PROVING THEOREMS WITH TWELF

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides has been taken from the Twelf tutorial, which is available on-line

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

Twelf

- We shall see how we can use twelf, a software developed by Pfenning *et al*. to certify that the proof of a theorem is correct.
- Twelf is built around a beautiful theory of *dependent types*.
  - We won't talk much about it, but the interested reader can find a discussion in human-readable format in Pierce's book♤.
- We can use Twelf to prove facts about languages with binding, e.g., that map names to terms.
  - But we will not talk about bindings here. Instead, we will try to keep this discussion as simple as possible.

Let us start our discussion with a simple question: how to define the natural numbers, e.g., zero, one, two, and such and such?

♤: *Types and Programming Languages*, by Benjamin C. Pierce

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

Natural Numbers

- Zero is a natural number!
- And the other numbers are successors of zero!
  - This gives us a inductive definition of numbers:

The natural numbers

Base case: zero is a number

Inductive case: $\dfrac{n \text{ is a number}}{succ\ n \text{ is a number}}$

1) What does this guy above have to do with these numbers?
2) How do you read this kind of inference rule?
3) How can you define the even numbers?

The Even Numbers

The natural numbers

Base case: zero is a number

Inductive case: $\dfrac{n \text{ is a number}}{\mathit{succ}\ n \text{ is a number}}$

Let's extend it a bit further: how can you define addition?

The even numbers

Base case: zero is even

Inductive case: $\dfrac{n \text{ is even}}{\mathit{succ}\ (\mathit{succ}\ n) \text{ is even}}$

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

Addition

The natural numbers

Base case: zero is a number

Inductive case: $\dfrac{n \text{ is a number}}{succ\ n \text{ is a number}}$

The even numbers

Base case: zero is even

Inductive case: $\dfrac{n \text{ is even}}{succ\ (succ\ n) \text{ is even}}$

Addition of natural numbers

Base case: *plus(zero, n, n)*

Inductive case: $\dfrac{plus(n_1, n_2, n_3)}{plus(succ(n_1), n_2, succ(n_3))}$

These definitions are precise enough so that we can formalize them directly in any declarative programming language. We shall use Twelf. Later on, we will show how we can prove facts about these relations.

Does **this** definition encompass every natural number? In other words, is there any number that cannot be added up with this definition?

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

A First Look into Twelf

`number.elf`

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).
```

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

A First Look into Twelf

`number.elf`

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).
```

A First Look into Twelf

`number.elf`

```
nat : type.
z : nat.
s : nat -> nat.

even   : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).
```

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

A First Look into Twelf

`number.elf`

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).
```

```
~$
~$ cd twelf/examples/fernando/dcc888/
~/twelf/examples/fernando/dcc888$ vim number.elf
~/twelf/examples/fernando/dcc888$ cd ../../../
~/twelf$ bin/twelf-server
. . .
make examples/fernando/dcc888/sources.cfg
. . .
%% OK %%
```

`sources.cfg`

```
number.elf
```

This is our first Twelf program. You can just type it in any text editor of your choice, and then use the twelf interpreter to test it. This program contains the definitions of relations on natural numbers that we have seen before. We can load it with the **make** command. In this case, we have a file sources.cfg, in the same folder as `number.elf`, which tells `make` which files constitute this script.

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

Representing the Real World

The natural numbers

Base case: zero is a number

Inductive case: $\dfrac{n \text{ is a number}}{\mathit{succ}\ n \text{ is a number}}$

The even natural numbers

Base case: zero is even

Inductive case: $\dfrac{n \text{ is even}}{\mathit{succ}\ (\mathit{succ}\ n) \text{ is even}}$

Addition of natural numbers

Base case: *plus(zero, n, n)*

Inductive case: $\dfrac{plus(n_1, n_2, n_3)}{plus(succ(n_1), n_2, succ(n_3))}$

`number.elf`

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3
  -> plus (s N1) N2 (s N3).
```

Can you see how the ELF representation matches our initial definitions?

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

Back on Track: Logic Programming with Twelf

number.elf

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3
  -> plus (s N1) N2 (s N3).
```

```
~/twelf$ bin/twelf-server
. . .
make examples/fernando/dcc888/sources.cfg
. . .
%% OK %%
top
?- even z.
Solving...
Empty Substitution.
More? y
No more solutions
?- plus (s z) (s (s z)) N.
Solving...
N = s (s (s z)).
More? y
No more solutions
?- plus N (s (s z)) (s (s (s (s z)))).
Solving...
N = s (s z).
More? y
No more solutions
```

The command **top** opens up the interactive query mode, which we can quit with CTRL + C

Can you explain the result that we get for each query?

Proving Properties about Numbers

The coolest thing about Twelf is that it helps us to verify proofs of theorems. Let's us show how this verification works with the following theorem: if N_1 and N_2 are even numbers, and $N_1 + N_2 = N_3$, then N_3 is an even number.

The even numbers

Base case: zero is even

Inductive case: $\dfrac{n \text{ is even}}{succ\ (succ\ n) \text{ is even}}$

Addition of natural numbers

Base case: *plus(zero, n, n)*

Inductive case: $\dfrac{plus(n_1, n_2, n_3)}{plus(succ(n_1), n_2, succ(n_3))}$

1) How can we prove this theorem?
2) Can you find any measure onto which you can apply induction?

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

The Sum of Even Numbers is Even

We must show that for any
N1, N2 and N3 such that:

- even(N1)
- even(N2)
- plus(N1, N2, N3)

We have that

- even(N3)

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

The Sum of Even Numbers is Even

We must show that for any N1, N2 and N3 such that:

- even(N1)
- even(N2)
- plus(N1, N2, N3)

We have that

- even(N3)

There are only two ways to show that **this** rule is true:

even(z) in this case N1 = z

$$\frac{\text{even(N1')}}{\text{even(succ(succ(N1')))}}$$

In this case, we know that there exists a N1', such that N1 = succ(succ(N1'))

Ok: that is already a lot of help. Can you find a way to show that even(N3) holds?

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

The Sum of Even Numbers is Even

if *even(z)* is true, then N1 = z. But *plus(z, N2, N2)* is true, by the definition of addition.

In this case, N2 = N3. We already know that *even(N2)* is true. Thus, we conclude our proof by showing that *even(N2)* is a proof of *even(N3)*.

The Sum of Even Numbers is Even

if *even(z)* is true, then N1 = z. But *plus(z, N2, N2)* is true, by the definition of addition.

In this case, N2 = N3. We already know that *even(N2)* is true. Thus, we conclude our proof by showing that *even(N2)* is a proof of *even(N3)*.

if N1 = *succ(succ(N1'))*, and *even(N1')* is true, we know that *plus(succ(succ(N1')), N2, succ(succ(N3')))*, by two applications of the inductive case of addition. Thus, we know that *plus(succ(N1'), N2, succ(N3'))*, and finally *plus(N1', N2, N3')*. We apply induction on *plus(N1', N2, N3')* and *even(N1')* to conclude that *even(N3')*.

So, we know that *even(N3')* is true. But, by applying the inductive step of even, we know also that *even(succ(succ(N3')))* is true. In other words, we know that *even(N3)* is true!

Mechanizing the Proof

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

Mechanizing the Proof

`number.thm`

```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.
%mode sum-evens +D1 +D2 +D3 -D4.

sez : sum-evens even-z EvenN2 plus-z EvenN2.

ses : sum-evens (even-s EvenN1') EvenN2 (plus-s (plus-s Plus)) (even-s EvenN3')
    <- sum-evens EvenN1' EvenN2 Plus EvenN3'.

%worlds () (sum-evens _ _ _ _).
%total D (sum-evens D _ _ _).
```

`sources.cfg`

```
number.elf
number.thm
```

This program is a proof that our theorem is true. Ok: I know that it is rather scary, but we can go over all of it, in hopes to understand what is going on here.

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

Mechanizing the Proof

```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.
%mode sum-evens +D1 +D2 +D3 -D4.

sez :

ses :

%wo
%tot
```

The **first** declaration states our theorem: for every N1, N2 and N3, such that even(N1), even(N2) and plus(N1, N2, N3), it is the case that even(N3) is true.

The second line is the *mode* of the theorem. The mode of the theorem tells us which facts are universally quantifiable, and which facts are existentially quantifiable. In other words, we are saying that, (i) for every way to show even(N1), (ii) every way to show even(N2), and (iii) every way to show plus(N1, N2, N3), there exists (iv) a way to show even(N3). We gave names to each one of these facts: i is called **D1**, ii is called **D2**, iii is called **D3**, and iv is called **D4**.

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

Mechanizing the Proof

```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.
%mode sum-evens +D1 +EvenN2 +D3 -EvenN3.

sez : sum-evens even-z EvenN2 plus-z EvenN2.
```

This is our first proof, and we are calling it **sez**. We are saying that in this case, fact D1 is **even-z**. Fact D3 must be **plus-z**, as there is no other pattern that handles N1 = z. We must show that EvenN3 is true. But, in this case, we know that N2 = N3, due to plus-z. Therefore, we have that EvenN3 = EvenN2, and we are done: we can feed **EvenN2**, a proof that N2 is even, in the place where a proof that N3 is even is expected. The Twelf type system will know that this proof is correct.

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3
  -> plus (s N1) N2 (s N3).
```

Proofs like Games

```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.
%mode sum-evens +D1 +EvenN2 +D3 -EvenN3.

sez : sum-evens even-z EvenN2 plus-z EvenN2.
```

```
nat : type.
z : nat.
s : nat -> nat.

even     : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3
  -> plus (s N1) N2 (s N3).
```

So, proving anything in Twelf is like playing a game. We are given a set of **inputs**. For these inputs, we must find a suitable **output**. Inputs and outputs are the facts that we know as true in our description of terms and types. These **facts** must conform to the **types** in the statement of the theorem.

