Matt Brown, Jens Palsberg: *Typed Self-Evaluation via Intensional Type Functions* (POPL 2017)

Gergő Érdi
http://gergo.erdi.hu/

Papers We Love.SG × HaskellSG
February 2017.

Setting the scene

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 some normal form of the term:
  if

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

  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

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

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

Selected lambda calculi – LC

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

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

$$\langle \text{type } \tau \rangle \models \tau_1 \to \tau_2$$
$$\langle \text{term } e \rangle \models x \mid \lambda x : \tau . e \mid e_1\ e_2$$

The simply typed lambda calculus

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

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

Selected lambda calculi – F

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

System F

- Strongly normalizing
- Parametric polymorphism (note: at any rank!)
- "Atomic" types
- Self-unquoter: see authors' paper from POPL 2016

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

Selected lambda calculi – F_ω

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

System F_ω

- Strongly normalizing
- Parametric polymorphism (note: at any rank!)
- Type constructors, type transformers, ...
- Self-unquoter: see authors' paper from POPL 2016

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

Typed evaluation of STLC in Haskell

STLC evaluator in Haskell – explicit variables

$$
\begin{array}{l}
\textbf{data}\ \mathit{Ctx}\ a = O \mid \mathit{Ctx}\ a :> a \\
\textbf{data}\ \mathit{Var}\ \mathit{ctx}\ t\ \textbf{where} \\
\quad \mathit{VZ} :: \mathit{Var}\ (\mathit{ts} :> t)\ t \\
\quad \mathit{VS} :: \mathit{Var}\ \mathit{ts}\ t \to \mathit{Var}\ (\mathit{ts} :> t_0)\ t \\
\textbf{data}\ \mathit{Exp}\ \mathit{ctx}\ t\ \textbf{where} \\
\quad \mathit{Var} :: \mathit{Var}\ \mathit{ctx}\ t \to \mathit{Exp}\ \mathit{ctx}\ t \\
\quad \mathit{App} :: \mathit{Exp}\ \mathit{ctx}\ (t_0 \to t) \to \mathit{Exp}\ \mathit{ctx}\ t_0 \to \mathit{Exp}\ \mathit{ctx}\ t \\
\quad \mathit{Abs} :: \mathit{Exp}\ (\mathit{ctx} :> t_1)\ t_2 \to \mathit{Exp}\ \mathit{ctx}\ (t_1 \to t_2)
\end{array}
$$

STLC evaluator in Haskell – explicit variables

$$
\begin{array}{l}
\textbf{data}\ \mathit{Ctx}\ a = O \mid \mathit{Ctx}\ a :> a \\
\textbf{data}\ \mathit{Var}\ \mathit{ctx}\ t\ \textbf{where} \\
\quad \mathit{VZ} :: \mathit{Var}\ (\mathit{ts} :> t)\ t \\
\quad \mathit{VS} :: \mathit{Var}\ \mathit{ts}\ t \to \mathit{Var}\ (\mathit{ts} :> t_0)\ t \\
\textbf{data}\ \mathit{Exp}\ \mathit{ctx}\ t\ \textbf{where} \\
\quad \mathit{Var} :: \mathit{Var}\ \mathit{ctx}\ t \to \mathit{Exp}\ \mathit{ctx}\ t \\
\quad \mathit{App} :: \mathit{Exp}\ \mathit{ctx}\ (t_0 \to t) \to \mathit{Exp}\ \mathit{ctx}\ t_0 \to \mathit{Exp}\ \mathit{ctx}\ t \\
\quad \mathit{Abs} :: \mathit{Exp}\ (\mathit{ctx} :> t_1)\ t_2 \to \mathit{Exp}\ \mathit{ctx}\ (t_1 \to t_2)
\end{array}
$$

But this requires capture-avoiding substitution...

