x = 1

Resources: The slides of this lecture were derived from [Järvi], with permission of the original author, by copy & paste or by selection, annotation, or rewording. [Järvi] is in turn based on [Pierce] as the underlying textbook.

let x = 1 in ...

x(1).

!x(1)

x.set(1)

Programming Language Theory

Type Systems

Ralf Lämmel

[Järvi] Slides by J. Järvi: "Programming Languages", CPSC 604 @ TAMU (2009)
[Pierce] B.C. Pierce: Types and Programming Languages, MIT Press, 2002

Quote

A type system is a tractable syntactic method for proving the absence of certain program behaviors by **classifying phrases according to the kinds of values they compute**. [B.C. Pierce]

Meaningless programs

- *While* programs of arguable use
  - **`while true do skip`** (loops indefinitely)
  - **`a := a + 1;`** (gets stuck because **`a`** may be undefined)
- Type systems are meant to reject (some) meaningless programs.

"C way" of dealing with meaningless programs

- Reject some meaningless programs at compile time.

```
char* p = 1;
```

- Allow some meaningless programs w/o well-defined behavior.

```
union { char* p; int i; } my_union;
void foo() {
    my_union.i = 1;
    char* p = my_union.p;
    *p = 'a';
}
```

"Java way" of dealing with meaningless programs

- Reject some meaningless programs at compile time.

```
int i = "Erroneous";
```

- Reject additional programs at runtime.

```
Stack s = new MyStack();
s.push("foo");
int i = (int)s.pop();
```

"Scheme way" of dealing with meaningless programs

- Reject none meaningless programs at compile time.
- Reject many programs at runtime.

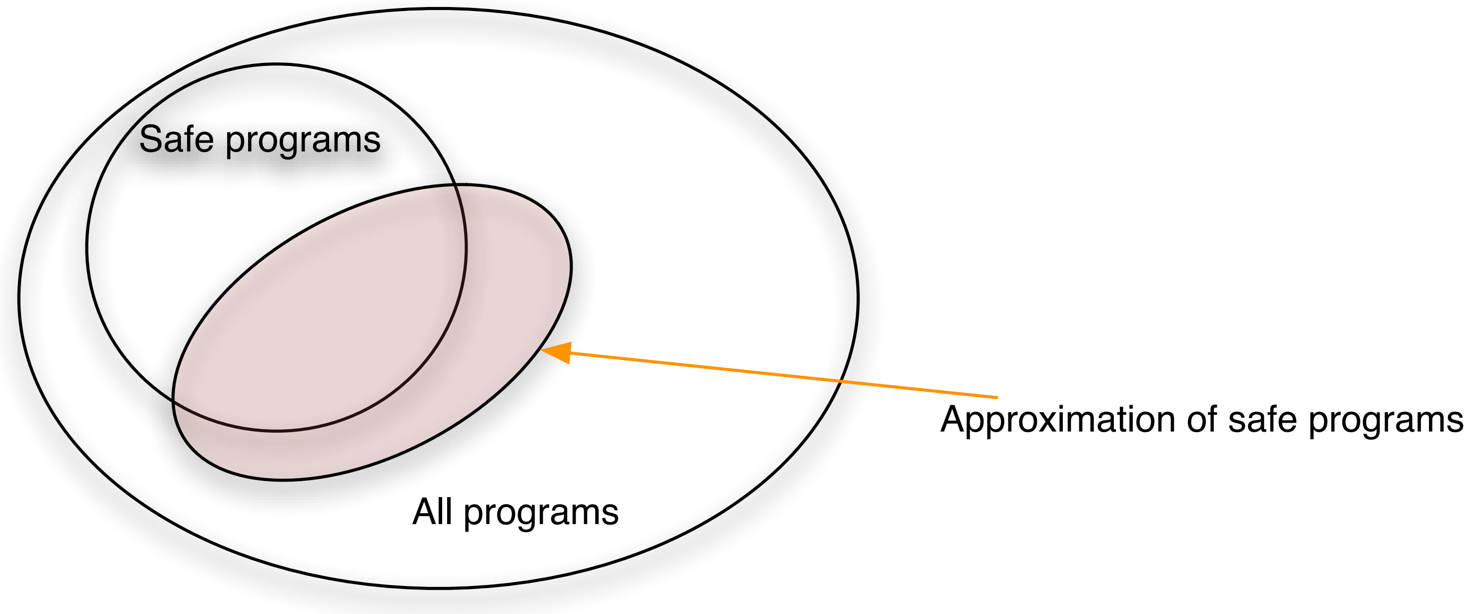
```
(car (cons 1 2))   ; ok

(car 5)            ; error at run-time
```

