Freezing Bidirectional Typing

Frost -- A novel approach to bidirectional type inference for first-class polymorphism

Wenhao Tang [1], Shengyi Jiang [2], Bruno C. d. S. Oliveira [2], Sam Lindley [1]

ML family workshop, 16th Oct 2025

1 THE UNIVERSITY of EDINBURGH

2 香 港 大 學 THE UNIVERSITY OF HONG KONG

First-Class Polymorphism (FCP)

	where quantifiers can appear	which types to use for instantiation	example
Let polymorphism (prenex polymorphism)	top level	monotypes	`∀a. a → a`
Higher-rank polymorphism	anywhere	monotypes	`(∀a. a → a) → Int`

First-class polymorphism (impredicative polymorphism)	anywhere	polytypes	`single id : List (∀a.a→a)`
			`where single : ∀a.a → List a`
			`id : ∀a.a → a`

Type Inference for FCP

	where quantifiers can appear	which types to use for instantiation	example
First-class polymorphism	anywhere	polytypes	`single id : List (∀a.a→a)`

- What do we want to infer?
  - where to introduce quantifiers `E[fun x → x]`
  - where to instantiate quantifiers `E[id]`
  - which polytype to use for instantiation `E[id]`
  - which polytype to use for unannotated function parameters `E[fun x → M]`
- Full type inference for FCP is undecidable
- Lots of different approaches in the literature based on different intuitions and heuristics
- Why yet another one?

Initial Motivation

- Better type inference for **Modal Effect Types**

OOPSLA

Sat 18 Oct 2025 10:45 - 11:00 at Orchid East - Type 2

★ Modal Effect Types

Effect handlers are a powerful abstraction for defining, customising, and composing computational effects. Statically ensuring that all effect operations are handled requires some form of effect system, but using a traditional effect system would require adding extensive effect annotations to the millions of lines of existing code in these languages. Recent proposals seek to address this problem by removing the need for explicit effect polymorphism. However, they typically rely on fragile syntactic mechanisms or on introducing a separate notion of second-class function. We introduce a novel semantic approach based on modal effect types.


Wenhao Tang
The University of Edinburgh
United Kingdom


Stephen Dolan
Jane Street
United Kingdom


Sam Lindley
The University of Edinburgh
United Kingdom


Leo White
Jane Street
United Kingdom


Daniel Hillerström
Category Labs and The University of Edinburgh
United Kingdom


Anton Lorenzen
University of Edinburgh
United Kingdom

Initial Motivation

- Better type inference for **Modal Effect Types**
- 👀 *Type inference for modal types is similar to type inference for first-class polymorphism*

- where to introduce **modalities**
- where to instantiate **modalities**
- which **modal type** to use for instantiation
- which **modal type** to use for unannotated function parameters

- where to introduce **quantifiers**
- where to instantiate **quantifiers**
- which **polytype** to use for instantiation
- which **polytype** to use for unannotated function parameters

Key Observation

- **Bidirectional type information flow** is useful to answer these questions
- But existing approaches do not make use of info bidirectionally as much as we hope
- In particular, for function application M N there are two directions of information flow
- **Frost** allows information to *flow* back and forth and uses *freezing* to control its direction

- Why called Frost: Frost forms when vapour *flows* to cold surfaces and *freezes* into ice crystals

Different Directions of Information Flow in Previous Approaches

Flowing from Fun to Arg

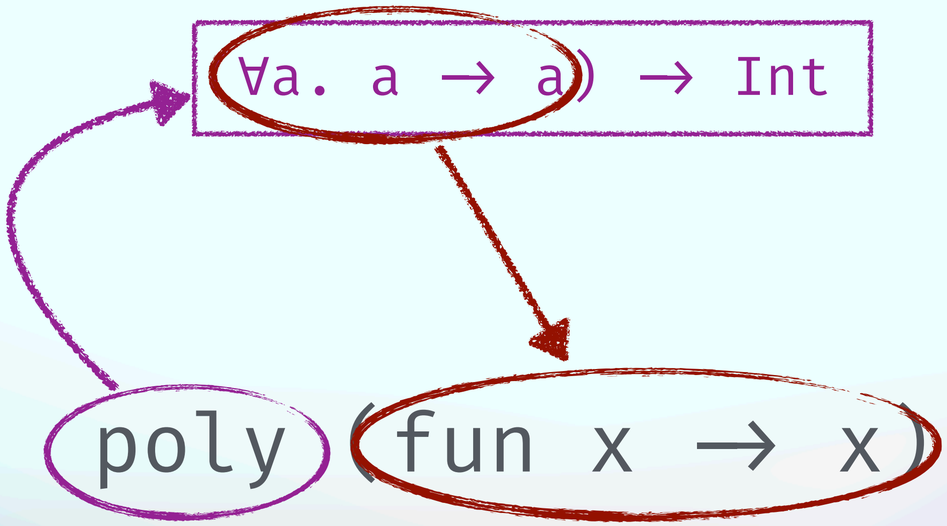
$$\frac{\Gamma \vdash M \Rightarrow A \qquad A \leq A_1 \to B \qquad \Gamma \vdash N \Leftarrow A_1}{\Gamma \vdash M\ N \Rightarrow B}$$

