

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

Gergő Érdi

<http://gergo.erd.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 $[\cdot]$ maps some closed term $e : \tau$ to another, normal-form term $[e] : \text{Exp } \tau$.

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:

$$\text{unquote } [e] \longrightarrow_{\beta}^* e$$

- ▶ *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

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

then

$$\text{reduce } [e] \longrightarrow_{\beta}^* [v]$$

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 ($\text{Exp } _ = \text{String}$), we have

$$\begin{aligned} \text{unquote}(\text{"3 + 5"}) &\longrightarrow^* 3 + 5 \\ \text{reduce}(\text{"3 + 5"}) &\longrightarrow^* \text{"8"} \end{aligned}$$

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

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

The simply typed lambda calculus

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

$const : A \rightarrow B \rightarrow A$

$const = \lambda x : A. \lambda y : B. x$

Selected lambda calculi – F

$\langle \text{kind } \kappa \rangle \models \star$

$\langle \text{type } \tau \rangle \models \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha : \kappa. \tau$

$\langle \text{term } e \rangle \models x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda \alpha : \kappa. e \mid e @ \tau$

System F

- ▶ Strongly normalizing
- ▶ Parametric polymorphism (note: at any rank!)
- ▶ “Atomic” types
- ▶ Self-unquoter: see authors’ paper from POPL 2016

$const : \forall \alpha : \star. (\alpha \rightarrow \forall \beta : \star. (\beta \rightarrow \alpha))$

$const = \Lambda \alpha : \star. \lambda x : \alpha. \Lambda \beta : \star. \lambda y : \beta. x$

Selected lambda calculi – F_ω

$\langle \text{kind } \kappa \rangle \models \star \mid \kappa_1 \rightarrow \kappa_2$

$\langle \text{type } \tau \rangle \models \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha : \kappa. \tau \mid \lambda \alpha : \kappa. \tau \mid \tau_1 \tau_2$

$\langle \text{term } e \rangle \models x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda \alpha : \kappa. e \mid e @ \tau$

System F_ω

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

$const : \forall \alpha : \star. (\alpha \rightarrow \forall \beta : \star. (\beta \rightarrow \alpha))$

$const = \Lambda \alpha : \star. \lambda x : \alpha. \Lambda \beta : \star. \lambda y : \beta. x$

Typed evaluation of STLC in Haskell

STLC evaluator in Haskell – explicit variables

```
data Ctx a = O | Ctx a :> a
```

```
data Var ctx t where
```

```
  VZ :: Var (ts :> t) t
```

```
  VS :: Var ts t → Var (ts :> t0) t
```

```
data Exp ctx t where
```

```
  Var :: Var ctx t → Exp ctx t
```

```
  App :: Exp ctx (t0 → t) → Exp ctx t0 → Exp ctx t
```

```
  Abs :: Exp (ctx :> t1) t2 → Exp ctx (t1 → t2)
```

STLC evaluator in Haskell – explicit variables

```
data Ctx a = O | Ctx a :> a
data Var ctx t where
  VZ :: Var (ts :> t) t
  VS :: Var ts t → Var (ts :> t0) t
data Exp ctx t where
  Var :: Var ctx t → Exp ctx t
  App :: Exp ctx (t0 → t) → Exp ctx t0 → Exp ctx t
  Abs :: Exp (ctx :> t1) t2 → Exp ctx (t1 → t2)
```

But this requires capture-avoiding substitution. . .

STLC evaluator in Haskell – HOAS

HOAS approach to writing a typed STLC evaluator:

data *Exp* *t* **where**

Abs :: (*Exp* *t*₁ → *Exp* *t*₂) → *Exp* (*t*₁ → *t*₂)

App :: *Exp* (*t*₀ → *t*) → *Exp* *t*₀ → *Exp* *t*

eval :: *Exp* *t* → *Exp* *t*

eval (*App* *e*₁ *e*₂) = **case** *eval* *e*₁ **of**

Abs *f* → *eval* (*f* *e*₂)

e'₁ → *App* *e*'₁ *e*₂

eval *e* = *e*

STLC evaluator in Haskell – HOAS

HOAS approach to writing a typed STLC evaluator:

```
{-# LANGUAGE GADTs #-}
data Exp t where
  Abs :: (Exp t1 -> Exp t2) -> Exp (t1 -> t2)
  App :: Exp (t0 -> t) -> Exp t0 -> Exp t
eval :: Exp t -> Exp t
eval (App e1 e2) = case eval e1 of
  Abs f -> eval (f e2)
  e1' -> App e1' e2
eval e = e
```

