TYPE SYSTEMS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from Chapter 8 of the book "Types and Programming Languages", by Benjamin C. Pierce.

What are Type Systems

- A type system is a lightweight method to ensure that a system behaves according to a given specification.
  - There are other lightweight methods: dataflow analyses and control flow analyses, for instance.

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.

- Types were not an invention of computer science.
  - They came from this intersection between logic, mathematics and philosophy that became fashionable in the early 20th century.

What are type systems good for?

Type Theory

- The theory of types was introduced by Bertrand Russell, as a way to solve paradoxes that were plaguing the effort to ground mathematics.
- In the early 20th century, this effort was mostly concentrated in Set Theory.
  - But set theory had a few problems with self-referential sets.

Some sets contain themselves. For instance, the set of all the ideas is an idea itself. There are sets that do not contain themselves. Like the set of every coin – it is not a coin itself.

What about **the set of all the sets S with the following property: "S does not contain itself"**. Does this set exist?†

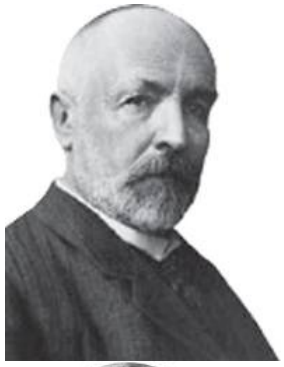
†: Logicomix, a novel by Apostolos Doxiadis and Christos Papadimitriou

Self References and Paradoxes

- Whenever something talks about itself, e.g, it makes a *self-reference*, we may be entering the realm of paradoxes:
  - The set of all the sets that do not contain themselves.
  - The book that cites all (and only) the books that do not cite themselves.
  - The person who shaves every person who does not shave him/herself.
  - Is the answer to this question "no"?
  - This statement is false!
  - What happens when Pinocchio says: "my nose will grow now"?

Sets and Types

- Georg Cantor gave origin to Set Theory between 1874 and 1884.
- Much of the effort to ground mathematics was based on set theory.
  - As an example, we have that Frege's Begriffsschrift (1879) relied heavily on set theory.
- Bertrand Russell came with the Set Paradox.
  - Transforming Set Theory in "Naïve Set Theory".
  - Russell then came with a theory of types to solve this problem sometime around 1903-1908.
  - This theory has evolved substantially since then.
- Eventually Kurt Goedel showed that these purely arithmetical systems are either unsound or incomplete.




Types and Self-Reference

- We can solve some of these paradoxes that rely on self-reference by defining a type system: things of higher types can refer to things of lower types.
- We establish a caste system in the kingdom of barbers: barbers of cast n can shave barbers of cast k, k > n.
  - We end up getting some very hirsute barbers in cast 1.

```
Standard ML of New Jersey
- fun f _ = f f;
Error: operator and operand don't agree [circularity]
  operator domain: 'Z
  operand:         'Z -> 'Y
  in expression:
    f f
-
```

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Why Type Systems are Useful

- Type systems are useful in a variety of ways:
  - Enforcing disciplined programming via type abstractions
  - giving developers the tools to document programs. Compilers can "check" this documentation, preventing inconsistencies.
  - Enabling the generation of more efficient machine code.
  - Allowing early detection of some programming errors.
    - Type systems lets us prove, formally, that programs will not contain a vast set of runtime errors.
    - In other words, well-typed programs will not go wrong due to, for instance, inconsistent operations.
    - This will be the focus on this presentation.

How could we use types to prove that an algorithm is correct?

Solving Jigsaws

- We will show how types can be used to prove that an algorithm that solves jigsaws is correct.
  - We will be giving only a bit of intuition here♤.

- A jigsaw is defined by a number of things:
  - A board, which must be filled with pieces.
  - A set of pieces B, which are already on the board.
  - A set of pieces O waiting to be placed on the board.

- An algorithm A that solves a jigsaw J at each step takes a piece outside the board, e.g., from O, and moves it onto the board, thus, placing it at B.
  - The algorithm is allowed to re-arrange pieces on the board, but it cannot backtrack: it cannot exchange a piece on the board by a piece outside the board.

How to prove that this algorithm is correct?

♤: For a formal and complete explanation about this example, see the PhD dissertation "Register Allocation by Puzzle Solving", UCLA, 2008

Solving Jigsaws

Why are these three conditions enough?

- An algorithm A that solves a jigsaw J at each step takes a piece outside the board, e.g., from O, and moves it onto the board, thus, placing it at B.
- Let's define the notion of a solvable state:
  - The system is in a solvable state if there is a way to move the n pieces in O onto the board in n steps, so that at the end of the n^{th} step all the pieces will be on the board.
- If we want to show that the algorithm is correct, we need to prove the following properties:
  1. If we are in a solvable state, the algorithm can take a step
  2. If we are in a solvable state, and the algorithm takes a step, then it produces a solvable state
  3. The number of steps is finite.

Progress and Preservation

- If we are in a solvable state, the algorithm can take a step.
  - This property is called *progress*.
- If we are in a solvable state, and the algorithm takes a step, then it produces a solvable state
  - This property is called *preservation*.
- The number of steps is finite
  - This property is called *termination*.

- In general we cannot guarantee termination of programs, but type systems give us the tools to prove progress and preservation.

But, what do jigsaws have to do with types?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Solvable States and Types

- Let's say that a solvable state has the type *Solvable* (S).
- And let's say that an unsolvable state has the type *Unsolvable* (U).
- If we have a way to "type-check" a state, then we can tell if that state has type S or U.
- If we can prove that the algorithm preserves types, at each step, then we are done with preservation.
- If we can show that the algorithm always takes a step, given a puzzle of type S, then we are done with progress.
- Finally, if we can show that each step reduces some notion of size, e.g., the number of pieces outside the board, then we are done with termination.

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Typing Rules

fernando@dcc.ufmg.br

Typed Arithmetic Expressions

- We shall leave jigsaws behind, and focus on our language of arithmetic expressions:

Syntax

t ::=
  true
| false
| if t then t else t
| 0
| succ t
| pred t
| iszero t

v ::=
  true
| false
| nv

nv ::=
  0
| succ nv

Semantics

$$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'}$$ [E-SUCC]

$$\text{pred } 0 \rightarrow 0$$ [E-PREDZERO]

$$\text{pred}(\text{succ } nv_1) \rightarrow nv_1$$ [E-PREDSUCC]

$$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'}$$ [E-PRED]

$$\text{iszero } 0 \rightarrow \text{true}$$ [E-ISZEROZERO]

$$\text{iszero (succ nv1)} \rightarrow \text{false}$$ [E-ISZEROSUCC]

$$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'}$$ [E-ISZERO]

$$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2$$ [E-IFTRUE]

$$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3$$ [E-IFFALSE]

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$ [E-IF]

Do you remember the notion of stuck term? Can you still give examples of terms that are stuck?

