Inductive Proofs about the Lambda Calculus

Two induction principles

Like before, we have two ways to prove that properties are true of the untyped lambda calculus.

- Structural induction on terms
- Induction on a derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$.

Let's look at an example of each.

Structural induction on terms

To show that a property $\mathcal{P}$ holds for all lambda-terms $\mathtt{t}$, it suffices to show that

- $\mathcal{P}$ holds when $\mathtt{t}$ is a variable;
- $\mathcal{P}$ holds when $\mathtt{t}$ is a lambda-abstraction $\lambda\mathtt{x}.\ \mathtt{t}_1$, assuming that $\mathcal{P}$ holds for the immediate subterm $\mathtt{t}_1$; and
- $\mathcal{P}$ holds when $\mathtt{t}$ is an application $\mathtt{t}_1\ \mathtt{t}_2$, assuming that $\mathcal{P}$ holds for the immediate subterms $\mathtt{t}_1$ and $\mathtt{t}_2$.

Structural induction on terms

To show that a property $\mathcal{P}$ holds for all lambda-terms $\mathtt{t}$, it suffices to show that

- $\mathcal{P}$ holds when $\mathtt{t}$ is a variable;
- $\mathcal{P}$ holds when $\mathtt{t}$ is a lambda-abstraction $\lambda\mathtt{x.}\ \mathtt{t}_1$, assuming that $\mathcal{P}$ holds for the immediate subterm $\mathtt{t}_1$; and
- $\mathcal{P}$ holds when $\mathtt{t}$ is an application $\mathtt{t}_1\ \mathtt{t}_2$, assuming that $\mathcal{P}$ holds for the immediate subterms $\mathtt{t}_1$ and $\mathtt{t}_2$.

N.b.: The variant of this principle where "immediate subterm" is replaced by "arbitrary subterm" is also valid. (Cf. *ordinary induction* vs. *complete induction* on the natural numbers.)

An example of structural induction on terms

Define the set of *free variables* in a lambda-term as follows:

$$
\begin{aligned}
FV(\mathtt{x}) &= \{\mathtt{x}\} \\
FV(\lambda\mathtt{x}.\mathtt{t}_1) &= FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \\
FV(\mathtt{t}_1\ \mathtt{t}_2) &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2)
\end{aligned}
$$

Define the *size* of a lambda-term as follows:

$$
\begin{aligned}
size(\mathtt{x}) &= 1 \\
size(\lambda\mathtt{x}.\mathtt{t}_1) &= size(\mathtt{t}_1) + 1 \\
size(\mathtt{t}_1\ \mathtt{t}_2) &= size(\mathtt{t}_1) + size(\mathtt{t}_2) + 1
\end{aligned}
$$

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

An example of structural induction on terms

Theorem: $|FV(\texttt{t})| \leq size(\texttt{t})$.

Proof: By induction on the structure of $\texttt{t}$.

- If $\texttt{t}$ is a variable, then $|FV(\texttt{t})| = 1 = size(\texttt{t})$.
- If $\texttt{t}$ is an abstraction $\lambda\texttt{x}.\ \texttt{t}_1$, then

$$
\begin{array}{cll}
 & |FV(\texttt{t})| & \\
= & |FV(\texttt{t}_1) \setminus \{\texttt{x}\}| & \text{by defn} \\
\leq & |FV(\texttt{t}_1)| & \text{by arithmetic} \\
\leq & size(\texttt{t}_1) & \text{by induction hypothesis} \\
< & size(\texttt{t}_1) + 1 & \text{by arithmetic} \\
= & size(\texttt{t}) & \text{by defn.}
\end{array}
$$

An example of structural induction on terms

Theorem: $|FV(\mathtt{t})| \leq size(\mathtt{t})$.

Proof: By induction on the structure of $\mathtt{t}$.