Actually, Haskell + GADTs

STLC evaluator in Haskell – HOAS

HOAS approach to writing a typed STLC evaluator:

```
{-# LANGUAGE GADTs #-}
data Exp t where
  Abs :: (Exp t1 → Exp t2) → Exp (t1 → t2)
  App :: Exp (t0 → t) → Exp t0 → Exp t
eval :: Exp t → Exp t
eval (App e1 e2) = case eval e1 of
  Abs f → eval (f e2)
  e'1 → App e'1 e2
eval e = e
```

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:

```
data Exp t
  =  $\forall t_1 t_2.$  Abs (Eq ( $t_1 \rightarrow t_2$ ) t) (Exp t1  $\rightarrow$  Exp t2)
  |  $\forall t_0.$  App (Exp (t0  $\rightarrow$  t)) (Exp t0)
```

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

refl :: *Eq a a*

sym :: *Eq a b* → *Eq b a*

trans :: *Eq a b* → *Eq b c* → *Eq a c*

-- It is a congruence over type constructors, allowing coercion

coerce :: *Eq a b* → *a* → *b*

eqApp :: *Eq t₁ t₂* → *Eq (f t₁) (f t₂)*

-- Injectivity of (→)

arrL :: *Eq (t₁ → t₂) (s₁ → s₂)* → *Eq t₁ s₁*

arrR :: *Eq (t₁ → t₂) (s₁ → s₂)* → *Eq t₂ s₂*

Using explicit equalities

```
eval :: Exp t → Exp t  
eval (App e1 e2) = case eval e1 of  
  Abs eq f → let eqL = eqApp (sym (arrL eq))  
                 eqR = eqApp (arrR eq)  
                 f'   = coerce eqR ∘ f ∘ coerce eqL  
                 in eval (f' e2)  
  e'1       → App e'1 e2  
eval e = e
```

Getting rid of some sugar – recursive types

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

newtype $\mu f a = \text{Fold } \{ \text{unFold} :: f (\mu f) a \}$

data $\text{ExpF } f t$

$= \forall t_1 t_2. \text{Abs } (\text{Eq } (t_1 \rightarrow t_2) t) (f t_1 \rightarrow f t_2)$
 $| \forall t_0. \text{App } (f (t_0 \rightarrow t)) (f t_0)$

type $\text{Exp} = \mu \text{ExpF}$

Getting rid of some sugar – datatypes, existentials

Standard Scott encoding of constructors: representation based on deconstructors instead

```
newtype ExpF f t = MkExpF { unExpF ::  
  ∀ r.  
    {-Abs -} (∀ t1 t2. Eq (t1 → t2) t → (f t1 → f t2) → r) →  
    {-App -} (∀ t0. f (t0 → t) → f t0 → r) →  
    r }  
type Exp = μ ExpF
```

This also transforms existentials into rank-2 universals.

Constructors and matching

Constructors are recovered by using the right deconstructors:

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

Matching is just applying as deconstructors modulo μ plumbing:

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

Getting rid of some sugar – Putting it all together

```
eval :: Exp t → Exp t  
eval e = matchExp e  
  (λ e1 e2 → let e'1 = eval e1  
    in matchExp e'1  
      (λ _ _ → app e'1 e2)  
      (λ eq f → let eqL = eqApp $ sym $ arrL eq  
        eqR = eqApp $ arrR eq  
        f' = coerce eqR ∘ f ∘ coerce eqL  
        in eval (f' e2)))  
  (λ _ _ → e)
```

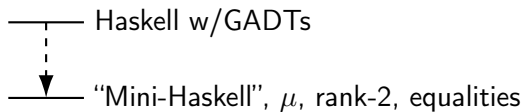
Host vs. target language

—— Haskell w/GADTs

..... Totality

—— STLC

Host vs. target language



..... Totality

— STLC

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

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:

$$Eq : \star \rightarrow \star \rightarrow \star$$

$$Eq \circ \alpha \circ \beta = \forall F : \star \rightarrow \star. F \alpha \rightarrow F \beta$$