STLC evaluator in Haskell – HOAS

HOAS approach to writing a typed STLC evaluator:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ t\ \textbf{where} \\
\quad \mathit{Abs} :: (\mathit{Exp}\ t_1 \rightarrow \mathit{Exp}\ t_2) \rightarrow \qquad\qquad\qquad\ \ \mathit{Exp}\ (t_1 \rightarrow t_2) \\
\quad \mathit{App} :: \mathit{Exp}\ (t_0 \rightarrow t) \qquad\quad \rightarrow \mathit{Exp}\ t_0 \rightarrow \mathit{Exp}\ t
\end{array}
$$

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Exp}\ t \rightarrow \mathit{Exp}\ t \\
\mathit{eval}\ (\mathit{App}\ e_1\ e_2) = \textbf{case}\ \mathit{eval}\ e_1\ \textbf{of} \\
\quad \mathit{Abs}\ f \rightarrow \mathit{eval}\ (f\ e_2) \\
\quad e_1' \qquad \rightarrow \mathit{App}\ e_1'\ e_2 \\
\mathit{eval}\ e = e
\end{array}
$$

STLC evaluator in Haskell – HOAS

HOAS approach to writing a typed STLC evaluator:

$$
\begin{array}{l}
\{\text{-\# LANGUAGE GADTs \#-}\} \\
\textbf{data}\ \mathit{Exp}\ t\ \textbf{where} \\
\quad \mathit{Abs} :: (\mathit{Exp}\ t_1 \to \mathit{Exp}\ t_2) \to \qquad\qquad\qquad \mathit{Exp}\ (t_1 \to t_2) \\
\quad \mathit{App} :: \mathit{Exp}\ (t_0 \to t) \qquad \to \mathit{Exp}\ t_0 \to \mathit{Exp}\ t \\
\mathit{eval} :: \mathit{Exp}\ t \to \mathit{Exp}\ t \\
\mathit{eval}\ (\mathit{App}\ e_1\ e_2) = \textbf{case}\ \mathit{eval}\ e_1\ \textbf{of} \\
\quad \mathit{Abs}\ f \to \mathit{eval}\ (f\ e_2) \\
\quad e_1' \qquad \to \mathit{App}\ e_1'\ e_2 \\
\mathit{eval}\ e = e
\end{array}
$$

Actually, Haskell + GADTs

STLC evaluator in Haskell – HOAS

HOAS approach to writing a typed STLC evaluator:

$$
\begin{array}{l}
\{\text{-\# LANGUAGE GADTs \#-}\} \\
\textbf{data}\ \mathit{Exp}\ t\ \textbf{where} \\
\quad \mathit{Abs} :: (\mathit{Exp}\ t_1 \rightarrow \mathit{Exp}\ t_2) \rightarrow \qquad\qquad\qquad\quad \mathit{Exp}\ (t_1 \rightarrow t_2) \\
\quad \mathit{App} :: \mathit{Exp}\ (t_0 \rightarrow t) \qquad\quad \rightarrow \mathit{Exp}\ t_0 \rightarrow \mathit{Exp}\ t \\
\mathit{eval} :: \mathit{Exp}\ t \rightarrow \mathit{Exp}\ t \\
\mathit{eval}\ (\mathit{App}\ e_1\ e_2) = \textbf{case}\ \mathit{eval}\ e_1\ \textbf{of} \\
\quad \mathit{Abs}\ f \rightarrow \mathit{eval}\ (f\ e_2) \\
\quad e_1' \qquad \rightarrow \mathit{App}\ e_1'\ e_2 \\
\mathit{eval}\ e = e
\end{array}
$$

Actually, Haskell + GADTs
Simple evaluator for a simple target language. . . written in a very rich & complex host language.

