Matt Brown, Jens Palsberg: *Breaking Through the Normalization Barrier: A Self-Interpreter for F_ω* (POPL 2016)

Gergő Érdi
http://gergo.erdi.hu/

Papers We Love.SG, November 2015.

Self-representation

- *Data*: in normal form
- *Quotation*: injective & total mapping of terms to data (*not* a function defined in the language! it is necessarily intensional)
- *Shallow vs. deep representation*: supports one or multiple operations
- Related: *embedding*, but that is not necessarily data

To summarize, the quotation mapping $\ulcorner \cdot \urcorner$ maps some closed term $e : \tau$ to another, normal-form term $\ulcorner e \urcorner : \mathrm{Exp}\ \tau$.

Note that Exp might be a constant type family, i.e. the representation might be untyped.

Unquoter vs. reducer

- *Unquoter*: a function, *defined in the language*, that, when applied on a quoted term, β-reduces to the term itself:

$$\text{unquote } \ulcorner e \urcorner \longrightarrow_{\beta}^{*} e$$

- *Reducer*: a function, *defined in the language*, that, when applied on a quoted term, β-reduces to the representation of the normal form of the term:
  if

$$e \longrightarrow_{\beta}^{*} v, \qquad v \text{ is in normal form}$$

  then

$$\text{reduce } \ulcorner e \urcorner \longrightarrow_{\beta}^{*} \ulcorner v \urcorner$$

Intuitive example

Suppose we have a language with

- Natural numbers
- Addition
- Strings

The following are all different terms of this language:

- `3 + 5`
- `"3 + 5"`
- `8`
- `"8"`

Then, by using a string-based representation ($\mathrm{Exp}\ \_ = \mathrm{String}$), we have

$$\mathrm{unquote}(\texttt{"3 + 5"}) \quad \longrightarrow^* \quad \texttt{3 + 5}$$

$$\mathrm{reduce}(\texttt{"3 + 5"}) \quad \longrightarrow^* \quad \texttt{"8"}$$

Selected lambda calculi – LC

$$\langle \text{term } e \rangle \models x \mid \lambda x \quad .e \mid e_1\ e_2$$

The untyped lambda calculus

- Not strongly normalizing (e.g. $(\lambda x.x\ x)\ (\lambda x.x\ x)$)
- Self-interpreter is no big deal & necessarily partial

$const = \lambda x.\lambda y.x$

Selected lambda calculi – STLC

$$\begin{array}{lcl} \langle \text{type } \tau \rangle & \models & \tau_1 \to \tau_2 \\ \langle \text{term } e \rangle & \models & x \mid \lambda x : \tau . e \mid e_1 \; e_2 \end{array}$$

The simply typed lambda calculus

- Strongly normalizing
- No type-level abstractions (incl. polymorphism)!
- Needs "base types"
- How would you type a generic self-interpreter. . . ?

$const : \mathrm{A} \to \mathrm{B} \to \mathrm{A}$
$const = \lambda x : \mathrm{A}.\lambda y : \mathrm{B}.x$

Selected lambda calculi – F

$$\begin{array}{lcl} \langle \text{kind } \kappa \rangle & \models & \star \\ \langle \text{type } \tau \rangle & \models & \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha : \kappa . \tau \\ \langle \text{term } e \rangle & \models & x \mid \lambda x : \tau . e \mid e_1\ e_2 \mid \Lambda \alpha : \kappa . e \mid e \mathbin{@} \tau \end{array}$$

System F

- Strongly normalizing
- Parametric polymorphism (note: at any rank!)
- "Atomic" types
- Self-interpreter possible?

$$const : \forall \alpha : \star . (\alpha \to \forall \beta : \star . (\beta \to \alpha))$$
$$const = \Lambda \alpha : \star . \lambda x : \alpha . \Lambda \beta : \star . \lambda y : \beta . x$$

Selected lambda calculi – F_ω

$$
\begin{array}{lcl}
\langle \text{kind } \kappa \rangle & \models & \star \mid \kappa_1 \to \kappa_2 \\
\langle \text{type } \tau \rangle & \models & \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha : \kappa . \tau \mid \lambda \alpha : \kappa . \tau \mid \tau_1 \; \tau_2 \\
\langle \text{term } e \rangle & \models & x \mid \lambda x : \tau . e \mid e_1 \; e_2 \mid \Lambda \alpha : \kappa . e \mid e \,@\, \tau
\end{array}
$$

System F_ω

- Strongly normalizing
- Parametric polymorphism (note: at any rank!)
- Type constructors, type transformers, ...
- Self-interpreter possible?