Flowing from Fun to Arg (1)

Where to introduce universal quantifiers

```
poly : (∀a. a → a) → Int
```

$$\frac{\Gamma \vdash M \Rightarrow A \qquad A \preceq A_1 \to B \qquad \Gamma \vdash N \Leftarrow A_1}{\Gamma \vdash M\ N \Rightarrow B}$$



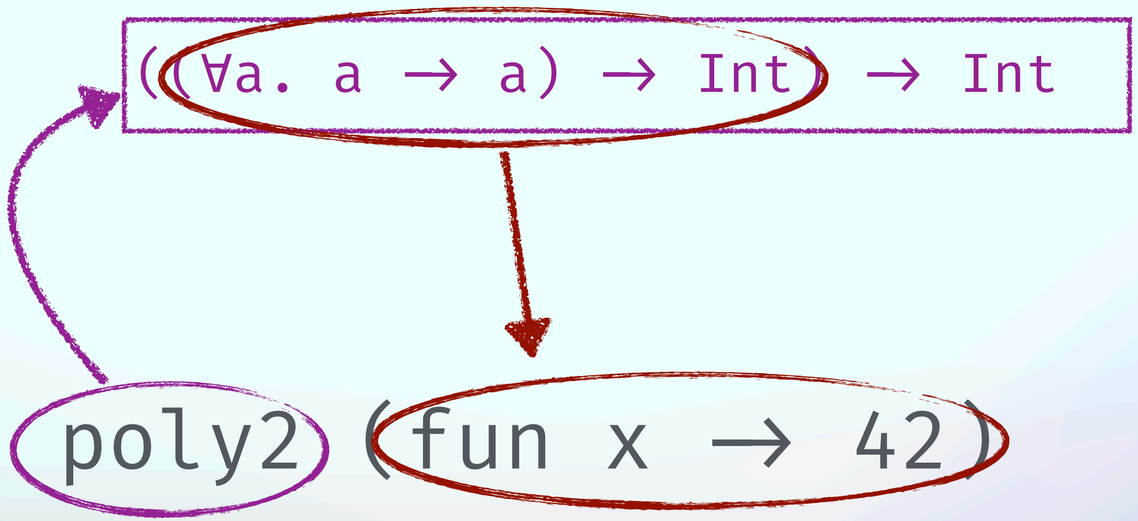
implicit introduction of ∀a

Flowing from Fun to Arg (2)

Which polytypes to use for function parameters

```
poly2 : ((∀a. a → a) → Int) → Int
```

$$\frac{\Gamma \vdash M \Rightarrow A \qquad A \preceq A_1 \to B \qquad \Gamma \vdash N \Leftarrow A_1}{\Gamma \vdash M\ N \Rightarrow B}$$



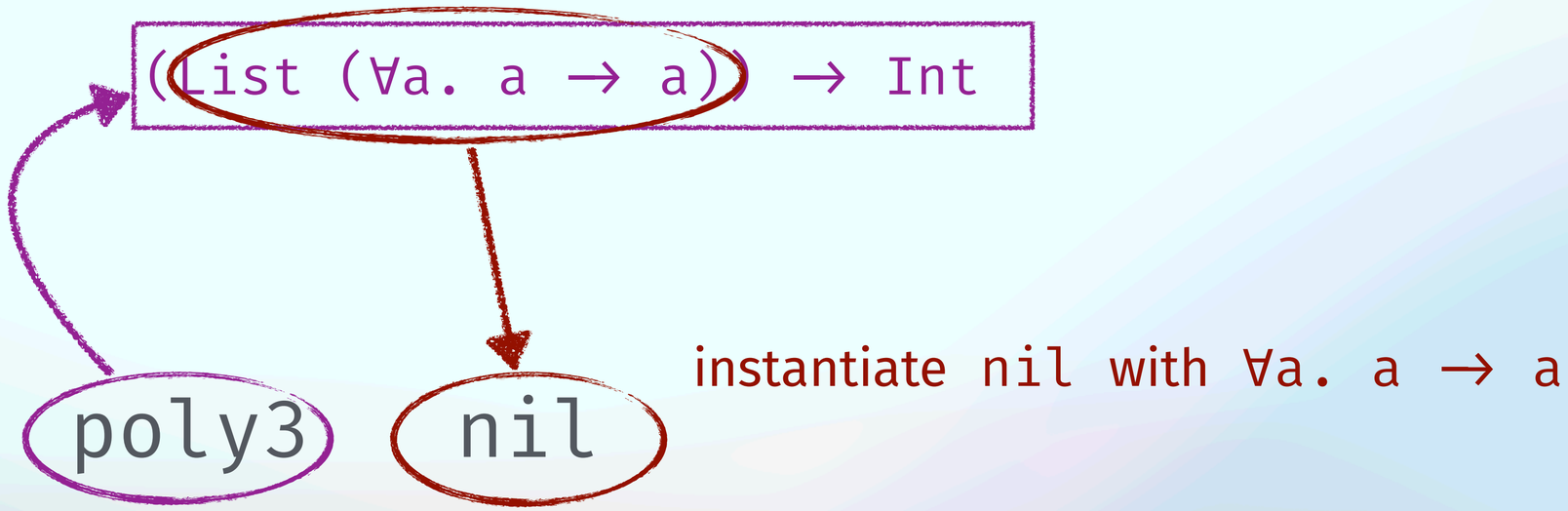
the argument x has type (∀a. a → a)

