CSCC24 – Principles of Programming Languages

Anya Tafliovich[1]

[1]with thanks to S.McIlraigh, G.Penn, P.Ragde

motivation

Consider the following Scheme function definition:

```
(define foobar
  (lambda (x)
    (if (even? x)
        (car x)
        (cdr x))))
```

Anything wrong?

`even?`	expects a number and
`car, cdr`	expect a pair

$\Longrightarrow$ we'll get an error **at run-time**.

We are able to figure it out before run-time. Can't a PL support this kind of reasoning?

motivation

- It could be worse!
- At least Scheme checks that `x` is a pair before accessing memory in the execution of `(cdr x)`.
- In fact, Scheme is type safe: it will never execute `(op arg)` if `op` is not applicable to `arg`.
- Languages which are not type safe (e.g., C) allow unsafe memory accesses: a major source of security vulnerabilities.

motivation

- Want to catch all errors at compile-time.
  - Extremely difficult, if not impossible.
- Identify a certain class of errors and guarantee to catch all of them.
- Catch as many errors as possible?
  - Costly: programmers' expertise, computational resources, etc.
  - Slower development, less expressive languages.
  - Used in development of safety-critical systems.
- Compromise: type systems.
  - Guarantees range widely: from basic safety of memory accesses to just about anything you want.

typing

A <u>type</u> is:

“A name for a set of values and some operations which can be performed on that set of values.”

“A collection of computational entities that share some common property.”

typing

Some examples of types (in Haskell notation):

- `Int` : the integers
- `Integer` : the integers, unbounded
- `Foat, Double` : the real floating point numbers, single and double precision
- `Char` : the characters
- `Bool` : the booleans (`True` and `False`)
- `Int` $\rightarrow$ `Bool` : the functions that take integers as input and return booleans as output.

What constitutes a type is language dependent.

typing

Benefits of having a type system:

- Easier to debug programs: compiler can catch many errors.
- Static analysis: a lot of useful information about the program can be obtained at compile-time.
- Efficiency: typing can be used by the compiler to generate quicker code.
- Correctness: typing can be used (by the programmer or by the compiler) to prove correctness of code.
- Documentation: types declare your intent with well-chosen names.

typing

A programming language is type safe if no program is allowed to violate its type distinctions.

The process of verifying and enforcing the constraints of types is called type checking.

Type checking can either occur at compile-time (static type checking) or at run-time (dynamic type checking).

static vs dynamic typing

Dynamic type checking:

- Performed at run-time.
- Slower execution: need to carry type information around, lots of run-time checks.
- More flexible.
- Easier refactoring.

Static type checking:

- Faster execution.
- Compiler can do a lot of optimization.
- Some argue that resulting programs are safer.
- Some argue that resulting programs are more elegant and modular.
- Some argue that programmers will write horrible code to get around a static type-checker.

static typing

<u>Explicit</u> static typing: code contains type annotations.

- For example, in Java:
  - Variable declarations
    `int x,y,z;`
  - Function headers
    `public static void main(String[] arg)`

<u>Type inference</u>: infer all types from the code that does not contain explicit type annotations.

- `foo (x, y, z) = if x then "hello, " ++ y else "hi, " ++ z`
- `x` must be boolean
- `y, z` must be strings
- the return value is a string
- `foo :: (Bool, [Char], [Char]) -> [Char]`

Haskell data types

Basic types:

- `()` — the only member is `()`.
- `Bool` — booleans.
- `Int, Integer` — integers.
- `Float, Double` — reals.
- `Char` — characters.

More types:

- $(\langle\texttt{type}_0\rangle, \langle\texttt{type}_1\rangle, \ldots, \langle\texttt{type}_n\rangle)$ — tuples.
- $[\langle\texttt{type}\rangle]$—: lists.
- $\langle\texttt{input-type}\rangle \to \langle\texttt{output-type}\rangle$ — functions.

Haskell basic types

(): (called "unit") this type has only one element ()

```
prompt> ()
()
prompt> :type ()
() :: ()
prompt> :t ()
() :: ()
prompt>
```

Reading the above snippet of the GHCi interpreter session:

- evaluate (), please
- what is the type of ()?
- what is the type of () (and I don't like typing)?

Haskell basic types

Bool: this type has two elements: `True` and `False`.

Operations on `Bool`s: `not`, `&&`, `||`, `if`, ...

```
prompt> True && False
False
prompt> True || False
True
prompt> not True
False
prompt> if False then 'a' else 'b'
'b'
```

Haskell basic types

Lots of number types (more on this later): `Int`, `Integer`, `Float`, `Double`

```
prompt> 3.14 + 3.14 ** 2 - 42
-29.0004
prompt> 3.14 + 3.14 ** 2 - 42 > -30
True
```

Read the documentation for all that is available for number types.

