Foundations of Software
Spring 2025

Week 5

Plan

Corresponding notes: Chapter 8 and 9 of TAPL

1. Some intuitions about types

2. typing arithmetic expressions to prevent errors (interleaved with Lean livecoding)

3. simply typed lambda calculus (STLC) (your upcoming project)

4. STLC and Propositional Logic: Toward Curry-Howard

Types

Problematic arithmetic expressions

Consider the term

$$\texttt{t} = \texttt{if true then (pred false) else 0}$$

Which of the following is true?

1. $\texttt{t}$ is stuck
2. $\texttt{t}$ is a closed term
3. $\texttt{t}$ is typeable, i.e., there exists $\texttt{T}$ such that $\texttt{t} : \texttt{T}$
4. there exists $\texttt{t}'$ such that $\texttt{t} \longrightarrow^* \texttt{t}'$ and $\texttt{t}'$ is stuck

The type of variables in STLC

Consider the term

$$\texttt{t} = \lambda \texttt{x} : \texttt{Bool}.\ \texttt{if x then false else true}$$

What is $\texttt{T}$ in $\texttt{t} : \texttt{T}$?

1. $\texttt{T} = \texttt{Bool}$
2. $\texttt{T} = \texttt{Bool} \to \texttt{Bool}$
3. there are multiple such $\texttt{T}$'s
4. none of the above

Recall: Arithmetic Expressions – Syntax

t ::=		*terms*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
	`0`	*constant zero*
	`succ t`	*successor*
	`pred t`	*predecessor*
	`iszero t`	*zero test*
v ::=		*values*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
nv ::=		*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*

Recall: Arithmetic Expressions – Evaluation Rules

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-PredZero)}$$

$$\texttt{pred (succ } nv_1\texttt{)} \longrightarrow nv_1 \qquad \text{(E-PredSucc)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(E-IszeroZero)}$$

$$\texttt{iszero (succ } nv_1\texttt{)} \longrightarrow \texttt{false} \qquad \text{(E-IszeroSucc)}$$

Recall: Arithmetic Expressions – Evaluation Rules

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(E-If)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \quad \text{(E-Succ)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \quad \text{(E-Pred)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero } t_1 \longrightarrow \texttt{iszero } t_1'} \quad \text{(E-IsZero)}$$

Types

In this language, values have two possible "shapes": they are either booleans or numbers.

`T`	::=	*types*
	`Bool`	*type of booleans*
	`Nat`	*type of numbers*

Types

In this language, values have two possible "shapes": they are either booleans or numbers.

`T`	::=	*types*
	`Bool`	*type of booleans*
	`Nat`	*type of numbers*

Intuitively, things go wrong only when we mistake a bool for a number, i.e. when we do something that "does not make sense".

```
succ true
```

```
iszero true
```

Solution

Problem: Intuitively, things go wrong only when we mistake a bool for a number, i.e. when we do something that "does not make sense" (does not have semantics).

Solution

Problem: Intuitively, things go wrong only when we mistake a bool for a number, i.e. when we do something that "does not make sense" (does not have semantics).

Idea: Forbid programs that do not make sense. But how?

Solution

Problem: Intuitively, things go wrong only when we mistake a bool for a number, i.e. when we do something that "does not make sense" (does not have semantics).

Idea: Forbid programs that do not make sense. But how?

Actionable Solution: Tag every expression to see if it is a bool or a number, and reject program that are passing a bool to `succ`

Typing Rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ } t_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred } t_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero } t_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Typing Rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ}\ t_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred}\ t_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero}\ t_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : \texttt{T} \quad t_3 : \texttt{T}}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : \texttt{T}} \qquad \text{(T-If)}$$

Typing Derivations

Every pair $(\texttt{t}, \texttt{T})$ in the typing relation can be justified by a *derivation tree* built from instances of the inference rules.

$$\dfrac{\dfrac{\dfrac{}{\texttt{0 : Nat}}\ \textsc{T-Zero}}{\texttt{iszero 0 : Bool}}\ \textsc{T-IsZero} \qquad \dfrac{}{\texttt{0 : Nat}}\ \textsc{T-Zero} \qquad \dfrac{\dfrac{}{\texttt{0 : Nat}}\ \textsc{T-Zero}}{\texttt{pred 0 : Nat}}\ \textsc{T-Pred}}{\texttt{if iszero 0 then 0 else pred 0 : Nat}}\ \textsc{T-If}$$

Proofs of properties about the typing relation often proceed by induction on typing derivations.

