WGT 2020 — Saturday, January 25th — New Orleans

Gradual Algebraic Data Types

Michael Greenberg

Pomona College

Stefan Malewski

University of Santiago of Chile

Éric Tanter

University of Chile

Friendship ended with HIGHER ORDER FUNCTIONS

~~`\f ->`~~

~~`(\x -> f (x x))`~~

~~`(\x -> f (x x))`~~

```
data Expr =
   Var VarName
  | Lam VarName Expr
  | App Expr Expr
  | Zero
  | Succ Expr
 deriving Eq
```

Now
Algebraic Data Types
are my best friend

~~`fold (+) 0`~~

Motivating Examples

```
(define (flatten x)
 (cond
  [(null? x) x]
  [(cons? x)
   (append (flatten (car xs))
           (flatten (cdr xs)))]
  [else (list xs)]))
```

```
<books>
 <book id="1" title="Solaris">
  <author name="Stanislaw Lem”
          age=“98" />
 </book>
 <book id="2" title="Foundation">
  <author name="Isaac Asimov” />
  <review>Best of the series!</review>
 </book>
</books>
```

```
[[Name "Stanislaw Lem", Age 98],
[Name "Isaac Asimov"]]
```

A Real-ish Model of ADTs

$G ::= B \mid G \to G \mid A$

$\Delta ::= \bullet \mid \Delta, A : C$

$\Xi ::= \bullet \mid \Xi, c : T \times \ldots \times T$

$c \in C$

$e ::= v \mid x \mid e\ e \mid c\ e \ldots e \mid$

$\quad \text{match}\ e\ \text{with}\ \{\ c\ x \ldots x \mapsto e; \ldots; c\ x \ldots x \mapsto e\ \}$

Adding the Gradual Type

$G ::= B \mid G \to G \mid A \mid ?$

$\Delta ::= \bullet \mid \Delta, A : C$

$\Xi ::= \bullet \mid \Xi, c : T \times \ldots \times T$

$c \in C$

$e ::= v \mid x \mid e\ e \mid c\ e \ldots e \mid$

$\qquad \text{match } e \text{ with } \{ c\ x \ldots x \mapsto e; \ldots; c\ x \ldots x \mapsto e \}$

Flatten

```
(define (flatten x)
 (cond
  [(null? x) x]
  [(cons? x)
   (append (flatten (car xs))
           (flatten (cdr xs)))]
  [else (list xs)]))
```

```
data List = Nil | Cons ? List

let flatten (l:?) =
 match l with
  | Nil => Nil
  | Cons v l' =>
    append (flatten v) (flatten l')
  | _ => Cons l Nil
 end
```

A Real-ish Model of ADTs

$G ::= B \mid G \to G \mid A$

$\Delta ::= \bullet \mid \Delta, A : C$

$\Xi ::= \bullet \mid \Xi, c : T \times \ldots \times T$

$c \in C$

$e ::= v \mid x \mid e\ e \mid c\ e \ldots e \mid$

$\quad \text{match } e \text{ with } \{\, c\ x \ldots x \mapsto e; \ldots; c\ x \ldots x \mapsto e \,\}$

Adding the Gradual Constructor

$G ::= B \mid G \rightarrow G \mid A \mid ?$

$\Delta ::= \bullet \mid \Delta, A : C \uplus \{¿\}$ $\Xi ::= \bullet \mid \Xi, c : T \times \ldots \times T$

$c \in C$ $g ::= c \mid ¿$

$e ::= v \mid x \mid e\ e \mid c\ e \ldots e \mid$

$\quad \text{match } e \text{ with } \{\, g\ x \ldots x \mapsto e; \ldots; g\ x \ldots x \mapsto e \,\}$

XML Processing

```
data Attribute = ¿

data XML = Text String | ¿

parseXML : String -> XML


let collectAuthors (xml : XML) : List =
 match xml with
 | Text str => Nil
 | Author attrs elems => Cons attrs Nil
 | ¿ attrs elems =>
  concat (map collectAuthors elems)
 end
```

```
<books>
 <book id="1" title="Solaris">
  <author name="Stanislaw Lem”
          age=“98" />
 </book>
 <book id="2" title="Foundation">
  <author name="Isaac Asimov” />
  <review>Best of the series!</review>
 </book>
</books>
```

```
[[Name "Stanislaw Lem", Age 98],
[Name "Isaac Asimov"]]
```

Just use CDuce?

```
(define (flatten x)
 (cond
  [(null? x) x]
  [(cons? x)
   (append (flatten (car xs))
           (flatten (cdr xs)))]
  [else (list xs)]))
```