Host vs. target language

Haskell w/GADTs

Totality

STLC

Getting rid of some sugar – GADTs

The feature that GADTs bring to the table is non-parametric types. We can encode that with explicit equalities:

```
  -- yeah we cheat here because Eq itself is a GADT
  -- (more on that later)
data Eq a b where
  Refl :: Eq a a
```

We are now left with regular ADTs and existentials:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ t \\
\quad = \forall t_1\ t_2 . \mathit{Abs}\ (\mathit{Eq}\ (t_1 \to t_2)\ t)\ (\mathit{Exp}\ t_1 \to \mathit{Exp}\ t_2) \\
\quad \mid\ \forall t_0 . \quad \mathit{App}\ (\mathit{Exp}\ (t_0 \to t))\ (\mathit{Exp}\ t_0)
\end{array}
$$

Theory of type equalities

We need a theory of equalities to be able to use them. These are easily provable in Haskell, not new axioms.

-- It is an equivalence relation
$\mathit{refl} :: \mathit{Eq}\ a\ a$
$\mathit{sym} :: \mathit{Eq}\ a\ b \to \mathit{Eq}\ b\ a$
$\mathit{trans} :: \mathit{Eq}\ a\ b \to \mathit{Eq}\ b\ c \to \mathit{Eq}\ a\ c$

-- It is a congruence over type constructors, allowing coercion
$\mathit{coerce} :: \mathit{Eq}\ a\ b \to a \to b$
$\mathit{eqApp} :: \mathit{Eq}\ t_1\ t_2 \to \mathit{Eq}\ (f\ t_1)\ (f\ t_2)$

-- Injectivity of $(\to)$
$\mathit{arrL} :: \mathit{Eq}\ (t_1 \to t_2)\ (s1 \to s2) \to \mathit{Eq}\ t_1\ s1$
$\mathit{arrR} :: \mathit{Eq}\ (t_1 \to t_2)\ (s1 \to s2) \to \mathit{Eq}\ t_2\ s2$

Using explicit equalities

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Exp}\ t \rightarrow \mathit{Exp}\ t\\
&\mathit{eval}\ (\mathit{App}\ e_1\ e_2) = \textbf{case}\ \mathit{eval}\ e_1\ \textbf{of}\\
&\quad \mathit{Abs}\ \mathit{eq}\ f \rightarrow \textbf{let}\ \mathit{eqL} = \mathit{eqApp}\ (\mathit{sym}\ (\mathit{arrL}\ \mathit{eq}))\\
&\qquad\qquad\qquad\qquad \mathit{eqR} = \mathit{eqApp}\ (\mathit{arrR}\ \mathit{eq})\\
&\qquad\qquad\qquad\qquad f' \quad = \mathit{coerce}\ \mathit{eqR} \circ f \circ \mathit{coerce}\ \mathit{eqL}\\
&\qquad\qquad\qquad\quad \textbf{in}\ \mathit{eval}\ (f'\ e_2)\\
&\quad e_1' \qquad\quad \rightarrow \mathit{App}\ e_1'\ e_2\\
&\mathit{eval}\ e = e
\end{aligned}
$$

Getting rid of some sugar – recursive types

We use standard iso-recursive μ types to encode recursive types:

$$
\begin{array}{l}
\textbf{newtype}\ \mu\ f\ a = \mathit{Fold}\ \{\mathit{unFold} :: f\ (\mu\ f)\ a\} \\
\textbf{data}\ \mathit{ExpF}\ f\ t \\
\quad = \forall t_1\ t_2.\mathit{Abs}\ (\mathit{Eq}\ (t_1 \to t_2)\ t)\ (f\ t_1 \to f\ t_2) \\
\quad \mid\ \forall t_0.\quad \mathit{App}\qquad (f\ (t_0 \to t))\ (f\ t_0) \\
\textbf{type}\ \mathit{Exp} = \mu\ \mathit{ExpF}
\end{array}
$$

Getting rid of some sugar – datatypes, existentials

Standard Scott encoding of constructors: representation based on deconstructors instead

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{ExpF}\ f\ t = \mathit{MkExpF}\ \{\mathit{unExpF} :: \\
\quad \forall r. \\
\quad\ \ \{\text{-Abs -}\}\ \ (\forall t_1\ t_2.\mathit{Eq}\ (t_1 \to t_2)\ t \to (f\ t_1 \to f\ t_2) \qquad \to r) \to \\
\quad\ \ \{\text{-App -}\}\ \ (\forall t_0. \qquad\qquad\qquad\qquad f\ (t_0 \to t) \to f\ t_0 \to r) \to \\
\quad r\} \\
\textbf{type}\ \mathit{Exp} = \mu\ \mathit{ExpF}
\end{array}
$$

This also transforms existentials into rank-2 universals.