- If $\mathtt{t}$ is an application $\mathtt{t_1}\ \mathtt{t_2}$, then

$$
\begin{array}{lll}
 & |FV(\mathtt{t})| & \\
= & |FV(\mathtt{t_1}) \cup FV(\mathtt{t_2})| & \text{by defn} \\
\leq & |FV(\mathtt{t_1})| + |FV(\mathtt{t_2})| & \text{by arithmetic} \\
\leq & size(\mathtt{t_1}) + size(\mathtt{t_2}) & \text{by IH and arithmetic} \\
< & size(\mathtt{t_1}) + size(\mathtt{t_2}) + 1 & \text{by arithmetic} \\
= & size(\mathtt{t}) & \text{by defn.}
\end{array}
$$

Induction on derivations

Recall that the reduction relation is defined as the smallest binary relation on terms satisfying the following rules:

$$(\lambda \mathtt{x}.\mathtt{t}_1)\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_1 \qquad \text{(E-AppAbs)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

Induction on derivations

Induction principle for the small-step evaluation relation.

To show that a property $\mathcal{P}$ holds for all derivations of $\texttt{t} \longrightarrow \texttt{t}'$, it suffices to show that

- $\mathcal{P}$ holds for all derivations that use the rule E-AppAbs;
- $\mathcal{P}$ holds for all derivations that end with a use of E-App1 assuming that $\mathcal{P}$ holds for all subderivations; and
- $\mathcal{P}$ holds for all derivations that end with a use of E-App2 assuming that $\mathcal{P}$ holds for all subderivations.

An example of induction on derivations

Theorem: if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

We must prove, for all derivations of $\mathtt{t} \longrightarrow \mathtt{t}'$, that $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

An example of induction on derivations

Theorem: if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

Proof: by induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$. There are three cases:

An example of induction on derivations

Theorem: if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

Proof: by induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$. There are three cases:

- If the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$ is just a use of E-AppAbs, then $\mathtt{t}$ is $(\lambda\mathtt{x}.\mathtt{t}_1)\mathtt{v}$ and $\mathtt{t}'$ is $[\mathtt{x} \mapsto \mathtt{v}]\mathtt{t}_1$. Reason as follows:

$$\begin{aligned} FV(\mathtt{t}) &= FV((\lambda\mathtt{x}.\mathtt{t}_1)\mathtt{v}) \\ &= FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \cup FV(\mathtt{v}) \\ &\supseteq FV([\mathtt{x} \mapsto \mathtt{v}]\mathtt{t}_1) \\ &= FV(\mathtt{t}') \end{aligned}$$

An example of induction on derivations

Theorem: if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $FV(\mathtt{t}) \supseteq FV(\mathtt{t}')$.

Proof: by induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$. There are three cases:

- If the derivation ends with a use of E-App1, then $\mathtt{t}$ has the form $\mathtt{t}_1\ \mathtt{t}_2$ and $\mathtt{t}'$ has the form $\mathtt{t}_1'\ \mathtt{t}_2$, and we have a subderivation of $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$

  By the induction hypothesis, $FV(\mathtt{t}_1) \supseteq FV(\mathtt{t}_1')$. Now calculate:

$$\begin{aligned} FV(t) &= FV(\mathtt{t}_1\ \mathtt{t}_2) \\ &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \\ &\supseteq FV(\mathtt{t}_1') \cup FV(\mathtt{t}_2) \\ &= FV(\mathtt{t}_1'\ \mathtt{t}_2) \\ &= FV(\mathtt{t}') \end{aligned}$$

- E-App2 is treated similarly.

Theory of Types
and Programming Languages
Fall 2022

Week 5

Plan

PREVIOUSLY: untyped lambda calculus

TODAY: types!!

1. Two example languages:
   1.1 typing arithmetic expressions
   1.2 simply typed lambda calculus (STLC)
2. For each:
   2.1 Define types
   2.2 Specify typing rules
   2.3 Prove soundness: *progress* and *preservation*

NEXT: lambda calculus extensions
NEXT: polymorphic typing

Types

Outline

1. begin with a set of terms, a set of values, and an evaluation relation
2. define a set of *types* classifying values according to their "shapes"
3. define a *typing relation* `t : T` that classifies terms according to the shape of the values that result from evaluating them
4. check that the typing relation is *sound* in the sense that,
   4.1 if `t : T` and $t \longrightarrow^* v$, then `v : T`
   4.2 if `t : T`, then evaluation of `t` will not get stuck

