

# A Typical Synergy

## Dynamic Types and Generalised Algebraic Datatypes

Thomas van Noort

joint work with

Peter Achten    Rinus Plasmeijer

Institute for Computing and Information Sciences  
Radboud University Nijmegen

MBSD Seminar  
December 11, 2009

## Setting the scene

In strongly typed functional languages, static typing is used to:

- ▶ Prevent erroneous behaviour at run-time
- ▶ Generate efficient code using knowledge at compile-time

## Setting the scene

In strongly typed functional languages, static typing is used to:

- ▶ Prevent erroneous behaviour at run-time
- ▶ Generate efficient code using knowledge at compile-time

But sometimes static typing gets in your (and my) way:

- ▶ Example: type-safe manipulation of heterogeneous structures
- ▶ Idea: use dynamic typing to elegantly define such a function

## Setting the scene

In strongly typed functional languages, static typing is used to:

- ▶ Prevent erroneous behaviour at run-time
- ▶ Generate efficient code using knowledge at compile-time

But sometimes static typing gets in your (and my) way:

- ▶ Example: type-safe manipulation of heterogeneous structures
- ▶ Idea: use dynamic typing to elegantly define such a function

### Dynamic types

- ▶ Wrap values with their types

### Generalised algebraic datatypes

- ▶ Heterogeneous structures

## Setting the scene

In strongly typed functional languages, static typing is used to:

- ▶ Prevent erroneous behaviour at run-time
- ▶ Generate efficient code using knowledge at compile-time

But sometimes static typing gets in your (and my) way:

- ▶ Example: type-safe manipulation of heterogeneous structures
- ▶ Idea: use dynamic typing to elegantly define such a function

### Dynamic types

- ▶ Wrap values with their types
- ▶ Full support in Clean
- ▶ Minimal support in Haskell, only monomorphic values

### Generalised algebraic datatypes

- ▶ Heterogeneous structures

## Setting the scene

In strongly typed functional languages, static typing is used to:

- ▶ Prevent erroneous behaviour at run-time
- ▶ Generate efficient code using knowledge at compile-time

But sometimes static typing gets in your (and my) way:

- ▶ Example: type-safe manipulation of heterogeneous structures
- ▶ Idea: use dynamic typing to elegantly define such a function

### Dynamic types

- ▶ Wrap values with their types
- ▶ Full support in Clean
- ▶ Minimal support in Haskell, only monomorphic values

### Generalised algebraic datatypes

- ▶ Heterogeneous structures
- ▶ No support in Clean, only regular ADTs
- ▶ Full support in Haskell

## Dynamic types - context-independent

Clean supports dynamics using the keyword **dynamic**:

*wrapInt* :: *Int* → *Dynamic*

*wrapInt* x = **dynamic** x

## Dynamic types - context-independent

Clean supports dynamics using the keyword **dynamic**:

```
wrapInt :: Int → Dynamic  
wrapInt x = dynamic x
```

Values are unwrapped using the :: annotation:

```
unwrapInt :: Dynamic → Int  
unwrapInt (x :: Int)    = x  
unwrapInt (x :: String) = stringToInt x  
unwrapInt _              = 0
```

Dynamic pattern matches can fail due to run-time type-unification

## Dynamic types - context-dependent

Values can be wrapped depending on the context:

$wrap :: TC\ \alpha \Rightarrow \alpha \rightarrow Dynamic$

$wrap\ x = \mathbf{dynamic}\ x$

## Dynamic types - context-dependent

Values can be wrapped depending on the context:

$$\text{wrap} :: TC\ \alpha \Rightarrow \alpha \rightarrow \text{Dynamic}$$
$$\text{wrap}\ x = \mathbf{dynamic}\ x$$

Use the  $\wedge$  notation to refer to context-dependent type variables:

$$\text{unwrap} :: TC\ \alpha \Rightarrow \text{Dynamic} \rightarrow \alpha$$
$$\text{unwrap}\ (x :: \alpha^\wedge) = x$$
$$\text{unwrap}\ \_ = \text{error}\ \text{"unwrap: incorrect type"}$$

