COMP3141

Software System Design and Implementation

Theory of Types

Liam O'Connor
University of Edinburgh LFCS (and UNSW)
Term 2 2020

Natural Deduction

Logic

We can specify a logical system as a *deductive system* by providing a set of rules and axioms that describe how to prove various connectives.

Each connective typically has *introduction* and *elimination* rules.
For example, to prove an implication $A \to B$ holds, we must show that B holds assuming A. This introduction rule is written as:

derivability
(if the top, then the bottom)

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}\ {\to}\mathrm{I}$$

entailment
(assuming the left, we can prove the right)

More rules

Implication also has an elimination rule, that is also called *modus ponens*:

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to\text{-E}$$

Conjunction (and) has an introduction rule that follows our intuition:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \land B} \land\text{-I}$$

It has two elimination rules:

$$\frac{\Gamma \vdash A \land B}{\Gamma \vdash A} \land\text{-E}_1 \qquad \frac{\Gamma \vdash A \land B}{\Gamma \vdash B} \land\text{-E}_2$$

More rules

Disjunction (or) has two introduction rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\vee\text{-I}_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\vee\text{-I}_2$$

Disjunction elimination is a little unusual:

$$\frac{\Gamma \vdash A \vee B \qquad A, \Gamma \vdash P \qquad B, \Gamma \vdash P}{\Gamma \vdash P}\vee\text{-E}$$

The true literal, written $\top$, has only an introduction:

$$\frac{}{\Gamma \vdash \top}$$

And false, written $\bot$, has just elimination (*ex falso quodlibet*):

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}$$

Example Proofs

Example

Prove:

- $A \wedge B \to B \wedge A$
- $A \vee \bot \to A$

What would negation be equivalent to?

Typically we just define

$$\neg A \equiv (A \to \bot)$$

.

Example

Prove:

- $A \to (\neg\neg A)$
- $(\neg\neg A) \to A$ We get stuck here!

Constructive Logic

The logic we have expressed so far does *not* admit the law of the excluded middle:

$$P \vee \neg P$$

Or the equivalent double negation elimination:

$$(\neg\neg P) \rightarrow P$$

This is because it is a *constructive* logic that does not allow us to do proof by contradiction.

Boiling Haskell Down

The theoretical properties we will describe also apply to Haskell, but we need a smaller language for demonstration purposes.

- No user-defined types, just a small set of built-in types.
- No polymorphism (type variables)
- Just lambdas ($\lambda x.e$) to define functions or bind variables.

This language is a very minimal functional language, called the simply typed lambda calculus, originally due to Alonzo Church.

Our small set of built-in types are intended to be enough to express most of the data types we would otherwise define.

We are going to use logical inference rules to specify how expressions are given types (*typing rules*).

Function Types

We create values of a function type $A \to B$ using lambda expressions:

$$\frac{x :: A, \Gamma \vdash e :: B}{\Gamma \vdash \lambda x.\ e :: A \to B}$$

The typing rule for function application is as follows:

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B}$$

What other types would be needed?

Composite Data Types

In addition to functions, most programming languages feature ways to *compose* types together to produce new types, such as:

Classes

Tuples

Structs

Unions

Records

Combining values conjunctively

We want to store two things in one value.

(might want to use non-compact slides for this one)

types

C Structs

Java

Has

```
type Point
midpoint (
  = ((x1+x2
```

```
typed
  flo
  flo
} poi
poin
  poi
  mid
  mid
  ret
}
```


“Better” Java

```
class Point {
  private float x;
  private float y;
  public Point (float x, float y) {
    this.x = x; this.y = y;
  }
  public float getX() {return this.x;}
  public float getY() {return this.y;}
  public float setX(float x) {this.x=x;}
  public float setY(float y) {this.y=y;}
}
Point midPoint (Point p1, Point p2) {
  return new Point((p1.getX() + p2.getX()) / 2.0,
                   (p2.getY() + p2.getY()) / 2.0);
```

