Week 8

Type theory

Katie Casamento
Fall 2018

Portland State University

Lambda calculus semantics

In week 6, we talked about the *small-step semantics* of untyped lambda calculus.

This is a *dynamic* semantics: it lets us **evaluate** a program down to a **value**.

A *static* semantics lets us **prove things** about programs **without evaluating them**.

Type systems are the most common form of static semantics.

Types

A *type* in a programming language can be thought of as a **set** of expressions (usually).

If an expression is a member of some type, we say it is *well-typed*.

If an expression is not a member of any type, we say it is *ill-typed*.

A *type system* is the part of a programming language that deals with types.

Types

The syntax "`e : P`" means "the expression `e` is a member of the type `P`".

$$
\begin{aligned}
\texttt{true} &: \texttt{bool} \\
(\texttt{if true then false else true}) &: \texttt{bool}
\end{aligned}
$$

$$
\begin{aligned}
0 &: \texttt{num} \\
(1+3) &: \texttt{num}
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.\ x) &: (\texttt{num} \ \rightarrow \ \texttt{num}) \\
(\lambda x.\ x+1) &: (\texttt{num} \ \rightarrow \ \texttt{num})
\end{aligned}
$$

In some but not all type systems, one expression can be a member of multiple types.

Type errors

This expression is ill-typed:

$(\lambda x.\ x) + 1$

We can't add a function to a number.

This expression is also stuck.

One of the main goals of using types in programming is:

Well-typed expressions do not get stuck.

Stuck expressions are like runtime errors, so what we're really saying is:

Well-typed expressions do not throw certain kinds of errors at runtime.

This property is known as *type safety*.

We have to design a type system very carefully in order to make it safe.

Type systems

The process of checking whether a term is well-typed is called *typechecking*.

A language is

- *untyped* if typechecking **never** happens
  - very uncommon outside of theory
  - assembly/machine code, untyped lambda calculus, set theory, Mathematica, ...
- *statically-typed* if typechecking must happen **before** runtime
  - types **are** part of the **static** semantics
  - types **may or may not be** part of the **dynamic** semantics (reflection)
  - C, C++, C♯, Java, Haskell, Scala, Rust, Swift, ...
- *dynamically-typed* if typechecking must happen **during** runtime
  - also occasionally called *unityped* (note the ‘i’)
  - types **are not** part of the **static** semantics
  - types **are** part of the **dynamic** semantics
  - Python, Ruby, JavaScript, Scheme, Clojure, ...
- *gradually-typed* if typechecking might happen **both before and during** runtime
  - usually synonymous with *optionally-typed*
  - types **are** part of both the **static and dynamic** semantics
  - historically uncommon, but increasingly popular in recent years
  - Perl, TypeScript, Python+MyPy, Julia, Typed Clojure, Nickle, ...

Program correctness

Program correctness is arguably the entire end goal of programming.

A program is *correct* if (1) there is a specification that (2) it always obeys without fail.

If you say you've (successfully) written a program, you're saying two things:

1. you know what the program is supposed to do, and
2. you believe that the program does what it is supposed to do.

Software testing **measures** program correctness.

Formal semantics let us **prove** program correctness.

You should always be able to give **some** argument that your program is correct.

How strong that argument needs to be depends on the purpose of your program.

Static vs. dynamic typing

Usually,

- **statically-typed** languages prioritize **program correctness**
- **dynamically-typed** languages prioritize **pace of development**

Programming projects sometimes benefit from a two-stage development process:

- **rapid prototyping** in a dynamically-typed language, followed by
- **careful implementation** in a statically-typed language.

Gradually-typed languages are designed to make this a continuous process.

Static typing

Benefits of static typing include:

- **safety:** "well-typed terms do not go wrong"[1]
  - some runtime errors become type errors instead
  - debugging type errors is very often easier than debugging runtime errors
- **documentation:** the type of an expression can be read as a partial specification
  - `swap` : $(a, b) \to (b, a)$
  - `concat` : $[[a]] \to [a]$
  - `insert` : $\texttt{Int} \to a \to [a] \to [a]$
- **organization:** types give us a high-level view of the structure of a program
  - functional design starts with a set of data types and function types
  - object-oriented design starts with a hierarchy of class types and method types
- **static analysis:** types aid tools that reason about other properties of programs
  - type-informed program optimization
  - program correctness proofs relying on type safety

[1] "Types and programming languages", Benjamin Pierce

Dynamic typing

Benefits of dynamic typing include:

