OPERATIONAL SEMANTICS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from Chapter 3 of the book "Types and Programming Languages", by Benjamin C. Pierce.

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

The Need for Formality

- Many properties about programming languages can be proved formally via mechanical techniques.
  - There exist tools that certify these proofs automatically, e.g., COQ, Twelf, Isabelle, etc
- Formal proofs are useful in many ways:
  - they give designers confidence that their project is correct.
  - they uncover flaws in the design.
  - they serve as documentation (which sometimes can be checked by machines)
- The main weapon that the compiler designer has to prove properties about programming languages is *induction*.

Induction

- Induction is the main technique used in computer science to demonstrate properties of abstract algorithms, formal definitions and concrete implementations.
- Induction comes in many forms. We are more used to induction on natural numbers:

PRINCIPLE OF ORDINARY INDUCTION ON NATURAL NUMBERS: Suppose that P is a predicate on the natural numbers. Then if P(0) and, for all i, P(i) implies P(i+1), then P(n) holds for all n.

PRINCIPLE OF COMPLETE INDUCTION ON NATURAL NUMBERS: Suppose that P is a predicate on the natural numbers. Then if, for each natural number n, given P(i) for all i < n we can show P(n), then P(n) holds for all n.

Ordinary Induction

- To prove something by ordinary induction, we need to show two things:
  - The **base case**: the statement holds when n is the lowest possible value.
  - The **induction step**: if the statement holds for n, then it also holds for n + 1
- For instance:
  - **Basis**: the first domino will fall
  - **Induction**: whenever a domino falls, its neighbor falls as well.

- Axiomatically:

$$(\forall P, P(0) \wedge (\forall k \in N, P(k) \Rightarrow P(k+1))) \Rightarrow \forall n \in N, P(n)$$

Example: $n \geq 1 \Rightarrow 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2$

1) How do we prove the base case?

2) What is the smallest n that the theorem uses in its statement?

3) Why is the theorem true for this smallest n?

Example: $n \geq 1 \Rightarrow 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2$

The smallest n that the statement assumes is n = 1

Base case: $2 + 2^2 + 2^3 + \ldots + 2^n = 2^1 = 2$

$2^{n+1} - 2 = 2^{1+1} - 2 = 2$

1) And how do we prove the induction step?
2) What can we assume that is true by induction?
3) And how can we extrapolate this assumption to a proof of the entire theorem?

Example: $n \geq 1 \Rightarrow 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2$

The smallest n that the statement assumes is n = 1

Base case: $2 + 2^2 + 2^3 + \ldots + 2^n = 2^1 = 2$

$2^{n+1} - 2 = 2^{1+1} - 2 = 2$

Induction step: assume that the theorem is true for n = k, e.g.:

$2 + 2^2 + 2^3 + \ldots + 2^k = 2^{k+1} - 2$

If we let n = k + 1

$2 + 2^2 + 2^3 + \ldots + 2^k + 2^{k+1} = \ldots$

Example: $n \geq 1 \Rightarrow 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2$

Induction step: assume that the theorem is true for n = k, e.g.:

$2 + 2^2 + 2^3 + \ldots + 2^k = 2^{k+1} - 2$

If we let n = k + 1

$2 + 2^2 + 2^3 + \ldots + 2^k + 2^{k+1} =$

$(2 + 2^2 + 2^3 + \ldots + 2^k) + 2^{k+1} =$

$(2^{k+1} - 2) + 2^{k+1}$

$2 \times 2^{k+1} - 2$

$2^1 \times 2^{k+1} - 2$

$2^{k+1+1} - 2$

$2^{(k+1)+1} - 2$

A Toy Language

- In the rest of this presentation we will use a toy language to demonstrate different kinds of induction applied on formal language specifications.
- The language is rather simple, but the techniques that we use to prove properties about it can be ported to more complex bodies without modifications.
- On the right we have a description of this language's syntax using the BNF notation.

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

BNF is a well-known notation used to describe grammars, but, what is its actual semantics? In our example, which program terms are part of our language?

