AN INTRODUCTION TO THE λ-CALCULUS AND TYPE THEORY

CS5209

Aquinas Hobor and Martin Henz

First Question

- What is a “λ”?

First Question

- What is a “λ”?

- Greek letter “lambda”. Pronounced like “L”

  - Lower case: λ
  - Upper case: Λ

Second question

- What is the λ-calculus about?

Second question

- What is the λ-calculus about?
- It is a method for writing and reasoning about *functions*.
- Functions, of course, are central to both mathematics and computer science.
- The ideas were developed by Alonzo Church in the 1930s during investigations into the foundations of mathematics.

Third question

- What is type theory?

Third question

- What is type theory?

- A method of classifying computations according to the types (kinds, sorts) of values produced.

- You are already familiar with the basics:
  - int myint;
  - Object myobject;

- Analogous to **units** in scientific computation.

Topics Covered Today

- Untyped lambda calculus
- Simply-typed lambda calculus
- Polymorphic lambda calculus (System F)

The Untyped λ-Calculus (Syntax)

- Syntax is very simple; only three kinds of terms:

e	=	x, y, z, …	(Variables)
		$\lambda x.\ e$	(Functions)
		$e_1\ e_2$	(Application)

Examples:

- x
- $\lambda x.\ x$
- z y
- $(\lambda x.\ x)\ (\lambda y.\ y)$

The Untyped λ-Calculus (Semantics)

- Semantics is also very simple – only one rule!

$$(\lambda\ x.\ e_1)\ e_2 \qquad \mapsto \qquad [x \to e_2]\ e_1$$

We substitute the term e_2 for x in the term e_1.

Examples:

- $(\lambda x.\ x)\ (\lambda y.\ y) \qquad \mapsto \qquad (\lambda y.\ y)$
- $(\lambda x.\ x\ x)\ (\lambda y.\ y) \qquad \mapsto \qquad (\lambda y.\ y)\ (\lambda y.\ y)$

$\qquad\qquad\qquad\qquad \mapsto \qquad (\lambda y.\ y)$

MORE EXAMPLES (RENAMING)

- $(\lambda x.\ \lambda y.\ x\ y)\ (\lambda y.\ y)\ z \quad \mapsto$
  $(\lambda y.\ (\lambda y.\ y)\ y)\ z \quad \mapsto$
  $(\lambda y.\ y)\ z \quad \mapsto$
  z

First question: what is the difference between $(\lambda y.\ y)$ and $(\lambda x.\ x)$?

Convention: these are **identical**: lambda terms are equal to any “uniform” renaming.

MORE EXAMPLES (RENAMING)

Our convention means we can rename as we like:

- $(\lambda x.\ \lambda y.\ x\ y)\ (\lambda y.\ y)\ z \mapsto$

  $(\lambda y.\ (\lambda a.\ a)\ y)\ z \mapsto$

  $(\lambda a.\ a)\ z \mapsto$

  z

This process helps avoid variable-capture, etc.

MORE EXAMPLES (EVALUATION ORDER)

Consider this example:

$((\lambda x.\ x)\ (\lambda y.\ y))\ ((\lambda a.\ a)\ (\lambda b.\ b))$

There are multiple ways that it could evaluate:

1. $(\lambda y.\ y)\ ((\lambda a.\ a)\ (\lambda b.\ b))$
   1. $((\lambda a.\ a)\ (\lambda b.\ b))$
      - $\lambda b.\ b$
   2. $(\lambda y.\ y)\ (\lambda b.\ b)$
      - $\lambda b.\ b$
2. $((\lambda x.\ x)\ (\lambda y.\ y))\ (\lambda b.\ b)$
   1. $(\lambda y.\ y)\ (\lambda b.\ b)$
      - $\lambda b.\ b$

Observation

- There are lots!
- But they lead to the same place – do we care?
- For now, leave this question aside. We will choose to follow the convention used in programming languages: to evaluate e_1 e_2,

1. Evaluate e_1 first until it reaches a λ-term
2. Evaluate e_2 as far as you can
3. Do the substitution

SEE THIS BEHAVIOR IN C

Suppose that `f` is a function pointer to a function that takes a single integer argument.

