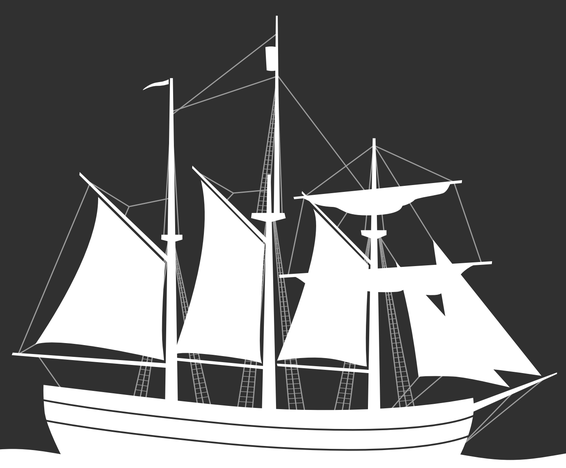
Patrycja Balik, Szymon Jędras, and Piotr Polesiuk

Fram



fram-lang.org

— a Tale of —

Theory-Driven Development,

Effect Inference

In the presence of higher-rank polymorphism!

```
let foo (f : Int ->[IO] Int) : [DB] Int = ...
let bar (g : {E : effect} -> _) = g (foo g)
```

Question: What is the type of bar?

- ({E} -> Int ->[E] Int) ->[DB] Int,
- ({E} -> Int ->[IO] Int) ->[DB,IO] Int,
- something else?

Answer: ({E} -> Int ->[E?p,a] Int) ->[DB,a] Int, assuming $a <: \mathtt{IO}$

$\varepsilon ::= \alpha \mid \mathsf{pure} \mid \varepsilon \cdot \varepsilon \mid \varepsilon ?\varphi$

Propositional formula representing delayed decision

```
tr_type(∀α^E.T) =
  let (P; Δ; τ) = tr_type(T) in
  fresh P_s = {p_β | β ∈ Δ} in
  fresh Δ' = {γ_β | β ∈ Δ} in
  (P ⊎ P_s; Δ'; ∀α^E.{β̄ ∈ Δ ↦ γ_β · α?p_β}*τ)

subtype(∀α^E.τ; ∀α^E.τ') =
  let (Ω, φ) = subtype(τ, τ') in
  ({α ↦ pure}*Ω; φ ∧ Ω[α])
```

$(\alpha)[\alpha] \triangleq \top$

$(\mathsf{pure})[\alpha] \triangleq \bot$

$(\varepsilon ?\varphi)[\alpha] \triangleq \varepsilon[\alpha] \wedge \varphi$

$(\beta)[\alpha] \triangleq \bot \quad (\alpha \neq \beta)$

$(\varepsilon_1 \cdot \varepsilon_2)[\alpha] \triangleq \varepsilon_1[\alpha] \vee \varepsilon_2[\alpha]$

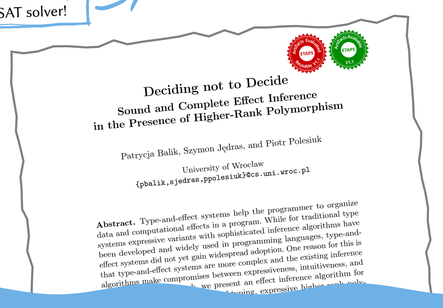
$\Omega[\alpha] \triangleq \bigwedge_{(\varepsilon_1 <: \varepsilon_2) \in \Omega} (\varepsilon_1[\alpha] \Rightarrow \varepsilon_2[\alpha])$

Solved later by a SAT solver!

Collected constraints are stored in algebraic type schemes (extension of ML polymorphism).

$\forall \overline{\alpha}.\, [\Omega]\ \tau$

Inference is still decidable for higher-rank polymorphism without constraints in schemes!



Deciding not to Decide
Sound and Complete Effect Inference in the Presence of Higher-Rank Polymorphism

Patrycja Balik, Szymon Jędras, and Piotr Polesiuk
University of Wrocław
{pbalik,sjedras,ppolesiuk}@cs.uni.wroc.pl

Abstract. Type-and-effect systems help the programmer to organize data and computational effects in a program. While for traditional type systems expressive variants with sophisticated inference algorithms have been developed and widely used in programming languages, type-and-effect systems did not yet gain widespread adoption. One reason for this is that type-and-effect systems are more complex and the existing inference algorithms make compromises between expressiveness, intuitiveness, and

Relational Models

Actual Core language from the implementation verified in Rocq!