Flowing from Fun to Arg (3)

Which polytype to use for instantiation

```
poly3 : (List (∀a. a → a)) → Int
nil   : ∀b. List b
```

$$\frac{\Gamma \vdash M \Rightarrow A \qquad A \preceq A_1 \to B \qquad \Gamma \vdash N \Leftarrow A_1}{\Gamma \vdash M\ N \Rightarrow B}$$



Flowing from Arg to Fun

$$\frac{\Gamma \mid \Psi \vdash N \Rightarrow A \qquad \Gamma \mid A, \Psi \vdash M \Rightarrow A \to B}{\Gamma \mid \Psi \vdash M\ N \Rightarrow B}$$

Let Arguments Go First [Xie and Oliveira 2018]

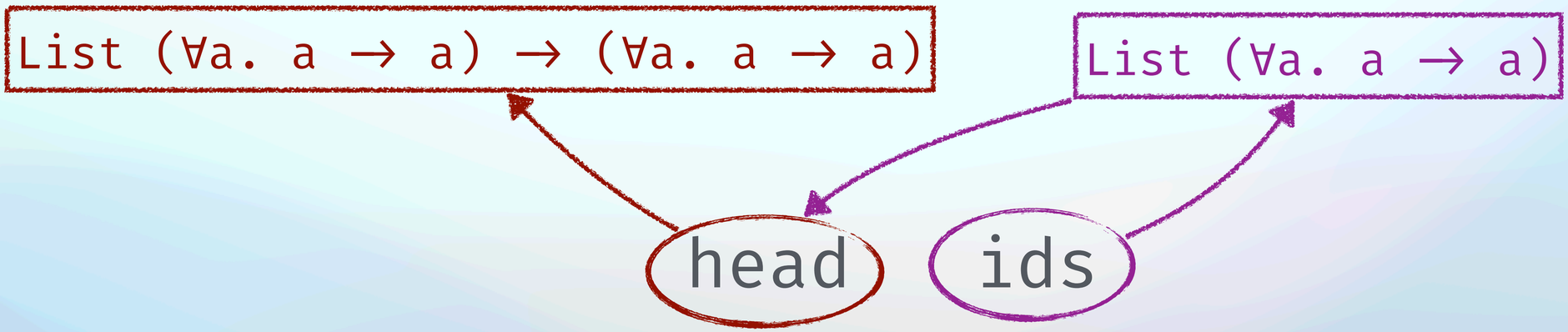
Flowing from Arg to Fun (1)

Which polytype to use for instantiation

```
head : ∀a. List a → a
ids : List (∀a. a → a)
```

$$\frac{\Gamma \mid \Psi \vdash N \Rightarrow A \qquad \Gamma \mid A, \Psi \vdash M \Rightarrow A \to B}{\Gamma \mid \Psi \vdash M\ N \Rightarrow B}$$

instantiate head with the polytype `∀a. a → a`

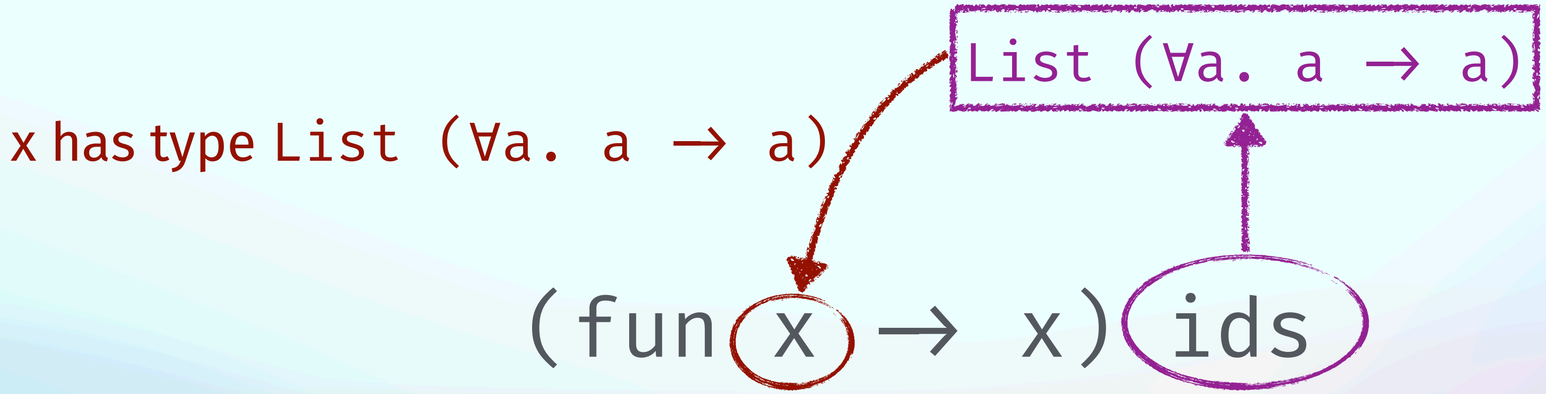


Flowing from Arg to Fun (2)