How does this line behave?

```
(*f) (3 + x);
```

1. Dereference `f`
2. Add `3 + x`
3. Call the function with the result

This is called, call-by-value

Call-By-Value, Formally

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}$$

$$\frac{e_2 \mapsto e_2'}{(\lambda\ x.\ e_1')\ e_2 \mapsto (\lambda\ x.\ e_1')\ e_2'}$$

$$\frac{}{(\lambda\ x.\ e_1')\ v \mapsto [x \to v]\ e_1'}$$

PROGRAMMING IN THE LAMBDA CALCULUS

- Since what we are really describing is a kind of computation, there is a natural question: how can we encode the standard ideas in programming?

- For example, if-then-else?

- Definitions:
  - fls $\equiv$ $\lambda x.\ \lambda y.\ y$
  - tru $\equiv$ $\lambda x.\ \lambda y.\ x$
  - if $\equiv$ $\lambda b.\ \lambda t.\ \lambda e.\ b\ t\ e$

(convention: "a b c" is "(a b) c")

If-Then-Else

- Definitions:
  - fls $\equiv$ $\lambda x.\ \lambda y.\ y$
  - tru $\equiv$ $\lambda x.\ \lambda y.\ x$
  - if $\equiv$ $\lambda b.\ \lambda t.\ \lambda e.\ b\ t\ e$

$$
\begin{array}{ll}
\text{if tru a b} & = \\
(\lambda b.\ \lambda t.\ \lambda e.\ b\ t\ e)\ (\lambda x.\ \lambda y.\ x)\ a\ b & \mapsto \\
(\lambda t.\ \lambda e.\ (\lambda x.\ \lambda y.\ x)\ t\ e)\ a\ b & \mapsto \\
(\lambda e.\ (\lambda x.\ \lambda y.\ x)\ a\ e)\ b & \mapsto \\
(\lambda x.\ \lambda y.\ x)\ a\ b & \mapsto \\
(\lambda y.\ a)\ b & \mapsto \\
a &
\end{array}
$$

IF-THEN-ELSE

- Definitions:
  - fls $\equiv$ λx. λy. y
  - tru $\equiv$ λx. λy. x
  - if $\equiv$ λb. λt. λe. b t e

$$
\begin{array}{ll}
\text{if fls a b} & = \\
(\lambda b.\ \lambda t.\ \lambda e.\ b\ t\ e)\ (\lambda x.\ \lambda y.\ y)\ a\ b & \mapsto \\
(\lambda t.\ \lambda e.\ (\lambda x.\ \lambda y.\ y)\ t\ e)\ a\ b & \mapsto \\
(\lambda e.\ (\lambda x.\ \lambda y.\ y)\ a\ e)\ b & \mapsto \\
(\lambda x.\ \lambda y.\ y)\ a\ b & \mapsto \\
(\lambda y.\ y)\ b & \mapsto \\
b &
\end{array}
$$

What about Numbers?

- Definitions:
  - zero ≡ λx. λy. y
  - one ≡ λx. λy. x y
  - two ≡ λx. λy. x (x y)
  - three ≡ λx. λy. x (x (x y))

  - succ ≡ λn. λx. λy. x (n x y)

- Notice: “zero” and “fls” are the same!

- This is common in computer science…

Calculating with numbers

- Definitions:
  - zero $\equiv$ $\lambda x.\ \lambda y.\ y$
  - one $\equiv$ $\lambda x.\ \lambda y.\ x\ y$
  - succ $\equiv$ $\lambda n.\ \lambda x.\ \lambda y.\ x\ (n\ x\ y)$

succ zero $=$

$(\lambda n.\ \lambda x.\ \lambda y.\ x\ (n\ x\ y))\ (\lambda x.\ \lambda y.\ y)$ $\mapsto$

$\lambda x.\ \lambda y.\ x\ ((\lambda x.\ \lambda y.\ y)\ x\ y)$

Is this the same as "one"?

Evaluation Order, Revisted

succ zero	=	λx. λy. x ((λx. λy. y) x y)
one	=	λx. λy. x y

Obviously these are not identical…
but look at what happens when we apply both to the arguments “a” and “b”…

Evaluation Order, Revisited

succ zero	=	$\lambda x.\ \lambda y.\ x\ ((\lambda x.\ \lambda y.\ y)\ x\ y)$
one	=	$\lambda x.\ \lambda y.\ x\ y$

$(\lambda x.\ \lambda y.\ x\ ((\lambda x.\ \lambda y.\ y)\ x\ y))\ a\ b \quad \mapsto$

$(\lambda y.\ a\ ((\lambda x.\ \lambda y.\ y)\ a\ y))\ b \quad \mapsto$

$a\ ((\lambda x.\ \lambda y.\ y)\ a\ b) \quad \mapsto$

$a\ ((\lambda y.\ y)\ b) \quad \mapsto$

$a\ b$

$(\lambda x.\ \lambda y.\ x\ y)\ a\ b \quad \mapsto^*$

$a\ b$

Evaluation Order, Revisited

- So while the functions are not the same, they are similar: they will reach the same final result

- Maybe a more familiar example of this kind of difference: both quicksort and mergesort produce the same result when applied to the same input – but they are not the same function (different running time!)