$\mathit{const} : \forall \alpha : \star . (\alpha \to \forall \beta : \star . (\beta \to \alpha))$
$\mathit{const} = \Lambda \alpha : \star . \lambda x : \alpha . \Lambda \beta : \star . \lambda y : \beta . x$

The normalization barrier

Is it possible to write a self-interpreter for a strongly normalizing λ-calculus?

The normalization barrier

Is it possible to write a self-interpreter for a strongly normalizing λ-calculus?

- Folklore says **no**.

The normalization barrier

Is it possible to write a self-interpreter for a strongly normalizing λ-calculus?

- Folklore says **no**.
- Previous results: interpretation of F in F_ω, F_ω in F_ω^+ (by encoding, for example, F $\forall$-types at $\star$ as F_ω type constructors at $\star \to \star$)
- The current paper's authors have also, previously, interpreted F_ω^+ and System U in System U (which is **not** strongly normalizing).

A proof for computable total functions

Definition

Let $\mathrm{Univ}(\mathbb{N} \to \mathbb{N})$ be the set of *universal functions* for $\mathbb{N} \to \mathbb{N}$: its elements are the functions $u : (\mathbb{N} \times \mathbb{N}) \hookrightarrow \mathbb{N}$ such that for every total, computable function $f : \mathbb{N} \to \mathbb{N}$, we have

$$\forall x \in \mathbb{N} : u(\ulcorner f \urcorner, x) = f(x)$$

(note that $\ulcorner \cdot \urcorner$ here maps total, computable functions to $\mathbb{N}$)

A proof for computable total functions (*cont'd.*)

Lemma

If $u \in \mathrm{Univ}(\mathbb{N} \to \mathbb{N})$, *then the Cantor-esque function* $d := x \mapsto u(x,x) + 1$ *is not total*

Proof.

Suppose $u \in \mathrm{Univ}(\mathbb{N} \to \mathbb{N})$ and d is total; then

$$d(\ulcorner d \urcorner) = u(\ulcorner d \urcorner, \ulcorner d \urcorner) + 1 = d(\ulcorner d \urcorner) + 1$$

which is a contradiction. $\square$

A proof for computable total functions (*cont'd.*)

Theorem

If $u \in \mathrm{Univ}(\mathbb{N} \to \mathbb{N})$, then u isn't total

Proof.

Suppose u is total. Then, $\forall x \in \mathbb{N}$, $u(x, x)$ is defined; so we have

$$d(x) = u(x, x) + 1$$

which is a perfectly cromulent definition (since $\cdot + 1$ is also total). In other words, d would be total. This contradicts our previous lemma. $\square$

So what about, e.g. F_ω instead of $\mathbb{N} \to \mathbb{N}$?

If we have a self-interpreter u for F_ω, the strong normalization of F_ω means $(u\ e)$ has a normal form for any well-typed e; in other words, u is total. So can we transform the previous theorem to say that u can't exist?

So what about, e.g. F_ω instead of $\mathbb{N} \to \mathbb{N}$?

If we have a self-interpreter u for F_ω, the strong normalization of F_ω means $(u\ e)$ has a normal form for any well-typed e; in other words, u is total. So can we transform the previous theorem to say that u can't exist?

Suppose we have an F_ω-self-interpreter u, and let's set $d := \lambda x.\lambda y.((u\ x)\ x)$. Then, if $d\ \ulcorner d \urcorner$ would be well-typed, we'd have

$$d\ \ulcorner d \urcorner \equiv_\beta \lambda y.((u\ \ulcorner d \urcorner)\ \ulcorner d \urcorner) \equiv_\beta \lambda y.(d\ \ulcorner d \urcorner)$$

which is clearly a contradiction (it'd lead to two "competing" normal forms $v \equiv_\beta \lambda y.v$). So **we can transform the lemma**.

So what about, e.g. F_ω instead of $\mathbb{N} \to \mathbb{N}$?

If we have a self-interpreter u for F_ω, the strong normalization of F_ω means $(u\ e)$ has a normal form for any well-typed e; in other words, u is total. So can we transform the previous theorem to say that u can't exist?

Suppose we have an F_ω-self-interpreter u, and let's set $d := \lambda x.\lambda y.((u\ x)\ x)$. Then, if $d\ \ulcorner d \urcorner$ would be well-typed, we'd have

$$d\ \ulcorner d \urcorner \equiv_\beta \lambda y.((u\ \ulcorner d \urcorner)\ \ulcorner d \urcorner) \equiv_\beta \lambda y.(d\ \ulcorner d \urcorner)$$

which is clearly a contradiction (it'd lead to two "competing" normal forms $v \equiv_\beta \lambda y.v$). So **we can transform the lemma**.

