Programming Abstractions

Lecture 36: Types

Stephen Checkoway

Static vs. Dynamic types

Dynamically-checked types

Dynamically-typed languages tag all of their values at runtime

- In Racket, we can ask what the type of a value is: `number?`, `list?`, `pair?`, `boolean?`, etc.

Functions are forced to check that the types of their input match the expected type

Scheme and Python are examples of dynamically-typed languages

What does this code do?

```
(define (mul x y)
  (if (= x 0)
      0
      (* x y)))

(mul 0 'blah)
```

A. Syntax error

B. Contract violation

C. Runtime error

D. Warning about `'blah`

E. Returns 0

Run-time type checks

No explicit error checking

```
(define (mul x y)
  (if (= x 0)
      0
      (* x y)))

(mul 10 'blah)
```

This gives a contract error:

```
*: contract violation
  expected: number?
  given: 'blah
```

Note that the contract error is on `*`, not `mul`

Run-time type checks

Explicit error checking

```
(define (mul x y)
  (cond [(not (number? x)) (error 'mul "not a number: ~s" x)]
        [(not (number? y)) (error 'mul "not a number: ~s" y)]
        [(= x 0) 0]
        [else (* x y)]))
```

```
(mul 0 'blah)
```

This gives a non-contract error:

```
mul: not a number: blah
```

Brief aside: Contracts

A contract is a predicate that declares some fact about a value that must be true

- `number?` — the value is a number
- `list?` — the value is a list
- `positive?` — the value is positive
- `real?` — the value is a real number
- `any/c` — every value satisfies this contract
- `pair?` or `cons?` — the value is a cons cell

Contract combinators

We can make complex contracts using combinators

- `(and/c c1 c2 ... cn)` — creates a contract that is satisfied only when all of the `ci` contracts are satisfied
- `(or/c c1 c2 ... cn)` — creates a contract that is satisfied if any of the `ci` contracts are satisfied
- `(not/c c)` — creates a contract that's satisfied if and only if contract `c` is not satisfied
- `(listof c)` — creates a contract that the value is a list, each of whose elements satisfy `c`
- `(-> c1 c2 ... cn c-result)` — creates a function contract

Contracts on functions

```
(-> arg1-contract ... argn-contract result-contract)
```

Specifies contracts for arguments

Specifies a contract for the return value

The runtime checks the contract on calls to functions to ensure arguments satisfy the contract

On returns, the runtime checks that the result value satisfies the contract

Run-time type checks

Contracts

```
(define/contract (mul x y)
  (-> number? number? number?)  ; x, y, and return value are numbers
  (if (= x 0)
      0
      (* x y)))
(mul 0 'blah)
```

This gives a contract error:

```
mul: contract violation
  expected: number?
  given: 'blah
  in: the 2nd argument of
      (-> number? number? number?)
```

Consider the function `(first lst)`, which contract best describes the `first` function?

A. `procedure?`

B. `(-> list? any/c)`

C. `(-> (not/c empty?) any/c)`

D. `(-> (and/c list? (not/c empty?)) any/c)`

Downside of dynamic-typing

Errors like passing and returning the wrong types of values are not caught until run time, even with contracts

```
(define/contract (collatz n)
  (-> (and/c positive? integer?) (listof integer?))
  (cond [(= n 1) 1]
        [(odd? n) (cons n (collatz (add1 (* 3 n))))]
        [else (cons n (collatz (/ n 2)))]))
```

This has a type error, but it won't be caught until runtime

collatz: broke its own contract
  promised: list?
  produced: '(4 2 . 1)

Statically-checked types

Statically-typed languages compute a static approximation of the runtime types

The type of an expression is computed from the types of its sub expressions

This can be used to rule out a whole class of type errors at compile time

C, Java, Rust, and Haskell are examples of statically-typed languages

Revisiting our buggy collatz function

```
#lang plai-typed

(define (collatz [n : number]) : (listof number)
  (cond [(= n 1) 1]
        [(odd? n) (cons n (collatz (add1 (* 3 n))))]
        [else (cons n (collatz (/ n 2)))]))
```

At compile time, we get an error

```
typecheck failed: number vs. (listof number) in: …
```

Quick Haskell introduction

Haskell

Functional programming language

Statically-typed with a really strong type system

Lazy: Values are not computed until they're needed

- We won't need this today but one consequence is there isn't really a distinction between a stream and a list as lists are lazily evaluated
- Can make reasoning about code run time a bit difficult

Pure: functions cannot have side effects like printing output or mutation (i.e., no set!)