- An interesting question: does the evaluation order only effect the running time?

More numbers…

- Definitions:
  - zero ≡ $\lambda x.\ \lambda y.\ y$
  - one ≡ $\lambda x.\ \lambda y.\ x\ y$
  - two ≡ $\lambda x.\ \lambda y.\ x\ (x\ y)$
  - three ≡ $\lambda x.\ \lambda y.\ x\ (x\ (x\ y))$

  - succ ≡ $\lambda n.\ \lambda x.\ \lambda y.\ x\ (n\ x\ y)$
  - plus ≡ $\lambda n.\ \lambda m.\ \lambda x.\ \lambda y.\ m\ x\ (n\ x\ y)$
  - mult ≡ $\lambda n.\ \lambda m.\ \lambda x.\ \lambda y.\ n\ (m\ x)\ y$
  - …
- Possible to define subtraction, etc. etc.

Other computation features…

- One feature you may have noticed: λ-terms are anonymous. That is, the functions are unnamed.

- λ x. x

- `int foo(int x) { return x; }`

- Here, the function has a name (`foo`).
  - There are other differences, too – like the types.
  - We will discuss these later

- Observation: the lambda calculus is **concise**.

NAMED VS. UNNAMED

- What do functions really need names for?
- For a function like `foo`, not much.
- But maybe another function wants to call it…
- … still, that issue can be worked around.
- More serious: what if you want a recursive function? Then you need a way to call yourself.

Nonterminating Computation

- Can we express nonterminating computation?

- Yes!

- Consider the term: diverge $\equiv (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$$\begin{array}{ll}
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) & \mapsto \\
[x \rightarrow (\lambda x.\ x\ x)]\ x\ x & = \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) & \mapsto \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) & \mapsto \\
\ldots &
\end{array}$$

Nonterminating Computation

- What if we want a more general form of nontermating computation?

- We can define a term

  fix ≡ … (will show later)

Now let's suppose we want to define the factorial function, written in pseudo-form like this:

fact ≡ λx. if (iszero x) 1 (mult x (fact (pred x)))

Nonterminating Computation

fact $\equiv$ λx. if (iszero x) 1 (mult x (fact (pred x)))

The problem with this definition is that in mathematics we are not allowed to write circular definitions…

We could write one iteration as follows:

λf. λx. if (iszero x) 1 (mult x (f (pred x)))

Now “f” is being used as the recursive call.

Nonterminating Computation

λf. λx. if (iszero x) 1 (mult x (f (pred x)))

What we want is to "tie the knot" – and this is what fix does!

fix (λ f. e) $\mapsto^*$ [f $\to$ (fix (λ f. e))] e

So, for example,

fix (λf. λx. if (iszero x) 1 (mult x (f (pred x)))) $\mapsto^*$
λx. if (iszero x) 1 (mult x (**(fix (λf. λx. if (iszero x) 1 (mult x (f (pred x)))))**)(pred x)))

Nonterminating computation

- fix thus lets us write recursive functions

- So what does fix actually look like?

$\text{fix} \equiv \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$

$\text{fbody} \equiv (\lambda f.\ \lambda x.\ \text{if (iszero } x)\ 1\ (x * (f\ (x - 1))))$

$\text{fix fbody} \quad \mapsto^*$
$\lambda x.\ \text{if (iszero } x)\ 1\ (x * ((\text{fix fbody})\ (x - 1)))$

Evaluation Order, Revisted Again

- The presence of recursive and nonterminating computation means that evaluation order is important; consider:

tru zero (λx. diverge) $\mapsto^*$
zero (under the call-by-value rules)

If we are allowed to evaluate functions anywhere:

tru zero (λx. diverge) $\mapsto^*$
tru zero (λx. diverge) (by always evaluating diverge)

AN INTERESTING THEOREM

- Except for running-time analysis and the possibility of divergence, **evaluation order does not affect the final result of computation**.