Product Types

For simply typed lambda calculus, we will accomplish this with tuples, also called *product types*.

$$(A, B)$$

We won't have type declarations, named fields or anything like that. More than two values can be combined by nesting products, for example a three dimensional vector:

$$(\texttt{Int}, (\texttt{Int}, \texttt{Int}))$$

Constructors and Eliminators

We can construct a product type the same as Haskell tuples:

$$\frac{\Gamma \vdash e_1 :: A \qquad \Gamma \vdash e_2 :: B}{\Gamma \vdash (e_1, e_2) :: (A, B)}$$

The only way to extract each component of the product is to use the `fst` and `snd` eliminators:

$$\frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{fst}\ e :: A} \qquad \frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{snd}\ e :: B}$$

Unit Types

Currently, we have no way to express a type with just one value. This may seem useless at first, but it becomes useful in combination with other types.

We'll introduce the unit type from Haskell, written (), which has exactly one inhabitant, also written ():

$$\frac{}{\Gamma \vdash () : ()}$$

Disjunctive Composition

We can't, with the types we have, express a type with exactly three values.

Example (Trivalued type)

```
data TrafficLight = Red | Amber | Green
```


In general we want to express data that can be one of multiple alternatives, that contain different bits of data.

Example (More elaborate alternatives)

```
type Length = Int
type Angle = Int
data Shape = Rect Length Length
           | Circle Length | Point
           | Triangle Angle Length Length
```


This is awkward in many languages. In Java we'd have to use inheritance. In C we'd have to use unions.

Sum Types

We'll build in the Haskell `Either` type to express the possibility that data may be one of two forms.

$$\texttt{Either}\ A\ B$$

These types are also called *sum types*.

Our `TrafficLight` type can be expressed (grotesquely) as a sum of units:

$$\texttt{TrafficLight} \simeq \texttt{Either ()}\ (\texttt{Either () ()})$$

Constructors and Eliminators for Sums

To make a value of type `Either` A B, we invoke one of the two constructors:

$$\frac{\Gamma \vdash e :: A}{\Gamma \vdash \mathsf{Left}\ e :: \texttt{Either}\ A\ B} \qquad \frac{\Gamma \vdash e :: B}{\Gamma \vdash \mathsf{Right}\ e :: \texttt{Either}\ A\ B}$$

We can branch based on which alternative is used using pattern matching:

$$\frac{\Gamma \vdash e :: \texttt{Either}\ A\ B \qquad x :: A, \Gamma \vdash e_1 :: P \qquad y :: B, \Gamma \vdash e_2 :: P}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{Left}\ x \rightarrow e_1;\ \mathsf{Right}\ y \rightarrow e_2) :: P}$$

Examples

Example (Traffic Lights)

Our traffic light type has three values as required:

$$\texttt{TrafficLight} \simeq \text{Either () (Either () ())}$$

$$\begin{array}{lcl} \texttt{Red} & \simeq & \text{Left ()} \\ \texttt{Amber} & \simeq & \text{Right (Left ())} \\ \texttt{Green} & \simeq & \text{Right (Right (Left ()))} \end{array}$$

The Empty Type

We add another type, called `Void`, that has no inhabitants. Because it is empty, there is no way to construct it.

We do have a way to eliminate it, however:

$$\frac{\Gamma \vdash e :: \mathtt{Void}}{\Gamma \vdash \mathsf{absurd}\ e :: P}$$

If I have a variable of the empty type in scope, we must be looking at an expression that will never be evaluated. Therefore, we can assign any type we like to this expression, because it will never be executed.