Haskell basic types

Char: $\{$ 'a', 'b', 'c', ... $\}$

Haskell data types

More types:

- $\texttt{(}\langle\texttt{type}_0\rangle, \langle\texttt{type}_1\rangle, \ldots, \langle\texttt{type}_n\rangle\texttt{)}$ — tuples.
- $\texttt{[}\langle\texttt{type}\rangle\texttt{]}$ — lists.
- $\langle\texttt{input-type}\rangle \to \langle\texttt{output-type}\rangle$ — functions.

Haskell types

A **tuple** packs together several types.

```
prompt> ('a','b',True)
('a','b',True)
prompt> :t ('a','b',True)
('a','b',True) :: (Char, Char, Bool)
```

Haskell types

In Haskell all elements in a **list** must have the same type.

```
prompt> [1,2,3]
[1,2,3]
prompt> ['a','b','c']
"abc"
prompt> [True,False]
[True,False]

prompt> [[42],[24,3]]
[[42],[24,3]]

prompt> [True,False,"nono"]
<interactive>...:
 Couldn't match expected type 'Bool' with actual type '[Char]'
 In the expression: "nono"
 In the expression: [True, False, "nono"]
 In an equation for 'it': it = [True, False, "nono"]
```

`[]` is the empty list.

Constructor: `:`

More operations: `++`, `null`, `length`, `map`, `foldr`, `...`

Haskell types — list

Some examples:

```
prompt> 1 : [2,3]
[1,2,3]
prompt> [1,2] ++ [2,3]
[1,2,2,3]
prompt> length [1,2]
2
prompt> map abs [1,-2,-3,4]
[1,2,3,4]
prompt> null [1,2]
False
prompt> null []
True
prompt> foldr (+) 0 [1,2,3] # more on this (+) later
6
```

Haskell syntax

A variable declaration in Haskell looks like:

- ⟨`name`⟩ `=` ⟨`expr`⟩
- `x = 42 + 24`
- `(define x (+ 42 24))`

Haskell functions

The syntax for anonymous **functions** in Haskell is:

- `\⟨arg⟩ -> ⟨body⟩`
- `\x -> x + 1`
- `(lambda (x) (+ x 1))`

Giving a name to a function:

- `⟨name⟩ = \⟨arg⟩ -> ⟨body⟩`
- `inc = \x -> x + 1`
- `(define inc (lambda (x) (+ x 1)))`

Or:

- `⟨name⟩ ⟨arg⟩ = ⟨body⟩`
- `inc x = x + 1`
- `(define (inc x) (+ x 1))`

In Haskell every function accepts exactly one argument.This argument could be a tuple.

Haskell types

The **type of a function** is determined by the type of the input and the type of the output.
Some examples:

```
prompt> implies (x, y) = if x then y else True
prompt> :t implies

implies :: (Bool, Bool) -> Bool

prompt> greet x = "hello, " ++ x
prompt> :t greet

greet :: [Char] -> [Char]

prompt> maybeGreet (g,x) =
            if g then "hello, " ++ x else x
prompt> :t maybeGreet

maybeGreet :: (Bool, [Char]) -> [Char]
```

parametric polymorphism

What is the type of `\x -> x`?

```
prompt> :t (\x -> x)
(\x -> x) :: t -> t
```

```
prompt> id =  \x -> x
prompt> id 324
324
prompt> id "foo"
"foo"
prompt> id [1,2,3]
[1,2,3]
prompt> :t (id "foo")
(id "foo") :: [Char]
```

parametric polymorphism

```
prompt> id = \x -> x
prompt> :t id
id :: t -> t
```

`t` is a type variable.

`id` is a polymorphic function.

$\alpha \to \alpha$ means "for every valid type α, $\alpha \to \alpha$"

$$\forall\alpha \cdot \alpha \to \alpha$$

When `id` is applied to `324`, the type variable α is instantiated to `int`.

parametric polymorphism

Examples:

```
choose (a,b,c) = if a then b else c
choose ::  (Bool, t, t) -> t
```

```
swap (x,y) = (y,x)
swap ::  (t1, t) -> (t, t1)
```

parametric polymorphism

What is the type of the following function?

```
repeatTwice lst =
  if null lst then lst
  else [head lst,head lst] ++ repeatTwice (tail lst)
```

```
repeatTwice ::  [a] -> [a]
```

List is a polymorphic data type.

pattern matching

Value declaration (general form): `<pat> = <exp>`

```
prompt> (foo,bar) = ('a','b')
prompt> foo
'a'
prompt> bar
'b'
prompt> [foo,bar] = ["foo","bar"]
prompt> foo
"foo"
prompt> bar
"bar"
prompt> x : xs = [1,2,3]
prompt> x
1
prompt> xs
[2,3]
```

pattern matching

"_" is "don't care": matches everything, binds nothing

```
prompt> (_,x,y) = (1,2,3)
prompt> x
2
prompt> y
3

prompt> [x,_,z] = [1,2,3]
prompt> x
1
prompt> z
3
```

pattern matching

Function declaration with pattern matching:

```
<name> <pattern1> = <exp1>
<name> <pattern2> = <exp2>
   .
   .
   .
<name> <patternN> = <expN>
```

