Matt Brown, Jens Palsberg:
Breaking Through the Normalization Barrier:
A Self-Interpreter for $F_\omega$ (POPL 2016)

Gergő Érdi
http://gergo.erdhi.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 $\bar{\cdot}$ maps some closed term $e : \tau$ to another, normal-form term $\bar{e} : \text{Exp} \ \tau$.

Note that $\text{Exp}$ 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, $\beta$-reduces to the term itself:

  \[
  \text{unquote } [e] \rightarrow^\ast e
  \]

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

  if
  \[
  e \rightarrow^\ast v, \quad v \text{ is in normal form}
  \]
  then
  \[
  \text{reduce } [e] \rightarrow^\ast [v]
  \]
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 ($\text{Exp} \ = \ \text{String}$), we have

\[
\begin{align*}
\text{unquote}("3 + 5") & \quad \rightarrow^* \quad 3 + 5 \\
\text{reduce}("3 + 5") & \quad \rightarrow^* \quad "8"
\end{align*}
\]
\[ \langle \text{term } e \rangle \models x \mid \lambda x \ . e \mid e_1 \ e_2 \]

The untyped lambda calculus

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

\[ \text{const} = \lambda x. \lambda y. x \]
The simply typed lambda calculus

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

\[\begin{align*}
\text{type } \tau & \models \tau_1 \rightarrow \tau_2 \\
\text{term } e & \models x \mid \lambda x : \tau. e \mid e_1 \; e_2
\end{align*}\]

\textit{const} : A \rightarrow B \rightarrow A
\textit{const} = \lambda x : A. \lambda y : B. x
Selected lambda calculi – F

\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 \circ \tau

System F

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

\textit{const} : \forall \alpha : \star.(\alpha \rightarrow \forall \beta : \star.(\beta \rightarrow \alpha))
\textit{const} = \Lambda \alpha : \star.\lambda x : \alpha.\Lambda \beta : \star.\lambda y : \beta. x
Selected lambda calculi – $F_\omega$

\[\langle \text{kind } \kappa \rangle \models * \mid \kappa_1 \rightarrow \kappa_2\]
\[\langle \text{type } \tau \rangle \models \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha : \kappa.\tau \mid \lambda \alpha : \kappa.\tau \mid \tau_1 \tau_2\]
\[\langle \text{term } e \rangle \models x \mid \lambda x : \tau.e \mid e_1 e_2 \mid \Lambda \alpha : \kappa.e \mid e \circ \tau\]

System $F_\omega$

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

\[
\text{const} : \forall \alpha : \star. (\alpha \rightarrow \forall \beta : \star. (\beta \rightarrow \alpha))
\]
\[
\text{const} = \Lambda \alpha : \star. \lambda x : \alpha. \Lambda \beta : \star. \lambda y : \beta.x
\]
The normalization barrier

Is it possible to write a self-interpreter for a strongly normalizing \(\lambda\)-calculus?
Is it possible to write a self-interpreter for a strongly normalizing $\lambda$-calculus?

- Folklore says no.
Is it possible to write a self-interpreter for a strongly normalizing $\lambda$-calculus?

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

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

\[
\forall x \in \mathbb{N} : u([f], x) = f(x)
\]

(note that \([\cdot] \) here maps total, computable functions to \( \mathbb{N} \))
Lemma

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

Proof.

Suppose \( u \in \text{Univ}(\mathbb{N} \rightarrow \mathbb{N}) \) and \( d \) is total; then
\[
d([d']) = u([d'], [d']) + 1 = d([d']) + 1
\]
which is a contradiction.
A proof for computable total functions (cont’d.)

**Theorem**

\[ u \in \text{Univ}(\mathbb{N} \rightarrow \mathbb{N}), \text{ then } u \text{ isn't total} \]

**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.
So what about, e.g. \( F_\omega \) instead of \( \mathbb{N} \to \mathbb{N} \)?

If we have a self-interpreter \( u \) for \( F_\omega \), the strong normalization of \( F_\omega \) means \((u \cdot e)\) 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?

However, just because \( u, d, v, d, w \) are well-typed, it doesn’t mean \( d, v, d, w \) needs to be well-typed! The diagonalization gadget is not expressible inside \( F_\omega \). The theorem hasn’t been successfully transformed!
So what about, e.g. \( F_\omega \) instead of \( \mathbb{N} \rightarrow \mathbb{N} \)?

If we have a self-interpreter \( u \) for \( F_\omega \), the strong normalization of \( F_\omega \) 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 \( F_\omega \)-self-interpreter \( u \), and let’s set
\[
d := \lambda x.\lambda y.((u \ x) \ x).
\]
Then, if \( d [d] \) would be well-typed, we’d have
\[
d [d] \equiv_\beta \lambda y.((u [d] [d]) \equiv_\beta_\beta \lambda y.(d [d])
\]
which is clearly a contradiction (it’d lead to two “competing” normal forms \( \nu \equiv_\beta \lambda y.\nu \)). So **we can transform the lemma.**
So what about, e.g. $F_\omega$ instead of $\mathbb{N} \rightarrow \mathbb{N}$?