Which polytype to use for function parameter

```
ids : List (∀a. a → a)
```

$$\frac{\Gamma \mid \Psi \vdash N \Rightarrow A \qquad \Gamma \mid A, \Psi \vdash M \Rightarrow A \to B}{\Gamma \mid \Psi \vdash M\ N \Rightarrow B}$$

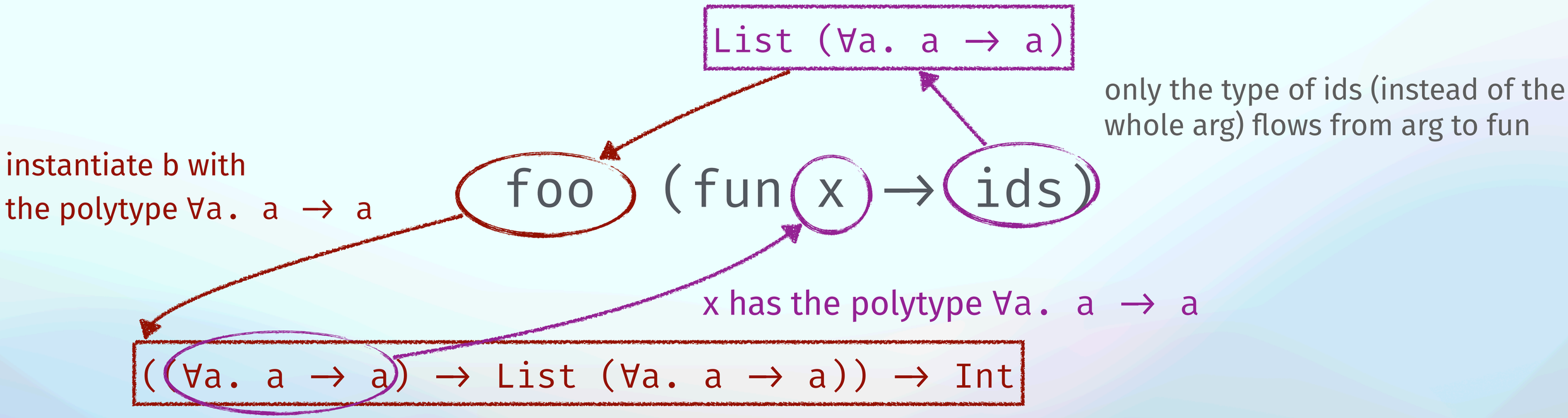


What if we want **both** directions ?

both directions

Sometimes we'd like to flow back and forth

```
foo : ∀b. (b → List b) → Int
ids : List (∀a. a → a)
```



But the previous two rules cannot propagate ***partial type information*** from arg to fun

Ghosts and Skeletons

Frost supports flowing back and forth by

- first passing **skeletons** from arg to fun
- then passing types from fun to arg

skeletons are types with ghosts

ghosts represent unknown information

$$\frac{\Gamma \mid P, \rho \vdash M \Longrightarrow A \to B \qquad \Gamma \vdash N \Longleftarrow A}{\Gamma \mid \rho \vdash M\ N \Longrightarrow B}$$

$$\Gamma \vdash N \Longrightarrow P$$

Ghosts and Skeletons

```
foo : ∀b. (b → List b) → Int
ids : List (∀a. a → a)
```

$$\frac{\Gamma \mid P, \rho \vdash M \Rightarrow A \to B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \mid \rho \vdash M\ N \Rightarrow B}$$

$$\Gamma \vdash N \Rightarrow\!\!\!\Rightarrow P$$

property: the skeleton of a term matches any of its potential types

two universal ghosts for unknown quantifiers

👻. 👻 → 👻. List (∀a. a → a)

a ghost for an unknown type

skeleton inference

instantiate with the polytype ∀a. a → a

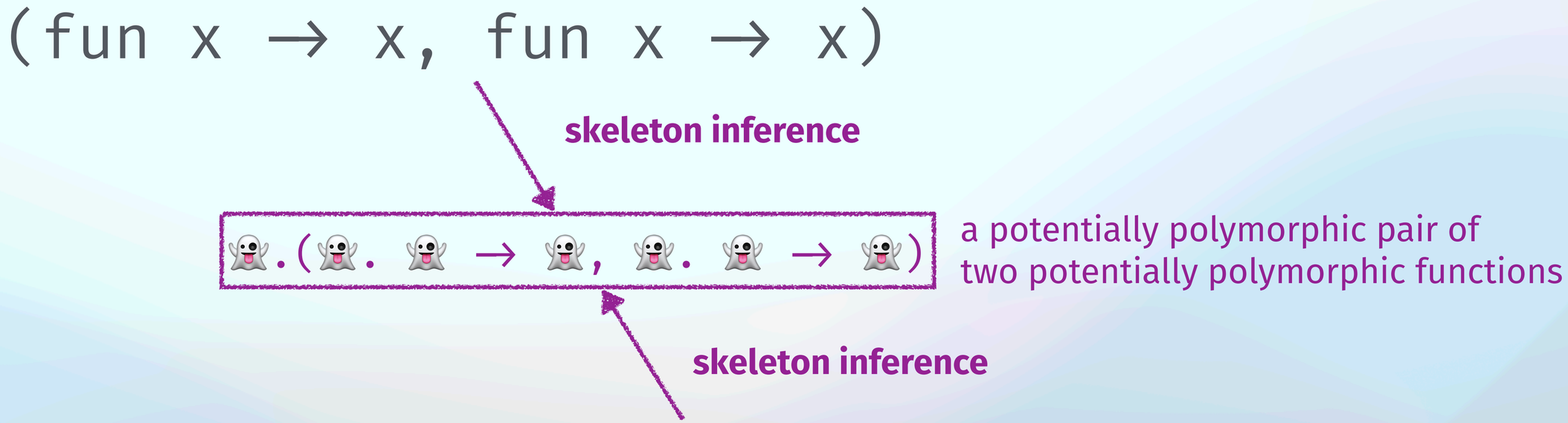
foo (fun x → ids)