Stuck Terms

- A term is stuck if it is in normal form, but it is not a value.
- Stuck terms are bad: they correspond to meaningless or erroneous programs, e.g.,
  - pred true; if 0 then true else false; iszero false; etc

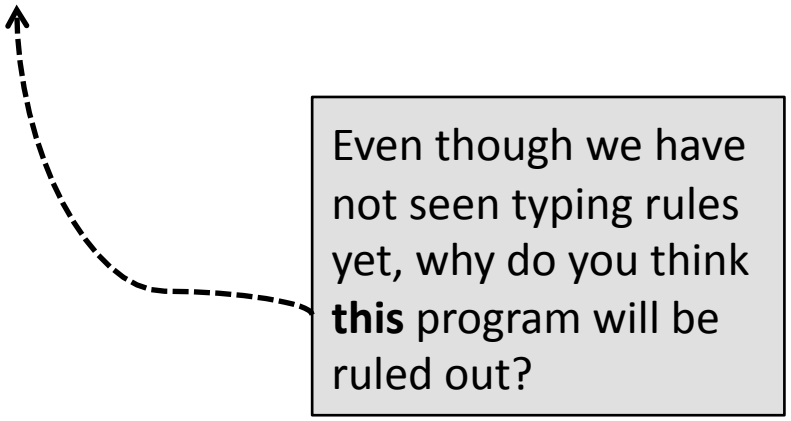
PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Stuck Terms and Types

- A term is stuck if it is in normal form, but it is not a value.
- Stuck terms are bad: they correspond to meaningless or erroneous programs, e.g.,
  - pred true; if 0 then true else false; iszero false; etc
- Types gives us a static way to tell if a program may become stuck or not.
- If a program in our toy language "type-checks", then we know that this term will eventually evaluate to a value.

Conservative Type System

- Type systems, like the other static analyses that we have seen, are conservative.
- They rule some programs out, even though they will evaluate to values.
- As an example, the program below does not type-check, even though it leads to a value:
  - if true then 0 else false



Typing Rules

Syntax	*Typing Rules*	
T ::=	true: Bool	[T-TRUE]
Bool		
Nat	false: Bool	[T-FALSE]
	$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T}$	[T-IF]
	0: Nat	[T-ZERO]
	$\frac{t_1: \text{Nat}}{\text{succ } t_1: \text{Nat}}$	[T-SUCC]
	$\frac{t_1: \text{Nat}}{\text{pred } t_1: \text{Nat}}$	[T-PRED]
	$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}}$	[T-ISZERO]

Typing Rules

Syntax

T ::=
  Bool
  Nat

Typing Rules

true: Bool [T-TRUE]

false: Bool [T-FALSE]

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T} \quad [\text{T-IF}]$$

0: Nat [T-ZERO]

$$\frac{t_1: \text{Nat}}{\text{succ } t_1: \text{Nat}} \quad [\text{T-SUCC}]$$

$$\frac{t_1: \text{Nat}}{\text{pred } t_1: \text{Nat}} \quad [\text{T-PRED}]$$

$$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}} \quad [\text{T-ISZERO}]$$

- We write t: T to mean "t has obviously type T", i.e., it evaluates to a value of the appropriate form

Typing Rules

Syntax	*Typing Rules*	
T ::=	true: Bool	[T-TRUE]
Bool		
Nat	false: Bool	[T-FALSE]
	$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T}$	[T-IF]
	0: Nat	[T-ZERO]
	$\frac{t_1: \text{Nat}}{\text{succ } t_1: \text{Nat}}$	[T-SUCC]
	$\frac{t_1: \text{Nat}}{\text{pred } t_1: \text{Nat}}$	[T-PRED]
	$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}}$	[T-ISZERO]

- We write t: T to mean "t has obviously type T", i.e., it evaluates to a value of the appropriate form.
- By obviously, we mean that we can see this fact statically, without having to evaluate the program.

Typing Rules

Syntax	*Typing Rules*	
T ::=	true: Bool	[T-TRUE]
Bool		
Nat	false: Bool	[T-FALSE]
	$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T}$	[T-IF]
	0: Nat	[T-ZERO]
	$\frac{t_1: \text{Nat}}{\text{succ } t_1: \text{Nat}}$	[T-SUCC]
	$\frac{t_1: \text{Nat}}{\text{pred } t_1: \text{Nat}}$	[T-PRED]
	$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}}$	[T-ISZERO]

- We write t: T to mean "t has obviously type T", i.e., it evaluates to a value of the appropriate form.
- By obviously, we mean that we can see this fact statically, without having to evaluate the program.
- [DEFINITION] A term t is *typable* if there exists a type T such that t: T

Example of Typing Rules

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad [\text{T-IF}]$$

Example of Typing Rules

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad [\text{T-IF}]$$

if true then 0 else false ✘

Lemma: Inversion of the 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.

1) Could you prove this lemma?
2) What does this lemma have to do with type inference?

Typing Derivations

What is the type of "**if iszero 0 then 0 else pred 0**"? How can you know this type from the rules on the right?

Syntax	*Typing Rules*	
T ::=	true: Bool	[T-TRUE]
Bool		
Nat	false: Bool	[T-FALSE]
	$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T}$	[T-IF]
	0: Nat	[T-ZERO]
	$\frac{t_1: \text{Nat}}{\text{succ } t_1: \text{Nat}}$	[T-SUCC]
	$\frac{t_1: \text{Nat}}{\text{pred } t_1: \text{Nat}}$	[T-PRED]
	$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}}$	[T-ISZERO]

Typing Derivations

$$0 : \mathsf{Nat} \quad [\text{T-ZERO}]$$

$$\frac{t_1 : \mathsf{Nat}}{\text{iszero } t_1 : \mathsf{Bool}} \; [\text{T-ISZERO}]$$

$$\frac{t_1 : \mathsf{Nat}}{\text{pred } t_1 : \mathsf{Nat}} \; [\text{T-PRED}]$$

$$\frac{t_1 : \mathsf{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \; [\text{T-IF}]$$

$$\dfrac{\dfrac{\dfrac{}{0 : \mathsf{Nat}}\,[\text{T-ZERO}]}{\text{iszero } 0 : \mathsf{Bool}}\,[\text{T-ISZERO}] \qquad \dfrac{}{0 : \mathsf{Nat}}\,[\text{T-ZERO}] \qquad \dfrac{\dfrac{}{0 : \mathsf{Nat}}\,[\text{T-ZERO}]}{\text{pred } 0 : \mathsf{Nat}}\,[\text{T-PRED}]}{\text{if iszero } 0 \text{ then } 0 \text{ else pred } 0 : \mathsf{Nat}}\,[\text{T-IF}]$$

Fernando Magno Quintão Pereira

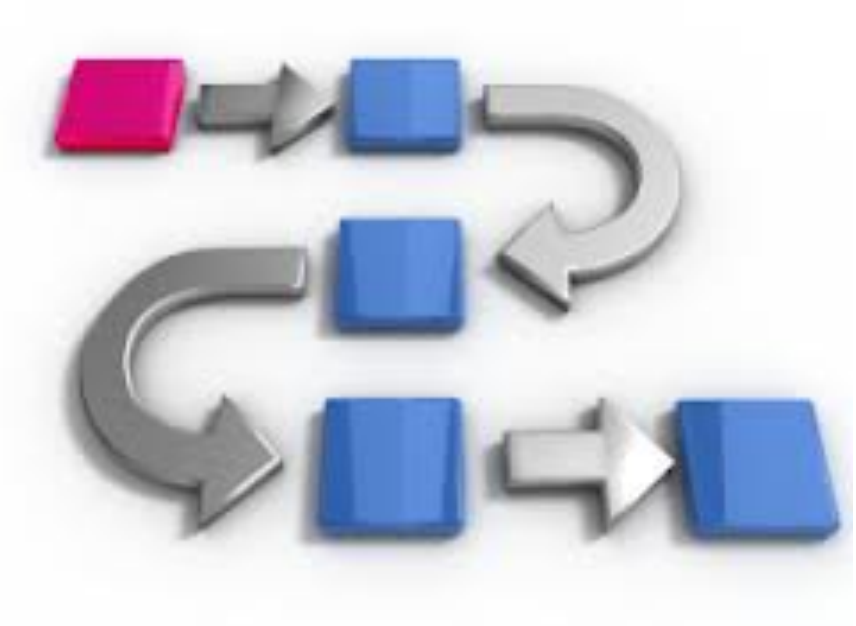
fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Progress



fernando@dcc.ufmg.br

DCC 888

Safety = Progress + Preservation

- Safety, or soundness, means that well typed terms will not go wrong.
- In other words, these terms will not reach a stuck state during evaluation.
- A stuck state is not a final value, but the evaluation rules do not tell us how to proceed with their evaluation, e.g., "pred true", or "if 0 then false else true".
- We prove that well-typed terms do not go wrong in two steps:
  - Progress: a well-typed term is either a value, or it can take a step according to the evaluation rules.
  - Preservation: if a well-typed term takes a step of evaluation, then the resulting term is also well typed.