- (Makes it easy to move between data and code.)

What programs to reject when?

- Reject all meaningless programs at compile time?
  - ✦ Other than by rejecting too many programs?
- Reject no meaningful programs at compile time?
  - ✦ This is impossible due to undecidability issues.
    - ★ Think of nontermination or division-by-zero.
- "Exact" type checking rules out important idioms.
  - ✦ Think of de-/serialization, reflection, etc.

What programs to reject when?



Type systems

A type is a set of terms.

- Define syntax.
- Define semantics.
- Define syntax of type expressions.
- **Categorize syntactic categories by types.**
    - **Use a rule-based system as in semantics.**
- **Prove type safety.**

Use Pierce's B, NB languages for today!

Introducing B and NB

- Languages
  - ✦ B ... Booleans
  - ✦ NB ... Naturals and Booleans
- Syntax definitions of B, NB
  - ✦ Grammar-style definition
  - ✦ Inductive rules (several styles)
  - ✦ Horn clauses (logic program)

Meaningless NB terms

- `iszero true`
- `if 0 then 1 else 2`
- `if true then 1 else false`

Syntax of the B language

- Grammar:

t ::=		terms:
	`true`	constant true
	`false`	constant false
	`if` t_1 `then` t_2 `else` t_3	conditional

- Defines a set of terms, and t ranges over those terms.
- Item t is a metavariable (as opposed to a variable of B).
- Term and expression mean the same thing for now.

Syntax of the NB language

t ::=		terms:
	`true`	constant true
	`false`	constant false
	`if` t_1 `then` t_2 `else` t_3	conditional
	`0`	constant zero
	`succ` t	successor
	`pred` t	predecessor
	`iszero` t	test for zero

Defining terms with inductive rules

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad \texttt{0} \in \mathcal{T} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T} \qquad \texttt{t}_2 \in \mathcal{T} \qquad \texttt{t}_3 \in \mathcal{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}}$$

Syntax definition based on Horn clauses

```
term(true).
term(false).
term(zero).
term(succ(T)) :- term(T).
term(pred(T)) :- term(T).
term(iszero(T)) :- term(T).
term(if(T1,T2,T3)) :- term(T1), term(T2), term(T3).
```

Effective as a syntax check

Semantics of B and NB

- Big-step semantics
- Small-step semantics
- Some properties
- Normal forms / values

Big-step semantics of B

B-True

$$\texttt{true} \Downarrow \texttt{true}$$

B-False

$$\texttt{false} \Downarrow \texttt{false}$$

B-IfTrue

$$\frac{\texttt{t}_1 \Downarrow \texttt{true} \qquad \texttt{t}_2 \Downarrow \texttt{t}_2'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \Downarrow \texttt{t}_2'}$$

B-IfFalse

$$\frac{\texttt{t}_1 \Downarrow \texttt{false} \qquad \texttt{t}_3 \Downarrow \texttt{t}_3'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \Downarrow \texttt{t}_3'}$$

Exercising the semantics

- Are these terms the same?
  - ✦ if true then false else true
  - ✦ if false then true else (if true then false else true)
- In a syntactic sense? No.
- In a semantic sense? Perhaps?

if true then false else true
= if false then true else (if true then false else true) ?