Indentation is important 😞

- Haskell programs have a 2D layout constraint which is…unusual

Arithmetic, booleans, lists

Aarithmetic works the way you'd like it to (mostly)!

- `3 + 10`
- `(x + 8) * y / 5`

Boolean values `True` and `False`

- Numeric comparisons ==, `/=`, <, >, etc. return `True` and `False`

Lists are homogeneous (meaning all elements have the same type)

- `[1, 2, 3]` —3-element list
- `[]` — empty list
- `[True, False, False, True]` — 4-element list
- `[True, 1]` — type error!

Function application

Rather than `(foo x y z)` we just write `foo x y z`

We use parentheses for grouping

```
ghci> not True
False
ghci> div 7 3
2
ghci> mod 7 3
1
ghci> mod 7 3 + 5
6
ghci> mod 7 (3 + 5)
7
```

Types

`expr :: type`

Every expression has a type

We can be explicit about the type of the expression by writing down its type

```
ghci> (False || True && False) :: Bool
False
ghci> (5 + 3) :: Int
8
```

Defining a function

```
add1 :: Int -> Int
add1 x = x + 1

fib :: Int -> Int
fib n = if n < 2
        then n
        else fib (n - 1) + fib (n - 2)

ghci> [0..10]
[0,1,2,3,4,5,6,7,8,9,10]
ghci> map fib [0..10]
[0,1,1,2,3,5,8,13,21,34,55]
```

Multiple argument functions

Average of two integers (as an integer, rounding down)

```
average :: Int -> Int -> Int
average x y = div (x + y) 2
```

The unusual type `Int -> Int -> Int` can be read one of two equivalent ways

- `average` takes two `Int` arguments and returns an `Int`
- `Int -> Int -> Int` is the same as `Int -> (Int -> Int)` which says average take an `Int` argument and returns a function of type `Int -> Int`
  - This is called Currying (named for mathematician Haskell Curry) and it's pretty cool
  - average 5 returns a one-argument function which computes the average of its argument and 5

Algebraic Data Types (ADTs)

Algebraic data types

Algebraic data types let us create types from other types

Two ways to combine types:

- Product types: these are tuples (think Cartesian products)
  - `(Int, Bool, String)` is a product type where every value is a tuple containing an int, a boolean, and a string, e.g., `(275, True, "Hello")`
- Sum types (or variant types): every value must be exactly one of the possible variants

We can combine product and sum types, most commonly as a sum of products

ADTs in Haskell

Tuples we can just use directly

```
splitListAt :: Int -> [Int] -> ([Int], [Int])
splitListAt n xs = (take n xs, drop n xs)
```

```
ghci> splitListAt 5 [0..15]
([0,1,2,3,4],[5,6,7,8,9,10,11,12,13,14,15])
```

Named product types

We define a new named type using the data keyword

```
data Foo = Foo String ([Int] -> Int) Bool
```

This defines a new data type named Foo

This is a product type consisting of a `String`, a function that takes a list of `Int`s and produces an `Int`, and a `Bool`

The pink `Foo` is a **constructor** and it's how we construct values of type `Foo`

- The constructor didn't need to match the name of the type; we'll see examples shortly
- `Foo "hi" length False`
  has type `Foo`

Aside

In order to print out our new types, we need append the line

```
  deriving (Show)
```

to our data type definition

```
data Foo = Foo String ([Int] -> Int) Bool
  deriving (Show)
```

I'm going to omit this line from all the examples

Sum types

```
data Bool = True
          | False
```

This is the standard definition of the Bool type

`True` and `False` are 0-argument constructors that create values of type `Bool`

When a type has one constructor, it's common for the constructor's name to match the type name

When a type has multiple constructors (i.e., it's a sum type), then the constructor names describe the variants rather than the overarching data type