COMP3141

Liam O'Conn
University of Edinburgh LFCS (an
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 t

For example, to pr B holds assuming A. Thi

derivability

(if the top, then the bottom)

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 (a

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

It has two elimination rules:

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

More rules

Disjunction (or) has two introduction rules:

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

Disjunction elim

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

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

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

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

Example Proofs

Example

Prove:

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

What would nega

Typically we just d

$$\neg A \equiv (A \to$$

.

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:

Or the equivalent d

$$(\neg\neg P) \to$$

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 polymor
- Just lambd

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 expres 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:

The typing rule for f

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

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:

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

```
typ
  f
  f
} poi
poin
  poi
  mid
  mid
  ret
}
```

Has

```
type Point

midpoint (
  = ((x1+x2
```



```
  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*.

We won't have type declarations, named fields or anything lik… values can be combined by nesting products, for example a thre…

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

Constructors and Eliminators

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

$$\frac{}{\quad}$$

The only way to extr ... `fst` and `snd` eliminators:

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

Unit Types

Currently, we have no way to express a type with just one value. This may seem useless at first, but it

We'll introduce th ne

inhabitant, also w

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

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 t
contain different

Example (Mor

```
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.

These types are also called *sum types*.

Our `TrafficLight` type can be expressed (grotesquel

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

Constructors and Eliminators for Sums

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

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

$$\frac{\Gamma \vdash e : \texttt{Either}\ A\ B \quad x :: A, \Gamma \vdash e \quad \ldots : P}{\Gamma \vdash (\textbf{case}\ e\ \textbf{of}\ \mathsf{Left}\ x \to e_1;}$$

Examples

Example (Traffic Lights)

Our traffic light ty

$$
\begin{aligned}
\texttt{Red} &\simeq \mathsf{Left}\ () \\
\texttt{Amber} &\simeq \mathsf{Right}\ (\mathsf{Le} \\
\texttt{Green} &\simeq \mathsf{Right}\ (\mathsf{Ri}
\end{aligned}
$$

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 el

$$\frac{}{\Gamma \vdash \text{absurd } e :}$$

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

Gathering Rules

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

$$\frac{\Gamma \vdash e :: \texttt{Either}\ A\ B \qquad x :: A, \Gamma \vdash e_1 :: P \qquad y :: B, \Gamma \vdash e_2 :: P}{\Gamma \vdash (\textbf{case}\ e\ \textbf{of}\ \textsf{Left}\ x \rightarrow e_1; \ \ldots}$$

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

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

Removing Terms...

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

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

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

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

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:

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

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

Examples

Example (Commutativity of Conjunction)

$$andComm :: (A, B) \quad (B, A)$$

This proves A

Example (Transitivity of Implication)

$$transitive :: (A \to B) \to (B$$
$$transitive\ f\ g\ x = g\ (f\ x)$$

Transitivity of implication is just function composition.

Translating

We can translate logical connectives to types and back:

`()`	True
`Void`	F

We can also translate our *equational reasoning* *plification*
on proofs!

Proof Simplification

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

$$\dfrac{\dfrac{A \land B}{B} \qquad \dfrac{\dfrac{\dfrac{}{} \quad \dfrac{}{}}{}}{A}}{B \land A}$$

Proof Simplification

Translating to types, we get:

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

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

We know that

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

Lets apply this simplification to our proof!

Proof Simplification

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

$$\dfrac{\dfrac{x :: (A, B)}{\;} \qquad \dfrac{x :: (A, B)}{\;}}{\;}$$

Back to logic:

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

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 progra
types (i.e. proofs).

Peano Arithme

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

Generally, dependent types allow us to use rich types not just for p
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:

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

$$proof_2 = pro$$

Most common calculi correspond to constructive logic, not cl

like the law of excluded middle or double negation elimination do not hold:

$$\neg\neg P \to P$$

Semiring Structure

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

Laws for `Either` and `Void`:

- Associativity: `Either` (`Either` A B) C `Either` A (`Either` B C)
- Identity:
- Commuta

Laws for tuples an

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

Combining the two:

- Distributivity: $(A,$ `Either` B $C) \simeq$ `Either` (A, B) (A, C)
- Absorption: $($`Void`$, A) \simeq$ `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 (Refa

We can use this reas


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

Can be simplified to the isomorphic `(Name, Mayb`

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

```
fst :: forall a b. (a
```


Or, in a more mathe

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

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

What is the analogue of ∀ 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

These quantifier

second-order logic, not first-order:

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

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 *m* [illegible] iables 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?

How about this typ

$$\forall a.\ a \to a$$

Polymorphic type signatures constrain implementation

Parametricity

Definition

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

More formally, su … phic on type a.

If run any arbitrary f … g, that will

give the same resul … e output

Example

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

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 param

Example (Ans

For any f:

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

More Examples

$$(++) :: \quad a.\ [a] \quad [a] \quad [a]$$

What's the param

Example (Answer)

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

More Examples

$$concat :: \quad a.\ [[a]] \qquad [a]$$

What's the param

Example (Answer)

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

Higher Order Functions

What's the param

Example (Ans

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

Parametricity Theorems

Follow a similar str *relational*
parametricity f r in
the famous paper,

Upshot: We can ask `lambdabot` on the Haskell IRC c

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

Wrap-up

1. That's the e
2. There is a quiz f
3. Next week dent type systems, and a **revision lecture** on Wednesday with Curtis..
4. Please come up with **questions** to ask Curtis fo over very quickly otherwise.