Inria Cambium, Paris, 14 October 2024

Type Inference Logics

Denis Carnier, François Pottier, Steven Keuchel

Outline

Refresher

Motivation

Approach (high-level)

Approach (nitty-gritty details)

Evaluation & small demo

Conclusions

Refresher

STLC with Booleans

$$
\begin{array}{lcl}
e \in \text{Exp} & ::= & x \mid \lambda x.e \mid \lambda x{:}\tau.e \mid e\ e \mid \text{true} \\
& \mid & \text{false} \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e
\end{array}
\qquad
\begin{array}{lcl}
\tau \in \text{Ty} & ::= & \tau \Rightarrow \tau \mid \text{bool} \\
\Gamma \in \text{Ctx} & ::= & \varepsilon \mid \Gamma, x : \tau
\end{array}
$$

T-Var

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_D x : \tau \rightsquigarrow x}$$

T-App

$$\frac{\Gamma \vdash_D e_1 : \tau_1 \Rightarrow \tau_2 \rightsquigarrow e_1' \qquad \Gamma \vdash_D e_2 : \tau_1 \rightsquigarrow e_2'}{\Gamma \vdash_D e_1\ e_2 : \tau_2 \rightsquigarrow e_1'\ e_2'}$$

T-Abs-Implicit

$$\frac{\Gamma, x : \tau_1 \vdash_D e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash_D \lambda x.e : \tau_1 \Rightarrow \tau_2 \rightsquigarrow \lambda x{:}\tau_1.e'}$$

T-Abs-Explicit

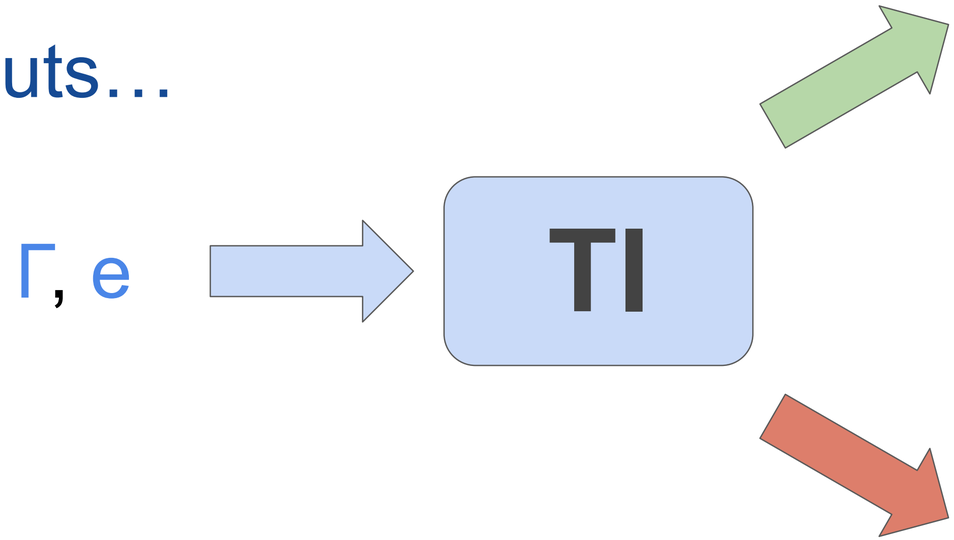
$$\frac{\Gamma, x : \tau_1 \vdash_D e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash_D \lambda x{:}\tau_1.e : \tau_1 \Rightarrow \tau_2 \rightsquigarrow \lambda x{:}\tau_1.e'}$$

Some typing rules (T-True, T-False, T-If) omitted…

What is type inference?

Inputs…

Γ, e



Γ ⊢D e : τ ~> e’

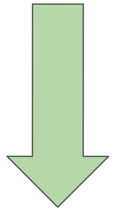
Cannot go wrong
Accept & **elaborate**

¬ Γ ⊢D e : τ ~> e’

May go wrong
Reject & report error

Elaboration

```
not =

  λx.          if x then false else true
```



- Decorate binders with their type
- (Make type application **explicit**)
- (**Explicit** passing of type class dictionaries)

```
not : bool ⇒ bool =

  λx : bool. if x then false else true
```

Metavariables and constraints

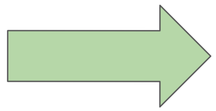
```
not : α =
  λx : β.    if x then false else true
```

```
α ~ (β ⇒ γ)   β ~ bool   γ ~ bool   γ ~ bool
```

Equality constraints
Metavariables (unification variables)
Where are they bound?

Elaboration with constraints

```
α ~ (β ⇒ γ)
β ~ bool
γ ~ bool
γ ~ bool
```



```
α := (bool ⇒ bool)
β := bool
γ := bool
```

Apply solution

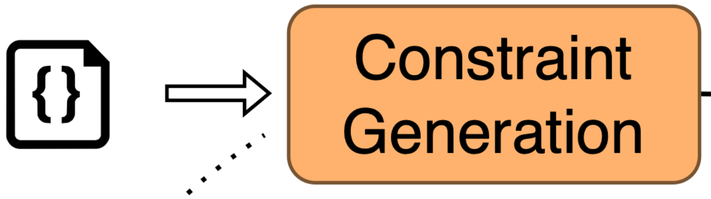
```
not : bool ⇒ bool =

  λx : bool. if x then false else true
```

Eager solving

1. Encounter a constraint
2. Solve it **fully**, immediately
   - Success: update solution
   - Failure: report

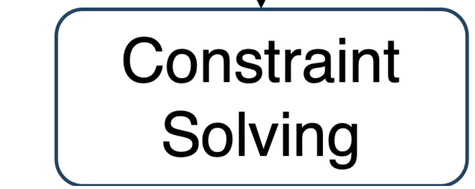
- "Order of encounter"-solving does not scale (class constraints, type families, ...)
- Full solution not optimal (occur check)

“Deferred solving”, “Phase separation”



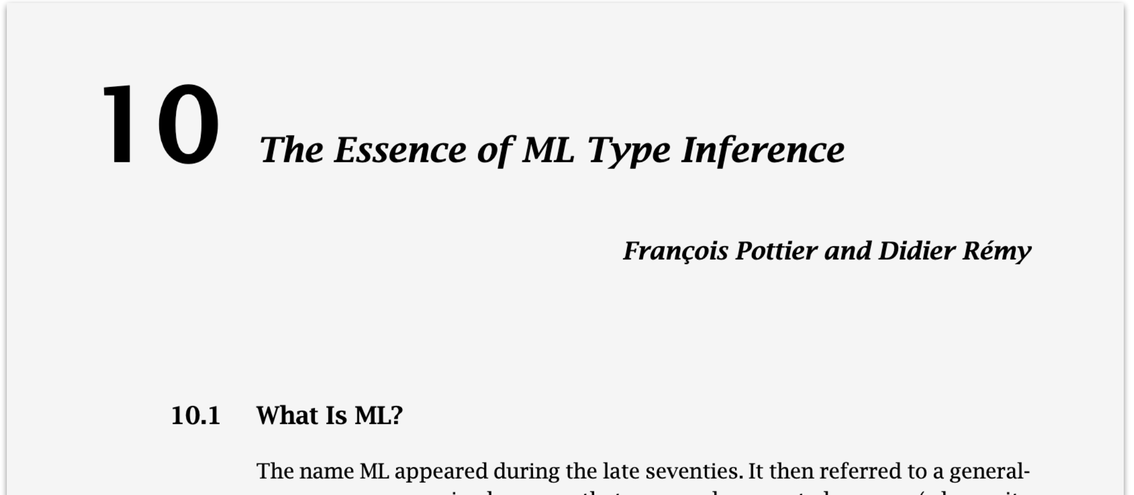
object language specific,
produces a **constraint**

C ::= T **|** ∃α.C | C ∧ C | t = s



constraint language specific,
produces a **substitution**

Logically separate, but possible to interleave phases



10 *The Essence of ML Type Inference*

François Pottier and Didier Rémy

10.1 What Is ML?

The name ML appeared during the late seventies. It then referred to a general-

François Pottier and Didier Rémy, ‘The essence of ML Type Inference’.
Advanced Topics in Types and Programming Languages, Chapter 10. MIT Press.