Constructors and matching

Constructors are recovered by using the right deconstructors:

$$
\begin{aligned}
&\mathit{app} :: \mathit{Exp}\ (t_0 \to t) \to \mathit{Exp}\ t_0 \to \mathit{Exp}\ t \\
&\mathit{app}\ e_1\ e_2 = \mathit{Fold}\ \$\ \mathit{MkExpF}\ \$\ \lambda\_\ \mathit{app} \to \mathit{app}\ e_1\ e_2 \\
&\mathit{abs} :: (\mathit{Exp}\ t_1 \to \mathit{Exp}\ t_2) \to \mathit{Exp}\ (t_1 \to t_2) \\
&\mathit{abs}\ f\ = \mathit{Fold}\ \$\ \mathit{MkExpF}\ \$\ \lambda \mathit{abs}\ \_ \to \mathit{abs}\ \mathit{refl}\ f
\end{aligned}
$$

Matching is just applying as deconstructors modulo μ plumbing:

$$
\begin{aligned}
\mathit{matchExp}\ &::\ \mathit{Exp}\ t \\
&\to (\forall t_0 . \mathit{Exp}\ (t_0 \to t) \to \mathit{Exp}\ t_0 \to r) \\
&\to (\forall t_1\ t_2 . \mathit{Eq}\ (t_1 \to t_2)\ t \to (\mathit{Exp}\ t_1 \to \mathit{Exp}\ t_2) \to r) \\
&\to r \\
\mathit{matchExp}\ &e\ \mathit{abs}\ \mathit{app} = \mathit{unExpF}\ (\mathit{unFold}\ e)\ \mathit{abs}\ \mathit{app}
\end{aligned}
$$

Getting rid of some sugar – Putting it all together

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Exp}\ t \rightarrow \mathit{Exp}\ t \\
\mathit{eval}\ e = \mathit{matchExp}\ e \\
\quad (\lambda e_1\ e_2 \rightarrow \mathbf{let}\ e_1' = \mathit{eval}\ e_1 \\
\qquad \mathbf{in}\ \mathit{matchExp}\ e_1' \\
\qquad\quad (\lambda\_\ \_ \rightarrow \mathit{app}\ e_1'\ e_2) \\
\qquad\quad (\lambda \mathit{eq}\ f \rightarrow \mathbf{let}\ \mathit{eqL} = \mathit{eqApp}\ \$\ \mathit{sym}\ \$\ \mathit{arrL}\ \mathit{eq} \\
\qquad\qquad \mathit{eqR} = \mathit{eqApp}\ \$\ \mathit{arrR}\ \mathit{eq} \\
\qquad\qquad f' = \mathit{coerce}\ \mathit{eqR} \circ f \circ \mathit{coerce}\ \mathit{eqL} \\
\qquad\qquad \mathbf{in}\ \mathit{eval}\ (f'\ e_2))) \\
\quad (\lambda\_\ \_ \rightarrow e)
\end{array}
$$

Host vs. target language

—— Haskell w/GADTs

.... Totality

—— STLC

Host vs. target language

Haskell w/GADTs

"Mini-Haskell", μ, rank-2, equalities

Totality

STLC

Towards $\mathrm{F}^{\mu i}_{\omega}$

About those pesky equalities...