In this example, the inputs were the rules even-z, EvenN2 and plus-z. EvenN2 is undefined: it can be any rule that matches the theorem statement, e.g., "even N2". We win this game by showing EvenN2 itself as a proof for this case.

Mechanizing the Proof

```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.
%mode sum-evens +D1 +D2 +D3 -D4.

sez : sum-evens even-z EvenN2 plus-z EvenN2.

ses : sum-evens (even-s EvenN1') EvenN2 (plus-s (plus-s Plus)) (even-s EvenN3')
      <- sum-evens EvenN1' EvenN2 Plus EvenN3'.

%worlds () (sum-evens _ _ _ _).
%total D (sum-evens D _ _ _).
```

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3
  -> plus (s N1) N2 (s N3).
```

The inductive step of the proof is substantially more complicated. Can you try to make head-or-tail of it, before we get into the details?

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

Mechanizing the Proof

sum-evens : **even N1** -> **even N2** -> **plus N1 N2 N3** -> even N3 -> type.

ses : sum-evens

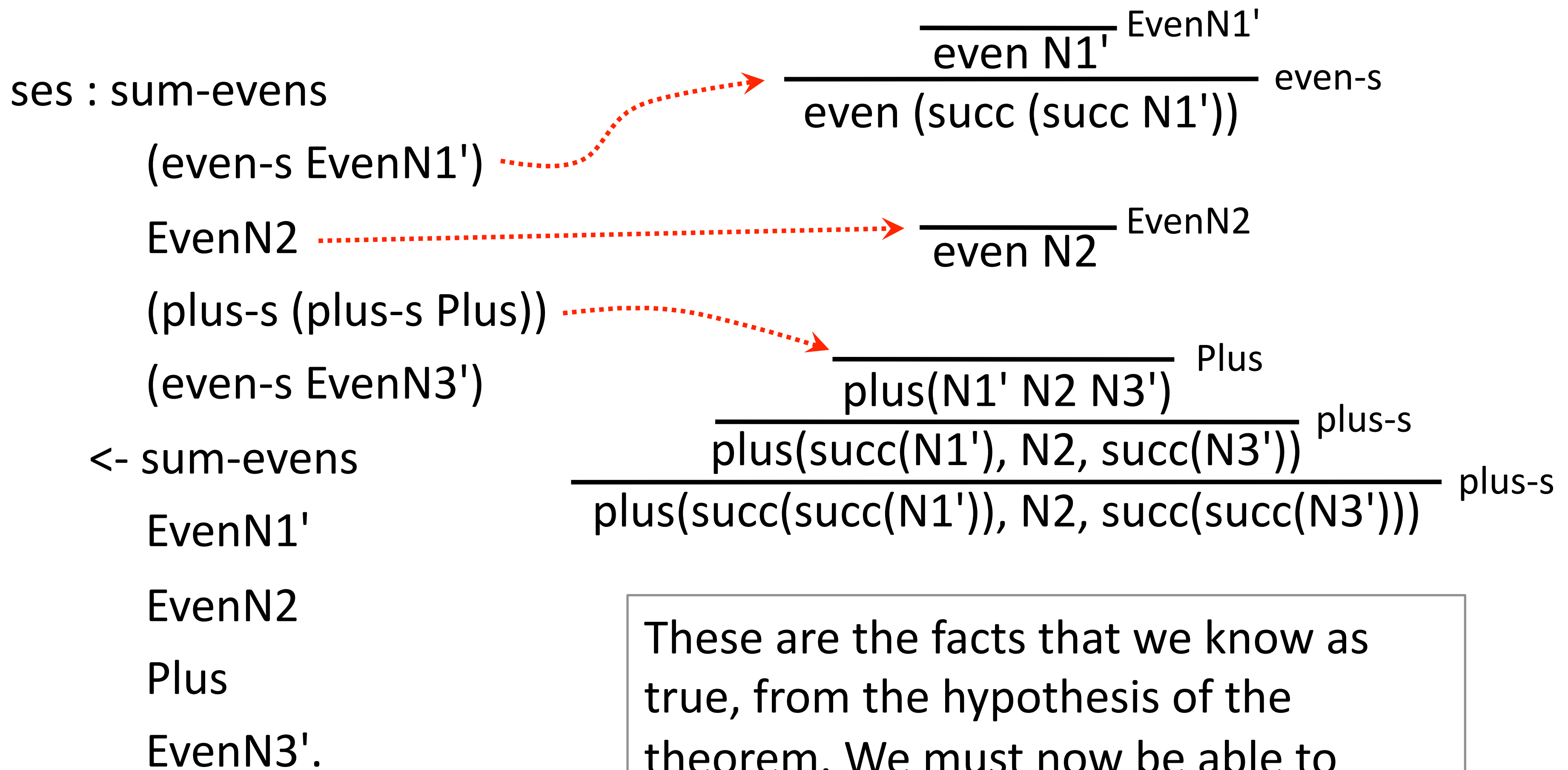
(even-s EvenN1')
EvenN2
(plus-s (plus-s Plus)) *Things that we know as true*

(even-s EvenN3') ← *The fact that we want to prove*

<- sum-evens EvenN1' EvenN2 Plus EvenN3'.
And this is what we assume as true by induction

PPGCC

Mechanizing the Proof



Mechanizing the Proof

sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.