$\mathcal{L}\varepsilon \mathbin{@} \tau_d/\varepsilon_d$ Interpretation of Computational Effects

$$\langle E[\mathcal{S}^\ell k.\, e] \rangle^\ell \longrightarrow \{k \mapsto \lambda x.\, \langle E[x] \rangle^\ell\}^* e$$

$$\frac{\Gamma \vdash v : \mathcal{L}\varepsilon \mathbin{@} \tau_d/\varepsilon_d \qquad \Gamma, k : \tau \to_{\varepsilon_d} \tau_d \vdash e : \tau_d/\varepsilon_d}{\Gamma \vdash \mathcal{S}^v k.\, e : \tau/\varepsilon}$$

$$\frac{\Gamma \vdash v : \mathcal{L}\varepsilon \mathbin{@} \tau_d/\varepsilon_d \qquad \Gamma \vdash e : \tau_d/\varepsilon \cdot \varepsilon_d}{\Gamma \vdash \langle e \rangle^v : \tau_d/\varepsilon_d} \qquad \frac{\alpha - \text{fresh} \qquad \Gamma, x : \mathcal{L}\alpha \mathbin{@} \tau_d/\varepsilon_d \vdash e : \tau/\varepsilon}{\Gamma \vdash \mathsf{label}\ x\ \mathsf{in}\ e : \tau/\varepsilon}$$

$$\ell \in [\![\mathcal{L}\varepsilon_0 \mathbin{@} \tau/\varepsilon]\!]_\eta \iff$$

$$[\![\varepsilon_0]\!]_\eta = \{\langle \mathcal{S}^\ell k.\, e, \ell, R \rangle \mid \forall v \in (R \to_{[\![\varepsilon]\!]_\eta} [\![\tau]\!]_\eta).\, \{k \mapsto v\}^* e \in \mathcal{E}[\![\tau]\!]_\eta [\![\varepsilon]\!]_\eta\}$$

$$\mathbf{Effect} \triangleq \mathbf{UPred}(\mathrm{Expr}_\varnothing \times \mathrm{Lbl} \times \mathbf{UPred}(\mathrm{Val}_\varnothing))$$

$$\mathcal{E}RF \triangleq R \cup \{\text{makes step}\} \cup \mathcal{S}RF$$

$$\mathcal{S}RF \triangleq \{E[e] \mid \exists \ell, R_o.\, \langle e, \ell, R_o \rangle \in F \wedge \ell\text{-free } E \wedge E \in R_o \to_F R\}$$

Borrowed ~~Stolen~~ from *Handle with Care*

$\mathcal{D}_r \tau_1 \simeq \tau_2$ Recursive Datatypes and Irrelevant Values

$$\frac{\Gamma \vdash_{\mathsf{irr}} v : \mathcal{D}_r \tau_1 \simeq \tau_2 \qquad \Gamma \vdash e : \tau_2/\varepsilon}{\Gamma \vdash \mathsf{roll}_v\ e : \tau_1/\varepsilon} \qquad \frac{\Gamma \vdash_{\mathsf{irr}} v : \mathcal{D}_r \tau_1 \simeq \tau_2 \qquad \Gamma \vdash e : \tau_1/\varepsilon}{\Gamma \vdash \mathsf{unroll}_v\ e : \tau_2/\mathrm{NTerm}?r \cdot \varepsilon}$$

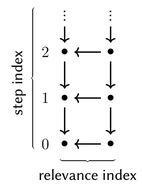
$$\frac{\alpha - \text{fresh} \qquad \alpha \text{ occurs positively in } \tau_2 \text{ when } r = + \qquad \Gamma, x :_{\mathsf{irr}} \mathcal{D}_r \alpha \simeq \tau_2 \vdash e : \tau/\varepsilon}{\Gamma \vdash \mathsf{label}\ x\ \mathsf{in}\ e : \tau/\varepsilon}$$

$$v \in [\![\mathcal{D}_r \tau_1 \simeq \tau_2]\!]_\eta \iff$$

$$\triangleright_{\mathsf{rel}} \bot \wedge$$

$$\square_{\mathsf{rel}}([\![\tau_2]\!]_\eta \subseteq [\![\tau_1]\!]_\eta) \wedge$$

$$\square_{\mathsf{rel}}([\![\tau_1]\!]_\eta \subseteq \triangleright_{?r} [\![\tau_2]\!]_\eta)$$



