Gabriel Scherer
Jan 2016 - Jul 2017: Northeastern University – with Amal Ahmed
Sep 2012 - Dec 2015: Gallium (INRIA Rocq.) – with Didier Rémy

Search for Program Structure

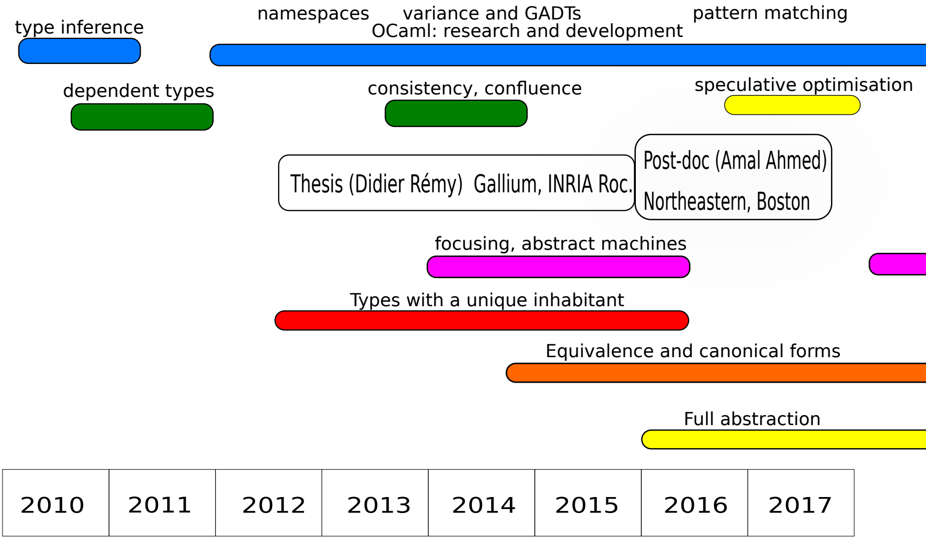
Theory, design and implementation of programming languages.

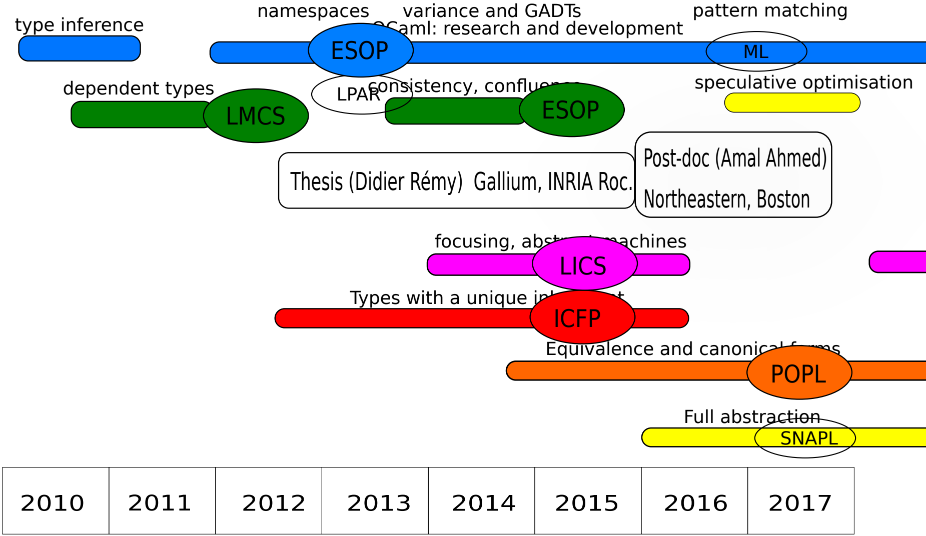
I am fond of programming.