x has the polytype ∀a. a → a

((∀a. a → a) → List (∀a. a → a)) → Int

More on Skeletons

- Types `A, B ::= a | 1 | A → B | ∀a.A`
- Skeletons `P,Q ::= a | 1 | P → Q | ∀a.P | 👻 | 👻.P`
- Skeleton inference rules mimic typing rules, propagating through program structures



Haunting

- For a variable binding x : A in the context, its skeleton should *be compatible with any type derived from instantiating and generalising A*
- For example,

```
id : ∀a. a → 👻

pair : ∀a b. a → Int * 👻
```

haunt | hɔːnt |

verb *[with object]*

(of a ghost) manifest itself at (a place) regularly

- `haunt(A)` gives such a skeleton for A

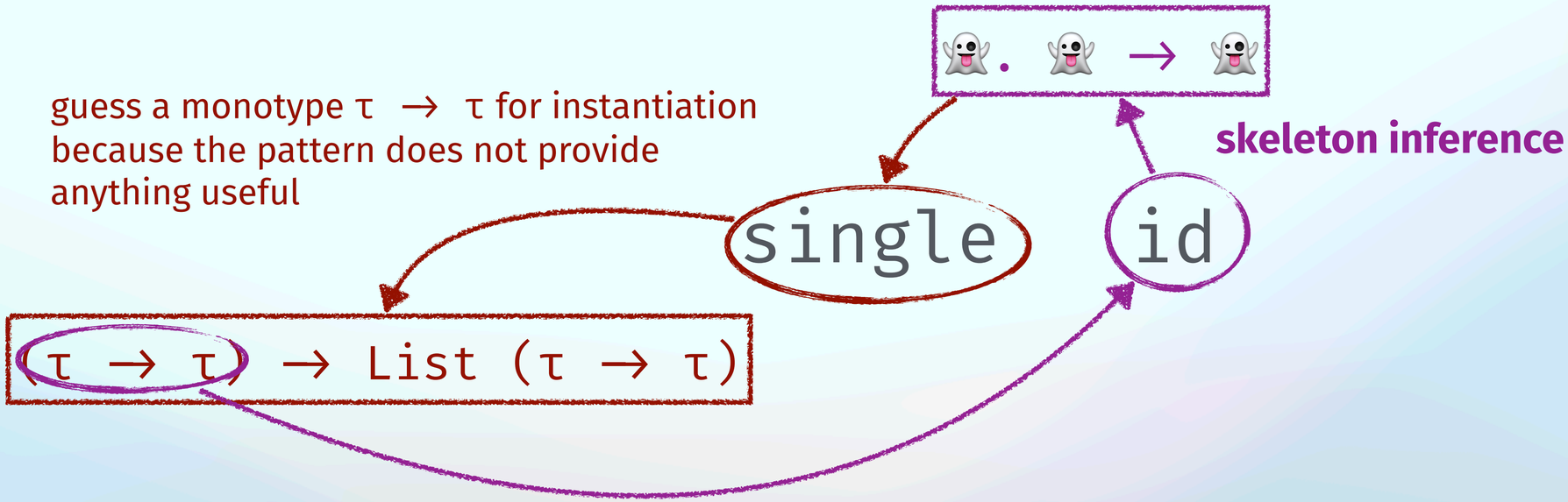
$$\begin{aligned} \text{haunt}(\forall\alpha.P) &= \text{haunt}(P[👻/\alpha]) \\ \text{haunt}(👻.P) &= \text{haunt}(P) \\ \text{haunt}(P) &= 👻.P, \text{ if } P \text{ guarded} \end{aligned}$$

What if the skeleton is not as informative as we want ?

