Gabriel Scherer
Jan 2016 - Jul 2017: Northeastern University
Sep 2012 - Dec 2015: Gallium (INRIA Rocq.)

Formally improving the programming experience

So far:

1. Implementation and research on OCaml
2. Type-directed program inference
3. Program equivalence and canonical representations

Project:

1. Canonical representations at higher types
2. Tools with program equivalence
3. Multi-language programming systems

Integration: Parsifal

So far:

1. Implementation and research on OCaml
2. Type-directed program inference
3. Program equivalence and canonical representations

Project:

1. Canonical representations at higher types
2. Tools with program equivalence
3. Multi-language programming systems

Integration: Parsifal

Result: deciding equivalence

Setting

We have tools to check that a program verifies a specification.

Few tools to check program equivalence.
Potential applications: verification of refactoring, consistency checking, program synthesis...

Pure functional programming: rich equivalences.
More useful, but more complex.

Fundamental challenge: equivalence is not well-understood.

Equivalence in the full simply-typed λ-calculus is decidable

"Deciding equivalence with sums and the empty type"
Gabriel Scherer
POPL 2017
https://arxiv.org/abs/1610.01213

History

Simple types: formal model of **datatypes** in programming.

Decidability of equivalence:

- $\Lambda C(\alpha, \to)$: Tait, 1967 or earlier.
- $\Lambda C(\alpha, \to, \times)$: essentially the same proof.
- $\Lambda C(\alpha, \to, \times, 1)$: essentially the same proof.
- $\Lambda C(\alpha, \to, \times, 1, +)$: Ghani, 1995; Altenkirch, Dybjer, Hoffman, Scott: 2001; Balat, Di Cosmo, Fiore: 2004; Lindley, 2007; Ahmad, Licata, Harper, 2010.
- $\Lambda C(\alpha, \to, \times, 1, +, 0)$: this work.

Open problem despite work: need a different approach.

```
module type PARAM = sig
  type error
  val process : input -> (output + error)
  ...
end

module Action (P : PARAM) = struct
  let process_or_stdout input =
    match process input with
    | σ1 out -> out
    | σ2 err -> report_error_stdout (); exit 1
  let process_or_email input =
    match process input with
    | σ1 out -> out
    | σ2 err -> report_error_email (); exit 2
  ...
end
```

Intuition

0 represents impossible cases.

$$\frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash u_1, u_2 : A}{\Gamma \vdash u_1 \approx_\eta u_2 : A}$$

Intuition

0 represents impossible cases.

$$\frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash u_1, u_2 : A}{\Gamma \vdash u_1 \approx_\eta u_2 : A}$$

```
let process_or_stdout input =
  match process input with
  | σ1 out -> out
  | σ2 err -> report_error_stdout (); exit 1

let process_or_email input =
  match process input with
  | σ1 out -> out
  | σ2 err -> report_error_email (); exit 2
```

Question

What is a **canonical form** for equivalence of simply-typed terms?

Redundancy: two (syntactically) distinct terms that are equivalent.

Canonical representation: a syntax of programs with no redundancy:

$$(\not\approx_{\texttt{stx}}) \implies (\not\approx_{\texttt{sem}})$$

Question

What is a **canonical form** for equivalence of simply-typed terms?

Redundancy: two (syntactically) distinct terms that are equivalent.

Canonical representation: a syntax of programs with no redundancy:

$$(\not\approx_{\mathtt{stx}}) \implies (\not\approx_{\mathtt{sem}})$$

With only functions and pairs, there is a reasonable notion of β-short η-long normal form.

Question

What is a **canonical form** for equivalence of simply-typed terms?

Redundancy: two (syntactically) distinct terms that are equivalent.

Canonical representation: a syntax of programs with no redundancy:

$$(\not\approx_{\mathtt{stx}}) \implies (\not\approx_{\mathtt{sem}})$$

With only functions and pairs, there is a reasonable notion of β-short η-long normal form. It does not scale to sums.

Idea

Curry-Howard, again: programs as proofs.

The structure of

canonical forms

corresponds to the structure of

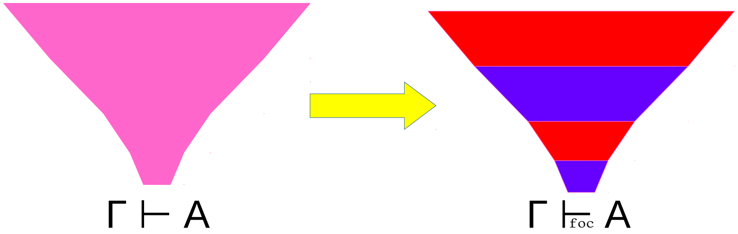
proof **search**

Restricting the search space restricts expression redundancy.

