

Embedding Polymorphic Dynamic Typing

Thomas van Noort

joint work with

Wouter Swierstra Peter Achten Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen

Workshop on Generic Programming
September 18, 2011

Introduction

Dynamic typing in a statically typed FL such as Clean/Haskell:

- ▶ Typically useful when interacting with the 'outside' world
 - ▶ Exchanging values between applications by (de)serialisation
 - ▶ Receiving input provided by a user
 - ▶ Obtaining values from a network connection
 - ▶ ...
- ▶ In such situations, types are not known at compile time

Introduction

Dynamic typing in a statically typed FL such as Clean/Haskell:

- ▶ Typically useful when interacting with the 'outside' world
 - ▶ Exchanging values between applications by (de)serialisation
 - ▶ Receiving input provided by a user
 - ▶ Obtaining values from a network connection
 - ▶ ...
- ▶ In such situations, types are not known at compile time

Dynamic typing defers type unification until run time:

- ▶ Values and type representations are wrapped in a black box
- ▶ Values are unwrapped by providing an expected type
- ▶ Fortunately, static type safety is not compromised

Dynamic typing in Haskell

Haskell offers dynamic typing via libraries:

- ▶ Only monomorphism is supported

Dynamic typing in Haskell

Haskell offers dynamic typing via libraries:

- ▶ Only monomorphism is supported

Values are wrapped using the library function *toDyn*:

idDyn :: *Dynamic*

idDyn = *toDyn* (($\lambda x \rightarrow x$) :: *Int* → *Int*)

Dynamic typing in Haskell

Haskell offers dynamic typing via libraries:

- ▶ Only monomorphism is supported

Values are wrapped using the library function *toDyn*:

```
idDyn :: Dynamic
```

```
idDyn = toDyn (( $\lambda x \rightarrow x$ ) :: Int  $\rightarrow$  Int)
```

The library function *fromDyn* unwraps a value:

```
idInt :: Maybe (Int  $\rightarrow$  Int)
```

```
idInt = fromDyn idDyn
```

Dynamic typing in Clean

Clean has built-in support for dynamic typing:

- ▶ Polymorphism is supported, amongst other type concepts

Dynamic typing in Clean

Clean has built-in support for dynamic typing:

- ▶ Polymorphism is supported, amongst other type concepts

Values are wrapped using the keyword **dynamic**:

idDyn :: *Dynamic*

idDyn = **dynamic** ($\lambda x \rightarrow x$)

Dynamic typing in Clean

Clean has built-in support for dynamic typing:

- ▶ Polymorphism is supported, amongst other type concepts

Values are wrapped using the keyword **dynamic**:

```
idDyn :: Dynamic
```

```
idDyn = dynamic ( $\lambda x \rightarrow x$ )
```

Values are unwrapped using the `::` annotation in a pattern match:

```
id :: Maybe (A.a : a → a)
```

```
id = case idDyn of
```

```
    (f :: A.a : a → a) → Just f
```

```
    –                     → Nothing
```

Dynamic typing in Clean

Clean has built-in support for dynamic typing:

- ▶ Polymorphism is supported, amongst other type concepts

Values are wrapped using the keyword **dynamic**:

```
idDyn :: Dynamic
```

```
idDyn = dynamic ( $\lambda x \rightarrow x$ )
```

Values are unwrapped using the `::` annotation in a pattern match:

```
id :: Maybe (A.a : a → a)
```

```
id = case idDyn of
```

```
    (f :: A.a : a → a) → Just f  
    –                    → Nothing
```

```
idInt :: Maybe (Int → Int)
```

```
idInt = case idDyn of
```

```
    (f :: Int → Int) → Just f  
    –                    → Nothing
```

Question

Interesting difference between the Clean and Haskell approach:

- ▶ Clean supports polymorphism, using a built-in system
 - ▶ Freedom in the implementation due to abstract syntax trees

Question

Interesting difference between the Clean and Haskell approach:

- ▶ Clean supports polymorphism, using a built-in system
 - ▶ Freedom in the implementation due to abstract syntax trees
- ▶ Haskell supports monomorphism, via libraries
 - ▶ No need to extend the core language, reduces complexity
 - ▶ An embedding demonstrates the expressivity of a language

Question

Interesting difference between the Clean and Haskell approach:

- ▶ Clean supports polymorphism, using a built-in system
 - ▶ Freedom in the implementation due to abstract syntax trees
- ▶ Haskell supports monomorphism, via libraries
 - ▶ No need to extend the core language, reduces complexity
 - ▶ An embedding demonstrates the expressivity of a language

What does it take for a functional language to embed polymorphic dynamic typing?