Unsatisfying Skeletons

```
single : ∀a. a → List a
id : ∀a. a → a
```

$$\frac{\Gamma \vdash N \Longrightarrow\!\!\!\!\Rightarrow P \qquad \Gamma \mid P, \rho \vdash M \Rightarrow A \to B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \mid \rho \vdash M\ N \Rightarrow B}$$

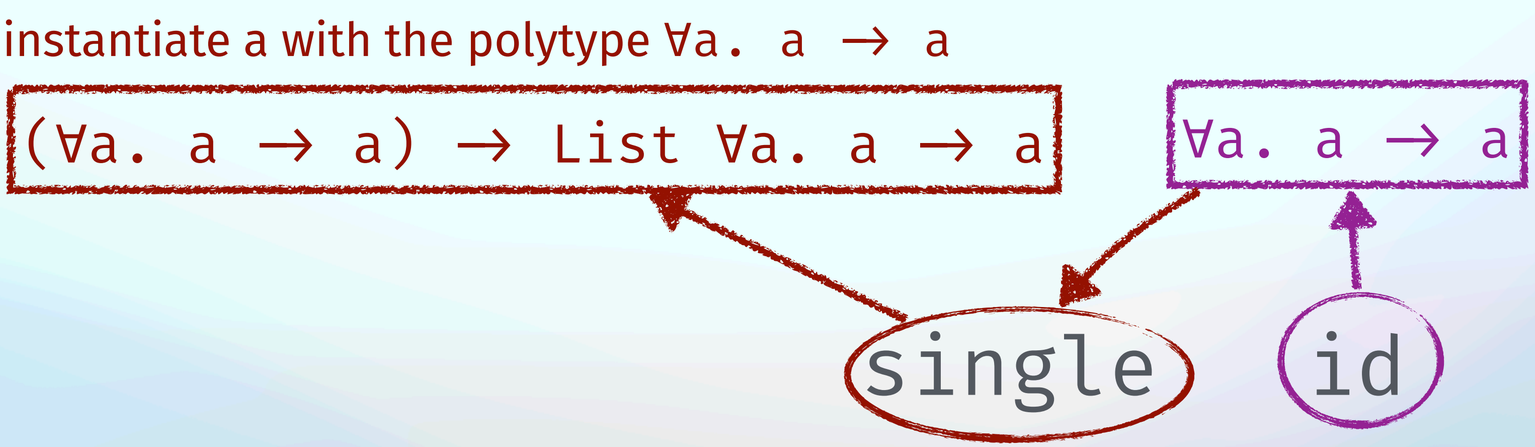


By default Frost gives `List (τ → τ)` for some monotype `τ`

Unsatisfying Skeletons

But if all information flows from arg to fun we have `List (∀a. a → a)`

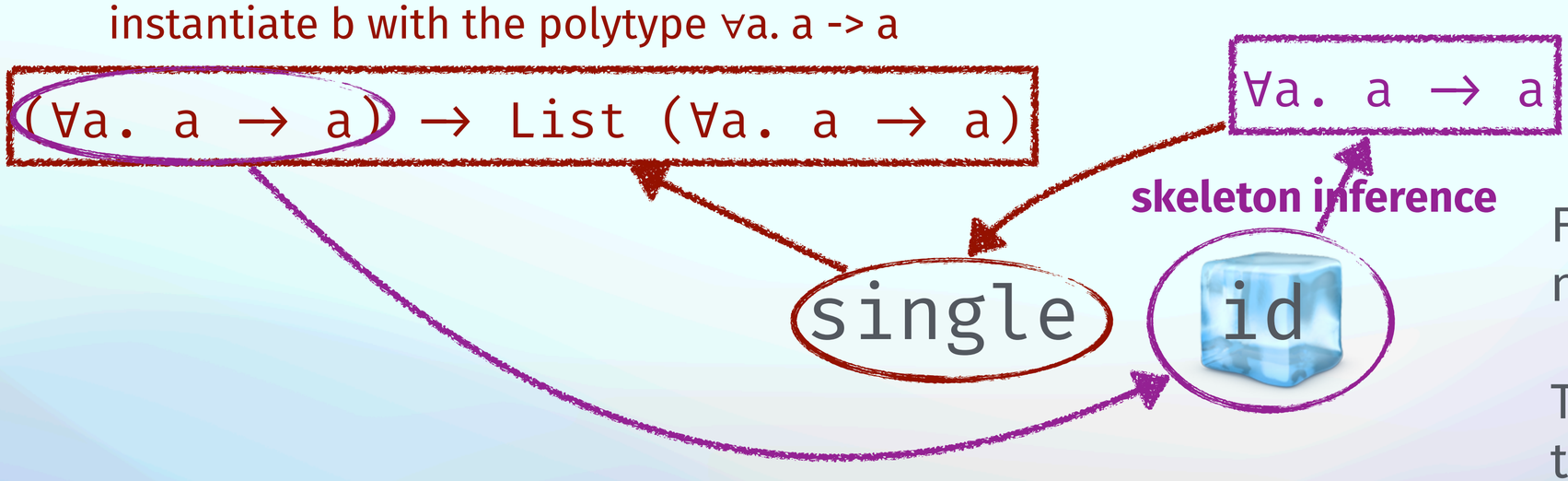
```
single : ∀a. a → List a
id : ∀a. a → a
```



Freezing Flow

Frost allows programmers to choose *which information they want to pass from arg to fun* via a **freezing** operator adapted from FreezeML [Emrich et al. 2020]

```
single : ∀b. b → List b
id : ∀a. a → a
```