- **brevity:** program text is usually shorter
  - no need to write down the types of everything
  - static *type inference* can approximate this in statically-typed languages
- **flexibility:** programs are sometimes easier to extend
  - "monkey-patching" is usually only possible with dynamic typing
  - option to skip the whole high-level design phase and just start hacking
- **freedom:** the language "trusts" you more and "gets in your way" less
  - only a good thing if you are actually a trustworthy programmer!
  - there are almost always valid programs we **can** write that a static typechecker disallows
  - there are not often valid programs we **should** write that a static typechecker disallows

“Strong” typing vs. “weak” typing

You may also sometimes hear the terms *strongly-typed* and *weakly-typed*.

These are not very well-defined and usually shouldn’t be used in technical language.

> Because of the wide divergence among these definitions, it is possible to defend claims about most programming languages that they are either strongly or weakly typed.
> - Wikipedia, “Strong and weak typing”

In everyday language, “strong” and “weak” are relative to use case.

Roughly, a “strong” type system **catches more errors** than a “weak” one.

If you ask which of two football players is stronger, the answer is probably complicated.

If you ask whether a football player is stronger than me, the answer is definitely yes.

It’s pretty much the same with type systems.

Lambda calculus and proof theory

In week 6, we studied **untyped lambda calculus** as a language of **programs**.

In week 7, we studied **intuitionistic proof theory** as a language of **proofs**.

Here are two questions that might not seem related, but are actually very similar.

- How do we decide whether a program is well-typed?
- How do we decide whether a proof is valid?

Propositions as types

A **proposition** in proof theory can be thought of as a **type** in a **language of proofs**.

The **connectives** are ***type formers*** - operators that **construct types** out of other types.

The **proofs** are **expressions** in the language.

This is sometimes called the *Curry-Howard correspondence*.

The application of this to some proof theories is more natural than to others.

It turns out to be very natural for intuitionistic logic.

Proof theory into type theory

Programs are written in a **one-dimensional** syntax: they're an array of characters.

Our editors display them in lines and columns, but a newline is just the `\n` character.

Proof trees are written in a **two-dimensional** syntax: they have width and height.

They have some nice structural properties, but they're pretty inconvenient, right?

We know how to define one-dimensional syntax, so we can do that for proofs too.

Proof theory into type theory

It turns out **lambda calculus** is a suitable syntax for proofs as well as programs.

We're going to make small changes to our **proof theory** to get a **type theory**.

The specific type theory we'll get is called ***simply-typed lambda calculus*** (STLC).

Along the way, we'll introduce some new kinds of lambda calculus expressions.

Simply-typed lambda calculus

STLC is untyped lambda calculus with a type system.

- introduced by Alonzo Church for proofs[2]
- can be interpreted in many domains:
  - as a functional programming language (programming language theory)
  - as an intuitionistic proof language (proof theory)
  - as the internal language of Cartesian closed categories (category theory)

The typing rules of STLC are a pretty good example of typing rules in general.

[2] "A formulation of the simple theory of types." 1940.

Deductive systems

A *judgement* is the kind of thing that we might put above or below a line in a rule.

"$\Gamma \vdash P$" and "$A \in \Gamma$" are judgements.

A *deductive system* tells us how to build proofs of judgements.

We define a deductive system with

- a set of *axioms*, which gives us a minimal set of proofs to start with, and
- a set of *inference rules*, which tell us how to combine proofs into bigger proofs.

A lot of interesting things in math and CS can be expressed as deductive systems.

(You should ask Steven Libby at PSU about logic programming.)

Deductive system syntax

We often define deductive systems in the two-dimensional syntax we've been using.

This style of definition was introduced for a system called "sequent calculus".[3]

Our use of the "$\vdash$" symbol is sometimes called "sequent notation".

reduction rule:

$$\frac{n_1 + n_2 = n_3}{n_1 + n_2 \Rightarrow n_3}\ \texttt{E-Plus}$$

logical rule:

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \land Q}\ \land\texttt{-Intro}$$

(These two rules aren't related, they're just examples of the style of definition.)

In this style, an axiom is just a rule with no premises.

[3] Gentzen, Gerhard (1934). "Untersuchungen über das logische Schließen. I".

Type judgements

The judgement “$\Gamma \vdash \mathrm{e} : P$” says “the expression e has type P in context Γ”.

In a proof language, this is also read as “e is a proof of P under Γ”.