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

$$\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_2 \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3 \qquad \text{(E-IfFalse)}$$

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

$$\texttt{pred (succ } \texttt{nv}_1\texttt{)} \longrightarrow \texttt{nv}_1 \qquad \text{(E-PredSucc)}$$

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

$$\texttt{iszero (succ } \texttt{nv}_1\texttt{)} \longrightarrow \texttt{false} \qquad \text{(E-IszeroSucc)}$$

Recall: Arithmetic Expressions – Evaluation Rules

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3} \quad \text{(E-If)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{succ}\ \mathtt{t}_1 \longrightarrow \mathtt{succ}\ \mathtt{t}_1'} \quad \text{(E-Succ)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{pred}\ \mathtt{t}_1 \longrightarrow \mathtt{pred}\ \mathtt{t}_1'} \quad \text{(E-Pred)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{iszero}\ \mathtt{t}_1 \longrightarrow \mathtt{iszero}\ \mathtt{t}_1'} \quad \text{(E-IsZero)}$$

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*

Typing Rules

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

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

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

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

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{succ } \texttt{t}_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{pred } \texttt{t}_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{iszero } \texttt{t}_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Typing Derivations

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

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

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.

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

Using this rule, we cannot assign a type to

`if true then 0 else false`

even though this term will certainly evaluate to a number.

Type Safety

The safety (or soundness) of this type system can be expressed by 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 $t' : T$.

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 $\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Bool}$, $\texttt{t}_2 : \texttt{R}$, and $\texttt{t}_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
6. If $\texttt{pred } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
7. If $\texttt{iszero } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $\texttt{t}_1 : \texttt{Nat}$.

Inversion

Lemma:

1. If $\mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
2. If $\mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
3. If $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\mathtt{t}_1 : \mathtt{Bool}$, $\mathtt{t}_2 : \mathtt{R}$, and $\mathtt{t}_3 : \mathtt{R}$.
4. If $\mathtt{0} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$.
5. If $\mathtt{succ}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
6. If $\mathtt{pred}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
7. If $\mathtt{iszero}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$ and $\mathtt{t}_1 : \mathtt{Nat}$.

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 $\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Bool}$, $\texttt{t}_2 : \texttt{R}$, and $\texttt{t}_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
6. If $\texttt{pred } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
7. If $\texttt{iszero } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $\texttt{t}_1 : \texttt{Nat}$.

Proof: ...

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

Typechecking Algorithm

```
typeof(t) = if t = true then Bool
            else if t = false then Bool
            else if t = if t1 then t2 else t3 then
              let T1 = typeof(t1) in
              let T2 = typeof(t2) in
              let T3 = typeof(t3) in
              if T1 = Bool and T2=T3 then T2
              else "not typable"
            else if t = 0 then Nat
            else if t = succ t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = pred t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = iszero t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Bool else "not typable"
```

Properties of the Typing Relation

Recall: Typing Rules

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

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

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

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

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{succ } \texttt{t}_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{pred } \texttt{t}_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{iszero } \texttt{t}_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Recall: 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 $\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Bool}$, $\texttt{t}_2 : \texttt{R}$, and $\texttt{t}_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
6. If $\texttt{pred } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
7. If $\texttt{iszero } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $\texttt{t}_1 : \texttt{Nat}$.

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof:

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v` ::=		*values*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
`nv` ::=		*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*

For part 1,

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v ::=`		*values*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
`nv ::=`		*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*

For part 1, if `v` is `true` or `false`, the result is immediate.

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v` ::=		*values*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
`nv` ::=		*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*

For part 1, if `v` is `true` or `false`, the result is immediate. But `v` cannot be `0` or `succ nv`, since the inversion lemma tells us that `v` would then have type `Nat`, not `Bool`.

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v`	::=		*values*
		`true`	*true value*
		`false`	*false value*
		`nv`	*numeric value*
`nv`	::=		*numeric values*
		`0`	*zero value*
		`succ nv`	*successor value*

For part 1, if `v` is `true` or `false`, the result is immediate. But `v` cannot be `0` or `succ nv`, since the inversion lemma tells us that `v` would then have type `Nat`, not `Bool`. Part 2 is similar.

