CSC324 Lecture 17

24 July 2020

Rachit Nigam
@notypes

Those who love Racket are doomed to recreate it in every language they program in.

Tweet übersetzen

1:28 nachm. · 22. Juli 2020 · Twitter Web App

Announcements

- Posted the revised asn2 question 9 (without the solution this time!)
  - There's an updated `partc_tests.rkt` as a result.
- We will do the original question 9 in lab next week.

Last time...

- We discussed the **typing rules** for a simple statically-typed language

Today:

- 🧙‍♂️ The simply typed lambda calculus 🧙‍♂️
- Wrapping up sum types
- **Polymorphism** and **type variables**

The simply-typed lambda calculus

Last time we talked about typing various expressions (and special forms like `if` and value constructors like `add1`) but we are missing something important: implementing arbitrary functions! We'll do this by giving the lambda calculus types, yielding the **simply-typed lambda calculus.**

The **simply-typed lambda calculus** is "simply" the lambda calculus you've already seen before, but with typing rules to formalise:

- The types of functions
- The types of function arguments, and the types of function bodies

The simply-typed lambda calculus

The **simply-typed lambda calculus** is "simply" the lambda calculus you've already seen before, but with typing rules to formalise:

- The type of all functions
- The types of function arguments, and the types of function bodies

The simply-typed lambda calculus

What are the things that we can do with the lambda calculus?

- Define **function abstractions**
- Perform **function application**

Let's show typing rules for these two operations.

A typing rule for function abstractions

Let's say we have some terms of certain types

```
t1 : T1
t2 : T2

What can we say about the type of the expression
```

```
(λ (t1) t2)
```

A typing rule for function abstractions

Let's say we have some terms of certain types

```
t1 : T1
t2 : T2
```

```
What can we say about the type of the expression
```

```
(λ (t1) t2)
```

$$\frac{t1{:}T1 \qquad t2{:}T2}{}$$

A typing rule for function abstractions

Let's say we have some terms of certain types

```
t1 : T1
t2 : T2
```

```
What can we say about the type of the expression
```

```
;; ??? -> ???
(λ (t1) t2)
```

$$\frac{t1:T1 \qquad t2:T2}{}$$

A typing rule for function abstractions

Let's say we have some terms of certain types

```
t1 : T1
t2 : T2
```

```
What can we say about the type of the expression
```

```
;; T1 -> T2
(λ (t1) t2)
```

$$\frac{t1:T1 \qquad t2:T2}{\quad}$$

A typing rule for function abstractions

Let's say we have some terms of certain types

```
t1 : T1
t2 : T2
```

```
What can we say about the type of the expression
```

```
;; T1 -> T2
(λ (t1) t2)
```

$$\frac{t1{:}T1 \qquad t2{:}T2}{(\lambda\ (t1)\ t2) : T1 \to T2}$$

A typing rule for function abstractions

Let's say we have some terms of certain types

```
t1 : T1
t2 : T2
```

What can we say about the type of the expression

```
;; T1 -> T2
(λ (t1) t2)
```

$$\frac{\texttt{t1:T1}\qquad\texttt{t2:T2}}{\texttt{(}\lambda\texttt{ (t1) t2) : T1->T2}}$$

Formally, -> designates a function type.

By creating this typing rule, we have formalised the use of the arrow symbol in a typing rule.

It's good to remember that, even though we're using formal logic to express types, we can pull in all the tools in our formal logic toolbox to answer questions about the typedness of a term.

If I have some function abstraction

`(λ (x) b): R`

where `x: T1`, what can I say about the type of the whole fn abstraction (designated `R` in this example)?

$$\frac{t1 : T1 \qquad t2 : T2}{(\lambda\ (t1)\ t2) : T1 \to T2}$$

If I have some function abstraction

```
(λ (x) b): R
```

where `x: T1`, what can I say about the type of the whole fn abstraction (designated `R` in this example)?

We know that R has to be a function type, so it will have an arrow…

```
??? -> ???
```

$$\frac{\texttt{t1:T1} \qquad \texttt{t2:T2}}{\texttt{(λ (t1) t2) : T1->T2}}$$

If I have some function abstraction

```
(λ (x) b): R
```

where `x: T1`, what can I say about the type of the whole fn abstraction (designated `R` in this example)?

The problem gives us T1...

```
T1 -> ???
```

```
   t1:T1    t2:T2
------------------------
(λ (t1) t2) : T1->T2
```

If I have some function abstraction

```
(λ (x) b): R
```

where `x: T1`, what can I say about the type of the whole fn abstraction (designated `R` in this example)?