Solving strategy

"French approach" fully separate

- Generate all constraints first
- Solve them together
  - Mitchell Wand. 1987. A Simple Algorithm and Proof for Type Inference. Fundamenta Informaticæ 10 (1987).
  - François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In ATAPL, Benjamin C. Pierce (Ed.).

Practical approach

- Interleave generation and solving

Why use constraints?

- Clear interface: their meaning is understood
- Separation of concerns
  - multiple solver implementations: e.g. performance vs good errors
- Reduce complexity
  - big source language to
  - small constraint language
- *Really* neat!

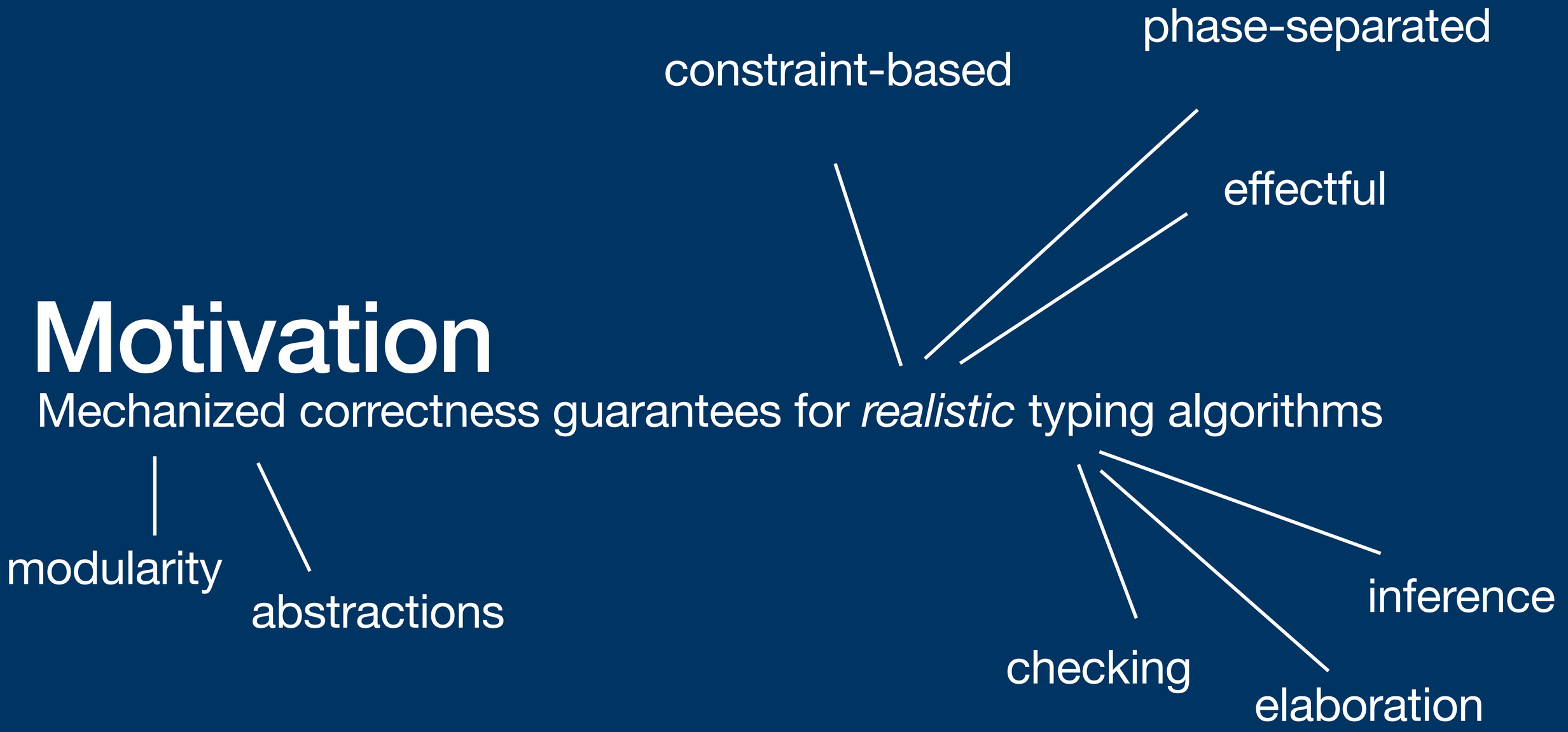
Motivation

Motivation

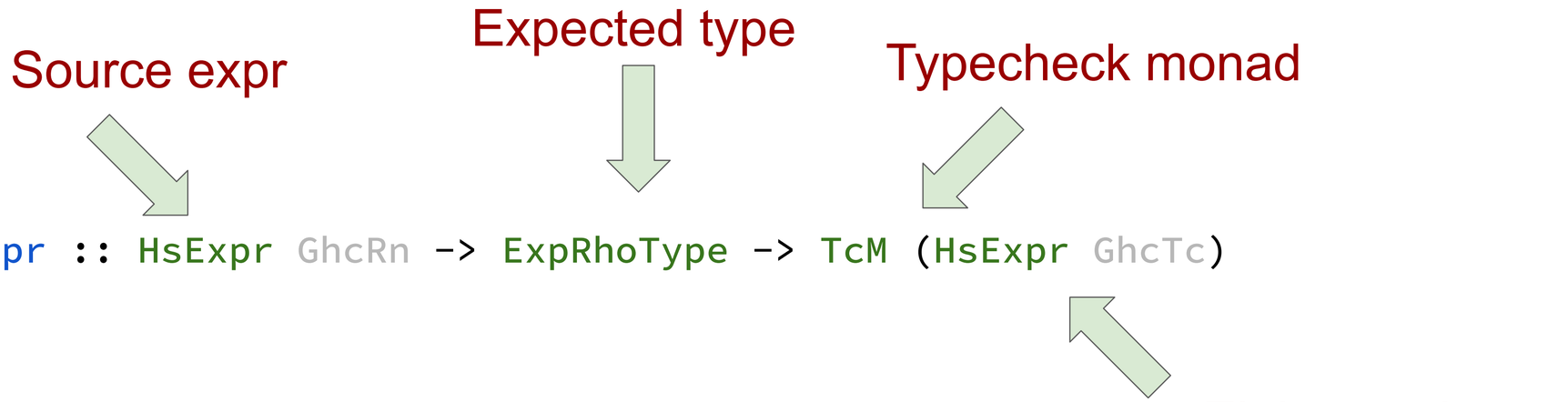
Mechanized correctness guarantees for *realistic* typing algorithms

Motivation

Mechanized correctness guarantees for *realistic* typing algorithms



Real-world example implementation: GHC




```
tcExpr (HsIf x pred b1 b2) res_ty
  = do { pred'    <- tcCheckMonoExpr pred boolTy
       ; (u1,b1') <- tcCollectingUsage $ tcMonoExpr b1 res_ty
       ; (u2,b2') <- tcCollectingUsage $ tcMonoExpr b2 res_ty
       ; tcEmitBindingUsage (supUE u1 u2)
       ; return (HsIf x pred' b1' b2') }
```

[ghc/compiler/GHC/Tc/Gen/Expr.hs]

Example mechanization: CakeML

Implementation

Source expr

```
(infer_e l ienv (If e1 e2 e3) =

  do t1 <- infer_e l ienv e1;
     () <- add_constraint l t1 (Infer_Tapp [] Tbool_num);
     t2 <- infer_e l ienv e2;
     t3 <- infer_e l ienv e3;
     () <- add_constraint l t2 t3;
     return t2
  od)
```

Synthesised type

No elaboration!?