Question

Interesting difference between the Clean and Haskell approach:

- ▶ Clean supports polymorphism, using a built-in system
 - ▶ Freedom in the implementation due to abstract syntax trees
- ▶ Haskell supports monomorphism, via libraries
 - ▶ No need to extend the core language, reduces complexity
 - ▶ An embedding demonstrates the expressivity of a language

What does it take for a functional language to embed polymorphic dynamic typing?

Part 1 Embedding polymorphic dynamic typing in Haskell

Question

Interesting difference between the Clean and Haskell approach:

- ▶ Clean supports polymorphism, using a built-in system
 - ▶ Freedom in the implementation due to abstract syntax trees
- ▶ Haskell supports monomorphism, via libraries
 - ▶ No need to extend the core language, reduces complexity
 - ▶ An embedding demonstrates the expressivity of a language

What does it take for a functional language to embed polymorphic dynamic typing?

Part 1 Embedding polymorphic dynamic typing in Haskell = not obvious

Question

Interesting difference between the Clean and Haskell approach:

- ▶ Clean supports polymorphism, using a built-in system
 - ▶ Freedom in the implementation due to abstract syntax trees
- ▶ Haskell supports monomorphism, via libraries
 - ▶ No need to extend the core language, reduces complexity
 - ▶ An embedding demonstrates the expressivity of a language

What does it take for a functional language to embed polymorphic dynamic typing?

Part 1 Embedding polymorphic dynamic typing in Haskell = not obvious

Part 2 Embedding polymorphic dynamic typing in Agda

Part 1

Embedding polymorphic dynamic typing in Haskell

A universe for the representation of polymorphic types

A dynamic constitutes a value and a representation of its type:

data *Dyn* = **forall** *a* . *Dyn* (V *a* ()) *a*

A universe for the representation of polymorphic types

A dynamic constitutes a value and a representation of its type:

```
data Dyn = forall a . Dyn (V a ()) a
```

The universe V binds variables and closes representations:

```
data V :: * → * → * where
```

```
  BASE    :: U a env → V a env
```

```
  FORALL :: V a (U b (), env) → V a env
```

A universe for the representation of polymorphic types

A dynamic constitutes a value and a representation of its type:

```
data Dyn = forall a . Dyn (V a ()) a
```

The universe V binds variables and closes representations:

```
data V :: * → * → * where  
  BASE   :: U a env → V a env  
  FORALL :: V a (U b (), env) → V a env
```

The base universe U includes types and references to variables:

```
data U :: * → * → * where  
  INT   :: U Int env  
  PAIR  :: U a env → U b env → U (a, b) env  
  (:=:) :: U a env → U b env → U (a → b) env  
  VAR  :: Ref (U a ()) env → U a env
```

De Bruijn indices as variable references

Variable references are De Bruijn indices in a typed environment:

```
data Ref :: * → * → * where  
  Rz :: Ref a (a, env)  
  Rs :: Ref a env → Ref a (b, env)
```

De Bruijn indices as variable references

Variable references are De Bruijn indices in a typed environment:

```
data Ref :: * → * → * where  
  Rz :: Ref a (a, env)  
  Rs :: Ref a env → Ref a (b, env)
```

Type **forall** $a . a \rightarrow a$ of the polymorphic identity function:

```
FORALL (BASE (VAR Rz :=> VAR Rz))
```

Deciding equality of type representations

A cast function that unwraps requires an equality proof on types:

```
data (:≡:) :: * → * → * where Refl :: a :≡: a
```

Deciding equality of type representations

A cast function that unwraps requires an equality proof on types:

data ($:\equiv:$) $:: * \rightarrow * \rightarrow *$ **where** $Refl :: a :\equiv: a$

Deciding equality of V representations structurally recurses:

$decV :: V a \dots \rightarrow V b \dots \rightarrow Maybe (a :\equiv: b)$

Deciding equality of type representations

A cast function that unwraps requires an equality proof on types:

```
data (≐) :: * → * → * where Refl :: a ≐ a
```