Lemma: CANONICAL FORMS

- If v is a value of type Bool, then v is either true or false.
- If v is a value of type Nat, then v is a numeric value according to the grammar on the right.

```
v ::=
      true
    | false
    |  nv

nv ::=
      0
    | succ nv
```

This lemma sounds quite obvious, but how can we prove it? Hint: we can use the **Inversion Lemma**.

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.

Lemma: CANONICAL FORMS

- If v is a value of type Bool, then v is either true or false.
- If v is a value of type Nat, then v is a numeric value according to the grammar on the right.

Proof: this lemma comes easily from the Inversion Lemma. The only values, according to the grammar, are in the set {true, false, 0, succ nv}, where nv is a numeric value.

From cases (1) and (2) of the Inversion Lemma, we know that the values true and false have the type Bool.

From cases (4) and (5) we know that the type of numeric values is Nat.

Syntax

```
v ::=
        true
      | false
      |  nv

nv ::=
        0
      | succ nv
```

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with t → t'

Syntax

T ::=
  Bool
  Nat

Typing Rules

true: Bool [T-TRUE]

false: Bool [T-FALSE]

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T}$$ [T-IF]

0: Nat [T-ZERO]

$$\frac{t_1: \text{Nat}}{\text{succ } t_1: \text{Nat}}$$ [T-SUCC]

$$\frac{t_1: \text{Nat}}{\text{pred } t_1: \text{Nat}}$$ [T-PRED]

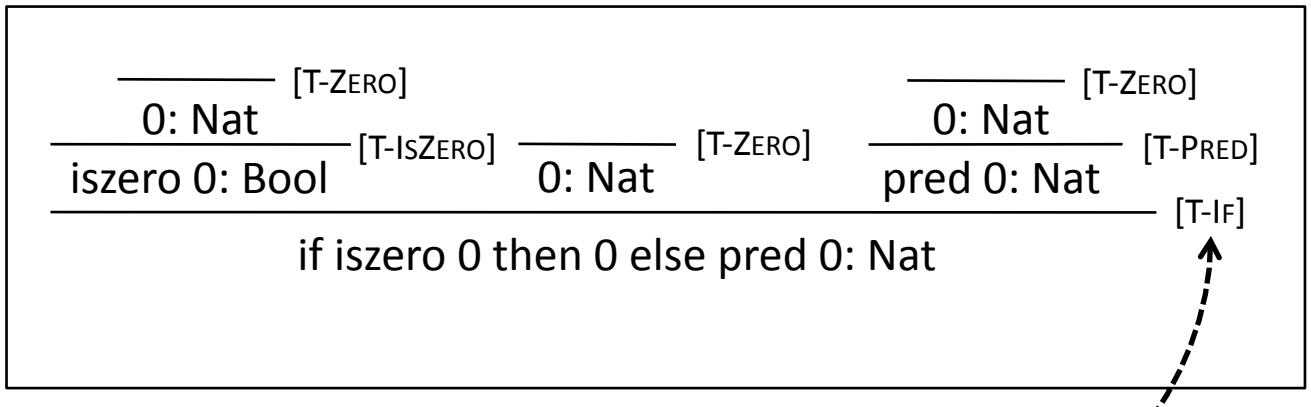
$$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}}$$ [T-ISZERO]

How can we prove this theorem?

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with t → t'

Proof: the proof goes by induction on a derivation of t:T. We must look into the last rule that has been used to show that t has type T.



How many cases do we have to consider? In other words, how many rules could have been the last rules to derive t:T?

Last rule!

CAse Analysis

Syntax	*Typing Rules*	
T ::=	true: Bool	[T-TRUE]
Bool		
Nat	false: Bool	[T-FALSE]
	$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T}$	[T-IF]
	0: Nat	[T-ZERO]
	$\frac{t_1: \text{Nat}}{\text{succ } t_1: \text{Nat}}$	[T-SUCC]
	$\frac{t_1: \text{Nat}}{\text{pred } t_1: \text{Nat}}$	[T-PRED]
	$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}}$	[T-ISZERO]

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with t → t'

Proof: the proof goes by induction on a derivation of t:T. We must look into the last rule that has been used to show that t has type T.

Case last rule = [T-TRUE], [T-FALSE] and [T-ZERO] are immediate, because they can only be applied onto values, so we have that t is a value.

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with $t \rightarrow t'$

Proof: the proof goes by induction on a derivation of t:T. We must look into the last rule that has been used to show that t has type T.

Case last rule = [T-TRUE], [T-FALSE] and [T-ZERO] are immediate, because they can only be applied onto values, so we have that t is a value.

Case last rule = [T-IF], then we know that t = *if* t_1 *then* t_2 *else* t_3, and t_1: Bool, and t_2: T, and t_3: T. By induction, either t_1 is a value, or there exists some t_1' such that $t_1 \rightarrow t_1'$. If t_1 is a value, then the Canonical Forms Lemma tells us that this value must be either true or false. In the former case, we can take a step through [E-IFTRUE]. In the latter, we take a step through [E-IFFALSE]. Otherwise, if $t_1 \rightarrow t_1'$, then we take a step via Rule [E-IF].

$$\frac{t_1\text{:Bool} \quad t_2\text{:T} \quad t_3\text{:T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3\text{: T}} \text{ [T-IF]}$$

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with t → t'

Proof: the proof goes by induction on a derivation of t:T. We must look into the last rule that has been used to show that t has type T.

Case last rule = [T-SUCC]:

1) What do you know for sure if last rule = [T-SUCC]?

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with $t \rightarrow t'$

Proof: the proof goes by induction on a derivation of t:T. We must look into the last rule that has been used to show that t has type T.

Case last rule = [T-SUCC]: we know that t = succ t_1, and that t_1: Nat

Could you do the case for [T-PRED]?

$$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \quad [\text{T-SUCC}]$$

$$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'} \quad [\text{E-SUCC}]$$

We must look now into t_1. If t_1 is a value, then we are done, because succ t_1 is also a numeric value (how do we know this?).
On the other hand, if t_1 is not a value, then, by induction, we know that it can take a step. In other words, there exists t_1', such that $t_1 \rightarrow t_1'$.
In this case, we can evaluate t through [E-Succ].

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with t → t'

Proof: the proof goes by induction on a derivation of t:T. We must look into the last rule that has been used to show that t has type T.

What about [T-ISZERO]?