## Dynamic types - context-dependent

Values can be wrapped depending on the context:

```
wrap :: TC  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Dynamic  
wrap  $x$  = dynamic  $x$ 
```

Use the  $\wedge$  notation to refer to context-dependent type variables:

```
unwrap :: TC  $\alpha$   $\Rightarrow$  Dynamic  $\rightarrow$   $\alpha$   
unwrap ( $x$  ::  $\alpha^\wedge$ ) =  $x$   
unwrap _ = error "unwrap: incorrect type"
```

The built-in type class *TC* provides type codes:

- ▶ Value representation of types, available for non-opaque types
- ▶ Required when (un)wrapped value is context-dependent

## Dynamic types - scoping and laziness

Scope of dynamic type variables is the complete function arm:

$apply :: TC\ \alpha \Rightarrow Dynamic \rightarrow Dynamic \rightarrow Maybe\ \alpha$

$apply\ (f :: \beta \rightarrow \alpha^{\wedge})\ (x :: \beta) = Just\ (f\ x)$

$apply\ \_ \quad \quad \quad \_ \quad \quad = Nothing$

## Dynamic types - scoping and laziness

Scope of dynamic type variables is the complete function arm:

$$\begin{aligned} \text{apply} &:: TC \alpha \Rightarrow \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Maybe } \alpha \\ \text{apply } (f &:: \beta \rightarrow \alpha^\wedge) (x &:: \beta) = \text{Just } (f \ x) \\ \text{apply } \_ & \quad \quad \quad \_ = \text{Nothing} \end{aligned}$$

This enforces type equality between the two arguments:

$$\begin{aligned} \text{apply } (\mathbf{dynamic} \text{ fst}) (\mathbf{dynamic} (1, "2")) &\rightsquigarrow \text{Just } 1 \\ \text{apply } (\mathbf{dynamic} \text{ fst}) (\mathbf{dynamic} 1) &\rightsquigarrow \text{Nothing} \end{aligned}$$

## Dynamic types - scoping and laziness

Scope of dynamic type variables is the complete function arm:

$$\begin{aligned} \text{apply} &:: TC \alpha \Rightarrow \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Maybe } \alpha \\ \text{apply } (f &:: \beta \rightarrow \alpha^\wedge) (x &:: \beta) = \text{Just } (f \ x) \\ \text{apply } \_ & \_ = \text{Nothing} \end{aligned}$$

This enforces type equality between the two arguments:

$$\begin{aligned} \text{apply } (\mathbf{dynamic} \text{ fst}) (\mathbf{dynamic} (1, "2")) &\rightsquigarrow \text{Just } 1 \\ \text{apply } (\mathbf{dynamic} \text{ fst}) (\mathbf{dynamic} 1) &\rightsquigarrow \text{Nothing} \end{aligned}$$

Dynamics preserve lazy behaviour of functional programs:

$$\text{apply } (\mathbf{dynamic} \text{ fst}) (\mathbf{dynamic} (1, \perp)) \rightsquigarrow \text{Just } 1$$

## Generalised algebraic datatypes - structural validity

Consider a Haskell algebraic datatype representing  $\lambda$ -terms:

```
data Lam = Undef  
        | Const Value  
        | App Lam Lam
```

## Generalised algebraic datatypes - structural validity

Consider a Haskell algebraic datatype representing  $\lambda$ -terms:

```
data Lam = Undef  
         | Const Value  
         | App Lam Lam
```

Since *Lam* is heterogeneous, values need to be enumerated:

```
data Value = VInt Int  
           | VFun (Value  $\rightarrow$  Value)
```

## Generalised algebraic datatypes - structural validity

Consider a Haskell algebraic datatype representing  $\lambda$ -terms:

```
data Lam = Undef
        | Const Value
        | App Lam Lam
```