Eq's theory reconstructed

coerce : $\forall \alpha : *. \forall \beta : *. Eq \alpha \beta \rightarrow \alpha \rightarrow \beta$

coerce \circ α \circ $\beta = \lambda eq : Eq \alpha \beta. eq \text{Id}$

refl :: $\forall \alpha : *. Eq \alpha \alpha$

refl \circ $\alpha = \Lambda F : * \rightarrow *. \lambda x : F \alpha. x$

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:

$$\begin{aligned} \mathbf{Typecase} &: (\star \rightarrow \star \rightarrow \star) \rightarrow \star \rightarrow \star \\ \mathbf{Typecase} \text{ Arr } (t_1 \rightarrow t_2) &\equiv \text{Arr } t_1 \ t_2 \end{aligned}$$

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

$$\begin{aligned} \mathit{arrL} &: \forall \alpha_1 \ \alpha_2 \ \beta_1 \ \beta_2 : \star. \mathit{Eq} (\alpha_1 \rightarrow \alpha_2) (\beta_1 \rightarrow \beta_2) \rightarrow \mathit{Eq} \ \alpha_1 \ \beta_1 \\ \mathit{arrL} \circledast \alpha_1 \circledast \alpha_2 \circledast \beta_1 \circledast \beta_2 &= \mathit{eqApp} \circledast (\alpha_1 \rightarrow \alpha_2) \circledast (\beta_1 \rightarrow \beta_2) \\ &\quad \circledast (\mathbf{Typecase} (\lambda \alpha : \star. \lambda \beta : \star. \alpha)) \end{aligned}$$

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 $\star \rightarrow \star$:

$$\mu : ((\star \rightarrow \star) \rightarrow \star \rightarrow \star) \rightarrow \star \rightarrow \star$$

$$\frac{\Gamma \vdash F : (\star \rightarrow \star) \rightarrow \star \rightarrow \star \quad \Gamma \vdash \tau : \star \quad \Gamma \vdash e : F (\mu F) \tau}{\Gamma \vdash \mathbf{fold}_{\circlearrowleft F \circlearrowleft \tau} e : \mu F \tau}$$

$$\frac{\Gamma \vdash F : (\star \rightarrow \star) \rightarrow \star \rightarrow \star \quad \Gamma \vdash \tau : \star \quad \Gamma \vdash e : \mu F \tau}{\Gamma \vdash \mathbf{unfold}_{\circlearrowleft F \circlearrowleft \tau} e : F (\mu F) \tau}$$

Value-level recursion

Traversing $\mu \text{Exp } \tau$ 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 $\mu \text{Exp } \tau$ 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:

$$R : (\star \rightarrow \star) \rightarrow \star \rightarrow \star$$

$$R f \alpha = f \alpha \rightarrow \alpha$$

$$\text{diag} : \forall \alpha : \star. (\mu R \alpha \rightarrow \alpha) \rightarrow \alpha$$

$$\text{diag} \circ \alpha = \lambda f : (\mu R \alpha \rightarrow \alpha). f \text{ (fold } f)$$

$$\text{fix} : \forall \alpha : \star. (\alpha \rightarrow \alpha) \rightarrow \alpha$$

$$\text{fix} \circ \alpha = \lambda f : (\alpha \rightarrow \alpha). \text{diag} \circ \alpha (f \circ \text{diag} \circ \alpha \circ \text{unfold} \circ R \circ \alpha)$$

$\langle \text{kind } \kappa \rangle \models \star \mid \kappa_1 \rightarrow \kappa_2$

$\langle \text{type } \tau \rangle \models \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha : \kappa. \tau \mid \lambda \alpha : \kappa. \tau \mid \tau_1 \tau_2 \mid$
 $\mu \mid \mathbf{Typecase}$

$\langle \text{term } e \rangle \models x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda \alpha : \kappa. e \mid e \otimes \tau \mid$
 $\mathbf{fold} \tau_1 \tau_2 e \mid \mathbf{unfold} \tau_1 \tau_2 e$