Imprecision of Typing

Like other *static program analyses*, type systems are generally *imprecise*: they do not predict exactly what kind of value will be returned by every program, but just a conservative (safe) approximation.

$$\frac{\texttt{t}_1 : \texttt{Bool} \qquad \texttt{t}_2 : \texttt{T} \qquad \texttt{t}_3 : \texttt{T}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}} \qquad (\text{T-IF})$$

Using this rule, we cannot assign a type to

```
if true then 0 else pred false
```

```
if true then 0 else false
```

even though these terms will certainly evaluate to a number.

Type Safety

The safety (or soundness) of this type system can be expressed as follows: a well-typed program can never get stuck.
If $\mathtt{t} \ : \ \mathtt{T}$, then $\forall \mathtt{t}', \mathtt{t} \longrightarrow^* \mathtt{t}'$ t' is not stuck
Reminder, t' is not stuck is defined by either:

- t' is a value, i.e. the computation finished.
- Or one can take at least one more step from t':
  $\exists \mathtt{t}'', \mathtt{t'} \longrightarrow \mathtt{t}''$

Type Safety

The safety (or soundness) of this type system can be expressed as follows: a well-typed program can never get stuck.
If $\mathtt{t} \ : \ \mathtt{T}$, then $\forall \mathtt{t}', \mathtt{t} \longrightarrow^* \mathtt{t}'$ t' is not stuck
Reminder, t' is not stuck is defined by either:

- t' is a value, i.e. the computation finished.
- Or one can take at least one more step from t':
  $\exists \mathtt{t}'', \mathtt{t'} \longrightarrow \mathtt{t}''$

In other words, evaluation of a typed term never "goes wrong"!

Decomposing Type Safety

The safety (or soundness) of type systems is very often decompose into two properties:

1. *Progress:* A well-typed term is not stuck
   If $t \ : \ T$, *then either* t *is a value or else* $t \longrightarrow t'$ *for some* t'.

2. *Preservation:* Types are preserved by one-step evaluation
   If $t \ : \ T$ *and* $t \longrightarrow t'$, *then* $t' \ : \ T$.

Decomposing Type Safety

The safety (or soundness) of type systems is very often decompose into two properties:

1. *Progress:* A well-typed term is not stuck
   If $t \ : \ T$, *then either* t *is a value or else* $t \longrightarrow t'$ *for some* t'.

2. *Preservation:* Types are preserved by one-step evaluation
   If $t \ : \ T$ *and* $t \longrightarrow t'$, *then* $t' \ : \ T$.

How do you conclude from these two lemmas?

Decomposing Type Safety

The safety (or soundness) of type systems is very often decompose into two properties:

1. *Progress:* A well-typed term is not stuck
   If $t \ : \ T$, *then either* t *is a value or else* $t \longrightarrow t'$ *for some* t'.

2. *Preservation:* Types are preserved by one-step evaluation
   If $t \ : \ T$ *and* $t \longrightarrow t'$, *then* $t' \ : \ T$.

How do you conclude from these two lemmas? Easy exercise left to

the reader

Inversion

Lemma:

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \texttt{R}$, then $t_1 : \texttt{Bool}$, $t_2 : \texttt{R}$, and $t_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
6. If $\texttt{pred } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
7. If $\texttt{iszero } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $t_1 : \texttt{Nat}$.

Inversion

Lemma:

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \texttt{R}$, then $t_1 : \texttt{Bool}$, $t_2 : \texttt{R}$, and $t_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
6. If $\texttt{pred } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
7. If $\texttt{iszero } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $t_1 : \texttt{Nat}$.

Proof: ...

Inversion

Lemma:

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \texttt{R}$, then $t_1 : \texttt{Bool}$, $t_2 : \texttt{R}$, and $t_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
6. If $\texttt{pred } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
7. If $\texttt{iszero } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $t_1 : \texttt{Nat}$.

