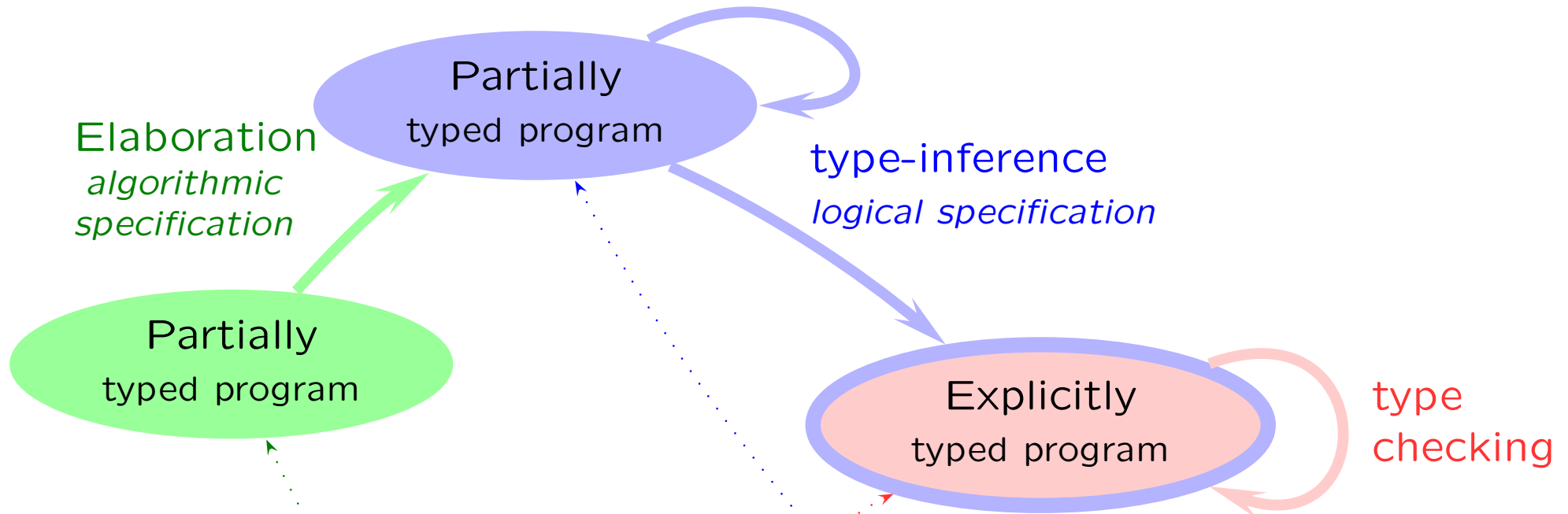




## Stratified type inference



## Stratified type inference



## This work

- ▶ A reference target language  $F(\leq)$  (for some instance relation  $\leq$ ).
- ▶ A robust, partially typed core language  $CF_{ML}$  with complete type inference.
- ▶ A surface language  $SF_{ML}$  with its elaboration into  $CF_{ML}$ .

## Assume

let *choose* =  $\lambda z y . \text{if } \dots \text{ then } z \text{ else } y$

let *id* =  $\lambda z . z$

$\forall \alpha . \alpha \rightarrow \alpha \rightarrow \alpha$

$\forall \beta . \beta \rightarrow \beta$

## The instantiate-generalize problem

*choose id*

▷ instantiate *id*, generalize the result?

▷ keep *id* polymorphic?

?

$\forall \beta . ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta))$

$(\forall \beta . \beta \rightarrow \beta) \rightarrow (\forall \beta . \beta \rightarrow \beta)$

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

▷ concisely describe all possible types?

▷ make such situations illegal?

▷ make one case ill-typed?

▷ pick one type and be incomplete?

} **difficult...**

**unsatisfactory**

# Typing rules for implicit System $F(X)$

6(1)/23

Terms:  $z \mid \lambda z. t \mid t t$

Types:  $\alpha \mid \sigma \rightarrow \sigma \mid \forall \alpha. \sigma$

Var

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

App

$$\frac{\Gamma \vdash t_1 : \sigma_2 \rightarrow \sigma_1 \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash t_1 t_2 : \sigma_1}$$

Fun

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

Gen

$$\frac{\Gamma \vdash t : \sigma \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash t : \forall \alpha. \sigma}$$

Inst

$$\frac{\Gamma \vdash t : \sigma \quad \sigma \leq_X \sigma'}{\Gamma \vdash t : \sigma'}$$

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

# Type-containment $\leq$

7(1)/23

$\leq$  for System F

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

Sub

$$\frac{\bar{\beta} \notin \text{ftv}(\forall \bar{\alpha}. \sigma)}{\forall \bar{\alpha}. \sigma \leq \forall \bar{\beta}. \sigma[\bar{\sigma}/\bar{\alpha}]}$$

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

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

Sub

$$\frac{\bar{\beta} \notin \text{ftv}(\forall \bar{\alpha}. \sigma)}{\forall \bar{\alpha}. \sigma \leq \forall \bar{\beta}. \sigma[\bar{\sigma}/\bar{\alpha}]}$$

Trans

$$\frac{\sigma \leq \sigma' \quad \sigma' \leq \sigma''}{\sigma \leq \sigma''}$$

Arrow

$$\frac{\sigma'_1 \leq \sigma_1 \quad \sigma_2 \leq \sigma'_2}{\sigma_1 \rightarrow \sigma_2 \leq \sigma'_1 \rightarrow \sigma'_2}$$

All

$$\frac{\sigma \leq \sigma'}{\forall \alpha. \sigma \leq \forall \alpha. \sigma'}$$

Distrib

$$\forall \alpha. \sigma \rightarrow \sigma' \leq (\forall \alpha. \sigma) \rightarrow \forall \alpha. \sigma'$$

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

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

Sub

$$\frac{\bar{\beta} \notin \text{ftv}(\forall \bar{\alpha}. \sigma)}{\forall \bar{\alpha}. \sigma \leq \forall \bar{\beta}. \sigma[\bar{\sigma}/\bar{\alpha}]}$$

Trans

$$\frac{\sigma \leq \sigma' \quad \sigma' \leq \sigma''}{\sigma \leq \sigma''}$$

Arrow

$$\frac{\sigma'_1 \leq \sigma_1 \quad \sigma_2 \leq \sigma'_2}{\sigma_1 \rightarrow \sigma_2 \leq \sigma'_1 \rightarrow \sigma'_2}$$

All

$$\frac{\sigma \leq \sigma'}{\forall \alpha. \sigma \leq \forall \alpha. \sigma'}$$

Distrib

$$\forall \alpha. \sigma \rightarrow \sigma' \leq (\forall \alpha. \sigma) \rightarrow \forall \alpha. \sigma'$$

Distrib-Left

$$\frac{\alpha \notin \text{ftv}(\sigma')}{\forall \alpha. \sigma \rightarrow \sigma' \leq (\forall \alpha. \sigma) \rightarrow \sigma'}$$

(derivable)

Distrib-Right

$$\frac{\alpha \notin \text{ftv}(\sigma)}{\forall \alpha. \sigma \rightarrow \sigma' \leq \sigma \rightarrow \forall \alpha. \sigma'}$$

(reversible, hence  $\equiv$ )

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

Distrib

$$\forall \beta. (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \leq^\eta (\forall \beta. \beta \rightarrow \beta) \rightarrow (\forall \beta. \beta \rightarrow \beta)$$

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.