However, just because u, d and $\ulcorner d \urcorner$ are well-typed, it doesn't mean $d\ \ulcorner d \urcorner$ needs to be well-typed! The diagonalization gadget is not expressible inside F_ω. **The theorem hasn't been successfully transformed!**

A cheap & cheerful self-interpreter for $\mathsf{F}/\mathsf{F}_\omega/\mathsf{F}_\omega^+$

Of course, just because one particular proof of impossibility failed, doesn't mean self-interpretation is possible. So the first proof the paper presents is a simple, *shallow* representation that only supports an unquoter.

A cheap & cheerful self-interpreter for $F/F_\omega/F_\omega^+$

Let's look at the following example:

$$\begin{aligned}
\mathit{const} : \quad & \forall \alpha : \star.\alpha \to (\forall \beta : \star.\beta \to \alpha) \\
\mathit{const} = \quad & \Lambda \alpha : \star.\lambda x : \alpha.\Lambda \beta : \star.\lambda y : \beta.x \\[1em]
\mathit{id} : \quad & \forall \alpha : \star.\alpha \to \alpha \\
\mathit{id} = \quad & \Lambda \alpha : \star.\lambda x : \alpha.x \\[1em]
\mathit{foo} : \quad & \forall \beta : \star.\beta \to \forall \alpha : \star.\alpha \to \alpha \\
\mathit{foo} = \quad & \mathit{const} \, @ (\forall \alpha : \star.\alpha \to \alpha) \ \mathit{id}
\end{aligned}$$

A cheap & cheerful self-interpreter for $F/F_\omega/F_\omega^+$

For reference, after inlining the helper definitions, we have

$$foo = (\Lambda\alpha : \star.\lambda x : \alpha.\Lambda\beta : \star.\lambda y : \beta.x) \,@(\forall\alpha : \star.\alpha \to \alpha)\,(\Lambda\alpha : \star.\lambda x : \alpha.x)$$

A smart-ass non-solution

Why not just represent everything as itself, and set *unquote* := *id*?

For reference, after inlining the helper definitions, we have

$$foo = (\Lambda\alpha : \star.\lambda x : \alpha.\Lambda\beta : \star.\lambda y : \beta.x) \mathbin{@} (\forall\alpha : \star.\alpha \to \alpha)\,(\Lambda\alpha : \star.\lambda x : \alpha.x)$$

A smart-ass non-solution

Why not just represent everything as itself, and set *unquote* := *id*? Because *foo* is not in normal form, so it isn't data! The type application, and then the outermost term application can be β-reduced away.

A cheap & cheerful self-interpreter for $F/F_\omega/F_\omega^+$

For reference, after inlining the helper definitions, we have

$$\mathit{foo} = (\Lambda\alpha:\star.\lambda x:\alpha.\Lambda\beta:\star.\lambda y:\beta.x)\,@(\forall\alpha:\star.\alpha\to\alpha)\,(\Lambda\alpha:\star.\lambda x:\alpha.x)$$

A smart-ass non-solution

Why not just represent everything as itself, and set *unquote* := *id*? Because *foo* is not in normal form, so it isn't data! The type application, and then the outermost term application can be β-reduced away.

Central idea of the paper: **replace the applications with application-markers**!

A cheap & cheerful self-interpreter for $F/F_\omega/F_\omega^+$

Central idea of the paper: **replace the applications with application-markers**!
Where are all the applications in our example?

$$foo = \boxed{\boxed{(\Lambda\alpha : \star.\lambda x : \alpha.\Lambda\beta : \star.\lambda y : \beta.x) \mathbin{@} (\forall\alpha : \star.\alpha \to \alpha)}\ (\Lambda\alpha : \star.\lambda x : \alpha.x)}$$

To ensure there are no (reducable) applications left, let's apply a marker $\diamond$, which is a free variable, on all terms which are applied on either types or terms:

$$\boxed{\diamond\ \boxed{\diamond\ \Lambda\alpha : \star.\lambda x : \alpha.\Lambda\beta : \star.\lambda y : \beta.x} \mathbin{@} (\forall\alpha : \star.\alpha \to \alpha)}\ (\Lambda\alpha : \star.\lambda x : \alpha.x)$$

A cheap & cheerful self-interpreter for F/F_ω/F_ω^+

$$\boxed{\diamond\ \boxed{\diamond\ \Lambda\alpha : \star.\lambda x : \alpha.\Lambda\beta : \star.\lambda y : \beta.x}\ @(\forall\alpha : \star.\alpha \to \alpha)}\ (\Lambda\alpha : \star.\lambda x : \alpha.x)$$