Gathering Rules

$$\frac{\Gamma \vdash e :: \texttt{Void}}{\Gamma \vdash \mathsf{absurd}\ e :: P} \qquad \frac{}{\Gamma \vdash () :: ()}$$

$$\frac{\Gamma \vdash e :: A}{\Gamma \vdash \mathsf{Left}\ e :: \texttt{Either}\ A\ B} \qquad \frac{\Gamma \vdash e :: B}{\Gamma \vdash \mathsf{Right}\ e :: \texttt{Either}\ A\ B}$$

$$\frac{\Gamma \vdash e :: \texttt{Either}\ A\ B \qquad x :: A, \Gamma \vdash e_1 :: P \qquad y :: B, \Gamma \vdash e_2 :: P}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{Left}\ x \to e_1; \mathsf{Right}\ y \to e_2) :: P}$$

$$\frac{\Gamma \vdash e_1 :: A \qquad \Gamma \vdash e_2 :: B}{\Gamma \vdash (e_1, e_2) :: (A, B)} \qquad \frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{fst}\ e :: A} \qquad \frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{snd}\ e :: B}$$

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \qquad \frac{x :: A, \Gamma \vdash e :: B}{\Gamma \vdash \lambda x.\ e :: A \to B}$$

Removing Terms...

$$\frac{\Gamma \vdash \texttt{Void}}{\Gamma \vdash P} \qquad \frac{}{\Gamma \vdash ()}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \texttt{Either}\ A\ B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash \texttt{Either}\ A\ B}$$

$$\frac{\Gamma \vdash \texttt{Either}\ A\ B \qquad A, \Gamma \vdash P \qquad B, \Gamma \vdash P}{\Gamma \vdash P}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash (A, B)} \qquad \frac{\Gamma \vdash (A, B)}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash (A, B)}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}$$

This looks exactly like constructive logic!

If we can construct a program of a certain type, we have also created a proof of a

The Curry-Howard Correspondence

This correspondence goes by many names, but is usually attributed to Haskell Curry and William Howard.

It is a *very deep* result:

Programming	Logic
Types	Propositions
Programs	Proofs
Evaluation	Proof Simplification

It turns out, no matter what logic you want to define, there is always a corresponding λ-calculus, and vice versa.

Typed λ-Calculus	Constructive Logic
Continuations	Classical Logic
Monads	Modal Logic
Linear Types, Session Types	Linear Logic
Region Types	Separation Logic

Examples

Example (Commutativity of Conjunction)

$$
\begin{aligned}
&andComm :: (A, B) \rightarrow (B, A) \\
&andComm\ p = (\mathrm{snd}\ p, \mathrm{fst}\ p)
\end{aligned}
$$

This proves $A \wedge B \rightarrow B \wedge A$.

Example (Transitivity of Implication)

$$
\begin{aligned}
&transitive :: (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C) \\
&transitive\ f\ g\ x = g\ (f\ x)
\end{aligned}
$$

Transitivity of implication is just function composition.

Translating

We can translate logical connectives to types and back:

Tuples	Conjunction ($\wedge$)
`Either`	Disjunction ($\vee$)
Functions	Implication
`()`	True
`Void`	False

We can also translate our *equational reasoning* on programs into *proof simplification* on proofs!

Proof Simplification

Assuming $A \wedge B$, we want to prove $B \wedge A$.
We have this unpleasant proof:

$$\frac{\dfrac{A \wedge B}{B} \qquad \dfrac{\dfrac{\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{A}}{A \wedge A}}{A}}{B \wedge A}$$

Proof Simplification

Translating to types, we get:

Assuming $x :: (A, B)$, we want to construct (B, A).

$$\dfrac{\dfrac{x :: (A, B)}{\mathsf{snd}\ x :: B} \qquad \dfrac{\dfrac{\dfrac{x :: (A, B)}{\mathsf{fst}\ x :: A} \qquad \dfrac{x :: (A, B)}{\mathsf{fst}\ x :: A}}{(\mathsf{fst}\ x, \mathsf{fst}\ x) :: (A, A)}}{\mathsf{snd}\ (\mathsf{fst}\ x, \mathsf{fst}\ x) :: A}}{(\mathsf{snd}\ x, \mathsf{snd}\ (\mathsf{fst}\ x, \mathsf{fst}\ x)) :: (B, A)}$$

We know that

$$(\mathsf{snd}\ x, \mathsf{snd}\ (\mathsf{fst}\ x, \mathsf{fst}\ x)) \quad = \quad (\mathsf{snd}\ x, \mathsf{fst}\ x)$$

Lets apply this simplification to our proof!