[cakeml/compiler/inference/inferScript.sml]

Example mechanization: CakeML

Soundness statement

Breaks state monad abstraction

```
Theorem infer_e_sound:
(!l ienv e st st' tenv tenvE t extra_constraints s.
    infer_e l ienv e st = (Success t, st') ∧
    ienv_ok (count st.next_uvar) ienv ∧
    env_rel_sound s ienv tenv tenvE ∧
    t_wfs st.subst ∧
    sub_completion (num_tvs tenvE) st'.next_uvar st'.subst extra_constraints s
    ⇒
    type_e tenv tenvE e (convert_t (t_walkstar s t)))
```

No generator / solver separation
(not modular)

cakeml/compiler/inference/proofs/infer_eSoundScript.sml

Our grocery list

Real-world TI algorithms…

- use **constraints**
- use **effects** (state, option/error, …)
- captured using **abstractions** (monads, applicatives)
- have to **elaborate** to fully type-annotated programs
- need to be flexible *and* efficient: **modular** phases

Problem statement:
how can we reason abstractly and modularly about this?

Approach:
The *friendly* overview

Constraints with semantic values

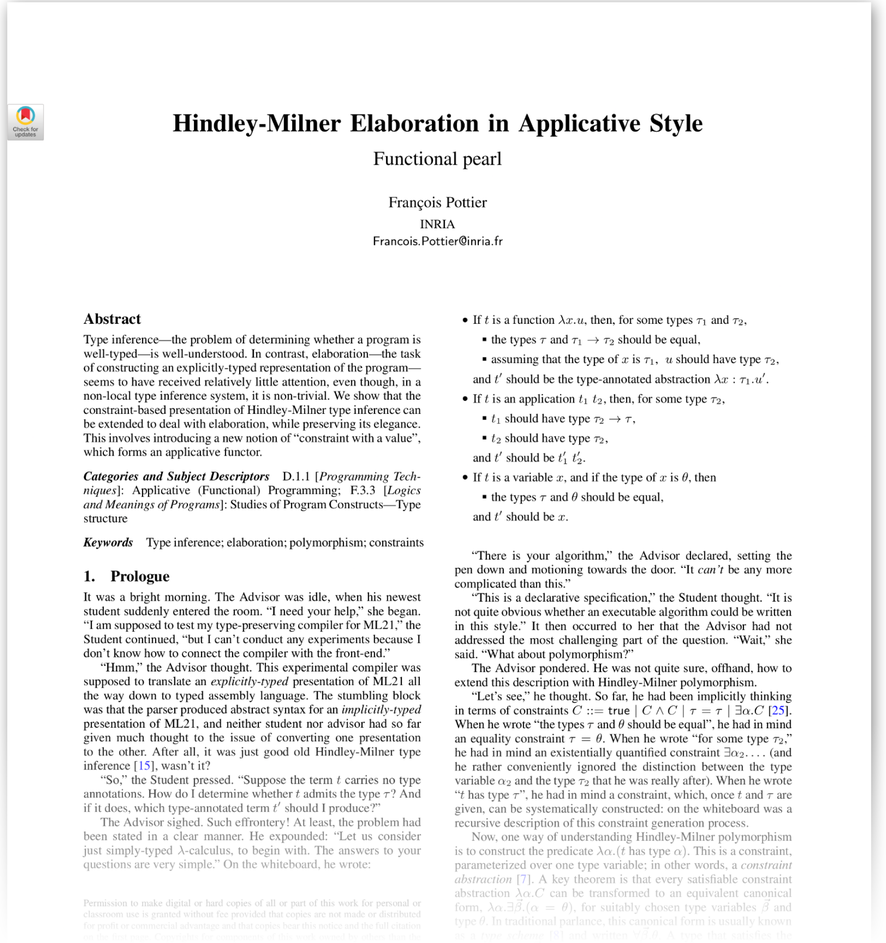
One final ingredient

François Pottier. 2014. Hindley-Milner elaboration in applicative style. In International Conference on Functional Programming (ICFP).

Constraints with values: Co A

At the same time

- Constraint that has to be solved
- A program that yields a value of type A, that may depend on the solution.



Hindley-Milner Elaboration in Applicative Style

Functional pearl

François Pottier
INRIA
Francois.Pottier@inria.fr

Abstract

Type inference—the problem of determining whether a program is well-typed—is well-understood. In contrast, elaboration—the task of constructing an explicitly-typed representation of the program—seems to have received relatively little attention, even though, in a non-local type inference system, it is non-trivial. We show that the constraint-based presentation of Hindley-Milner type inference can be extended to deal with elaboration, while preserving its elegance. This involves introducing a new notion of "constraint with a value", which forms an applicative functor.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

Keywords Type inference; elaboration; polymorphism; constraints

1. Prologue

It was a bright morning. The Advisor was idle, when his newest student suddenly entered the room. "I need your help," she began. "I am supposed to test my type-preserving compiler for ML21," the Student continued, "but I can't conduct any experiments because I don't know how to connect the compiler with the front-end."

"Hmm," the Advisor thought. This experimental compiler was supposed to translate an *explicitly-typed* presentation of ML21 all the way down to typed assembly language. The stumbling block was that the parser produced abstract syntax for an *implicitly-typed* presentation of ML21, and neither student nor advisor had so far given much thought to the issue of converting one presentation to the other. After all, it was just good old Hindley-Milner type inference [15], wasn't it?

"So," the Student pressed. "Suppose the term t carries no type annotations. How do I determine whether t admits the type τ? And if it does, which type-annotated term t' should I produce?"

The Advisor sighed. Such effrontery! At least, the problem had been stated in a clear manner. He expounded: "Let us consider just simply-typed λ-calculus, to begin with. The answers to your questions are very simple." On the whiteboard, he wrote:

- If t is a function $\lambda x.u$, then, for some types τ_1 and τ_2,
  - the types τ and $\tau_1 \to \tau_2$ should be equal,
  - assuming that the type of x is τ_1, u should have type τ_2,

  and t' should be the type-annotated abstraction $\lambda x : \tau_1.u'$.
- If t is an application $t_1\ t_2$, then, for some type τ_2,
  - t_1 should have type $\tau_2 \to \tau$,
  - t_2 should have type τ_2,

  and t' should be $t'_1\ t'_2$.
- If t is a variable x, and if the type of x is θ, then
  - the types τ and θ should be equal,

  and t' should be x.

"There is your algorithm," the Advisor declared, setting the pen down and motioning towards the door. "It *can't* be any more complicated than this."

"This is a declarative specification," the Student thought. "It is not quite obvious whether an executable algorithm could be written in this style." It then occurred to her that the Advisor had not addressed the most challenging part of the question. "Wait," she said. "What about polymorphism?"

The Advisor pondered. He was not quite sure, offhand, how to extend this description with Hindley-Milner polymorphism.

"Let's see," he thought. So far, he had been implicitly thinking in terms of constraints $C ::= \mathsf{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha.C$ [25]. When he wrote "the types τ and θ should be equal", he had in mind an equality constraint $\tau = \theta$. When he wrote "for some type τ_2," he had in mind an existentially quantified constraint $\exists \alpha_2 \ldots$ (and he rather conveniently ignored the distinction between the type variable α_2 and the type τ_2 that he was really after). When he wrote "t has type τ", he had in mind a constraint, which, once t and τ are given, can be systematically constructed: on the whiteboard was a recursive description of this constraint generation process.

Now, one way of understanding Hindley-Milner polymorphism is to construct the predicate $\lambda\alpha.(t \text{ has type } \alpha)$. This is a constraint, parameterized over one type variable; in other words, a *constraint abstraction* [7]. A key theorem is that every satisfiable constraint abstraction $\lambda\alpha.C$ can be transformed to an equivalent canonical form, $\lambda\alpha.\exists\vec{\beta}.(\alpha = \theta)$, for suitably chosen type variables $\vec{\beta}$ and type θ. In traditional parlance, this canonical form is usually known as a type scheme [14] and written $\forall\vec{\beta}.\theta$. Open than, consider the

How can we verify this kind of TI code?

> “Every monad deserves a dedicated program logic, and consequently, a proof over a monadic program ought to take place within a Floyd-Hoare logic built for the occasion.”

Pierre Nigron and Pierre-Évariste Dagand. 2021. Reaching for the Star: Tale of a Monad in Coq. In Interactive Theorem Proving (ITP) (LIPIcs, Vol. 193).

Monad interface

Let’s try!

```
class CstrM (M : Type → Type) :=

  pure   : A → M A
  (>>=)  : M A → (A → M B) → M B
  fail   : M A
  (~)    : Ty → Ty → M ()
  pick   : M Ty
```

Useful for a constraint generator?