Research transfer from proof theory.

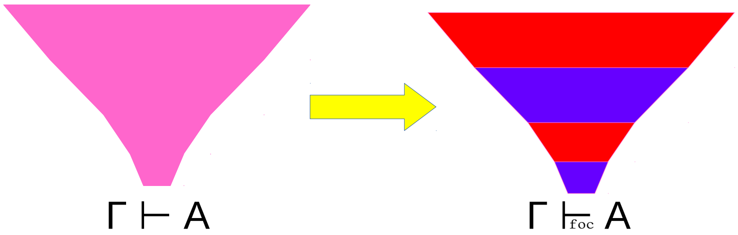
Proof search: Focusing

(existing work)



Proof search: Focusing

(existing work)

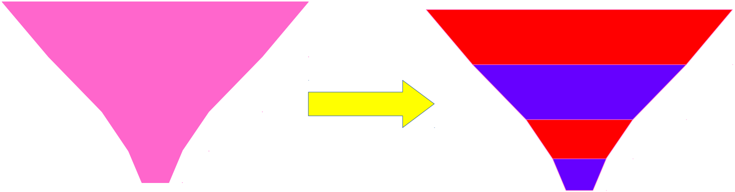


Gives a term representation ($\vdash_{\texttt{foc}}$).
Not yet canonical.

And it preserves computational content!

Proof search: Focusing

(existing work)



$\Gamma \vdash A$ $\qquad\qquad$ $\Gamma \vdash_{\mathtt{foc}} A$

Gives a term representation ($\vdash_{\mathtt{foc}}$).
Not yet canonical.

And it preserves computational content!

$$\Gamma \vdash t : A \qquad \overset{\text{(new)}}{\Longrightarrow} \qquad \exists v \approx_{\beta\eta} t, \quad \Gamma \vdash_{\mathtt{foc}} v : A$$

Proof search: Saturation

(my contribution).

Non-invertible steps: either ($p : P$) (value) or (`let` $x = n[y : N]$ `in ...`) (environment).

Idea: make all possible deductions from the environment first.

Canonical representation, (locally) complete.

Proof search: Saturation

(my contribution).

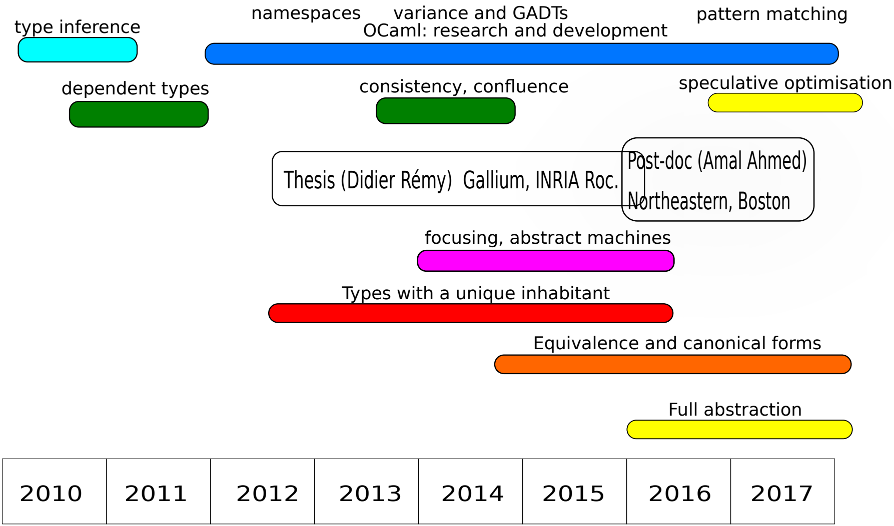
Non-invertible steps: either $(p : P)$ (value) or $(\texttt{let}\ x = n[y : N]\ \texttt{in}\ \ldots)$ (environment).

Idea: make all possible deductions from the environment first.