BNF and Induction

- The BNF grammar provides an inductive definition of the terms that are part of our toy language:

- [INDUCTIVE DEFINITION OF TERMS] The set of terms T is the smallest set such that:
  - $\{true, false, 0\} \in T$
  - if $t_1 \in T$, then $\{succ\ t_1, pred\ t_1, iszero\ t_1\} \subseteq T$
  - if $t_1 \in T$, $t_2 \in T$ and $t_3 \in T$, then $if\ t_1\ then\ t_2\ else\ t_3 \in T$

- We can rewrite this definition using the inference rules that usually we see in logical systems.

Do you remember the notation for these rules?

Inference Rules

- [Inductive Definition of Terms via Inference Rules] The set of terms T is the smallest set that satisfies the rules below:

$$\text{true} \in T \qquad \text{false} \in T \qquad 0 \in T$$

$$\frac{t_1 \in T}{\text{succ } t_1 \in T} \qquad \frac{t_1 \in T}{\text{pred } t_1 \in T} \qquad \frac{t_1 \in T}{\text{iszero } t_1 \in T}$$

$$\frac{t_1 \in T \quad t_2 \in T \quad t_3 \in T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in T}$$

Rules without premises are called axioms. How can we identify axioms in the BNF?

Concrete Definition of Terms

- [NATURAL DEFINITION OF TERMS] For each natural number i, define a set S_i as follows:
  - $S_0 = \{\}$
  - $S_{i+1} = \{true, false, 0\} \cup$
    $\{succ\ t_1, pred\ t_1, iszero\ t_1 \mid t_1 \in S_i\} \cup$
    $\{if\ t_1\ then\ t_2\ else\ t_3 \mid t_1, t_2, t_3 \in S_i\}$

Can you show that for each i, we have $S_i \subseteq S_{i+1}$?

  - We let "$S = \cup S_i, \forall i$" be the set of all terms

Notice that some terms are not well formed, e.g., pred true, or if 0 then true else 0

A First Proof by Induction

- Can you show that for each i, we have $S_i \subseteq S_{i+1}$?

Proof by ordinary induction on the natural numbers:

Base: i = 0
the statement is true, because $S_0 = \{\}$ which is a subset of every set

Inductive step:
by the induction hypothesis, the statement is true for i = k - 1, e.g., $S_{k-1} \subseteq S_k$, we must now show that it also holds for i = k, e.g, $S_k \subseteq S_{k+1}$. We must show that any term t in S_k is also in S_{k+1}. We know that S_k is the union of three sets.
• If $t \in \{true, false, 0\}$, then we are done.
• If $t = succ\ t_1$, $t_1 \in S_{k-1}$, then by the induction hypothesis, $t_1 \in S_k$. By the definition of S_{k+1}, we have that $t_1 \in S_{k+1}$.
• The case of $t = if\ t_1\ then\ t_2\ else\ t_3$ is similar: t_1, t_2 and $t_3 \in S_{k-1}$. By induction, t_1, t_2 and $t_3 \in S_k$. Hence, these terms are in S_{k+1}, by definition. □

Proving Equivalence

- The two definitions of terms seem very different, but they are in fact equivalent. How can we show this equivalence?

- [INDUCTIVE DEFINITION OF TERMS] The set of terms T is the smallest set such that:
  1. {true, false, 0} $\in$ T
  2. if $t_1 \in T$, then {succ t_1, pred t_1, iszero t_1} $\subseteq$ T
  3. if $t_1 \in T$, $t_2 \in T$ and $t_3 \in T$, then if t_1 then t_2 else $t_3 \in T$

- [NATURAL DEFINITION OF TERMS] For each natural number i, define a set S_i as follows:
  - $S_0 = \{\}$
  - $S_{i+1} = \{true, false, 0\} \cup \{succ\ t_1, pred\ t_1, iszero\ t_1 \mid t_1 \in S_i\} \cup \{if\ t_1\ then\ t_2\ else\ t_3 \mid t_1, t_2, t_3 \in S_i\}$

We let $S = \cup S_i, \forall i$ be the set of all terms

Proving Equivalence

- The two definitions of terms seem very different, but they are in fact equivalent. How can we show this equivalence?