The problem gives us T1...

```
T1 -> ???
```

What else do we know?

$$\frac{t1:T1 \qquad t2:T2}{(\lambda\ (t1)\ t2)\ :\ T1\text{->}T2}$$

If I have some function abstraction

```
(λ (x) b): R
```

where `x: T1`, what can I say about the type of the whole fn abstraction (designated `R` in this example)?

```
T1 -> ???
```

We also know that b has some type…

```
b : B
```

$$\frac{t1:T1 \quad\quad t2:T2}{(\lambda\ (t1)\ t2) : T1 \to T2}$$

If I have some function abstraction

`(λ (x) b): R`

where `x: T1`, what can I say about the type of the whole fn abstraction (designated `R` in this example)?

`T1 -> B`

`for some type B that satisfies:`

`b : B`

$$\frac{\texttt{t1:T1} \qquad \texttt{t2:T2}}{\texttt{(λ (t1) t2) : T1->T2}}$$

If I have some function abstraction

`(λ (x) b): R`

where `x: T1`, what can I say about the type of the whole fn abstraction (designated `R` in this example)?

Existential quantification!

`T1 -> B`

`for some type B that satisfies:`

`b : B`

$$\frac{\texttt{t1:T1} \qquad \texttt{t2:T2}}{\texttt{(λ (t1) t2) : T1->T2}}$$

A typing rule for function application

Now let's do the opposite:

If I have `f`, with type T1->T2, and I perform function application with something of type T1, what type does the application evaluate to?

A typing rule for function application

Now let's do the opposite:

If I have `f`, with type T1->T2, and I perform function application with something of type T1, what type does the application evaluate to?

$$\frac{\texttt{f: T1->T2} \qquad \texttt{t:T1}}{\quad}$$

A typing rule for function application

Now let's do the opposite:

If I have `f`, with type T1->T2, and I perform function application with something of type T1, what type does the application evaluate to?

$$\frac{\texttt{f: T1->T2} \quad \texttt{t:T1}}{\texttt{(f t) : ???}}$$

A typing rule for function application

Now let's do the opposite:

If I have `f`, with type T1->T2, and I perform function application with something of type T1, what type does the application evaluate to?

$$\frac{\texttt{f: T1->T2} \quad \texttt{t:T1}}{\texttt{(f t) : T2}}$$

What do we get from these typing rules?

Recall that we can treat these typing rules either as mathematical formalisms, or to imagine type systems as **"compile-time embedded programming languages"**.

By describing product types formally last class (ie. `T1xT2`), we saw how:

- `x` describes a new mathematical formalism that forms a type from two types
- `x` is "an infix binary operator" in our "programming language of types"

What do we get from these typing rules?

Recall that we can treat these typing rules either as mathematical formalisms, or to imagine type systems as **"compile-time embedded programming languages"**.

Similarly, the simply-typed lambda calculus gives us -> and a way to combine and decompose terms that contain it.

$$\frac{\texttt{t1:T1} \quad \texttt{t2:T2}}{(\lambda\ \texttt{(t1) t2) : T1->T2}}$$

$$\frac{\texttt{f: T1->T2} \quad \texttt{t:T1}}{\texttt{(f t) : T2}}$$

Typing rules for sum types

Last time, we motivated sum types in terms of their cardinality, but we didn't discuss the practicalities of their typing rules. Let's do that now.

We'll talk through a few possible ways of typing sum types, not because one of them is strictly better than the others, but because viewing them through different lenses opens up new ways of thinking about them.

Typing rules for sum types, attempt 1...

The first rule says that if t is of type `T`, then it can be "lifted" into a sum type with another type `T'`

$$\frac{\texttt{t: T}}{\texttt{t: T+T'}}$$

$$\frac{\texttt{t: T}}{\texttt{t: T'+T}}$$

We have a problem though…

...what the heck is `T'`???

Typing rules for sum types, attempt 2...

OK, so why don't we assume that we have some t' that is "in scope"? Then the logical implication works fine.

$$\frac{t : T \quad t' : T'}{t : T + T'}$$

$$\frac{t : T \quad t' : T'}{t : T' + T}$$

This is a bit constricting, though; we can only define a `Wednesday` as a Day so long as we have all the other Days to hand to the typechecker

Typing rules for sum types, attempt 2...