Since *Lam* is heterogeneous, values need to be enumerated:

```
data Value = VInt Int
           | VFun (Value  $\rightarrow$  Value)
```

Moreover, structural validity cannot be ensured statically:

```
eval :: Lam  $\rightarrow$  Value
eval Undef                =  $\perp$ 
eval (Const x)           = x
eval (App (Const (VFun f)) x) = f (eval x)
eval (App _ _            _ ) = error "eval: not a function"
```

## Generalised algebraic datatypes - constructor types

Haskell supports heterogeneous structures via GADTs:

**data** *Lam* :: \* → \* **where**

*Undef* :: *Lam* α

*Const* :: α → *Lam* α

*App* :: *Lam* (β → α) → *Lam* β → *Lam* α

*term* :: *Lam* Int

*term* = *App* (*Const* *abs*) (*Const* 1)

## Generalised algebraic datatypes - constructor types

Haskell supports heterogeneous structures via GADTs:

**data** *Lam* :: \* → \* **where**

*Undef* :: *Lam* α

*Const* :: α → *Lam* α

*App* :: *Lam* (β → α) → *Lam* β → *Lam* α

*term* :: *Lam* Int

*term* = *App* (*Const* *abs*) (*Const* 1)

Structural information is employed in evaluation function:

*eval* :: *Lam* α → α

*eval* *Undef* = ⊥

*eval* (*Const* *x*) = *x*

*eval* (*App* *f* *x*) = *eval* *f* (*eval* *x*)

Result type is (possibly) different for each constructor occurrence

## Goal

Our goal is to elegantly define a type-safe GADT update function:

$$\text{update} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam } \alpha$$

## Goal

Our goal is to elegantly define a type-safe GADT update function:

$$\text{update} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam } \alpha$$

Challenge is to ensure type equality between old and new values:

$$\text{term} :: \text{Lam } \text{Int}$$
$$\text{term} = \text{App } (\text{Const } \text{abs}) (\text{Const } 1)$$

## Goal

Our goal is to elegantly define a type-safe GADT update function:

$$\text{update} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam } \alpha$$

Challenge is to ensure type equality between old and new values:

$$\text{term} :: \text{Lam } \text{Int}$$
$$\text{term} = \text{App } (\text{Const } \text{abs}) (\text{Const } 1)$$
$$\text{update term } [0,0] \text{ neg} \rightsquigarrow \text{App } (\text{Const } \text{neg}) (\text{Const } 1)$$

## Goal

Our goal is to elegantly define a type-safe GADT update function:

$$\text{update} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam } \alpha$$

Challenge is to ensure type equality between old and new values:

$$\text{term} :: \text{Lam Int}$$
$$\text{term} = \text{App } (\text{Const abs}) (\text{Const 1})$$
$$\text{update term [0,0] neg} \rightsquigarrow \text{App } (\text{Const neg}) (\text{Const 1})$$
$$\text{update term [1,0] 2} \rightsquigarrow \text{App } (\text{Const abs}) (\text{Const 2})$$

## Goal

Our goal is to elegantly define a type-safe GADT update function:

$$\text{update} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam } \alpha$$

Challenge is to ensure type equality between old and new values:

$$\text{term} :: \text{Lam Int}$$
$$\text{term} = \text{App } (\text{Const abs}) (\text{Const 1})$$
$$\text{update term [0,0] neg} \rightsquigarrow \text{App } (\text{Const neg}) (\text{Const 1})$$
$$\text{update term [1,0] 2} \rightsquigarrow \text{App } (\text{Const abs}) (\text{Const 2})$$
$$\text{update term [1,0] "2"} \rightsquigarrow \text{App } (\text{Const abs}) (\text{Const 1})$$

## Goal

Our goal is to elegantly define a type-safe GADT update function:

$$\text{update} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam } \alpha$$