Case last rule = [T-PRED]: we know that t = pred t_1, and that t_1: Nat

$$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \quad [\text{T-PRED}]$$

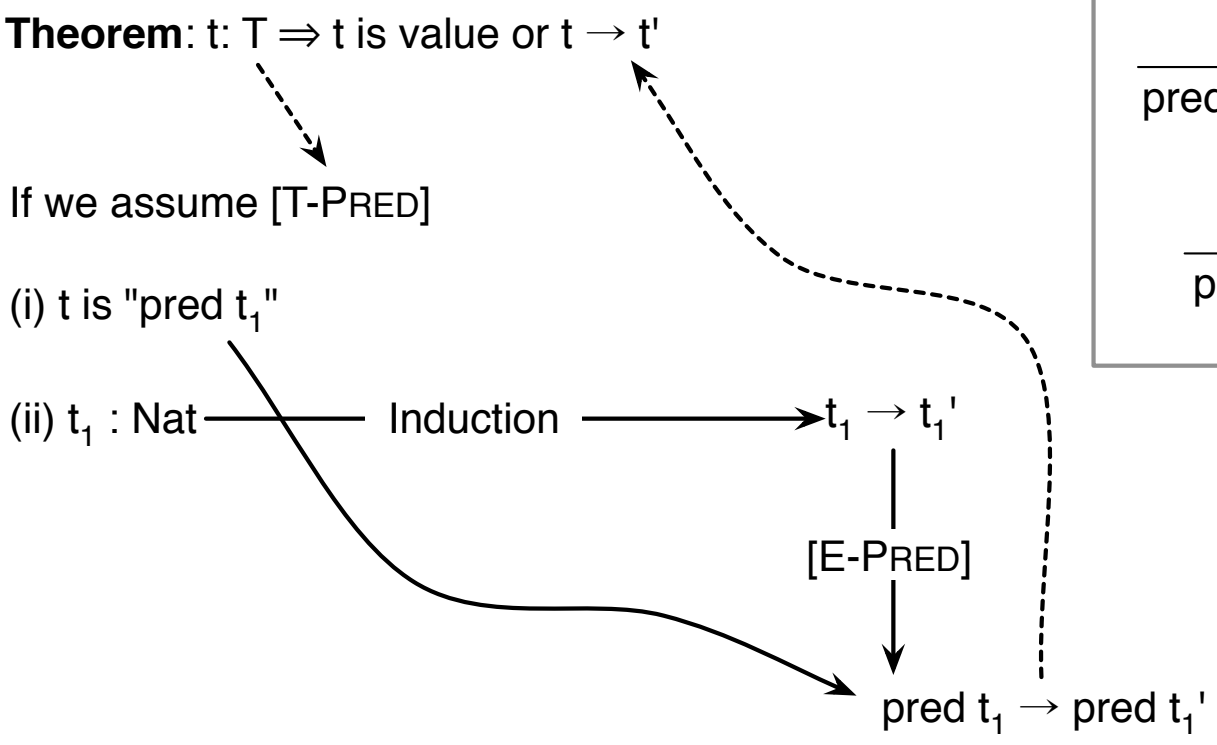
$$\text{pred } 0 \rightarrow 0 \quad [\text{E-PREDZERO}]$$

$$\text{pred}(\text{succ } nv_1) \rightarrow nv_1 \quad [\text{E-PREDSUCC}]$$

$$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'} \quad [\text{E-PRED}]$$

By induction either t_1 is a value, or t_1 admits an evaluation step.
If t_1 is a value, by the Canonical Forms Lemma, it must be a numeric value. By the syntax of our language, there exist only two numeric values: 0 or succ nv. In the former case, we can take a step via [E-PREDZERO]. In the latter, we can take a step through [E-PREDSUCC].
Otherwise, t_1 admits an evaluation step, and we can evaluate t by Rule [E-PRED].

PROGRESS: The Proof in Pictures



$$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'} \quad \text{[E-PRED]}$$

$$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \quad \text{[T-PRED]}$$

What is the metric that has decreased so that we could apply induction?

Theorem: PROGRESS

- Suppose t is a well-typed term (t:T for some type T). Then either t is a value or else there is some t' with t → t'

Proof: the proof goes by induction on a derivation of t:T. We must look into the last rule that has been used to show that t has type T.

Case last rule = [T-ISZERO]: we know that t = iszero t_1, and that t_1: Nat

$$\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}} \quad \text{[T-ISZERO]}$$

This case is analogous to [T-PRED]. By induction either t_1 is a value, or t_1 admits an evaluation step.

$$\text{iszero } 0 \rightarrow \text{true} \quad \text{[E-ISZEROZERO]}$$

$$\text{iszero (succ nv1)} \rightarrow \text{false} \quad \text{[E-ISZEROSUCC]}$$

$$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'} \quad \text{[E-ISZERO]}$$

If t_1 is a value, by the Canonical Forms Lemma, it must be a numeric value. By the syntax of our language, there exist only two numeric values: 0 or succ nv. In the former case, we can take a step via [E-ISZEROZERO]. In the latter, we can take a step through [E-ISZEROSUCC]. Otherwise, t_1 admits an evaluation step, and we can evaluate t by Rule [E-ISZERO]. □

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

PRESERVATION

fernando@dcc.ufmg.br

DCC 888

Theorem: PRESERVATION

- If t:T and t → t', then t':T

- There are two ways to prove preservation:
  - We can do induction on the derivation rules of the type system, e.g., the rules that we use to conclude that t:T
  - We can do induction on the derivation rules of the semantics, e.g., the rules that we use to show t →t'

1) If we do induction on the typing rules, how many cases would we need to consider?

2) In the proof of Progress we applied induction on the typing rules. Why couldn't we do it by induction on the evaluation rules?

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Proof: we will do an induction on a derivation of t:T. Like in the proof of progress, we must look into the last rule that has been used to show that t has type T.

Case last rule = [T-TRUE]

true: Bool [T-TRUE]

If the last rule used to show t:T was [T-True], what do we know about t and T?

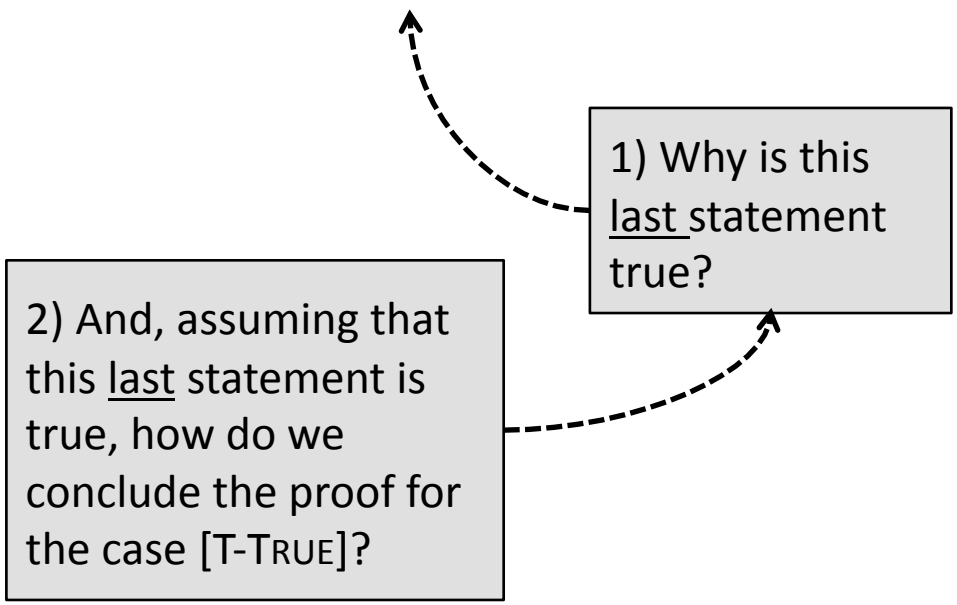
Theorem: PRESERVATION

- If t:T and t → t', then t':T

Proof: we will do an induction on a derivation of t:T. Like in the proof of progress, we must look into the last rule that has been used to show that t has type T.

Case last rule = [T-TRUE] We know that t = true, and T = Bool
However, in this case, there exists no t' such that t → t'.

true: Bool [T-TRUE]



Theorem: PRESERVATION

- If t:T and t → t', then t':T

Proof: we will do an induction on a derivation of t:T. Like in the proof of progress, we must look into the last rule that has been used to show that t has type T.