- That is why it is ok to define succ in a way such that “succ zero” is not equal to “one” – but on any (terminating) input, they are the same.

AN INTERESTING QUESTION

- We have lots of features normally found in programming languges:
  - If-then-else
  - Functions
  - Recursion
  - Arithmetic
  - …

- How powerful is the lambda calculus?

An interesting question

- We have lots of features normally found in programming languges:
  - If-then-else
  - Functions
  - Recursion
  - Arithmetic
  - …
- How powerful is the lambda calculus?

CHURCH-TURING THESIS

Says:

- Turing Machines (the standard theoretical model for computation) and the lambda calculus have the same computational power.
- Thus, any algorithm you would like can be encoded and run in the lambda calculus.

Topics Covered Today

- ~~Untyped lambda calculus~~
- Simply-typed lambda calculus
- Polymorphic lambda calculus (System F)

Simply-type Lambda Calculus

- Observation: writing programs is **hard**

- Lots of bugs!

- Most bugs are "stupid errors"
  - Forget to cast a number
  - Use a pointer instead of dereferencing it
  - Forget to check boundary condition

UNITS

- The same kinds of problems occur in other areas
- Physics:
  - When you are learning physics, they teach you units
  - kilograms, meters, seconds, Newtons, etc.
- When you do a calculation, you are told that it is very important to keep track of the units:
  - 3 m * (2 kg / m) = 6 kg
- Why?

UNITS

- 3 m * (2 kg / m) = 6 kg

- Because it lets you do a “sanity check” on the result – if you are expecting kilograms, but get meters per second – you have made a mistake!

- The unit calculations (m * kg / m = kg) are much simpler than the numerical calculations

Units in Programming

- **Types** are the units in a programming language
- You know many of these from C, Java, etc.:

int	(integers)
bool	(Booleans)
char*	(pointers to characters)

- The compiler automatically checks for a misuse, helping to find bugs.

Adding types to the lambda calculus

- First, let's add a few extra terms:

e	=	…	
		T	(Boolean true constant)
		F	(Boolean false constant)

Now let's define some types:

t	=	bool	(Boolean type)
		$t \to t$	(function type)

Our notation for lambda terms is a bit different:

$$\lambda x : t.\ e$$

Adding types to the lambda calculus

Our notation for lambda terms is a bit different:

$$\lambda x : t.\ e$$

The idea is that this function can only be applied to arguments that have type t.

How do we know what the type of an arguments is?

Typing Rules

- We can define a series of inductive **typing rules** that tell us how to type lambda terms:

A context Γ is a function from variables to types.

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

Typing Rules

- We can define a series of inductive **typing rules** that tell us how to type lambda terms:

Boolean values are easy to type.

$$\frac{}{\Gamma \vdash \mathrm{T} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash \mathrm{F} : \mathrm{bool}}$$

Typing Rules

- We can define a series of inductive **typing rules** that tell us how to type lambda terms:

How do we type function application?

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

Typing Rules

- We can define a series of inductive **typing rules** that tell us how to type lambda terms:

All that is missing is how to type lambda terms:

$$\frac{\Gamma, (x : t_1) \vdash e : t_2}{\Gamma \vdash (\lambda\, x : t_1.\ e) : t_1 \to t_2}$$

Typing Rules

- We can define a series of inductive **typing rules** that tell us how to type lambda terms:

Let's see all of the typing rules together now:

$$\frac{}{\Gamma \vdash \mathrm{T} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash \mathrm{F} : \mathrm{bool}} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

$$\frac{\Gamma, (x : t_1) \vdash e : t_2}{\Gamma \vdash (\lambda\, x : t_1.\ e) : t_1 \to t_2}$$

Using the typing rules

- Consider the term

$$(\lambda\ f : \text{bool} \to \text{bool}.\ f\ T)\ (\lambda\ b : \text{bool}.\ b)$$

Prove that in any context Γ, this has type bool.

(see blackboard!)

Why are types important?

- Informally, we have noticed that type errors indicate bugs in the program.
- e.g. "T F" does have any type since "T" is of type bool, and not of type bool $\rightarrow t_2$
- Is there something more formal that we can say?

WHY ARE TYPES IMPORTANT?

- Yes!

- “Well-typed programs don’t go wrong”

- Recall from the Hoare logic lecture (and the HW), the idea of safety: a state σ is safe if for any reachable state σ’, that state can either take another step or has reached some well-defined final value. (As opposed to “getting stuck”, for example with “T F”.)