Requires that type variables may only be instantiated by monotypes.

$$\begin{array}{ll} \text{monotypes} & \tau \mid \alpha \mid \tau \rightarrow \tau \\ \text{(poly)types} & \sigma \mid \tau \mid \sigma \rightarrow \sigma \mid \forall \alpha. \sigma \end{array}$$

Rule Sub is replaced by its predicative version:

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

Defines  $\leq_p^F$  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.

## 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:

<b>Refl</b> $\sigma \leq \sigma$	<b>Arrow</b> $\frac{\sigma_1 \leq \sigma'_1 \quad \sigma'_2 \leq \sigma_2}{\sigma_2 \rightarrow \sigma_1 \leq \sigma'_2 \rightarrow \sigma'_1}$	<b>All-I</b> $\frac{\sigma \leq \sigma' \quad \alpha \notin \text{ftv}(\sigma)}{\sigma \leq \forall \alpha. \sigma'}$	<b>All-E</b> $\frac{\sigma[\tau/\alpha] \leq \sigma'}{\forall \alpha. \sigma \leq \sigma'}$
-------------------------------------	--	--	--

## Checking $\leq_p^{\eta}$

Let  $\text{prf}(\sigma)$  compute the prenex form of  $\sigma$  applying the rewrite rule:

$$\sigma' \rightarrow \forall \alpha. \sigma \rightsquigarrow \forall \alpha. \sigma' \rightarrow \sigma$$

**Theorem** [Peyton Jones et al., 2005b]

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

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

## Goal

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

## Expressions

$t ::= x \mid \lambda z. t \mid t_1 \ t_2 \quad \mid \text{let } z = t_1 \text{ in } t_2$

## Goal

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

## Expressions

$t ::= x \mid \lambda z. t \mid t_1 (t_2 : \theta) \mid \text{let } z = (t_1 : \theta) \text{ in } t_2$

## Annotations

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

- ▶  $\sigma$  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

▷ 11(1)/23

Var

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

Inst

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

Gen

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

Fun

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

App

$$\frac{\Gamma \vdash t_1 : \tau_2 \quad \rightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau}$$

Let

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

# Typing rules for ML with annotations

▷ 11(2)/23

Var

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

Inst

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

Gen

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

Fun

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

App

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

Let

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

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

▷ 11(3)/23

Var

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

Inst

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

Gen

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

Fun

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

App

$$\frac{\Gamma \vdash t_1 : \sigma_2[\bar{\tau}/\bar{\beta}] \rightarrow \sigma \quad \Gamma \vdash t_2 : \sigma_2[\bar{\tau}/\bar{\beta}]}{\Gamma \vdash t_1 (t_2 : \exists \bar{\beta}. \sigma_2) : \sigma}$$

Let

$$\frac{\Gamma \vdash t_1 : \forall \bar{\alpha}. \sigma_1[\bar{\tau}/\bar{\beta}] \quad \Gamma, z : \forall \bar{\alpha}. \sigma_1[\bar{\tau}/\bar{\beta}] \vdash t_2 : \sigma_2}{\Gamma \vdash \text{let } z = (t_1 : \exists \bar{\beta}. \sigma_1) \text{ in } t_2 : \sigma_2}$$

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

▷ 11(4)/23

Var-Inst

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

Gen

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

Fun

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

App-Rho

$$\frac{\Gamma \vdash t_1 : \sigma_2[\bar{\tau}/\bar{\beta}] \rightarrow \rho \quad \Gamma \vdash t_2 : \sigma_2[\bar{\tau}/\bar{\beta}]}{\Gamma \vdash t_1 (t_2 : \exists \bar{\beta}. \sigma_2) : \rho}$$

Let-Gen

$$\frac{\Gamma \vdash t_1 : \sigma_1[\bar{\tau}/\bar{\beta}] \quad \Gamma, z : \forall \setminus \Gamma. \sigma_1[\bar{\tau}/\bar{\beta}] \vdash t_2 : \rho_2}{\Gamma \vdash \text{let } z = (t_1 : \exists \bar{\beta}. \sigma_1) \text{ in } t_2 : \rho_2}$$

(quasi) syntax-directed presentation.

# Type inference via type constraints

---

▷ 12(1)/23

See the paper

## As in ML

let  $id = \lambda z . z$

$id \ id$

let  ~~$auto = \lambda f . f f$~~

## Plus...

let  $auto = (\lambda f . f f : \exists \beta . (\forall \alpha . \alpha \rightarrow \alpha) \rightarrow \beta)$

$auto \ (\lambda z . z : \forall \alpha . \alpha \rightarrow \alpha)$

$(\lambda f . f f) \ (id : \forall \alpha . \alpha \rightarrow \alpha)$

$(\lambda z . z) \ (auto : \exists \beta . (\forall \alpha . \alpha \rightarrow \alpha) \rightarrow \beta)$

## Elaboration

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

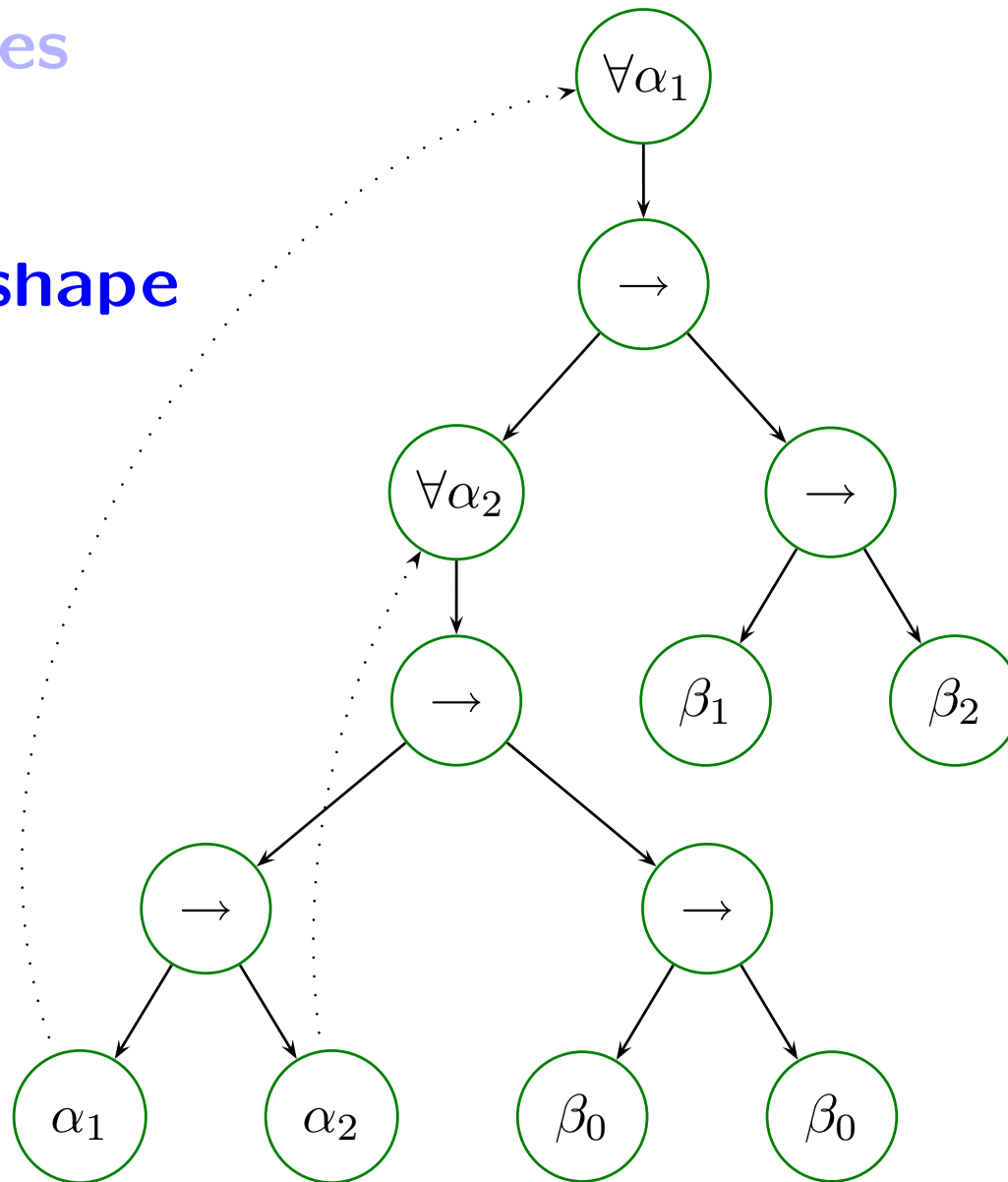
- ▶ In  $CF_{ML}$  monotypes are inferred, polytypes are checked.
- ▶ Shapes abstract away monomorphic parts of polytypes.

