Haskell: Types!

COS 441 Slides 4

Slide content credits:
Ranjit Jhala (UCSD)
Benjamin Pierce (UPenn)
Agenda

• Last time:
  – intro to Haskell
    • basic values: Int, Char, String, [a], ...
    • simple function definitions
      – key principle: abstract out repeated code
      – key principle: design for reuse
  – reasoning about Haskell programs
    • unfolding definitions
    • using simple laws of arithmetic or other facts/lemmas
    • induction for recursive programs
    • (re)folding definitions

• This time:
  – Haskell type definitions
    • key principle: a powerful way to define new abstractions
DEFINING NEW HASKELL TYPES
It is often convenient (and helps document a program) to give names to types:

```
type SquareT = (Float, Float, Float)
```

All type names (but not type variables) are capitalized.

```
(x,y)  \downarrow \downarrow \downarrow  x \quad y \quad s

\{ \quad \}   \rightarrow \quad s
```

```
(x,y)
```

```
type SquareT = (Float, Float, Float)
```
Type Synonyms

• It is often convenient (and helps document a program) to give names to types:

all type names (but not type variables) are capitalized

```
(type SquareT = (Float, Float, Float)

aSquare :: SquareT
aSquare = (2.0, 1.5, 3)

area :: SquareT -> Float
area (_, _, s) = s * s
```

• Using type names does not change the meaning of a program
  — SquareT is everywhere interchangeable with (Float, Float, Float)
Type Synonyms

• Adding circles:

  type SquareT = (Float, Float, Float)

  area :: SquareT -> Float
  area (_, _, s) = s * s

  type CircleT = (Float, Float, Float)

  circ :: CircleT
  circ = (3.0, 4.0, 6)
Type Synonyms

• Adding circles:

  \[
  \text{type } \text{SquareT} = (\text{Float}, \text{Float}, \text{Float})
  \]

  \[
  \text{area} :: \text{SquareT} \rightarrow \text{Float}
  \]

  \[
  \text{area} (_, _, s) = s \times s
  \]

  \[
  \text{type } \text{CircleT} = (\text{Float}, \text{Float}, \text{Float})
  \]

  \[
  \text{circ} :: \text{CircleT}
  \]

  \[
  \text{circ} = (3.0, 4.0, 6)
  \]

  \[
  \text{circA} = \text{area} \text{circ}
  \]
Type Synonyms

• Adding circles:

```haskell
type SquareT = (Float, Float, Float)

area :: SquareT -> Float
area (_, _, s) = s * s
```

```haskell
type CircleT = (Float, Float, Float)

circ :: CircleT
circ = (3.0, 4.0, 6)
```

```haskell
circA = area circ
```

Oops! Meant to work on squares! The type checker doesn’t alert us that we have violated our abstraction.

Said another way: Type synonyms don’t create enforced abstractions.
Data Types

- Data types create enforced data abstractions

```haskell
data CircleDataType = Circle (Float, Float, Float)
data SquareDataType = Square (Float, Float, Float)
```

- These declarations do three things:
  - create a new types called `CircleDataType` and `SquareDataType`
    - these types are different from any other type (and each other)
  - create constructors `Circle` and `Square`
    - the constructors are used to build new values with the type
  - create new patterns for deconstructing Circles and Squares
Data Types

data CircleDataType = Circle (Float, Float, Float)

data SquareDataType = Square (Float, Float, Float)

sq :: SquareDataType
sq = Square (2.0, 1.5, 3)

circ :: CircleDataType
circ = Circle (2.0, 1.5, 3)

area :: SquareDataType -> Float
area (Square (_, _, s)) = s * s

Constructors create protective wrappers.
Patterns unwrap data structures, allowing their contents to be used.
data CircleDataType = Circle (Float, Float, Float)

data SquareDataType = Square (Float, Float, Float)

sq :: SquareDataType
sq = Square (2.0, 1.5, 3)

circ :: CircleDataType
circ = Circle (2.0, 1.5, 3)

area :: SquareDataType -> Float
area (Square (_, _, s)) = s * s

circArea = area circ

myArea = area (3.0, 4.0, 5.0)

type mismatch: CircleDataType vs SquareDataType

type mismatch: (Float, Float, Float) vs SquareDataType
• Computing area properly:

```haskell
data CircleDataType = Circle (Float, Float, Float)
data SquareDataType = Square (Float, Float, Float)

areaSq :: SquareDataType -> Float
areaSq (Square (_, _, s)) = s * s

areaCirc :: CircDataType -> Float
areaCirc (Circle (_, _, r)) = pi * r * r
```