Proof: ...

This leads directly to a recursive algorithm for calculating the type of a term...

Typechecking Algorithm

```
typeof(t) = match t with
              | true => Bool
              | false => Bool
              | 0 => Nat
              | succ t1 =>
                let T1 = typeof(t1) in
                if T1 = Nat then Nat else error "not typable"
              | pred t1 =>
                let T1 = typeof(t1) in
                if T1 = Nat then Nat else error "not typable"
              | iszero t1 =>
                let T1 = typeof(t1) in
                if T1 = Nat then Bool else "not typable"
              | ```if t1 then t2 else t3```  =>
            -- This ^ if is the if of the arith language
                let T1 = typeof(t1) in
                let T2 = typeof(t2) in
                let T3 = typeof(t3) in
                if T1 = Bool and T2=T3 then T2
          -- this ^ one is the if of the "host" language
                else error "not typable"
```

Proving progress and preservation

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof:

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof: By induction on a derivation of $\mathtt{t} : \mathtt{T}$.

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof: By induction on a derivation of $\mathtt{t} : \mathtt{T}$.

The T-True, T-False, and T-Zero cases are immediate, since $\mathtt{t}$ in these cases is a value.

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof: By induction on a derivation of $\mathtt{t} : \mathtt{T}$.

The T-True, T-False, and T-Zero cases are immediate, since $\mathtt{t}$ in these cases is a value.

Case T-If: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$

$\mathtt{t}_1 : \mathtt{Bool} \qquad \mathtt{t}_2 : \mathtt{T} \qquad \mathtt{t}_3 : \mathtt{T}$

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof: By induction on a derivation of $\mathtt{t} : \mathtt{T}$.

The T-True, T-False, and T-Zero cases are immediate, since $\mathtt{t}$ in these cases is a value.

Case T-If: $\quad \mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$
$\mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$

By the induction hypothesis, either $\mathtt{t}_1$ is a value or else there is some $\mathtt{t}_1'$ such that $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$. If $\mathtt{t}_1$ is a value, then the canonical forms lemma tells us that it must be either $\mathtt{true}$ or $\mathtt{false}$, in which case either E-IfTrue or E-IfFalse applies to $\mathtt{t}$. On the other hand, if $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$, then, by E-If, $\mathtt{t} \longrightarrow \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$.

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t} : \mathtt{T}$ for some type $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$.

Proof: By induction on a derivation of $\mathtt{t} : \mathtt{T}$.

The cases for rules T-Zero, T-Succ, T-Pred, and T-IsZero are similar.

(Recommended: Try to reconstruct them.)

Preservation

Theorem: If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\mathtt{t}' : \mathtt{T}$.

Preservation

Theorem: If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\mathtt{t}' : \mathtt{T}$.

Proof: By induction on the given typing derivation.

Preservation

Theorem: If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\mathtt{t}' : \mathtt{T}$.

Proof: By induction on the given typing derivation.

Case T-TRUE: $\mathtt{t} = \mathtt{true}$ $\mathtt{T} = \mathtt{Bool}$

Then $\mathtt{t}$ is a value.

Preservation

Theorem: If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\mathtt{t}' : \mathtt{T}$.

Proof: By induction on the given typing derivation.

Case T-If:

$$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \quad \mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$$

There are three evaluation rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

Preservation

Theorem: If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\mathtt{t}' : \mathtt{T}$.

Proof: By induction on the given typing derivation.

Case T-If:

$$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \quad \mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$$

There are three evaluation rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

Subcase E-IfTrue: $\mathtt{t}_1 = \mathtt{true} \qquad \mathtt{t}' = \mathtt{t}_2$

Immediate, by the assumption $\mathtt{t}_2 : \mathtt{T}$.

(E-IfFalse subcase: Similar.)

Preservation

Theorem: If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\mathtt{t}' : \mathtt{T}$.

Proof: By induction on the given typing derivation.

