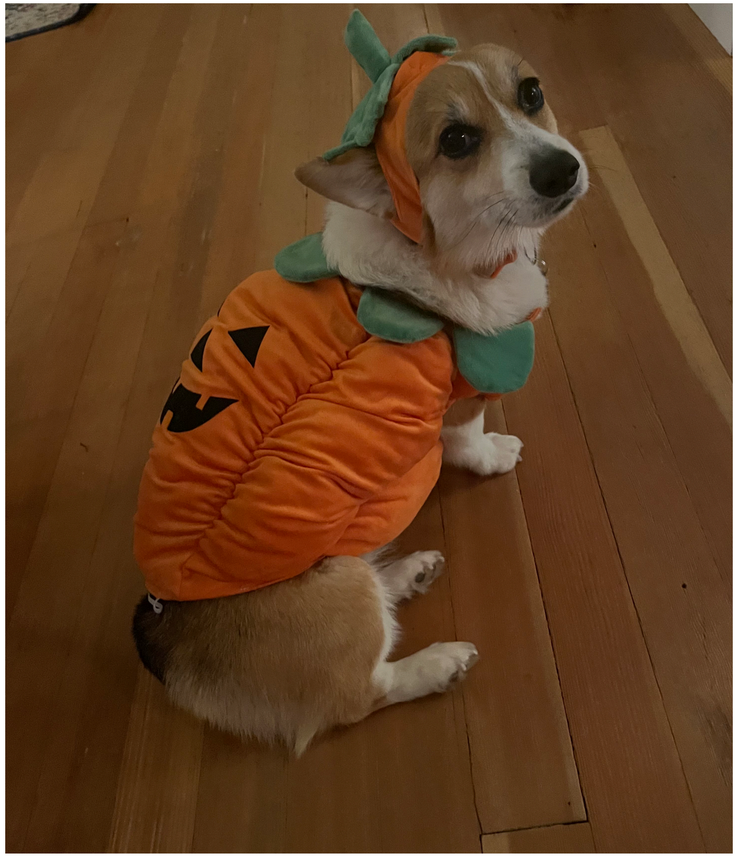
CMPT 383

Lecture 12: Haskell to Rust



Anders Miltner

- What is the Rust analogue for various bits of Haskell syntax?

Integers and Floats and More

Haskell

```
i :: Int
i = 3

f :: Float
f = 3.0
```

Rust

```
const INTVALUE: i8 = 3

const FLOATVALUE: f32 = 3.0

Const UNSIGNEDINT: u8 = 3
```

ALSO i16, i32, i64, i128

ALSO f64

ALSO u16, u32, u64, u128

Example 1, Average

Haskell

```
average :: Float -> Float -> Float
average x1 x2 = (x1 + x2) / 2
```

Rust

```
fn average(x1: f32, x2: f32) -> f32 {
    (x1+x2)/2.0
}
```

Functions!

Haskell

```
functionName :: InputType1 -> InputType2 -> OutputType
functionName inputArg1 inputArg2 = expressionToOutput
```

Rust

```
fn functionName(inputArg1: inputType1, inputArg2:
inputType2) -> outputType {
    expressionToOutput
}
```

Can also return expressionToOutput;

Example 1.5, Average

Haskell

```haskell
average :: Float -> Float -> Float
average x1 x2 =
  let sum = x1 + x2 in
  sum / 2.0
```

Rust

```rust
fn average(x1: f32, x2: f32) -> f32 {
    let sum = x1 + x2;
    sum / 2.0
}
```

HOFs are easy in Haskell

- Also easy in Rust
- But let’s wait on them for a bit…

Variables in Rust

- Can be mutable
- But that’s discussed on Friday
- For now, we can treat the same as Haskell

Characters!

- The type “Char” is inhabited by Unicode characters
- Chars are 4 bytes of data
- Chars are exactly u8
- Built-in functions (and more) described here: https://doc.rust-lang.org/std/primitive.char.html

Character Example

```
x :: Char
x = 'x'

-- >>> isUpper x
-- False
```

Booleans!