Of course, $\diamond$ needs to be polymorphic, so we'll need to sprinkle our code with some type applications that duplicate the types of the originally-applied functions:

$$\begin{array}{l} \diamond\ @((\forall\alpha : \star.\alpha \to \alpha) \to \forall\beta : \star.\beta \to (\forall\alpha : \star.\alpha \to \alpha)) \\ \quad \boxed{\diamond\ \ @(\forall\alpha : \star.\alpha \to (\forall\beta : \star.\beta \to \alpha))\ \ \mathit{const}}\ @(\forall\alpha : \star.\alpha \to \alpha)) \\ \quad \mathit{id} \end{array}$$

A cheap & cheerful self-interpreter for $F/F_\omega/F_\omega^+$

If we now close this by putting it under a $\diamond$-binding λ, we have our representation:

$$
\begin{aligned}
\ulcorner \mathit{foo} \urcorner :\quad & \mathrm{Exp}\ (\forall\beta:\star.\beta \to \forall\alpha:\star.\alpha \to \alpha) \\
\ulcorner \mathit{foo} \urcorner = \quad & \lambda\diamond : (\forall\iota:\star.\iota \to \iota). \\
& \ \diamond @((\forall\alpha:\star.\alpha\to\alpha) \to \forall\beta:\star.\beta \to (\forall\alpha:\star.\alpha\to\alpha)) \\
& \quad ((\diamond @(\forall\alpha:\star.\alpha \to (\forall\beta:\star.\beta \to \alpha))\ \mathit{const})\, @(\forall\alpha:\star.\alpha\to\alpha)) \\
& \quad \mathit{id}
\end{aligned}
$$

With

$$\mathrm{Exp}\ \tau = (\forall\iota:\star.\iota\to\iota) \to \tau$$

and all $\mathrm{unquote}$ needs to do is plug in *id* (the “un-marker”) as the marker:

$$
\begin{aligned}
\mathrm{unquote} &: \forall\alpha:\star.((\forall\iota:\star.\iota\to\iota)\to\alpha)\to\alpha \\
\mathrm{unquote} &= \Lambda\alpha:\star.\lambda q:(\forall\iota:\star.\iota\to\iota)\to\alpha.q\ \mathit{id}
\end{aligned}
$$

A cheap & cheerful self-interpreter for $\mathsf{F}/\mathsf{F}_\omega/\mathsf{F}_\omega^+$

Theorem

If $\{\} \vdash e : \tau$ *then* $\{\} \vdash \ulcorner e \urcorner : (\forall \iota : \star . \iota \to \iota) \to \tau$

Theorem

If $\{\} \vdash e : \tau$ *then* $\mathrm{unquote} \mathbin{@} \tau \ \ulcorner e \urcorner \longrightarrow^* e$

Summary of the technique

- Suspend reducability by marking each applied term with a free variable
- Process the representation by plugging in a suitable function for the marker

Summary of the technique

- Suspend reducability by marking each applied term with a free variable
- Process the representation by plugging in a suitable function for the marker

But is this just a cheap trick? So what if there's a bunch of places where we can apply *id*?

Summary of the technique

- Suspend reducability by marking each applied term with a free variable
- Process the representation by plugging in a suitable function for the marker

But is this just a cheap trick? So what if there's a bunch of places where we can apply *id*?
Not at all!
The marker's type just happens to be the trivial $\iota \to \iota$ in this shallow unquoter case, but the technique generalizes by mapping subresults (of some type) to a larger result (of some, possibly different, type); i.e., a fold.

A deep self-representation of F_ω

The grand result of the paper is a *deep* self-representation of F_ω (unlike the previous, shallow representation, this doesn't readily work in F or F_ω^+)

- Deep representation means the same representation supports multiple operations (late binding of the operation)
- Examples from the paper:
  - *isAbs*, *isNF*, *size*
  - *unquote*
  - *CPS*

A deep self-representation of F_ω

The grand result of the paper is a *deep* self-representation of F_ω (unlike the previous, shallow representation, this doesn't readily work in F or F_ω^+)

- Deep representation means the same representation supports multiple operations (late binding of the operation)
- Examples from the paper:
  - *isAbs*, *isNF*, *size*
  - *unquote*
  - *CPS*
- I will only cover it cursorily in this talk; see the paper for details

A deep self-representation of F_ω

Given $\{\} \vdash e : \tau$, first τ is transformed into $\ulcorner\tau\urcorner$ by iteratively wrapping each $\star$-kinded (non-type variable) subtree in a (free type variable) type constructor $F : \star \to \star$

Example:

$$\ulcorner\forall\alpha : \star.\alpha \to \alpha\urcorner = \forall\alpha : \star.\ F\ (F\ \alpha \to F\ \alpha)$$

This means types at κ become types at $\ulcorner\kappa\urcorner := (\star \to \star) \to \kappa$; in particular, types at $\star$ become types at $\ulcorner\star\urcorner = (\star \to \star) \to \star$.