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, $\mathtt{t : T}$ 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 $\mathtt{t : T}$.

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t : T}$ 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 $\mathtt{t : T}$.

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, $\mathtt{t : T}$ 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 $\mathtt{t : T}$.

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

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

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

Progress

Theorem: Suppose $\mathtt{t}$ is a well-typed term (that is, $\mathtt{t : T}$ 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 $\mathtt{t : T}$.

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

Case T-If: $\quad \mathtt{t} = \mathtt{if\ t_1\ then\ t_2\ else\ t_3}$

$$\mathtt{t_1 : Bool} \qquad \mathtt{t_2 : T} \qquad \mathtt{t_3 : T}$$

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, $\mathtt{t} \longrightarrow \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}$.

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: $\quad \mathtt{t} = \mathtt{true} \qquad \mathtt{T} = \mathtt{Bool}$

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: $\quad \mathtt{t}_1 = \mathtt{true} \qquad \mathtt{t}' = \mathtt{t}_2$

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: $\quad \mathtt{t_1} \longrightarrow \mathtt{t_1'} \qquad \mathtt{t}' = \mathtt{if}\ \mathtt{t_1'}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3}$

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, $\mathtt{t}' : \mathtt{T}$.

Messing With It

Messing with it: Remove a rule

What if we remove E-PredZero ?

Messing with it: Remove a rule

What if we remove E-PredZero ?

Then `pred 0` type checks, but it is stuck and is not a value. Thus the progress theorem fails.

Messing with it: If

What if we change the rule for typing `if`'s to the following?:

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

Messing with it: If

What if we change the rule for typing `if`'s to the following ?:

$$\frac{\mathtt{t_1} : \mathtt{Bool} \qquad \mathtt{t_2} : \mathtt{Nat} \qquad \mathtt{t_3} : \mathtt{Nat}}{\mathtt{if}\ \mathtt{t_1}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3} : \mathtt{Nat}} \qquad \text{(T-IF)}$$

The system is still sound. Some `if`'s do not type, but those that do are fine.

Messing with it: adding bit

$\mathtt{t}$	$::=$		*terms*
		$\ldots$	
		$bit(t)$	*boolean to natural*

What needs to be done?

Messing with it: adding bit

$$
\begin{array}{lll}
\texttt{t} & ::= & \qquad\qquad\qquad\qquad \textit{terms} \\
 & \quad \dots & \\
 & \quad \mathit{bit}(t) & \qquad\qquad\qquad\qquad \textit{boolean to natural}
\end{array}
$$

What needs to be done?

1. new evaluation rules
2. new typing rules

Messing with it: adding bit

$\texttt{t} ::=$ *terms*

$\dots$

$bit(t)$ *boolean to natural*

What needs to be done?

1. new evaluation rules
2. new typing rules
3. progress and preservation updates

Messing with it: adding bit

$\mathtt{t} ::=$ *terms*

$\ldots$

$\mathit{bit}(t)$ *boolean to natural*

What needs to be done?

1. new evaluation rules
2. new typing rules
3. progress and preservation updates

Alternative Approach: Desugaring

$\mathit{bit}(\mathtt{t}) = \mathtt{if\ t\ then\ 1\ else\ 0}$

Messing with it: adding bit

$\mathtt{t}$ ::=		*terms*
	...	
	$bit(t)$	*boolean to natural*

What needs to be done?

1. new evaluation rules
2. new typing rules
3. progress and preservation updates

Alternative Approach: Desugaring

$$bit(\mathtt{t}) = \mathtt{if\ t\ then\ 1\ else\ 0}$$

Need to ensure it follows the intended semantics.

Messing with it: adding bit

`t` ::=		*terms*
	...	
	$bit(t)$	*boolean to natural*

What needs to be done?

1. new evaluation rules
2. new typing rules
3. progress and preservation updates

Alternative Approach: Desugaring

$bit(\texttt{t}) = \texttt{if t then 1 else 0}$

Need to ensure it follows the intended semantics.