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

No surprises here:

$$ExpF : (\star \rightarrow \star) \rightarrow \star \rightarrow \star$$

$$ExpF F t =$$

$$\forall r : \star.$$

$$\{-Abs -\} (\forall t_1 t_2 : \star. Eq (t_1 \rightarrow t_2) t \rightarrow (F t_1 \rightarrow F t_2) \rightarrow r) \rightarrow$$

$$\{-App -\} (\forall t_0 : \star. F (t_0 \rightarrow t) \rightarrow F t \rightarrow r) \rightarrow$$

$$r$$

$$Exp : \star \rightarrow \star$$

$$Exp = \mu ExpF$$

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

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

$$PExpF : (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$$

$$PExpF V F t =$$

$$\forall r : \star.$$

$$\{-Var -\} (V \alpha \rightarrow r) \rightarrow$$

$$\{-Abs -\} (\forall t_1 t_2 : \star. Eq (t_1 \rightarrow t_2) t \rightarrow (F t_1 \rightarrow F t_2) \rightarrow r) \rightarrow$$

$$\{-App -\} (\forall t_0 : \star. F (t_0 \rightarrow t) \rightarrow F t \rightarrow r) \rightarrow$$

r

$$PExp : (\star \rightarrow \star) \rightarrow \star \rightarrow \star$$

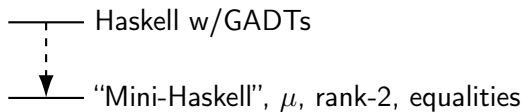
$$PExp V = \mu (PExpF V)$$

-- A (meta-)closed expression

$$Exp : \star \rightarrow \star$$

$$Exp \alpha = \forall V : \star \rightarrow \star. PExp V \alpha$$

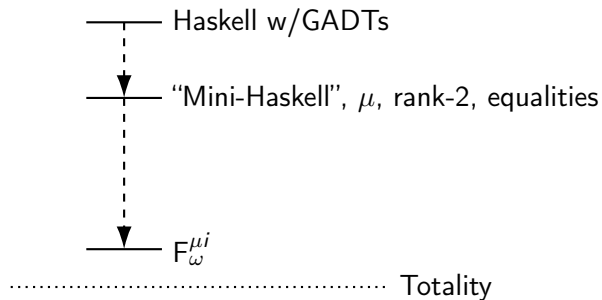
Host vs. target language



..... Totality

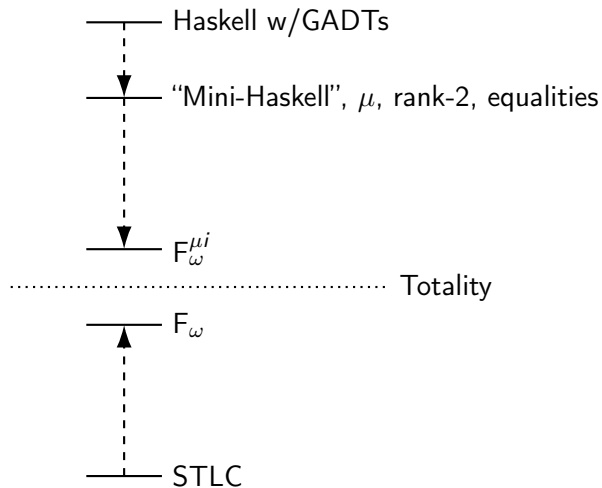
—— STLC

Host vs. target language



— STLC

Host vs. target language



$PExpF : (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$ $PExpF \ V \ F \ t =$ $\forall r : \star.$ $\{-Var, Abs, App \ omitted \ -\}$ $\{-TAbs \ -\} \ (IsAll \ t \rightarrow \hspace{15em} \rightarrow \ Push \ F \ t \rightarrow r) \rightarrow$ $\{-TApp \ -\} \ (\forall t_0 : \star. IsAll \ t_0 \rightarrow Inst \ t_0 \ t \rightarrow F \ t_0 \hspace{2em} \rightarrow r) \rightarrow$ r

$$PExpF : (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$$

$$PExpF \vee F t =$$

$$\forall r : \star.$$

$$\{-\text{Var, Abs, App omitted -}\}$$

$$\{-\text{TAbs -}\} (IsAll t \rightarrow \quad \quad \quad \rightarrow Push F t \rightarrow r) \rightarrow$$

$$\{-\text{TApp -}\} (\forall t_0 : \star. IsAll t_0 \rightarrow Inst t_0 t \rightarrow F t_0 \quad \rightarrow r) \rightarrow$$