- Meaning of `if true then true else false`:

$$\dfrac{\texttt{true} \Downarrow \texttt{true}\ \text{B-True} \qquad \texttt{false} \Downarrow \texttt{false}\ \text{B-False}}{\texttt{if true then false else true} \Downarrow \texttt{false}}\ \text{B-IfTrue}$$

- Meaning of `if false then true else (if true then false else true)`:

$$\dfrac{\text{false} \Downarrow \text{false}\ \text{B-False} \qquad \dfrac{\text{true} \Downarrow \text{true}\ \text{B-True} \qquad \text{false} \Downarrow \text{false}\ \text{B-False}}{\text{if true then false else true} \Downarrow \text{false}}\ \text{B-IfTrue}}{\text{if false then true else (if true then false else true)} \Downarrow \text{false}}\ \text{B-IfFalse}$$

A property of the semantics

- **Theorem: Evaluation is a total function.**
- Proof:
  - ✦ Lemma: Evaluation is deterministic.
  - ✦ Lemma: Every term evaluates to something.
  - ✦ Totality trivially follows.

Exercise for you

Lemma (Evaluation is deterministic)

$\mathcal{E}$ is a partial function. That is, if $t \Downarrow t_1$ and $t \Downarrow t_2$ then $t_1 = t_2$.

Proof.

By induction on t. Let $P(t) \stackrel{\text{def}}{=} (t \Downarrow t_1 \wedge t \Downarrow t_2) \implies t_1 = t_2$.
Base cases, Case: $t = \mathtt{true}$. The only rule matching $\mathtt{true}$ is $\mathtt{true} \Downarrow \mathtt{true}$, thus $P(\mathtt{true})$ holds. Case: $t = \mathtt{false}$. Similar.
Case: $t = \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3$. From $P(t_1)$, if for all t_1', $t_1 \not\Downarrow t_1'$, no rule matches and thus $P(t)$ holds vacuously. Assume then $t_1 \Downarrow t_1'$, which is unique by $P(t_1)$.

1. If $t_1' = \mathtt{true}$ and either $t_2 \Downarrow t_2'$ for some unique t_2', or for all t_2', $t_2 \not\Downarrow t_2'$. In the first case, $t \Downarrow t_2'$, in the second, for all t', $t \not\Downarrow t'$. $P(t)$ thus holds.
2. If $t_1' = \mathtt{false}$ similar.
3. If t_1' is neither $\mathtt{true}$ or $\mathtt{false}$, no rule applies and thus $P(t)$ holds vacuously.

□

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Exercise for you

Lemma (Every term evaluates to something)

For all $t \in \mathcal{B}$, there exists a term $t' \in \mathcal{B}$, such that $t \Downarrow t'$.

Proof.

By structural induction on t. Let's make a slightly stronger induction hypothesis:
$P(t) \stackrel{\text{def}}{=} (t \Downarrow \texttt{true} \vee t \Downarrow \texttt{false})$.
Cases: $t = \texttt{true}$, $t = \texttt{false}$. Trivial.
Case: $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$. By induction hypothesis either

- $t_1 \Downarrow \texttt{true}$. Then further by i.h., either
  - $t_2 \Downarrow \texttt{true}$, and thus $t \Downarrow \texttt{true}$, or
  - $t_2 \Downarrow \texttt{false}$, and thus $t \Downarrow \texttt{false}$.
- $t_1 \Downarrow \texttt{false}$. Then further by i.h., either
  - $t_3 \Downarrow \texttt{true}$, and thus $t \Downarrow \texttt{true}$, or
  - $t_3 \Downarrow \texttt{false}$, and thus $t \Downarrow \texttt{false}$.

Thus $P(t)$ holds. As P implies the original property (t evaluates to some term), the lemma follows.

Recall syntax of the NB language

t ::=		terms:
	`true`	constant true
	`false`	constant false
	`if` t_1 `then` t_2 `else` t_3	conditional
	`0`	constant zero
	`succ` t	successor
	`pred` t	predecessor
	`iszero` t	test for zero

In order to define the evaluation relation for this language concisely, it is useful to define a few syntactic categories, and give them distinct metavariables.