A deep self-representation of F_ω

Then, for $\ulcorner e \urcorner$, each λ-abstraction, application, Λ-abstraction, and type application of the term is wrapped into calls of one of four free variables, of types

$$\begin{aligned}
\mathit{Abs}\ F &= \forall\alpha:\star.\forall\beta:\star.\ (F\ \ulcorner\alpha\urcorner \to F\ \ulcorner\beta\urcorner) \to F\ \ulcorner\alpha\to\beta\urcorner \\
\mathit{App}\ F &= \forall\alpha:\star.\forall\beta:\star.\ F\ \ulcorner\alpha\to\beta\urcorner \to F\ \ulcorner\alpha\urcorner \to F\ \ulcorner\beta\urcorner \\
\mathit{TAbs}\ F &= \forall\alpha:\star.\ \mathit{Strip}\ F\ \alpha \to \alpha \to F\ \ulcorner\alpha\urcorner
\end{aligned}$$

$$\mathit{TApp}\ F = \forall\alpha:\star.\ F\ \ulcorner\alpha\urcorner \to \forall\beta:\star.(\alpha \to F\ \beta) \to F\ \ulcorner\beta\urcorner$$

A deep self-representation of F_ω

Then, for $\ulcorner e \urcorner$, each λ-abstraction, application, Λ-abstraction, and type application of the term is wrapped into calls of one of four free variables, of types

$$\begin{aligned}
\mathit{Abs}\ F \quad &= \forall \alpha : \star . \forall \beta : \star .\ (F\ \ulcorner \alpha \urcorner \to F\ \ulcorner \beta \urcorner) \to F\ \ulcorner \alpha \to \beta \urcorner \\
\mathit{App}\ F \quad &= \forall \alpha : \star . \forall \beta : \star .\ F\ \ulcorner \alpha \to \beta \urcorner \to F\ \ulcorner \alpha \urcorner \to F\ \ulcorner \beta \urcorner \\
\mathit{TAbs}\ F \quad &= \forall \alpha : \star .\ \mathit{Strip}\ F\ \alpha \to \alpha \to F\ \ulcorner \alpha \urcorner \\
&\qquad \text{where } \mathit{Strip}\ F\ \alpha = \forall \beta : \star .\ (\forall \gamma : \star .\ F\ \gamma \to \beta) \to \alpha \to \beta \\
\mathit{TApp}\ F \quad &= \forall \alpha : \star .\ F\ \ulcorner \alpha \urcorner \to \forall \beta : \star . (\alpha \to F\ \beta) \to F\ \ulcorner \beta \urcorner
\end{aligned}$$

A deep self-representation of F_ω

The representation of a term $\{\} \vdash e : \tau$ thus becomes a term $\{\} \vdash \ulcorner e \urcorner : Exp\ (\lambda F : \star \to \star.\ \ulcorner \tau \urcorner)$, with

$$
\begin{aligned}
Exp \quad = & \lambda\alpha : \ulcorner \star \urcorner.\ \forall F : \star \to \star. \\
& Abs\ F \to App\ F \to TAbs\ F \to TApp\ F \to \\
& F\ (\alpha\ F)
\end{aligned}
$$

Note that $Exp\ \ulcorner \tau \urcorner$ is still parametrized over the choice of $F : \star \to \star$, which is what ultimately allows the late-binding of the choice of operation over the representation.

Summary

- F applied (iteratively) only to types of kind $\star \Rightarrow$ no need to abstract F's type over kinds
- Parametric (in F) HOAS representation for terms; variables range over representations
- A particular operation defines its own choice of F, and implements the four "callbacks"
  - For *unquote*, $F = \lambda\alpha : \star.\ \alpha$, so e.g. *app* $: \forall\alpha : \star.\ \forall\beta : \star.\ (\alpha \to \beta) \to \alpha \to \beta$
  - For *isAbs*, $F = \lambda\alpha : \star.\ \mathrm{Bool}$, and so *app* $: \forall\alpha : \star.\ \forall\beta : \star.\ \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool}$
  - For *size*, $F = \lambda\alpha : \star.\ \mathrm{Nat}$, giving *app* $: \forall\alpha : \star.\ \forall\beta : \star.\ \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$