Case last rule = [T-TRUE] We know that t = true, and T = Bool
However, in this case, there exists no t' such that t → t'.
Given that there exists no t → t', the requirements of the theorem are vacuously true. This is just logic: if Mount Everest is made of melted cheese with a caramel crust, then Brazil is the greatest country in the world…

true: Bool [T-TRUE]

Notice that the cases for [T-FALSE] and [T-ZERO] are very similar.

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Proof: we will do an induction on a derivation of t:T; thus, we must look into the last rule that has been used to show that t has type T.

Case last rule = [T-IF], then we know that t = *if* t_1 *then* t_2 *else* t_3, and t_1: Bool, and t_2:T and t_3:T. We also know that t → t', as this is part of the premise of the theorem. We must show that t':T for each possible way in which t → t'.

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3: T}\ [\text{T-IF}]$$

In how many ways can t → t'?

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Proof: we will do an induction on a derivation of t:T; thus, we must look into the last rule that has been used to show that t has type T.

Case last rule = [T-IF], then we know that t = if t_1 then t_2 else t_3, and t_1: Bool, and t_2:T and t_3:T. We also know that t → t', as this is part of the premise of the theorem. We must show that t':T for each possible way in which t → t'.

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad [\text{T-IF}]$$

What about the case when t_1 = true?

$$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2 \quad [\text{E-IFTRUE}]$$

$$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3 \quad [\text{E-IFFALSE}]$$

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad [\text{E-IF}]$$

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Case last rule = [T-IF], then we know that t = if t_1 then t_2 else t_3, and t_1: Bool, and t_2:T and t_3:T. We also know that t → t', as this is part of the premise of the theorem. We must show that t':T for each possible way in which t → t'.

If we evaluate t via Rule [E-IfTrue], then we know that t_1 = true, and t' = t_2.
But we already know that t_2:T; therefore, t':T

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad [\text{T-IF}]$$

if true then t_2 else t_3 → t_2 [E-IFTRUE]

if false then t_2 else t_3 → t_3 [E-IFFALSE]

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad [\text{E-IF}]$$

1) Why do we know that t_2:T?

2) What if t → t' through [E-IFFALSE]

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Case last rule = [T-IF], then we know that t = if t_1 then t_2 else t_3, and t_1: Bool, and t_2:T and t_3:T. We also know that t → t', as this is part of the premise of the theorem. We must show that t':T for each possible way in which t → t'.

If we evaluate t via Rule [E-IFFALSE], then we know that t_1 = false, and t' = t_3.
But we already know that t_3:T, as this is a premise of the theorem; therefore, t':T

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad [\text{T-IF}]$$

if true then t_2 else t_3 → t_2 [E-IFTRUE]

if false then t_2 else t_3 → t_3 [E-IFFALSE]

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad [\text{E-IF}]$$

What if t → t' through [E-IF]

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Case last rule = [T-IF], then we know that t = if t_1 then t_2 else t_3, and t_1: Bool, and t_2:T and t_3:T. We also know that t → t', as this is part of the premise of the theorem. We must show that t':T for each possible way in which t → t'.

If we evaluate t via Rule [E-IF], then we know that there exists a t_1', such that $t_1 \rightarrow t_1'$. We also know that t_1: Bool. Thus, we can apply induction on these two facts, e.g., t_1: Bool and $t_1 \rightarrow t_1'$.

$$\frac{t_1\text{:Bool} \quad t_2\text{:T} \quad t_3\text{:T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3\text{: T}} \quad [\text{T-IF}]$$

$$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2 \quad [\text{E-IFTRUE}]$$

$$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3 \quad [\text{E-IFFALSE}]$$

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad [\text{E-IF}]$$

What do we conclude from **this** induction?

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Case last rule = [T-IF], then we know that t = if t_1 then t_2 else t_3, and t_1: Bool, and t_2:T and t_3:T. We also know that t → t', as this is part of the premise of the theorem. We must show that t':T for each possible way in which t → t'.

$$\frac{t_1\text{:Bool} \quad t_2\text{:T} \quad t_3\text{:T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3\text{: T}} \quad [\text{T-IF}]$$

If we evaluate t via Rule [E-IF], then we know that there exists a t_1', such that $t_1 \rightarrow t_1$'. We also know that t_1: Bool. Thus, we can apply induction on these two facts, e.g., t_1: Bool and $t_1 \rightarrow t_1$'.

$$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2 \quad [\text{E-IFTRUE}]$$

$$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3 \quad [\text{E-IFFALSE}]$$

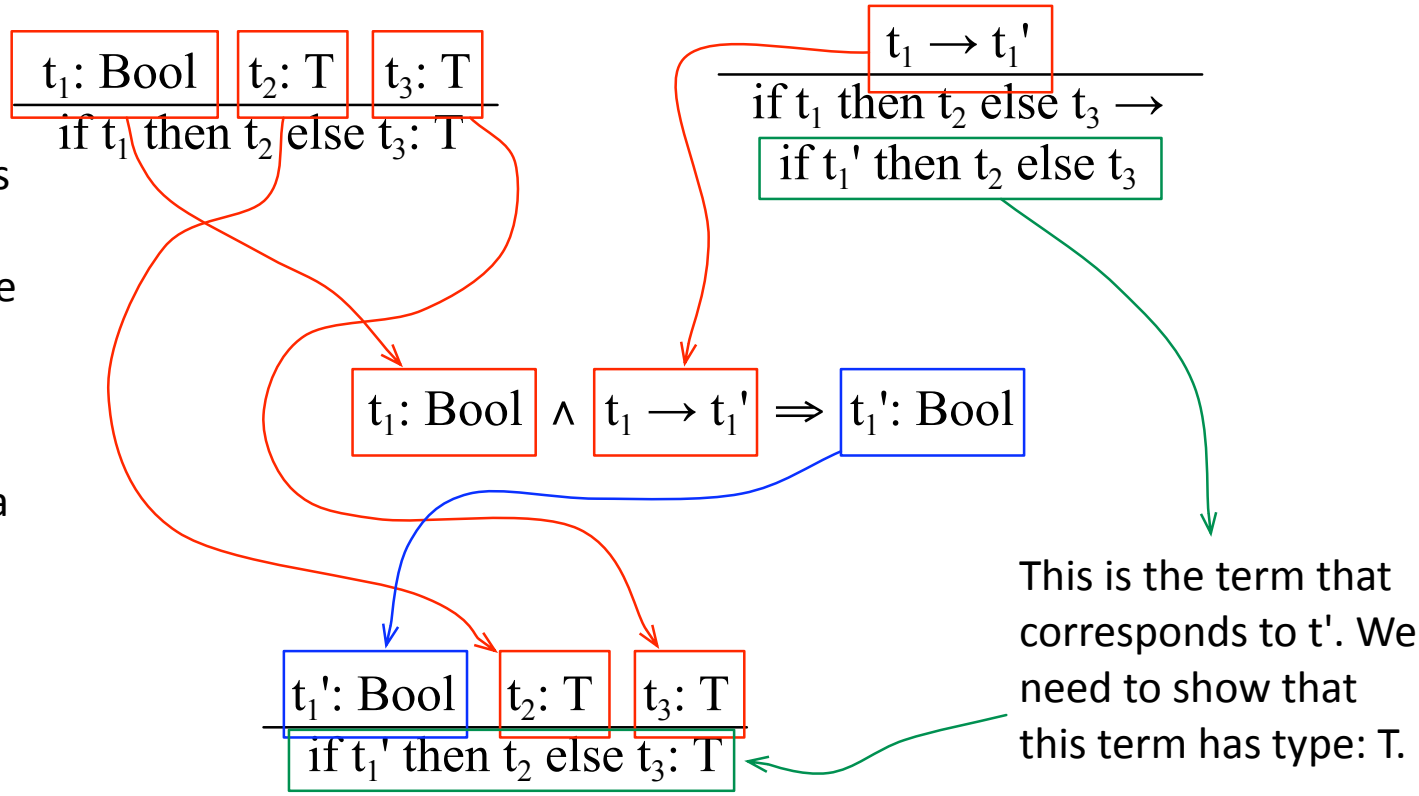
$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad [\text{E-IF}]$$

From this induction we conclude that t_1': Bool. Therefore, we can use these three facts, e.g,. t_1': Bool, t_2:T and t_3:T, to conclude that *if* t_1 *then* t_2 *else* t_3: T, via Rule [T-IF].