Judgements of the form “$\Gamma \vdash e : P$” are called *type judgements*.

A proof of a type judgement is a *type derivation*.

A *type theory* is a **language** of expressions, types, and type derivations.

The unit type

Let's start with an easy rule: the introduction rule for the trivial proposition $\mathtt{T}$.

$$\frac{}{\Gamma \vdash \mathtt{T}}\ \texttt{T-Intro}$$

In type theory, we call "$\mathtt{T}$" the "unit type". We'll write it as `Unit`.

We'll also define a new expression: the expression `unit` always has type `Unit`.

$$\frac{}{\Gamma \vdash \mathtt{unit} : \mathtt{Unit}}\ \texttt{Unit-Intro}$$

`Unit` can be useful to implement the "`void`" type from C and similar languages.

Confusingly, the proposition $\mathtt{F}$ is sometimes called "`Void`" in type theory.

We won't say much more about $\mathtt{F}$ in this class, though, so don't think about that now.

Primitive types

A *primitive type* is one that is not defined with any type formers.

We also sometimes call these *built-in* types or *atomic* types.

`Unit` is a primitive type.

We've also been working with numbers and Booleans in lambda calculus.

We'll call these primitive types `Num` and `Bool`.

$$\frac{}{\Gamma \vdash n : \texttt{Num}}\ \texttt{Num} \qquad\qquad \frac{}{\Gamma \vdash b : \texttt{Bool}}\ \texttt{Bool}$$

(As usual, n is a numeric literal and b is a Boolean literal.)

Primitive operations

A *primitive operation* is a function over primitive types that is not defined with the language's mechanisms for creating functions.

Numeric operators and `if/then/else` are primitive operations.

$$\frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num}}{\Gamma \vdash e_1 + e_2 : \texttt{Num}}\ \texttt{Plus} \qquad \frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num}}{\Gamma \vdash e_1 * e_2 : \texttt{Num}}\ \texttt{Times}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num}}{\Gamma \vdash e_1 < e_2 : \texttt{Bool}}\ \texttt{Less}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : P \qquad \Gamma \vdash e_3 : P}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : P}\ \texttt{If}$$

Note that **both cases** of an `if/then/else` in this type system must be **the same type**.

This is roughly true in most type systems, although subtyping complicates the matter.

(We'll talk about subtyping in a later lecture.)

Variables

The context of a proof is a list of **assumptions**.

$$\frac{B \in [A, B, C]}{[A, B, C] \vdash B}\ \texttt{Ass}$$

The context of a type derivation is a list of **named assumptions**.

$$\frac{[x : A,\ y : B,\ z : C](y) = B}{[A, B, C] \vdash B}\ \texttt{Var}$$

The syntax "$\Gamma(y) = B$" means

in the list [x : A, y : B, z : C], the assumption B has the name y.

In type theory, we call a named assumption a *variable*.

Variables

In logic, the assumption rule in general says

if P is in Γ, then there is a proof of $\Gamma \vdash P$.

$$\frac{P \in \Gamma}{\Gamma \vdash P}\ \texttt{Ass}$$

In type theory, the variable rule in general says

if P has the name x in Γ, then x has type P in context Γ.

$$\frac{\Gamma(x) = P}{\Gamma \vdash x : P}\ \texttt{Var}$$

This is the rule that tells us what a variable's type is in a program.

Variable shadowing

Specifically, "$\Gamma(x) = P$" means the **leftmost** occurrence of x in Γ is paired with P.

$$[x : A,\ x : B,\ y : C](x) = A$$

$$[x : A,\ x : B,\ y : C](x) \neq B$$

This is called *variable shadowing*: we say the $x : A$ variable shadows the $x : B$ one.

Variable binding

When we write a variable name in a program, we mean it in one of two different ways.

- a *binding* (or *declaration*) is a place in a program a variable is **introduced**
  - $\lambda x. \ldots$
  - `int x = ...`
  - `int f(int x) { ... }`
  - `let x = 1 in ...`
- a *use* is a place in a program where a variable is **referenced**
  - `printf("%d", x)`
  - `x + x`
  - `let x = y in ...`

The *scope* of a binding x is the part of the program where x refers to that binding.

We say a variable use is *well-scoped* if it refers to **exactly one** binding.

A variable is *shadowed* if there is more than one binding that it **might** correspond to.

The *scoping rules* of a language tell us which binding (if any) is correct in these cases.

Scoping rules

The scoping rules of a language specify the scope of each kind of binding.