• That’s ok, but circles and squares are similar. There may be a lot of operations that are defined for both: area, grow, shrink, draw, move, ... can we define a **new, combined abstraction** for **shapes** that are either Circles or Squares?
• A shape abstraction:

```haskell
data SimpleShape =
    Circle (Float, Float, Float)
  | Square (Float, Float, Float)
```
A shape abstraction:

```haskell
data SimpleShape =  
    Circle (Float, Float, Float)  
    | Square (Float, Float, Float)

sq :: SimpleShape
sq = Square (1.1, 2.2, 3.3)

circ :: SimpleShape
circ = Circle (0.0, 0.0, 24)
```
• A shape abstraction:

```haskell
data SimpleShape =
    Circle (Float, Float, Float)
  | Square (Float, Float, Float)

sq :: SimpleShape
sq = Square (1.1, 2.2, 3.3)

circ :: SimpleShape
circ = Circle (0.0, 0.0, 24)

area :: SimpleShape -> Float
area (Square (_, _, s)) = s * s
area(Circle (_, _, r)) = pi * r * r
```
• Let’s develop some routines over a more general set of shapes. We will ignore the position of the shape for now and specify its dimensions only.

```haskell
data Shape =
    Rectangle Float Float
  | Ellipse Float Float
  | RtTriangle Float Float
  | Polygon [(Float, Float)]

Rectangle s1 s2 =

Ellipse r1 r2 =

RtTriangle s1 s2 =

Polygon [v1, ..., v5] =
```

\[
\begin{align*}
\text{Rectangle } & s1 & s2 = \\
\text{Ellipse } & r1 & r2 = \\
\text{RtTriangle } & s1 & s2 = \\
\text{Polygon } & v1, ..., v5 = \\
\end{align*}
\]
More General Shapes

• Type Synonyms improve documentation:

```haskell
data Shape =
  Rectangle Side Side
| Ellipse Radius Radius
| RtTriangle Side Side
| Polygon [Vertex]

type Side = Float
type Radius = Float
type Vertex = (Float, Float)
```

Rectangle s1 s2 =

Ellipse r1 r2 =

RtTriangle s1 s2 =

v1 = (1.0, 1.0)
...
v5 = (0.4, 0.4)

Polygon [v1, ..., v5] =
Computing Area:

```haskell
area :: Shape -> Float

area (Rectangle s1 s2) = s1 * s2
area (Ellipse r1 r2) = pi * r1 * r2
area (RtTriangle s1 s2) = s1 * s2 / 2
area (Polygon vs) = ... ?
```

```haskell
data Shape =
    Rectangle Side Side |
    Ellipse Radius Radius |
    RtTriangle Side Side |
    Polygon [Vertex]

type Side = Float
type Radius = Float
type Vertex = (Float, Float)
```
Computing Area

• How do we compute polygon area?
• For convex polygons:
  – Compute the area of the triangle formed by the first three vertices
  – Delete the second vertex to form a new polygon
  – Sum the area of the new polygon and the area of the triangle from the first step
Computing Area

\[
\text{area (Polygon (v1:v2:v3:vs))} = \text{triArea } v1 \ v2 \ v3 + \text{area (Polygon (v1:v3:vs))}
\]
\[
\text{area (Polygon _)} = 0
\]
Computing Area

area (Polygon (v1:v2:v3:vs)) = triArea v1 v2 v3 + area (Polygon (v1:v3:vs))
area (Polygon _) = 0

triArea :: Vertex -> Vertex -> Vertex -> Float
triArea v1 v2 v3 =
  let a = dist v1 v2
      b = dist v2 v3
      c = dist v3 v1
      s = 0.5 * (a + b + c)
  in
      sqrt (s * (s - a) * (s - b) * (s - c))

dist :: Vertex -> Vertex -> Float
dist (x1, y1) (x2, y2) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)
Computing Area: Alternatives

**Version 1:**

\[
\text{area (Polygon (v1:v2:v3:vs))} = \text{triArea v1 v2 v3} + \text{area (Polygon (v1:v3:vs))}
\]

\[
\text{area (Polygon _)} = 0
\]