Proof Simplification

Assuming $x :: (A, B)$, we want to construct (B, A).

$$\dfrac{\dfrac{x :: (A, B)}{\mathsf{snd}\ x :: B} \qquad \dfrac{x :: (A, B)}{\mathsf{fst}\ x :: A}}{(\mathsf{snd}\ x, \mathsf{fst}\ x) :: (B, A)}$$

Back to logic:

$$\dfrac{\dfrac{A \wedge B}{B} \qquad \dfrac{A \wedge B}{A}}{B \wedge A}$$

Applications

As mentioned before, in dependently typed languages such as Agda and Idris, the distinction between value-level and type-level languages is removed, allowing us to refer to our program in types (i.e. propositions) and then construct programs of those types (i.e. proofs).

Peano Arithmetic

If there's time, Liam will demo how to prove some basic facts of natural numbers in Agda, a dependently typed language.

Generally, dependent types allow us to use rich types not just for programming, but also for verification via the Curry-Howard correspondence.

Caveats

All functions we define have to be total and terminating.

Otherwise we get an *inconsistent* logic that lets us prove false things:

$$proof_1 :: \mathsf{P} = \mathsf{NP}$$
$$proof_1 = proof_1$$

$$proof_2 :: \mathsf{P} \neq \mathsf{NP}$$
$$proof_2 = proof_2$$

Most common calculi correspond to constructive logic, not classical ones, so principles like the law of excluded middle or double negation elimination do not hold:

$$\neg\neg P \rightarrow P$$

Semiring Structure

These types we have defined form an algebraic structure called a *commutative semiring*.

Laws for `Either` and `Void`:

- Associativity: $\texttt{Either}\ (\texttt{Either}\ A\ B)\ C\ \simeq\ \texttt{Either}\ A\ (\texttt{Either}\ B\ C)$
- Identity: $\texttt{Either Void}\ A\ \simeq\ A$
- Commutativity: $\texttt{Either}\ A\ B\ \simeq\ \texttt{Either}\ B\ A$

Laws for tuples and `1`

- Associativity: $((A, B), C)\ \simeq\ (A, (B, C))$
- Identity: $((), A)\ \simeq\ A$
- Commutativity: $(A, B)\ \simeq\ (B, A)$

Combining the two:

- Distributivity: $(A, \texttt{Either}\ B\ C)\ \simeq\ \texttt{Either}\ (A, B)\ (A, C)$
- Absorption: $(\texttt{Void}, A)\ \simeq\ \texttt{Void}$

What does $\simeq$ mean here? It's more than logical equivalence.

Isomorphism

Two types A and B are *isomorphic*, written $A \simeq B$, if there exists a *bijection* between them. This means that for each value in A we can find a unique value in B and vice versa.

Example (Refactoring)

We can use this reasoning to simplify type definitions. For example:


```
data Switch = On Name Int
            | Off Name
```

Can be simplified to the isomorphic `(Name, Maybe Int)`.

Generic Programming

Representing data types generically as sums and products is the foundation for generic programming libraries such as GHC generics. This allows us to define algorithms that work on arbitrary data structures.

Type Quantifiers

Consider the type of `fst`:

```
fst :: (a,b) -> a
```

This can be written more verbosely as:

```
fst :: forall a b. (a,b) -> a
```


Or, in a more mathematical notation:

$$\text{fst} :: \forall a\ b.\ (a, b) \to a$$

This kind of quantification over type variables is called parametric polymorphism or just polymorphism for short.
(It's also called generics in some languages, but this terminology is bad)

What is the analogue of $\forall$ in logic? (via Curry-Howard)?