Why are types important?

- Yes!
- “Well-typed programs don’t go wrong”
- We say that a type system is **sound** if all well-typed programs are safe.
- Examples of sound type systems:
  - Simply-typed lambda calculus
  - Java
  - ML

Why are types important?

- Examples of **unsound** type systems:
  - C
  - C++
  - Python

- In these systems, the type system may help out a little – but a lack of type errors does not mean safety: in C, it's easy to segfault

How can we prove a type system is sound?

Usually, we want to prove a pair of results:

Definition: a term e is a **value** if e $\in$ {λ-term, T, F}

- Progress: if {} $\vdash$ e : t, then either e is a value (λ-term, T, F), or there exists an e' such that e $\mapsto$ e'.
  - For example, this will not be true if there is a type t such that {} $\vdash$ T F : t
  - {} here is the empty context
- Preservation: if {} $\vdash$ e : t and e $\mapsto$ e', then {} $\vdash$ e' : t
  - That is, well-typed terms remain well-typed

Proving Safety

- Safety follows due to the combination of Progress and Preservation.

- Informally: Safety = Progress + Preservation

- Theorem: if $\{\} \vdash e : t$, then e is safe.

PROOF

We assume $\{\} \vdash e_0 : t$. By Progress, either e_0 is a value, or there exists e_1 such that $e_0 \mapsto e_1$.

By Preservation, $\{\} \vdash e_1 : t$. Thus, by Progress, either e_1 is a value, or there exists e_2 such that $e_1 \mapsto e_2$.

By Preservation, $\{\} \vdash e_2 : t$. Thus, by Progress, either e_1 is a value, or there exists e_3 such that $e_2 \mapsto e_3$.

… this is doing induction on the $\mapsto^*$ relation …

Proving Progress

- Induction on the typing judgment

$$\frac{}{\Gamma \vdash \mathrm{T} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash \mathrm{F} : \mathrm{bool}} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

$$\frac{\Gamma, (x : t_1) \vdash e : t_2}{\Gamma \vdash (\lambda\, x : t_1.\ e) : t_1 \rightarrow t_2}$$

PROVING PROGRESS

Suppose $\{\} \vdash e : t$. We do induction:

Cases 1 and 2:

$$\frac{}{\{\} \vdash T : \text{bool}} \qquad \frac{}{\{\} \vdash F : \text{bool}}$$

In these cases, e = T or e = F – and so e is already a value! So we are done.

Proving Progress

Suppose $\{\} \vdash e : t$. We do induction:

Case 3:

$$\frac{\{\}(x) = t}{\{\} \vdash x : t}$$

This case is impossible, since the empty context "{}" does not map x to anything! So we are done.

Proving Progress

Suppose $\{\} \vdash e : t$. We do induction:

Case 5 (we will come back to case 4):

$$\frac{\{x : t_1\} \vdash e : t_2}{\{\} \vdash (\lambda\ x : t_1.\ e) : t_1 \rightarrow t_2}$$

This case is just like cases 1 & 2: λ-terms are already values! So we are done.

Proving Progress

Suppose $\{\} \vdash e : t$. We do induction:

Case 4 (the only case where we must do work):

$$\frac{\{\} \vdash e_1 : t_1 \rightarrow t_2 \qquad \{\} \vdash e_2 : t_1}{\{\} \vdash e_1\ e_2 : t_2}$$

Our induction hypothesis tells us that either e_1 steps or is a value. If it steps to e_1', we are done (since call-by-value means that $e_1\ e_2 \mapsto e_1'\ e_2$).

Proving Progress

Suppose $\{\} \vdash e : t$. We do induction:

Case 4 (the only case where we must do work):

$$\frac{\{\} \vdash e_1 : t_1 \rightarrow t_2 \qquad \{\} \vdash e_2 : t_1}{\{\} \vdash e_1\ e_2 : t_2}$$

Our induction hypothesis tells us that either e_1 steps or is a value. If it is a value, then e_1 must be a lambda-term ($e_1 = \lambda$ x. e_1') since T and F have type bool, not type $t_1 \rightarrow t_2$.

Proving Progress

Suppose $\{\} \vdash e : t$. We do induction:

Case 4 (the only case where we must do work):

$$\frac{\{\} \vdash \lambda x.\ e_1' : t_1 \rightarrow t_2 \qquad \{\} \vdash e_2 : t_1}{\{\} \vdash (\lambda x.\ e_1')\ e_2 : t_2}$$

Our induction hypothesis tells us that either e_2 steps or is a value. If it steps to e_2', we are done (call-by-value means that $(\lambda x.\ e_1')\ e_2 \mapsto (\lambda x.\ e_1')\ e_2'$).