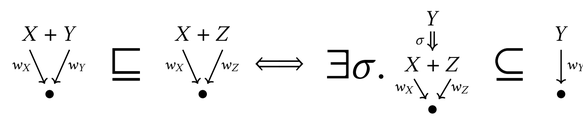
$\Gamma, x : \circ\tau$ Productivity of General Recursion

$$\frac{\Gamma, x : \circ\tau \vdash e_1 : \tau/\mathsf{pure} \qquad \Gamma, x : \tau \vdash e_2 : \tau'/\varepsilon}{\Gamma \vdash \mathsf{letrec}\ x = e_1\ \mathsf{in}\ e_2 : \tau'/\varepsilon}$$

$$\frac{\Gamma, \Gamma' \vdash e : \tau/\varepsilon}{\Gamma, \circ\Gamma' \vdash \circ e : \tau/\mathrm{NTerm} \cdot \varepsilon}$$

$$\frac{e_1 \longrightarrow e_1'}{\mathtt{letrec}\ x = e_1\ \mathtt{in}\ e_2 \longrightarrow \mathtt{letrec}\ x = e_1'\ \mathtt{in}\ e_2}$$

$$\mathrm{World} \triangleq \Sigma X.\, X \to \mathbf{UPred}(\mathrm{Val}_\varnothing \cup \{\mu x.\, v\})$$



Coherence

Fram has implicit parameters...

```
let foo bar =
  let ~a = 13 in bar ()
let ~a = 42 in
foo (fn () => ~a)
```

What should the above code evaluate to?

- 42?
- 13?

...and parameter sections.

```
parameter ~n
parameter ~m

let baz x = ...
```

Which parameters should `baz` generalize?

- ~n?
- ~m?
- Both?

Source of ambiguity

Semantics is given by a translation of the typing derivation.

$$\frac{\Gamma; \overline{\sim n}, \sim m \vdash \overline{d} \rightsquigarrow \overline{d} \Rightarrow \Gamma'}{\Gamma; \overline{\sim n} \vdash \mathtt{parameter}\ \sim m\ \ \overline{d} \rightsquigarrow \overline{d} \Rightarrow \Gamma'} \qquad \frac{\overline{\{\sim m\} \subseteq \{\sim n\}} \quad \Gamma \setminus \{\overline{\sim n}\}, \overline{\sim m : \tau} \vdash e : \sigma \rightsquigarrow e \quad \Gamma, x : \{\overline{\sim m : \tau}\} \to \sigma; \overline{\sim n} \vdash \overline{d} \rightsquigarrow \overline{d} \Rightarrow \Gamma'}{\Gamma; \overline{\sim n} \vdash \mathtt{fun}\ x = e\ \ \overline{d} \rightsquigarrow \mathtt{fun}\ x\ \overline{\sim m} = e\ \ \overline{d} \Rightarrow \Gamma'}$$

$$\frac{\Gamma(x) = \{\overline{\sim m : \tau'}\} \to \sigma \qquad \overline{\Gamma \vdash e : \tau \rightsquigarrow e} \qquad \overline{\Gamma, \overline{\sim n : \tau} \vdash \sim m : \tau' \rightsquigarrow \sim m}}{\Gamma \vdash x\{\overline{\sim n = e}\} : \sigma \rightsquigarrow \mathtt{let}\ \overline{\sim n = e}\ \mathtt{in}\ x\ \overline{\sim m}}$$

Problem:

$$\left.\begin{array}{l} \Gamma \vdash e : \tau \rightsquigarrow e_1 \\ \Gamma \vdash e : \tau \rightsquigarrow e_2 \end{array}\right\} \overset{?}{\Longrightarrow} e_1 \equiv e_2$$

Solution: Equivalent untyped semantics which does not depend on a particular type derivation.

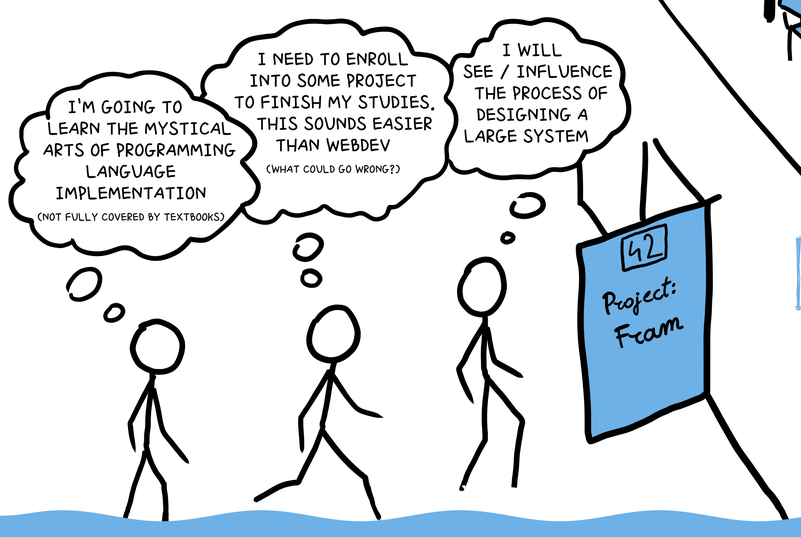
$$\frac{\rho; \overline{\sim n}, \sim m \vdash \overline{d} \Downarrow \rho'}{\rho; \overline{\sim n} \vdash \mathtt{parameter}\ \sim m\ \ \overline{d} \Downarrow \rho'} \qquad \frac{\rho[x \mapsto \langle \rho \setminus \{\overline{\sim n}\}, \lambda\{\overline{\sim n}\}.\, e \rangle]; \overline{\sim n} \vdash \overline{d} \Downarrow \rho'}{\rho; \overline{\sim n} \vdash \mathtt{fun}\ x = e\ \ \overline{d} \Downarrow \rho'}$$