In STLC, the definition of the judgement $\Gamma(x) = P$ is a scoping rule.

It says **inner** variables shadow **outer** variables.

This is also true in our small-step rules (because of capture-avoiding substitution).

$(\lambda x.\ (\lambda x.\ x))\ 1\ 2 \Rightarrow^* 2$

The scoping rules of a type system must match those of the runtime semantics.

Scoping rules in C

In C:

Local variables shadow **global** variables.

This program prints “2”.

```
int x = 1;
void main() {
  int x = 2;
  printf("%d", x)
}
```

The **arguments** to a function are in scope in the **body** of the function.

```
void foo(int x) {
  // x is in scope
}
// x is out of scope
```

Scoping rules

Scoping rules can get pretty complicated.

This is valid Java code:

```
class x { x x(x x) { return (x) x(x); } }
```

It's best to just avoid writing code like this.

Still, if you work with a language a lot, it's important to know its scoping rules.

They can always be found in the language specification.

Function creation

In proof theory, we prove an implication by assuming its premise and proving its conclusion.

$$\frac{P :: \Gamma \vdash Q}{\Gamma \vdash P \to Q}\ \to\texttt{-Intro}$$

In type theory, we create a function (lambda) expression by assuming a variable of the input type and constructing an expression of the output type.

$$\frac{(x : P) :: \Gamma \vdash e : Q}{\Gamma \vdash (\lambda x.\ e) : P \to Q}\ \texttt{Lambda}$$

Function creation examples

Axioms:

$$\frac{\Gamma(x) = P}{\Gamma \vdash x : P}\,\texttt{Var} \qquad\qquad \frac{(x : P) :: \Gamma \vdash e : Q}{\Gamma \vdash (\lambda x.\ e) : P \to Q}\,\texttt{Lambda}$$

Type derivations:

$$\frac{\dfrac{[x : P](x) = P}{[x : P] \vdash x : P}\,\texttt{Var}}{[] \vdash (\lambda x.\ x) : P \to P}\,\texttt{Lambda} \qquad \frac{\dfrac{\dfrac{[y : Q, x : P](x) = P}{[y : Q, x : P] \vdash x : P}\,\texttt{Var}}{[x : P] \vdash (\lambda y.\ x) : Q \to P}\,\texttt{Lambda}}{[] \vdash (\lambda x.\ (\lambda y.\ x)) : P \to (Q \to P)}\,\texttt{Lambda}$$

Function usage

In proof theory, we use a proof of an implication by applying it to a proof of its premise.

$$\frac{\Gamma \vdash P \to Q \qquad \Gamma \vdash P}{\Gamma \vdash Q} \to\texttt{-Elim}$$

In type theory, we use a function by applying it to an argument of its input type.

$$\frac{\Gamma \vdash e_1 : P \to Q \qquad \Gamma \vdash e_2 : P}{\Gamma \vdash (e_1\ e_2) : Q} \texttt{App}$$

Function vocabulary

The *signature* of a function is the part that gives names (and types) to the parameters.

The *body* of a function is the part that defines what the function does.

A *parameter* to a function is an input variable in the function signature.

(A parameter is **always a variable.**)

An *argument* to a function is an expression passed in when the function is called.

$$(\overline{\lambda x}.\ \underline{x})\ \boxed{1}$$

```
int f(int x){ return g( 2 , x ); }
```

Pair type creation

In proof theory, we prove $P \wedge Q$ by proving each of P and Q.

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \wedge\texttt{-Intro}$$

In type theory, we construct an expression of a *pair* of two types by constructing an expression of each of the types.

$$\frac{\Gamma \vdash e_1 : P \qquad \Gamma \vdash e_2 : Q}{\Gamma \vdash (e_1, e_2) : (P, Q)} \texttt{Pair}$$

Pair types are also called *product* types and *tuple* types.