Proving Progress

Suppose $\{\} \vdash e : t$. We do induction:

Case 4 (the only case where we must do work):

$$\frac{\{\} \vdash \lambda x.\ e_1' : t_1 \to t_2 \qquad \{\} \vdash e_2 : t_1}{\{\} \vdash (\lambda x.\ e_1')\ e_2 : t_2}$$

Our induction hypothesis tells us that either e_2 steps or is a value. If e_2 is a value, then we have

$(\lambda x.\ e_1')\ e_2 \quad \mapsto \quad [x \to e_2]\ e_1'$

PROVING PROGRESS

- So we have proved progress by induction!

$$\frac{}{\Gamma \vdash \mathrm{T} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash \mathrm{F} : \mathrm{bool}} \qquad \frac{\Gamma(\mathrm{x}) = \mathrm{t}}{\Gamma \vdash \mathrm{x} : \mathrm{t}}$$

$$\frac{\Gamma \vdash \mathrm{e}_1 : \mathrm{t}_1 \rightarrow \mathrm{t}_2 \qquad \Gamma \vdash \mathrm{e}_2 : \mathrm{t}_1}{\Gamma \vdash \mathrm{e}_1\ \mathrm{e}_2 : \mathrm{t}_2}$$

$$\frac{\Gamma, (\mathrm{x} : \mathrm{t}_1) \vdash \mathrm{e} : \mathrm{t}_2}{\Gamma \vdash (\lambda\, \mathrm{x} : \mathrm{t}_1 .\ \mathrm{e}) : \mathrm{t}_1 \rightarrow \mathrm{t}_2}$$

Proving Preservation

- How do we prove preservation? By induction on the step relation:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}$$

$$\frac{e_2 \mapsto e_2'}{(\lambda\ x : t.\ e_1')\ e_2 \mapsto (\lambda\ x.\ e_1')\ e_2'}$$

$$\frac{}{(\lambda\ x : t.\ e_1')\ v \mapsto [x \to v]\ e_1'}$$

Proving Preservation

- Suppose $\{\} \vdash e : t$ and $e \mapsto e'$. We do induction:

Only case where we evaluate is $e = e_1\ e_2$.
Examination of the typing rules tells us:

$\{\} \vdash e_1 : t_1 \rightarrow t_2$ and $\{\} \vdash e_2 : t_1$

Proving Preservation

- Suppose $\{\} \vdash e : t$ and $e \mapsto e'$. We do induction:

Case 1 (we know $\{\} \vdash e_1 : t_1 \to t_2$) :

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}$$

By induction hypothesis, $\{\} \vdash e_1' : t_1 \to t_2$

Thus, we can type $e_1'\ e_2$ in the same way as $e_1\ e_2$.

Proving Preservation

- Suppose $\{\} \vdash e : t$ and $e \mapsto e'$. We do induction:

Case 2 (we know $\{\} \vdash e_2 : t_1$):

$$\frac{e_2 \mapsto e_2'}{(\lambda\ x : t_1.\ e_1')\ e_2 \mapsto (\lambda\ x : t_1.\ e_1')\ e_2'}$$

By induction hypothesis, $\{\} \vdash e_2' : t_1$

Thus, we can type $(\lambda\ x : t_1.\ e_1')\ e_2'$ in the same way as $(\lambda\ x : t_1.\ e_1')\ e_2$.

Proving Preservation

- Suppose $\{\} \vdash e : t$ and $e \mapsto e'$. We do induction:

Case 3 ($\{\} \vdash \lambda\, x : t_1.\ e_1' : t_1 \rightarrow t_2$ and $\{\} \vdash v : t_1$):

$$\frac{}{(\lambda\, x : t_1.\ e_1')\ v \mapsto [x \rightarrow v]\ e_1'}$$

To type ($\lambda\, x : t_1.\ e_1'$), we typed it assuming that x had type t_1. Since v **does** have type t_1, that typing judgment will still hold.

PROVING PRESERVATION

- Thus, we have proved Preservation by induction on the step relation.