Deciding equality of V representations structurally recurses:

```
decV :: V a ... → V b ... → Maybe (a ≐ b)
```

```
decV (BASE u1) (BASE u2) =
```

```
  case decU u1 u2 of
```

```
    Just Refl → Just Refl
```

```
    Nothing → Nothing
```

Deciding equality of type representations

A cast function that unwraps requires an equality proof on types:

```
data (≐) :: * → * → * where Refl :: a ≐ a
```

Deciding equality of V representations structurally recurses:

```
decV :: V a ... → V b ... → Maybe (a ≐ b)
```

```
decV (BASE u1) (BASE u2) =
```

```
  case decU u1 u2 of
```

```
    Just Refl → Just Refl
```

```
    Nothing → Nothing
```

```
decV (FORALL v1) (FORALL v2) =
```

```
  case decV v1 v2 of
```

```
    Just Refl → Just Refl
```

```
    Nothing → Nothing
```

Deciding equality of type representations

A cast function that unwraps requires an equality proof on types:

data (*≐*) :: * → * → * **where** *Refl* :: a *≐* a

Deciding equality of *V* representations structurally recurses:

decV :: *V* a ... → *V* b ... → *Maybe* (a *≐* b)

decV (*BASE* u1) (*BASE* u2) =

case *decU* u1 u2 **of**

Just Refl → *Just Refl*

Nothing → *Nothing*

decV (*FORALL* v1) (*FORALL* v2) =

case *decV* v1 v2 **of**

Just Refl → *Just Refl*

Nothing → *Nothing*

decV _ _ = *Nothing*

Deciding equality of type representations

A cast function that unwraps requires an equality proof on types:

data ($:\equiv$) $:: * \rightarrow * \rightarrow *$ **where** $Refl :: a :\equiv a$

Deciding equality of V representations structurally recurses:

$decV :: V a \dots \rightarrow V b \dots \rightarrow Maybe (a :\equiv b)$

$decV (BASE u1) (BASE u2) =$

case $decU u1 u2$ **of**

$Just Refl \rightarrow Just Refl$

$Nothing \rightarrow Nothing$

$decV (FORALL v1) (FORALL v2) =$

case $decV v1 v2$ **of**

$Just Refl \rightarrow Just Refl$

$Nothing \rightarrow Nothing$

$decV _ _ = Nothing$

Extensions of both environments are not known to be equal!

$FORALL :: V a (U b (), env) \rightarrow V a env$

Deciding equality of type representations

A cast function that unwraps requires an equality proof on types:

data ($:\equiv$) $:: * \rightarrow * \rightarrow *$ **where** $Refl :: a :\equiv a$

Deciding equality of V representations structurally recurses:

$decV :: V a env1 \rightarrow V b env2 \rightarrow Maybe (a :\equiv b)$

$decV (BASE u1) (BASE u2) =$

case $decU u1 u2$ **of**

$Just Refl \rightarrow Just Refl$

$Nothing \rightarrow Nothing$

$decV (FORALL v1) (FORALL v2) =$

case $decV v1 v2$ **of**

$Just Refl \rightarrow Just Refl$

$Nothing \rightarrow Nothing$

$decV _ _ = Nothing$

Extensions of both environments are not known to be equal!

$FORALL :: V a (U b (), env) \rightarrow V a env$

Deciding equality of variable references

Deciding equality of U representations is straightforward:

$decU :: U\ a\ env1 \rightarrow U\ b\ env2 \rightarrow Maybe\ (a\ \equiv\ b)$

...

$decU\ (VAR\ i)\ (VAR\ j) =$

case $decRef\ i\ j$ **of**

$Just\ Refl \rightarrow Just\ Refl$

$Nothing \rightarrow Nothing$

$decU\ _ \quad _ = Nothing$

Deciding equality of variable references

Deciding equality of U representations is straightforward:

$$\text{decU} :: U\ a\ \text{env1} \rightarrow U\ b\ \text{env2} \rightarrow \text{Maybe}\ (a \equiv b)$$

...

$$\text{decU}\ (\text{VAR}\ i)\ (\text{VAR}\ j) =$$

case $\text{decRef}\ i\ j$ **of**

$\text{Just}\ \text{Refl} \rightarrow \text{Just}\ \text{Refl}$

$\text{Nothing} \rightarrow \text{Nothing}$

$$\text{decU}\ _ \quad _ \quad = \text{Nothing}$$

Fails when references to different environments are compared!