This is what Haskell makes us do, though: we define a sum type "all at once" so all the `T'` types are enumerated.

$$\frac{t:\ T \qquad t':T'}{t:\ T+T'}$$

$$\frac{t:\ T \qquad t':T'}{t:\ T'+T}$$

Day.hs

```haskell
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday deriving (Show)
```

Typing rules for sum types, attempt 2...

This suggests that the definition of a Haskell sum type is **closed**; the type is not extensible elsewhere in the codebase

$$\frac{t\colon T \qquad t'\colon T'}{t\colon T+T'}$$

$$\frac{t\colon T \qquad t'\colon T'}{t\colon T'+T}$$

Day.hs

```haskell
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday deriving (Show)
```

Typing rules for sum types, attempt 2.5...

Are sum types like restricted inheritance? My example in yesterday's class seemed to suggest a connection…

Industrial languages like Scala and Kotlin, under the hood, implement sum types in terms of a class hierarchy.

```
interface Nat {};

class Zero implements Nat {
    public Zero() { }

    public String toString() {
        return "Zero";
    }
}

class Add1 implements Nat {
    private Nat n;

    public Add1(Nat n) {
        this.n = n;
    }

    public String toString() {
        String rec = this.n.toString();
        return "(Add1 " + rec + ")";
    }
}
```

Typing rules for sum types, attempt 2.5...

By contrast, this "sum type" is said to be **open**; there is nothing preventing another piece of the program from adding another child class to this "sum type".

(Note: Some languages have the notion of a **sealed class**, which is a class that can't be extended.)

```java
interface Nat {};

class Zero implements Nat {
    public Zero() { }

    public String toString() {
        return "Zero";
    }
}

class Add1 implements Nat {
    private Nat n;

    public Add1(Nat n) {
        this.n = n;
    }

    public String toString() {
        String rec = this.n.toString();
        return "(Add1 " + rec + ")";
    }
}
```

Typing rules for sum types, attempt 3...

Could there be a "for all T" quantifier?

Maybe! Adding a universal quantifier makes sense given that we're already operating in the realm of formal logic

$$\frac{t\colon T}{\forall T',\ t\colon T+T'}$$

$$\frac{t\colon T}{\forall T',\ t\colon T'+T}$$

Our favourite combination of sum and product types

What could typing rules for lists look like?

```
; A list of Nats is:
; - 'empty, or
; - (cons x xs), where:
;   - x is a Nat, and
;   - xs is a list of Nats
```

Our favourite combination of sum and product types

What could typing rules for lists look like?

```
; A list of Nats is:
; - 'empty, or
; - (cons x xs), where:
;   - x is a Nat, and
;   - xs is a list of Nats
```

'empty: List

cons: List

$$\frac{\texttt{t1:Nat} \quad \texttt{t2: List}}{\texttt{(cons t1 t2): List}}$$

$$\frac{\texttt{t: cons}}{\texttt{(car t) : Nat}}$$

$$\frac{\texttt{t: cons}}{\texttt{(cdr t) : List}}$$

The algebraic laws for lists

Remember this, from way back in Lecture 3?

```
For all x and y,
```

- `(car (cons x xs)) ≡ x` *(the law of car)*
- `(cdr (cons x xs)) ≡ xs` *(the law of cdr)*

The type view

We've seen how to demonstrate that the **law of car** and the **law of cdr** holds at the type level

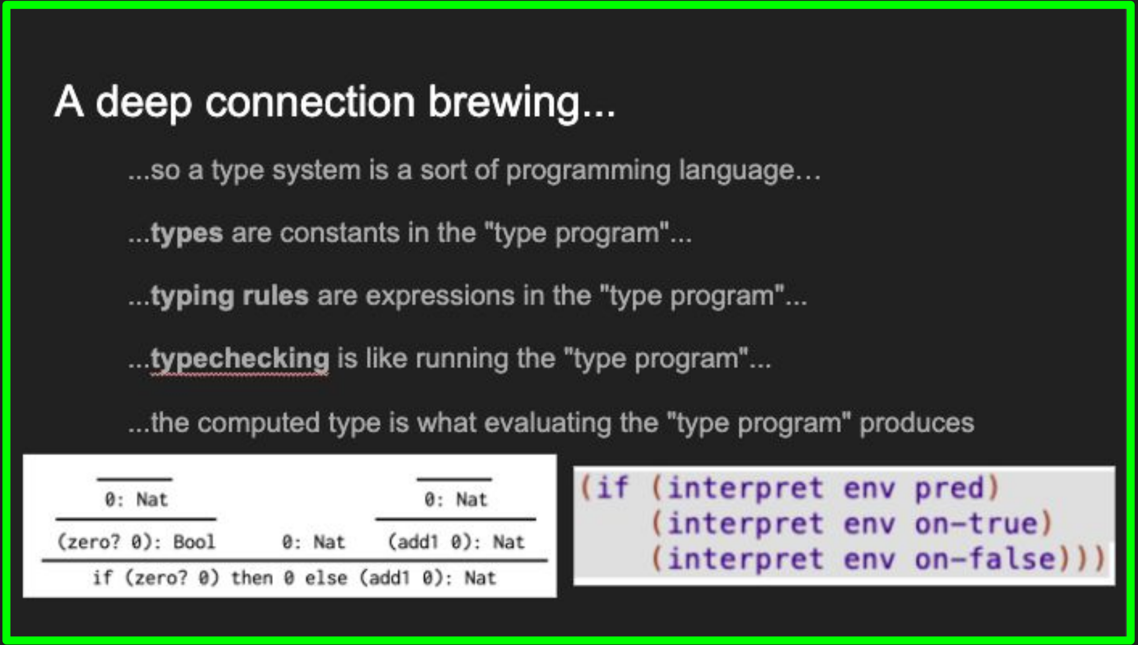
(When we talk about polymorphic types, we will be able to prove that the **values** are the same, too, not just the **types**!)