Pair type usage

In logic, we use a proof of $P \wedge Q$ by combining it with a proof that assumes P and Q.

$$\frac{\Gamma \vdash P \wedge Q \qquad P :: Q :: \Gamma \vdash R}{\Gamma \vdash R}\ \wedge\texttt{-Elim}$$

In type theory, we use an expression of type (P, Q) by combining it with an expression that assumes variables of types P and Q.

$$\frac{\Gamma \vdash e_1 : (P, Q) \qquad (x : P) :: (y : Q) :: \Gamma \vdash e_2 : R}{\Gamma \vdash \texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2 : R}\ \texttt{Let}$$

Pair type usage example

Axioms:

$$\frac{\Gamma(x) = P}{\Gamma \vdash x : P}\ \texttt{Var} \qquad \frac{(x : P) :: \Gamma \vdash e : Q}{\Gamma \vdash (\lambda x.\ e) : P \to Q}\ \texttt{Lambda}$$

$$\frac{\Gamma \vdash e_1 : P \qquad \Gamma \vdash e_2 : Q}{\Gamma \vdash (e_1, e_2) : (P, Q)}\ \texttt{Pair}$$

$$\frac{\Gamma \vdash e_1 : (P, Q) \qquad (x : P) :: (y : Q) :: \Gamma \vdash e_2 : R}{\Gamma \vdash \texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2 : R}\ \texttt{Let}$$

Derivation:

$$\frac{\dfrac{[x : (P, Q)](x) = (P, Q)}{[x : (P, Q)] \vdash x : (P, Q)}\ \texttt{Var} \qquad \dfrac{[x_1 : P,\ x_2 : Q,\ x : (P, Q)](x_1) = P}{[x_1 : P,\ x_2 : Q,\ x : (P, Q)] \vdash x_1 : P}\ \texttt{Var}}{\dfrac{[x : (P, Q)] \vdash \texttt{let}\ (x_1, x_2) = x\ \texttt{in}\ x_1 : P}{[] \vdash \lambda x.\ \texttt{let}\ (x_1, x_2) = x\ \texttt{in}\ x_1 : (P, Q) \to P}\ \texttt{Lambda}}\ \texttt{Let}$$

Pair type evaluation

We choose arbitrarily to evaluate the left side of a pair first.

$$\frac{e_1 \Rightarrow e_1'}{(e_1, e_2) \Rightarrow (e_1', e_2)}\ \texttt{E-Pair1} \qquad \frac{e_2 \Rightarrow e_2'}{(v_1, e_2) \Rightarrow (v_1, e_2')}\ \texttt{E-Pair2}$$

Here are the steps to evaluate "$\texttt{let}\ (x_1, x_2)\ = e_1\ \texttt{in}\ e_2$".

1. evaluate e_1 down to a pair of values (v_1, v_2)
2. substitute v_1 and v_2 for x_1 and x_2 in e_2

$$\frac{e_1 \Rightarrow e_2'}{\texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{let}\ (x_1, x_2) = e_1'\ \texttt{in}\ e_2}\ \texttt{E-Let1}$$

$$\frac{}{\texttt{let}\ (x_1, x_2) = (v_1, v_2)\ \texttt{in}\ e_2 \Rightarrow e_2[v_1/x_1][v_2/x_2]}\ \texttt{E-Let2}$$

Tagged union type creation

In proof theory, we prove $P \vee Q$ by proving either P or Q.

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \vee_L\texttt{-Intro} \qquad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \vee_R\texttt{-Intro}$$

In type theory, we construct an expression of a *tagged union* of two types by constructing an expression of either of the types.

$$\frac{\Gamma \vdash e : P}{\Gamma \vdash \texttt{left}\ e : (P|Q)} \texttt{Left} \qquad \frac{\Gamma \vdash e : Q}{\Gamma \vdash \texttt{right}\ e : (P|Q)} \texttt{Right}$$

Tagged unions are also called *discriminated union* types, *sum* types, and *variant* types.

Tagged union type usage

In proof theory, we use a proof of $P \vee Q$ by combining it with a proof that assumes P and a proof that assumes Q.

$$\frac{\Gamma \vdash P \vee Q \qquad P :: \Gamma \vdash R \qquad Q :: \Gamma \vdash R}{\Gamma \vdash R}\ \vee\texttt{-Elim}$$

In type theory, we use an expression of type $(P|Q)$ by combining it with an expression that assumes a variable of type P and an expression that assumes a variable of type Q.

$$\frac{\Gamma \vdash e_1 : (P|Q) \qquad (x_1 : P) :: \Gamma \vdash e_2 : R \qquad (x_2 : Q) :: \Gamma \vdash e_3 : R}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ \{\ \texttt{left}\ x_1 \to e_2 \mid \texttt{right}\ x_2 \to e_3\ \} : R}\ \texttt{Case}$$

Tagged union type evaluation

We can evaluate under `left` and `right`.

$$\frac{e \Rightarrow e'}{\texttt{left}\ e \Rightarrow \texttt{left}\ e'}\ \texttt{E-Left} \qquad \frac{e \Rightarrow e'}{\texttt{right}\ e \Rightarrow \texttt{right}\ e'}\ \texttt{E-Right}$$

Here are the steps to evaluate "$\texttt{case}\ e_1\ \texttt{of}\ \{\ \texttt{left}\ x_1 \rightarrow e_2\ ;\ \texttt{right}\ x_2 \rightarrow e_3\ \}$".