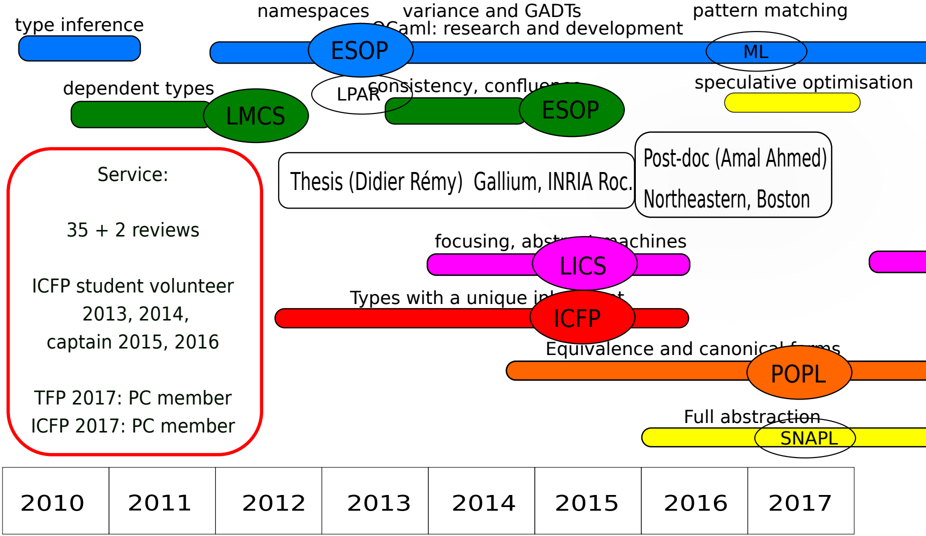
I want to make it even better.

Designing good programming languages and tools is difficult.
We rely a lot on subjective opinions, gut feelings.

I try to capture usability aspects through formalism.
And implement the resulting designs.


type inference
namespaces
variance and GADTs
pattern matching
OCaml: research and development
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

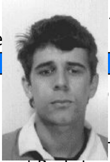

type inference
namespaces
variance and GADTs
OCaml: research and development
pattern matching
ESOP
ML
dependent types
LMCS
LPAR
consistency, conflu
ESOP
speculative optimisation
Thesis (Didier Rémy) Gallium, INRIA Roc.
Post-doc (Amal Ahmed)
Northeastern, Boston
focusing, abst machines
LICS
Types with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

type i

names

ance and GADTs
research and deve



ttern

ypes

ency, co

ecula

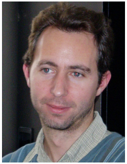
ation

hesis (Didier Rémy) Gal

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.

Project: Search for Program Structure

Program equivalence

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

Few tools to check program equivalence.
(richer programming languages $\Rightarrow$ more complex equivalences.)

Untapped potential for applications; tools for:

- verified refactoring
- consistency checking for implicit programming
- program synthesis (see further)

Challenge: undecidability.

Past result: Full simply-typed equivalence is decidable

"Deciding equivalence with sums and the empty type"
Gabriel Scherer (at Northeastern)
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; easy
- $\Lambda C(\alpha, \to, \times)$: 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. difficult
- $\Lambda C(\alpha, \to, \times, 1, +, 0)$:
  Open problem that needed a different approach. hard

History

Simple types: formal model of datatypes in programming.

Decidability of equivalence:

- $\Lambda C(\alpha, \to)$: Tait, 1967 or earlier; easy
- $\Lambda C(\alpha, \to, \times)$: 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. difficult
- $\Lambda C(\alpha, \to, \times, 1, +, 0)$:
  Open problem that needed a different approach. hard

  my work (POPL 2017)

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

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

Intuition

0 represents impossible cases.

```
module P = struct
  type error = 0
  let process : input -> (output + 0) = ...
end
let process_or_stdout input =
  match P.process input with
  | σ1 out → out
  | σ2 err → report_error_stdout (); exit 1
let process_or_email input =
  match P.process input with
  | σ1 out → out
  | σ2 err → report_error_email (); exit 2
```

Intuition

0 represents impossible cases.

```
module P = struct
  type error = 0
  let process : input -> (output + 0) = ...
end
let process_or_stdout input =
  match P.process input with
  | σ1 out → out
  | σ2 err → report_error_stdout (); exit 1
let process_or_email input =
  match P.process input with
  | σ1 out → out
  | σ2 err → report_error_email (); exit 2
```

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

Question

What is a canonical form for simply-typed terms?