```
ses : sum-evens
        (even-s EvenN1')
        EvenN2
        (plus-s (plus-s Plus))
        (even-s EvenN3')
    <- sum-evens
        EvenN1'
        EvenN2
        Plus
        EvenN3'.
```

$$\dfrac{\dfrac{}{\text{even N1'}}\text{ EvenN1'}}{\text{even (succ (succ N1'))}}\text{ even-s}$$

$$\dfrac{}{\text{even N2}}\text{ EvenN2}$$

$$\dfrac{\dfrac{\dfrac{}{\text{plus(N1' N2 N3')}}\text{ Plus}}{\text{plus(succ(N1'), N2, succ(N3'))}}\text{ plus-s}}{\text{plus(succ(succ(N1')), N2, succ(succ(N3')))}}\text{ plus-s}$$

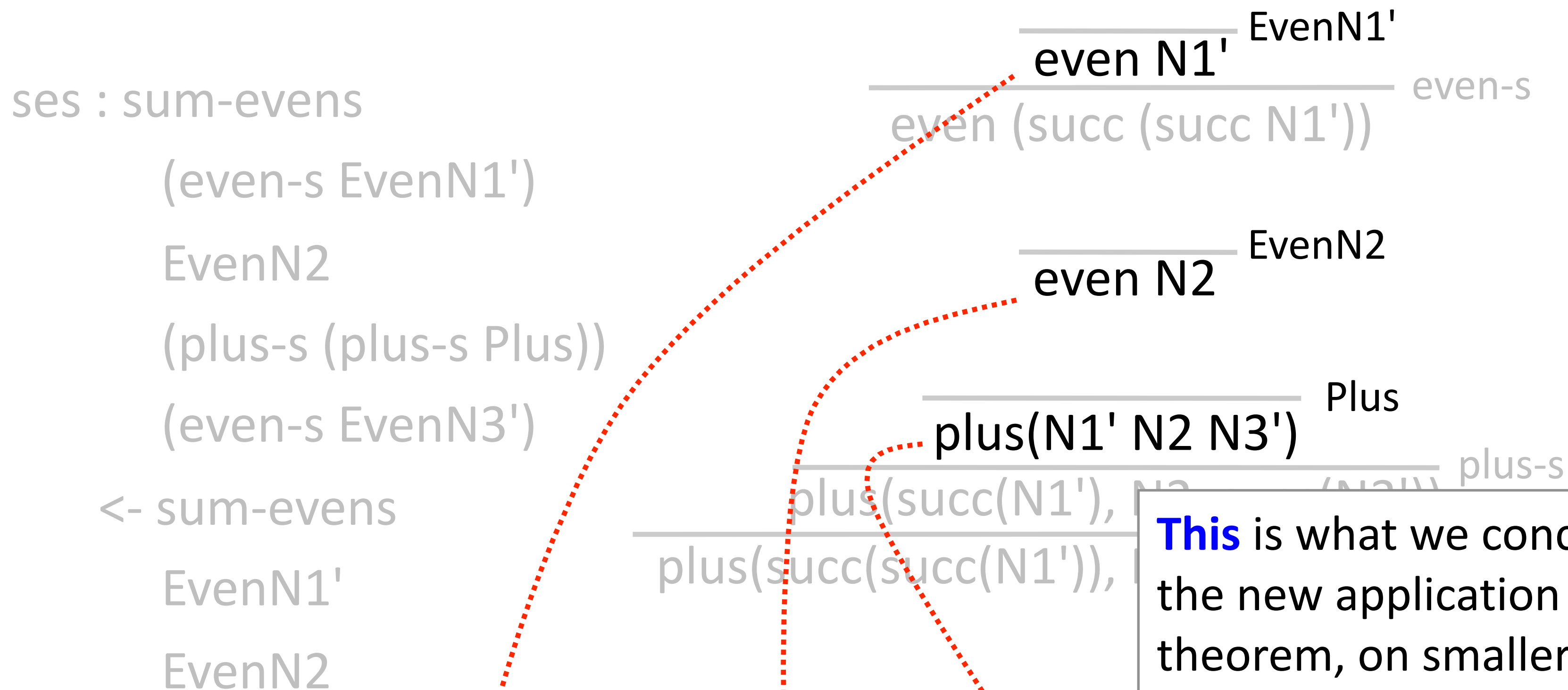
1) Can we apply the theorem on these three facts: EvenN1', EvenN2 and Plus?

2) What can we conclude from this new application of the theorem?

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

Mechanizing the Proof

sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.



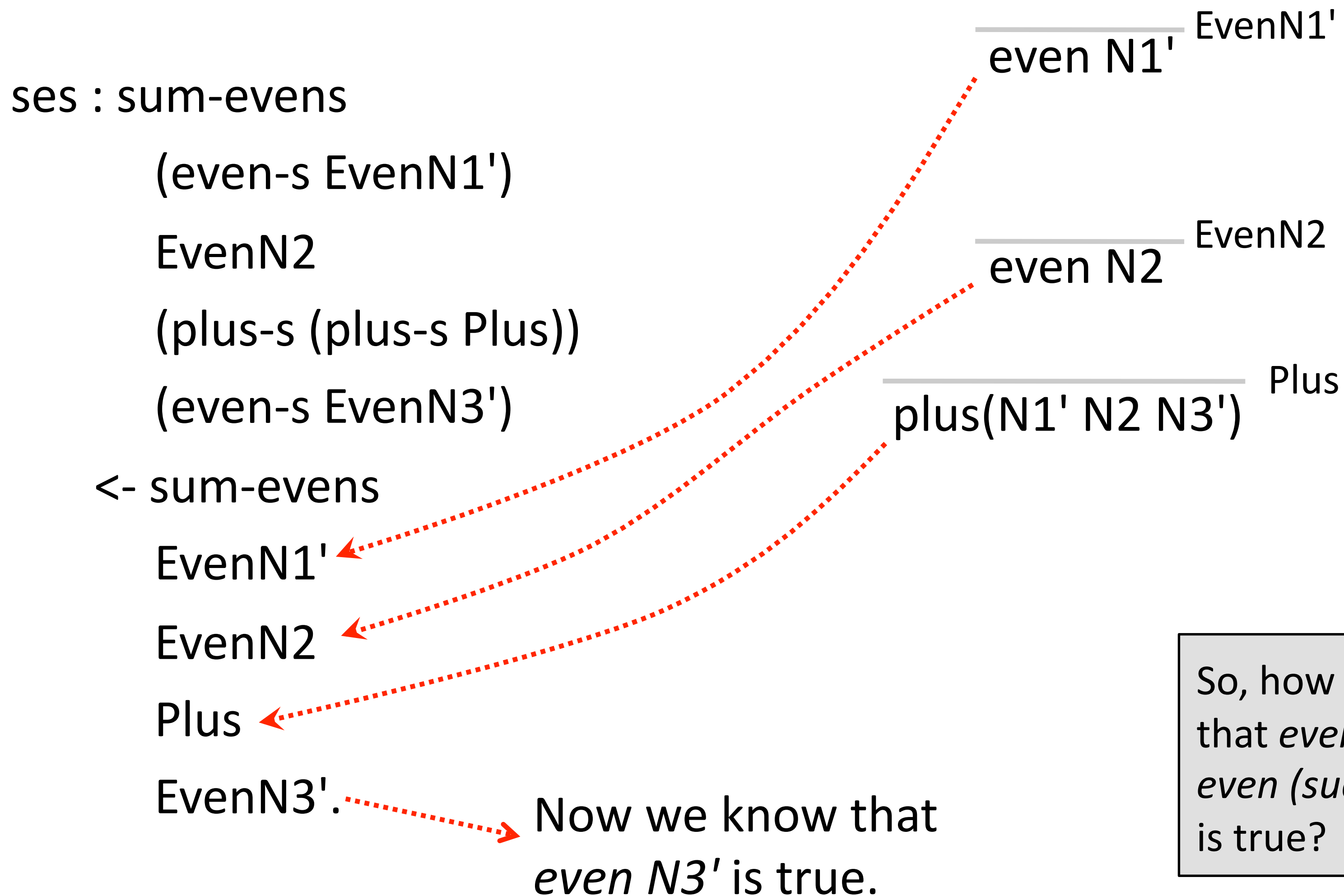
This is what we conclude by the new application of the theorem, on smaller terms.

sum-evens : EvenN1' -> EvenN2 -> Plus -> **EvenN3'** -> type.

sum-evens (even N1') (even N2) (plus N1' N2 N3') (**even N3'**)

Mechanizing the Proof

sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.

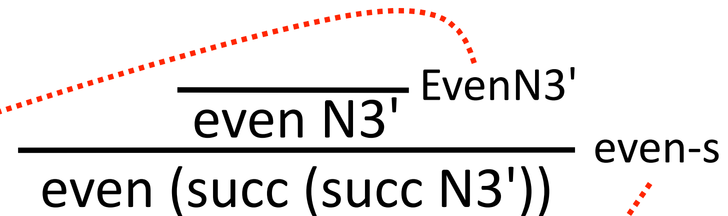


So, how can we infer that *even N3*, which is *even (succ (succ N3'))*, is true?

Mechanizing the Proof

```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.

ses : sum-evens
        (even-s EvenN1')
        EvenN2
        (plus-s (plus-s Plus))
        (even-s EvenN3')
    <- sum-evens
        EvenN1'
        EvenN2
        Plus
        EvenN3'.
```



The Importance of Induction

```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.
```

```
ses : sum-evens
        (even-s EvenN1')
        EvenN2
        (plus-s (plus-s Plus))
        (even-s EvenN3')
    <- sum-evens
        EvenN1'
        EvenN2
        Plus
        EvenN3'.
```

To prove that *even N3'* is true, we had to apply the theorem again, this time on a smaller term. This is induction on the number of derivations. In this case, it is the first part of the theorem, e.g., even N1, that is decreasing. It is very important that the Twelf type system can know that the size of this first clause is being reduced, or it would not be able to assume that the conclusion of the theorem, on shorter derivations, is true.

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

The Importance of Induction

number.thm

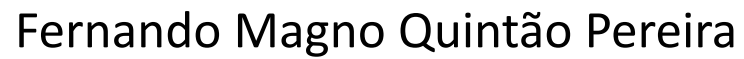
```
sum-evens : even N1 -> even N2 -> plus N1 N2 N3 -> even N3 -> type.
%mode sum-evens +D1 +D2 +D3 -D4.

sez : sum-evens even-z EvenN2 plus-z EvenN2.

ses : sum-evens (even-s EvenN1') EvenN2 (plus-s (plus-s Plus)) (even-s EvenN3')
    <- sum-evens EvenN1' EvenN2 Plus EvenN3'.

%worlds () (sum-evens _ _ _ _).
%total D (sum-evens D _ _ _).
```

If we go back to the original twelf file, we will find a clause there called %total. This clause explains to Twelf which part of the theorem we are using as the metric for the induction. In this case, we are doing induction on the **first rule**, which we have called D.



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

MORE EXAMPLES OF TWELF PROOFS

fernando@dcc.ufmg.br

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

Less than or Equal and Greater than

- Let us see other cool relations in Twelf. This time we will be talking about how to order natural numbers.
  - We can establish an ordering among them by defining a less-than-or-equal (leq) relationship.
  - Let us define a greater-than (grt) relationship as well.

Given that natural numbers are defined as:

```
nat: type.
z: nat.
s: nat -> nat.
```

How would you define the grt and leq relations?

Less than or Equal and Greater than

- Let us see another cool relations in Twelf. This time we will be talking about how to order natural numbers.
  - We can establish an ordering among them by defining a less-than-or-equal (leq) relationship.
  - Let us define a greater-than (grt) relationship as well.