We want to move further away from Haskell, but it turns out *Eq* still brings in a ton of Haskell baggage:

- *Eq* is still a GADT
- Its theory is implemented using GADT pattern matching

Can we make do with just F_ω?
Let's use Leibniz equality instead:

$$\begin{aligned}
&Eq : \star \to \star \to \star \\
&Eq \mathbin{@} \alpha \mathbin{@} \beta = \forall F : \star \to \star . F\ \alpha \to F\ \beta
\end{aligned}$$

Eq's theory reconstructed

$$\mathit{coerce} : \forall\alpha : *.\forall\beta : *.\mathit{Eq}\ \alpha\ \beta \to \alpha \to \beta$$
$$\mathit{coerce}\ @\ \alpha\ @\ \beta = \lambda \mathit{eq} : \mathit{Eq}\ \alpha\ \beta.\ \mathit{eq}\ \mathit{Id}$$

$$\mathit{refl} :: \forall\alpha : \star.\mathit{Eq}\ \alpha\ \alpha$$
$$\mathit{refl}\ @\ \alpha = \Lambda F : \star \to \star.\lambda x : F\ \alpha.\ x$$

sym, *trans*, *eqApp* similarly

arrL and *arrR* are intensional!

The problem with defining *arrL* and *arrR* internally is that they require inspection of a function type to be able to decompose it. Clearly, this needs external support.
We add to our host language a type-level type inspection facility for this: **Typecase** with a non-parametric type equivalence:

$$\begin{array}{l}\textbf{Typecase} : (\star \to \star \to \star) \to \star \to \star \\ \textbf{Typecase}\ \mathit{Arr}\ (t_1 \to t_2) \equiv \mathit{Arr}\ t_1\ t_2\end{array}$$

This enables writing *arrL* in terms of *eqApp* (and *arrR* similarly):

$$\begin{array}{l}\mathit{arrL} : \forall \alpha_1\ \alpha_2\ \beta_1\ \beta_2 : \star. \mathit{Eq}\ (\alpha_1 \to \alpha_2)\ (\beta_1 \to \beta_2) \to \mathit{Eq}\ \alpha_1\ \beta_1 \\ \mathit{arrL} \mathbin{@} \alpha_1 \mathbin{@} \alpha_2 \mathbin{@} \beta_1 \mathbin{@} \beta_2 = \mathit{eqApp} \mathbin{@}(\alpha_1 \to \alpha_2) \mathbin{@}(\beta_1 \to \beta_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathbin{@}(\textbf{Typecase}\ (\lambda\alpha : \star.\, \lambda\beta : \star.\, \alpha))\end{array}$$

Iso-recursive μ types

Let's just add explicit support for μ types to our host language. It is enough to do it for unary type constructors over $\star$, i.e. the fixed points will all be of kind $\star \to \star$:

$$\mu : ((\star \to \star) \to \star \to \star) \to \star \to \star$$

$$\frac{\Gamma \vdash F : (\star \to \star) \to \star \to \star \qquad \Gamma \vdash \tau : \star \qquad \Gamma \vdash e : F\ (\mu\ F)\ \tau}{\Gamma \vdash \mathbf{fold}\ {\scriptstyle @}\, F\ {\scriptstyle @}\, \tau\ e : \mu\ F\ \tau}$$

$$\frac{\Gamma \vdash F : (\star \to \star) \to \star \to \star \qquad \Gamma \vdash \tau : \star \qquad \Gamma \vdash e : \mu\ F\ \tau}{\Gamma \vdash \mathbf{unfold}\ {\scriptstyle @}\, F\ {\scriptstyle @}\, \tau\ e : F\ (\mu\ F)\ \tau}$$

Value-level recursion

Traversing μ *Exp* τ involved recursion (*eval* was a recursive definition). Do we need to add recursion as a further primitive to our host language?

