CSCI 275: Programming Abstractions

Lecture 14: Types & Computation

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Questions? Concerns?

Functional Language of the Week: Haskell

- Haskell was first released in 1990, started in 1987

- Language developed "by committee"

"The committee's primary goal was to design a language that satisfied these constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages."

https://www.haskell.org/onlinereport/preface-jfp.html

Functional Language of the Week: Haskell

- Seen as a test bed for a lot of advanced PL features
  - The GHC (Glasgow Haskell Compiler) specifically has made a lot of innovations in compilers
- Its logo is a lambda! Described as a ”an advanced, purely functional programming language”
- Haskell operates with a lazy semantics (sometimes referred to as *call-by-need* semantics) – this is different than what Racket and most languages use, stay tuned!

Functional Language of the Week: Haskell

Implementations from https://en.wikipedia.org/wiki/Haskell

```
factorial :: (Integral a) => a -> a

-- Using recursion (with the "ifthenelse" expression)
factorial n = if n < 2
              then 1
              else n * factorial (n - 1)

-- Using recursion (with pattern matching)
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- Using a list and the "product" function
factorial n = product [1..n]

-- Using fold (implements "product")
factorial n = foldl (*) 1 [1..n]
```

If you're interested, Simon Peyton Jones (main lead of the Haskell compiler) hour long talk on Haskell history: https://www.youtube.com/watch?v=re96UgMk6GQ

Types Continued

Which of the calls below will fail the type checker?

```
(: bsum (-> (Listof Number) Number))
(define (bsum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (bsum (rest lst)))]))

(: csum (-> (Listof Integer) Integer))
(define (csum lst)
  (foldr + 0 lst))

(bsum (list 1 2 3 4)) ;A
(bsum (list 1.1 2.2 3.3 4.4)) ;B
(csum (list 1 2 3 4)) ;C
(csum (list 1.1 2.2 3.3 4.4)) ;D
```

E. None of the above

Type Checking in Racket

```
(: bsum (-> (Listof Number) Number))
(define (bsum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (bsum (rest lst)))]))

(: csum (-> (Listof Integer) Integer))
(define (csum lst)
  (foldr + 0 lst))

(bsum (list 1 2 3 4)) ;A
(bsum (list 1.1 2.2 3.3 4.4)) ;B
(csum (list 1 2 3 4)) ;C
(csum (list 1.1 2.2 3.3 4.4)) ;D
```

```
Welcome to DrRacket, version 8.5 [cs].
Language: typed/racket, with debugging; memory limit: 128 MB.
Type Checker: type mismatch
  expected: Integer
  given: Positive-Float-No-NaN in: 1.1
Type Checker: type mismatch
  expected: Integer
  given: Positive-Float-No-NaN in: 2.2
Type Checker: type mismatch
  expected: Integer
  given: Positive-Float-No-NaN in: 3.3
Type Checker: type mismatch
  expected: Integer
  given: Positive-Float-No-NaN in: 4.4
Type Checker: Summary: 4 errors encountered in:
  1.1
  2.2
  3.3
  4.4
```

Notice even though D throws the error, we do not get any output from the previous three calls

Typed Racket includes a Type Checking Pass before evaluation!

Typed Racket

- Basic types like `Number`
- Function types like `(: negate (-> Integer Integer))`
- Type constructors like `(Listof Boolean)`
- Union types like `(U False (Listof Number))`

Creating your own types

Writing out type annotations is something we do a lot

AND

We probably want to be able to make new types for new data, etc

```
(define-type N3N (-> Number Number Number))

(define-type FalseNum (U False (Listof Number))
```

Reminder: Tree definition

; Definition of tree datatype

```
(struct tree (value children) #:transparent)
```

; An empty tree is represented by null

```
(define empty-tree null)
```

; `(empty-tree? empty-tree)` returns `#t`

```
(define empty-tree? null?)
```

; Convenience constructor

; (make-tree v c1 c2 ... cn) is equivalent to

; (tree v (list c1 c2 ... cn))

```
(define (make-tree value . children)
  (tree value children))
```

Reminder: variadic function!

How do we create a typed `Number` tree?

Reminder, the untyped version:

```
(struct tree (value children) #:transparent)
```

A.
```
(struct tree ([value: Number]
       [children: (Listof tree)]))
```

B.
```
(struct tree ([value: Number]
          [children: (Listof Number)]))
```

C.
```
(struct tree ([value: Number] [children: Number]))
```

D.
```
(struct tree ([value children] : Number))
```

E. Something else

Reminder of our leaf checker below. What type is it?

```
(define (leaf? t)
  (cond [(empty-tree? t) #f]
        [else (empty? (tree-children t))]))
```

```
A. (: leaf? (-> tree tree))
```

```
B.(: leaf? (-> Boolean tree))
```

```
C.(: leaf (-> tree Boolean))
```

```
D.(: leaf (-> tree False))
```

E. Something else

Types for Variadic Functions

Specifies the type of the remaining arguments

```
(: make-tree (->* (Number) #:rest tree tree))
(define (make-tree value . children)
  (tree value children))
```

Reminder: variadic function!

Now we can *enforce* numeric trees!

```
(define T1 (make-tree 50))
(define T2 (make-tree 22))
(define T3 (make-tree 10))
(define T6 (make-tree 73 T1 T2 T3))

(define T4 (make-tree 'a)) ✖
```

```
Welcome to DrRacket, version 8.5 [cs].
Language: typed/racket, with debugging; memory limit: 128 MB.
 Type Checker: type mismatch
  expected: Number
  given: 'a in: (quote a)
>
```

Recursive Types

Struct typing is a special case of *Recursive Types*

We can define the `tree` type by saying that the children is of type “list of trees”

However, we cannot do something like

```
(define-type forest (U Number forest))
```

Type defined completely by its own definition

Types, Leveled Up

Assume we write 2 variants of the `member` procedure: one for Numbers, one for Strings. They have the type signatures:

```
(: nmem (-> Number (Listof Number)
            (U False (Listof Number))))
(: smem (-> String (Listof String)
            (U False (Listof String))))
```

Which of the following is true?