```
nat : type.
z : nat.
s : nat -> nat.

grt   : nat -> nat -> type.
grt_z : grt (s N) z.
grt_s : grt (s N) (s M) <- grt N M.

leq   : nat -> nat -> type.
leq_z : leq z N.
leq_s : leq (s N) (s M)   <- leq N M.
```

This notation is another way to write implications. We can either write A -> B, or B <- A.

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

Less than or Equal and Greater than

- Let us see another cool relations in Twelf. This time we will be talking about how to order natural numbers.
  - We can establish an ordering among them by defining a less-than-or-equal (leq) relationship.
  - Let us define a greater-than (grt) relationship as well.

```
nat : type.
z : nat.
s : nat -> nat.

grt   : nat -> nat -> type.
grt_z : grt (s N) z.
grt_s : grt (s N) (s M) <- grt N M.

leq   : nat -> nat -> type.
leq_z : leq z N.
leq_s : leq (s N) (s M)   <- leq N M.
```

This notation is another way to write implications. We can either write A -> B, or B <- A.

Now, prove the following theorem: if N1 > N2, then N2 ≤ N1. Start easy: what if you concluded that N1 > N2 via grt_z?

Less than or Equal and Greater than

Challenge: show that $N1 > N2 \Rightarrow N2 \leq N1$.

There are only two ways $N1 > N2$: either we have concluded this via grt_z, or we have concluded it through grt_s.

```
nat : type.
z : nat.
s : nat -> nat.

grt   : nat -> nat -> type.
grt_z : grt (s N) z.
grt_s : grt (s N) (s M) <- grt N M.

leq   : nat -> nat -> type.
leq_z : leq z N.
leq_s : leq (s N) (s M)   <- leq N M.
```

If we know that $N1 > N2$ by grt_z, then we know that:

- $\exists$ nat N, such that N1 = s N
- N2 = z

Which rule can we use to show that $N2 \leq N1$? There are only two options: leq_z or leq_s

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

Coverage

`number.thm`

```
inv_grt : grt N1 N2 -> leq N2 N1 -> type.
%mode inv_grt +D1 -D4.

inv_grt_z : inv_grt grt_z leq_z.

%worlds () (inv_grt _ _).
%total D (inv_grt D _).
```

If we know that N1 > N2 by **grt_z**, then we know that:

- ∃ nat N, such that N1 = s N
- N2 = z

But, if N2 = z, then we can conclude N2 ≤ N1 via **leq_z**.

`number.elf`

```
grt   : nat -> nat -> type.
grt_z : grt (s N) z.
grt_s : grt (s N) (s M) <- grt N M.

leq   : nat -> nat -> type.
leq_z : leq z N.
leq_s : leq (s N) (s M)   <- leq N M.
```

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

Coverage

number.thm

```
inv_grt : grt N1 N2 -> leq N2 N1 -> type.
%mode inv_grt +D1 -D4.

inv_grt_z : inv_grt grt_z leq_z.

%worlds () (inv_grt _ _).
%total D (inv_grt D _).
```

If we know that N1 > N2 by **grt_z**, then we know that:

- ∃ nat N, such that N1 = s N
- N2 = z

But, if N2 = z, then we can conclude N2 ≤ N1 via **leq_z**.

number.elf

```
grt   : nat -> nat -> type.
grt_z : grt (s N) z.
grt_s : grt (s N) (s M) <- grt N M.

leq   : nat -> nat -> type.
leq_z : leq z N.
leq_s : leq (s N) (s M)   <- leq N M.
```

But, now we get this error...

```
Coverage error --- missing cases:

{X1:nat} {X2:nat}
{X3:grt X1 X2}
{X4:leq (s X2) (s X1)}
    |- inv_grt (grt_s X3) X4.

%% ABORT %%
```

Coverage

number.thm

```
inv_grt : grt N1 N2 -> leq N2 N1 -> type.
%mode inv_grt +D1 -D4.

inv_grt_z : inv_grt grt_z leq_z.

%worlds () (inv_grt _ _).
%total D (inv_grt D _).
```

number.elf

```
grt   : nat -> nat -> type.
grt_z : grt (s N) z.
grt_s : grt (s N) (s M) <- grt N M.

leq   : nat -> nat -> type.
leq_z : leq z N.
leq_s : leq (s N) (s M)   <- leq N M.
```

We are handling the case in which N2 = z, but it is possible that N2 ≠ 0. This situation happens if we inferred grt N1 N2 via Rule grt_s. In this case, we must show a proof that N2 ≤ N1 considering that N2 = s N, for some natural N. The **%worlds** declaration forces us to take care of every possible pattern .

Remember: we must handle every possible pattern that could have been used to derive grt N1 N2.

So, how can we conclude the proof, assuming that N2 = s N, for some natural N?

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

Coverage

number.thm

```
inv_grt : grt N1 N2 -> leq N2 N1 -> type.
%mode inv_grt +D1 -D4.

inv_grt_s : inv_grt (grt_s GRT) (leq_s LTH)
    <- inv_grt GRT LTH.
```

number.elf

```
grt   : nat -> nat -> type.
grt_z : grt (s N) z.
grt_s : grt (s N) (s M) <- grt N M.

leq   : nat -> nat -> type.
leq_z : leq z N.
leq_s : leq (s N) (s M)   <- leq N M.
```

If we know that N1 > N2 via grt_s, then we know the following facts:

- N1 = s N
- N2 = s M
- grt N M

We can apply induction on grt N M. In other words, if N > M, then M ≤ N. We write this as **inv_grt (N > M) (M ≤ N)**.

Now that we know that M ≥ N, we also know that (s M) ≥ (s N), via leq_s

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

N1 > N2 ⇒ N2 ≤ N1

`number.thm`

```
inv_grt : grt N1 N2 -> leq N2 N1 -> type.
%mode inv_grt +D1 -D4.

inv_grt_z : inv_grt grt_z leq_z.

inv_grt_s : inv_grt (grt_s GRT) (leq_s LTH)
    <- inv_grt GRT LTH.

%worlds () (inv_grt _ _).
%total D (inv_grt D _).
```

```
make dcc888/sources.cfg
...
%worlds () (inv_grt _ _).
%total D (inv_grt D _).
[Closing file dcc888/number.thm]
%% OK %%
```

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

The Double of a Number

1) Do you remember the definition of naturals in twelf?
2) Can you write the predicate double in twelf?

The dobule of a number

Base case: the double of zero is zero

Inductive case:

$$\frac{\textit{n2} \text{ is the double of } \textit{n1}}{\textit{succ(succ n2) is the double of succ(n1)}}$$

Revisiting the Definition of Naturals

number.elf

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).
```

Can you augment these definitions with a predicate "double(N1, N2)" that is true when N2 = 2 * N1?

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

The predicate Double

`number.elf`

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).

double : nat -> nat -> type.
double-z : double z z.
double-s : double N1 N2 -> double (s N1) (s (s N2)).
```

The predicate Double

`number.elf`

```
nat : type.
z : nat.
s : nat -> nat.

even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

plus   : nat -> nat -> nat -> type.
plus-z : plus z N2 N2.
plus-s : plus N1 N2 N3 -> plus (s N1) N2 (s N3).

double : nat -> nat -> type.
double-z : double z z.
double-s : double N1 N2 -> double (s N1) (s (s N2)).
```

How can you write the theorem: "the double of a number is even" in twelf?

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

The Double of a Number is Even

`number.thm`

```
even-double : double N1 N2 -> even N2 -> type.
%mode even-double +D1 -D2.

...

%worlds () (even-double _ _).
%total D (even-double D _).
```

What are the two cases that must be considered?

The Double of a Number is Even

- **double-z**: even (double Zero) = even Zero = true
- **double-s**:
  - even (double (Succ n)) = even (Succ (Succ (double n)))
  - *By induction*:
    - even (double n) = true
  - *Thus*:
    - even (Succ (Succ (double n))) = true

```
even    : nat -> type.
even-z  : even z.
even-s  : even N -> even (s (s N)).

double : nat -> nat -> type.
double-z : double z z.
double-s : double N1 N2 -> double (s N1) (s (s N2)).
```

And how can we write this proof in twelf?

The Double of a Number is Even

`number.thm`

```
even-double : double N1 N2 -> even N2 -> type.
%mode even-double +D1 -D2.

edz : even-double double-z even-z.

eds : even-double (double-s Double) (even-s Even)
      <- even-double Double Even.

%worlds () (even-double _ _).
%total D (even-double D _).
```

The Double of a Number is Even

number.thm

```
even-double : double N1 N2 -> even N2 -> type.
%mode even-double +D1 -D2.

edz : even-double double-z even-z.

eds : even-double (double-s Double) (even-s Even)
      <- even-double Double Even.

