Balance in programming research

Gabriel Scherer

Parsifal, INRIA Saclay

March 21, 2019
hard questions

unsolvable terms
complexity of \( \beta \)-reduction

?  

untyped (pure) \( \lambda \)-calculus
simply-typed System F

decidable checking?
consistency?

hard systems (MLTT, Iris...)
hard questions

unsolvable terms
complexity of \( \beta \)-reduction
binders
effects

decidable checking?
consistency?

untyped (pure) λ-calculus
simply-typed System F
hard systems (MLTT, Iris...)

2
hard questions

unsolvable terms

complexity of \( \beta \)-reduction

binders

effects

proof nets

decidable checking? consistency?

untyped (pure) simply-typed hard systems
\( \lambda \)-calculus System F (MLTT, Iris...)

linear logic \( \_\_ \)
hard questions

unsolvable terms
complexity of \( \beta \)-reduction
binders
effects
proof nets
equivalence
canonicity

decidable checking?
consistency?

untyped (pure) \lambda\text{-calculus}
simply-typed System F
linear logic\textsubscript{2}
hard systems (MLTT, Iris...)
hard questions

unsolvable terms
complexity of $\beta$-reduction

binders
effects
proof nets
equivalence
canonicity

decidable checking?
consistency?

untyped (pure) $\lambda$-calculus
simply-typed System F
linear logic$_2$

hard systems (MLTT, Iris...)
hard questions

unsolvable terms
complexity of $\beta$-reduction

binders
effects
proof nets
equivalence
canonicity

decidable checking?
consistency?

untyped (pure) $\lambda$-calculus
simply-typed System F
linear logic
Prog. Lang. (OCaml)
hard systems (MLTT, Iris...)
OCaml

Strongly typed functional language – ML family. Widely used in our research communities, niche outside.


Industrial successes: languages (Rust, Webassembly), finance (at Jane Street), program analysis (at Facebook), blockchain (Tezos), unikernels (at Docker).

Important common infrastructure.

Free Software project, maintained by a distributed group of 17 volunteers. (France, UK, Japan)
I’m one of the most active maintainers.
OCaml research

Active project: more applied research for OCaml.
(Inspiration: what SPJ does beautifully for Haskell)

Last year:

- internship: safely unboxing mutually-recursive declarations
- internship: a type system for recursive value declarations
- collaboration: a paper on Merlin (ICFP Experience Report)
Focus: recursive value declarations

let rec x(t) = x(t)
let rec x = 1 + x
let rec x = 1 :: x
Focus: recursive value declarations

```plaintext
let rec x(t) = x(t)                                    fun t -> (x:Delay)(t)
let rec x = 1 + x                                        1 :: (x : Guard)
let rec x = 1 :: x                                             1 + (x : Dereference)
```
Focus: recursive value declarations

\[
\begin{align*}
\text{let rec } x(t) &= x(t) & \text{fun } t \rightarrow (x:\text{Delay})(t) \\
\text{let rec } x &= 1 + x & 1 :: (x : \text{Guard}) \\
\text{let rec } x &= 1 :: x & 1 + (x : \text{Dereference})
\end{align*}
\]

\[
m ::= \text{Ignore} \mid \text{Delay} \mid \text{Guard} \mid \text{Return} \mid \text{Dereference} \quad \Gamma ::= (x \mapsto m)^*
\]

\[\Gamma \vdash t : m\]

How to check a declaration?

\[
\text{let rec } x_1 = e_1 \ldots \text{ and } x_n = e_n \text{ in body}
\]
Focus: recursive value declarations

\[
\begin{align*}
\text{let rec } \ x(t) &= \ x(t) & \text{fun } t \rightarrow (x:\text{Delay})(t) \\
\text{let rec } x &= 1 + x & 1 :: (x : \text{Guard}) \\
\text{let rec } x &= 1 :: x & 1 + (x : \text{Dereference})
\end{align*}
\]

\[
m ::= \text{Ignore} \mid \text{Delay} \mid \text{Guard} \mid \text{Return} \mid \text{Dereference} \\
\Gamma ::= (x \mapsto m)^*
\]

\[
\Gamma \vdash t : m
\]

How to check a declaration?

\[
\text{let rec } x_1 = e_1 \ldots \text{ and } x_n = e_n \text{ in body}
\]

\[
? \vdash e_i : \text{Return}
\]
Focus: recursive value declarations

```
let rec x(t) = x(t)                     fun t -> (x:Delay)(t)
let rec x = 1 + x                     1 :: (x : Guard)
let rec x = 1 :: x                     1 + (x : Dereference)
```

\[ m ::= \text{Ignore} \mid \text{Delay} \mid \text{Guard} \mid \text{Return} \mid \text{Dereference} \]
\[ \Gamma ::= (x \mapsto m)^* \]
\[ \Gamma \vdash t : m \]

How to check a declaration?

let rec \( x_1 = e_1 \ldots \) and \( x_n = e_n \) in body

\[ ? \vdash e_i : \text{Return} \]
\[ \Gamma_i \vdash e_i : \text{Return} \]
Focus: recursive value declarations

\[
\text{let rec } x(t) = x(t) \quad \text{fun } t \to (x:\text{Delay})(t)
\]
\[
\text{let rec } x = 1 + x \quad 1 :: (x : \text{Guard})
\]
\[
\text{let rec } x = 1 :: x \quad 1 + (x : \text{Dereference})
\]

\[
m ::= \text{Ignore} \mid \text{Delay} \mid \text{Guard} \mid \text{Return} \mid \text{Dereference} \quad \Gamma ::= (x \mapsto m)^*
\]

\[
\Gamma \vdash t : m
\]

How to check a declaration?

let rec \(x_1 = e_1 \ldots \text{ and } x_n = e_n\) in body

\[
? \vdash e_i : \text{Return}
\]

\[
\Gamma_i \vdash e_i : \text{Return}
\]

\[
\forall \Gamma_i, \forall x_j, \quad \Gamma_i(x_j) \leq \text{Guard}
\]
Transition slide.
Canonicity

What is the identity of programs (\(\lambda\)-terms)?