Challenge is to ensure type equality between old and new values:

$$\text{term} :: \text{Lam Int}$$
$$\text{term} = \text{App} (\text{Const abs}) (\text{Const 1})$$
$$\text{update term [0,0] neg} \rightsquigarrow \text{App} (\text{Const neg}) (\text{Const 1})$$
$$\text{update term [1,0] 2} \rightsquigarrow \text{App} (\text{Const abs}) (\text{Const 2})$$
$$\text{update term [1,0] "2"} \rightsquigarrow \text{App} (\text{Const abs}) (\text{Const 1})$$

For now we only consider updating the only field of *Const*

## Conventional approach - decorating types

Conventional GADT updating uses lightweight dynamics:

- ▶ Arthur Baars and Doaitse Swierstra, *Typing dynamic typing*
- ▶ James Cheney and Ralf Hinze, *A lightweight implementation of generics and dynamics*

## Conventional approach - decorating types

Conventional GADT updating uses lightweight dynamics:

- ▶ Arthur Baars and Doaitse Swierstra, *Typing dynamic typing*
- ▶ James Cheney and Ralf Hinze, *A lightweight implementation of generics and dynamics*

Include type representation in GADT to enforce type equality:

**data**  $Lam_R :: * \rightarrow *$  **where**

$Undef_R :: Lam_R \alpha$

$Const_R :: RepOf \alpha \rightarrow Lam_R \alpha$

$App_R :: Lam_R (\alpha \rightarrow \beta) \rightarrow Lam_R \alpha \rightarrow Lam_R \beta$

## Conventional approach - decorating types

Conventional GADT updating uses lightweight dynamics:

- ▶ Arthur Baars and Doaitse Swierstra, *Typing dynamic typing*
- ▶ James Cheney and Ralf Hinze, *A lightweight implementation of generics and dynamics*

Include type representation in GADT to enforce type equality:

```
data LamR :: * → * where  
  UndefR :: LamR α  
  ConstR  :: RepOf α → LamR α  
  AppR    :: LamR (α → β) → LamR α → LamR β
```

The difference is that the field of *Const* is wrapped with its type:

```
type RepOf α = (α, Rep α)
```

## Conventional approach - type representations

Type representations are defined by another GADT:

```
data Rep :: * → * where  
  RInt  :: Rep Int  
  RFun :: Rep α → Rep β → Rep (α → β)
```

## Conventional approach - type representations

Type representations are defined by another GADT:

```
data Rep :: * → * where  
  RInt  :: Rep Int  
  RFun :: Rep α → Rep β → Rep (α → β)
```

Equality of types is enforced by constructing a proof:

```
data Equal :: * → * → * where  
  Refl :: Equal α α
```

## Conventional approach - type representations

Type representations are defined by another GADT:

```
data Rep :: * → * where  
  RInt  :: Rep Int  
  RFun :: Rep α → Rep β → Rep (α → β)
```

Equality of types is enforced by constructing a proof:

```
data Equal :: * → * → * where  
  Refl :: Equal α α
```

The proof is provided by point-wise comparison:

```
eqRep :: Rep α → Rep β → Maybe (Equal α β)
```

## Conventional approach - decorating functions

The update function operates on the decorated type:

$$\text{update}_R :: \text{Lam}_R \alpha \rightarrow \text{Path} \rightarrow \text{RepOf } \beta \rightarrow \text{Lam}_R \alpha$$
$$\text{update}_R \text{ Undef}_R \quad [] \quad \_ \quad = \text{Undef}_R$$
$$\text{update}_R (\text{Const}_R (x, rx)) [0] \quad (y, ry) =$$

**case eqRep rx ry of**

*Just Refl*  $\rightarrow \text{Const}_R (y, ry)$

*Nothing*  $\rightarrow \text{Const}_R (x, rx)$

$$\text{update}_R (\text{App}_R f x) \quad (0 : p) y \quad = \text{App}_R (\text{update}_R f p y) x$$
$$\text{update}_R (\text{App}_R f x) \quad (1 : p) y \quad = \text{App}_R f (\text{update}_R x p y)$$
$$\text{update}_R x \quad \_ \quad \_ \quad = x$$