The constraint generator

```
synth (Γ : Env) (e : Exp) : M (Ty * Exp) :=
  match e with
  | x          → if (x:τ)∈Γ then pure (τ,x) else fail
  | λx.e       → do τ₁     ← pick
                    τ₂,e' ← synth (Γ,x:τ₁) e
                    pure (τ₁⇒τ₂, λx:τ₁.e')
  | e₁ e₂      → do τ₁, e₁' ← synth Γ e₁
                    τ₂, e₂' ← synth Γ e₂
                    τ₃       ← pick
                    τ₁ ~ τ₂⇒τ₃
                    pure (τ₃, e₁' e₂')
  | ...
```

Can we implement M?

Free monad instance

Construct a free monad

- captures the syntax of operations
- without specifying semantics
- operations become term constructors
- constructors carry inputs
- constructors carry a continuation k for the output

```
data Free (A : Type) :=
  Pure (a : A)
  Fail
  Eq (τ₁ τ₂ : Ty) (k : Free A)
  Pick (k : Ty → Free A)
```

Higher-order abstract syntax representations!

What about the semantics? Cannot seem to run it!?

Do they agree? Can we prove they do?

synth (Γ : Env) (e : Exp)
: Free (Ty * Exp)

inference algorithm

T-Var
$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_D x : \tau \rightsquigarrow x}$$

T-App
$$\frac{\Gamma \vdash_D e_1 : \tau_1 \Rightarrow \tau_2 \rightsquigarrow e_1' \qquad \Gamma \vdash_D e_2 : \tau_1 \rightsquigarrow e_2'}{\Gamma \vdash_D e_1\ e_2 : \tau_2 \rightsquigarrow e_1'\ e_2'}$$

T-Abs-Implicit
$$\frac{\Gamma, x : \tau_1 \vdash_D e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash_D \lambda x.e : \tau_1 \Rightarrow \tau_2 \rightsquigarrow \lambda x{:}\tau_1.e'}$$

T-Abs-Explicit
$$\frac{\Gamma, x : \tau_1 \vdash_D e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash_D \lambda x{:}\tau_1.e : \tau_1 \Rightarrow \tau_2 \rightsquigarrow \lambda x{:}\tau_1.e'}$$

Declarative type system

Need a way to describe (compute) the infinite set of postconditions that this constraint satisfies

Denotation of constraints

Idea: Interpret the constraint as a proposition!

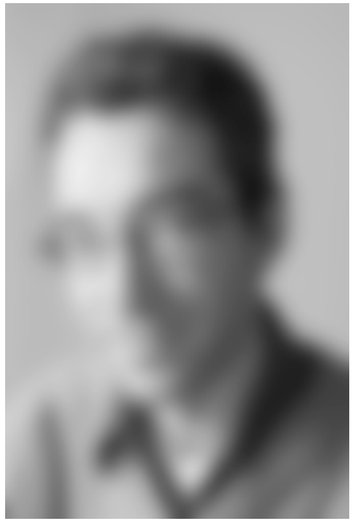
```
WP : Free A → (A → ℙ) → ℙ
WP (Pure a)      Φ  :=  Φ a
WP (Fail)        Φ  :=  ⊥
WP (Eq τ₁ τ₂ k)  Φ  :=  τ₁=τ₂ ∧ WP k Φ
WP (Pick k)      Φ  :=  ∃ τ. WP (k τ) Φ
```

Denotation of constraints

Idea: Interpret the constraint as a proposition!

```
WP : Free A → (A → ℙ) → ℙ
WP (Pure a)      Φ  :=  Φ a
WP (Fail)        Φ  :=  ⊥
WP (Eq τ₁ τ₂ k)  Φ  :=  τ₁=τ₂ ∧ WP k Φ
WP (Pick k)      Φ  :=  ∃ τ. WP (k τ) Φ
```

If **WP C Q**, then
there is an **a : A** that **C** can produce,
for which **Q** holds.



Some result of **C** satsifies **Q.**

C *can* satisfy **Q**.

“Algorithmic” typing

still nondeterministic

$$\Gamma \vdash_A e : \tau \rightsquigarrow e$$

```
Γ ⊢A e : τ ⇝ e' :=
  WP (synth Γ e)
    (λ(τ',e''). τ=τ' ∧ e'=e'')
```

Intuitively: synth *can* produce exactly the pair (τ, e’)

Correctness

synth (Γ : Env) (e : Exp)
: Free (Ty * Exp)

“algorithmic” typing

$$\textsc{T-Var}\ \frac{(x : \tau) \in \Gamma}{\Gamma \vdash_D x : \tau \rightsquigarrow x}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash_D e_1 : \tau_1 \Rightarrow \tau_2 \rightsquigarrow e_1' \qquad \Gamma \vdash_D e_2 : \tau_1 \rightsquigarrow e_2'}{\Gamma \vdash_D e_1\ e_2 : \tau_2 \rightsquigarrow e_1'\ e_2'}$$

$$\textsc{T-Abs-Implicit}\ \frac{\Gamma, x : \tau_1 \vdash_D e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash_D \lambda x.e : \tau_1 \Rightarrow \tau_2 \rightsquigarrow \lambda x{:}\tau_1.e'}$$

$$\textsc{T-Abs-Explicit}\ \frac{\Gamma, x : \tau_1 \vdash_D e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash_D \lambda x{:}\tau_1.e : \tau_1 \Rightarrow \tau_2 \rightsquigarrow \lambda x{:}\tau_1.e'}$$

“declarative typing”

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \leftrightarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$$

Correctness

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \leftrightarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$$

Correctness

Soundness

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \rightarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$$

Completeness

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \leftarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$$

Completeness

"Any derivable type is obtainable by the inference algorithm"

$$\Gamma \vdash_D e : \tau_1 \rightsquigarrow e' \rightarrow \Gamma \vdash_A e : \tau_1 \rightsquigarrow e'$$

Proof: by induction on the typing derivation.

The goal is to construct a WP statement using program logic rules for pure, eq, pick, …

Soundness

"All obtained types are valid typed"

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \rightarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$$

Proof: by destructing the WP statement?!

Soundness

"All obtained types are valid types"

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \rightarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$$

Proof: by destructing the WP statement?!

Instead: let's define an alternative semantics to WP (call it WLP…)

```
WLP : Free A → (A → ℙ) → ℙ

WLP (Pure a)     Φ  :=  Φ a
WLP (Fail)       Φ  :=  ⊤
WLP (Eq τ₁ τ₂ k) Φ  :=  τ₁=τ₂ → WLP k Φ
WLP (Pick k)     Φ  :=  ∀ τ. WLP (k τ) Φ
```

Theorem:
(WP C P → Q) ↔ (WLP C (P → Q))

Soundness

"All obtained types are valid typed"

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \rightarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$$

Proof: by ~~destructing the WP~~ constructing a WLP statement!

$$\mathrm{WLP}\ (\mathrm{synth}\ \Gamma\ e)\ (\lambda(\tau, e').\ \Gamma \vdash_D e : \tau \rightsquigarrow e')$$

Approach:
The *friendly* overview

Approach:
The ~~*friendly overview*~~
nitty-gritty details

The pesky HOAS representation

```
data Free (A : Type) :=
  Pure (a : A)
  Fail
  Eq (τ₁ τ₂ : Ty) (k : Free A)
  Pick (k : Ty → Free A)
```

Cannot rely on meta language function space for freshness and scoping of evars

Go to a first-order representation.