Refined syntax definition with categories of *values*

t	::=		terms:
		v	value
		if t_1 then t_2 else t_3	conditional
		succ t	successor
		pred t	predecessor
		iszero t	test for zero
v	::=		values:
		true	constant true
		false	constant false
		nv	numeric value
nv	::=		numeric values:
		0	zero value
		succ nv	successor value

Big-step semantics of NB

B-Value

$$\texttt{v} \Downarrow \texttt{v}$$

N-Succ

$$\frac{t \Downarrow nv}{\texttt{succ}\ t \Downarrow \texttt{succ}\ nv}$$

N-IszeroZero

$$\frac{t \Downarrow 0}{\texttt{iszero}\ t \Downarrow \texttt{true}}$$

N-IszeroSucc

$$\frac{t \Downarrow \texttt{succ}\ nv}{\texttt{iszero}\ t \Downarrow \texttt{false}}$$

N-PredZero

$$\frac{t \Downarrow 0}{\texttt{pred}\ t \Downarrow 0}$$

N-PredSucc

$$\frac{t \Downarrow \texttt{succ}\ nv}{\texttt{pred}\ t \Downarrow nv}$$

B-IfTrue

$$\frac{t_1 \Downarrow \texttt{true} \qquad t_2 \Downarrow v_2}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \ \Downarrow v_2}$$

B-IfFalse

$$\frac{t_1 \Downarrow \texttt{false} \qquad t_3 \Downarrow v_3}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \ \Downarrow v_3}$$

The choices of metavariables are significant.

Small-step semantics of NB

E-Iszero

$$\frac{t \longrightarrow t'}{\texttt{iszero } t \longrightarrow \texttt{iszero } t'}$$

E-IszeroZero

$$\texttt{iszero 0} \longrightarrow \texttt{true}$$

E-IszeroSucc

$$\texttt{iszero (succ } nv\texttt{)} \longrightarrow \texttt{false}$$

E-Succ

$$\frac{t \longrightarrow t'}{\texttt{succ } t \longrightarrow \texttt{succ } t'}$$

E-Pred

$$\frac{t \longrightarrow t'}{\texttt{pred } t \longrightarrow \texttt{pred } t'}$$

E-PredZero

$$\texttt{pred 0} \longrightarrow \texttt{0}$$

E-PredSucc

$$\texttt{pred (succ } nv\texttt{)} \longrightarrow nv$$

E-IfTrue

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2$$

E-IfFalse

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3$$

E-If

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$$

Do $\mathcal{B}$'s properties carry over to $\mathcal{NB}$?

Lemma ((?) Evaluation is deterministic)

Evaluation relation is a partial function. That is, if $t \Downarrow t_1$ and $t \Downarrow t_2$ then $t_1 = t_2$.

Yes

Lemma ((?) Every term evaluates to something)

For all $t \in \mathcal{NB}$, there exists a term $t' \in \mathcal{NB}$, such that $t \Downarrow t'$.

No

Counter example for 2nd claim:

`iszero true`

(So we are getting stuck.)

Type system

Think of a type as a set of terms.

- Can't we use syntax for typing?
- Components of a type system
  - ✦ Types (type expressions) for NB
  - ✦ Type relation for NB
  - ✦ Typing rules for NB

Syntactic categories as types

```
bterm(true).
bterm(false).
bterm(iszero(T)) :- nterm(T).

nterm(zero).
nterm(succ(T)) :- nterm(T).
nterm(pred(T)) :- nterm(T).
```

b ... boolean
n ... number

How to model "if"?

Types in NB

`T ::=`		types:
	`Bool`	the Boolean type
	`Nat`	the type of numeric values

Informally by saying "term t is of type T", we imply that we can see (without evaluating t) that t evaluates to some normal form t' which has type T.

Typing relation

- The notation for t is of type T is:
  $t : T$
  or
  $t \in T$
- And more commonly:
  $\Gamma \vdash t : T$
  where Γ is the context, or typing environment

To be defined by typing rules

Not needed for NB (which has no names)

NB typing rules

T-True

$$\texttt{true} : \texttt{Bool}$$

T-False

$$\texttt{false} : \texttt{Bool}$$

T-If

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T}$$

T-Zero

$$\texttt{0} : \texttt{Nat}$$

T-Succ

$$\frac{t : \texttt{Nat}}{\texttt{succ } t : \texttt{Nat}}$$

T-Pred

$$\frac{t : \texttt{Nat}}{\texttt{pred } t : \texttt{Nat}}$$

T-Iszero

$$\frac{t : \texttt{Nat}}{\texttt{iszero } t : \texttt{Bool}}$$

We say that a term t is typ(e)able, or well-typed if there is some T such that $t : T$.