$$\dfrac{\dfrac{\dfrac{}{\texttt{0:Nat}} \qquad \dfrac{}{\texttt{'empty:List}}}{\texttt{(cons 0 'empty):cons}}}{\texttt{(car (cons 0 'empty)):Nat}}$$

$$\dfrac{\dfrac{\dfrac{}{\texttt{0:Nat}} \qquad \dfrac{}{\texttt{'empty:List}}}{\texttt{(cons 0 'empty):cons}}}{\texttt{(cdr (cons 0 'empty)):List}}$$

Recall this analogy:

In our "programming language of types", types like Nat and Bool were compared to constant values in a conventional programming language



...is there an analogy for **binding values to identifiers** in this analogy, too?

Polymorphism

Definition: Type systems that allow a single piece of code to be used with different types are said to be **polymorphic**.

There are multiple approaches for a type system to take to support polymorphism. In this class we will talk about three:

- parametric polymorphism (today)
- ad-hoc polymorphism & typeclasses (later)
- subtyping (even later!)

Parametric polymorphism

Parametric polymorphism allows a single piece of code to be typed "generically", using a type variable in place of actual types. This encodes the fact that a term can be used in different concrete contexts with different concrete types.

The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

https://docs.oracle.com/javase/tutorial/java/generics/why.html

Type variable

Definition: In the context of some term, a free identifier is said to be a **type variable**. Just like how an identifier "stands in" for a value that it's bound to, a type variable "stands in" for **a type** that it's bound to.

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of `Object` are replaced by `T`. A type variable can be any type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

https://docs.oracle.com/javase/tutorial/java/generics/types.html

Parametric polymorphism

Intuitively, type variables are just placeholders for some actual types, whose exact identities we don't yet know.

(You may see parametric polymorphism informally referred to as "generics". Impress your friends and family by using the lengthier terminology!!!)

Parametric polymorphism

With parametric polymorphism, type variables are held abstract during typechecking. This ensures that any well-typed term will behave correctly no matter what concrete type is substituted later on.

So, if a piece of parametrically-polymorphic code typechecks, it will typecheck **for all types** that are substituted for the type variable.

```
class Box<T> {
    // T stands for "type"
    private T t;

    public Box(T t) { set(t); }

    public void set(T t) {
        this.t = t;
    }
    public T get() { return t; }

    public String toString() {
        return "Box(" + t.toString() + ")";
    }
}
```

Parametric polymorphism

With parametric polymorphism, type variables are held abstract during typechecking. This ensures that any well-typed term will behave correctly no matter what concrete type is substituted later on.

```
-> Box<Integer> b1 = new Box<Integer>(42);
|  Added variable b1 of type Box<Integer> with initial value Box(42)

-> Box<String> b2 = new Box<String>("I'm a box!");
|  Added variable b2 of type Box<String> with initial value Box(I'm a box!)

-> Box<Box<String>> b3 = new Box<Box<String>>(new Box<String>("What's in the box???"))
|  Added variable b3 of type Box<Box<String>> with initial value Box(Box(What's in the box???))

->
```

Parametric polymorphism

With parametric polymorphism, type variables are held abstract during typechecking. This ensures that any well-typed term will behave correctly no matter what concrete type is substituted later on.