$$\text{decRef} :: \text{Ref}\ a\ \text{env1} \rightarrow \text{Ref}\ b\ \text{env2} \rightarrow \text{Maybe}\ (a \equiv b)$$
$$\text{decRef}\ Rz\ Rz = \dots$$
$$\text{decRef}\ (Rs\ i)\ (Rs\ j) =$$

case $\text{decRef}\ i\ j$ **of**

$\text{Just}\ \text{Refl} \rightarrow \text{Just}\ \text{Refl}$

$\text{Nothing} \rightarrow \text{Nothing}$

$$\text{decRef}\ _ \quad _ \quad = \text{Nothing}$$

Difficulties

Difficulty in Haskell lies in attaching the type to a representation:

- ▶ Essential to define relation between representations and values
- ▶ Obstructs their comparison due to incomparable references

Difficulties

Difficulty in Haskell lies in attaching the type to a representation:

- ▶ Essential to define relation between representations and values
- ▶ Obstructs their comparison due to incomparable references

Solution is to separate the representation from its interpretation:

- ▶ Postpone attachment of types and the use of environments
- ▶ Requires Haskell type-level computations that are not obvious

Difficulties

Difficulty in Haskell lies in attaching the type to a representation:

- ▶ Essential to define relation between representations and values
- ▶ Obstructs their comparison due to incomparable references

Solution is to separate the representation from its interpretation:

- ▶ Postpone attachment of types and the use of environments
- ▶ Requires Haskell type-level computations that are not obvious

A dependently-typed language provides better tools for this job

Part 2

Embedding polymorphic dynamic typing in Agda

A universe without types attached

In Agda, a dynamic interprets a representation to obtain the type:

```
data Dyn : Set where
```

```
  dyn : (v : V Zero) → eIV0 v → Dyn
```

A universe without types attached

In Agda, a dynamic interprets a representation to obtain the type:

```
data Dyn : Set where  
  dyn : (v : V Zero) → eIV0 v → Dyn
```

Now, the universes V and U only mention the available variables:

```
data V (n : Nat) : Set where  
  BASE    : U n → V n  
  FORALL  : V (Succ n) → V n
```

A universe without types attached

In Agda, a dynamic interprets a representation to obtain the type:

```
data Dyn : Set where  
  dyn : (v : V Zero) → eIV0 v → Dyn
```

Now, the universes V and U only mention the available variables:

```
data V (n : Nat) : Set where  
  BASE    : U n → V n  
  FORALL  : V (Succ n) → V n
```

```
data U (n : Nat) : Set where  
  NAT     : U n  
  PAIR    : U n → U n → U n  
  _⇒_    : U n → U n → U n  
  VAR     : Fin n → U n
```

Finite values as variable references

The datatype `Fin n` describes references between Zero and `n`:

```
data Fin : Nat → Set where  
  Fz : forall {n} → Fin (Succ n)  
  Fs : forall {n} → Fin n → Fin (Succ n)
```

Finite values as variable references

The datatype $\text{Fin } n$ describes references between Zero and n :

```
data Fin : Nat → Set where  
  Fz : forall {n} → Fin (Succ n)  
  Fs : forall {n} → Fin n → Fin (Succ n)
```

Type of the polymorphic identity function:

```
FORALL (BASE (VAR Fz ⇒ VAR Fz))
```

From a representation to a type

We need functions to interpret representations and obtain types:

$$\text{eIV0} : \text{V Zero} \rightarrow \text{Set}$$
$$\text{eIV0} (\text{BASE } u) = \text{eIU0 } u$$
$$\text{eIV0} (\text{FORALL } v) = \text{eIV} (\text{FORALL } v) \text{ Nil}$$

From a representation to a type

We need functions to interpret representations and obtain types:

$$\text{eIV0} : \text{V Zero} \rightarrow \text{Set}$$
$$\text{eIV0} (\text{BASE } u) = \text{eIU0 } u$$
$$\text{eIV0} (\text{FORALL } v) = \text{eIV} (\text{FORALL } v) \text{ Nil}$$
$$\text{eIU0} : \text{U Zero} \rightarrow \text{Set}$$
$$\text{eIU0} \text{ NAT} = \text{Nat}$$
$$\text{eIU0} (\text{PAIR } u \ u') = \text{Pair} (\text{eIU0 } u) (\text{eIU0 } u')$$
$$\text{eIU0} (u \Rightarrow u') = \text{eIU0 } u \rightarrow \text{eIU0 } u'$$
$$\text{eIU0} (\text{VAR } ())$$