## Definition

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

# Operation on shapes

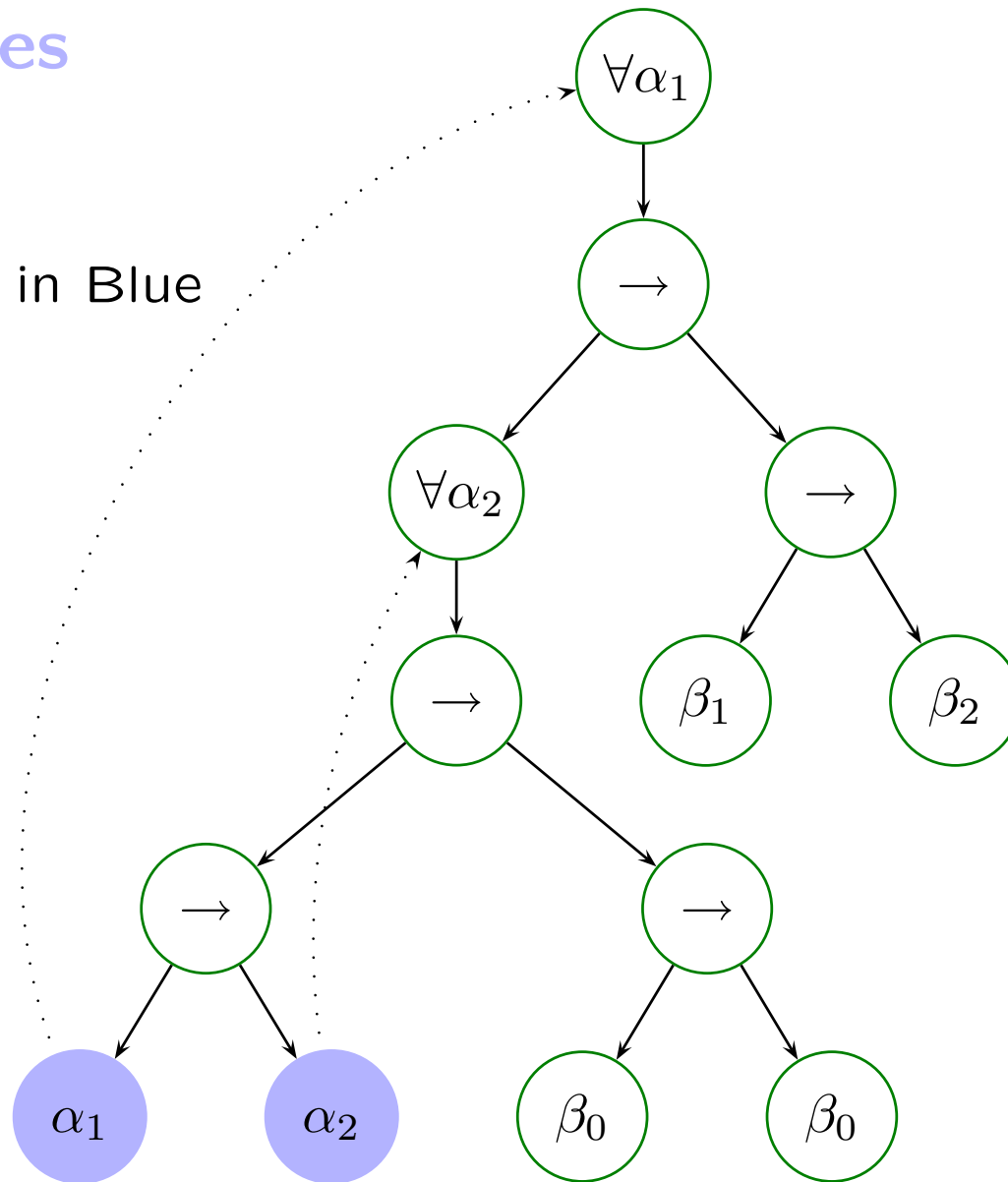
## ▷ Computing the shape



$\sigma$

# Operation on shapes

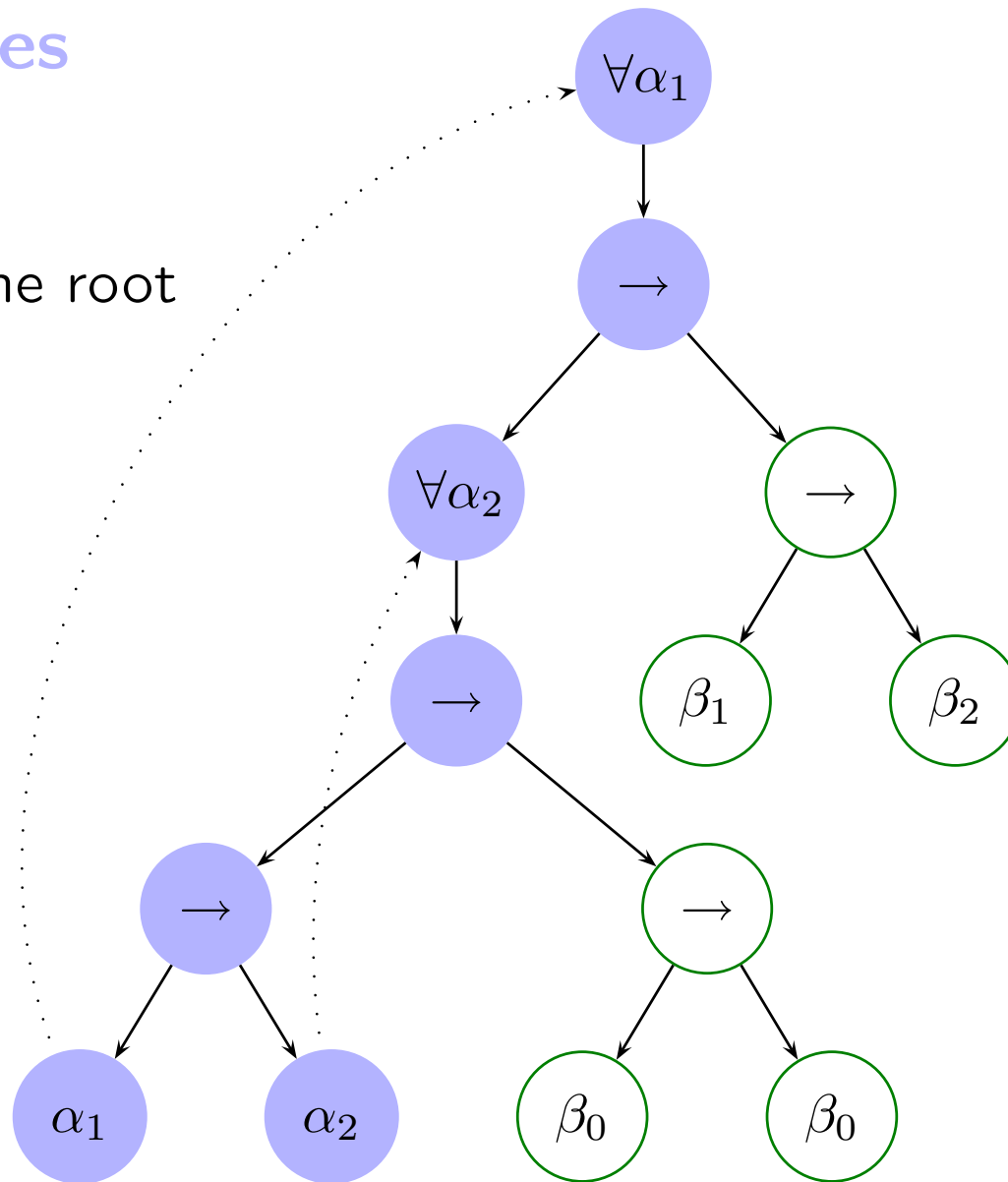
▷ Mark bound variables in Blue



$\sigma$

# Operation on shapes

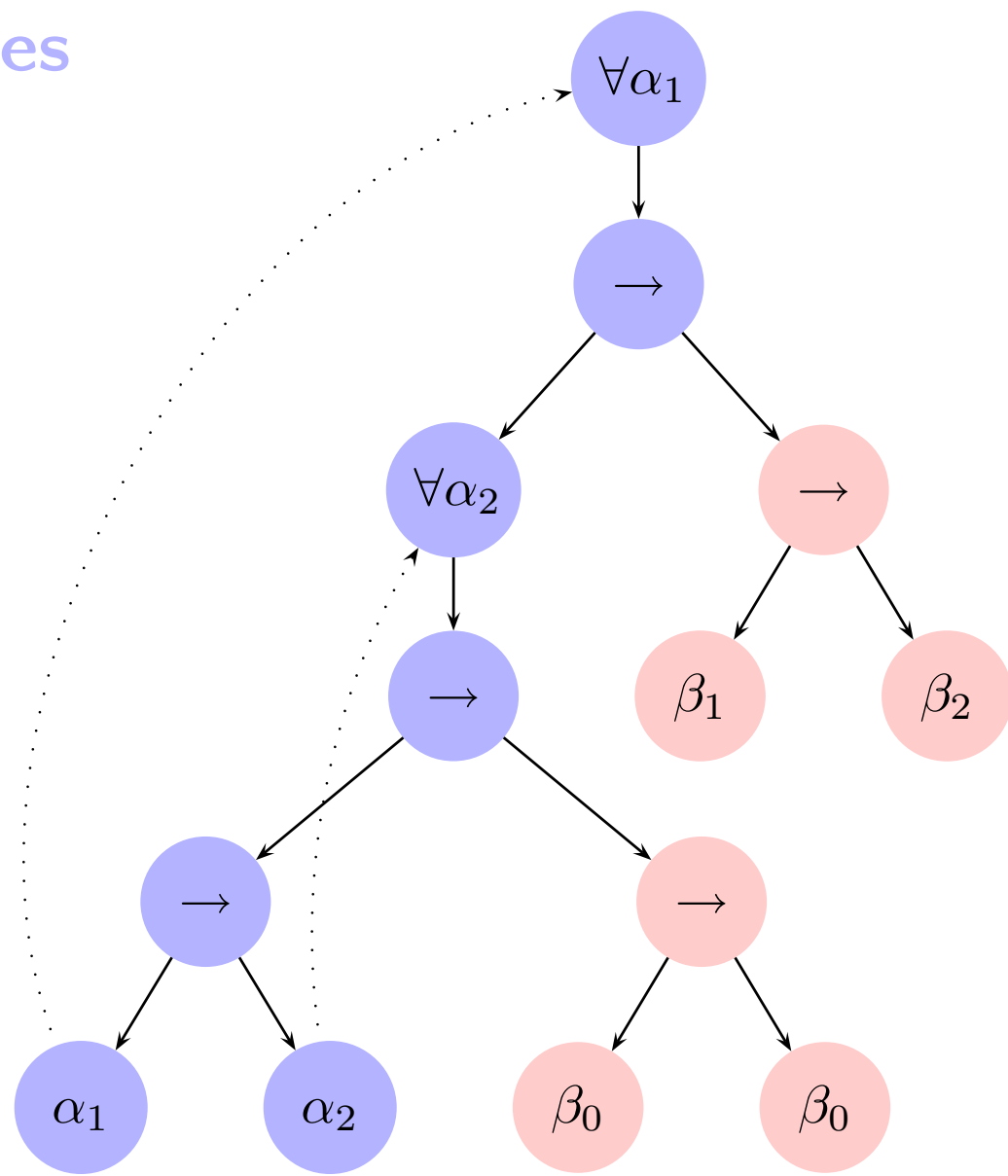
▷ Spread blue toward the root



$\lceil \sigma \rceil$