1. evaluate e_1 down to a value `left` v_1 or `right` v_1
2. if you get `left` v_1, substitute v_1 for x_1 in e_2
   if you get `right` v_1, substitute v_1 for x_2 in e_3

$$\frac{e_1 \Rightarrow e_1'}{\texttt{case}\ e_1\ \texttt{of}\ \{\ \ldots\ \} \Rightarrow \texttt{case}\ e_1'\ \texttt{of}\ \{\ \ldots\ \}}\ \texttt{E-Case1}$$

$$\frac{}{\texttt{case}\ (\texttt{left}\ v_1)\ \texttt{of}\ \{\ \texttt{left}\ x_1 \rightarrow e_2\ ;\ \texttt{right}\ x_2 \rightarrow e_3\ \} \Rightarrow e_2[v_1/x_1]}\ \texttt{E-Case2L}$$

$$\frac{}{\texttt{case}\ (\texttt{left}\ v_1)\ \texttt{of}\ \{\ \texttt{left}\ x_1 \rightarrow e_2\ ;\ \texttt{right}\ x_2 \rightarrow e_3\ \} \Rightarrow e_3[v_2/x_2]}\ \texttt{E-Case2R}$$

Untagged union types

In C, an *untagged union* type is a value that can be viewed as multiple different types.

The size of an untagged union of types `A` and `B` is `max(sizeof(A), sizeof(B))`.

```
union charIntUntagged { char c; int i; }
```

```
union charIntUntagged x;
```

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

```
x.c = ’A’;
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1

```
printf(’%c’, x.c); // prints "A"
printf(’%d’, x.i); // prints "65"
```

```
x.i = 90;
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0

```
printf(’%c’, x.c); // prints "Z"
printf(’%d’, x.i); // prints "90"
```

Tagged union implementation

An **untagged** union of `A` and `B` is **both** an `A` and a `B`.

A **tagged** union of `A` and `B` is **either** an `A` or a `B`.

A **tagged** union is implemented as an **untagged** union paired with a "tag" value.

```
union charIntUntagged { char c; int i; }
struct charIntTagged { bool tag; union charIntUntagged val; }
```

tag	val

`charIntTagged` is an implementation of the tagged union type (`char`|`int`).

We implement `left` and `right` as follows:

- a `left` value is one **constructed** with `tag = true` and a `char` in `val`

true	<char bits>

- a `right` value is one **constructed** with `tag = false` and an `int` in `val`

false	<int bits>

Tagged union implementation

Now we'll generalize to any arbitrary types `A` and `B`.

```
union untagged { A a; B b; }
struct tagged { bool tag; union untagged val; }

struct tagged left(A a) {
  struct tagged t;
  t.tag = true;
  t.val.a = a;
  return t;
}

struct charIntTagged right(B b) {
  struct tagged t;
  t.tag = false;
  t.val.b = b;
  return t;
}
```

Tagged union implementation

The case/of construct can be implemented with a conditional.

```
union untagged { A a; B b; }
struct tagged { bool tag; union untagged val; }

C caseOf(struct tagged t) {
  if (t.tag) {
    ... // do something with t.val.a and return something of type C
  } else {
    ... // do something with t.val.b and return something of type C
  }
}
```

Tagged union implementation

We implement `left` and `right` as follows:

- a `left` value is one **constructed** with `tag = true` and a `char` in `val`
- a `right` value is one **constructed** with `tag = false` and an `int` in `val`

Note that as defined, these rules are **conventions**, not **part of C**.

We make an honor system promise to only use `t.val.a` when `t.tag = true`.

C will not enforce this rule.

This is why it's useful to have tagged union types in the definition of a language.

Algebraic data types

We've looked at three type formers.

- function types ($\rightarrow$)
- pair types (,)
- tagged union types ($|$)

A type made with these type formers is an *algebraic* data type (ADT).

ADTs are also sometimes called *inductive* data types, by connection to proof theory.

Algebraic data types in practice

Consider a data type representing an HTTP request.