Passing along an environment

Interpretation with variables takes an environment Env of length n :

$eV : \mathbf{forall} \{n\} \rightarrow V \ n \rightarrow \text{Env} \ n \rightarrow \text{Set}$

$eV (\text{BASE } u) \ \text{env} = eU \ u \ \text{env}$

$eV (\text{FORALL } v) \ \text{env} = \mathbf{forall} \{u\} \rightarrow eV \ v \ (\text{Cons } u \ \text{env})$

Passing along an environment

Interpretation with variables takes an environment Env of length n :

$\text{elV} : \mathbf{forall} \{n\} \rightarrow V \ n \rightarrow \text{Env} \ n \rightarrow \text{Set}$

$\text{elV} (\text{BASE } u) \ \text{env} = \text{elU } u \ \text{env}$

$\text{elV} (\text{FORALL } v) \ \text{env} = \mathbf{forall} \{u\} \rightarrow \text{elV } v \ (\text{Cons } u \ \text{env})$

$\text{elU} : \mathbf{forall} \{n\} \rightarrow U \ n \rightarrow \text{Env} \ n \rightarrow \text{Set}$

$\text{elU } \text{NAT} \quad _ = \text{Nat}$

$\text{elU} (\text{PAIR } u \ u') \ \text{env} = \text{Pair} (\text{elU } u \ \text{env}) (\text{elU } u' \ \text{env})$

$\text{elU} (u \Rightarrow u') \ \text{env} = \text{elU } u \ \text{env} \rightarrow \text{elU } u' \ \text{env}$

$\text{elU} (\text{VAR } i) \ \text{env} = \text{elU0} (\text{lookup } i \ \text{env})$

Deciding equality without environments

Now, a cast function uses an equality proof on values:

```
data _≡_ {a : Set} (x : a) : a → Set where Refl : x ≡ x
```

Deciding equality without environments

Now, a cast function uses an equality proof on values:

```
data _≡_ {a : Set} (x : a) : a → Set where Refl : x ≡ x
```

Equality functions only mention number of variables available:

```
decV : forall {n} → (v1 v2 : V n) → Maybe (v1 ≡ v2)
```

```
decU : forall {n} → (u1 u2 : U n) → Maybe (u1 ≡ u2)
```

Deciding equality without environments

Now, a cast function uses an equality proof on values:

```
data _≡_ {a : Set} (x : a) : a → Set where Refl : x ≡ x
```

Equality functions only mention number of variables available:

```
decV : forall {n} → (v1 v2 : V n) → Maybe (v1 ≡ v2)
```

```
decU : forall {n} → (u1 u2 : U n) → Maybe (u1 ≡ u2)
```

Unlike in Haskell, comparison of references poses no problems:

```
decFin : forall {n} → (i j : Fin n) → Maybe (i ≡ j)
```

```
decFin Fz Fz = Just Refl
```

```
decFin (Fs i) (Fs j) with decFin i j
```

```
decFin (Fs i) (Fs .i) | Just Refl = Just Refl
```

```
decFin (Fs _) (Fs _) | Nothing = Nothing
```

```
decFin _ _ = Nothing
```

Casting a value with equality of representations

The cast function deploys the proof of equality:

```
cast : (v1 : V Zero) → Dyn → Maybe (eIV0 v1)
cast v1 (dyn v2 x) with decV v1 v2
cast v1 (dyn .v1 x) | Just Refl = Just x
cast v1 (dyn v2 x) | Nothing  = Nothing
```

Equal representations result in equal types after interpretation

From equality to an instance-of check

Using equality of representations in a cast function does not cut it:

`idType : V Zero`

`idType = FORALL (BASE (VAR Fz \Rightarrow VAR Fz))`

`idDyn : Dyn`

`idDyn = dyn idType ($\lambda x \rightarrow x$)`

From equality to an instance-of check

Using equality of representations in a cast function does not cut it:

```
idType : V Zero
```

```
idType = FORALL (BASE (VAR Fz ⇒ VAR Fz))
```

```
idDyn : Dyn
```

```
idDyn = dyn idType (λ x → x)
```

We want casting to a more specific type also to succeed:

```
cast (BASE (NAT ⇒ NAT)) idDyn
```