# Operation on shapes

▷ Mark the rest in red

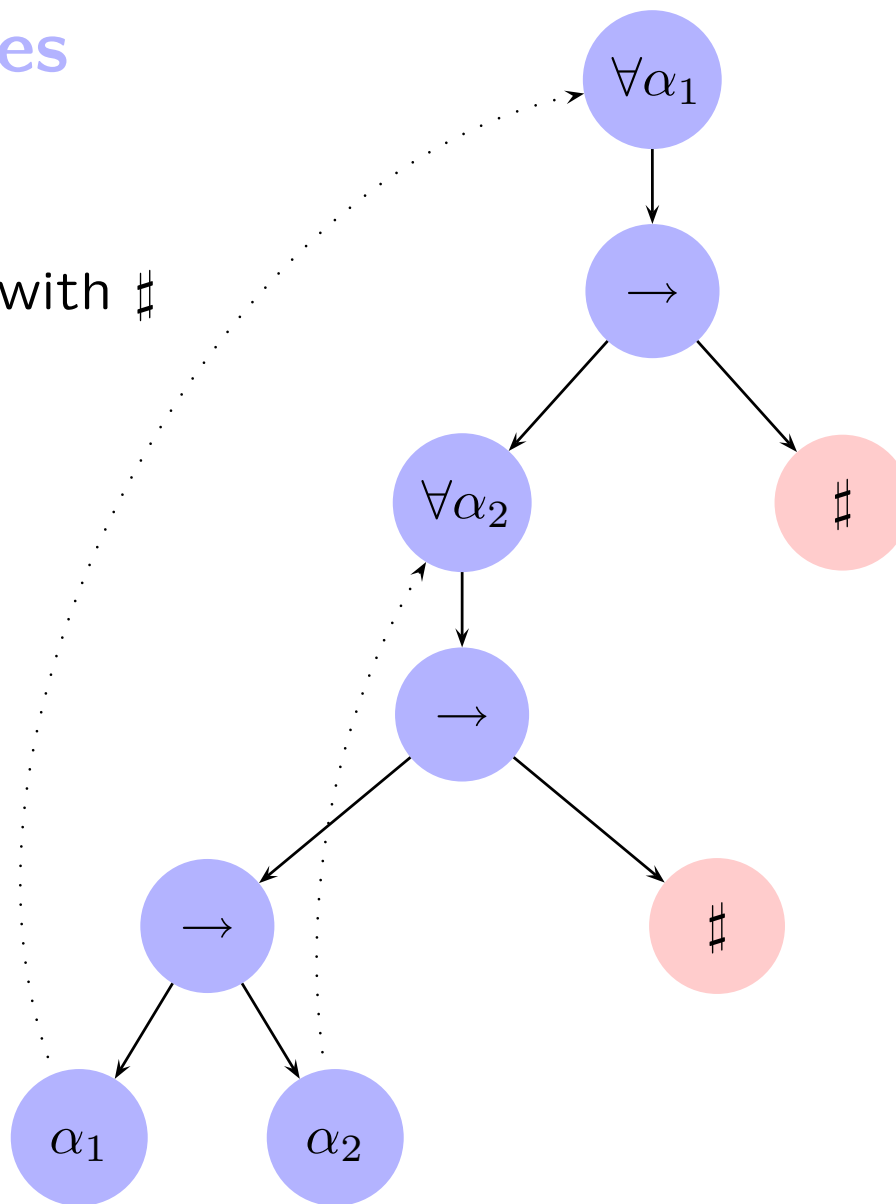


$[\sigma]$

# Operation on shapes

▷ Replace red subtrees with  $\#$

$\lceil \sigma \rceil$

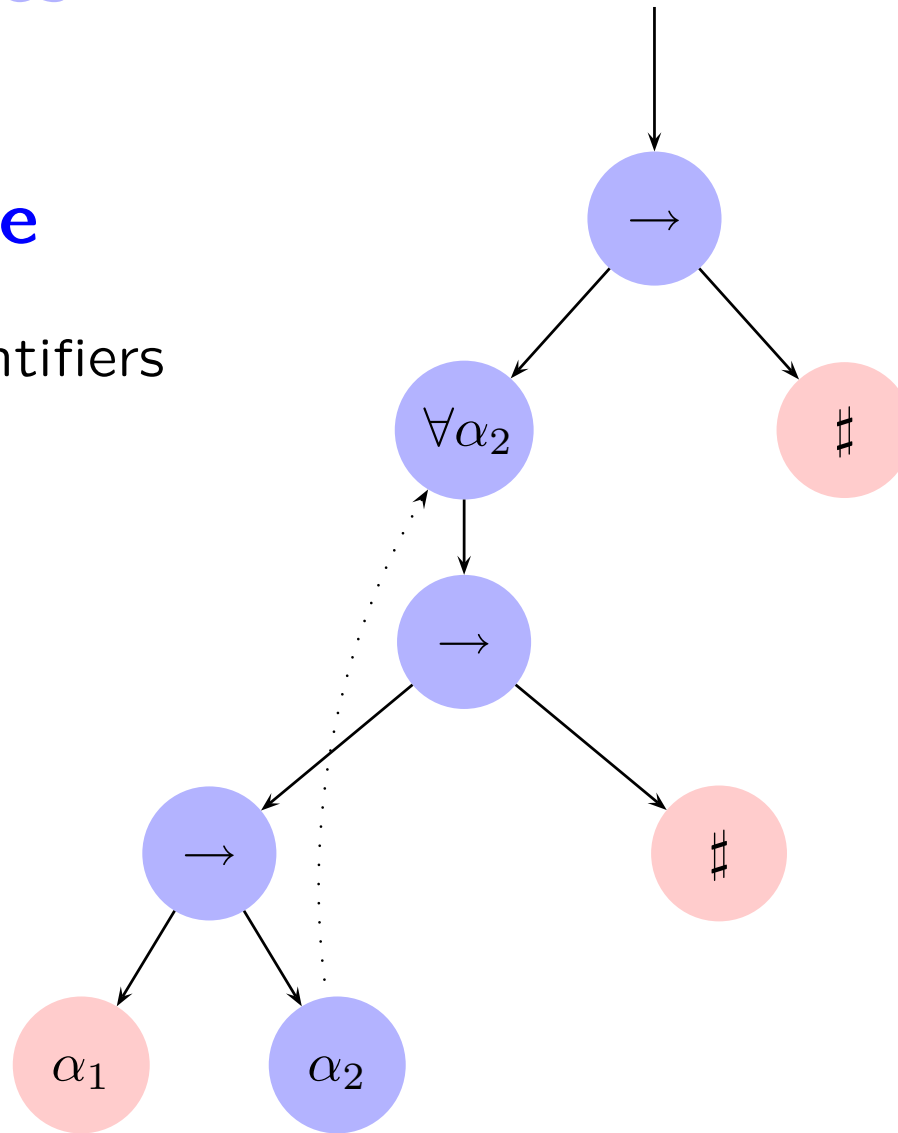


# Operation on shapes

## ▷ Stripping a shape

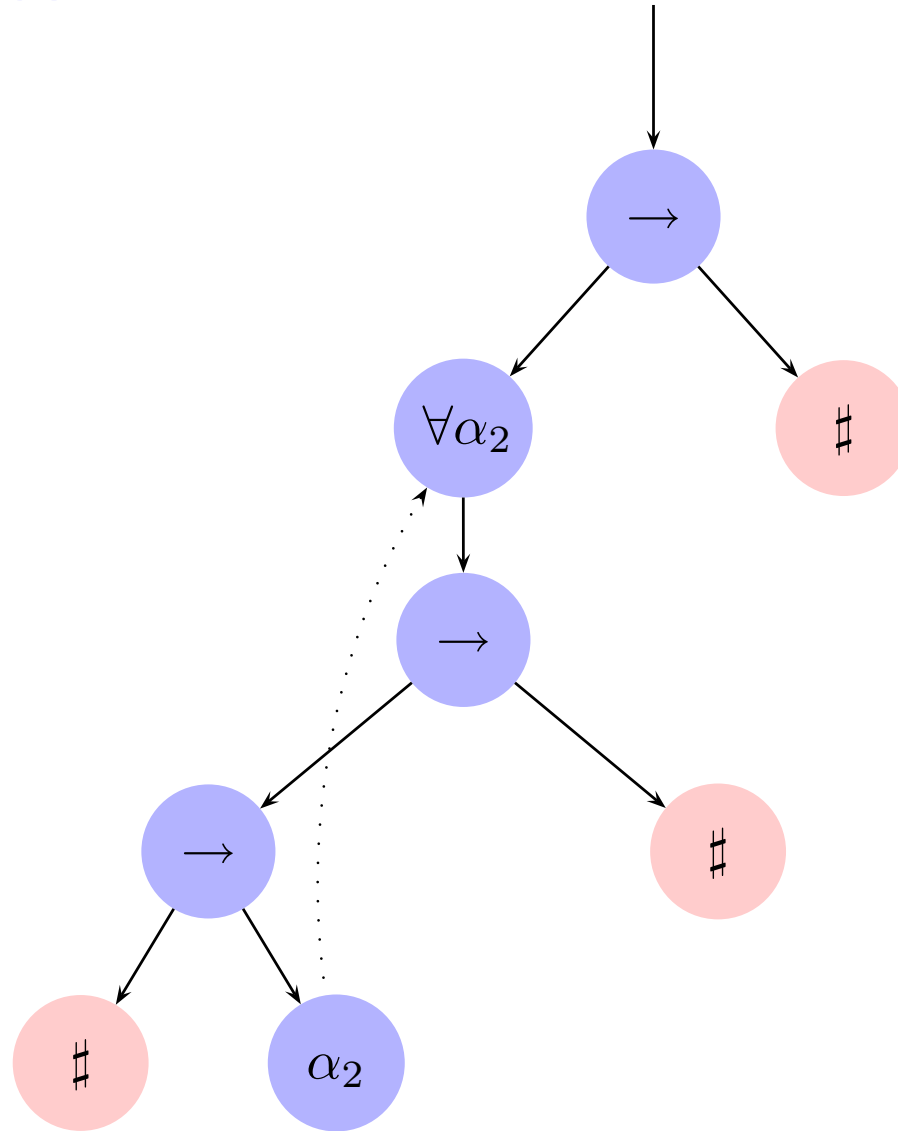
Remove toplevel quantifiers

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