Curry-Howard

The type quantifier $\forall$ corresponds to a universal quantifier $\forall$, but it is not the same as the $\forall$ from first-order logic. What's the difference?

First-order logic quantifiers range over a set of *individuals* or values, for example the natural numbers:

$$\forall x.\ x + 1 > x$$

These quantifiers range over propositions (types) themselves. It is analogous to *second-order logic*, not first-order:

$$\forall A.\ \forall B.\ A \wedge B \to B \wedge A$$
$$\forall A.\ \forall B.\ (A, B) \to (B, A)$$

The first-order quantifier has a type-theoretic analogue too (type indices), but this is not nearly as common as polymorphism.

Generality

If we need a function of type $\texttt{Int} \rightarrow \texttt{Int}$, a polymorphic function of type $\forall a.\ a \rightarrow a$ will do just fine, we can just instantiate the type variable to $\texttt{Int}$. But the reverse is not true. This gives rise to an ordering.

Generality

A type A is *more general* than a type B, often written $A \sqsubseteq B$, if type variables in A can be instantiated to give the type B.

Example (Functions)

$$\texttt{Int} \rightarrow \texttt{Int} \quad \sqsupseteq \quad \forall z.\ z \rightarrow z \quad \sqsupseteq \quad \forall x\ y.\ x \rightarrow y \quad \sqsupseteq \quad \forall a.\ a$$

Constraining Implementations

How many possible total, terminating implementations are there of a function of the following type?

$$\texttt{Int} \rightarrow \texttt{Int}$$

How about this type?

$$\forall a.\ a \rightarrow a$$

Polymorphic type signatures constrain implementations.

Parametricity

Definition

The principle of parametricity states that the result of polymorphic functions cannot depend on values of an abstracted type.

More formally, suppose I have a polymorphic function g that is polymorphic on type a. If run any arbitrary function $f :: a \to a$ on all the a values in the input of g, that will give the same results as running g first, then f on all the a values of the output.

Example

$$foo :: \forall a.\ [a] \to [a]$$

We know that **every** element of the output occurs in the input.
The parametricity theorem we get is, for all f:

$$foo \circ (map\ f) = (map\ f) \circ foo$$

More Examples

$$head :: \forall a.\ [a] \rightarrow a$$

What's the parametricity theorem?

Example (Answer)

For any f:

$$f\ (head\ \ell) = head\ (map\ f\ \ell)$$

More Examples

$$(++) :: \forall a.\ [a] \to [a] \to [a]$$

What's the parametricity theorem?

Example (Answer)

$$map\ f\ (a ++ b) = map\ f\ a ++ map\ f\ b$$

More Examples

$$concat :: \forall a.\ [[a]] \rightarrow [a]$$

What's the parametricity theorem?

Example (Answer)

$$map\ f\ (concat\ ls) = concat\ (map\ (map\ f)\ ls)$$

Higher Order Functions

$$filter :: \forall a.\ (a \to Bool) \ \to [a] \to [a]$$

What's the parametricity theorem?

Example (Answer)

$$filter\ p\ (map\ f\ ls) = map\ f\ (filter\ (p \circ f)\ ls)$$

Parametricity Theorems

Follow a similar structure. In fact it can be mechanically derived, using the *relational parametricity* framework invented by John C. Reynolds and popularised by Wadler in the famous paper, "Theorems for Free!"[1].

Upshot: We can ask `lambdabot` on the Haskell IRC channel for these theorems.

[1] `https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf`

Wrap-up

1. That's the entirety of the assessable course content for COMP3141.
2. There is a quiz for this week, but no exercise (there's still Assignment 2)

3. Next week's lectures consist of a **extension lecture** video on dependent type systems, and a **revision lecture** on Wednesday with Curtis..
4. Please come up with **questions** to ask Curtis for the revision lecture! It will be over very quickly otherwise.