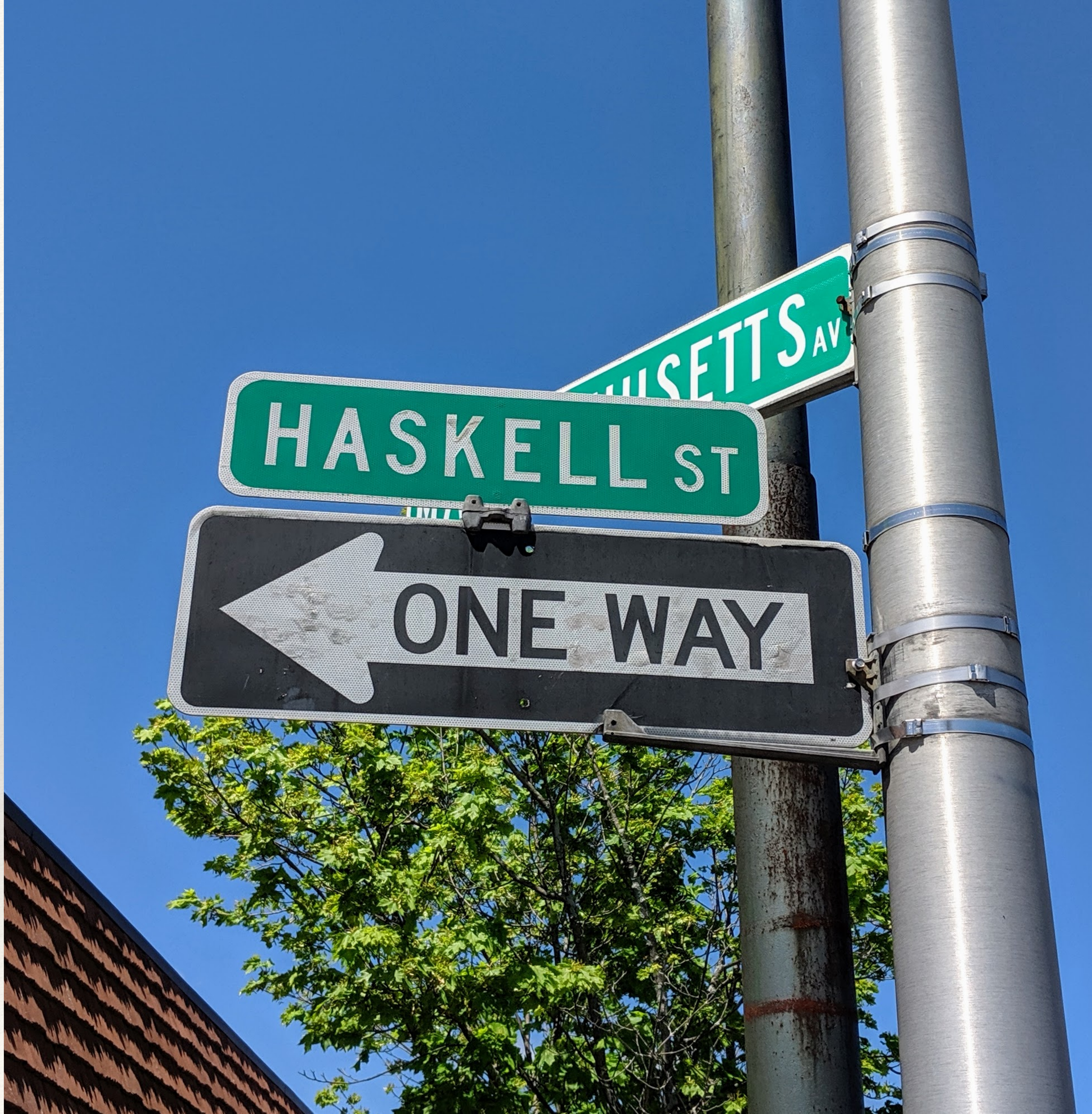
```
type Tree('a) =
 (‘a\[Any*]) | [ (Tree('a))* ]

let flatten ( (Tree('a)) -> ['a*])
  | [] -> []
  | [h ;t] -> (flatten h)@(flatten t)
  | x -> [x]
```

Just use Haskell?

```haskell
{-# LANGUAGE
   GADTs, TypeApplications, ScopedTypeVariables, ViewPatterns,
   PolyKinds, DataKinds
#-}
module Flatten where

import Data.Dynamic
import Type.Reflection

data MaybeMatch (a :: k1) (b :: k2) where
  Match :: MaybeMatch a a
  NoMatch :: MaybeMatch a b

isType :: forall a b. Typeable a => TypeRep b -> MaybeMatch a b
isType (eqTypeRep (typeRep @a) -> Just HRefl) = Match
isType _ = NoMatch

smartToDyn :: TypeRep a -> a -> Dynamic
smartToDyn (isType @Dynamic -> Match) x = x
smartToDyn rep                        x = Dynamic rep x
```

```haskell
flatten :: [Dynamic] -> [Dynamic]
flatten [] = []
flatten (dx@(Dynamic rep x):dxs) =
  x' ++ flatten dxs
  where
    x' | App (isType @[] -> Match) arg <- rep
       = flatten (map (smartToDyn arg) x)
       | otherwise
       = [dx]
```

Who wants this?



The Motivating Examples are Not Good Ones

- Haskell and CDuce can already write flatten
  - OCaml less easily
- XML processing in typed languages is well understood
- What's the point?
  - Expressivity!
  - Porting!
  - Interop?

See my SNAPL 2019 paper,
"The Dynamic Practice and Static Theory of Gradual Typing"

Desiderata and Key Questions

- Nominal-ish systems, like Haskell and OCaml
- What is a complete pattern match?
- What does it look like to name a constructor not statically included in any datatype? In construction? In pattern matching?
- What about models of nested matching? When should we communicate mismatched branch types to the programmer and when should they be coerced to the dynamic type?
- Who is this for?
  - Static FP folks are maybe not so interested—to their detriment!

Row types

- Coming up next: a nicely developed gradual interpretration of row types!
  - Hits lots of our desiderata!
  - Cool relationship to polymorphic variants!