## Conventional approach - decorating functions

The update function operates on the decorated type:

$$\text{update}_R :: \text{Lam}_R \alpha \rightarrow \text{Path} \rightarrow \text{RepOf } \beta \rightarrow \text{Lam}_R \alpha$$
$$\text{update}_R \text{ Undef}_R \quad [] \quad \_ = \text{Undef}_R$$
$$\text{update}_R (\text{Const}_R (x, rx)) [0] (y, ry) =$$

**case eqRep rx ry of**

*Just Refl*  $\rightarrow \text{Const}_R (y, ry)$

*Nothing*  $\rightarrow \text{Const}_R (x, rx)$

$$\text{update}_R (\text{App}_R f x) \quad (0 : p) y \quad = \text{App}_R (\text{update}_R f p y) x$$
$$\text{update}_R (\text{App}_R f x) \quad (1 : p) y \quad = \text{App}_R f (\text{update}_R x p y)$$
$$\text{update}_R x \quad \_ \quad \_ = x$$
$$\text{update}_R (\text{Const}_R (\text{abs}, \text{RFun RInt RInt})) [0] (\text{neg}, \text{RFun RInt RInt})$$
$$\rightsquigarrow \text{Const}_R (\text{neg}, \text{RFun RInt RInt})$$

## Conventional approach - decorating functions

The update function operates on the decorated type:

$$\text{update}_R :: \text{Lam}_R \alpha \rightarrow \text{Path} \rightarrow \text{RepOf } \beta \rightarrow \text{Lam}_R \alpha$$
$$\text{update}_R \text{ Undef}_R \quad [] \quad \_ = \text{Undef}_R$$
$$\text{update}_R (\text{Const}_R (x, rx)) [0] (y, ry) =$$

**case**  $\text{eqRep } rx \text{ } ry$  **of**

$\text{Just Refl} \rightarrow \text{Const}_R (y, ry)$

$\text{Nothing} \rightarrow \text{Const}_R (x, rx)$

$$\text{update}_R (\text{App}_R f x) \quad (0 : p) y \quad = \text{App}_R (\text{update}_R f p y) x$$
$$\text{update}_R (\text{App}_R f x) \quad (1 : p) y \quad = \text{App}_R f (\text{update}_R x p y)$$
$$\text{update}_R x \quad \_ \quad \_ = x$$
$$\text{update}_R (\text{Const}_R (\text{abs}, \text{RFun } R\text{Int } R\text{Int})) [0] (\text{neg}, \text{RFun } R\text{Int } R\text{Int})$$
$$\rightsquigarrow \text{Const}_R (\text{neg}, \text{RFun } R\text{Int } R\text{Int})$$

This approach is not very elegant:

- ▶ Original datatype needs to be adapted
- ▶ GADT type representation definition is closed
- ▶ Functions are cluttered with type equality proofs

## The synergy

Annotate the field of a GADT to be of the desired type:

$$\begin{aligned} \text{update} &:: TC \beta \Rightarrow Lam \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha \\ \text{update } Undef & \quad [] \quad \_ = Undef \\ \text{update } (Const (x ::^G \beta^{\wedge})) & [0] \quad y = Const y \\ \text{update } (App f x) & \quad (0 : p) y = App (update f p y) x \\ \text{update } (App f x) & \quad (1 : p) y = App f (update x p y) \\ \text{update } x & \quad \_ \quad \_ = x \end{aligned}$$