An (oversimplified) HTTP request is **one** of the following:

- a "GET request" containing a URL
- a "POST request" containing a URL and a body of text (a string)
- a "LINK request" containing two URLs

We can define this type as an algebraic data type.

$$\texttt{URL} \mid (\texttt{URL}, \texttt{String}) \mid (\texttt{URL}, \texttt{URL})$$

Haskell data types are algebraic data types.

```
data Req = Get URL | Post URL String | Link URL URL
```

Traditional "real-world" languages usually don't include tagged union types.

Working with things like HTTP requests is traditionally awkward in those languages.

Algebraic data types in modern language design

ADTs are one of the most common defining features of recent languages.

- Swift (iOS)
- Kotlin (Android)
- Rust (low-level)
- Scala (backend web)
- Elm, TypeScript (frontend web)

Many problems are naturally phrased in terms of functions over ADTs.

In this class, we've been using ADTs in Haskell to represent **syntax trees**.

Simply-typed lambda calculus

Here are all the typing rules for STLC.

$$\frac{\Gamma(x) = P}{\Gamma \vdash x : P}\ \texttt{Var}$$

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}}\ \texttt{Unit-Intro} \qquad \frac{}{\Gamma \vdash n : \texttt{Num}}\ \texttt{Num} \qquad \frac{}{\Gamma \vdash b : \texttt{Bool}}\ \texttt{Bool}$$

$$\frac{(x : P) :: \Gamma \vdash e : Q}{\Gamma \vdash (\lambda x.\ e) : P \to Q}\ \texttt{Lambda} \qquad \frac{\Gamma \vdash e_1 : P \to Q \qquad \Gamma \vdash e_2 : P}{\Gamma \vdash (e_1\ e_2) : Q}\ \texttt{App}$$

$$\frac{\Gamma \vdash e_1 : P \qquad \Gamma \vdash e_2 : Q}{\Gamma \vdash (e_1, e_2) : (P, Q)}\ \texttt{Pair}$$

$$\frac{\Gamma \vdash e_1 : (P, Q) \qquad (x : P) :: (y : Q) :: \Gamma \vdash e_2 : R}{\Gamma \vdash \texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2 : R}\ \texttt{Let}$$

$$\frac{\Gamma \vdash e : P}{\Gamma \vdash \texttt{left}\ e : (P|Q)}\ \texttt{Left} \qquad \frac{\Gamma \vdash e : Q}{\Gamma \vdash \texttt{right}\ e : (P|Q)}\ \texttt{Right}$$

$$\frac{\Gamma \vdash e_1 : (P|Q) \qquad (x_1 : P) :: \Gamma \vdash e_2 : R \qquad (x_2 : Q) :: \Gamma \vdash e_3 : R}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ \{\ \texttt{left}\ x_1 \to e_2 \mid \texttt{right}\ x_2 \to e_3\ \} : R}\ \texttt{Case}$$

Ill-typed terms

"$\texttt{true} + 1$" is ill-typed in STLC.

$$\frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num}}{\Gamma \vdash e_1 + e_2 : \texttt{Num}}\ \texttt{Plus}$$

$$\frac{\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \Gamma \vdash 1 : \texttt{Num}}{\Gamma \vdash \texttt{true} + 1 : \texttt{Num}}\ \texttt{Plus}$$

"$\texttt{true} + 1$" is also stuck.

In a safe type system, all stuck terms are ill-typed.

Are all ill-typed terms stuck?

Can you come up with an ill-typed STLC term that evaluates to a value without error?

Ill-typed terms

This term is ill-typed, but it evaluates to 3.

$$\dfrac{\dfrac{\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \Gamma \vdash \texttt{1} : \texttt{Num} \qquad \Gamma \vdash \texttt{false} : \texttt{Bool}}{\Gamma \vdash \texttt{if true then 1 else false} : \texttt{Num}}\ \texttt{If} \qquad \Gamma \vdash \texttt{2} : \texttt{Num}}{\Gamma \vdash \texttt{(if true then 1 else false) + 2} : \texttt{Num}}\ \texttt{Plus}$$

$$\texttt{(if true then 1 else false) + 2} \Rightarrow \texttt{1 + 2} \Rightarrow \texttt{3}$$

There are almost always terms like this in a statically-typed language.

Most type systems don't have full *coverage* - they exclude some valid expressions.

It's rare that a term like this is **useful** in practice, but it comes up occasionally.

This is one of the (arguable) downsides to static typing.

Termination in STLC

This definition of STLC is *strongly normalizing*, or *total*.