# Operation on shapes

▷ Reshape



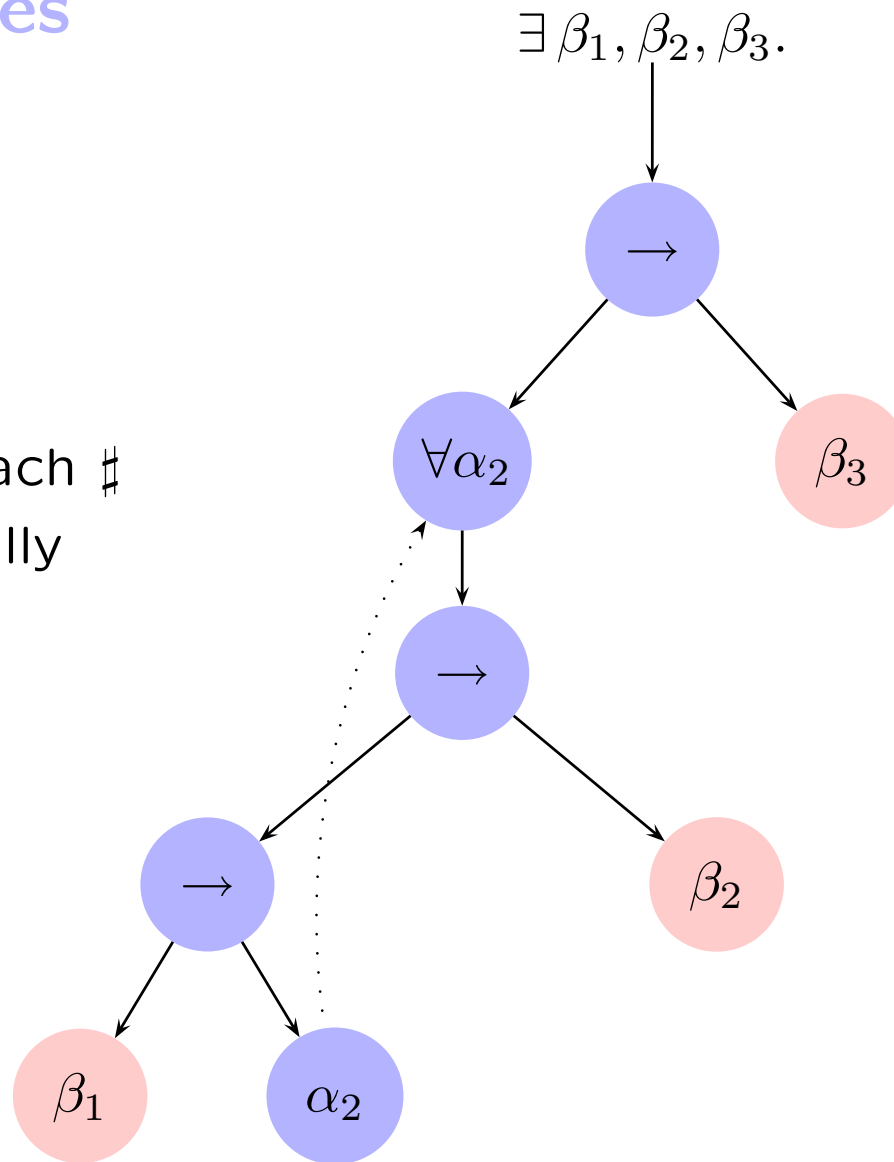
$$\mathcal{S} = \sigma^b$$

# Operation on shapes

## ▷ Building an annotation

from a shape: turn each  $\#$  into a fresh existentially quantified variable.

$[S]$



**Lemma 1** *If  $\Gamma \vdash t : \sigma$  then  $\Gamma \vdash \llbracket t \rrbracket : \sigma$ .*

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,  $\vdash \lambda z. z z : \sigma \rightarrow \sigma$  for types  $\sigma$  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  $\#$  for unknown shapes.

## Allow missing annotations on source terms

$$t ::= \dots \mid \text{let } z = t_1 \text{ in } t_2 \mid t_1 t_2 \mid \lambda z : \theta . t$$

and, as a counterpart, annotations on function parameters.