## The synergy

Annotate the field of a GADT to be of the desired type:

$$\begin{aligned} \text{update} &:: TC \beta \Rightarrow Lam \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha \\ \text{update } Undef & \quad [] \quad \_ = Undef \\ \text{update } (Const (x ::^G \beta^{\wedge})) & [0] \quad y = Const y \\ \text{update } (App f x) & \quad (0 : p) y = App (update f p y) x \\ \text{update } (App f x) & \quad (1 : p) y = App f (update x p y) \\ \text{update } x & \quad \_ \quad \_ = x \\ \text{update } (Const abs) [0] neg & \rightsquigarrow Const neg \end{aligned}$$

## The synergy

Annotate the field of a GADT to be of the desired type:

$$\begin{aligned} \text{update} &:: TC \beta \Rightarrow Lam \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha \\ \text{update } Undef & \quad \quad \quad [] \quad \quad \_ = Undef \\ \text{update } (Const (x ::^{\mathcal{G}} \beta^{\wedge})) & [0] \quad y = Const y \\ \text{update } (App f x) & \quad \quad (0 : p) y = App (update f p y) x \\ \text{update } (App f x) & \quad \quad (1 : p) y = App f (update x p y) \\ \text{update } x & \quad \quad \quad \_ \quad \quad \_ = x \\ \text{update } (Const abs) [0] neg & \quad \rightsquigarrow Const neg \end{aligned}$$

Comparing this approach to the conventional approach:

- ▶ Original datatype is used in the function definition
- ▶ The  $::^{\mathcal{G}}$  annotation restricts the type of the GADT field
- ▶ Type equality proofs provided by dynamics

## Semantics - mechanical translation

GADTs have to contain extra type information, naive approach:

- ▶ Always inject type information at construction site
- ▶ Always pattern match type information at destruction site

## Semantics - mechanical translation

GADTs have to contain extra type information, naive approach:

- ▶ Always inject type information at construction site
- ▶ Always pattern match type information at destruction site

A more fine-grained translation scheme is less invasive:

- ▶ Translate to extended GADT, living next to the original
- ▶ Define conversion from the original to extended GADT
- ▶ Only translate patterns in functions that use  $::^{\mathcal{G}}$  annotations

This scheme realizes a mechanical translation to known concepts!

## Semantics - translating GADTs

Translate to extended GADT, living next to the original:

**data** *Lam* :: \* → \* **where**

*Undef* :: *Lam* α

*Const* :: α → *Lam* α

*App* :: *Lam* (α → β) → *Lam* α → *Lam* β

## Semantics - translating GADTs

Translate to extended GADT, living next to the original:

**data**  $Lam^\circ :: * \rightarrow *$  **where**

$Undef^\circ :: Lam^\circ \alpha$

$Const^\circ :: \alpha \rightarrow Lam^\circ \alpha$

$App^\circ :: Lam (\alpha \rightarrow \beta) \rightarrow Lam \alpha \rightarrow Lam^\circ \beta$

## Semantics - translating GADTs

Translate to extended GADT, living next to the original:

**data**  $Lam^\circ :: * \rightarrow *$  **where**

$Undef^\circ :: Lam^\circ \alpha$

$Const^\circ :: \text{TypeOf } \alpha \rightarrow Lam^\circ \alpha$

$App^\circ :: Lam (\alpha \rightarrow \beta) \rightarrow Lam \alpha \rightarrow Lam^\circ \beta$

## Semantics - translating GADTs

Translate to extended GADT, living next to the original:

**data**  $Lam^\circ :: * \rightarrow *$  **where**

$Undef^\circ :: Lam^\circ \alpha$

$Const^\circ :: TC \alpha \Rightarrow TypeOf \alpha \rightarrow Lam^\circ \alpha$

$App^\circ :: Lam (\alpha \rightarrow \beta) \rightarrow Lam \alpha \rightarrow Lam^\circ \beta$

## Semantics - translating GADTs

Translate to extended GADT, living next to the original:

```
data Lam◦ :: * → * where  
  Undef◦ :: Lam◦ α  
  Const◦  :: TC α ⇒ TypeOf α → Lam◦ α  
  App◦    :: Lam (α → β) → Lam α → Lam◦ β
```