Worlds

enforcing well-scopedness

$$
\begin{array}{lll}
\alpha, \beta & : \mathrm{Evar} & \\
w & : \mathrm{World} & ::= \epsilon \mid w, \alpha \\
\hat{\tau} & : \hat{\mathrm{Ty}}\ w & ::= \alpha\ (\mathrm{if}\ \alpha \in w) \mid \hat{\mathrm{bool}} \mid \hat{\tau} \Rightarrow \hat{\tau} \\
\hat{\Gamma} & : \hat{\mathrm{Ctx}}\ w & ::= \varepsilon \mid \hat{\Gamma}, x : \hat{\tau}\ (\mathrm{if}\ \hat{\tau} \in \hat{\mathrm{Ty}}\ w) \\
\lceil \_ \rceil & : \forall w.\ A \to \hat{A}\ w &
\end{array}
$$

Worlds

enforcing well-scopedness

Worlds are just sets of evars

$$
\begin{array}{lll}
\alpha, \beta & : \mathrm{Evar} & \\
w & : \mathrm{World} & ::= \epsilon \mid w, \alpha \\
\hat{\tau} & : \hat{\mathrm{Ty}}\ w & ::= \alpha\ (\mathrm{if}\ \alpha \in w) \mid \hat{\mathrm{bool}} \mid \hat{\tau} \Rightarrow \hat{\tau} \\
\hat{\Gamma} & : \hat{\mathrm{Ctx}}\ w & ::= \varepsilon \mid \hat{\Gamma}, x : \hat{\tau}\ (\mathrm{if}\ \hat{\tau} \in \hat{\mathrm{Ty}}\ w) \\
\lceil \_ \rceil & : \forall w.\ A \to \hat{A}\ w &
\end{array}
$$

Worlds

enforcing well-scopedness

Worlds are just sets of evars

$$
\begin{array}{lll}
\alpha, \beta & : \mathrm{Evar} & \\
w & : \mathrm{World} & ::= \epsilon \mid w, \alpha \\
\hat{\tau} & : \hat{\mathrm{Ty}}\ w & ::= \alpha\ (\mathrm{if}\ \alpha \in w) \mid \hat{\mathrm{bool}} \mid \hat{\tau} \Rightarrow \hat{\tau} \\
\hat{\Gamma} & : \hat{\mathrm{Ctx}}\ w & ::= \varepsilon \mid \hat{\Gamma}, x : \hat{\tau}\ (\mathrm{if}\ \hat{\tau} \in \hat{\mathrm{Ty}}\ w) \\
\lceil \_ \rceil & : \forall w.\ A \to \hat{A}\ w &
\end{array}
$$

Types can only reference evars contained in the world

First-order representation

$$
\begin{aligned}
&\textbf{data}\ \hat{\mathrm{Free}}\ (A : \hat{\mathrm{Type}})\ (w : \mathrm{World}) : \mathrm{Type} :=\\
&\quad \hat{\mathrm{Pure}}\ (a : A\ w)\\
&\quad \hat{\mathrm{Fail}}\\
&\quad \hat{\mathrm{Eq}}\ (\hat{\tau}_1\ \hat{\tau}_2 : \hat{\mathrm{Ty}}\ w)\ (k : \hat{\mathrm{Free}}\ A\ w)\\
&\quad \hat{\mathrm{Pick}}\ (\alpha : \mathrm{Evar})\ (k : \hat{\mathrm{Free}}\ A\ (w, \alpha))
\end{aligned}
$$

The previously higher-order function k is now first-order

Revisiting $m \gg\!= f$

A naïve type signature for bind…

$$\forall w.\ \hat{\text{Free}}\ A\ w \to (A\ w \to \hat{\text{Free}}\ B\ w) \to \hat{\text{Free}}\ B\ w.$$

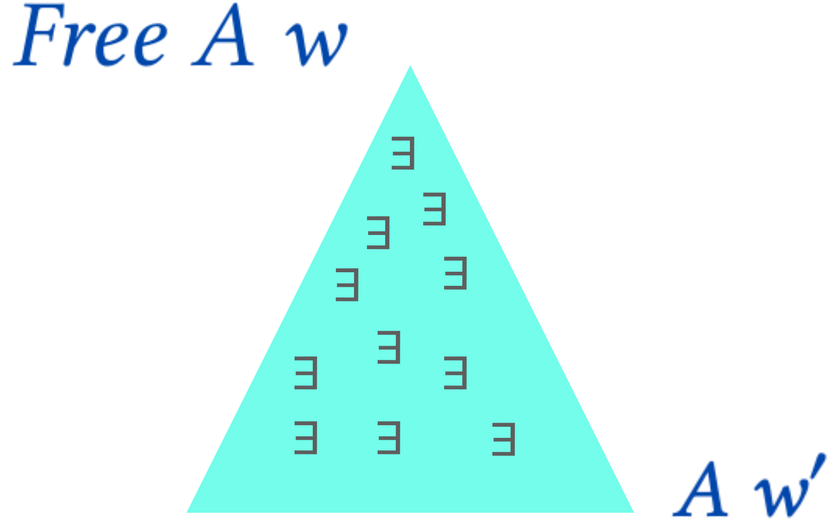
Revisiting $m \gg\!= f$

A naïve type signature for bind…

$$\forall w.\ \hat{\text{Free}}\ A\ w \to (A\ w \to \hat{\text{Free}}\ B\ w) \to \hat{\text{Free}}\ B\ w.$$

But what if the left-hand side a : A appears under binders (Pick) of new evars?

Then, when a references those evars, the scope will be violated.

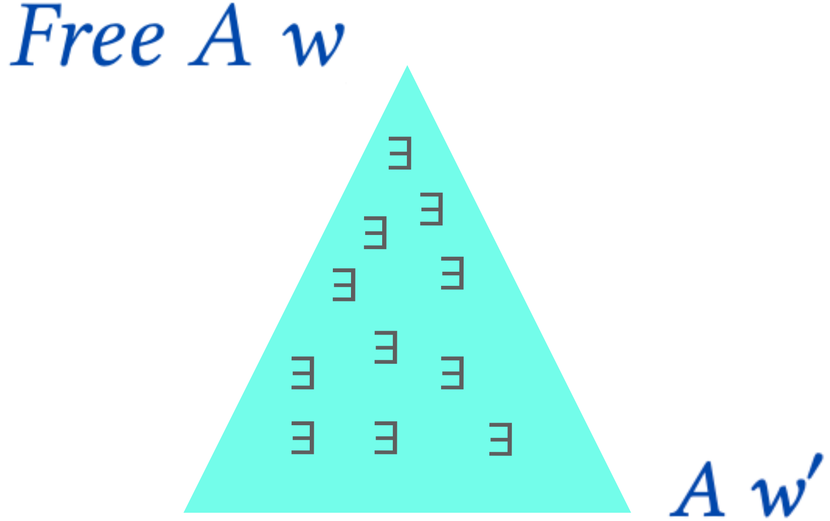


Revisiting $m \gg\!= f$

A naïve type signature for bind…

$$\forall w.\ \hat{\mathrm{F}}\mathrm{ree}\ A\ w \to (A\ w \to \hat{\mathrm{F}}\mathrm{ree}\ B\ w) \to \hat{\mathrm{F}}\mathrm{ree}\ B\ w.$$

But what if the left-hand side a : A appears under binders (Pick) of new evars?



Then, when a references those evars, the scope will be violated.

Instead, we should relax the type signature and allow scope changes

$$\forall w.\ \hat{\mathrm{F}}\mathrm{ree}\ A\ w \to (\forall w'.\, w \sqsubseteq w' \to A\ w' \to \hat{\mathrm{F}}\mathrm{ree}\ B\ w') \to \hat{\mathrm{F}}\mathrm{ree}\ B\ w.$$

Some handy notations

$$
\begin{aligned}
\models A \quad & := \forall\{w\}.\ A\ w \\
A \twoheadrightarrow B \quad & := \lambda w.\ A\ w \to B\ w \\
\Box A \quad & := \lambda w.\ \forall\{w'\}.w \sqsubseteq w' \to A\ w'
\end{aligned}
$$

worlds are a monotonic state, we pass it explicitly but use notation to hide it

$$\forall w.\ \hat{\mathrm{F}}\mathrm{ree}\ A\ w \to (\forall w'.w \sqsubseteq w' \to A\ w' \to \hat{\mathrm{F}}\mathrm{ree}\ B\ w') \to \hat{\mathrm{F}}\mathrm{ree}\ B\ w.$$

… becomes …

$$\models \hat{\mathrm{F}}\mathrm{ree}\ A \twoheadrightarrow \Box(A \twoheadrightarrow \hat{\mathrm{F}}\mathrm{ree}\ B) \twoheadrightarrow \hat{\mathrm{F}}\mathrm{ree}\ B$$