Theorem: PRESERVATION

- This kind of proof is mostly mechanical. For instance:

Theorem $t{:}T \;\wedge\; t \rightarrow t' \;\Rightarrow\; t'{:}\,T$



These facts in red boxes are the truths that we know, from the premisses of the theorem.

The fact in the blue box was inferred via induction.

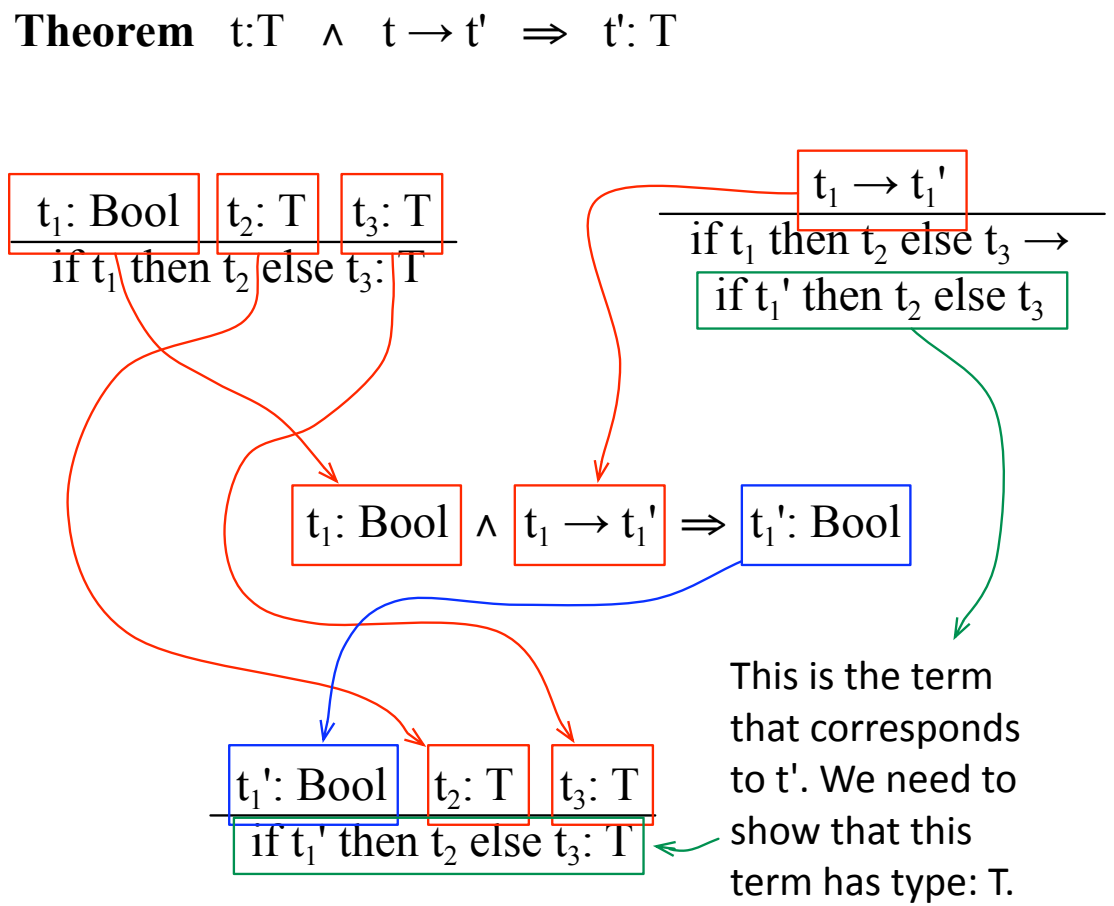
This is the term that corresponds to t'. We need to show that this term has type: T.

Theorem: PRESERVATION

Could you draw a schema similar to that one on the right, if t = succ t_1?

$$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \quad [\text{T-SUCC}]$$

$$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'} \quad [\text{E-SUCC}]$$



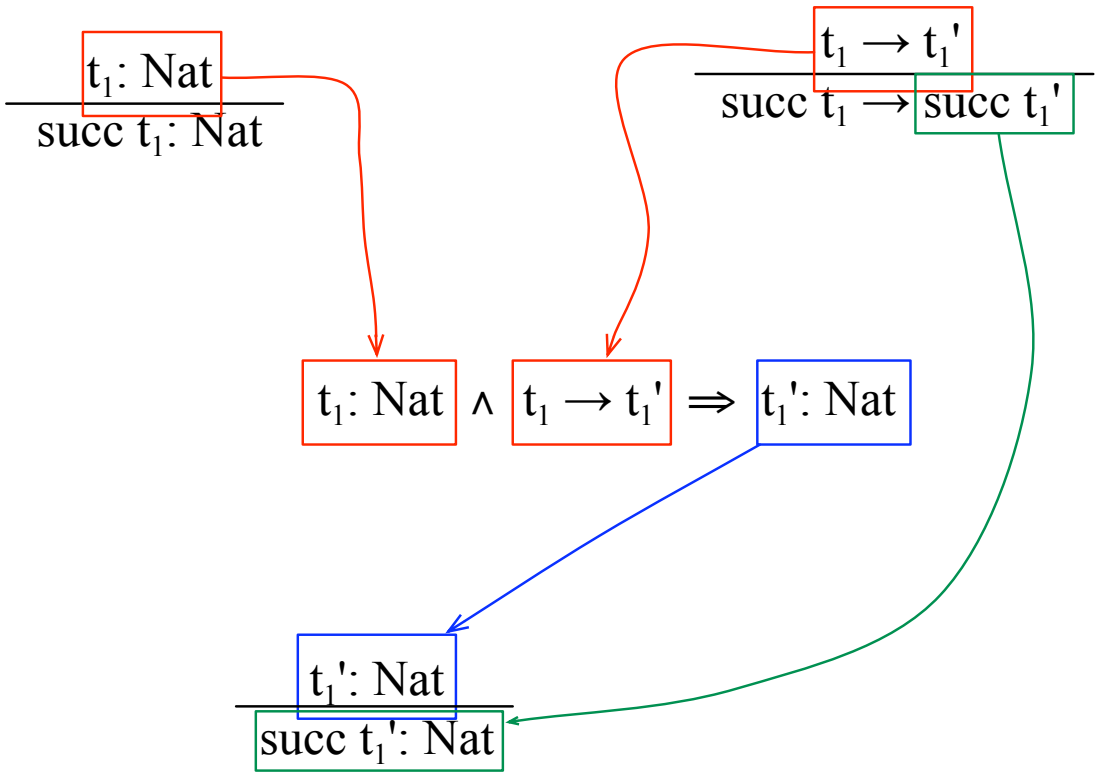
Theorem: PRESERVATION

What about [T-PRED]? In how many ways can we take a step if we type a term with this rule?

$$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \quad \text{[T-SUCC]}$$

$$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'} \quad \text{[E-SUCC]}$$

Theorem $t{:}T \;\wedge\; t \rightarrow t' \;\Rightarrow\; t'{:}\,T$



Theorem: PRESERVATION

- If t:T and $t \rightarrow t'$, then t':T

What about [T-PRED]? In how many ways can we take a step if we type a term with this rule?

$$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \quad [\text{T-PRED}]$$

$$\text{pred } 0 \rightarrow 0 \quad [\text{E-PREDZERO}]$$

$$\text{pred}(\text{succ } nv_1) \rightarrow nv_1 \quad [\text{E-PREDSUCC}]$$

$$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'} \quad [\text{E-PRED}]$$

Case last rule = [T-PRED], then we know that t = pred t_1. We must look into the different ways that $t \rightarrow t'$. In this case, there exists three different evaluation rules.