Other desugaring example: local variables in lambda calculus...

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 $\lambda_{\rightarrow}$ for short.

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

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

Untyped lambda-calculus with booleans

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

"Simple Types"

`T ::=`		*types*
	`Bool`	*type of booleans*
	`T → T`	*types of functions*

Important: function types are *right-associated*

$T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$, **not** $(T_1 \to T_2) \to T_3$

"Simple Types"

`T ::=`		*types*
	`Bool`	*type of booleans*
	`T → T`	*types of functions*

Important: function types are *right-associated*

$T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$, **not** $(T_1 \to T_2) \to T_3$

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

  $$\lambda \texttt{x:T}_1\texttt{. t}_2$$

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

  $$\lambda \texttt{x. t}_2$$

  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 Context

Γ	$::=$	*contexts*
	ε	*empty context*
	$\Gamma, \mathtt{x}:\mathtt{T}$	*non-empty context*

Typing Context

Γ ::= *contexts*

ε *empty context*

$\Gamma, \mathtt{x}:\mathtt{T}$ *non-empty context*

Definition: write $\mathtt{x}:\mathtt{T} \in \Gamma$ to denote "$\mathtt{x}$ is bound to $\mathtt{T}$ in Γ"

$$\mathtt{x}:\mathtt{T} \in \Gamma, \mathtt{x}:\mathtt{T}$$

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma \qquad \mathtt{x} \neq \mathtt{y}}{\mathtt{x}:\mathtt{T} \in \Gamma, \mathtt{y}:\mathtt{S}}$$

Typing rules

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

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

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

Typing rules

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

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

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

$$\frac{???}{\lambda\texttt{x}{:}\texttt{T}_1.\texttt{t}_2 : \texttt{T}_1{\to}\texttt{T}_2} \qquad \text{(T-Abs)}$$

Typing rules

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

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

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

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

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

Typing rules

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

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{(T-If)}$$

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

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

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

Typing Derivations

Notation: instead of "$\varepsilon \vdash t : \mathtt{T}$", we'll often just write "$\vdash t : \mathtt{T}$"

Typing Derivations

Notation: instead of "$\varepsilon \vdash t : \mathtt{T}$", we'll often just write "$\vdash t : \mathtt{T}$"

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`

Properties of $\lambda_{\rightarrow}$

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

1. *Progress:* A closed, well-typed term is not stuck
   If $\vdash t : T$, *then either* t *is a value or else* $t \longrightarrow t'$ *for some* t'.
2. *Preservation:* Types are preserved by one-step evaluation
   If $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$, *then* $\Gamma \vdash t' : T$.

Proving progress

Same steps as before...

Proving progress

Same steps as before...

- inversion lemma for typing relation
- canonical forms lemma
- progress theorem

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\Gamma \vdash \texttt{t}_1 : \texttt{Bool}$ and $\Gamma \vdash \texttt{t}_2, \texttt{t}_3 : \texttt{R}$.

Inversion

Lemma:

1. If $\Gamma \vdash \mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
2. If $\Gamma \vdash \mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
3. If $\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool}$ and $\Gamma \vdash \mathtt{t}_2, \mathtt{t}_3 : \mathtt{R}$.
4. If $\Gamma \vdash \mathtt{x} : \mathtt{R}$, then

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\Gamma \vdash \texttt{t}_1 : \texttt{Bool}$ and $\Gamma \vdash \texttt{t}_2, \texttt{t}_3 : \texttt{R}$.
4. If $\Gamma \vdash \texttt{x} : \texttt{R}$, then $\texttt{x:R} \in \Gamma$.