New monad interface

$$
\begin{aligned}
&\textbf{class}\ \hat{\mathrm{C}}\mathrm{strM}\ (M : \hat{\mathrm{T}}\mathrm{ype} \to \hat{\mathrm{T}}\mathrm{ype}) \\
&\quad \hat{\mathrm{p}}\mathrm{ure} \ : \models A \twoheadrightarrow M\ A \\
&\quad (\gg\!=) : \models M\ A \twoheadrightarrow \Box(A \twoheadrightarrow M\ B) \twoheadrightarrow M\ B \\
&\quad \hat{\mathrm{f}}\mathrm{ail} \ : \models M\ A \\
&\quad (\sim) \ : \models \hat{\mathrm{T}}\mathrm{y} \twoheadrightarrow \hat{\mathrm{T}}\mathrm{y} \twoheadrightarrow M\ () \\
&\quad \hat{\mathrm{p}}\mathrm{ick} \ : \models M\ \hat{\mathrm{T}}\mathrm{y}
\end{aligned}
$$

$$[\theta]\, x \leftarrow m\,;\, b \ := \ m \gg\!= (\lambda\theta\, x.\ b)$$

Free monad instance

$$
\begin{aligned}
&(\gg\!=_{\mathrm{F}}) : \models \hat{\mathrm{F}}\mathrm{ree}\ A \to \Box(A \to \hat{\mathrm{F}}\mathrm{ree}\ B) \to \hat{\mathrm{F}}\mathrm{ree}\ B \\
&\hat{\mathrm{P}}\mathrm{ure}\ a \quad\ \gg\!=_{\mathrm{F}} f := f\ \mathrm{refl}\ a \\
&\hat{\mathrm{F}}\mathrm{ail} \quad\ \gg\!=_{\mathrm{F}} f := \hat{\mathrm{F}}\mathrm{ail} \\
&\hat{\mathrm{E}}\mathrm{q}\ \hat{\tau}_1\ \hat{\tau}_2\ k \gg\!=_{\mathrm{F}} f := \hat{\mathrm{E}}\mathrm{q}\ \hat{\tau}_1\ \hat{\tau}_2\ (k \gg\!=_{\mathrm{F}} f) \\
&\hat{\mathrm{P}}\mathrm{ick}\ \alpha\ k \gg\!=_{\mathrm{F}} f := \hat{\mathrm{P}}\mathrm{ick}\ \alpha\ (k \gg\!=_{\mathrm{F}} f[\mathrm{new}_\alpha])
\end{aligned}
$$

Pushing the cont k under f requires
a weakening substitution

Constraint generation (again)

$$\hat{\mathrm{synth}} : \models \hat{\mathrm{Ctx}} \twoheadrightarrow \mathrm{Const\ Exp} \twoheadrightarrow M\ (\hat{\mathrm{Ty}} \times \hat{\mathrm{Exp}})$$

$$\begin{array}{l}
\hat{\mathrm{synth}}\ \hat{\Gamma}\ (\lambda x.e) := \\
\quad [\theta_1] \quad \hat{\tau}_1 \leftarrow \hat{\mathrm{pick}} \\
\quad [\theta_2] \quad \hat{\tau}_2, \hat{e} \leftarrow \hat{\mathrm{synth}}\ (\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)\ e \\
\qquad\quad \hat{\mathrm{pure}}\ (\hat{\tau}_1[\theta_2] \Rightarrow \hat{\tau}_2,\ \hat{\lambda} x{:}\hat{\tau}_1[\theta_2].\hat{e})
\end{array}$$

$$\begin{array}{l}
\hat{\mathrm{synth}}\ \hat{\Gamma}\ (e_1\ e_2) := \\
\quad [\theta_1] \quad \hat{\tau}_1, \hat{e}_1 \leftarrow \hat{\mathrm{synth}}\ \hat{\Gamma}\ e_1 \\
\quad [\theta_2] \quad \hat{\tau}_2, \hat{e}_2 \leftarrow \hat{\mathrm{synth}}\ \hat{\Gamma}[\theta_1]\ e_2 \\
\quad [\theta_3] \quad \hat{\tau}_3 \leftarrow \hat{\mathrm{pick}} \\
\quad [\theta_4] \quad _ \leftarrow \hat{\tau}_1[\theta_2\theta_3] \sim \hat{\tau}_2[\theta_3] \Rightarrow \hat{\tau}_3 \\
\qquad\quad \hat{\mathrm{pure}}\ (\hat{\tau}_3[\theta_4],\ \hat{e}_1[\theta_2\theta_3\theta_4]\ \hat{e}_2[\theta_3\theta_4])
\end{array}$$

Constraint generation (again)

$$\hat{\text{synth}} : \models \hat{\text{Ctx}} \twoheadrightarrow \text{Const Exp} \twoheadrightarrow M\ (\hat{\text{Ty}} \times \hat{\text{Exp}})$$

$$\begin{aligned}
&\hat{\text{synth}}\ \hat{\Gamma}\ (\lambda x.e) := \\
&\quad [\theta_1] \quad \hat{\tau}_1 \leftarrow \hat{\text{pick}} \\
&\quad [\theta_2] \quad \hat{\tau}_2, \hat{e} \leftarrow \hat{\text{synth}}\ (\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)\ e \\
&\qquad\quad \hat{\text{pure}}\ (\hat{\tau}_1[\theta_2] \Rightarrow \hat{\tau}_2,\ \hat{\lambda} x{:}\hat{\tau}_1[\theta_2].\hat{e})
\end{aligned}$$

$$\begin{aligned}
&\hat{\text{synth}}\ \hat{\Gamma}\ (e_1\ e_2) := \\
&\quad [\theta_1] \quad \hat{\tau}_1, \hat{e}_1 \leftarrow \hat{\text{synth}}\ \hat{\Gamma}\ e_1 \\
&\quad [\theta_2] \quad \hat{\tau}_2, \hat{e}_2 \leftarrow \hat{\text{synth}}\ \hat{\Gamma}[\theta_1]\ e_2 \\
&\quad [\theta_3] \quad \hat{\tau}_3 \leftarrow \hat{\text{pick}} \\
&\quad [\theta_4] \quad _ \leftarrow \hat{\tau}_1[\theta_2\theta_3] \sim \hat{\tau}_2[\theta_3] \Rightarrow \hat{\tau}_3 \\
&\qquad\quad \hat{\text{pure}}\ (\hat{\tau}_3[\theta_4],\ \hat{e}_1[\theta_2\theta_3\theta_4]\ \hat{e}_2[\theta_3\theta_4])
\end{aligned}$$

θ1 is a weakening that adds 1 fresh evar to the base world (implicit)

Constraint solving

$$\mathrm{solve} : \models \hat{\mathrm{L}}\mathrm{ist}\ (\hat{\mathrm{T}}\mathrm{y} \times \hat{\mathrm{T}}\mathrm{y}) \twoheadrightarrow \mathrm{Solved}\ () \qquad \mathrm{Solved}\ A := \hat{\mathrm{O}}\mathrm{ption}\ \Diamond A$$

- First-order unification
- outer recursion on worlds (single variable elimination)
- inner structural recursion over type equalities

$$\begin{aligned}
&\mathrm{run} : \models \hat{\mathrm{F}}\mathrm{ree}\ A \twoheadrightarrow \mathrm{Solved}\ A := \lambda m. \\
&\quad [\theta_1]\ eqs, a \leftarrow \mathrm{prenex}\ m\ ; \\
&\quad [\theta_2]\ _ \leftarrow \mathrm{solve}\ eqs\ ; \\
&\quad \hat{\mathrm{p}}\mathrm{ure}\ a[\theta_2]
\end{aligned}$$

End-to-end implementation

$$\begin{aligned}&\text{reconstruct } (\Gamma : \text{Ctx}) \ (e : \text{Exp}) : \text{Option } (\exists w.\hat{\text{Ty}}\ w \times \hat{\text{Exp}}\ w) := \\ &\quad (\{w\}, \theta, \hat{\tau}, \hat{e}) \leftarrow \text{run } (\hat{\text{synth}}\ \lceil \Gamma \rceil\ e); \\ &\quad \text{Some } (w, \hat{\tau}, \hat{e})\end{aligned}$$

N.B. No arbitrary grounding of unconstrained evars!