**Version 2:**

\[
\text{area (Polygon (v1:vs))} = \text{polyArea vs}
\]

where

\[
\text{polyArea :: [Vertex] -> Float}
\]

\[
\text{polyArea (v2 : v3 : vs')} = \text{triArea v1 v2 v3} + \text{polyArea (v3:vs')}
\]

\[
\text{polyArea _} = 0
\]
Computing Area: Alternatives

Version 1:

\[
\text{area (Polygon (v1:v2:v3:vs))} = \text{triArea v1 v2 v3} + \text{area (Polygon (v1:v3:vs))}
\]

\[
\text{area (Polygon _)} = 0
\]

Version 2:

\[
\text{area (Polygon (v1:vs))} = \text{polyArea vs}
\]

where

\[
\text{polyArea :: [Vertex] \rightarrow Float}
\]

\[
\text{polyArea (v2 : v3 : vs')} = \text{triArea v1 v2 v3} + \text{polyArea (v3:vs')}
\]

\[
\text{polyArea _} = 0
\]

uses Polygon constructor at each recursive call

does not use Polygon at each recursive call
Computing Area: Alternatives

Version 1:
\[
\text{area (Polygon (v1:v2:v3:vs))} = \text{triArea v1 v2 v3} + \text{area (Polygon (v1:v3:vs))}
\]
\[
\text{area (Polygon _)} = 0
\]

Version 2:
\[
\text{area (Polygon (v1:vs))} = \text{polyArea vs}
\]

where
\[
\text{polyArea :: [Vertex] -> Float}
\]
\[
\text{polyArea (v2 : v3 : vs')} = \text{triArea v1 v2 v3} + \text{polyArea (v3:vs')}
\]
\[
\text{polyArea _} = 0
\]

prepends v1 on to list at each recursive call

does not prepend v1 on to list at each recursive call
Computing Area: Alternatives

Version 1:

\[
\text{area (Polygon (v1:v2:v3:vs))} = \text{triArea } v1 \ v2 \ v3 + \text{area (Polygon (v1:v3:vs))}
\]

\[
\text{area (Polygon _)} = 0
\]

Version 2:

\[
\text{area (Polygon (v1:vs))} = \text{polyArea } vs
\]

where

\[
\text{polyArea :: [Vertex] -> Float}
\]

\[
\text{polyArea (v2 : v3 : vs')} = \text{triArea } v1 \ v2 \ v3 + \text{polyArea (v3:vs')}
\]

\[
\text{polyArea (Polygon _)} = 0
\]
Computing Areas: Alternatives

• Summary of differences:
  – A small decrease in readability for a small increase in efficiency

• Usually, a bad trade!
  – Machines are fast
  – Programmers are slow
  – We should be optimizing for programmer speed first!
  – Moreover, programmers are terrible at predicting which optimizations matter in real programs

• Moral:
  – write code that is manifestly correct
  – use the scientific method to optimize:
    • measure performance
    • tune bottlenecks as needed
  – if performance is way out of line, you may need completely different algorithms; minor tweaks won’t get it done
• Consider the following session in the ghci interpreter:

```haskell
data Foo = Bar | Baz
```

```
Prelude> :l badData
[1 of 1] Compiling Main             ( badData.hs, interpreted )
Ok, modules loaded: Main.
*Main> Bar

<interactive>:1:1:
  No instance for (Show Foo)
      arising from a use of `print'
Possible fix: add an instance declaration for (Show Foo)
In a stmt of an interactive GHCi command: print it
```
One Last Note: The Fix

- Write “deriving (Show)” after each data definition to enable printing (ie, “show”ing):

```haskell
badData.hs:
data Foo = Bar | Baz deriving (Show)
```

shell:

```haskell
*Main> :l badData
[1 of 1] Compiling Main  ( badData.hs, interpreted )
Ok, modules loaded: Main.
*Main> Bar
Bar
```

hooray!!
SUMMARY!
• Type definitions
  – type T = ... creates a type synonym
    • no enforced abstraction, but useful documentation
  – data T = ... creates a new abstract type
    • enforced abstraction
    • defines: new type, new constructors, new patterns
    • can include many variants

• Premature optimization may be harmful
  – think carefully about your high-level algorithm first
  – write the clearest code that implements your algorithm directly
  – use the scientific method
    • measure performance and optimize if and where necessary