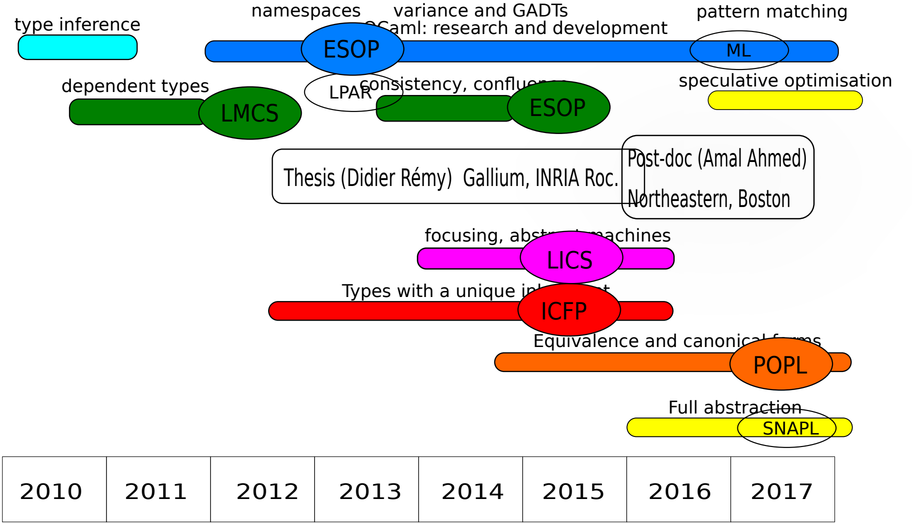
Canonical representation, (locally) complete.

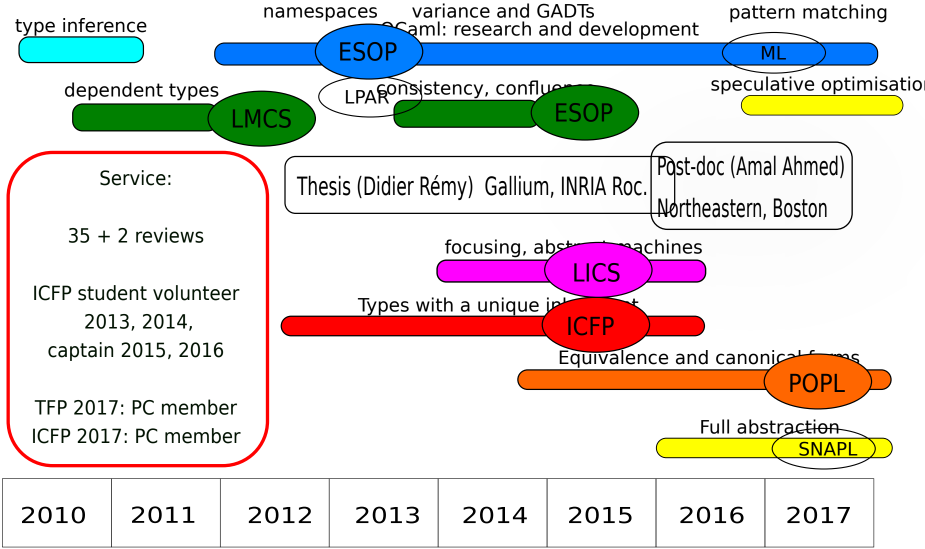
$$\frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash u_1, u_2 : A}{\Gamma \vdash u_1 \approx_\eta u_2 : A}$$

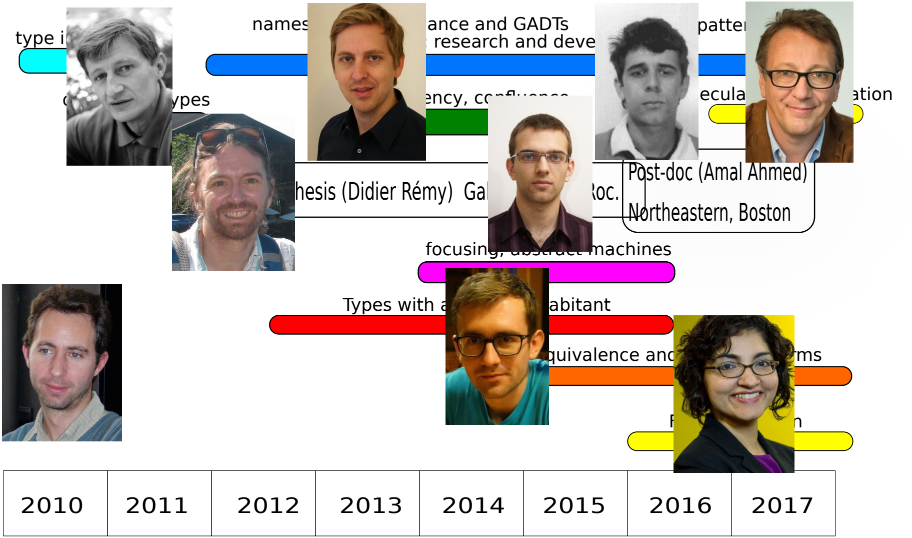
Saturation discovers t.