From equality to an instance-of check

Using equality of representations in a cast function does not cut it:

```
idType : V Zero
idType = FORALL (BASE (VAR Fz ⇒ VAR Fz))
idDyn  : Dyn
idDyn  = dyn idType (λ x → x)
```

We want casting to a more specific type also to succeed:

```
cast (BASE (NAT ⇒ NAT)) idDyn
```

While casting to a more general type must fail:

```
incDyn : Dyn
incDyn = dyn (BASE (NAT ⇒ NAT)) (λ x → x + 1)
cast idType incDyn
```

Reusing an existing unification algorithm

Determine if one type (representation) is an instance of another:

- ▶ Our instance-of algorithm is based on unification
iof : **forall** {n} → U n → U n → Maybe (Subst n)
- ▶ Unification algorithm in Agda requires great care
 - ▶ Functions in Agda must be structurally recursive
 - ▶ Common algorithms apply results from one branch to another
- ▶ We reuse McBride's (2003) work in Epigram on
First-order unification by structural recursion

Reusing an existing unification algorithm

Determine if one type (representation) is an instance of another:

- ▶ Our instance-of algorithm is based on unification
 $\text{iof} : \mathbf{forall} \{n\} \rightarrow U\ n \rightarrow U\ n \rightarrow \text{Maybe (Subst } n)$
- ▶ Unification algorithm in Agda requires great care
 - ▶ Functions in Agda must be structurally recursive
 - ▶ Common algorithms apply results from one branch to another
- ▶ We reuse McBride's (2003) work in Epigram on
First-order unification by structural recursion

Original algorithm is easily turned into an instance-of check:

$$\begin{aligned} \text{iofAcc (VAR } i) (\text{VAR } j) (\text{Witness } _ \text{ Nil}) &= \text{Just (flexFlex } j\ i) \\ \text{iofAcc (VAR } i) v \quad (\text{Witness } _ \text{ Nil}) &= \text{flexRigid } i\ v \\ \text{iofAcc } v \quad (\text{VAR } j) (\text{Witness } _ \text{ Nil}) &= \text{flexRigid } j\ v \end{aligned}$$

Reusing an existing unification algorithm

Determine if one type (representation) is an instance of another:

- ▶ Our instance-of algorithm is based on unification
 $\text{iof} : \mathbf{forall} \{n\} \rightarrow U\ n \rightarrow U\ n \rightarrow \text{Maybe} (\text{Subst}\ n)$
- ▶ Unification algorithm in Agda requires great care
 - ▶ Functions in Agda must be structurally recursive
 - ▶ Common algorithms apply results from one branch to another
- ▶ We reuse McBride's (2003) work in Epigram on
First-order unification by structural recursion

Original algorithm is easily turned into an instance-of check:

$$\begin{aligned} \text{iofAcc} (\text{VAR } i) (\text{VAR } j) (\text{Witness } _ \text{ Nil}) &= \text{Just} (\text{flexFlex } j\ i) \\ \text{iofAcc} (\text{VAR } i) v \text{-----} (\text{Witness } _ \text{ Nil}) &= \text{flexRigid } i\ v \\ \text{iofAcc } v \quad (\text{VAR } j) (\text{Witness } _ \text{ Nil}) &= \text{flexRigid } j\ v \end{aligned}$$

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe (eIV0 v1)}$$

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe (elV0 v1)}$$

How to get from a value in a dynamic of type elV0 v2 ...

... to returning a value of type elV0 v1 , without an equality proof?

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe (eIV0 v1)}$$

How to get from a value in a dynamic of type eIV0 v2 ...

1. Unpack the empty environment

$$(\text{env} : \text{Env Zero}) \rightarrow \text{eIV v2 env}$$

... to returning a value of type eIV0 v1, without an equality proof?

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe} (\text{elV0 } v1)$$

How to get from a value in a dynamic of type $\text{elV0 } v2$...

1. Unpack the empty environment

$$(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v2 \text{ env}$$

2. Strip the quantifiers

$$(\text{env} : \text{Env} (\text{vars } v2)) \rightarrow \text{elU} (\text{strip } v2) \text{ env}$$

... to returning a value of type $\text{elV0 } v1$, without an equality proof?

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe (eIV0 v1)}$$

How to get from a value in a dynamic of type eIV0 v2 ...

