LECTURE 07

Theory and Design of PL (CS 538)

February 12, 2020

MODELING DATATYPES

GENERAL PATTERN

1. Add a new type
2. Add constructor expressions
3. Add destructor expressions
4. Add typing rules for new expressions
5. Add evaluation rules for new expressions

EXAMPLE: PRODUCTS

INTRODUCING PRODUCTS

- Given types t_1 and t_2, product type $t_1 \times t_2$
- Can extend to triples: $t_1 \times (t_2 \times t_3)$, etc.
  - Often write just $t_1 \times t_2 \times t_3$
- Constructor
  - Pairing: from terms e_1 and e_2, form (e_1, e_2)

DESTRUCTORS: PROJECTIONS

- Given a pair term e, can project out terms:
  - First element: $\mathrm{p}_1(\mathrm{e})$
  - Second element: $\mathrm{p}_2(\mathrm{e})$

TYPING/EVALUATION RULES

EXAMPLE: SUM TYPES

INTRODUCING SUMS

- Given types t_1 and t_2, sum type $t_1 + t_2$
- Can extend to triples: $t_1 + (t_2 + t_3)$, etc.
  - Often write just $t_1 + t_2 + t_3$
- Constructors
  - Left injection: from term e_1, form $\mathrm{inl}(e_1)$
  - Right injection: from term e_2, form $\mathrm{inr}(e_2)$

DESTRUCTORS: CASE

- Given a sum term e, add case analysis expression:

$$\mathrm{case}(\mathrm{e})\ \mathrm{of}\ \mathrm{inl}(\mathrm{x}_1) \to \mathrm{e}_1 \mid \mathrm{inr}(\mathrm{x}_2) \to \mathrm{e}_2$$

- "If e is left option, do e_1. If e is right option, do e_2."
- Branch e_1 can use variable x_1, branch e_2 can use x_2

TYPING/EVALUATION RULES

EXAMPLE: LIST TYPES

INTRODUCING LISTS

- Given type t, have type $\mathrm{List}(\mathrm{t})$ of lists of t
- Constructors
  - Empty: (from nothing) form expression Nil
  - Cons: from term e and tail e', form $\mathrm{Cons}(\mathrm{e}, \mathrm{e}')$

CONSUMING LISTS

- Very much like sums
- Given a list term e, add case analysis expression:

$$\mathrm{case}(\mathrm{e})\ \mathrm{of}\ \mathrm{Nil} \to \mathrm{e}_1 \mid \mathrm{Cons}(\mathrm{x}, \mathrm{xs}) \to \mathrm{e}_2$$

- "If e is empty, do e_1. If e is not empty, do e_2."
- Branch e_2 can use variables x and xs

TYPING/EVALUATION RULES

PARAMETRIC FUNCTIONS

"FOR ALL" PARAMETERS

- We've already seen: types have *type variables*:

```
fst :: (a, b) -> a
Cons :: a -> [a] -> [a]
```

- These must work *for all* types `a`
- Concrete type inferred automatically when calling:

```
fst (1, True) :: Int        -- type param a is Int
Cons True []  :: List Bool -- type param a is Bool
```

MUST BEHAVE UNIFORMLY

- Function behavior can't depend on particular type!
  - No "peeking" at what type `a` is
  - Not allowed: if `a` is Bool then … if `a` is Int then …
- Also called *polymorphism* in type theory
  - Note: *not* the same as OO "polymorphism"

WHY IS THIS GOOD?

- Polymorphism *constrains* what a function can do
- More constraints:
  - More annoying
  - Fewer wrong implementations
- Sometimes, only one function is possible

FREE THEOREMS

- What does our mystery function do?

```
mystery1 :: a -> a
```

- Polymorphism: must work the *same way* for all `a`
- Can prove: it can only be the *identity* function
  - (Ignoring non-termination...)

```
mystery1 x = x
```

FREE THEOREMS

- What does our mystery function do?

```haskell
mystery2 :: (a, a) -> a
```

- Can prove: either always returns first, or second

```haskell
mystery2 (x, y) = x -- Possibility 1
mystery2 (x, y) = y -- Possibility 2
```

FREE THEOREMS

- What does our mystery function do?

```
mystery3 :: List a -> Maybe a
```

- If output is `Just x`, then `x` must be in input list
- Index can only depend on the *length* of the list

```
x1 = mystery3 [1, 2]
x2 = mystery3 ['a', 'b']
-- (x1, x2) is either:
-- (Nothing, Nothing), (Just 1, Just 'a'), (Just 2, Just 'b')
```

SOMETIMES: TOO LIMITING

WHAT TYPES?