%worlds () (even-double _ _).
%total D (even-double D _).
```

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

Big Step Semantics

$$\frac{t_1 \Downarrow \text{true} \quad t_2 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v} \quad \text{[B-TRUE]}$$

$$\frac{t_1 \Downarrow \text{false} \quad t_3 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v} \quad \text{[B-FALSE]}$$

$$\frac{t \Downarrow \text{true}}{\text{not } t \Downarrow \text{false}} \quad \text{[B-NOTTRUE]}$$

$$\frac{t \Downarrow \text{false}}{\text{not } t \Downarrow \text{true}} \quad \text{[B-NOTFALSE]}$$

Can you write these rules in twelf?

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

Big Step Semantics (Twelf version)

$$\frac{t_1 \Downarrow true \quad t_2 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v} \text{ [B-TRUE]}$$

$$\frac{t_1 \Downarrow false \quad t_3 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v} \text{ [B-FALSE]}$$

$$\frac{t \Downarrow true}{\text{not } t \Downarrow false} \text{ [B-NOTTRUE]}$$

$$\frac{t \Downarrow false}{\text{not } t \Downarrow true} \text{ [B-NOTFALSE]}$$

```
tm : type.
true : tm.
false : tm.
not : tm -> tm.
if : tm -> tm -> tm -> tm.
eval: tm -> tm -> type.

eval_not_t : eval (not T) false
   <- eval T true.
eval_not_f : eval (not T) true
   <- eval T false.
eval_if_t : eval (if T1 T2 _) V
   <- eval T1 true
   <- eval T2 V.
eval_if_f : eval (if T1 _ T3) V
   <- eval T1 false
   <- eval T3 V.
```

lang.elf

Proving Program Equivalence

1) What does it mean to show that two programs are equivalent?

2) Can you show that if t_1 then t_2 else t_3 is equivalent to if not t_1 then t_3 else t_2?

```
tm : type.
true : tm.
false : tm.
not : tm -> tm.
if : tm -> tm -> tm -> tm.
eval: tm -> tm -> type.

eval_not_t : eval (not T) false
   <- eval T true.
eval_not_f : eval (not T) true
   <- eval T false.
eval_if_t : eval (if T1 T2 _) V
   <- eval T1 true
   <- eval T2 V.
eval_if_f : eval (if T1 _ T3) V
   <- eval T1 false
   <- eval T3 V.
```

`lang.elf`

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

Proving Equivalence if we assume [B-FALSE]§

$$\frac{t_1 \Downarrow true \quad t_2 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v} \text{[B-TRUE]}$$

$$\frac{t_1 \Downarrow false \quad t_3 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v} \text{[B-FALSE]}$$

$$\frac{t \Downarrow true}{\text{not } t \Downarrow false} \text{[B-NOTTRUE]}$$

$$\frac{t \Downarrow false}{\text{not } t \Downarrow true} \text{[B-NOTFALSE]}$$

How can you write this proof in Twelf?

if t_1 then t_2 else t_3
≡
if not t_1 then t_3 else t_2?

If we use the rule [B-FALSE], then we know that:
(i) $t_1 \Downarrow$ false
(ii) $t_3 \Downarrow$ v
(iii) if t_1 then t_2 else $t_3 \Downarrow$ v

From (i) plus [B-NOTFALSE], we know that:
(iv) not $t_1 \Downarrow$ true

From rule [B-TRUE], plus (iv), plus (ii), we know that:
if not t_1 then t_3 else $t_2 \Downarrow$ v

§: This slide is also available in the class on "Operational Semantics"

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

The proof of equivalence in Twelf

lang.thm

```
ifCommutes : eval (if B T1 T2) X -> eval (if (not B) T2 T1) X -> type.
%mode ifCommutes +IF1 -IF2.

ifCommutes_f:
  ifCommutes (eval_if_f EvalV EvalF) (eval_if_t EvalV (eval_not_f EvalF)).

%worlds () (ifCommutes _ _).
%total E (ifCommutes E _).
```

```
eval_not_t : eval (not T) false
   <- eval T true.
eval_not_f : eval (not T) true
   <- eval T false.
eval_if_t : eval (if T1 T2 _) V
   <- eval T1 true
   <- eval T2 V.
eval_if_f : eval (if T1 _ T3) V
   <- eval T1 false
   <- eval T3 V.
```

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

The proof of equivalence in Twelf

`lang.thm`

```
ifCommutes : eval (if B T1 T2) X -> eval (if (not B) T2 T1) X -> type.
%mode ifCommutes +IF1 -IF2.

ifCommutes_f:
  ifCommutes (eval_if_f EvalV EvalF) (eval_if_t EvalV (eval_not_f EvalF)).

%worlds () (ifCommutes _ _).
%total E (ifCommutes E _).
```

What does this message represent?

```
Coverage error --- missing cases:

{X1:tm} {X2:tm} {X3:tm} {X4:tm}
{X5:eval X2 X4} {X6:eval X1 true}
   {X7:eval (if (not X1) X3 X2) X4}
   |- ifCommutes (eval_if_t X5 X6) X7.
```

Error Messages

$$\dfrac{\dfrac{}{X_1 \Downarrow true}\ X_6 \qquad \dfrac{}{X_2 \Downarrow X_4}\ X_5}{\text{if } X_1 \text{ then } X_2 \text{ else } X_3 \Downarrow X_4}\ \text{eval_if_t}$$

So, we need to derive X7, which is equal to:
eval (if (not X1) X3 X2) X4
How can we do it?

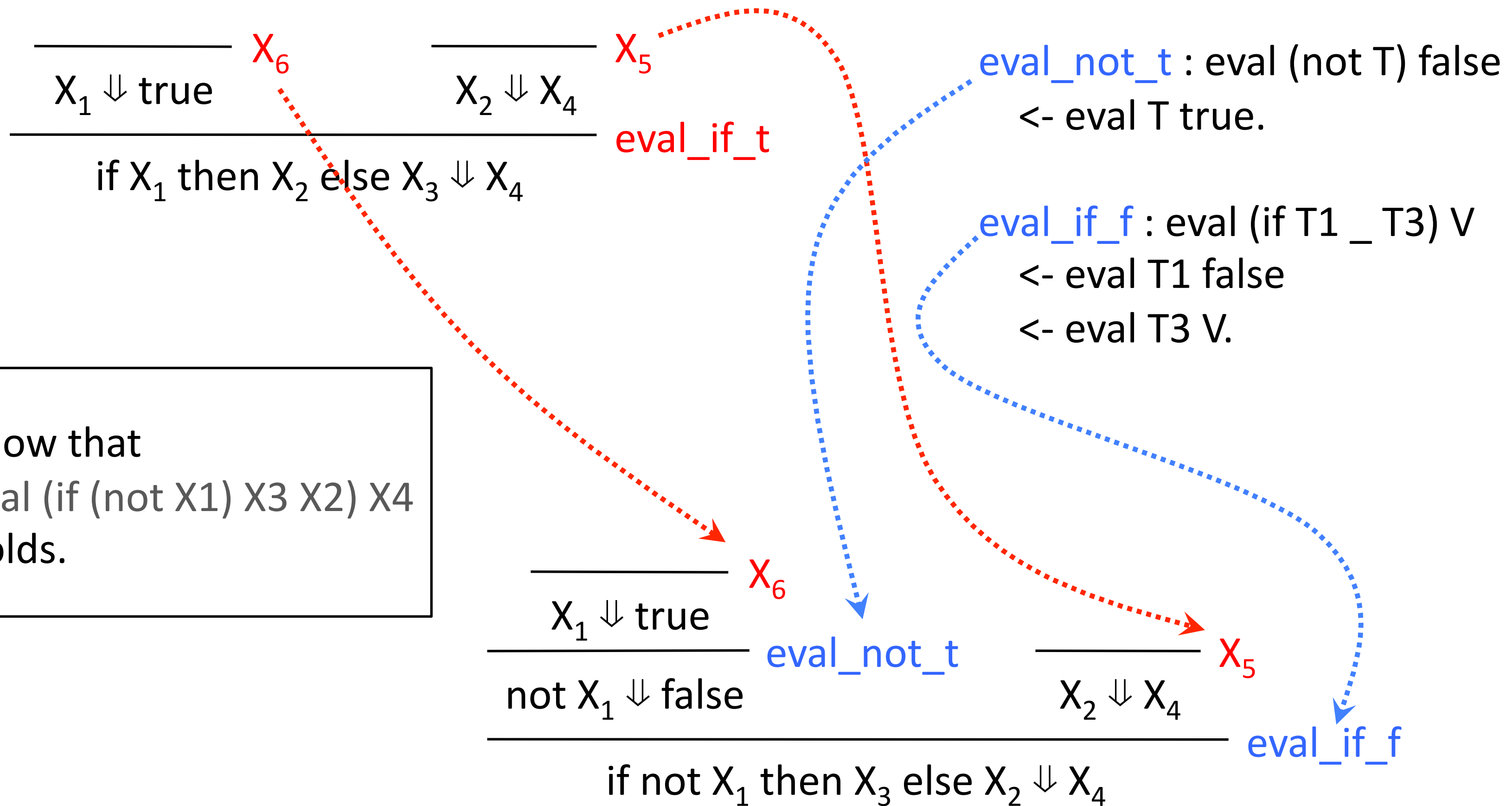
```
Coverage error --- missing cases:

{X1:tm} {X2:tm} {X3:tm} {X4:tm}
{X5:eval X2 X4} {X6:eval X1 true}
   {X7:eval (if (not X1) X3 X2) X4}
   |- ifCommutes (eval_if_t X5 X6) X7.
```

ifCommutes : eval (if B T1 T2) X -> eval (if (not B) T2 T1) X -> type.

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

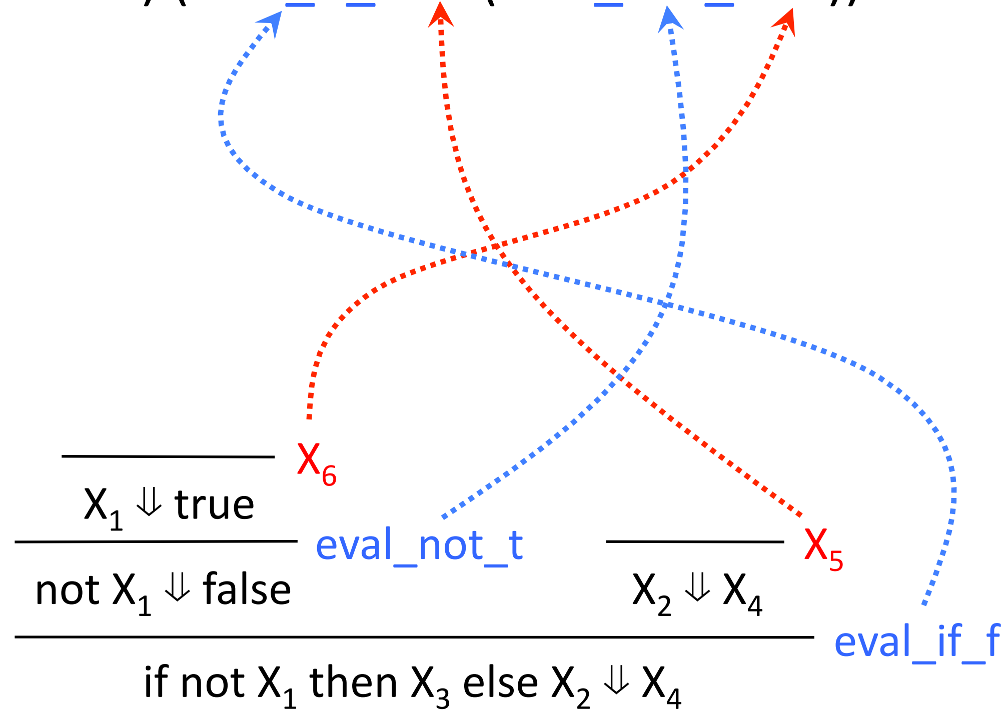
Using the Error Message to Guide the Proof



And how can you write this derivation in Twelf?

Writing the Rules in Twelf

ifCommutes_t:
ifCommutes (eval_if_t X5 X6) (eval_if_f X5 (eval_not_t X6)).



Can you show the whole theorem?

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

And we are done!

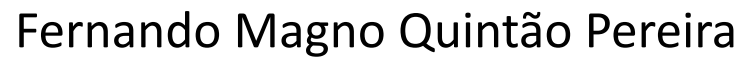
```
ifCommutes : eval (if B T1 T2) X -> eval (if (not B) T2 T1) X -> type.
%mode ifCommutes +IF1 -IF2.

ifCommutes_t:
  ifCommutes (eval_if_t EvalV EvalT) (eval_if_f EvalV (eval_not_t EvalT)).

ifCommutes_f:
  ifCommutes (eval_if_f EvalV EvalF) (eval_if_t EvalV (eval_not_f EvalF)).

%worlds () (ifCommutes _ _).
%total E (ifCommutes E _).
```

lang.thm


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

DEFINING A SIMPLE LANGUAGE IN TWELF

DCC 888

fernando@dcc.ufmg.br

The Language of Numbers and Booleans

When we discussed types, we saw a simple programming language, which has natural numbers and booleans:

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]

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

How can we formalize all of this in Twelf?

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

The Terms of the Language

Syntax:

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

Let's start with the terms. How can we describe them in Twelf?

The Terms of the Language

Syntax:

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

```
tm : type.
zero : tm.
true : tm.
false : tm.
succ : tm -> tm.
pred : tm -> tm.
iszero : tm -> tm.
if : tm -> tm -> tm -> tm.
```

sources.cfg

```
lang.elf
```

```
make examples/fernando/dcc888/
sources.cfg
tm : type.
zero : tm.
true : tm.
false : tm.
succ : tm -> tm.
pred : tm -> tm.
iszero : tm -> tm.
if : tm -> tm -> tm -> tm.

%% OK %%
```

What about the formalization of the evaluation rules?

Semantics

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

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

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

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

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

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

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

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

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

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

sources.cfg

lang.elf

```
eval: tm -> tm -> type.
eval_succ : eval (succ T) (succ T')
   <- eval T T'.
eval_pred_z : eval (pred zero) zero.
eval_pred_s : eval (pred (succ N)) N.
eval_pred : eval (pred T) (pred T')
   <- eval T T'.
eval_iszero_z : eval (iszero zero) true.
eval_iszero_s : eval (iszero (succ T)) false.
eval_iszero : eval (iszero T) (iszero T')
   <- eval T T'.
eval_ifTrue : eval (if true T2 T3) T2.
eval_ifFalse : eval (if false T2 T3) T3.
eval_if : eval (if T1 T2 T3) (if T1' T2 T3)
   <- eval T1 T1'.
```

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

Logic Programming

```
eval: tm -> tm -> type.
eval_succ : eval (succ T) (succ T')
   <- eval T T'.
eval_pred_z : eval (pred zero) zero.
eval_pred_s : eval (pred (succ N)) N.
eval_pred : eval (pred T) (pred T')
   <- eval T T'.
eval_iszero_z : eval (iszero zero) true.
eval_iszero_s : eval (iszero (succ T)) false.
eval_iszero : eval (iszero T) (iszero T')
   <- eval T T'.
eval_ifTrue : eval (if true T2 T3) T2.
eval_ifFalse : eval (if false T2 T3) T3.
eval_if : eval (if T1 T2 T3) (if T1' T2 T3)
   <- eval T1 T1'.
```

```
top
?- eval (succ zero) T.
Solving...
No more solutions
?- eval (iszero (succ zero)) T.
Solving...
T = false.
More? y
No more solutions
?- eval (iszero (pred (succ zero))) T.
Solving...
T = iszero zero.
More? y
No more solutions
```

And now there is all that discussion about types. Do you remember the type system of our toy language?

Type System

Syntax

T ::=
- Bool
- Nat

Typing Rules

$$\text{true: Bool} \quad \text{[T-True]}$$

$$\text{false: Bool} \quad \text{[T-False]}$$

$$\frac{t_1\text{:Bool} \qquad t_2\text{:T} \qquad t_3\text{:T}}{\text{if }_1\text{ then } t_2 \text{ else } t_3\text{: T}} \quad \text{[T-If]}$$

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

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

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

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

How can we formalize these rules in Twelf?

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

Type System in Twelf

```
tp : type.
nat : tp.
bool : tp.

check : tm -> tp -> type.
check_zero : check zero nat.
check_true : check true bool.
check_false : check false bool.
check_succ : check (succ T1) nat
   <- check T1 nat.
check_pred : check (pred T1) nat
   <- check T1 nat.
check_iszero : check (iszero T1) bool
   <- check T1 nat.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.
```

Syntax

T ::=
   Bool
   Nat

Typing Rules

$$\text{true: Bool} \quad \text{[T-TRUE]}$$

$$\text{false: Bool} \quad \text{[T-FALSE]}$$

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

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

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

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

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

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

Checking a Few Types

```
tp : type.
nat : tp.
bool : tp.

check : tm -> tp -> type.
check_zero : check zero nat.
check_true : check true bool.
check_false : check false bool.
check_succ : check (succ T1) nat
   <- check T1 nat.
check_pred : check (pred T1) nat
   <- check T1 nat.
check_iszero : check (iszero T1) bool
   <- check T1 nat.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.
```