Freezing a term means its type can never be influenced by its context

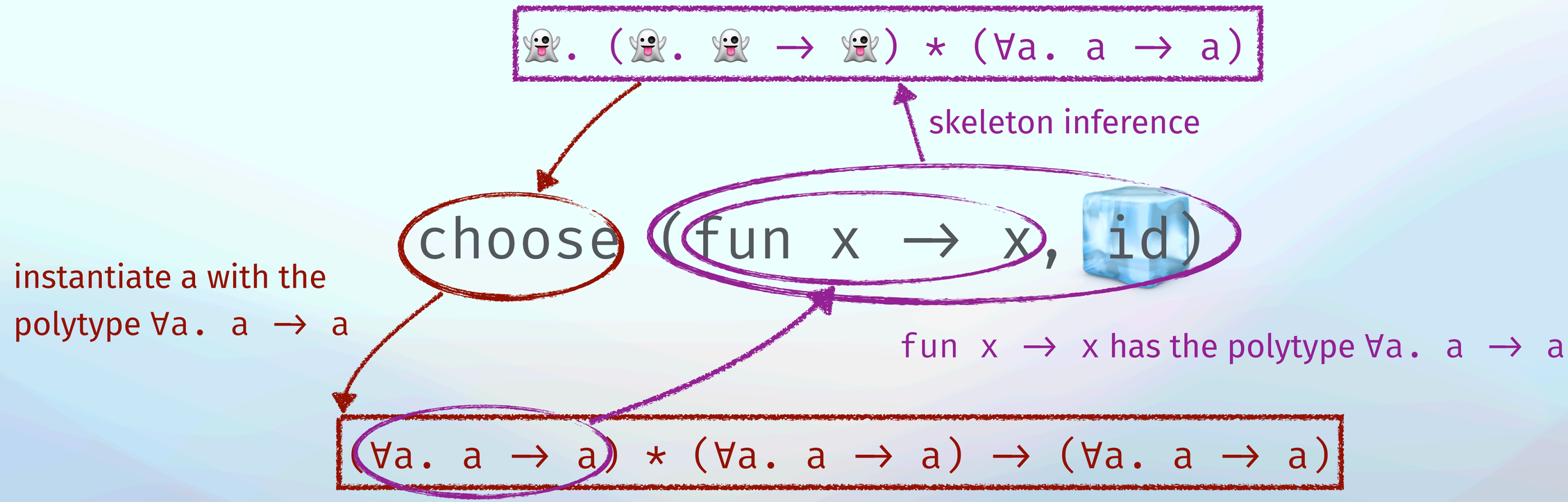
Thus we can use its type to guide the type inference of its context

Now we have `List (∀a. a → a)`

One More Example

```
choose : ∀a. a * a → a
id : ∀a. a → a
```

$$\frac{\Gamma \vdash N \Longrightarrow\!\!\!\!\Rightarrow P \qquad \Gamma \mid P, \rho \vdash M \Rightarrow A \to B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \mid \rho \vdash M\ N \Rightarrow B}$$



		Frost	QL	GI	HMF	FreezeML
A	**Polymorphic instantiation**					
A1	$\lambda x\, y.y$	●	●	●	●	●
A2	`choose id`	●	●	●	●	●
A3	`choose nil ids`	●	●	●	●	●
A4	$\lambda(x : \forall a.a \to a).x\, x$	●	●	●	●	●
A5	`id auto`	●	●	●	●	●
A6	`id auto′`	●	○	●	●	●
A7	`choose id auto`	●	●	●	●	●
A8	`choose id auto′`	○	○	○	○	○
A9	f `(choose id) ids` where $f : \forall a.(a \to a) \to$ `List` $a \to a$	●	●	○	○	◐
A10	`poly id` | `poly` $(\lambda x.x)$ | `id poly` $(\lambda x.x)$	●	●	●	●	◐
A11 [21]	$k\ (\lambda f.(f\ 42, f\ \mathtt{true}))\ xs$ where $k : \forall a.a \to$ `List` $a \to$ `Int`, $xs :$ `List` $((\forall a.a \to a) \to (\mathtt{Int}, \mathtt{Bool}))$	●	●	○	○	○
A12	`app poly id` | `revapp id poly`	●	●	●	●	◐
A13	`app runST argST` | `revapp argST runST`	●	●	●	●	◐
B	**Functions on polymorphic lists**					
B1	`tail ids` | `head ids` | `single id`	●	●	●	●	●
B2	`cons id ids`	●	●	●	●	◐
B3	`cons` $(\lambda x.x)$ `ids`	●	●	●	●	◐
B4	`append (single inc) (single id)`	●	●	●	●	●
B5 [21]	`append (single id) ids`	●	●	○	○	◐
B6	`map poly (single id)`	●	●	○	○	◐
B7	`map head (single ids)`	○	●	●	●	●
B8	`head ids true`	●	●	●	●	◐
C	**Inference of polymorphic lambda binders and generalisation points**					
C1a	$\lambda f.(f\ 42, f\ \mathtt{true})$	○	○	○	○	○
C1b	$\lambda(f : \forall a.a \to a).(f\ 42, f\ \mathtt{true})$	●	●	●	●	●
C1c [21]	$g\ (\lambda f.(f\ 42, f\ \mathtt{true}))$ where $g : ((\forall a.a \to a) \to \mathtt{Int} \times \mathtt{Bool}) \to 1$	●	●	○	○	○
C2	$r\ (\lambda x\, y.y)$ where $r : (\forall a.a \to \forall b.b \to b) \to$ `Int`	●	●	○	○	◐
E	***η*-expansion**					
E1a	$k\ h\ lst$	○	○	○	○	○
E1b	$k\ (\lambda x.h\ x)\ lst$ where $lst :$ `List` $(\forall a.\mathtt{Int} \to a \to a)$, $k : \forall a.a \to$ `List` $a \to a$, $h :$ `Int` $\to \forall a.a \to a$	●	●	●	●	◐
E2a [21]	$\lambda x.$`poly` x	○	○	○	○	○
E2b [21]	$(\lambda x.$`poly` $x) : (\forall a.a \to a) \to$ `Int` $\times$ `Bool`	●	●	○	●	○
E3a	`app poly id`	●	●	●	●	◐
E3b [21]	`app` $(\lambda x.$`poly` $x)$ `id` Frost: `app` $(\lambda x.$`poly` $x)$ ⌈`id`⌉	◐	○	○	○	○
E4a [21]	`map poly ids`	●	●	●	●	●
E4b [21]	`map` $(\lambda x.$`poly` $x)$ `ids`	●	●	○	○	○
E5a [21]	`compose poly head`	○	●	●	●	●
E5b [21]	$\lambda xs.$`poly (head` xs`)`	○	○	○	○	○