If we have a self-interpreter $u$ for $F_\omega$, the strong normalization of $F_\omega$ 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 $F_\omega$-self-interpreter $u$, and let’s set $d := \lambda x.\lambda y.((u \ x) \ x)$. Then, if $d \ [d']$ would be well-typed, we’d have

$$d \ [d'] \equiv_\beta \lambda y.((u \ [d']) \ [d']) \equiv_\beta \lambda y.(d \ [d'])$$

which is clearly a contradiction (it’d lead to two “competing” normal forms $\nu \equiv_\beta \lambda y.\nu$). So we can transform the lemma.

However, just because $u$, $d$ and $[d']$ are well-typed, it doesn’t mean $d \ [d']$ needs to be well-typed! The diagonalization gadget is not expressible inside $F_\omega$. The theorem hasn’t been successfully transformed!
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.
Let’s look at the following example:

\[
\begin{align*}
\text{const} : & \quad \forall \alpha : \star.\alpha \to (\forall \beta : \star.\beta \to \alpha) \\
\text{const} = & \quad \Lambda \alpha : \star.\lambda x : \alpha.\Lambda \beta : \star.\lambda y : \beta.x \\
\text{id} : & \quad \forall \alpha : \star.\alpha \to \alpha \\
\text{id} = & \quad \Lambda \alpha : \star.\lambda x : \alpha.x \\
\text{foo} : & \quad \forall \beta : \star.\beta \to \forall \alpha : \star.\alpha \to \alpha \\
\text{foo} = & \quad \text{const} \circ (\forall \alpha : \star.\alpha \to \alpha) \ id
\end{align*}
\]
For reference, after inlining the helper definitions, we have

\[ \text{foo} = (\Lambda \alpha : \star. \lambda x : \alpha. \Lambda \beta : \star. \lambda y : \beta. x) \circ (\forall \alpha : \star. \alpha \rightarrow \alpha) (\Lambda \alpha : \star. \lambda x : \alpha. x) \]

A smart-ass non-solution

Why not just represent everything as itself, and set \textit{unquote} := \textit{id}?
For reference, after inlining the helper definitions, we have

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

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 \(\beta\)-reduced away.
For reference, after inlining the helper definitions, we have

$$foo = (\Lambda \alpha : \ast. \lambda x : \alpha. \Lambda \beta : \ast. \lambda y : \beta. x) \circ (\forall \alpha : \ast. \alpha \rightarrow \alpha) \ (\Lambda \alpha : \ast. \lambda x : \alpha. x)$$

A smart-ass non-solution

Why not just represent everything as itself, and set \textit{unquote} := \textit{id}?

Because \textit{foo} is not in normal form, so it isn’t data! The type application, and then the outermost term application can be \(\beta\)-reduced away.

Central idea of the paper: \textbf{replace the applications with application-markers!}
Central idea of the paper: replace the applications with application-markers!

Where are all the applications in our example?

\[
\text{foo} = \boxed{(\Lambda \alpha : \star. \lambda x : \alpha. \Lambda \beta : \star. \lambda y : \beta. x) \circ (\forall \alpha : \star. \alpha \to \alpha)} \quad (\Lambda \alpha : \star. \lambda x : \alpha. x)
\]

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

\[
\diamond \diamond \boxed{\Lambda \alpha : \star. \lambda x : \alpha. \Lambda \beta : \star. \lambda y : \beta. x} \circ (\forall \alpha : \star. \alpha \to \alpha) \quad (\Lambda \alpha : \star. \lambda x : \alpha. x)
\]
A cheap & cheerful self-interpreter for $F/F^+_\omega$/$F^+_\omega$

\[
\diamond \diamond \Lambda \alpha : \star. \lambda x : \alpha. \Lambda \beta : \star. \lambda y : \beta. x \ominus (\forall \alpha : \star. \alpha \to \alpha) \quad (\Lambda \alpha : \star. \lambda x : \alpha. x)
\]

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:

\[
\diamond \ominus ((\forall \alpha : \star. \alpha \to \alpha) \to \forall \beta : \star. \beta \to (\forall \alpha : \star. \alpha \to \alpha))
\]

\[
\diamond \ominus (\forall \alpha : \star. \alpha \to (\forall \beta : \star. \beta \to \alpha)) \quad \text{const} \quad \ominus (\forall \alpha : \star. \alpha \to \alpha)
\]

\[id\]
If we now close this by putting it under a ◇-binding λ, we have our representation:

\[
\begin{align*}
&[\text{foo}] : \quad \text{Exp} \ (\forall \beta : \star.\beta \to \forall \alpha : \star.\alpha \to \alpha) \\
&[\text{foo}] = \quad \lambda \ \diamond : (\forall \iota : \star.\iota \to \iota). \\
&\quad \diamond \circ ((\forall \alpha : \star.\alpha \to \alpha) \to \forall \beta : \star.\beta \to (\forall \alpha : \star.\alpha \to \alpha)) \\
&\quad ((\diamond \circ (\forall \alpha : \star.\alpha \to (\forall \beta : \star.\beta \to \alpha)) \ const) \circ (\forall \alpha : \star.\alpha \to \alpha)) \\
&\quad id
\end{align*}
\]