```
top
?- check zero T.
Solving...
T = nat.
More? y
No more solutions
?- check (if false zero (succ zero)) T.
Solving...
T = nat.
More? y
No more solutions
?- check (if (iszero (pred (succ zero))) false true) T.
Solving...
T = bool.
More? y
No more solutions
```

Let's make it harder: how can you state Preservation in Twelf?

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

Checking a Few Types

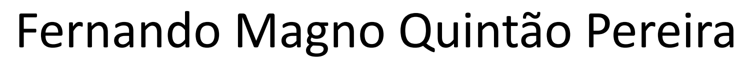
```
tp : type.
nat : tp.
bool : tp.

check : tm -> tp -> type.
check_zero : check zero nat.
check_true : check true bool.
check_false : check false bool.
check_succ : check (succ T1) nat
   <- check T1 nat.
check_pred : check (pred T1) nat
   <- check T1 nat.
check_iszero : check (iszero T1) bool
   <- check T1 nat.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.
```

```
top
?- check zero T.
Solving...
T = nat.
More? y
No more solutions
?- check (if false zero (succ zero)) T.
Solving...
T = nat.
More? y
No more solutions
?- check (if (iszero (pred (succ zero))) false true) T.
Solving...
T = bool.
More? y
No more solutions
```

Let's make it harder: how can you state Preservation in Twelf?

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



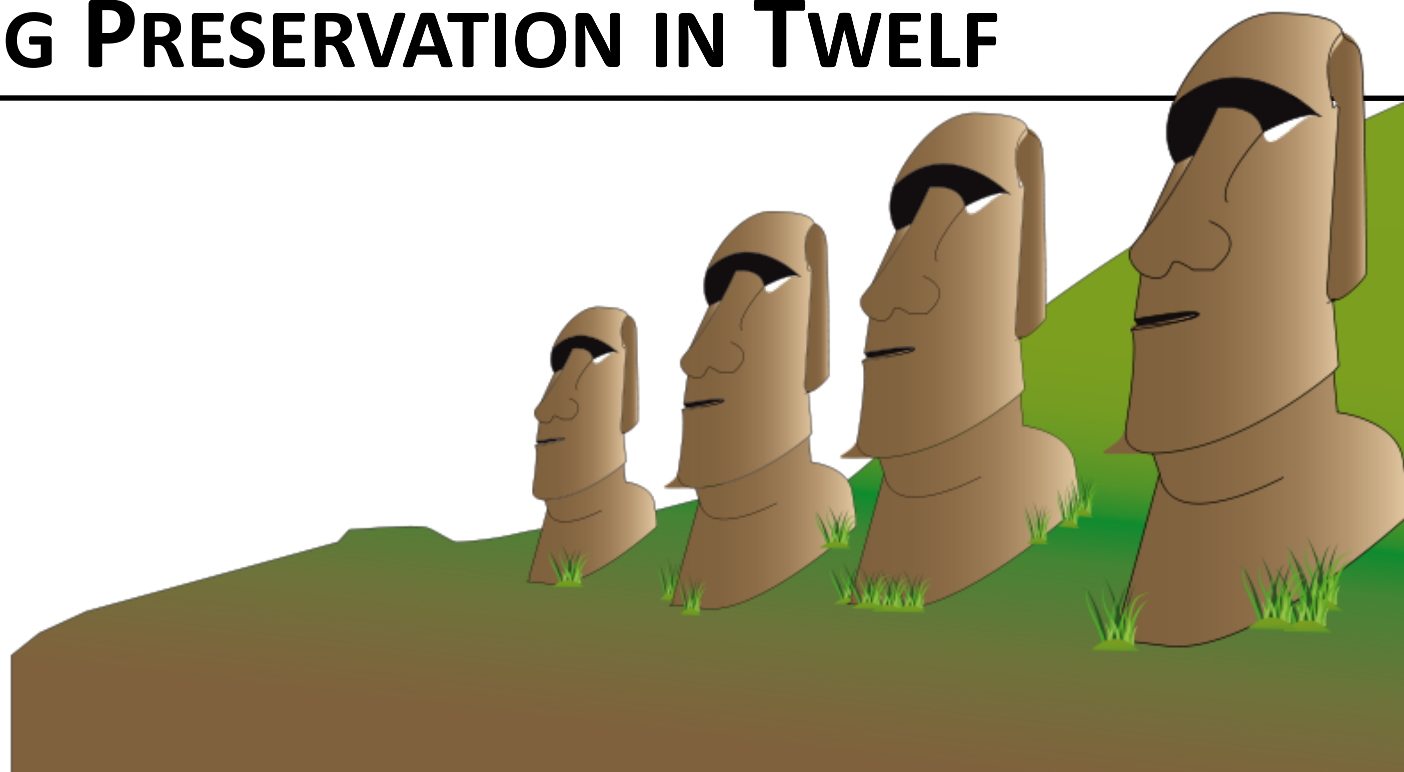
fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

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

PROVING PRESERVATION IN TWELF



fernando@dcc.ufmg.br

Statement of Preservation in Twelf

```
preservation : eval E E' -> check E T -> check E' T -> type.
%mode preservation +Eval +Check -Eval'.

%worlds () (preservation _ _ _).
%total E (preservation E _ _).
```

`sources.cfg`

```
lang.elf
lang.thm
```

Worlds

```
preservation : eval E E' -> check E T -> check E' T -> type.
%mode preservation +Eval +Check -Eval'.

%worlds () (preservation _ _ _).
%total E (preservation E _ _).
```

sources.cfg

```
lang.elf
lang.thm
```

1) What is happening if we try to build lang.thm just like this?

2) What does the error message is saying?

3) How can we fix this problem?

```
Coverage error --- missing cases:
{X1:tm} {X2:tm} {X3:eval X1 X2} {X4:check X1 nat} {X5:check X2 nat}
   |- preservation X3 X4 X5,
{X1:tm} {X2:tm} {X3:eval X1 X2} {X4:check X1 bool} {X5:check X2 bool}
   |- preservation X3 X4 X5.
```

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

Proving Preservation

```
preservation : eval E E' -> check E T -> check E' T -> type.
%mode preservation +Eval +Check -Eval'.

%worlds () (preservation _ _ _).
%total E (preservation E _ _).
```

```
eval: tm -> tm -> type.
eval_succ : eval (succ T) (succ T')
   <- eval T T'.
eval_pred_z : eval (pred zero) zero.
eval_pred_s : eval (pred (succ N)) N.
eval_pred : eval (pred T) (pred T')
   <- eval T T'.
eval_iszero_z : eval (iszero zero) true.
eval_iszero_s : eval (iszero (succ T)) false.
eval_iszero : eval (iszero T) (iszero T')
   <- eval T T'.
eval_ifTrue : eval (if true T2 T3) T2.
eval_ifFalse : eval (if false T2 T3) T3.
eval_if : eval (if T1 T2 T3) (if T1' T2 T3)
   <- eval T1 T1'.
```

We will prove preservation by doing induction on the **evaluation rules**. This means that, for each evaluation rule, we will have to show that it preserves types. We have 10 evaluation rules; hence, our proof will have 10 clauses.

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

Preservation of succ

```
eval: tm -> tm -> type.
eval_succ : eval (succ T) (succ T')
   <- eval T T'.

check : tm -> tp -> type.
check_succ : check (succ T) nat
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.
```

1) How can we prove preservation for the evaluation of succ?

2) Will we have to use induction?

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

Preservation of succ

```
eval: tm -> tm -> type.
eval_succ : eval (succ T) (succ T')
   <- eval T T'.

check : tm -> tp -> type.
check_succ : check (succ T) nat
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_succ : preservation
     (eval_succ Eval)
     (check_succ Check)
     (check_succ Check')
   <- preservation Eval Check Check'.
```

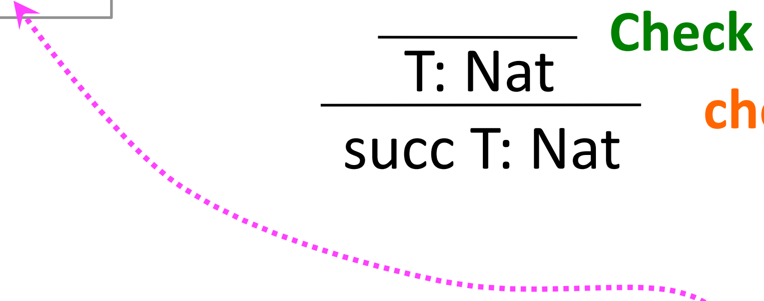
Do you understand what this proof is saying?

Understanding the Proof

```
preservation_succ : preservation
    (eval_succ Eval)
    (check_succ Check)
    (check_succ Check')
  <- preservation Eval Check Check'.
```

$$\dfrac{\overline{T \rightarrow T'}\ \textbf{Eval}}{(\text{succ } T) \rightarrow \text{succ } T'}\ \textbf{eval_succ}$$

```
eval: tm -> tm -> type.
eval_succ : eval (succ T) (succ T')
  <- eval T T'.
```



check_succ

```
check : tm -> tp -> type.
check_succ : check (succ T) nat
  <- check T nat.
```

$T \rightarrow T'$ and $T{:}Nat$ $\Rightarrow$ $T'{:}Nat$