Case T-If:

$\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \quad \mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$

There are three evaluation rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

Subcase E-If: $\mathtt{t}_1 \longrightarrow \mathtt{t}_1' \qquad \mathtt{t}' = \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$

Applying the IH to the subderivation of $\mathtt{t}_1 : \mathtt{Bool}$ yields $\mathtt{t}_1' : \mathtt{Bool}$. Combining this with the assumptions that $\mathtt{t}_2 : \mathtt{T}$ and $\mathtt{t}_3 : \mathtt{T}$, we can apply rule T-If to conclude that $\mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}$, that is, $\mathtt{t}' : \mathtt{T}$.

The Simply Typed Lambda-Calculus

The simply typed lambda-calculus

The system we are about to define is commonly called the *simply typed lambda-calculus*, or $\lambda_{\to}$ for short.

Unlike the untyped lambda-calculus, the "pure" form of $\lambda_{\to}$ (with no primitive values or operations) is not very interesting; to talk about $\lambda_{\to}$, we always begin with some set of "base types."

- So, strictly speaking, there are *many* variants of $\lambda_{\to}$, depending on the choice of base types.
- For now, we'll work with a variant constructed over the booleans.

The simply typed lambda-calculus

The system we are about to define is commonly called the *simply typed lambda-calculus*, or $\lambda_{\to}$ for short.

Unlike the untyped lambda-calculus, the "pure" form of $\lambda_{\to}$ (with no primitive values or operations) is not very interesting; to talk about $\lambda_{\to}$, we always begin with some set of "base types."

- So, strictly speaking, there are *many* variants of $\lambda_{\to}$, depending on the choice of base types.
- For now, we'll work with a variant constructed over the booleans.

What could "go wrong"?

The simply typed lambda-calculus

The system we are about to define is commonly called the *simply typed lambda-calculus*, or $\lambda_{\to}$ for short.

Unlike the untyped lambda-calculus, the "pure" form of $\lambda_{\to}$ (with no primitive values or operations) is not very interesting; to talk about $\lambda_{\to}$, we always begin with some set of "base types."

- So, strictly speaking, there are *many* variants of $\lambda_{\to}$, depending on the choice of base types.
- For now, we'll work with a variant constructed over the booleans.

What could "go wrong"?

```
true false
```

```
λx.true x
```

Untyped lambda-calculus with booleans

t ::=		*terms*
	x	*variable*
	λx.t	*abstraction*
	t t	*application*
	true	*constant true*
	false	*constant false*
	if t then t else t	*conditional*
v ::=		*values*
	λx.t	*abstraction value*
	true	*true value*
	false	*false value*

"Simple Types"

`T`	::=	*types*
	`Bool`	*type of booleans*
	`T`$\rightarrow$`T`	*types of functions*

What are some examples?

Type Annotations

We now have a choice to make. Do we...

- annotate lambda-abstractions with the expected type of the argument

  $$\lambda x{:}T_1.\ t_2$$

  (as in most mainstream programming languages), or
- continue to write lambda-abstractions as before

  $$\lambda x.\ t_2$$

  and ask the typing rules to "guess" an appropriate annotation (as in OCaml)?

Both are reasonable choices, but the first makes the job of defining the typing rules simpler. Let's take this choice for now.

Typing rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

Typing rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

$$\frac{???}{\lambda x{:}T_1 . t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

Typing rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 . t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

Typing rules

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-App)}$$

Typing Derivations

What derivations justify the following typing statements?

- $\vdash$ (λx:Bool.x) true : Bool
- f:Bool→Bool $\vdash$
  f (if false then true else false) : Bool
- f:Bool→Bool $\vdash$
  λx:Bool. f (if x then false else x) : Bool→Bool

Safety of $\lambda_{\rightarrow}$ with Bools

The fundamental property of the type system we have just defined is *soundness* with respect to the operational semantics.

It can be proven by decomposing into proving *progress* and *preservation*.

Safety of $\lambda_{\rightarrow}$ with Bools

The fundamental property of the type system we have just defined is *soundness* with respect to the operational semantics.

It can be proven by decomposing into proving *progress* and *preservation*.

What if we remove the booleans and just keep the core lambda calculus?

