

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

International Symposium on
Implementation and Application of Functional Languages
September 23-25, 2009

Setting the scene

A large class of problems (e.g., structure editors) involves:

- ▶ Untyped input that is provided by a user
- ▶ Updating a typed heterogeneous structure accordingly

Setting the scene

A large class of problems (e.g., structure editors) involves:

- ▶ Untyped input that is provided by a user
- ▶ Updating a typed heterogeneous structure accordingly

Dynamic types

- ▶ Wrap user input uniformly

Generalised algebraic datatypes

- ▶ Heterogeneous structures

Setting the scene

A large class of problems (e.g., structure editors) involves:

- ▶ Untyped input that is provided by a user
- ▶ Updating a typed heterogeneous structure accordingly

Dynamic types

- ▶ Wrap user input uniformly
- ▶ Full support in Clean
- ▶ Minimal support in Haskell, only monomorphic values

Generalised algebraic datatypes

- ▶ Heterogeneous structures

Setting the scene

A large class of problems (e.g., structure editors) involves:

- ▶ Untyped input that is provided by a user
- ▶ Updating a typed heterogeneous structure accordingly

Dynamic types

- ▶ Wrap user input uniformly
- ▶ 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

Setting the scene

A large class of problems (e.g., structure editors) involves:

- ▶ Untyped input that is provided by a user
- ▶ Updating a typed heterogeneous structure accordingly

Dynamic types

- ▶ Wrap user input uniformly
- ▶ 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

Ideally, updating involves Clean's dynamics and Haskell's GADTs

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:

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

Dynamic types - context-dependent

Values can be wrapped depending on the context:

$$\begin{aligned} \text{wrap} &:: \alpha \rightarrow \text{Dynamic} \mid \text{TC } \alpha \\ \text{wrap } x &= \mathbf{dynamic} \ x \end{aligned}$$

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

$$\begin{aligned} \text{unwrap} &:: \text{Dynamic} \rightarrow \alpha \mid \text{TC } \alpha \\ \text{unwrap } (x :: \alpha^\wedge) &= x \\ \text{unwrap } _ &= \text{error "incorrect type"} \end{aligned}$$

Dynamic types - context-dependent

Values can be wrapped depending on the context:

$$\begin{aligned} \text{wrap} &:: \alpha \rightarrow \text{Dynamic} \mid \text{TC } \alpha \\ \text{wrap } x &= \mathbf{dynamic} \ x \end{aligned}$$

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

$$\begin{aligned} \text{unwrap} &:: \text{Dynamic} \rightarrow \alpha \mid \text{TC } \alpha \\ \text{unwrap } (x :: \alpha^\wedge) &= x \\ \text{unwrap } _ &= \text{error "incorrect type"} \end{aligned}$$

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:

$apply :: Dynamic \rightarrow Dynamic \rightarrow Maybe \beta \mid TC \beta$

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

$apply _ _ = Nothing$

Dynamic types - scoping and laziness

Scope of dynamic type variables is the complete function arm:

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

This enforces type equality between the two arguments:

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

Dynamic types - scoping and laziness

Scope of dynamic type variables is the complete function arm:

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

This enforces type equality between the two arguments:

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

Dynamics preserve lazy behaviour of functional programs:

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

Generalised algebraic datatypes

Haskell supports heterogeneous structures via GADTs:

data *Lam* :: * → * **where**

Undef :: *Lam* α

Const :: α → *Lam* α

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

term :: *Lam* Int

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

Generalised algebraic datatypes

Haskell supports heterogeneous structures via GADTs:

```
data Lam :: * → * where  
  Undef :: Lam α  
  Const :: α → Lam α  
  App   :: Lam (α → β) → Lam α → Lam β  
  
term :: Lam Int  
term = App (Const abs) (Const 1)
```

Additional type 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:

$$update :: Lam\ \alpha \rightarrow Path \rightarrow Dynamic \rightarrow Lam\ \alpha$$