$$r$$

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

$$PExpF : (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$$

$$PExpF \vee F t =$$

$$\forall r : \star.$$

$$\{-\text{Var, Abs, App omitted -}\}$$

$$\{-\text{TAbs -}\} (IsAll t \rightarrow \quad \quad \quad \rightarrow Push F t \rightarrow r) \rightarrow$$

$$\{-\text{TApp -}\} (\forall t_0 : \star. IsAll t_0 \rightarrow Inst t_0 t \rightarrow F t_0 \quad \rightarrow r) \rightarrow$$

$$r$$

- ▶ *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 $F_\omega^{\mu i}$: More **Typecase**

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

Typecase : $(\star \rightarrow \star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$

Typecase Arr Out In $(t_1 \rightarrow t_2) \equiv \text{Arr } t_1 \ t_2$

Typecase Arr Out In $(\forall \alpha : \kappa. t) \equiv \text{Out } (\forall \alpha : \kappa. \text{In } t) \quad \alpha \notin \text{FV}(\text{In})$

F_ω in $F_\omega^{\mu i}$: More **Typecase**

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

Typecase : $(\star \rightarrow \star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$

Typecase *Arr Out In* $(t_1 \rightarrow t_2) \equiv \text{Arr } t_1 \ t_2$

Typecase *Arr Out In* $(\forall \alpha : \kappa. t) \equiv \text{Out } (\forall \alpha : \kappa. \text{In } t) \quad \alpha \notin \text{FV}(\text{In})$

We can also define *StripAll* and *UnderAll* and add them to *TABs* to enable traversals where the result is a constant type, e.g.

size : $\mathbb{N} \rightarrow \mathbb{N}$, i.e. when we need to go between $\forall \alpha : \kappa. \mathbb{N}$ and \mathbb{N} .

Self-reduction

Typecase for μ types

Typecase : $(\star \rightarrow \star \rightarrow \star) \rightarrow$
 $(\star \rightarrow \star) \rightarrow$
 $(\star \rightarrow \star) \rightarrow$
 $((\star \rightarrow \star) \rightarrow \star \rightarrow \star) \rightarrow \star \rightarrow \star) \rightarrow$
 $\star \rightarrow$
 \star

Typecase *Arr Out In Fix* $(t_1 \rightarrow t_2) \equiv \text{Arr } t_1 t_2$

Typecase *Arr Out In Fix* $(\forall \alpha : \kappa. t) \equiv \text{Out } (\forall \alpha : \kappa. \text{In } t) \quad \alpha \notin \text{FV}(t)$

Typecase *Arr Out In Fix* $(\mu f t) \equiv \text{Fix } f t$

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

$PExpF : (\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow \star \rightarrow \star$

$PExpF \ V \ F \ t =$

$\forall r : \star.$

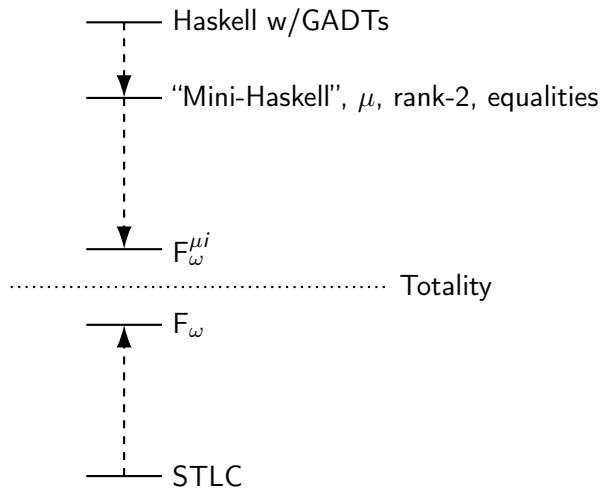
{-Var, Abs, App, TAbs, TApp omitted -}

{-Fold -} $(\forall g \ t_0. Eq (\mu \ g \ t_0) \ t \rightarrow F (g (\mu \ g) \ t_0) \rightarrow r) \rightarrow$

{-Unfold -} $(\forall g \ t_0. Eq (g (\mu \ g) \ t_0) \ t \rightarrow F (\mu \ g \ t_0) \rightarrow r) \rightarrow$

r

Host vs. target language



Host vs. target language