Value-level recursion

Traversing $\mu\ \mathit{Exp}\ \tau$ involved recursion (*eval* was a recursive definition). Do we need to add recursion as a further primitive to our host language?
No, because μ is enough:

$$R : (\star \to \star) \to \star \to \star$$
$$R\ f\ \alpha = f\ \alpha \to \alpha$$

$$\mathit{diag} : \forall \alpha : \star.(\mu\ R\ \alpha \to \alpha) \to \alpha$$
$$\mathit{diag}\ @\ \alpha = \lambda f : (\mu\ R\ \alpha \to \alpha).f\ (\mathbf{fold}\ f)$$

$$\mathit{fix} : \forall \alpha : \star.(\alpha \to \alpha) \to \alpha$$
$$\mathit{fix}\ @\ \alpha = \lambda f : (\alpha \to \alpha).\mathit{diag}\ @\ \alpha\ (f \circ \mathit{diag}\ @\ \alpha \circ \mathbf{unfold}\ @\ R\ @\ \alpha)$$

System $F_\omega^{\mu i}$

$$
\begin{array}{lcl}
\langle \text{kind } \kappa \rangle & \models & \star \mid \kappa_1 \rightarrow \kappa_2 \\
\langle \text{type } \tau \rangle & \models & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha : \kappa . \tau \mid \lambda \alpha : \kappa . \tau \mid \tau_1 \ \tau_2 \mid \\
 & & \mu \mid \textbf{Typecase} \\
\langle \text{term } e \rangle & \models & x \mid \lambda x : \tau . e \mid e_1 \ e_2 \mid \Lambda \alpha : \kappa . e \mid e \mathbin{@} \tau \mid \\
 & & \textbf{fold } \tau_1 \ \tau_2 \ e \mid \textbf{unfold } \tau_1 \ \tau_2 \ e
\end{array}
$$

- Parametric polymorphism (note: at any rank!)
- Type constructors, type transformers, ...
- Iso-recursive types
- Intensional typecase, but only on type level (no RTTI)

STLC in $F_\omega^{\mu i}$

No surprises here:

$$
\begin{array}{l}
\mathit{ExpF} : (\star \to \star) \to \star \to \star \\
\mathit{ExpF}\ F\ t = \\
\quad \forall r : \star. \\
\quad\ \ \text{\{-Abs -\}}\ \ (\forall t_1\ t_2 : \star. \mathit{Eq}\ (t_1 \to t_2)\ t \to (F\ t_1 \to F\ t_2) \to \qquad r) \to \\
\quad\ \ \text{\{-App -\}}\ \ (\forall t_0 : \star. \qquad\qquad\qquad\qquad\qquad F\ (t_0 \to t) \to F\ t \to r) \to \\
\quad r
\end{array}
$$

$$
\begin{array}{l}
\mathit{Exp} : \star \to \star \\
\mathit{Exp} = \mu\ \mathit{ExpF}
\end{array}
$$

STLC in $F_\omega^{\mu i}$ with meta-variables

We'll switch to a PHOAS representation, useful for some intensional processing:

$$
\begin{array}{l}
PExpF : (\star \to \star) \to (\star \to \star) \to \star \to \star \\
PExpF\ V\ F\ t = \\
\quad \forall r : \star. \\
\quad\quad \text{\{-Var -\}}\ \ (V\ \alpha \to \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r) \to \\
\quad\quad \text{\{-Abs -\}}\ \ (\forall t_1\ t_2 : \star. Eq\ (t_1 \to t_2)\ t \to (F\ t_1 \to F\ t_2) \to \quad r) \to \\
\quad\quad \text{\{-App -\}}\ \ (\forall t_0 : \star. \qquad\qquad\qquad\qquad F\ (t_0 \to t) \to F\ t \to r) \to \\
\quad r
\end{array}
$$

$$
\begin{array}{l}
PExp : (\star \to \star) \to \star \to \star \\
PExp\ V = \mu\ (PExpF\ V)
\end{array}
$$

-- A (meta-)closed expression

$$
\begin{array}{l}
Exp : \star \to \star \\
Exp\ \alpha = \forall V : \star \to \star. PExp\ V\ \alpha
\end{array}
$$