End-to-end implementation

$$
\begin{aligned}
&\mathrm{reconstruct}\ (\Gamma : \mathrm{Ctx})\ (e : \mathrm{Exp}) : \mathrm{Option}\ (\exists w.\hat{\mathrm{T}}\mathrm{y}\ w \times \hat{\mathrm{E}}\mathrm{xp}\ w) := \\
&\quad (\{w\}, \theta, \hat{\tau}, \hat{e}) \leftarrow \mathrm{run}\ (\hat{\mathrm{s}}\mathrm{ynth}\ \lceil \Gamma \rceil\ e); \\
&\quad \mathrm{Some}\ (w, \hat{\tau}, \hat{e})
\end{aligned}
$$

N.B. No arbitrary grounding of unconstrained evars!

Doing so would make our solution incomplete

End-to-end implementation

$$\begin{aligned}&\text{reconstruct } (\Gamma : \text{Ctx}) \ (e : \text{Exp}) : \text{Option } (\exists w.\hat{\text{Ty}}\ w \times \hat{\text{Exp}}\ w) := \\ &\quad (\{w\}, \theta, \hat{\tau}, \hat{e}) \leftarrow \text{run } (\hat{\text{synth}}\ \lceil \Gamma \rceil\ e); \\ &\quad \text{Some } (w, \hat{\tau}, \hat{e})\end{aligned}$$

N.B. No arbitrary grounding of unconstrained evars!

Doing so would make our solution **incomplete**

But now we're still stuck with these Ĥats everywhere

What about that correctness property from earlier?

$$\Gamma \vdash_A e : \tau \rightsquigarrow e' \leftrightarrow \Gamma \vdash_D e : \tau \rightsquigarrow e'$$

Can we keep this proof modular?
i.e. Simply “click” proofs of the different phases together?

Closed algorithmic typing

reconstruct produces a **world-indexed** Ty and Exp…
(some evars are unconstrained)

How can we relate its output to the declarative typing (on closed types, exp)?

$$
\begin{aligned}
&\Gamma \vdash_A e : \tau \rightsquigarrow e' := \\
&\quad \textbf{match } \mathrm{reconstruct}\ \Gamma\ e \textbf{ with} \\
&\quad \mid \mathrm{Some}\ (w, \hat{\tau}, \hat{e}) \rightarrow \\
&\qquad \exists (\iota : \mathrm{Assignment}\ w). \\
&\qquad\quad \hat{\tau}[\iota] = \tau \wedge \hat{e}[\iota] = e' \\
&\quad \mid \mathrm{None} \qquad\qquad\quad \rightarrow \bot
\end{aligned}
$$

"Give me any evar, and I will give you a monotype"

Let's use ghost state!

Track assignments to evars

$$\text{Pred}\ w := \underbrace{\text{Assignment}\ w} \to \mathbb{P}$$

"Give me any evar, and I will give you a monotype"

Track **additional knowledge** during the proofs

Intuition: lets us "pretend" that we already have a solution for the evars

Lifting connectives to Pred

$$P \dashv\vdash Q := \forall \iota : \text{Assignment } w,\ P\ \iota \leftrightarrow Q\ \iota$$
$$P \vdash Q := \forall \iota : \text{Assignment } w,\ P\ \iota \rightarrow Q\ \iota$$

… and the others (->, /\, …), omitted here

Internal equality

Two things u, v are Pred equal

$$u \approx v := \lambda \iota . u[\iota] = v[\iota]$$

If they are Prop equal after instantiation

Updates to the ghost state

$$\mathrm{wp}\ \{w_0\ w_1\}\ (\theta : w_0 \sqsubseteq w_1)\ (Q : \mathrm{Pred}\ w_1) : \mathrm{Pred}\ w_0$$

Updates to the ghost state

$$\begin{aligned}&\mathrm{wp}\ \{w_0\ w_1\}\ (\theta : w_0 \sqsubseteq w_1)\ (Q : \mathrm{Pred}\ w_1) : \mathrm{Pred}\ w_0 :=\\ &\ \lambda(\iota_0 : \mathrm{Assignment}\ w_0).\ \exists(\iota_1 : \mathrm{Assignment}\ w_1).\ \iota_1 \circ \theta = \iota_0 \wedge Q\ \iota_1\end{aligned}$$

Intuitively:

* **increase knowledge about the ghost state**
* **keep ghost state implicit**

Updates to the ghost state

Dual for universals

$$\begin{aligned}&\mathrm{wp}\ \{w_0\ w_1\}\ (\theta : w_0 \sqsubseteq w_1)\ (Q : \mathrm{Pred}\ w_1) : \mathrm{Pred}\ w_0 :=\\ &\ \lambda(\iota_0 : \mathrm{Assignment}\ w_0).\ \exists(\iota_1 : \mathrm{Assignment}\ w_1).\ \iota_1 \circ \theta = \iota_0 \wedge Q\ \iota_1\end{aligned}$$

$$\begin{aligned}&\mathrm{wlp}\ \{w_0\ w_1\}\ (\theta : w_0 \sqsubseteq w_1)\ (Q : \mathrm{Pred}\ w_1) : \mathrm{Pred}\ w_0 :=\\ &\ \lambda(\iota_0 : \mathrm{Assignment}\ w_0).\ \forall(\iota_1 : \mathrm{Assignment}\ w_1).\ \iota_1 \circ \theta = \iota_0 \to Q\ \iota_1\end{aligned}$$

Redefining the WP for Free

Recall from earlier:

$$\mathrm{WP} : \ \mathrm{Free}\ A \to (A \to \mathbb{P}) \to \mathbb{P}$$

Now indexed:

$$\hat{\mathrm{WP}}_{\mathrm{F}} : \models \hat{\mathrm{Free}}\ A \twoheadrightarrow \Box(A \twoheadrightarrow \mathrm{Pred}) \twoheadrightarrow \mathrm{Pred}$$

Redefining the WP for Free

$$\hat{\mathrm{WP}}_{\mathrm{F}} : \models \hat{\mathrm{Free}}\ A \twoheadrightarrow \square(A \twoheadrightarrow \mathrm{Pred}) \twoheadrightarrow \mathrm{Pred}$$

$$\begin{aligned}
\hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Pure}}\ a)\ Q &:= Q\ \mathrm{refl}\ a \\
\hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Fail}})\ Q &:= \bot \\
\hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Eq}}\ \hat{\tau}_1\ \hat{\tau}_2\ m)\ Q &:= \hat{\tau}_1 \approx \hat{\tau}_2 \wedge \mathrm{WP}_{\mathrm{F}}\ m\ Q \\
\hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Pick}}\ \alpha\ m)\ Q &:= \mathrm{wp}\ \mathrm{new}_\alpha\ (\hat{\mathrm{WP}}_{\mathrm{F}}\ m\ Q[\mathrm{new}_\alpha])
\end{aligned}$$

forms a modality for substitutions

Open algorithmic typing

$$\hat{\Gamma} \vdash_{\hat{A}} e : \hat{\tau} \rightsquigarrow \hat{e} :=$$

$$\hat{\mathrm{WP}}\ (\hat{\mathrm{synth}}\ \hat{\Gamma}\ e)\ (\lambda\theta\ (\hat{\tau}', \hat{e}').\ \hat{\tau}[\theta] \approx \hat{\tau}' \wedge \hat{e}[\theta] \approx \hat{e}').$$

Correctness statement, generalized

$$\hat{\Gamma} \vdash_{\hat{A}} e : \hat{\tau} \rightsquigarrow \hat{e} \dashv\vdash \hat{\Gamma} \vdash_{\hat{D}} e : \hat{\tau} \rightsquigarrow \hat{e}$$

Split into soundness and completeness, just as before.