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}$$

$$\frac{e_2 \mapsto e_2'}{(\lambda\ x : t.\ e_1')\ e_2 \mapsto (\lambda\ x.\ e_1')\ e_2}$$

$$\frac{}{(\lambda\ x : t.\ e_1')\ v \mapsto [x \rightarrow v]\ e_1'}$$

Computational Power, Revisted

- An interesting question: has adding the types changed the computational power?

- (Of course, we can't add more power – the question is, have we lost power!)

Computational Power, Revisted

- An interesting question: has adding the types changed the computational power?
- Answer: yes! In fact, we can no longer express nonterminating computation.
- (see board for why “diverge” is not typable)
- Theorem: If $\{\} \vdash e : t$, then e will step (in some finite number of steps) to a value.
- Proof: (we have mercy and will spare you!)

So what can we do?

- Two basic choices (not mutually exclusive):

1. Add a new kind of term called “fix” as a primitive in the language. “fix” will not be **definable** in the simply-typed calculus, but it will be **usable** in it.
   - This is what is done in most programming languages! Coding up your own recursion technique (e.g., with explicit function pointers) is very rare in real code.
2. Add more complex types
   - For example, recursive types

ADDING MORE TYPES

- There is a tension that results: types are good because they reduce bugs and provide guarantees of safety – but they are bad because they reduce the number of allowable programs.

- Here we have covered two basic types: bool and function. However, real programming languages have lots of other types:
  - Integers
  - Strings
  - Pointers
  - Recursive types
  - Arrays
  - Objects

Adding more types

- For the rest of this lecture, we will focus on one particular kind of addition, which is polymorphic types (also known as generics)

Topics Covered Today

- ~~Untyped lambda calculus~~
- ~~Simply-typed lambda calculus~~
- Polymorphic lambda calculus (System F)

WHY POLYMORPHIC TYPES

- Consider the identity function in the untyped lambda calculus: λx. x

- How would we write this in the typed calculus?

WHY POLYMORPHIC TYPES

- Consider the identity function in the untyped lambda calculus: λx. x
- How would we write this in the typed calculus?
- λ x : bool. x

Why polymorphic types

- Consider the identity function in the untyped lambda calculus: λx. x

- How would we write this in the typed calculus?

- λ x : bool. x
- λ x : bool $\to$ bool. x

Why polymorphic types

- Consider the identity function in the untyped lambda calculus: λx. x

- How would we write this in the typed calculus?

- λ x : bool. x
- λ x : bool $\rightarrow$ bool. x
- λ x : bool $\rightarrow$ (bool $\rightarrow$ bool). x

Why polymorphic types

- Consider the identity function in the untyped lambda calculus: λx. x
- How would we write this in the typed calculus?
- λ x : bool. x
- λ x : bool $\rightarrow$ bool. x
- λ x : bool $\rightarrow$ (bool $\rightarrow$ bool). x
- λ x : (bool $\rightarrow$ bool) $\rightarrow$ bool. x

Why polymorphic types

- Consider the identity function in the untyped lambda calculus: λx. x
- How would we write this in the typed calculus?
- λ x : bool. x
- λ x : bool $\rightarrow$ bool. x
- λ x : bool $\rightarrow$ (bool $\rightarrow$ bool). x
- λ x : (bool $\rightarrow$ bool) $\rightarrow$ bool. x
- λ x : (bool $\rightarrow$ bool) $\rightarrow$ (bool $\rightarrow$ bool). x

Why polymorphic types

- λ x : bool. x
- λ x : bool $\rightarrow$ bool. x
- λ x : bool $\rightarrow$ (bool $\rightarrow$ bool). x
- λ x : (bool $\rightarrow$ bool) $\rightarrow$ bool. x
- λ x : (bool $\rightarrow$ bool) $\rightarrow$ (bool $\rightarrow$ bool). x

- This is very annoying! All of these functions have very similar execution behavior – why do we have to write so many copies?

- Is there something we can do?

Why polymorphic types

- λ x : bool. x
- λ x : bool $\rightarrow$ bool. x
- λ x : bool $\rightarrow$ (bool $\rightarrow$ bool). x
- λ x : (bool $\rightarrow$ bool) $\rightarrow$ bool. x
- λ x : (bool $\rightarrow$ bool) $\rightarrow$ (bool $\rightarrow$ bool). x

- This is very annoying! All of these functions have very similar execution behavior – why do we have to write so many copies?

- Yes – the solution is called polymorphic types.