A. `nmem` and `smem` probably use the type of the arguments in their implementations
B. `nmem` and `smem` probably do *not* use the type of the arguments in their implementations
C. `nmem` and `smem`’s type signatures have the same general structure
D. More than one of the above
E. None of the above

We want a type signature for a general member!

```
(: nmem (-> Number (Listof Number)
            (U False (Listof Number))))
(: smem (-> String (Listof String)
            (U False (Listof String))))
```

↓

```
(: mem (-> X (Listof X)
           (U False (Listof X))))
```

Parametric Polymorphism

Polymorph – ”Many forms”

Typed Racket (and many functional languages!) support **parametric polymorphism**

This allows us to write code *without knowing the actual type of the arguments*

parametric!

Thanks to TAPL by Pierce and Steve Chong https://groups.seas.harvard.edu/courses/cs152/2015sp/lectures/lec14-polymorphism.pdf

Parametric Polymorphism in Typed Racket

Typed Racket introduces the `All` type constructor

`All` takes a list of type variables and a body type – the type variable can be *free* in the body of the type

So for a general length method, we would get the type

```
(: length (All (A) (-> (Listof A) Integer)))
```

If this is the polymorphic type for `length`:

```
(: length (All (A) (-> (Listof A) Integer)))
```

what is it for our generic `mem` member procedure?

A.
```
(: mem (-> A (Listof A)
          (U False (Listof A))))
```

B.
```
(: mem (-> Number (Listof A)
          (U False (Listof A))))
```

C.
```
(: mem (All (A) (-> A (Listof A)
          (U False (Listof A)))))
```

D. Something else

Other Types of Polymorphism

Always good to use an adjective when you're discussing polymorphism for this reason!

You likely have encountered other kinds of polymorphism!

Subtype Polymorphism: if you define a procedure for a Number, you can use it for a Float or an Integer as well ("subsumption rule")

Ad-hoc Polymorphism: you can use the + operator on Strings and on Integers. You can also overload + for your own class! (this *looks* like polymorphism, but is many implementations)

Thanks to TAPL by Pierce and Steve Chong https://groups.seas.harvard.edu/courses/cs152/2015sp/lectures/lec14-polymorphism.pdf

Fun Facts

Java Generics are an implementation of parametric polymorphism using wildcards

This is a **new feature in Java, relatively speaking:** it was only added in 2004 and is based on decades of research by the PL community on generics in Java

Thanks to TAPL by Pierce and Steve Chong https://groups.seas.harvard.edu/courses/cs152/2015sp/lectures/lec14-polymorphism.pdf

Taming Wildcards in Java's Type System *

Ross Tate
University of California, San Diego
rtate@cs.ucsd.edu

Alan Leung
University of California, San Diego
aleung@cs.ucsd.edu

Sorin Lerner
University of California, San Diego
lerner@cs.ucsd.edu

Abstract

Wildcards have become an important part of Java's type system since their introduction 7 years ago. Yet there are still many open problems with Java's wildcards. For example, there are no known sound and complete algorithms for subtyping (and consequently type checking) Java wildcards, and in fact subtyping is suspected to be undecidable because wildcards are a form of bounded existential types. Furthermore, some Java types with wildcards have no joins, making inference of type arguments for generic methods particularly difficult. Although there has been progress on these fronts, we have identified significant shortcomings of the current state of the art, along with new problems that have not been addressed.

In this paper, we illustrate how these shortcomings reflect the subtle complexity of the problem domain, and then present major improvements to the current algorithms for wildcards by making slight restrictions on the usage of wildcards. Our survey of existing Java programs suggests that realistic code should already satisfy our restrictions without any modifications. We present a simple algorithm for subtyping which is both sound and complete with our restrictions, an algorithm for lazily joining types with wildcards which addresses some of the shortcomings of prior work, and techniques for improving the Java type system as a whole. Lastly, we describe various extensions to wildcards that would be compatible with our algorithms.

https://rosstate.org/publications/tamewild/tamewild-tate-pldi11.pdf

Fun Facts

Java Generics are an implementation of parametric polymorphism using wildcards

This is a **new feature in Java, relatively speaking:** it was only added in 2004 and is based on decades of research by the PL community on generics in Java

The classic model for parametric polymorphism is called System F (this was developed in the *1970s*)

Thanks to TAPL by Pierce and Steve Chong https://groups.seas.harvard.edu/courses/cs152/2015sp/lectures/lec14-polymorphism.pdf

Type-Related Algorithms

- Types give us additional functionality and the ability to do better error detection
- We would need some additional tools/time to go into these ideas in proper detail 😟

Type Checking

Are these types consistent?

Type Inference

Can I guess types in a consistent way?

Facts about Type-Related Algorithms

- Robin Milner won the Turing Award in 1991 partially for building “ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism”
  - The most well-known type inference algorithm is called **Hindley-Milner type inference**

- Type inference in the full parametric polymorphism environment we talked about is undecidable

Type Inference Limits in Typed Racket

Typed Racket in it's "Caveats and Limitations" notes "Typed Racket's local type inference algorithm is currently not able to infer types for polymorphic functions that are used on higher-order arguments that are themselves polymorphic."

Example that doesn't type check:

```
(map cons '(a b c d) '(1 2 3 4))
```

`map` is polymorphic and `cons` is too - too much polymorphism!