Automation demo

Using Iris Proof Mode for a custom embedded logic

```
Lemma osoundness_aux e :
  ∀ w (G : OEnv w),
    ⊢ WLP (osynth e G) (fun _ θ '(t,ee) => G[θ] |--ₚ e; t ~> ee).
Proof.
  induction e; cbn; intros w G; iStartProof; wlpauto.
  destruct lookup eqn:HGx; wlpauto. iStopProof; pred_unfold.
  constructor. now rewrite lookup_inst HGx.
Qed.
```

Evaluation & Demo

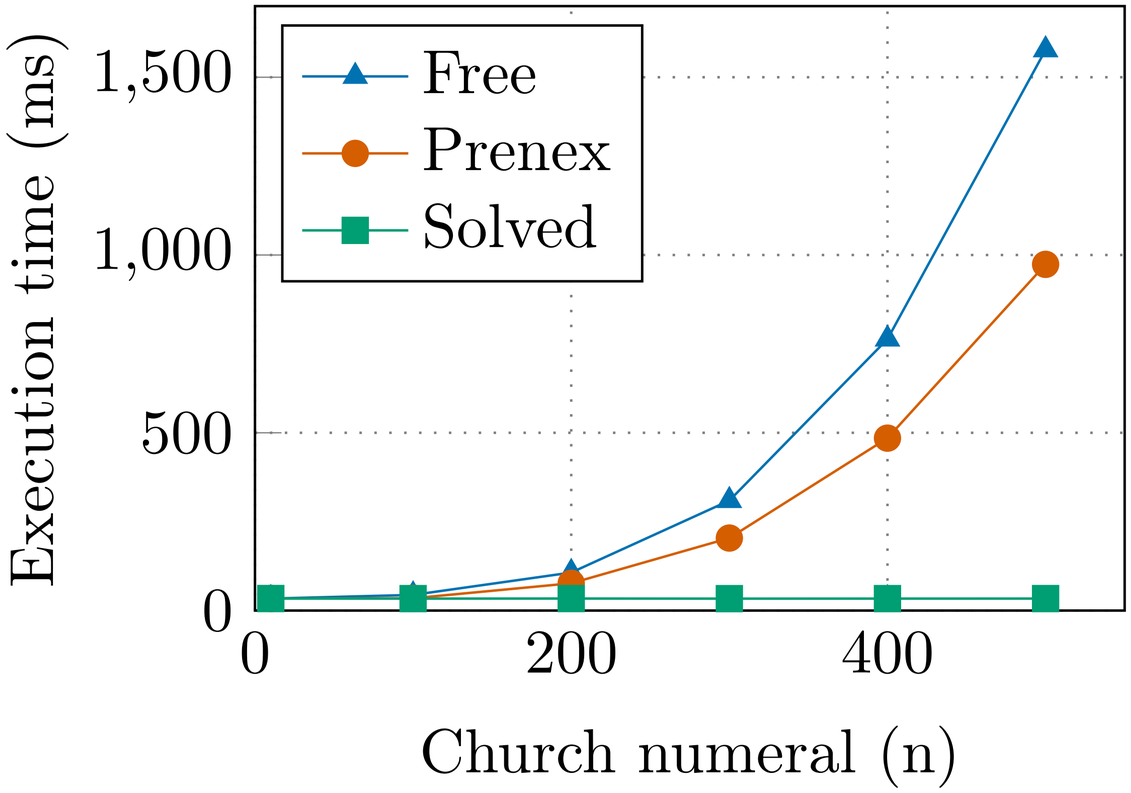
Proof effort

CATEGORY		Spec	Proof
I	**Generic**	**1583**	**1035**
	Base logic	533	325
	Infrastructure	282	154
	Unification	186	129
	Monad interface	126	146
	Logical relation	139	89
	Free monad	67	57
	Monad interface (HOAS)	99	16
	Open modality	44	54
	Free monad (HOAS)	51	4
	Prenex monad	22	31
	Solved monad	14	21
	Prenex conversion	20	9

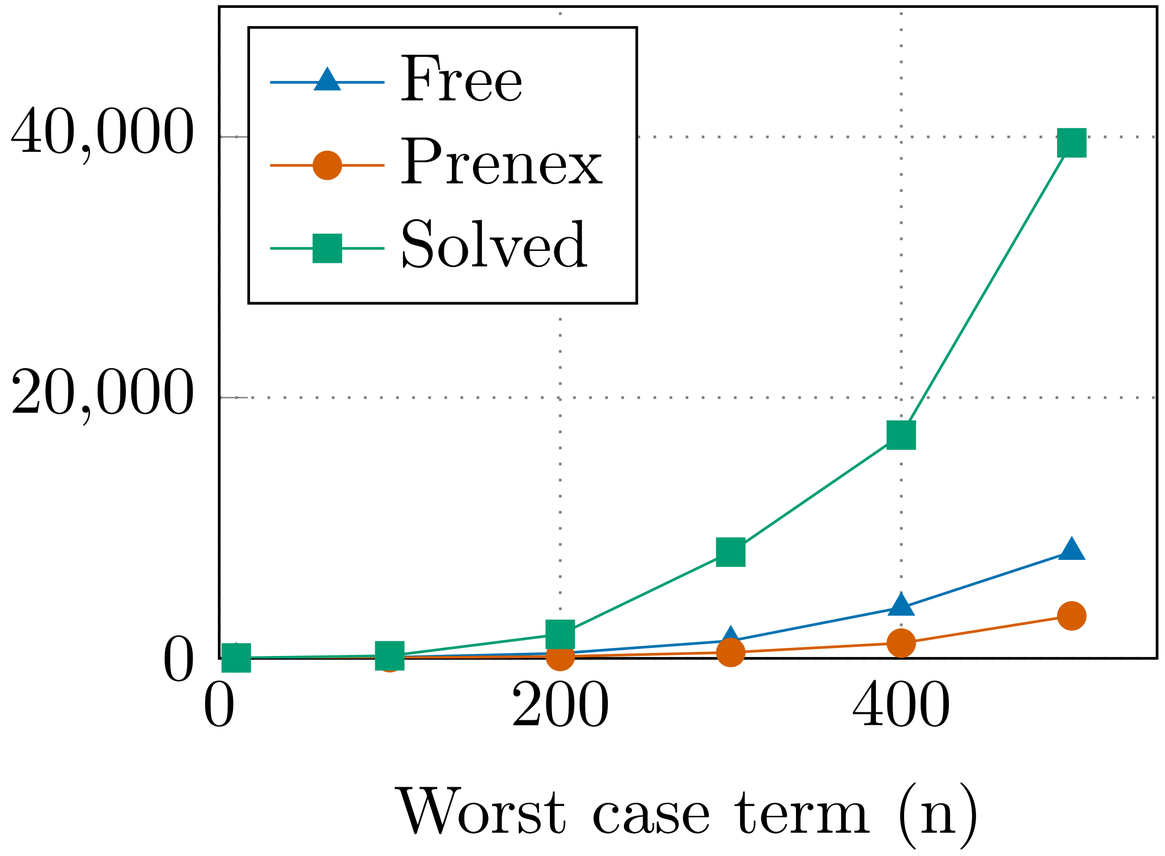
CATEGORY		Spec	Proof
II	**Specific ($\lambda_{\mathbb{B}}$)**	**777**	**557**
	Generators (HOAS)	213	160
	Generator (bidir)	110	77
	Relatedness	63	87
	Generator (check)	76	50
	Infrastructure	58	66
	Generator (synth)	68	49
	Composition	58	37
	Specification	70	4
	Unification	46	27
	Extraction	15	0
TOTAL		**2360**	**1592**

Code extraction

To Haskell, with a toy parser and prettyprinter



λf.λx.f(f(f x))), for n=3



λf.λx1.λy2.λ3.(f x1 x2 x3), for n=3

Wrapping up

Things I did not talk about

But that you can read about in the paper and the Coq code

- Alternative correctness proof via LR
- Embedded logic in Coq
- Strategies for proof automation (using Iris Proof Mode)

Things I did not talk about

And that are also not part of *this* paper

- Constraint solving (except briefly)
- Generalizations to various interesting type systems
  - HM
  - Qualified types
  - Odersky-Läufer
  - …

Conclusion

Three stories to tell…

… One about modularity of phases

… One about monadic constraints with semantic values

… One about custom base and program logics

Links

OOPSLA24 paper

https://doi.org/10.1145/3689786

Coq formalization

https://github.com/decrn/tilogics