## Return a term with missing annotations filled in.

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

inference mode

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

checking mode

## Typechecking $SF_{ML}$ by elaboration into $CF_{ML}$

$$\Gamma \vdash_{SF_{ML}} t : \sigma \iff \Gamma \vdash_{\uparrow} t : [\sigma] \Rightarrow t' \wedge \Gamma \vdash_{CF_{ML}} t' : \sigma$$

Var-I      Var-C      Gen-C      Let<sub>α</sub>-I      Let<sub>α</sub>-C      Let-I      Let-C      Fun<sub>α</sub>-C      Fun<sub>α</sub>-I

## Example of easy rules

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

$$\text{Fun-I} \quad \frac{\Gamma, z : \# \vdash_{\uparrow} t : \mathcal{S} \Rightarrow t'}{\Gamma \vdash_{\uparrow} \lambda z. t : \# \rightarrow \mathcal{S} \Rightarrow \lambda z. t'}$$

## More challenging rules: applications

App-C

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

App<sub>α</sub>-C

App<sub>α</sub>-I

App-I

Var-I      Var-C      Gen-C      Let<sub>α</sub>-I      Let<sub>α</sub>-C      Let-I      Let-C      Fun<sub>α</sub>-C      Fun<sub>α</sub>-I

## Example of easy rules

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

$$\text{Fun-I} \quad \frac{\Gamma, z : \# \vdash_{\uparrow} t : \mathcal{S} \Rightarrow t'}{\Gamma \vdash_{\uparrow} \lambda z . t : \# \rightarrow \mathcal{S} \Rightarrow \lambda z . t'}$$

## More challenging rules: applications

App-C-Variant

$$\frac{\Gamma \vdash_{\uparrow} t_1 : \mathcal{S} \Rightarrow t'_1 \quad \mathcal{S}^b = \mathcal{S}_2 \rightarrow \mathcal{S}'_1 \quad \Gamma \vdash_{\uparrow} t_2 : \mathcal{S}'_2 \Rightarrow t'_2 \quad \mathcal{S}'_1 \lesssim \mathcal{S}_1 \quad \mathcal{S}'_2 \lesssim \mathcal{S}_2}{\Gamma \vdash_{\downarrow} t_1 t_2 : \mathcal{S}_1 \Rightarrow t'_1 (t'_2 : \lfloor \mathcal{S}_2 \rfloor)}$$

App<sub>α</sub>-C

App<sub>α</sub>-I

App-I

## A significant restriction...

- ▶ apply  $(\lambda f . \lambda z . f z)$  (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:  $(\lambda z . z)$  *auto* need to be annotated, even though the argument `z` is not used polymorphically.

## 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.

►  $(\lambda z. z : (\forall \alpha. \alpha \rightarrow \alpha \triangleright \sigma \rightarrow \sigma))$  (*auto* :  $\sigma$ )

where  $\sigma$  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.

## Predicative system, effect-free semantics

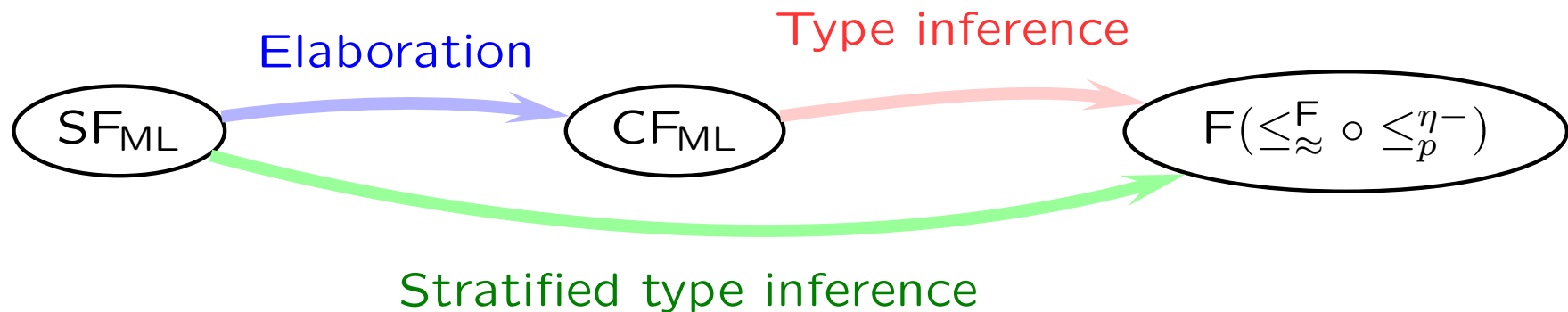
About as expressive as [Peyton Jones and Shields, 2003]  
(which is implemented in the Haskell compiler).

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

Still, some annotations remain unpleasant.

## With predicative polymorphism

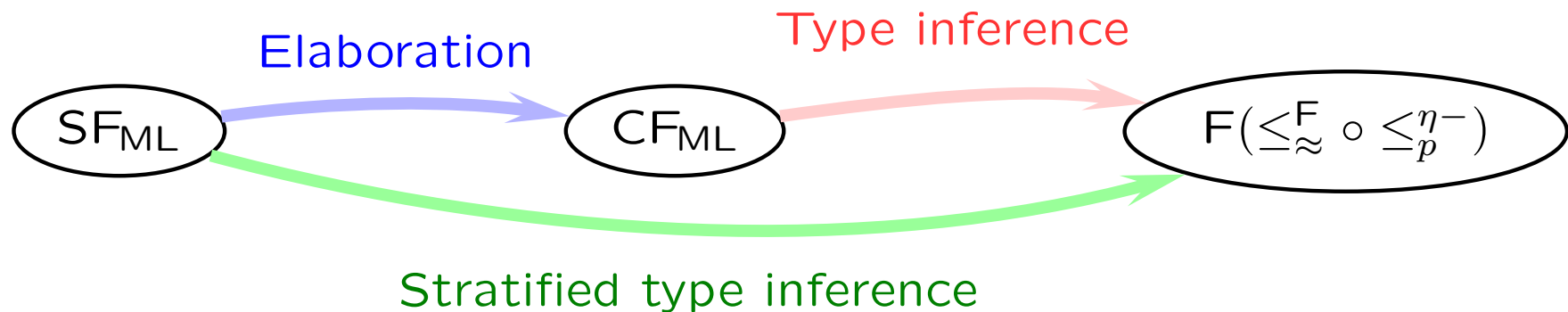
- ▶ naive elaboration of coercions is not very satisfactory.
- ▶ tension between elaboration of predicative polymorphism and impredicative polymorphism at application nodes.
- ▶ more *ad hoc* rules may be used, e.g. “smart applications” that treats multi-applications of the form  $z\ t_1 \dots t_n$ 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



## Stratified type inference is good

- ▶ Each side is easier to specify and understand, separately.
- ▶ SF<sub>ML</sub> is simple, but not entirely satisfactory.
- ▶ While CF<sub>ML</sub> 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?



**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.
- ▶  $ML^F$  is theoretically more involved but also significantly more powerful than  $SF_{ML}$  and maybe a more promising direction, because it really addresses the instantiate/generalize dilemma.

Thank you.



## Explicit introduction of polymorphism

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

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

*auto id*

**Available in OCaml !**

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.

```
type id  $\alpha = Id$  of  $\forall \alpha. (\alpha \rightarrow \alpha)$ 
```

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

```
let id = Id (fun x  $\rightarrow$  x)
```

```
let auto (Id f) = f f
```

```
auto id
```

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.