1. Unpack the empty environment

$$(\text{env} : \text{Env Zero}) \rightarrow \text{eIV v2 env}$$

2. Strip the quantifiers

$$(\text{env} : \text{Env (vars v2)}) \rightarrow \text{eIU (strip v2) env}$$

3. Instantiate the environment using the substitution

$$(\text{env} : \text{Env (vars v1)}) \rightarrow \text{eIU (apply subst (strip v2)) env}$$

... to returning a value of type eIV0 v1, without an equality proof?

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe} (\text{elV0 } v1)$$

How to get from a value in a dynamic of type $\text{elV0 } v2 \dots$

1. Unpack the empty environment

$$(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v2 \text{ env}$$

2. Strip the quantifiers

$$(\text{env} : \text{Env} (\text{vars } v2)) \rightarrow \text{elU} (\text{strip } v2) \text{ env}$$

3. Instantiate the environment using the substitution

$$(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{apply subst} (\text{strip } v2)) \text{ env}$$

4. Coerce using a correctness proof of instance-of algorithm

$$(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{strip } v1) \text{ env}$$

... to returning a value of type $\text{elV0 } v1$, without an equality proof?

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe} (\text{elV0 } v1)$$

How to get from a value in a dynamic of type $\text{elV0 } v2 \dots$

1. Unpack the empty environment
 $(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v2 \text{ env}$
2. Strip the quantifiers
 $(\text{env} : \text{Env} (\text{vars } v2)) \rightarrow \text{elU} (\text{strip } v2) \text{ env}$
3. Instantiate the environment using the substitution
 $(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{apply subst} (\text{strip } v2)) \text{ env}$
4. Coerce using a correctness proof of instance-of algorithm
 $(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{strip } v1) \text{ env}$
5. Dress with quantifiers
 $(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v1 \text{ env}$

... to returning a value of type $\text{elV0 } v1$, without an equality proof?

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe} (\text{elV0 } v1)$$

How to get from a value in a dynamic of type $\text{elV0 } v2 \dots$

1. Unpack the empty environment

$$(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v2 \text{ env}$$

2. Strip the quantifiers

$$(\text{env} : \text{Env} (\text{vars } v2)) \rightarrow \text{elU} (\text{strip } v2) \text{ env}$$

3. Instantiate the environment using the substitution

$$(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{apply subst} (\text{strip } v2)) \text{ env}$$

4. Coerce using a correctness proof of instance-of algorithm

$$(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{strip } v1) \text{ env}$$

5. Dress with quantifiers

$$(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v1 \text{ env}$$

6. Pack the empty environment

$$\text{elV0 } v1$$

... to returning a value of type $\text{elV0 } v1$, without an equality proof?

A framework for the cast function

Using the instance-of algorithm we define the cast function again:

$$\text{cast} : (v1 : V \text{ Zero}) \rightarrow \text{Dyn} \rightarrow \text{Maybe} (\text{elV0 } v1)$$

How to get from a value in a dynamic of type $\text{elV0 } v2 \dots$

1. Unpack the empty environment

$$(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v2 \text{ env}$$

2. Strip the quantifiers

$$(\text{env} : \text{Env} (\text{vars } v2)) \rightarrow \text{elU} (\text{strip } v2) \text{ env}$$

3. Instantiate the environment using the substitution

postulate $(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{apply subst} (\text{strip } v2)) \text{ env}$

4. Coerce using a correctness proof of instance-of algorithm

postulate $(\text{env} : \text{Env} (\text{vars } v1)) \rightarrow \text{elU} (\text{strip } v1) \text{ env}$

5. Dress with quantifiers

$$(\text{env} : \text{Env Zero}) \rightarrow \text{elV } v1 \text{ env}$$

6. Pack the empty environment

$$\text{elV0 } v1$$

... to returning a value of type $\text{elV0 } v1$, without an equality proof?

Conclusion

Agda provides better tools to perform the embedding:

- ▶ In Haskell, an attached interpretation poses difficulties
- ▶ Separate the representation from its interpretation
- ▶ Postpone attaching meaning until after comparison

Conclusion

Agda provides better tools to perform the embedding:

- ▶ In Haskell, an attached interpretation poses difficulties
- ▶ Separate the representation from its interpretation
- ▶ Postpone attaching meaning until after comparison

Several ideas for future work:

- ▶ A backport from the Agda implementation to Haskell
- ▶ Resolve the two remaining postulates in the Agda framework
- ▶ Simplify the universe V and hence the intricate cast function