type inference
namespaces
variance and GADTs
OCaml: research and development
pattern matching
dependent types
consistency, confluence
speculative optimisation
Thesis (Didier Rémy) Gallium, INRIA Roc.
Post-doc (Amal Ahmed)
Northeastern, Boston
focusing, abstract machines
Types with a unique inhabitant
Equivalence and canonical forms
Full abstraction
2010
2011
2012
2013
2014
2015
2016
2017


type inference
namespaces
variance and GADTs
OCaml: research and development
pattern matching
ESOP
ML
dependent types
LMCS
LPAR
consistency, conflu
ESOP
speculative optimisation
Thesis (Didier Rémy) Gallium, INRIA Roc.
Post-doc (Amal Ahmed)
Northeastern, Boston
focusing, abs
machines
LICS
Types with a unique i
ICFP
Equivalence and canonical
POPL
Full abstraction
SNAPL
2010
2011
2012
2013
2014
2015
2016
2017


type inference
namespaces
variance and GADTs
aml: research and development
pattern matching
ESOP
ML
dependent types
LMCS
LPAR
onsistency, conflu
ESOP
speculative optimisatio
Service:
35 + 2 reviews
ICFP student volunteer
2013, 2014,
captain 2015, 2016
TFP 2017: PC member
ICFP 2017: PC member
Thesis (Didier Rémy) Gallium, INRIA Roc.
Post-doc (Amal Ahmed)
Northeastern, Boston
focusing, abs
achines
LICS
Types with a unique i
ICFP
Equivalence and canonica
POPL
Full abstraction
SNAPL
2010
2011
2012
2013
2014
2015
2016
2017


type i
names
ance and GADTs
research and deve
patter
ypes
ency, co
ecula
ation
hesis (Didier Rémy) Ga
Roc.
Post-doc (Amal Ahmed)
Northeastern, Boston
focusing,
machines
Types with a
abitant
quivalence and
rms
F
n
2010
2011
2012
2013
2014
2015
2016
2017

2012-2017: Research and development on OCaml

- technical contributions to the implementation (committer #2)
- community building: opening the development process
  (github, code reviews, social events)
  20 contributors in 2012, 93 in 2017
- research problems identified and studied

Example: ambiguous pattern variables, with Luc Maranget

- bug report from the Why3 team
- research and publication – ML workshop post-proceedings
- patch to the compiler, merged in 4.04.0
- cross-language discussions with Haskell, Rust designers

Community recognition:
PC member for the OCaml Workshop 2016, PC chair for 2017.

Theory, design and implementation of programming languages.

So far:

1. Implementation and research on OCaml
2. Type-directed program inference
3. Program equivalence and canonical representations

Project:

1. Canonical representations at higher types
2. Tools with program equivalence
3. Multi-language programming systems

Theory, design and implementation of programming languages.

So far:

1. Implementation and research on OCaml
2. Type-directed program inference
3. Program equivalence and canonical representations

Project:

1. Canonical representations at higher types
2. Tools with program equivalence
3. Multi-language programming systems

Project: multi-language programming systems

The Ultimate Language may not exist

Ideal (general-purpose) language design:
simplicity/power compromise using powerful, orthogonal concepts.

More and more problem domains for general-purpose languages:
distributed programming, web/mobile development...

Languages of today tend to evolve into behemoths by piling features up:
C++, Scala, GHC Haskell, OCaml...

Does managing this complexity require super-human feats?

Multi-language systems

Proposal: Multi-language programming systems.
Several smaller languages working together to cover the feature space.
(simpler?)

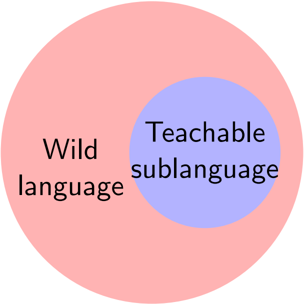
(Done in practice, but no design guarantees.)