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

## 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.



- [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

---

32(1)/23

Var-I

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

Var-C

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

# Elaboration rules

32(2)/23

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

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

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

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

Let<sub>a</sub>-I

$$\frac{\Gamma \vdash_{\downarrow} t_1 : [\sigma] \Rightarrow t'_1 \quad \Gamma, z : [\sigma] \vdash_{\uparrow} t_2 : \mathcal{S}_2 \Rightarrow t'_2}{\Gamma \vdash_{\uparrow} \text{let } z = (t_1 : \exists \bar{\beta}. \sigma) \text{ in } t_2 : \mathcal{S}_2 \Rightarrow \text{let } z = (t'_1 : \exists \bar{\beta}. \sigma) \text{ in } t'_2}$$

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

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

Let<sub>a</sub>-C

$$\frac{\Gamma \vdash_{\downarrow} t_1 : [\sigma] \Rightarrow t'_1 \quad \Gamma, z : [\sigma] \vdash_{\downarrow} t_2 : \mathcal{S}_2 \Rightarrow t'_2}{\Gamma \vdash_{\downarrow} \text{let } z = (t_1 : \exists \bar{\beta}. \sigma) \text{ in } t_2 : \mathcal{S}_2 \Rightarrow \text{let } z = (t'_1 : \exists \bar{\beta}. \sigma) \text{ in } t'_2}$$

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

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

Let-C

$$\frac{\Gamma \vdash_{\uparrow} t_1 : \mathcal{S}_1 \Rightarrow t'_1 \quad \Gamma, z : \mathcal{S}_1 \vdash_{\downarrow} t_2 : \mathcal{S}_2 \Rightarrow t'_2}{\Gamma \vdash_{\downarrow} \text{let } z = t_1 \text{ in } t_2 : \mathcal{S}_2 \Rightarrow \text{let } z = (t'_1 : [\mathcal{S}_1]) \text{ in } t'_2}$$

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

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

Let-I

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

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<sub>a</sub>-C

$$\frac{\Gamma, z : [\sigma] \vdash_{\downarrow} t : \mathcal{S}_1 \Rightarrow t'}{\Gamma \vdash_{\downarrow} \lambda z : \exists \beta. \sigma. t : \mathcal{S}_2 \rightarrow \mathcal{S}_1 \Rightarrow \lambda z. \text{let } z = (z : \exists \bar{\beta}. \sigma) \text{ in } t'}$$

Fun<sub>a</sub>-I

$$\frac{\Gamma, z : [\sigma] \vdash_{\uparrow} t : \mathcal{S} \Rightarrow t'}{\Gamma \vdash_{\uparrow} \lambda z : \exists \bar{\beta}. \sigma. t : [\sigma] \rightarrow \mathcal{S} \Rightarrow \lambda z. \text{let } z = (z : \exists \bar{\beta}. \sigma) \text{ in } t'}$$

Fun-I

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

App<sub>a</sub>-C

$$\frac{\Gamma \vdash_{\downarrow} t_1 : [\sigma] \rightarrow \mathcal{S} \Rightarrow t'_1 \quad \Gamma \vdash_{\downarrow} t_2 : [\sigma] \Rightarrow t'_2}{\Gamma \vdash_{\downarrow} t_1 (t_2 : \exists \bar{\alpha}. \sigma) : \mathcal{S} \Rightarrow t'_1 (t'_2 : \exists \bar{\alpha}. \sigma)}$$

App<sub>a</sub>-C

$$\frac{\Gamma \vdash_{\downarrow} t_1 : [\sigma] \rightarrow \mathcal{S} \Rightarrow t'_1 \quad \Gamma \vdash_{\downarrow} t_2 : [\sigma] \Rightarrow t'_2}{\Gamma \vdash_{\downarrow} t_1 (t_2 : \exists \bar{\alpha}. \sigma) : \mathcal{S} \Rightarrow t'_1 (t'_2 : \exists \bar{\alpha}. \sigma)}$$

App<sub>a</sub>-I

$$\frac{\Gamma \vdash_{\uparrow} t_1 : \mathcal{S} \Rightarrow t'_1 \quad \mathcal{S}^b = \mathcal{S}_2 \rightarrow \mathcal{S}_1 \quad \Gamma \vdash_{\downarrow} t_2 : [\sigma] \Rightarrow t'_2}{\Gamma \vdash_{\uparrow} t_1 (t_2 : \exists \bar{\alpha}. \sigma) : \mathcal{S}_1 \Rightarrow t'_1 (t'_2 : \exists \bar{\alpha}. \sigma)}$$

App-C

$$\frac{\Gamma \vdash_{\uparrow} t_1 : \mathcal{S} \Rightarrow t'_1 \quad \mathcal{S}^b = \mathcal{S}_2 \rightarrow \mathcal{S}_1 \quad \Gamma \vdash_{\downarrow} t_2 : \mathcal{S}_2 \Rightarrow t'_2}{\Gamma \vdash_{\downarrow} t_1 t_2 : \mathcal{R}_1 \Rightarrow t_1 (t_2 : [\mathcal{S}_2])}$$

App-I

$$\frac{\Gamma \vdash_{\uparrow} t_1 : \mathcal{S} \Rightarrow t'_1 \quad \mathcal{S}^b = \mathcal{S}_2 \rightarrow \mathcal{S}_1 \quad \Gamma \vdash_{\downarrow} t_2 : \mathcal{S}_2 \Rightarrow t'_2}{\Gamma \vdash_{\uparrow} t_1 t_2 : \mathcal{S}_1 \Rightarrow t_1 (t_2 : [\mathcal{S}_2])}$$

- ▶ 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.

$$\begin{aligned} \llbracket x : \rho \rrbracket &\longrightarrow x \preceq \rho \\ \llbracket \lambda z. t : \alpha \rrbracket &\longrightarrow \exists \beta_1 \beta_2. (\llbracket \lambda z. t : \beta_1 \rightarrow \beta_2 \rrbracket \wedge \beta_1 \rightarrow \beta_2 \leq \alpha) \\ \llbracket \lambda z. t : \sigma_2 \rightarrow \sigma_1 \rrbracket &\longrightarrow \text{let } z : \sigma_2 \text{ in } \llbracket t : \sigma_1 \rrbracket \\ \llbracket t_1 (t_2 : \exists \bar{\beta}. \sigma_2) : \rho_1 \rrbracket &\longrightarrow \exists \bar{\beta}. (\llbracket t_1 : \sigma_2 \rightarrow \rho_1 \rrbracket \wedge \llbracket t_2 : \sigma_2 \rrbracket) \\ \llbracket \text{let } z = (t_1 : \exists \bar{\beta}. \sigma_1) \text{ in } t_2 : \rho_2 \rrbracket &\longrightarrow \text{let } z : \forall \bar{\beta} [\llbracket t_1 : \sigma_1 \rrbracket]. \sigma_1 \text{ in } \llbracket t_2 : \rho_2 \rrbracket \\ \llbracket t : \forall \bar{\alpha}. \rho \rrbracket &\longrightarrow \forall \bar{\alpha}. \llbracket t : \rho \rrbracket \end{aligned}$$

## 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  $\exists \bar{\beta}. (C \wedge \sigma \leq \sigma')$
4.  $\leq$  constraints are interpreted by  $\leq_p^{\eta^-}$

$\tau \leq \tau' \longrightarrow \tau = \tau'$	Refl
$\sigma_1 \rightarrow \sigma_2 \leq \sigma'_1 \rightarrow \sigma'_2 \longrightarrow \sigma'_1 \leq \sigma_1 \wedge \sigma_2 \rightarrow \sigma'_2$	Arrow
$\forall \alpha. \sigma \leq \rho \longrightarrow \exists \alpha. (\sigma \leq \rho)$	All-E
$\sigma \leq \forall \alpha. \sigma' \longrightarrow \forall \alpha. (\sigma \leq \sigma')$	All-I

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

## Inference is first order

- ▶ No meta variables for  $\sigma$  or  $\rho$ , only  $\alpha$  for  $\tau$ .
- ▶ Polymorphic shapes are only checked.