```
preservation : eval E E' -> check E T -> check E' T -> type.
```

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

Preservation of the Evaluation of pred_z

```
eval: tm -> tm -> type.
eval_pred_z : eval (pred zero) zero.

check : tm -> tp -> type.
check_pred : check (pred T1) nat
   <- check T1 nat.

preservation : eval E E' -> check E T -> check E' T -> type.
```

1) How can we prove preservation for the evaluation of eval_pred_z?

2) Will we need induction on this proof?

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

Preservation of the Evaluation of pred_z

```
eval: tm -> tm -> type.
eval_pred_z : eval (pred zero) zero.

check : tm -> tp -> type.
check_pred : check (pred T1) nat
   <- check T1 nat.

check_zero : check zero nat.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_pred_z : preservation
     eval_pred_z
     (check_pred _)
     check_zero.
```

Why did we not need induction in this case?

What is this symbol '_' in the proof?

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

Preservation of the Evaluation of pred_s

```
eval: tm -> tm -> type.
eval_pred_s : eval (pred (succ N)) N.

check : tm -> tp -> type.
check_pred : check (pred T1) nat
   <- check T1 nat.

preservation : eval E E' -> check E T -> check E' T -> type.
```

How can we prove preservation for this evaluation rule?

Preservation of the Evaluation of pred_s

```
eval: tm -> tm -> type.
eval_pred_s : eval (pred (succ N)) N.

check : tm -> tp -> type.
check_pred : check (pred T1) nat
   <- check T1 nat.

check_succ : check (succ T) nat
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_pred_s : preservation
     eval_pred_s
     (check_pred (check_succ Check))
     Check.
```

Where did this **check_succ** come from?

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

Preservation of the Evaluation of pred_s

```
eval: tm -> tm -> type.
eval_pred_s : eval (pred (succ N)) N.

check : tm -> tp -> type.
check_pred : check (pred T1) nat
   <- check T1 nat.

check_succ : check (succ T) nat
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_pred_s : preservation
     eval_pred_s
     (check_pred (check_succ Check))
     Check.
```

eval (pred (succ N)) N

$$\frac{\dfrac{\text{Check}}{\text{check (succ N) nat}}}{\text{check (pred (succ N)) nat}}$$

If we consider eval_pred_s, then we know that we are talking about a term like (pred (succ N)). The only way to check the subterm is through check_succ.

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

Preservation of the Evaluation of pred

```
eval: tm -> tm -> type.
eval_pred : eval (pred T) (pred T')
   <- eval T T'.

check : tm -> tp -> type.
check_pred : check (pred T) nat
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.
```

1) How can we prove preservation for this evaluation rule?
2) Will we require induction in this case?

Preservation of the Evaluation of pred

```
eval: tm -> tm -> type.
eval_pred : eval (pred T) (pred T')
   <- eval T T'.

check : tm -> tp -> type.
check_pred : check (pred T) nat
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_pred : preservation
     (eval_pred Eval)
     (check_pred Check)
     (check_pred Check')
   <- preservation Eval Check Check'.
```

Preservation of the Evaluation of iszero_z

```
eval: tm -> tm -> type.
eval_iszero_z : eval (iszero zero) true

check : tm -> tp -> type.
check_iszero : check (iszero T1) bool
   <- check T1 nat.

preservation : eval E E' -> check E T -> check E' T -> type.
```

How can we prove preservation in this case?

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

Preservation of the Evaluation of iszero_z

```
eval: tm -> tm -> type.
eval_iszero_z : eval (iszero zero) true

check : tm -> tp -> type.
check_iszero : check (iszero T1) bool
  <- check T1 nat.

check_true : check true bool.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_iszero_z : preservation
    eval_iszero_z
    (check_iszero _)
    check_true.
```

The output of (iszero zero) is true, which is a boolean. Thus, we must conclude the proof with a **rule** that shows that true is well-typed.

Preservation of the Evaluation of iszero_s

```
eval: tm -> tm -> type.
eval_iszero_s : eval (iszero (succ T)) false.

check : tm -> tp -> type.
check_iszero : check (iszero T1) bool
   <- check T1 nat.

preservation : eval E E' -> check E T -> check E' T -> type.
```

How can we prove preservation in this case?

Preservation of the Evaluation of iszero_s

```
eval: tm -> tm -> type.
eval_iszero_s : eval (iszero (succ T)) false.

check : tm -> tp -> type.
check_iszero : check (iszero T1) bool
   <- check T1 nat.

check_false : check false bool.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_iszero_s : preservation
      eval_iszero_s
      (check_iszero _)
      check_false.
```

Preservation of the Evaluation of iszero

```
eval: tm -> tm -> type.
eval_iszero : eval (iszero T) (iszero T')
   <- eval T T'.

check : tm -> tp -> type.
check_iszero : check (iszero T) bool
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.
```

For this proof we will need induction. How do we know it?

Preservation of the Evaluation of iszero

```
eval: tm -> tm -> type.
eval_iszero : eval (iszero T) (iszero T')
   <- eval T T'.

check : tm -> tp -> type.
check_iszero : check (iszero T) bool
   <- check T nat.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_iszero : preservation
     (eval_iszero Eval)
     (check_iszero Check)
     (check_iszero Check')
   <- preservation Eval Check Check'.
```

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

Preservation of the Evaluation of ifTrue

```
eval: tm -> tm -> type.
eval_ifTrue : eval (if true T2 T3) T2.

check : tm -> tp -> type.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.

preservation : eval E E' -> check E T -> check E' T -> type.
```

Can you implement the proof of ifTrue in Twelf?

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

Preservation of the Evaluation of ifTrue

```
eval: tm -> tm -> type.
eval_ifTrue : eval (if true T2 T3) T2.

check : tm -> tp -> type.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_ifTrue : preservation
     eval_ifTrue
     (check_if CheckT3 CheckT2 CheckT1)
     CheckT2.
```

Notice the way rules are written: the last one, i.e., "check T3 T" is the **first** in the proof of the theorem.

Preservation of the Evaluation of ifFalse

```
eval: tm -> tm -> type.
eval_ifFalse : eval (if false T2 T3) T3.

check : tm -> tp -> type.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.

preservation : eval E E' -> check E T -> check E' T -> type.
```

Can you implement the proof of ifFalse in Twelf?

Preservation of the Evaluation of ifFalse

```
eval: tm -> tm -> type.
eval_ifFalse : eval (if false T2 T3) T3.

check : tm -> tp -> type.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_ifFalse : preservation
     eval_ifFalse
     (check_if CheckT3 CheckT2 CheckT1)
     CheckT3.
```

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

Preservation of the Evaluation of if

```
eval: tm -> tm -> type.
eval_if : eval (if T1 T2 T3) (if T1' T2 T3)
   <- eval T1 T1'.

check : tm -> tp -> type.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.

preservation : eval E E' -> check E T -> check E' T -> type.
```

And for this last proof, will we need induction?

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

Preservation of the Evaluation of if

```
eval: tm -> tm -> type.
eval_if : eval (if T1 T2 T3) (if T1' T2 T3)
   <- eval T1 T1'.

check : tm -> tp -> type.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.

preservation : eval E E' -> check E T -> check E' T -> type.

preservation_if : preservation
     (eval_if Eval)
     (check_if CheckT3 CheckT2 CheckT1)
     (check_if CheckT3 CheckT2 CheckT1')
     <- preservation Eval CheckT1 CheckT1'.
```

Can you explain, on the paper, how this proof works?

The order may look a bit weird. Can you match **these** rules with those in check_if?

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

Preservation of the Evaluation of if

```
check : tm -> tp -> type.
check_if : check (if T1 T2 T3) T
   <- check T1 bool
   <- check T2 T
   <- check T3 T.
```

```
preservation_if : preservation
      (eval_if Eval)
      (check_if
            CheckT3
            CheckT2
            CheckT1
      )
      (check_if CheckT3 CheckT2 CheckT1')
      <- preservation Eval CheckT1 CheckT1'.
```

This, sometimes, is a source of confusion to beginners: the rules in applications come in the inverse order of their declaration.

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

A Bit of History

- Most of the work on Twelf is due to Frank Pfenning and his collaborators.
- There are many tools, other than Twelf, that can be used to describe logical systems: ACL2, AUTOMATH, Coq, HOL, LEGO, Isabelle, MetaPRL, NuPRL PVS, TPS, Idris, etc.
- There are many, really many, papers in important conferences that use logical systems to prove properties.

- Pfenning, F., and Schurmann, C., "System description: Twelf – a meta-logical framework for deductive systems" CADE (1999) pp. 202-206
- Lee, D. Crary, K., and Harper, R. "Towards a mechanized metatheory of standard ML", POPL (2007), pp. 173-184
- Leroy, X., "Formal certification of a compiler back-end or: programming a compiler with a proof assistant", POPL (2006), pp. 42-54