Goal

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

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

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

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

Goal

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

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

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

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

Goal

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

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

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

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

Goal

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

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

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

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

Goal

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

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

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

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

**A natural synergy between polymorphic types in
Clean's dynamics and Haskell's GADTs**

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*

Decorate the new value with a type representation:

$$\text{update}_R :: \text{Lam}_R \alpha \rightarrow \text{Path} \rightarrow (\beta, \text{Rep } \beta) \rightarrow \text{Lam}_R \alpha$$

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*

Decorate the new value with a type representation:

$$update_R :: Lam_R \alpha \rightarrow Path \rightarrow (\beta, Rep \beta) \rightarrow Lam_R \alpha$$

Also include type representation in GADT to enforce type equality:

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

$Undef_R :: Lam_R \alpha$

$Const_R :: (\alpha, Rep \alpha) \rightarrow Lam_R \alpha$

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

Conventional approach - type representations

Type representations are defined by another GADT:

```
data Rep :: * → * where  
  RInt  :: Rep Int  
  RBool :: Rep Bool  
  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  
  RBool :: Rep Bool  
  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  
  RBool :: Rep Bool  
  RFun  :: Rep  $\alpha$  → Rep  $\beta$  → Rep ( $\alpha$  →  $\beta$ )
```

Equality of types is enforced by constructing a proof:

```
data Equal :: * → * → * where  
  Refl :: Equal  $\alpha$   $\alpha$ 
```

The proof is provided by point-wise comparison:

```
eqRep :: Rep  $\alpha$  → Rep  $\beta$  → Maybe (Equal  $\alpha$   $\beta$ )
```

Conventional approach - decorating functions

The update function operates on the decorated type:

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

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:

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

case eqRep rx ry of

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

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

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

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 Haskell's GADTs to relate types to Clean's dynamics:

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

The synergy

Annotate Haskell's GADTs to relate types to Clean's dynamics:

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

Comparing this approach to the conventional approach:

- ▶ Original datatype is used in the function definition
- ▶ The $::^+$ annotation provides access to GADT type information
- ▶ Type equality proofs provided by dynamic pattern match

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 $::^+$ 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 $Lam_T :: * \rightarrow *$ **where**

$Undef_T :: Lam_T \alpha$

$Const_T :: \alpha \rightarrow Lam_T \alpha$

$App_T :: Lam (\alpha \rightarrow \beta) \rightarrow Lam \alpha \rightarrow Lam_T \beta$

Semantics - translating GADTs

Translate to extended GADT, living next to the original:

data $Lam_T :: * \rightarrow *$ **where**

$Undef_T :: Lam_T \alpha$

$Const_T :: (\alpha, Type \alpha) \rightarrow Lam_T \alpha \mid TC \alpha$

$App_T :: Lam (\alpha \rightarrow \beta) \rightarrow Lam \alpha \rightarrow Lam_T \beta$

Semantics - translating GADTs

Translate to extended GADT, living next to the original:

data $Lam_T :: * \rightarrow *$ **where**

$Undef_T :: Lam_T \alpha$

$Const_T :: (\alpha, Type \alpha) \rightarrow Lam_T \alpha \mid TC \alpha$

$App_T :: Lam (\alpha \rightarrow \beta) \rightarrow Lam \alpha \rightarrow Lam_T \beta$

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

type $Type \alpha = Dynamic$

Semantics - translating GADTs

Translate to extended GADT, living next to the original:

data $Lam_T :: * \rightarrow *$ **where**

$Undef_T :: Lam_T \alpha$

$Const_T :: (\alpha, Type \alpha) \rightarrow Lam_T \alpha \mid TC \alpha$

$App_T :: Lam (\alpha \rightarrow \beta) \rightarrow Lam \alpha \rightarrow Lam_T \beta$

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

type $Type \alpha = Dynamic$

Correctness by construction of phantom typed dynamic:

$typeOf :: Type \alpha \mid TC \alpha$

$typeOf = \mathbf{dynamic} \perp :: \alpha^\wedge$

Context of $typeOf$ determines the type that is stored

Semantics - converting GADTs

Define conversion from original to extended GADT:

$$\text{toLam}_{\mathcal{T}} :: \text{Lam } \alpha \rightarrow \text{Lam}_{\mathcal{T}} \alpha$$
$$\text{toLam}_{\mathcal{T}} \text{Undef} = \text{Undef}_{\mathcal{T}}$$
$$\text{toLam}_{\mathcal{T}} (\text{Const } x) = \text{Const}_{\mathcal{T}} (x, \text{typeOf})$$
$$\text{toLam}_{\mathcal{T}} (\text{App } f \ x) = \text{App}_{\mathcal{T}} f \ x$$

Semantics - converting GADTs

Define conversion from original to extended GADT:

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

The *TC* requirement of Const_T 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 $::^+$ annotations:

update $:: Lam \ \alpha \rightarrow Path \rightarrow Dynamic \rightarrow Lam \ \alpha$

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

update (*Const* (*x* $::^+ \beta$)) $[0] \ (y :: \beta) = Const \ y$

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

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

update *x* $_ \ _ = x$

Semantics - translating functions

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

$$\begin{array}{l} \text{update}_{\mathcal{T}} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \text{Dynamic} \rightarrow \text{Lam } \alpha \\ \text{update}_{\mathcal{T}} \text{ Undef} \quad \quad \quad [] \quad \quad _ \quad \quad = \text{Undef} \\ \text{update}_{\mathcal{T}} (\text{Const } (x \quad ::^+ \beta)) [0] \quad (y :: \beta) = \text{Const } y \\ \text{update}_{\mathcal{T}} (\text{App } f \ x) \quad \quad \quad (0 : \rho) \ y \quad \quad = \text{App } (\text{update } f \ \rho \ y) \ x \\ \text{update}_{\mathcal{T}} (\text{App } f \ x) \quad \quad \quad (1 : \rho) \ y \quad \quad = \text{App } f \ (\text{update } x \ \rho \ y) \\ \text{update}_{\mathcal{T}} \ x \quad \quad \quad _ \quad \quad _ \quad \quad = x \end{array}$$

Semantics - translating functions

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

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

Semantics - translating functions

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

$update_T :: Lam_T \alpha \rightarrow Path \rightarrow Dynamic \rightarrow Lam \alpha$

$update_T Undef_T \quad [] \quad - \quad = Undef$

$update_T (Const_T (x, (- :: \beta))) \quad [0] \quad (y :: \beta) = Const y$

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

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

$update_T x \quad - \quad - \quad = x$

Semantics - translating functions

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

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

Body of the function is unchanged, calling original function:

$$\begin{aligned} \text{update} &:: \text{Lam} \alpha \rightarrow \text{Path} \rightarrow \text{Dynamic} \rightarrow \text{Lam} \alpha \\ \text{update } x \ p \ d &= \text{update}_T (\text{toLam}_T x) \ p \ d \end{aligned}$$

The conversion is localized and not exposed in the signature

Conclusion

**A natural synergy between polymorphic types in
Clean's dynamics and Haskell's GADTs**

Conclusion

A natural synergy between polymorphic types in Clean's dynamics and Haskell's GADTs

New $::^+$ 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

A natural synergy between polymorphic types in Clean's dynamics and Haskell's GADTs

New $::^+$ 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:

$$\text{pretty} (\text{Const } (x ::^+ [\beta])) = \text{prettyList } x$$

- ▶ Enforcing type constraints between GADTs

$$\text{teq} (\text{Const } (x ::^+ \beta)) (\text{Const } (y ::^+ \beta)) = \text{eq } x \ y$$

This requires a more elaborate translation