To manage complexity, one should be able to **ignore** some languages of the system – and not pay for it.

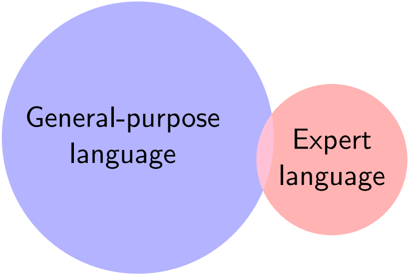
Multi-language system require specific **design** for graceful interoperation.

We must learn how to achieve this, rigorously.

Multi-language stories



Abstraction leak?



Graceful interoperation?

Full abstraction

(existing work)

$[\![\_]\!] : S \longrightarrow T$ fully abstract:

$$a \approx b \implies [\![a]\!] \approx [\![b]\!]$$

Full abstraction preserves (equational) reasoning.

(Program equivalence again)

Full abstraction

(existing work)

$[\![\_]\!] : S \longrightarrow T$ fully abstract:

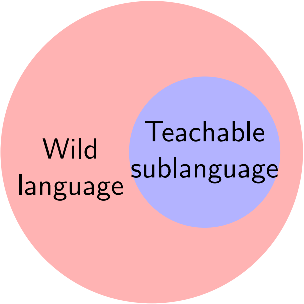
$$a \approx b \implies [\![a]\!] \approx [\![b]\!]$$

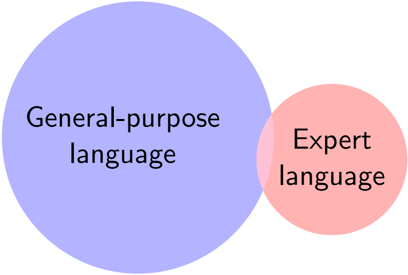
Full abstraction preserves (equational) reasoning.

(Program equivalence again)

(new) **Claim**: full abstraction can be used to **formally** capture the **usability** properties of multi-language design.

Full abstraction for multi-language systems

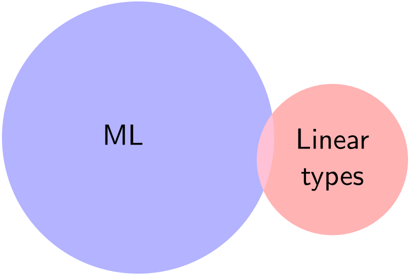




No abstraction leaks: $T \xrightarrow{f.a.} W$

Graceful interoperation: $G \xrightarrow{f.a.} (G + E)$
(not symmetric)

First instance in the works



Expert linear language allows safe lower-level programming.
Efficiency and safety complement.

Other potential cases: Coq+OCaml, Why3+ML, safe FFI,
interaction between proof assistants (Coq, Agda, Abella, Dedukti)...

Challenges

Full abstraction not yet well-understood. Theoretical advances required. (simply-typed with recursion $\rightarrow$ untyped: was POPL 2016 article)

How to weaken full-abstraction when it cannot hold?

Can this scale to full-fledged *n*-languages designs?

G.S. and Amal Ahmed. "Search for Program Structure". **SNAPL**. 2017.

G.S. "Deciding equivalence with sums and the empty type". **POPL**. 2017.

G.S. and Didier Rémy. "Which simple types have a unique inhabitant?" **ICFP**. 2015.

Guillaume Munch-Maccagnoni and G.S. "Polarised Intermediate Representation of Lambda Calculus with Sums". **LICS**. 2015.

G.S. "Multi-focusing on extensional rewriting with sums". **TLCA**. 2015.

G.S. and Didier Rémy. "Full reduction in the face of absurdity". **ESOP**. 2015.

Pierre-Évariste Dagand and G.S. "Normalization by realizability also evaluates". **JFLA**. 2015.

G.S. and Jan Hoffmann. "Tracking Data-Flow with Open Closure Types". **LPAR**. 2013.

G.S. and Didier Rémy. "GADTs meet subtyping". **ESOP**. 2013.

Andreas Abel and G.S. "On Irrelevance and Algorithmic Equality in Predicative Type Theory". **Logical Methods in Computer Science** (2012).

G.S. and Jérôme Vouillon. "Macaque: Interrogation sûre et flexible de bases de données depuis OCaml". **JFLA**. 2010.