Case [E-PREDZERO], then we know that t' = 0. From Rule [T-Zero], we know that t':Nat

Case [E-PREDSUCC], then we know that t_1 = succ nv_1. From t_1: Nat, we know that succ nv_1: Nat. From Rule [T-SUCC], we know that nv_1: Nat, by the inversion of the typing relation. Given that t' = nv_1, we have that t':Nat

Could you make a drawing of this last case, e.g., [E-PREDSUCC]?

Theorem: PRESERVATION

- If t:T and t → t', then t':T

Now, the last rule that is missing is [E-PRED]. We will need induction. Can you do it?

$$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \quad [\text{T-PRED}]$$

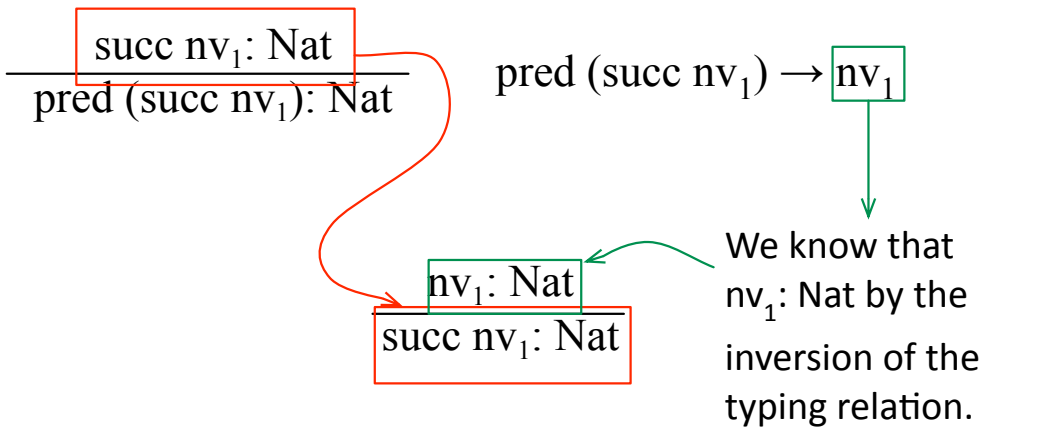
$$\text{pred } 0 \rightarrow 0 \quad [\text{E-PREDZERO}]$$

$$\text{pred}(\text{succ } nv_1) \rightarrow nv_1 \quad [\text{E-PREDSUCC}]$$

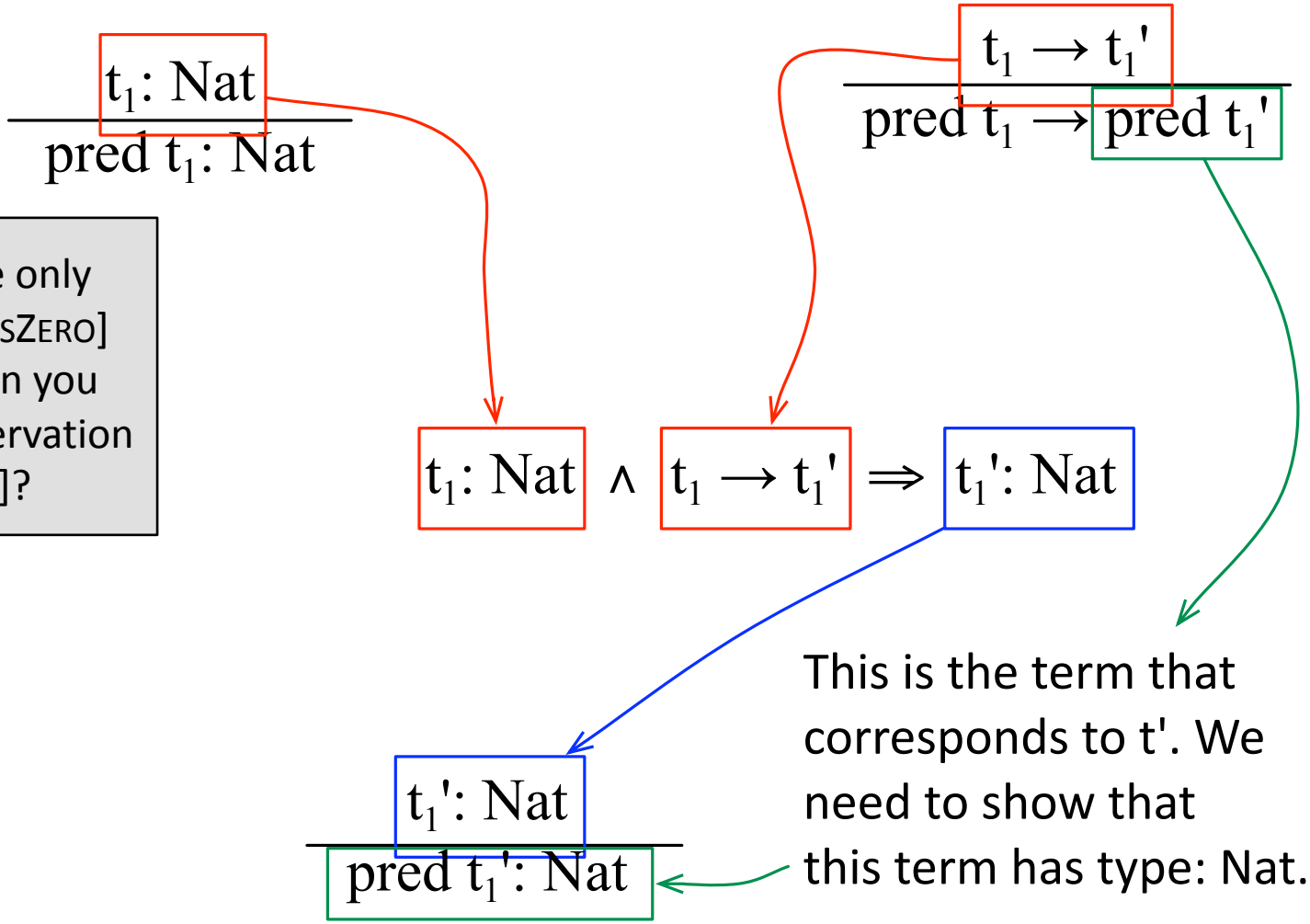
$$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'} \quad [\text{E-PRED}]$$

Case [E-PREDSUCC], then we know that t_1 = succ nv_1. From t_1: Nat, we know that succ nv_1: Nat. From Rule [T-SUCC], we know that nv_1: Nat by the inversion of the typing relation. Given that t' = nv_1, we have that t':Nat

Theorem $t:T \ \wedge \ t \rightarrow t' \ \Rightarrow \ t': T$



Theorem $t{:}T \;\wedge\; t \rightarrow t' \;\Rightarrow\; t'{:}\,T$



Theorem: PRESERVATION

- If t:T and t → t', then t':T

Case last rule = [T-ISZERO], then we use the same strategy as we did when the last rule was [T-PRED]. We know that t = iszero t_1. We must look into the three different ways in which this term, iszero t_1, can take a step.