Untyped lambda-calculus

`t` ::=		*terms*
	`x`	*variable*
	`λx.t`	*abstraction*
	`t t`	*application*
`v` ::=		*values*
	`λx.t`	*abstraction value*

"Simple Types"

T ::=		*types*
	A1, ..., An	*Base types*
	T→T	*types of functions*

What are some examples?

$$A1 \to A1$$

$$A1 \to A1 \to A2$$

Typing rules for pure STLC

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

Typing rules for pure STLC

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

What does safety of the typesystem means for this language?

Typing rules for pure STLC

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

What does safety of the typesystem means for this language?
Closed terms in pure lambda calculus already do not go wrong!

So, STLC (for pure lambda calculus) is useless?

Seemingly unrelated: Propositional calculus!

Back to early stage of your undergrad education: Propositions:

X	::=	*Atoms*
	A,B,C,...	*Propositional variables*
t	::=	*terms*
	true	*constant true*
	false	*constant false*
	X	*Atom*
	t $\wedge$ t	*And*
	$\neg$ t	*Not*
	t $\vee$ t	*Or*
	t $\Rightarrow$ t	*Implication*
	t $\Leftrightarrow$ t	*Equivalence*

Seemingly unrelated: Propositional calculus!

Back to early stage of your undergrad education: Propositions:

X ::=		*Atoms*
	A,B,C,...	*Propositional variables*
t ::=		*terms*
	true	*constant true*
	false	*constant false*
	X	*Atom*
	t ∧ t	*And*
	¬ t	*Not*
	t ∨ t	*Or*
	t ⇒ t	*Implication*
	t ⇔ t	*Equivalence*
v ::=		*values*
	true	*true value*
	false	*false value*

Seemingly unrelated: Propositional calculus!

Back to early stage of your undergrad education: Propositions:

X ::=		*Atoms*
	A,B,C,...	*Propositional variables*
t ::=		*terms*
	true	*constant true*
	false	*constant false*
	X	*Atom*
	t ∧ t	*And*
	¬ t	*Not*
	t ∨ t	*Or*
	t ⇒ t	*Implication*
	t ⇔ t	*Equivalence*
v ::=		*values*
	true	*true value*
	false	*false value*

Propositional Logic

Logical expressivity of this language: very limited, no natural numbers, no equality, no quantifiers. It is the Duplo of logic (Lego/Logic for children).

Propositional Logic

Logical expressivity of this language: very limited, no natural numbers, no equality, no quantifiers. It is the Duplo of logic (Lego/Logic for children).
We can still have theorems in this language.
We might be interested in proving that the following propositions are tautologies:

$$((\mathtt{A} \Rightarrow \mathtt{B}) \Rightarrow A \Rightarrow B$$

$$((\mathtt{A} \Rightarrow \mathtt{B})\&\&A) \Rightarrow (A\&\&B)$$

Propositional Logic

Logical expressivity of this language: very limited, no natural numbers, no equality, no quantifiers. It is the Duplo of logic (Lego/Logic for children).
We can still have theorems in this language.
We might be interested in proving that the following propositions are tautologies:

$$((\mathtt{A} \Rightarrow \mathtt{B}) \Rightarrow A \Rightarrow B$$

$$((\mathtt{A} \Rightarrow \mathtt{B})\&\&A) \Rightarrow (A\&\&B)$$

Do a truth table! Little issues:

- Exponential size
- Disconnected from the "intuitive proof": in our head we don't do brute force. We do deductions! Let's invent a language to formalize the notion of deduction we do in our head, and let's call it "Natural Deduction".

Propositional Logic

Logical expressivity of this language: very limited, no natural numbers, no equality, no quantifiers. It is the Duplo of logic (Lego/Logic for children).
We can still have theorems in this language.
We might be interested in proving that the following propositions are tautologies:

$$((\mathtt{A} \Rightarrow \mathtt{B}) \Rightarrow A \Rightarrow B$$

$$((\mathtt{A} \Rightarrow \mathtt{B})\&\&A) \Rightarrow (A\&\&B)$$

Do a truth table! Little issues:

- Exponential size
- Disconnected from the "intuitive proof": in our head we don't do brute force. We do deductions! Let's invent a language to formalize the notion of deduction we do in our head, and let's call it "Natural Deduction".

Sales speech for Propositional Logic: Theorems are decidable!