Host vs. target language

Haskell w/GADTs

"Mini-Haskell", μ, rank-2, equalities

Totality

STLC

Host vs. target language

Haskell w/GADTs

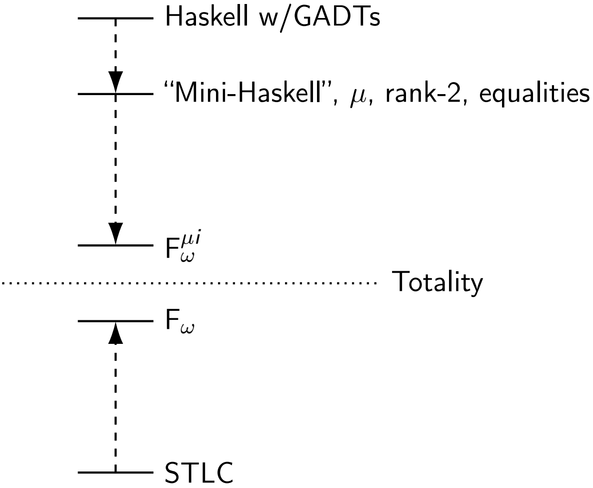
"Mini-Haskell", μ, rank-2, equalities

$\mathsf{F}_{\omega}^{\mu i}$

Totality

STLC

Host vs. target language



$$
\begin{array}{l}
PExpF : (\star \to \star) \to (\star \to \star) \to \star \to \star \\
PExpF\ V\ F\ t = \\
\quad \forall r : \star. \\
\quad\ \text{\{-Var, Abs, App omitted -\}} \\
\quad\ \text{\{-TAbs -\}}\ \ (IsAll\ t \to \qquad\qquad\qquad\qquad \to Push\ F\ t \to r) \to \\
\quad\ \text{\{-TApp -\}}\ \ (\forall t_0 : \star. IsAll\ t_0 \to Inst\ t_0\ t \to F\ t_0 \qquad \to r) \to \\
\quad r
\end{array}
$$

F_ω in $\mathsf{F}_\omega^{\mu i}$

$$
\begin{array}{l}
\mathit{PExpF} : (\star \to \star) \to (\star \to \star) \to \star \to \star \\
\mathit{PExpF}\ V\ F\ t = \\
\quad \forall r : \star. \\
\quad\ \ \text{\{-Var, Abs, App omitted -\}} \\
\quad\ \ \text{\{-TAbs -\}}\ \ (\mathit{IsAll}\ t \to \qquad\qquad\qquad\qquad\qquad\quad \to \mathit{Push}\ F\ t \to r) \to \\
\quad\ \ \text{\{-TApp -\}}\ \ (\forall t_0 : \star. \mathit{IsAll}\ t_0 \to \mathit{Inst}\ t_0\ t \to F\ t_0 \qquad\ \ \to r) \to \\
\quad r
\end{array}
$$

- *IsAll* and *Inst* will need to be clever witnesses of polymorphism

$$
\begin{array}{l}
\mathit{PExpF} : (\star \to \star) \to (\star \to \star) \to \star \to \star \\
\mathit{PExpF}\ V\ F\ t = \\
\quad \forall r : \star. \\
\quad\ \ \text{\{-Var, Abs, App omitted -\}} \\
\quad\ \ \text{\{-TAbs -\}}\ \ (\mathit{IsAll}\ t \to \qquad\qquad\qquad\qquad\quad \to \mathit{Push}\ F\ t \to r) \to \\
\quad\ \ \text{\{-TApp -\}}\ \ (\forall t_0 : \star. \mathit{IsAll}\ t_0 \to \mathit{Inst}\ t_0\ t \to F\ t_0 \qquad\quad \to r) \to \\
\quad r
\end{array}
$$

- *IsAll* and *Inst* will need to be clever witnesses of polymorphism
- *Push* f t needs to be $\forall \alpha : \kappa. f\ t$ for the right α and κ