$$\frac{\rho(x) = \langle \rho', \lambda\{\overline{\sim m}\}.\, e' \rangle \qquad \overline{\rho \vdash e \Downarrow v} \qquad \rho' \uplus (\rho[\overline{\sim n \mapsto v}])|_{\overline{\sim m}} \vdash e' \Downarrow v'}{\rho \vdash x\{\overline{\sim n = e}\} \Downarrow v'}$$

Student Involvement, and

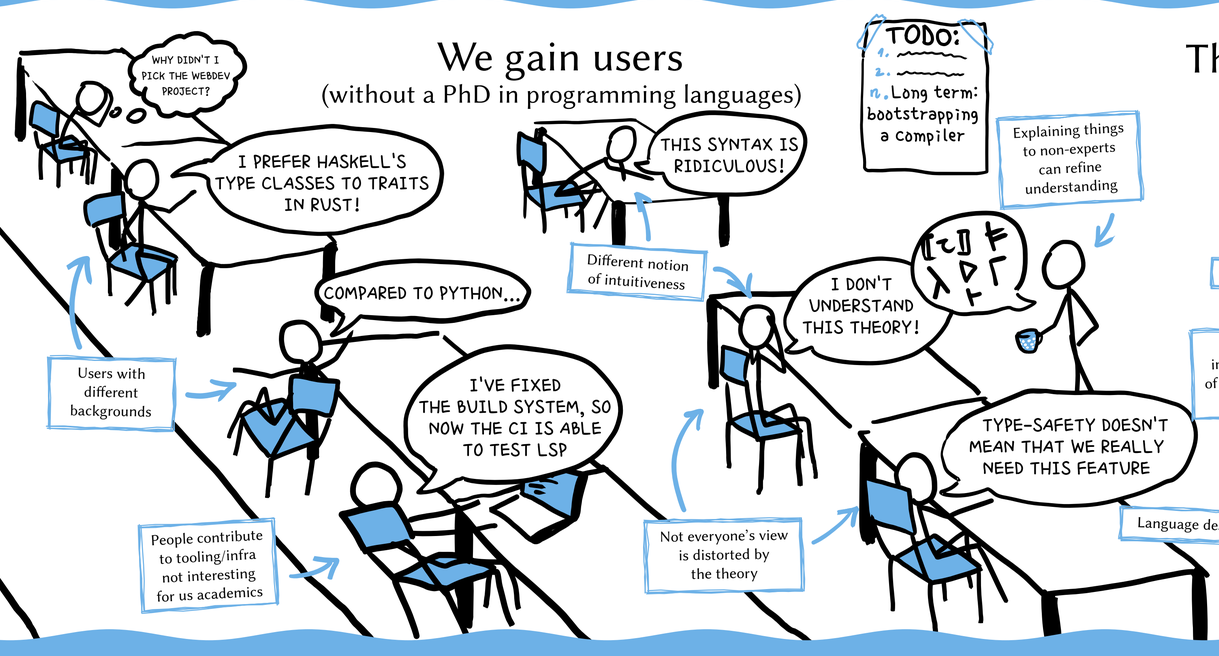
Students come with expectations

(mostly aligned with reality)