- To string
  - `toString :: a -> String`
- Equality
  - `(==) :: a -> a -> Bool`
- Ordering
  - `(<) :: a -> a -> Bool`
- These polymorphic functions must *ignore* input(s)!

“AD HOC” POLYMORPHISM

- Same function name, works on different types
- Behavior can depend on the concrete type
- Can’t work on *all* types, but we should be able to easily extend function to handle new types

HASKELL'S SOLUTION: TYPECLASSES

DECLARING A TYPECLASS

- Give list of associated operations (*methods*)
- Example: Equality typeclass:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)
```

- Last two lines are *default implementations*
  - Defining either $==$ or $/=$ is enough

SUBCLASSING

- Some typeclasses require other typeclasses
- Example: Ordered type needs a notion of equality

```
class Eq a => Ord a where
  (<)  :: a -> a -> Bool
  (>)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
```

- Any type satisfying `Ord` needs to satisfy `Eq`
- Can require multiple parent typeclasses

TYPECLASS “CONSTRAINT”

- Functions can require type variables to be instances
- Add a “constraint” before the type signature

```
-- Can be applied as long as type `a` is an instance of `Eq a`
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = (x == y) || elem x ys

-- `(==)` function from `Eq` typeclass
```

- Define functions at the right level of generality!

A TOUR OF TYPECLASSES

SHOW AND READ

- `Show`: can be converted to a string

```
class Show a where
  show :: a -> String
```

- `Read`: can be converted from a string

```
-- Main useful function:
readMaybe :: Read a => String -> Maybe a
```

ENUM AND BOUNDED

- `Enum`: can be enumerated

```
class Enum a where
  toEnum   :: Int -> a
  fromEnum :: a -> Int
```

- `Bounded`: has max and min element

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

NUMERIC TYPECLASSES

- Most general is `Num`: things generalizing integers

```
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  abs :: a -> a                     -- absolute value
  negate :: a -> a                  -- negation
  signum :: a -> a                  -- sign: +1 or -1
  fromInteger :: Integer -> a
```

- More specific typeclasses: `Integral`, `Floating`
- Numeric hierarchy for number-like things

MONOID

- Monoid is a type with:
  - A binary operation
  - An identity for the operation
- Think: lists, with list append and empty list

```
class Monoid a where
  mempty  :: a               -- identity element
  mappend :: a -> a -> a -- binary operation
```

- Lots more in Haskell’s algebraic hierarchy

MAKING NEW TYPECLASS INSTANCES

DIRECT METHOD

- Provide concrete definitions for typeclass operations
- Supply enough for minimally complete definition
- Undefined things given default implementations

```
data Nat = Zero | Succ Nat

instance Ord Nat where
  Zero   < Zero   = False
  Succ _ < Zero   = False
  Zero   < Succ _ = True
  Succ n < Succ m = n < m

  Zero   <= _      = True
  Succ _ <= Zero   = False
  Succ n <= Succ m = n <= m
```

REQUIRE OTHER INSTANCES

- Often: defining instances for parametrized types
- Need to require type variables satisfy some instance

```haskell
-- Custom type of pairs
data MyPair a = MkPair a a

instance Show a => Show MyPair a where
  show (MkPair x x') = "MyPair of " ++ (show x)
                                    ++ " and " ++ (show x')

instance Ord a => Ord MyPair a where
  (MkPair x x') < (MkPair y y') = (x < y) || (x == y && x' < y')
```

AUTOMATIC METHOD

- Often: typeclass instances are boring (“boilerplate”)
  - Usually clear how to define `Eq` typeclass, …
- Have compiler derive default instances for you

```
data Nat = Zero | Succ Nat deriving (Eq)
data Colors = Red | Green | Blue deriving (Enum, Eq, Show)
```