This means: the function name is `name`. It takes one argument (as any other Haskell function). It tries to match the argument to `pattern1`. If it succeeds, it returns value of `exp1`. Otherwise, tries to match the argument to `pattern2`. Etc, etc.

pattern matching

For example we can (and definitely should!) rewrite `repeatTwice` to use pattern matching:

```
repeatTwice [] = []
repeatTwice (x : xs) = x : x : repeatTwice xs
```

```
len [] = 0
len (x : xs) = 1 + len xs
```

Even better:

```
len [] = 0
len (_ : xs) = 1 + len xs
```

pattern matching

Function `firstlist` takes a list of pairs and returns the list consisting of the first elements only. For example:

```
firstlist [] ==> []
firstlist [(1,2),(1,3)] ==> [1,1]
firstlist [(1,"a"),(2,"b"),(3,"c")] ==> [1,2,3]
firstlist [([],"a"),([1],"b"),([1,2],"c")] ==>
    [[],[1],[1,2]]
```

```
firstlist [] = []
firstlist ((a,_) : rest) = a : firstlist rest
firstlist :: [(a, b)] -> [a]
```

Haskell notes

```
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
```

Let:

```
reverse xs =
    let
        rev ([], rs) = rs
        rev (x : xs, rs) = rev (xs, x : rs)
    in rev (xs, [])
```

Haskell notes

Where:

```
reverse xs = rev (xs, []) where
        rev ([], rs) = rs
        rev (x : xs, rs) = rev (xs, x : rs)
```

Guards:

```
abs x = if x >= 0 then x else -x

abs x | x >= 0 = x
      | otherwise = -x
```

type classes

So, what is the type of 42?

```
42 :: Num a => a
```

What does this mean?
`Num` is a type class.

`Num a` is a type class constraint.

`Num a => a` means some type a in the class `Num`.

And what is the `Num` class? All types in this class must implement addition, subtraction, multiplication, negation, absolute value, and other. See Haskell documentation for details.

type classes

Type classes offer a controlled approach to overloading.

There are a number of predefined type classes: `Eq`, `Ord`, `Show`, `Read`, `Num`, and more.

You can create instances of these classes.

You can also create your own classes and instantiate them.

(These are not like Python/Java classes. More like Java interfaces.)

type classes

The Eq type class is all types with equality defined.

Types in this class provide == and /=.

```
member (_, []) = False
member (y, x : xs) = x == y || member (y, xs)
```

```
prompt> :type member
member :: (Eq t) => (t, [t]) -> Bool
```

(Eq t) is a type class constraint.

All the base types (Int, Bool, etc.) are members of Eq.

Much more on type classes later.

currying

- the process of transforming a function that takes multiple arguments in a tuple as its argument, into a function that takes just a single argument and returns another function which accepts further arguments, one by one, that the original function would receive in the rest of that tuple. [HaskellWiki]

- the technique of converting a function that takes multiple arguments into a sequence of functions that each take a single argument. [Wikipedia]

- A function that "takes two arguments and returns the result" is a function that "takes one arguments and returns a function that takes one argument and returns the result".

- `(a,b) -> c` is transformed to `a -> b -> c`

currying

- `(a,b) -> c` is transformed to `a -> b -> c`

- `(a,b,c) -> d` is transformed to `a -> b -> c -> d`

- Notice that -> is right associative and, accordingly, function application is left-associative. That is,
  - `a -> b -> c` is the same as `a -> (b -> c)` , and
  - `f x y` is the same as `(f x) y`

currying

With named functions:

```
sum (x,y) = x + y          -- not curried version
sum' x y = x + y           -- curried version
```

```
prompt> :t sum
```

```
sum :: Num a => (a, a) -> a
```

```
prompt> :t sum'
```

```
sum' :: Num a => a -> a -> a
```

currying

Currying gives us partial application:

```
sum' x y = x + y          -- curried version
sum' :: Num a => a -> a -> a

prompt> sum' 2 3
5

prompt> sum' 2            -- partial application

sum' 2 :: Num a => a -> a

add2 = sum' 2
add2 :: Num a => a -> a

prompt> add2 3
5
```

currying

With anonymous functions:

```
prompt> :t (\x -> \y -> x + y)

(\x -> \y -> x + y) :: Num a => a -> a -> a

prompt> :t (\x -> \y -> x + y) 42

(\x -> \y -> x + y) 42 :: Num a => a -> a
```

currying

Built-in `curry` and `uncurry`:

```
curry :: ((a, b) -> c) -> a -> b -> c
```

```
prompt> member
member :: Eq t => (t, [t]) -> Bool
prompt> curry member
```

```
curry member :: Eq t => t -> [t] -> Bool
prompt> memberC = curry member
prompt> memberC 1 [1, 2, 3]
True
```

currying

Built-in `curry` and `uncurry`:

```
uncurry :: (a -> b -> c) -> (a, b) -> c

prompt> sum'
sum' :: Num a => a -> a -> a
prompt> uncurry sum'

uncurry sum' :: Num c => (c, c) -> c
prompt> sum'' = uncurry sum'
prompt> sum'' (2, 3)
5
```