Canonical representation: a syntactic description of the representatives of the (contextual) equivalence classes.

\[
t, u \text{ canonical} \implies t \neq_\alpha u \implies t \neq_{\text{ctx}} u
\]

Application: deciding equivalence, program synthesis (maybe?).
Just darn interesting.
Canonicity

What is the **identity** of programs ($\lambda$-terms)?

Canonical representation: a syntactic description of the representatives of the (contextual) equivalence classes.

\[ t, u \text{ canonical} \implies t \neq_\alpha u \implies t \neq_{\text{ctx}} u \]

Application: deciding equivalence, program synthesis (maybe?).

Just darn interesting.

\[ \Lambda C(\alpha, \to, \times): \beta\text{-short } \eta\text{-long normal forms.} \]
\[ \Lambda C(\alpha, \to, \times, +): \ldots \]
\[ \Lambda C(\alpha, \to, \times, 1, +, 0): ? \]

Solution proposed in 2017 using (maximal multi-)**focusing**.
Canonicity

What is the identity of programs ($\lambda$-terms)?

Canonical representation: a syntactic description of the representatives of the (contextual) equivalence classes.

\[ t, u \text{ canonical} \implies t \neq_\alpha u \implies t \neq_{\text{ctx}} u \]

Application: deciding equivalence, program synthesis (maybe?).
Just darn interesting.

$\Lambda C(\alpha, \to, \times)$: $\beta$-short $\eta$-long normal forms.
$\Lambda C(\alpha, \to, \times, +)$: ...
$\Lambda C(\alpha, \to, \times, 1, +, 0)$: ?

Solution proposed in 2017 using (maximal multi-)focusing.

Goal: richer types.
Canonicity: future work

System F: no subformula property.

\[ \Gamma, A[B/\alpha] \vdash C \]

\[ \Gamma \ni \forall \alpha. A \vdash C \]

Equivalence is undecidable in F: no decidable canonical forms.

Could we have a partial algorithm that works sometimes?
Eliminating polymorphism

Idea: probe the structure of ∀α. A through (canonical) proof search.

\[ \Gamma \vdash A \quad \Gamma \vdash B \]
\[ \Gamma \overset{\text{def}}{=} A \rightarrow B \rightarrow \alpha \vdash \alpha \]
\[ \vdash \forall \alpha. (A \rightarrow B \rightarrow \alpha) \rightarrow \alpha \]
Eliminating polymorphism

Idea: probe the structure of $\forall \alpha. A$ through (canonical) proof search.

\[
\begin{align*}
\Gamma \vdash A & \quad \Gamma \vdash B \\
\Gamma & \overset{\text{def}}{=} A \rightarrow B \rightarrow \alpha \vdash \alpha \\
\vdash \forall \alpha. (A \rightarrow B \rightarrow \alpha) \rightarrow \alpha
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A & \quad \Gamma \vdash B \\
\Gamma & \overset{\text{def}}{=} A \rightarrow \alpha, B \rightarrow \alpha \vdash \alpha \\
\vdash \forall \alpha. (A \rightarrow \alpha) \rightarrow (B \rightarrow \alpha) \rightarrow \alpha
\end{align*}
\]
Eliminating polymorphism

Idea: probe the structure of $\forall \alpha. A$ through (canonical) proof search.

\[
\begin{align*}
\Gamma \vdash A & \quad \Gamma \vdash B \\
\Gamma & \overset{\text{def}}{=} A \rightarrow B \rightarrow \alpha \vdash \alpha \\
\vdash \forall \alpha. (A \rightarrow B \rightarrow \alpha) \rightarrow \alpha
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A & \quad \Gamma \vdash B \\
\Gamma & \overset{\text{def}}{=} A \rightarrow \alpha, B \rightarrow \alpha \vdash \alpha \\
\vdash \forall \alpha. (A \rightarrow \alpha) \rightarrow (B \rightarrow \alpha) \rightarrow \alpha
\end{align*}
\]
Eliminating polymorphism

Idea: probe the structure of $\forall \alpha. A$ through (canonical) proof search.

$$
\begin{array}{c}
\Gamma \vdash A \\
\Gamma \vdash B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash A \\
\Gamma \vdash B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash A \\
\Gamma \vdash B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash A \\
\Gamma \vdash B
\end{array}
$$

$$
\begin{array}{c}
\Gamma \text{ def } A \rightarrow B \rightarrow \alpha \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, B \rightarrow \alpha \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, B \rightarrow \alpha \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, B \rightarrow \alpha \vdash \alpha
\end{array}
$$

$$
\begin{array}{c}
\Gamma \vdash \alpha \\
\Gamma \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash \alpha \rightarrow \alpha \\
\Gamma \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash \alpha \rightarrow \alpha \\
\Gamma \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, \alpha \vdash \alpha
\end{array}
$$

$$
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, \alpha \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, \alpha \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, \alpha \vdash \alpha
\end{array}
\quad
\begin{array}{c}
\Gamma \text{ def } A \rightarrow \alpha, \alpha \vdash \alpha
\end{array}
$$

On which fragments can this idea work?
Zooming out

Goal: balance between applied and theoretical research.
Zooming out

Goal: balance between applied and theoretical research.

Theory

OCaml

Practice

Types

Implicits

Effects

STLC

\( F \mu \Pi \)