Adding polymorphic types to the lambda calculus

- First, let's add a few extra terms:

e	=	…	
		Λ t. e	(Type function)
		e [t]	(Type application)

Now let's add some types:

t	=	α	(type variable)
		$\forall\, \alpha$. e	(polymorphic type)

A EXAMPLE

These can seem a little weird, so an example:

The polymorphic identity function is:

$\mathrm{id}_\alpha \quad \equiv \quad \Lambda\, t.\ \lambda\, x : t.\ x$

This function has type $\forall\, \alpha.\ \alpha \rightarrow \alpha$

How do we use id_α?

A EXAMPLE

$\text{id}_\alpha \quad \equiv \quad \Lambda\, t.\ \lambda\, x : t.\ x$

How do we use id_α?

We first apply it to a particular type t. For example:

$\text{id}_\alpha\ [\text{bool}]\ \text{T} \qquad\qquad \equiv$

$(\Lambda\, t.\ \lambda\, x : t.\ x)\ [\text{bool}]\ \text{T} \qquad\qquad \mapsto$

A EXAMPLE

$\text{id}_\alpha \quad \equiv \quad \Lambda\, t.\ \lambda\, x : t.\ x$

How do we use id_α?

We first apply it to a particular type t. For example:

$$\begin{array}{ll}
\text{id}_\alpha\ [\text{bool}]\ T & \equiv \\
(\Lambda\, t.\ \lambda\, x : t.\ x)\ [\text{bool}]\ T & \mapsto \\
(\lambda\, x : \text{bool}.\ x)\ T & \mapsto \\
T &
\end{array}$$

A EXAMPLE

$\mathrm{id}_\alpha \quad \equiv \quad \Lambda\, t.\ \lambda\, x : t.\ x$

How do we use id_α?

$$\begin{array}{ll}
\mathrm{id}_\alpha\ [\mathrm{bool} \to \mathrm{bool}]\ (\mathrm{id}_\alpha\ [\mathrm{bool}]) & \equiv \\
(\Lambda\, t.\ \lambda\, x : t.\ x)\ [\mathrm{bool} \to \mathrm{bool}]\ (\mathrm{id}_\alpha\ [\mathrm{bool}]) & \mapsto \\
(\lambda\, x : \mathrm{bool} \to \mathrm{bool}.\ x)\ (\mathrm{id}_\alpha\ [\mathrm{bool}]) & \mapsto \\
(\lambda\, x : \mathrm{bool} \to \mathrm{bool}.\ x)\ ((\Lambda\, t.\ \lambda\, x : t.\ x)\ [\mathrm{bool}]) & \mapsto \\
(\lambda\, x : \mathrm{bool} \to \mathrm{bool}.\ x)\ (\lambda\, x : \mathrm{bool}.\ x) & \mapsto \\
\lambda\, x : \mathrm{bool}.\ x &
\end{array}$$

Typing rules in System F

Now that we understand a bit better how the terms work in this calculus, the next question is, what do the typing rules look like?

We will extend the context Γ to keep track of which type variables are being used. The empty context {} contains no variables, and we can add a variable X by writing "Γ, X"

TYPING RULES IN SYSTEM F

Now that we understand a bit better how the terms work in this calculus, the next question is, what do the typing rules look like?

There are two rules that are very similar to the rules for function abstraction / application

$$\frac{\Gamma, X \vdash e : t}{\Gamma \vdash \Lambda X.\ e : \forall X.\ t}$$

$$\frac{\Gamma \vdash e_1 : \forall X.\ t_1}{\Gamma \vdash e_1\ [t_2] : [X \to t_2]\ t_1}$$

USING THE TYPING RULES

- Consider the term

$$(\Lambda\ t.\ (\lambda\ f : \text{bool} \to t.\ f\ T))\ [\text{bool}]\ (\lambda\ b : \text{bool}.\ b)$$

Prove that in any context Γ, this has type bool.

(see blackboard!)

Soundness of System F

- Is this calculus sound?
- Yes! And the proof is similar to the one for the simply-typed calculus given earlier:
  - Progress (by induction on typing judgment)
  - Preservation (by induction on step relation)
  - Safety = Progress + Preservation
- You can work out the details if you like.

Expressive power of System F

- The polymorphic lambda calculus is clearly more powerful (can type more programs) than the simply-typed lambda calculus. Is it as powerful as the untyped lambda calculus?
- No – in fact, every program in this calculus will terminate… this is not easy to prove
- In fact, a lot of functions that you would think of as recursive can be expressed here
- General recursion, however, is not possible

QUESTIONS?

- That’s it for this topic!