If we take a step via Rule [E-ISZEROZERO], then we know that t' is true, and by Rule [T-TRUE], we have that t': Bool

$$\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}} \quad [\text{T-ISZERO}]$$

$$\text{iszero } 0 \rightarrow \text{true} \quad [\text{E-ISZEROZERO}]$$

$$\text{iszero (succ nv1)} \rightarrow \text{false} \quad [\text{E-ISZEROSUCC}]$$

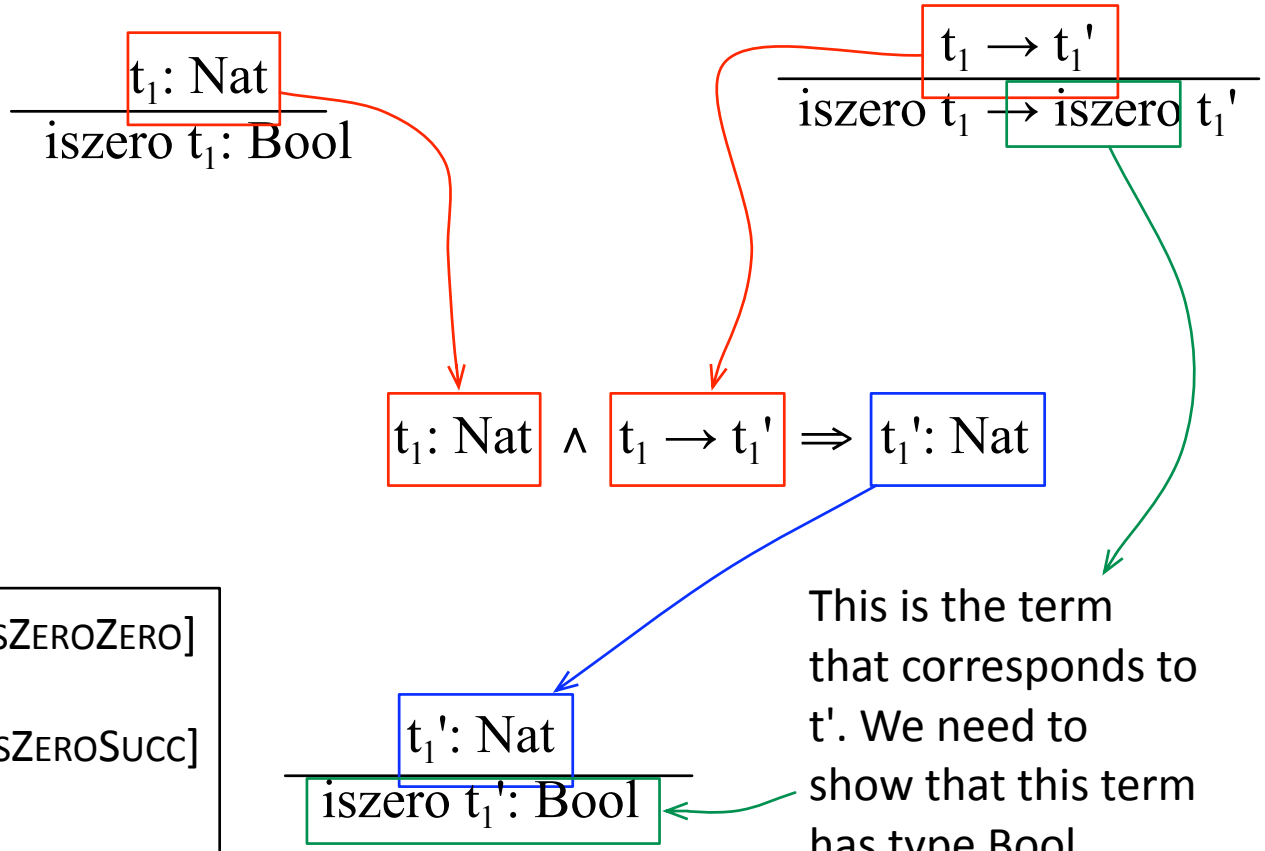
$$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'} \quad [\text{E-ISZERO}]$$

If we take a step via Rule [E-ISZEROSUCC], then we know that t' is false, and by Rule [T-FALSE], we have that t': Bool

By now we must be good with the drawings. Could you sketch the proof when the step is taken via [E-IsZero]?

Theorem: PRESERVATION

Theorem $t:T \;\wedge\; t \longrightarrow t' \;\Rightarrow\; t': T$



$$\frac{t_1: \text{Nat}}{\text{iszero } t_1: \text{Bool}} \quad \text{[T-IsZero]}$$

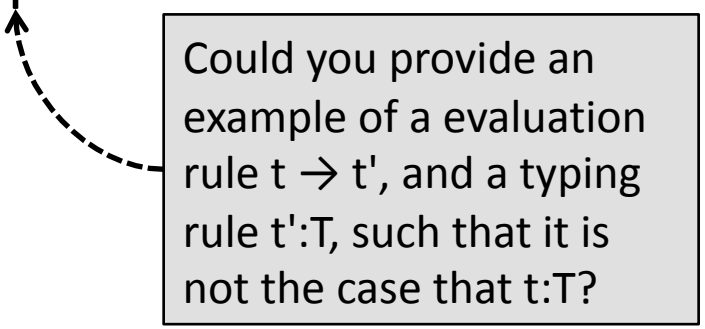
$$\text{iszero } 0 \rightarrow \text{true} \quad \text{[E-IsZeroZero]}$$

$$\text{iszero (succ nv1)} \rightarrow \text{false} \quad \text{[E-IsZeroSucc]}$$

$$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'} \quad \text{[E-IsZero]}$$

Preservation = Subject Reduction

- Preservation is also called subject reduction.
- A sentence such as "t has type T" has t as the subject.
- When we evaluate t, taking a step, we are usually reducing the size of t.
  - So that is the name, *subject reduction*.
- Notice that the reverse theorem, *Subject Expansion*, is not true to our simple language, as we can easily see via a counter-example.



Preservation = Subject Reduction

- Preservation is also called subject reduction.
- A sentence such as "t has type T" has t as the subject.
- When we evaluate t, taking a step, we are usually reducing the size of t.
  - So that is the name, *subject reduction*.
- Notice that the reverse theorem, *Subject Expansion*, is not true to our simple language, as we can easily see via a counter-example.

Here is a counter-example. The term (if false then true else 0) is ill-typed, but this term evaluates to 0, which is well-typed. In other words, "*if false then true else 0* → *0* and *0: Nat*", but *if false then true else 0* has no type.

A Bit of History

- Type systems were invented in logic/mathematics, as a way to solve some paradoxes, such as the *Russell's Paradox*.
- In Computer Science, the earliest type systems would separate integers from floating-point numbers.
- ML was one of the first languages to include the notion of type inference.
- Andrew Wright and Matthias Felleisen formulated the definition of type safety that we have used, based on progress and preservation.

- Cardelli, L. "Type Systems", Handbook of Computer Science and Engineering. (1996)
- Backus, J. "The History of Fortran I, II, and III", HOPL (1981) pp 25-25
- Whitehead, A. and Russel, B. "Principia Mathematica" (1910)
- Milner, R. and Tofte, M. and Harper, R. "The Definition of Standard ML" (1990)
- Wright, Andrew K.; Matthias Felleisen. "A Syntactic Approach to Type Soundness". Information and Computation. 115 (1): 38–94. (1994)