		Frost	QL	GI	HMF	FreezeML
E	***η*-expansion (new)**					
E6	$(\lambda f.$`app` $f)$ `poly id`	●	○	○	○	○
E7	$(\lambda f.$`app poly` $f)$ `id` Frost: $(\lambda f.$`app poly` $f)$ ⌈`id`⌉	◐	○	○	○	○
𝔹	***β*-expansion (new)**					
𝔹1	$(\lambda x.$`app`$)$ 42 `poly id`	●	○	●	●	◐
𝔹2	(**let** $x = 42$ **in** `app`) `poly id`	●	○	●	●	◐
𝔹3	`app` $((\lambda x.$`poly`$)$ 42$)$ `id`	●	○	●	●	◐
F	**Freezing and bidirectional information flow (new)**					
F1	$f\ (\lambda x.$`ids`$)$ where $f : \forall a.(\mathtt{Int} \to a) \to a$	●	○	●	●	●
F2	$f\ (\lambda x.(x,$ `ids`$))$ where $f : \forall a.(\forall b.b \to b \times a) \to a$	●	○	●	●	◐
F3	$f\ (\lambda x.$`ids`$)$ where $f : \forall a.((\forall b.b \to b) \to a) \to a$	●	○	○	○	○
F4	$(\lambda f.(f\ 42, f\ \mathtt{true}))$ `id` Frost: $(\lambda f.(f\ 42, f\ \mathtt{true}))$ ⌈`id`⌉	◐	○	○	○	○
F5	`pair` $(\lambda x.x)\ 42 : ((\forall a.a \to a) \to$ `Int`$) \times$ `Int`	●	●	○	○	○
F6	`choose churchNil churchIds`	●	○	●	●	●
F7	`churchHead (choose churchNil churchIds)` Frost: `churchHead (choose churchNil` ⌈`churchIds`⌉`)`	◐	○	●†	●†	○
F8a	`polys (single id)`	●	●	○	○	◐
F8b	**let** $xs =$ `single id` **in** `polys` xs Frost: **let** $xs =$ `single` ⌈`id`⌉ **in** `polys` xs	◐	○	○	○	◐

Serrano et al. 2020

Coloured Frost

What I've shown so far is what we have at the point of submitting this extended abstract

After that I realised we can give a much cleaner declarative presentation of Frost inspired by Colored Local Type Inference [Odersky et al. 2002]

$$
\begin{array}{ll}
\text{Inherited types} & A, B ::= \alpha \mid A \to B \mid \forall \alpha.A \\
\text{Synthesised types} & A, B ::= \alpha \mid A \to B \mid \forall \alpha.A \\
\text{Mixed types} & A, B ::= \alpha \mid A \to B \mid \forall \alpha.A \mid \forall \alpha.A
\end{array}
$$

quantifiers are coloured to show whether they are inherited `∀a.`A or synthesised `∀a.`A

Coloured Frost

Old rule:

first infer a skeleton for N

$$\frac{\Gamma \mid P, \rho \vdash M \Rightarrow A \to B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \mid \rho \vdash M\ N \Rightarrow B} \qquad \Gamma \vdash N \Rightarrow P$$

intuitive but kind of algorithmic

use the skeleton P to guide typing of M

New rule:

reverse the colour in A

$$\frac{\Gamma \vdash N : A \qquad \Gamma \vdash M : \widetilde{A} \to B}{\Gamma \vdash M\ N : B}$$

more declarative

More to appear

- Work in progress
- A declarative specification based on ghosts and skeletons
- A more declarative specification using colours
- A simple sound and complete type inference algorithm
- Implementation with both first-class polymorphism and modal effect types

Takeaway

- Allow type information to flow bidirectionally via **ghosts** 👻
- and let programmers to control its directions via **freezing** 🧊