Inversion

Lemma:

1. If $\Gamma \vdash \mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
2. If $\Gamma \vdash \mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
3. If $\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool}$ and $\Gamma \vdash \mathtt{t}_2, \mathtt{t}_3 : \mathtt{R}$.
4. If $\Gamma \vdash \mathtt{x} : \mathtt{R}$, then $\mathtt{x}{:}\mathtt{R} \in \Gamma$.
5. If $\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{R}$, then

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\Gamma \vdash \texttt{t}_1 : \texttt{Bool}$ and $\Gamma \vdash \texttt{t}_2, \texttt{t}_3 : \texttt{R}$.
4. If $\Gamma \vdash \texttt{x} : \texttt{R}$, then $\texttt{x}{:}\texttt{R} \in \Gamma$.
5. If $\Gamma \vdash \lambda \texttt{x}{:}\texttt{T}_1 . \texttt{t}_2 : \texttt{R}$, then $\texttt{R} = \texttt{T}_1 \to \texttt{R}_2$ for some $\texttt{R}_2$ with $\Gamma, \texttt{x}{:}\texttt{T}_1 \vdash \texttt{t}_2 : \texttt{R}_2$.

Inversion

Lemma:

1. If $\Gamma \vdash \mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
2. If $\Gamma \vdash \mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
3. If $\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool}$ and $\Gamma \vdash \mathtt{t}_2, \mathtt{t}_3 : \mathtt{R}$.
4. If $\Gamma \vdash \mathtt{x} : \mathtt{R}$, then $\mathtt{x}{:}\mathtt{R} \in \Gamma$.
5. If $\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{R}$, then $\mathtt{R} = \mathtt{T}_1 \to \mathtt{R}_2$ for some $\mathtt{R}_2$ with $\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{R}_2$.
6. If $\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{R}$, then

Inversion

Lemma:

1. If $\Gamma \vdash \mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
2. If $\Gamma \vdash \mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
3. If $\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool}$ and $\Gamma \vdash \mathtt{t}_2, \mathtt{t}_3 : \mathtt{R}$.
4. If $\Gamma \vdash \mathtt{x} : \mathtt{R}$, then $\mathtt{x}{:}\mathtt{R} \in \Gamma$.
5. If $\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{R}$, then $\mathtt{R} = \mathtt{T}_1 \to \mathtt{R}_2$ for some $\mathtt{R}_2$ with $\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{R}_2$.
6. If $\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{R}$, then there is some type $\mathtt{T}_{11}$ such that $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{R}$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$.

Canonical Forms

Lemma:

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.

Canonical Forms

Lemma:

1. If $\mathtt{v}$ is a value of type $\mathtt{Bool}$, then $\mathtt{v}$ is either $\mathtt{true}$ or $\mathtt{false}$.
2. If $\mathtt{v}$ is a value of type $\mathtt{T}_1 \rightarrow \mathtt{T}_2$, then

Canonical Forms

Lemma:

1. If $\mathtt{v}$ is a value of type $\mathtt{Bool}$, then $\mathtt{v}$ is either $\mathtt{true}$ or $\mathtt{false}$.
2. If $\mathtt{v}$ is a value of type $\mathtt{T_1 \to T_2}$, then $\mathtt{v}$ has the form $\lambda\mathtt{x{:}T_1.t_2}$.

Progress

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

Proof: By induction

Progress

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

Proof: By induction on typing derivations.

Progress

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

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because $\mathtt{t}$ is closed). The abstraction case is immediate, since abstractions are values.

Progress

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

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because $\mathtt{t}$ is closed). The abstraction case is immediate, since abstractions are values.
Consider the case for application, where $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathtt{T}_{11} \rightarrow \mathtt{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathtt{T}_{11}$.

Progress

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

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because $\mathtt{t}$ is closed). The abstraction case is immediate, since abstractions are values.
Consider the case for application, where $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathtt{T}_{11}$. By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation, and likewise $\mathtt{t}_2$.

Progress

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

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because $\mathtt{t}$ is closed). The abstraction case is immediate, since abstractions are values.

Consider the case for application, where $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathtt{T}_{11}$. By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation, and likewise $\mathtt{t}_2$. If $\mathtt{t}_1$ can take a step, then rule E-App1 applies to $\mathtt{t}$. If $\mathtt{t}_1$ is a value and $\mathtt{t}_2$ can take a step, then rule E-App2 applies. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then the canonical forms lemma tells us that $\mathtt{t}_1$ has the form $\lambda\mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12}$, and so rule E-AppAbs applies to $\mathtt{t}$.