Types are stored using dynamics, but now expose the stored type:

```
type TypeOf α = (α, Dynamic)
```

## Semantics - translating GADTs

Translate to extended GADT, living next to the original:

```
data Lamo :: * → * where  
  Undefo :: Lamo α  
  Consto :: TC α ⇒ TypeOf α → Lamo α  
  Appo   :: Lam (α → β) → Lam α → Lamo β
```

Types are stored using dynamics, but now expose the stored type:

```
type TypeOf α = (α, Dynamic)
```

Correctness by construction of stored dynamic:

```
typeOf :: TC α ⇒ α → TypeOf α  
typeOf x = (x, dynamic ⊥ :: α^)
```

Context of *typeOf* determines the type that is stored

## Semantics - converting GADTs

Define conversion from original to extended GADT:

$$\text{toLam}^\circ :: \text{Lam } \alpha \rightarrow \text{Lam}^\circ \alpha$$
$$\text{toLam}^\circ \text{Undef} = \text{Undef}^\circ$$
$$\text{toLam}^\circ (\text{Const } x) = \text{Const}^\circ (\text{typeOf } x)$$
$$\text{toLam}^\circ (\text{App } f \ x) = \text{App}^\circ f \ x$$

## Semantics - converting GADTs

Define conversion from original to extended GADT:

$$\text{toLam}^\circ :: \text{Lam } \alpha \rightarrow \text{Lam}^\circ \alpha$$
$$\text{toLam}^\circ \text{Undef} = \text{Undef}^\circ$$
$$\text{toLam}^\circ (\text{Const } x) = \text{Const}^\circ (\text{typeOf } x)$$
$$\text{toLam}^\circ (\text{App } f \ x) = \text{App}^\circ f \ x$$

The *TC* requirement of  $\text{Const}^\circ$  propagates to original *Const*:

- ▶ This requirement is silently added to original definition
- ▶ No additional constraints since there is a *TC* for "everyone"
- ▶ Minimal run-time overhead expected due to lazy dictionaries

## Semantics - translating functions

Only translate patterns in functions that use  $::^{\mathcal{G}}$  annotations:

$update :: TC \beta \Rightarrow Lam \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha$

$update \ Undef \quad [] \quad \_ = Undef$

$update \ (Const \ (x \ ::^{\mathcal{G}} \ \beta)) \ [0] \quad (y \ :: \ \beta^{\wedge}) = Const \ y$

$update \ (App \ f \ x) \quad (0 : p) \ y = App \ (update \ f \ p \ y) \ x$

$update \ (App \ f \ x) \quad (1 : p) \ y = App \ f \ (update \ x \ p \ y)$

$update \ x \ \_ \ \_ = x$

## Semantics - translating functions

Only translate patterns in functions that use  $::^{\mathcal{G}}$  annotations:

$$\text{update}^{\circ} :: TC \beta \Rightarrow Lam \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha$$
$$\text{update}^{\circ} \text{Undef} \quad [] \quad \_ = \text{Undef}$$
$$\text{update}^{\circ} (\text{Const } (x ::^{\mathcal{G}} \beta)) [0] \quad (y :: \beta^{\wedge}) = \text{Const } y$$
$$\text{update}^{\circ} (\text{App } f \ x) \quad (0 : p) \ y = \text{App } (\text{update } f \ p \ y) \ x$$
$$\text{update}^{\circ} (\text{App } f \ x) \quad (1 : p) \ y = \text{App } f \ (\text{update } x \ p \ y)$$
$$\text{update}^{\circ} \ x \ \_ \ \_ = x$$