With

\[
\text{Exp} \ \tau = (\forall \iota : \star.\iota \to \iota) \to \tau
\]

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

\[
\begin{align*}
&\text{unquote} : \forall \alpha : \star.((\forall \iota : \star.\iota \to \iota) \to \alpha) \to \alpha \\
&\text{unquote} = \Lambda \alpha : \star.\lambda q : (\forall \iota : \star.\iota \to \iota) \to \alpha.q \ id
\end{align*}
\]
A cheap & cheerful self-interpreter for $F/F_\omega/F_\omega^+$

Theorem

\[
\text{If } \{ \} \vdash e : \tau \text{ then } \{ \} \vdash [e] : (\forall \ell : \star. \ell \rightarrow \ell) \rightarrow \tau
\]

Theorem

\[
\text{If } \{ \} \vdash e : \tau \text{ then } \text{unquote } \odot \tau [e] \rightarrow^* e
\]
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 \textit{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.
The grand result of the paper is a *deep* self-representation of $F_\omega$ (unlike the previous, shallow representation, this doesn’t readily work in $F$ or $F^+_\omega$)

- Deep representation means the same representation supports multiple operations (late binding of the operation)
- Examples from the paper:
  - `isAbs`, `isNF`, `size`
  - `unquote`
  - `CPS`
The grand result of the paper is a *deep* self-representation of $F_\omega$ (unlike the previous, shallow representation, this doesn’t readily work in $F$ or $F_\omega^+$)

- 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 $F_\omega$

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

Example:

\[
[^\forall \alpha : \star \cdot \alpha \rightarrow \alpha] = \forall \alpha : \star. \ F (F \alpha \rightarrow F \alpha)
\]

This means types at $\kappa$ become types at $[^\kappa] := (\star \rightarrow \star) \rightarrow \kappa$; in particular, types at $\star$ become types at $[^\star] = (\star \rightarrow \star) \rightarrow \star$. 
Then, for $[e]$, each $\lambda$-abstraction, application, $\Lambda$-abstraction, and type application of the term is wrapped into calls of one of four free variables, of types

- **Abs $F$**
  \[= \forall \alpha : \ast . \forall \beta : \ast . (F [\alpha] \rightarrow F [\beta]) \rightarrow F [\alpha \rightarrow \beta] \]

- **App $F$**
  \[= \forall \alpha : \ast . \forall \beta : \ast . F [\alpha \rightarrow \beta] \rightarrow F [\alpha] \rightarrow F [\beta] \]

- **TAbs $F$**
  \[= \forall \alpha : \ast. \text{Strip } F \alpha \rightarrow \alpha \rightarrow F [\alpha] \]

- **TApp $F$**
  \[= \forall \alpha : \ast . F [\alpha] \rightarrow \forall \beta : \ast . (\alpha \rightarrow F \beta) \rightarrow F [\beta] \]
A deep self-representation of $F_\omega$

Then, for $[e]$, each $\lambda$-abstraction, application, $\Lambda$-abstraction, and type application of the term is wrapped into calls of one of four free variables, of types

$$\begin{align*}
\text{Abs } F &= \forall \alpha : *. \forall \beta : *. (F_\alpha \to F_\beta) \to F_{\alpha \to \beta} \\
\text{App } F &= \forall \alpha : *. \forall \beta : *. F_\alpha \to \beta \to F_{\alpha} \to F_{\beta} \\
\text{TAbs } F &= \forall \alpha : *. \text{Strip } F \alpha \to \alpha \to F_\alpha \\
\text{where } \text{Strip } F \alpha &= \forall \beta : *. (\forall \gamma : *. F_\gamma \gamma \to \beta) \to \alpha \to \beta \\
\text{TApp } F &= \forall \alpha : *. F_{\alpha} \to \forall \beta : *. (\alpha \to F_{\beta}) \to F_{\beta}
\end{align*}$$
A deep self-representation of $F_\omega$

The representation of a term $\{\} \vdash e : \tau$ thus becomes a term $\{\} \vdash [e] : Exp (\lambda F : * \rightarrow *. [\tau])$, with

$$Exp = \lambda \alpha : [*]. \forall F : * \rightarrow *.$$

Abs $F \rightarrow$ App $F \rightarrow$ TAbs $F \rightarrow$ TApp $F \rightarrow$

$F (\alpha F)$

Note that $Exp$ $[\tau]$ is still parametrized over the choice of $F : * \rightarrow *$, 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.
    - $\text{app} : \forall \alpha : \star. \forall \beta : \star. (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$
  - For `isAbs`, $F = \lambda \alpha : \star. \text{Bool}$, and so
    - $\text{app} : \forall \alpha : \star. \forall \beta : \star. \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$
  - For `size`, $F = \lambda \alpha : \star. \text{Nat}$, giving
    - $\text{app} : \forall \alpha : \star. \forall \beta : \star. \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$