This is powerful, but constricting: we can't assume anything about `T`, so we're limited in what we can actually do with it inside the class. (We can print the `T` in `toString` only because that's a method implemented on every object in Java, so every T is guaranteed to have such a method.)

```
class Box<T> {
    // T stands for "type"
    private T t;

    public Box(T t) { set(t); }

    public void set(T t) {
        this.t = t;
    }
    public T get() { return t; }

    public String toString() {
        return "Box(" + t.toString() + ")";
    }
}
```

So rather than having to write a fresh data definition for every type you might want to put in a list, a type variable can stand in for the elements in the list.

```
; A list of Nats is:
; - 'empty
; - (cons x xs) where:
;     - x is a Nat
;     - xs is a list of Nats

; A list of String is:
; - 'empty
; - (cons x xs) where:
;     - x is a String
;     - xs is a list of String

; A list of (Int -> Bool) is:
; - 'empty
; - (cons x xs) where:
;     - x is a function from Int to Bool
;     - xs is a list of (Int -> Bool)

;...
```

Here, T is the type variable that can stand in for an actual type, like `Nat`, `(Int -> Bool)`, etc, etc…

We've been doing this informally in Racket this whole term.

```
; A list of Nats is:
; - 'empty
; - (cons x xs) where:
;    - x is a Nat
```

```
; A list of T is
; - 'empty
; - (cons x xs) where:
;    - x is a T
;    - xs is a list of T
```

```
; - (cons x xs) where:
;    - x is a function from Int to Bool
;    - xs is a list of (Int -> Bool)

;...
```

Here, T is the type variable that can stand in for an actual type, like `Nat`, `(Int -> Bool)`, etc, etc…

We've been doing this informally in Racket this whole term.

...notice that there's an implicit "...for all types T" in this data definition...

```
; A list of T is
; - 'empty
; - (cons x xs) where:
;    - x is a T
;    - xs is a list of T
```

```
; A list of Nats is:
; - 'empty
; - (cons x xs) where:
;    - x is a Nat
; - (cons x xs) where:
;    - x is a function from Int to Bool
;    - xs is a list of (Int -> Bool)

;...
```

Polymorphic types in Haskell

```haskell
data ListOfInts = EmptyListOfInts
                | ConsNat Int ListOfInts
data ListOfStrings = EmptyListOfStrings
                   | ConsStrings String ListOfStrings
data ListOfFnsFromIntToString = EmptyListOfFnsFromIntToString
                              | FnFromIntToString (Int -> String) ListOfFnsFromIntToString
```

Polymorphic types in Haskell

```haskell
data ListOfInts = EmptyListOfInts
                | ConsNat Int ListOfInts
data ListOfStrings = EmptyListOfStrings
                   | ConsStrings String ListOfStrings
data ListOfFnsFromIntToString = EmptyListOfFnsFromIntToString
                              | FnFromIntToString (Int -> String) ListOfFnsFromIntToString

data MyList a = MyEmpty
              | Cons a (MyList a)
```

This type constructor has a type variable `a` (by convention, Haskell uses single-character letters for type variables)

```haskell
data MyList a = MyEmpty
              | Cons a (MyList a)

```

This type constructor has a type variable `a` (by convention, Haskell uses single-character letters for type variables)

The type constructor for `MyList` has a type variable called `a ...`

```haskell
data MyList a = MyEmpty
              | Cons a (MyList a)

```

This type constructor has a type variable a (by convention, Haskell uses single-character letters for type variables)

...that can be used in various value constructors.

```
data MyList a = MyEmpty
              | Cons a (MyList a)
```

Quadtree.hs QuadTree.java

```haskell
data Quadtree a = Leaf a
                | Node (Quadtree a) (Quadtree a) (Quadtree a) (Quadtree a)
                deriving (Show)

height :: Quadtree -> Int
height (Leaf _) = 1
height (Node tl tr bl br) = 1 + (max (max (height tl) (height tr))
                                     (max (height bl) (height br)))
```

```java
/* A subclass of QuadTree, holding four subtrees. */
class QuadNode<T> extends QuadTree<T> {
    private QuadTree<T> tl, tr, bl, br;

    public QuadNode(QuadTree<T> tl, QuadTree<T> tr, QuadTree<T> bl, QuadTree<T>
) {
        this.tl = tl;
```

NORMAL master QuadTree.java jav... 2% ≡ 1/43 ln :

Next time:

An orthogonal kind of polymorphism: **ad-hoc polymorphism** and **typeclasses**