Propositional calculus - Intuitive proof

$$((\mathtt{A} \Rightarrow \mathtt{B}) \Rightarrow A \Rightarrow B$$

Propositional calculus - Intuitive proof

$$((\mathtt{A} \Rightarrow \mathtt{B}) \Rightarrow A \Rightarrow B$$

If we assume that A implies B is true, and we assume that A is true, then necessarily B is true.

Rules of Inference for Implication

$$\frac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\,(\to_i)\,u \qquad \frac{\varphi \qquad \varphi \to \psi}{\psi}\,(\to_e)$$

$$((\mathtt{A} \Rightarrow \mathtt{B}) \Rightarrow A \Rightarrow B$$

Rules of Inference for Conjunction

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi}\,(\wedge_i) \qquad \frac{\varphi \wedge \psi}{\varphi}\,(\wedge_e)$$

$$((\mathtt{A} \Rightarrow \mathtt{B})\&\&A) \Rightarrow (A\&\&B)$$

Toward Curry-Howard Isomorphism

STLC:

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

Toward Curry-Howard Isomorphism

STLC:

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 . t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

First Curry-Howard Isomorphism

Logic side	Programming side
hypotheses	free variables
implication elimination (*modus ponens*)	application
implication introduction	abstraction

Example of Application of Curry-Howard

If we can write a lambda term a lambda term that has type:

$$((\mathtt{A} \to \mathtt{B}) \to A \to B$$

Then we have proven the theorem that the following proposition is a tautology, without having to do a truth table!

$$((\mathtt{A} \Rightarrow \mathtt{B}) \Rightarrow A \Rightarrow B$$

Example of Application of Curry-Howard

If we can write a lambda term a lambda term that has type:

$$((\texttt{A} \to \texttt{B}) \to A \to B$$

Then we have proven the theorem that the following proposition is a tautology, without having to do a truth table!

$$((\texttt{A} \Rightarrow \texttt{B}) \Rightarrow A \Rightarrow B$$

What about:

$$\lambda\texttt{(x: A} \to \texttt{B)(y:A).x y}$$

Already limited?

What about:

$$((\mathtt{A} \Rightarrow \mathtt{B})\&\&A) \Rightarrow (A\&\&B)$$

Already limited?

What about:

$$((\mathtt{A} \Rightarrow \mathtt{B})\&\&A) \Rightarrow (A\&\&B)$$

Logic side	Programming side
formula	type
proof	term
formula is true	type has an element
formula is false	type does not have an element
logical constant $\top$ (truth)	unit type
logical constant $\bot$ (falsehood)	empty type
implication	function type
conjunction	product type
disjunction	sum type
universal quantification	dependent product type
existential quantification	dependent sum type

Product types

What about:

$$((\mathtt{A} \Rightarrow \mathtt{B})\&\&A) \Rightarrow (A\&\&B)$$

$$\lambda(\mathtt{x}\mathtt{:}\ (\mathtt{A} \to \mathtt{B}) \times \mathtt{A}).(\mathtt{fst}\ \mathtt{x})\ (\mathtt{snd}\ \mathtt{x})$$

Curry-Howard's Zoo

Système fonctionnel	Système formel
Calcul des constructions (Thierry Coquand)	Logique intuitionniste d'ordre supérieur
Système F (Jean-Yves Girard)	Arithmétique de Peano du second ordre / Logique intuitionniste du second ordre
Système T (Kurt Gödel)	Arithmétique de Peano du premier ordre / Logique intuitionniste du premier ordre
Système T1	?
T0 (Récursion primitive) (Stephen Cole Kleene ? Thoralf Skolem ?)	Arithmétique primitive récursive
Lambda-calcul simplement typé	Calcul propositionnel minimal implicatif (déduction naturelle)
Logique combinatoire	Calcul propositionnel implicatif (à la Hilbert)
Calcul lambda-μ de Parigot	Déduction naturelle en calcul propositionnel classique
Calcul lambda-μ-μ~ de Curien et Herbelin	Calcul des séquents classique
Calcul symétrique de Berardi et calcul dual de Wadler	Calcul des séquents avec $\vee$ et $\wedge$
Réalisabilité classique / Lambda-calcul avec contrôle / Machine de Krivine	Logique classique du deuxième ordre