This means evaluation always terminates for every well-typed term.

(Try giving a type derivation for Ω from the week 6 slides.)

We can build non-termination into STLC with a *fixed-point* operator.

$$\begin{array}{rl} \texttt{untyped} & \Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\ \texttt{typed} & \Omega = \texttt{fix}\ (\lambda x.\ x\ x) \end{array}$$

$$\frac{\Gamma \vdash e : P \to P}{\Gamma \vdash \texttt{fix}\ e : P}\ \texttt{Fix}$$

This is how we implement *recursion* in type theory.

$\texttt{fact} = \texttt{fix (}\lambda\texttt{ f x. if x = 0 then 1 else x * (f (x-1)))}$

Typechecking

Typechecking is the process of trying to show whether “$\Gamma \vdash e : P$“ has a derivation.

In STLC (and most languages), this is very mechanical and easy to automate.

We can read the typing rules of STLC as a **typechecking algorithm**.

We’ll go through a couple examples.

We assume that we know what type each expression is **supposed to** be.

This is the purpose of *type annotations* in programs, like function signatures.

Typechecking `Plus`

$$\frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num}}{\Gamma \vdash e_1 + e_2 : \texttt{Num}}\ \texttt{Plus}$$

To typecheck "$e_1 + e_2 : \texttt{Num}$" under context Γ:

- check that e_1 has type `Num` under Γ
- check that e_2 has type `Num` under Γ

Typechecking `Var`

$$\frac{\Gamma(x) = P}{\Gamma \vdash x : P}\ \texttt{Var}$$

To typecheck "$x : P$" under context Γ:

- check that $\Gamma(x)$ is P

Typechecking `Lambda`

$$\frac{(x : P) :: \Gamma \vdash e : Q}{\Gamma \vdash (\lambda x.\ e) : P \to Q}\ \texttt{Lambda}$$

To typecheck "$\lambda x.\ e : P \to Q$" under context Γ:

- check that e has type Q under context $(x : P) :: \Gamma$

Typechecking `App`

$$\frac{\Gamma \vdash e_1 : P \to Q \qquad \Gamma \vdash e_2 : P}{\Gamma \vdash (e_1\ e_2) : Q}\ \texttt{App}$$

To typecheck $(e_1\ e_2) : Q$ under context Γ:

- check that e_1 has type $P \to Q$ under Γ
- check that e_2 has type P under Γ

Type inference

Type inference is the process of trying to **find** a P such that $\Gamma \vdash e : P$ has a derivation.

In STLC, this is pretty easy - especially if we annotate the types of lambda arguments.

$\lambda(x : \texttt{Num}).\ x + 1 : \texttt{Num} \rightarrow \texttt{Num}$

Most statically-typed languages have some form of type inference.

`auto` in C++, `var` in C$\sharp$, <> in Java, ...

Some languages are designed for **full** type inference.

Haskell, OCaml, Elm, ...

Type annotations aren't **necessary** in these languages, but they can still be **useful**.

Proof theory epilogue

What does this all mean for proof theory?

We can build **proof checkers** with the same techniques we use for **type checkers**.

We can build **proof search** with the same techniques we use for **type inference**.

A *proof assistant* is a set of tools for writing and checking proofs on a computer.

This is a relatively young field, but it's had a couple big successes.

If you're interested, here are some languages to check out:

- Coq (math, verified programming, proof automation)
- Agda (math, verified programming)
- Idris (verified programming)
- Isabelle (math)
- TLA+ (verified concurrent algorithms)

Philosophy epilogue

What does all this mean for philosophical foundations of logic?

From the BHK interpretation (week 7):

- A proof of $P \wedge Q$ is a proof of P **and** a proof of Q.
- A proof of $P \vee Q$ is a proof of P **or** a proof of Q.
- A proof of $P \rightarrow Q$ is a **procedure** to turn a proof of P into a proof of Q.

We can use the language of lambda calculus to give meaning to the **bold** words.

This means we can give an intuitionistic logical foundation in terms of computation.

This is arguably a philosophical win, if you believe in computation more than logic.

Every proof has a *normal* (minimal) form: eliminations followed by introductions.

Evaluation corresponds to **normalization**: converting a proof to normal form.

Looking ahead

In the next lectures, we'll extend STLC into a minimal C-style *imperative* language.

After that, we'll cover some more advanced types (in less depth than the STLC ones).

At the end of the quarter, we'll do some comparative study of a variety of languages.