- The type “Bool” is inhabited by 2 values: true and false
- There’s a number of useful operations on bools:

Basic Boolean Operations

```
>>> True & False
False

>>> False | True
True

>>> !False
True

>>> 2 == 2
True

>>> False && True
False

>>> True || False
True
```

|| and &&

- Lazy | and &
- (Also | and & are defined on more general values)
  - | and & on ints is the bitwise or and bitwise and
- Don’t evaluate the second argument unless you have to

Basic Branching

```
fn func(x1: f32, x2: f32) -> f32 {
    if x1 == 0.0 {
        1.0
    }
    else {
        2.0
    }
}
```

Pattern Matching

```
fn func(x1: i32, x2: i32) -> f32 {
    match x1 {
        0 => 1.0,
        _ => 2.0
    }
}
```

- Evaluate to a value
- Go from top to bottom and see what pattern is hit first
- _ means anything goes

The Power of Pattern Matching

- Pattern matching is an *incredibly* powerful tool
- Hopefully you don’t need this spiel again :)

What can you pattern match?

- “What *can’t* you pattern match?” is a better question
  - The answer to which is “Functions” — kinda
- Any value that isn’t a function, you can match on
  - NEW: Rust. You also can’t match on Floats!
    - Probably due to how horrible Floats are

All Types:

- Base Types
- Tuples / Structs
- Enums
- Arrays (but we’ll get there later)

Making Tuples!

- It’s pretty difficult, get ready
- x = (y,z)
- If y has type t1, and z has type t2
- x has type (t1,t2)

Destructing Tuples

- The exact same as Haskell
- Pattern match them!

```
fn func((x1,x2): (i32,i32)) -> f32 {
    match x1 {
        0 => 1.0,
        _ => 2.0
    }
}
```

```
fn func(x: (i32,i32)) -> f32 {
    let (x1,x2) = x;
    match x1 {
        0 => 1.0,
        _ => 2.0
    }
}
```

Defining Structs

- Same as Haskell records
- Same as tuples under-the-hood
- Can reference parts of the tuple by name

```
struct Point {
  x: i32,
  y: i32,
  z: i32
}
```

Destructing Structs

```
struct Point {
  x: i32,
  y: i32,
  z: i32
}
```

```
let z = Point {x:1,y:2,z:3};
return z.x  //returns 1
```

```
let z = Point {x:1,y:2,z:3};
let Point (x:zx,y:zy,z:zz) = z
return zz // returns 3
```

Lists!

- Implemented in Rust as Vectors!
- Also known as ArrayLists in other languages
    - In other words, resizeable arrays

Creating Vectors

```
let v1: Vec<i32> = Vec::new();

let v2: Vec<i32> = vec![1,2,3]
```

Using Vectors

- Elements retrieved via indexing

```
let v2: Vec<i32> = vec![1,2,3]

let q = v2[1] //q is set to 2
```

- Elements set via indexing (must be mutable, see Friday)

```
let mut v2: Vec<i32> = vec![1,2,3]

v2[1] = 100 //v2 is now [1,100,3]
```

- Elements added via push (must be mutable, see Friday)

```
let mut v2: Vec<i32> = vec![1,2,3]

v2.push(4); // v2 is now [1,2,3,4]
```

Enums

- Like Haskell’s algebraic data types
- Funky things happen with recursive enums, so we’ll ignore them for now

Enum Definition

```
enum IntOrChar {
  Int(i32),
  Char(char)
}
```

Constructing Enums

```
enum IntOrChar {
  Int(i32),
  Char(char)
}
```

```
x = IntOrChar::Char('c');

y = IntOrChar::Int(2);
```

Extracting Data from Enums

- Pattern matching of course!

```
enum IntOrChar {
  Int(i32),
  Char(char)
}
```

```
x = IntOrChar::Char(‘c’);

let z = match x {
    IntOrChar::Int(i) => ‘z’
    IntOrChar::Char(c) => c
}; // z is set to ‘c’
```

Nice Pattern Matching Syntactic Sugar!

```
x = IntOrChar::Char(‘z’);

if let IntOrChar::Char(c) = x {
  print!(“{}”,c)
}; //prints z
```

Traits!

- Traits are type classes
- Basically exactly, there’s not really any difference except for maybe some low-level implementation details

Defining Traits

Haskell

```
class Size a where
  getSize :: a -> Int
```

Rust

```
trait Size {
  fn getSize(&self) -> i32;
}
```

Defining Traits

Haskell

```
class Default a where
  default :: a
```

Rust

```
trait Default {
  fn default() -> Self;
}
```

Implementing Traits

```
trait Size {
  fn getSize(&self) -> i32;
}
```

```
impl Size for IntOrChar {
    fn getSize(&self) -> i32 {
      match self {
        IntOrChar::Int(_) => 32,
        IntOrChar::Char(_) => 8
      }
    }
}
```

Implementing Traits

```
trait Default {
  fn default() -> Self;
}
```

```
impl Default for IntOrChar {
    fn default() -> {
      IntOrChar::Int(0)
    }
}
```