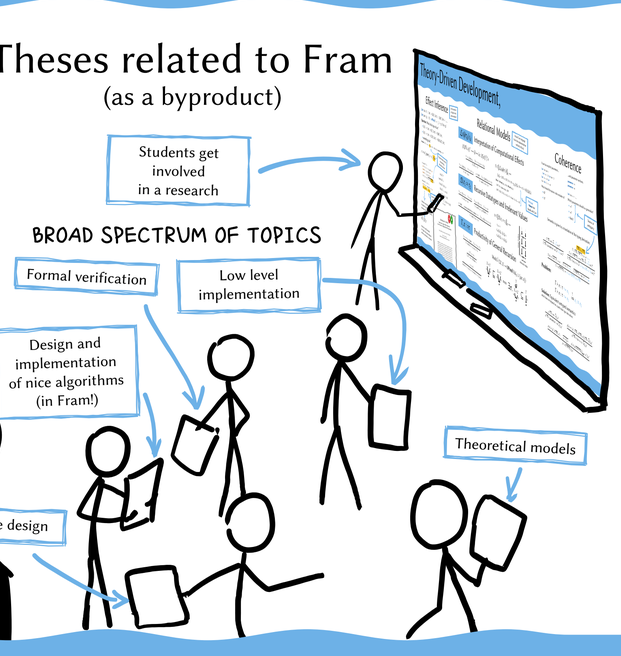
We gain users

(without a PhD in programming languages)



Theses related to Fram

(as a byproduct)



Minimal, but Expressive Design

Named Parameters

Fram supports Rank-N types.

```
let foo (id : {X} -> X -> X) =
  (id True, id {X = Int})
```

What else can we put in the curly {braces}?

Named type parameters

Implicit parameters

Optional and regular named parameters

```
parameter ~say

let greeting {~say} name =
  ~say ("Hello " + name + "!")

let farewell name =
  ~say ("Bye " + name + "!")

let conversation {name, ?msg} () =
  greeting name; msg.iter ~say; farewell name

let _ =
  let ~say = printStrLn
  let msg = "Isn't the weather nice?"
  in
  conversation { msg, name = "Fram" } ()
```

Sections

Declared parameters are abstracted implicitly when used

Method constraints

Packed/unpacked modules

Nice Features for Free!

Records

```
data Vec = { x : Int, y : Int }
```

is a syntactic sugar for

```
data Vec = Vec of { x : Int, y : Int }
method x (Vec { x }) = x
method y (Vec { y }) = y
```

so we can write

```
method add (v1 : Vec) (v2 : Vec) =
  Vec { x = v1.x + v2.x, y = v1.y + v2.y }
```

Functors

```
data MapS Key = Map of
  { T : type -> type
  , empty : {Val} -> T Val
  , method add : {Val} -> T Val -> Key -> Val ->[] T Val
  ...
  }

make : { Key, method lt : Key -> Key ->[] Bool } -> MapS Key

let Map { module IntMap } = make { Key = Int }
```

Existential Types

```
data Stream X =
  | Stream of {St}, St, (St ->[] Pair X St)

let mapi f (Stream {St=OldSt} st next) =
  Stream {St=Pair Int OldSt}
    (0, st)
    (fn (i, st) =>
      let (x, st') = next st in
      (f i x, (i+1, st')))
```

Effect Handlers

```
data BT E =
  { flip : Unit ->[E] Bool
  , fail : {X} -> Unit ->[E] Bool
  }

method rec select (bt : BT _) a b =
  if a >= b then bt.fail ()
  else if bt.flip () then a
  else bt.select (a + 1) b
```

```
let triples (bt : BT _) n =
  let a = bt.select 1 n
  let b = bt.select a n
  let c = bt.select b n
  in
  if a * a + b * b == c * c then (a, b, c)
  else bt.fail ()

let printTriples n =
  handle bt = BT
    { effect flip () = resume True; resume False
    , effect fail () = ()
    }
  in
  printStrLn (triples bt n).show

let hAny =
  handler BT
    { effect flip () = resume True >.orElse (fn () => resume False)
    , effect fail () = None
    }
  return x => Some x
  end

let anyTriple n =
  handle bt with hAny in triples bt n
```

Using datatypes for effect capabilities

Lexically scoped effect handlers!

Effect system guarantees that effects doesn't escape their scope

Handler defines an effect capability and introduces a new effect

Don't tell anyone that this is the shift_0 delimited control operator

Almost invisible to the programmer thanks to the effect inference

First-class effect handlers for greater modularity

An Example: Prolog

```
parameter ~bt : BT _
parameter ~st : VarState _
parameter ~kb : KnowledgeBase _
parameter ~fr : Fresh _

let rec unify (t1 : Term) (t2 : Term) =
  match t1.view, t2.view with
  ...
  | Var x, t =>
    t.occursCheck x;
    ~st.set x t
  | Fun f ts1, Fun g ts2 =>
    if f == g then List.iter2 {~onError = ~bt.fail} unify ts1 ts2
    else ~bt.fail ()
  end

let rec eval (t : Term) =
  let Clause t' ts = ~bt.choose ~kb.ask >.refresh in
  unify t t';
  List.iter eval ts
```