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

Canonical representation: a syntax of programs with no redundancy:

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

(Decides equivalence.)

Question

What is a canonical form for simply-typed terms?

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

Canonical representation: a syntax of programs with no redundancy:

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

(Decides equivalence.)

With only functions and pairs, easy.

Question

What is a canonical form for simply-typed terms?

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

Canonical representation: a syntax of programs with no redundancy:

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

(Decides equivalence.)

With only functions and pairs, easy.
It does not scale to sums (even booleans !).

Idea

Curry-Howard, again: programs as proofs.

The structure of

canonical forms

corresponds to the structure of

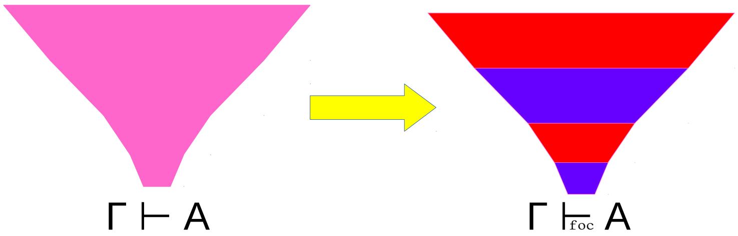
proof search

Restricting the search space restricts expression redundancy.

Research transfer from proof theory.

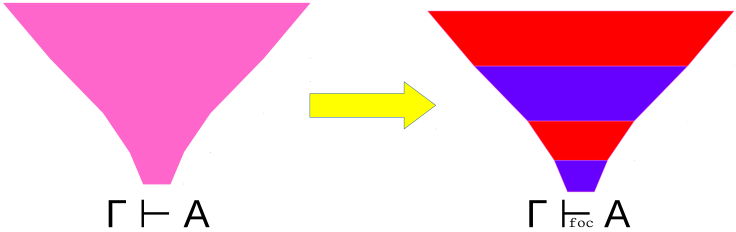
Proof search: Focusing

(existing work)



Proof search: Focusing

(existing work)



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

Proof search: Saturation

(my contribution).

Idea: make all possible deductions from the environment first.

Canonical representation.

Proof search: Saturation

(my contribution).

Idea: make all possible deductions from the environment first.

Canonical representation.

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

Saturation discovers t.

Proof search: Saturation

(my contribution).

Idea: make all possible deductions from the environment first.

Canonical representation.

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

Saturation discovers t.

(Booleans $\Rightarrow$ BDDs)

Application: program synthesis

Canonical representations tell us about program structure.

Program synthesis by searching among canonical representations.

Discussions with synthesis groups at MIT, UPenn, Princeton.
Heuristics subsumed by focusing.

Project: Search for Program Structure

Transfer proof representation techniques
to programming language applications.

Gives strong results in restricted setting (simple types),
also useful in richer languages – “more canonical” representations.

Applications: new programming language features and tools.

Continued exchange between logic and programming techniques
(inductives, second-order logic, dependent types) is necessary.

(Not detailed here)
Multi-language programming and interoperation.

Parsifal

Expertise in proof theory, focusing, automated theorem proving.

Applied mostly to proof systems so far.

Me: expertise and application goals in programming languages.

Foundational proof certificates for prover interoperability
$\leftrightarrow$ programming languages interoperation.

Ambitious programming projects (Abella, Psyche, Bedwyr, Mætning...).
$\leftrightarrow$ OCaml expertise.

Timeline

Short term

- Verified refactoring.
- Canonicity and polymorphism.
- OCaml plus expert languages.

Medium term

- Program synthesis for dependent types.
- Focusing, abstract machines and CBPV.
- Verified/unverified interoperability.

Long term

- Understanding pure program structure.
- Generic focusing and canonicity.
- Hybrid proof/program synthesis for effective verified programming.

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.