## Semantics - translating functions

Only translate patterns in functions that use  $::^{\mathcal{G}}$  annotations:

$$\begin{aligned} \text{update}^{\circ} &:: TC \beta \Rightarrow Lam^{\circ} \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha \\ \text{update}^{\circ} \text{Undef}^{\circ} & \quad [] \quad \_ = \text{Undef} \\ \text{update}^{\circ} (\text{Const}^{\circ} (x \ ::^{\mathcal{G}} \beta)) & [0] \quad (y \ :: \beta^{\wedge}) = \text{Const } y \\ \text{update}^{\circ} (\text{App}^{\circ} f \ x) & \quad (0 : p) \ y = \text{App } (\text{update } f \ p \ y) \ x \\ \text{update}^{\circ} (\text{App}^{\circ} f \ x) & \quad (1 : p) \ y = \text{App } f \ (\text{update } x \ p \ y) \\ \text{update}^{\circ} x \ \_ \ \_ & \quad = x \end{aligned}$$

## Semantics - translating functions

Only translate patterns in functions that use  $::^{\mathcal{G}}$  annotations:

$$\begin{aligned} \text{update}^{\circ} &:: TC \beta \Rightarrow Lam^{\circ} \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha \\ \text{update}^{\circ} \text{Undef}^{\circ} & \quad [] \quad \_ = \text{Undef} \\ \text{update}^{\circ} (\text{Const}^{\circ} (x, \_ :: \beta)) & [0] \quad (y :: \beta^{\wedge}) = \text{Const } y \\ \text{update}^{\circ} (\text{App}^{\circ} f x) & \quad (0 : p) y = \text{App } (\text{update } f \ p \ y) \ x \\ \text{update}^{\circ} (\text{App}^{\circ} f x) & \quad (1 : p) y = \text{App } f \ (\text{update } x \ p \ y) \\ \text{update}^{\circ} x \ \_ \ \_ & \quad = x \end{aligned}$$

## Semantics - translating functions

Only translate patterns in functions that use  $::^G$  annotations:

$$\begin{aligned} \text{update}^\circ &:: TC \beta \Rightarrow Lam^\circ \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha \\ \text{update}^\circ \text{Undef}^\circ & \quad [] \quad \_ = \text{Undef} \\ \text{update}^\circ (\text{Const}^\circ (x, \_ :: \beta)) & [0] \quad (y :: \beta^\wedge) = \text{Const } y \\ \text{update}^\circ (\text{App}^\circ f x) & \quad (0 : p) y = \text{App } (\text{update } f \ p \ y) \ x \\ \text{update}^\circ (\text{App}^\circ f x) & \quad (1 : p) y = \text{App } f \ (\text{update } x \ p \ y) \\ \text{update}^\circ x \ \_ \ \_ & \quad = x \end{aligned}$$

Body of the function is unchanged, calling original function:

$$\begin{aligned} \text{update} &:: TC \beta \Rightarrow Lam \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam \alpha \\ \text{update } x \ p \ d &= \text{update}^\circ (\text{toLam}^\circ x) \ p \ d \end{aligned}$$

The conversion is localized and not exposed in the signature

## Conclusion

New  $::\mathcal{G}$  annotation provides access to GADT type information:

- ▶ No adaptation of original datatypes required
- ▶ Type equality proofs provided by dynamics
- ▶ Minimal run-time overhead expected due to lazy dictionaries

## Conclusion

New  $::^{\mathcal{G}}$  annotation provides access to GADT type information:

- ▶ No adaptation of original datatypes required
- ▶ Type equality proofs provided by dynamics
- ▶ Minimal run-time overhead expected due to lazy dictionaries

Translation to dynamics offers novel opportunities:

- ▶ Type dispatching on GADT constructors:

$$\mathit{pretty} (\mathit{Const} (x ::^{\mathcal{G}} [\beta])) = \mathit{prettyList} x$$

- ▶ Enforcing type constraints between GADTs

$$\mathit{teq} (\mathit{Const} (x ::^{\mathcal{G}} \beta)) (\mathit{Const} (y ::^{\mathcal{G}} \beta)) = \mathit{eq} x y$$

This requires a more elaborate translation