Examples

- What are the types of these terms?
  - **`succ (succ 0)`**
  - **`if iszero 0 then 0 else succ 0`**
  - **`if iszero 0 then 0 else false`**
- Draw the derivation trees.

succ (succ 0)

Derivation tree

$$\dfrac{\dfrac{\mathtt{0} : \mathtt{Nat}}{\mathtt{succ}\ \mathtt{0} : \mathtt{Nat}}\ \text{T-Succ}}{\mathtt{succ}\ (\mathtt{succ}\ \mathtt{0}) : \mathtt{Nat}}\ \text{T-Succ}$$

Typing rules

T-Zero

$$\mathtt{0} : \mathtt{Nat}$$

T-Succ

$$\dfrac{t : \mathtt{Nat}}{\mathtt{succ}\ t : \mathtt{Nat}}$$

`if iszero 0 then 0 else succ 0`

$$\dfrac{\dfrac{0 : \mathrm{Nat}\ \text{T-Zero}}{\texttt{iszero}\ 0 : \mathrm{Bool}}\ \text{T-Succ} \qquad 0 : \mathrm{Nat}\ \text{T-Zero} \qquad \dfrac{0 : \mathrm{Nat}\ \text{T-Zero}}{\texttt{succ}\ 0 : \mathrm{Nat}}\ \text{T-Succ}}{\texttt{if iszero 0 then 0 else succ 0} : \mathrm{Nat}}\ \text{T-If}$$

if iszero 0 then 0 else false

$$\dfrac{\dfrac{\mathtt{0} : \mathsf{Nat}\ \text{T-Zero}}{\mathtt{iszero}\ \mathtt{0} : \mathsf{Bool}}\ \text{T-Succ} \qquad \mathtt{0} : T(?) \qquad \mathtt{false} : T(?)}{\mathtt{if}\ \mathtt{iszero}\ \mathtt{0}\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ \mathtt{false} : \mathsf{Nat}}\ \text{T-If}$$

Uniqueness of types

No term has more than one type. That is, if $t : T_1$ and $t : T_2$, then $T_1 = T_2$.

This is clearly a desirable property.

Theorem (Uniqueness of types)

No term has more than one type. That is, if $t : T_1$ *and* $t : T_2$*, then* $T_1 = T_2$.

Proof.

By induction on the structure of t (using inversion lemma). □

See next slide.

- In fact, a stronger property holds for $\mathcal{NB}$:

Theorem (Uniqueness of typing derivations)

If $t : T_1$ *and* $t : T_2$*, then the typing derivations of* $t : T_1$ *and* $t : T_2$ *are equal.*

Inversion

Exercise for you

The Inversion lemma reads the typing relation backwards, allowing us to limit the possible types for many terms (by looking at their top-level syntactic form)

Lemma (Inversion of typing relation)

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

Proof.

Follows directly from the typing relation.

About uniqueness

- Uniqueness theorem does not hold for more complex languages.
- Consider, for example, a system with subtyping:

```
class A { ... };

class B extends A { ... };

B b; // b has both type B and type A
```

A key property: **Type safety**, aka soundness

- Definition (first attempt)

  Each well-typed term evaluates to a value.

  Evaluation does not get stuck.

- Challenges for this (simplified) definition
  - Nontermination
  - Disagreement between predicted and actual type

Type safety

- Type safety = progress + preservation
  - Progress:

    A well typed term is either a value, or some evaluation rule applies.
  - Preservation:

    Evaluation relation preserves well-typedness of a term.

Progress (first side of type safety)

Exercise for you

Theorem (Progress)

Assume $t : T$ (i.e., t is well-typed). Then, either t is a value, or $t \longrightarrow t'$ for some t'.