F_ω in $\mathsf{F}_\omega^{\mu i}$: More **Typecase**

IsAll, *Inst* and *Push* can all be implemented if we extend **Typecase** to handle polymorphic types:

$$
\begin{array}{ll}
\textbf{Typecase} : (\star \to \star \to \star) \to (\star \to \star) \to (\star \to \star) \to \star \to \star & \\
\textbf{Typecase}\ \mathit{Arr}\ \mathit{Out}\ \mathit{In}\ (t_1 \to t_2) \equiv \mathit{Arr}\ t_1\ t_2 & \\
\textbf{Typecase}\ \mathit{Arr}\ \mathit{Out}\ \mathit{In}\ (\forall \alpha : \kappa . t) \equiv \mathit{Out}\ (\forall \alpha : \kappa . \mathit{In}\ t) & \alpha \notin \mathrm{FV}(\mathit{In})
\end{array}
$$

F_ω in $\mathsf{F}_\omega^{\mu i}$: More **Typecase**

IsAll, *Inst* and *Push* can all be implemented if we extend **Typecase** to handle polymorphic types:

$$\begin{array}{ll}
\textbf{Typecase} : (\star \to \star \to \star) \to (\star \to \star) \to (\star \to \star) \to \star \to \star & \\
\textbf{Typecase}\ \mathit{Arr}\ \mathit{Out}\ \mathit{In}\ (t_1 \to t_2) \equiv \mathit{Arr}\ t_1\ t_2 & \\
\textbf{Typecase}\ \mathit{Arr}\ \mathit{Out}\ \mathit{In}\ (\forall \alpha : \kappa . t) \equiv \mathit{Out}\ (\forall \alpha : \kappa . \mathit{In}\ t) & \alpha \notin \mathrm{FV}(\mathit{In})
\end{array}$$

We can also define *StripAll* and *UnderAll* and add them to *TAbs* to enable traversals where the result is a constant type, e.g. $\mathit{size} : \mathbb{N} \to \mathbb{N}$, i.e. when we need to go between $\forall \alpha : \kappa . \mathbb{N}$ and $\mathbb{N}$.

Self-reduction

Typecase for μ types

$$
\begin{aligned}
\textbf{Typecase} : {} & (\star \to \star \to \star) && \to \\
& (\star \to \star) && \to \\
& (\star \to \star) && \to \\
& (((\star \to \star) \to \star \to \star) \to \star \to \star) && \to \\
& \star && \to \\
& \star
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{Typecase}\ \mathit{Arr}\ \mathit{Out}\ \mathit{In}\ \mathit{Fix}\ (t_1 \to t_2) &&\equiv \mathit{Arr}\ t_1\ t_2 \\
&\textbf{Typecase}\ \mathit{Arr}\ \mathit{Out}\ \mathit{In}\ \mathit{Fix}\ (\forall\alpha:\kappa.t) &&\equiv \mathit{Out}\ (\forall\alpha:\kappa.\mathit{In}\ t) \qquad \alpha \notin \mathrm{FV}(\\
&\textbf{Typecase}\ \mathit{Arr}\ \mathit{Out}\ \mathit{In}\ \mathit{Fix}\ (\mu\ f\ t) &&\equiv \mathit{Fix}\ f\ t
\end{aligned}
$$

Tying the knot: $F_\omega^{\mu i}$ in $F_\omega^{\mu i}$

$$
\begin{array}{l}
PExpF : (\star \to \star) \to (\star \to \star) \to \star \to \star \\
PExpF\ V\ F\ t = \\
\quad \forall r : \star. \\
\quad\ \ \text{\{-Var, Abs, App, TAbs, TApp omitted -\}} \\
\quad\ \ \text{\{-Fold -\}} \quad (\forall g\ t_0. Eq\ (\mu\ g\ t_0)\ t \to F\ (g\ (\mu\ g)\ t_0) \to r) \to \\
\quad\ \ \text{\{-Unfold -\}}\ (\forall g\ t_0. Eq\ (g\ (\mu\ g)\ t_0)\ t \to F\ (\mu\ g\ t_0) \to r) \to \\
\quad r
\end{array}
$$

Host vs. target language

Haskell w/GADTs

"Mini-Haskell", μ, rank-2, equalities

$\mathsf{F}_\omega^{\mu i}$

Totality

F_ω

STLC

Host vs. target language