Proof.

By induction on typing derivation $t : T$. Trivial if the last rule used is T-True, T-False, or T-Zero (t is a value).

Case T-If: t is of the form $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$, where $t_1 : \texttt{Bool}$, $t_2 : T$, and $t_3 : T$. By the induction hypothesis, t_1, t_2, and t_3 each either are values or evaluate (respectively) to some terms t_1', t_2', and t_3'. If t_1 is a value, from the canonical forms lemma we see it must be either `true` or `false`, and thus either $t \longrightarrow t_2$ or $t \longrightarrow t_3$ using E-IfTrue or E-IfFalse. If $t_1 \longrightarrow t_1'$, then $t \longrightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3$ by E-If.

Recall

T-If

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}$$

E-IfTrue

$$\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_2$$

E-IfFalse

$$\texttt{if false then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_3$$

Canonical forms

This lemma allows us to limit the shapes of terms (in fact, terms that are values) of different types.

Lemma (Canonical forms)

1. *If v is a value and has type* `Bool`*, then v is either* `true` *or* `false`*.*
2. *If v is a value and has type* `Nat`*, then v is a numeric value as specified in our grammar.*

Proof.

Immediate from the grammar and inversion lemma. □

Progress **cont'd**

Exercise for you

Theorem (Progress)

Assume $t : T$ (i.e., t is well-typed). Then, either t is a value, or $t \longrightarrow t'$ for some t'.

Proof.

Case T-Pred: t is of the form $\texttt{pred}\ t_1$, where $t_1 : \texttt{Nat}$. By the induction hypothesis, t_1 is either a value or evaluates to some term t_1'. If t_1 is a value, from the canonical forms lemma we see it must be a numeric value, and thus either $t_1 = \texttt{0}$ or $t_1 = \texttt{succ}\ nv$. If $t_1 = \texttt{0}$, then $t = \texttt{pred 0} \longrightarrow \texttt{0}$ using the rule E-PredZero. If $t_1 = \texttt{succ}\ nv$, then $t = \texttt{pred (succ}\ nv\texttt{)} \longrightarrow nv$. If $t_1 \longrightarrow t_1'$, then rule E-Pred applies and $t = \texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'$.

Case T-Succ: Exercise.

□

Preservation (second side of type safety)

- Preservation theorem is also known as subject reduction:

  Theorem (Preservation of well-typedness)

  If $t : T$ *and* $t \longrightarrow t'$, *then* $t' : T'$, *for some* T'.

- For NB, we can prove a stronger preservation theorem:

  Theorem (Preservation of typing)

  If $t : T$ *and* $t \longrightarrow t'$, *then* $t' : T$.

Proof of preservation property

Exercise for you

Theorem (Preservation of typing)

If $t : T$ and $t \longrightarrow t'$, then $t' : T$.

By induction on typing derivation $t : T$.
Vacuously true for T-True, T-False, and T-Zero.
Case T-If: t is of the form $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$, where $t_1 : \texttt{Bool}$, $t_2 : T$, and $t_3 : T$. There are three possible rules for $t \longrightarrow t'$:

1. If $t_1 = \texttt{true}$, by E-IfTrue t evaluates to t_2 which is of type T.
2. If $t_1 = \texttt{false}$, by E-IfFalse t evaluates to t_3 which has type T.
3. Otherwise E-If must apply and $t_1 \longrightarrow t_1'$ for some t_1'. By induction hypothesis, t_1' is of the same type as t_1: type $\texttt{Bool}$. Thus $t' = \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3$, where $t_1' : \texttt{Bool}$, $t_2 : T$, and $t_3 : T$. The type of this t' is thus T.

Cases T-Pred and T-Succ omitted.

- **Summary**: *Type systems*
  - *Reject meaningless programs.*
  - *Use a rule-based specification, again.*
  - *Type safety relates semantics and type system.*
- **Prepping**: *"Types and Programming Languages"*
  - *Chapters 1, 3 and 8*
- **Outlook**:
  - *The lambda calculus*