- [INDUCTIVE DEFINITION OF TERMS] The set of terms T is the smallest set such that:
  1. {true, false, 0} $\in$ T
  2. if $t_1 \in T$, then {succ t_1, pred t_1, iszero t_1} $\subseteq$ T
  3. if $t_1 \in T$, $t_2 \in T$ and $t_3 \in T$, then if t_1 then t_2 else $t_3 \in T$

- [NATURAL DEFINITION OF TERMS] For each natural number i, define a set S_i as follows:
  - $S_0 = \{\}$
  - $S_{i+1} = \{true, false, 0\} \cup \{succ\ t_1, pred\ t_1, iszero\ t_1 \mid t_1 \in S_i\} \cup \{if\ t_1\ then\ t_2\ else\ t_3 \mid t_1, t_2, t_3 \in S_i\}$

We let $S = \cup S_i, \forall i$ be the set of all terms

We need to show two facts:

1) S satisfies the conditions that define T
2) Any set that satisfies the conditions in T contains S

Lets try to show 1 first. How to start?

Proving Equivalence

- Condition 1: {true, false, 0} ∈ T
  - This one is easy: these constants are in the definition of each S_i

- [Inductive Definition of Terms] The set of terms T is the smallest set such that:
  1. {true, false, 0} ∈ T
  2. if t_1 ∈ T, then {succ t_1, pred t_1, iszero t_1} ⊆ T
  3. if t_1 ∈ T, t_2 ∈ T and t_3 ∈ T, then if t_1 then t_2 else t_3 ∈ T

- [Natural Definition of Terms] For each natural number i, define a set S_i as follows:
  - $S_0 = \{\}$
  - S_{i+1} = {true, false, 0} ∪ {succ t_1, pred t_1, iszero t_1 | $t_1 \in S_i$} ∪ {if t_1 then t_2 else t_3 | $t_1, t_2, t_3 \in S_i$}

We let $S = \cup S_i, \forall i$ be the set of all terms

We need to show two facts:

1) S satisfies the conditions that define T ⇦
2) Any set that satisfies the conditions in T contains S

Proving Equivalence

- Condition 2 (same as 3): if $t_1 \in T$, then $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \subseteq T$
  - We are assuming that $t_1 \in T$. Thus, there exist some i, such that $t_1 \in S_i$
  - But, by the definition of S_{i+1}, we must have that $\text{succ } t_1 \in S_{i+1}$.
  - By the definition of S, we have that $\text{succ } t_1 \in S$

- [NATURAL DEFINITION OF TERMS] For each natural number i, define a set S_i as follows:
  - $S_0 = \{\}$
  - $S_{i+1} = \{\text{true}, \text{false}, 0\} \cup \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in S_i\} \cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\}$

  We let $S = \cup S_i, \forall i$ be the set of all terms

We need to show two facts:

1) S satisfies the conditions that define T ⇦
2) Any set that satisfies the conditions in T contains S

Now, we need to show number 2

Proving Equivalence

- Now we assume that there exists some set S' that satisfies the three conditions in the definition of terms. The idea is to show, by complete induction, that every $S_i \subseteq S'$

- [INDUCTIVE DEFINITION OF TERMS] The set of terms T is the smallest set such that:
  1. {true, false, 0} $\in$ T
  2. if $t_1 \in T$, then {succ t_1, pred t_1, iszero t_1} $\subseteq$ T
  3. if $t_1 \in T$, $t_2 \in T$ and $t_3 \in T$, then if t_1 then t_2 else $t_3 \in T$

Remember

PRINCIPLE OF COMPLETE INDUCTION ON NATURAL NUMBERS: Suppose that P is a predicate on the natural numbers. Then if, for each natural number n, given P(i) for all i < n we can show P(n), then P(n) holds for all n.

In our case, we assume that $S_j \subseteq S'$ for all j < i; thus, we must show that $S_i \subseteq S'$

We need to show two facts:

1) S satisfies the conditions that define T
2) Any set that satisfies the conditions in T contains S 

Proving Equivalence

- We assume that $S_j \subseteq S'$ for all $j < i$; thus, we can show that $S \subseteq S'$

- [NATURAL DEFINITION OF TERMS] For each natural number i, define a set S_i as follows:
  1. $S_0 = \{\}$
  2. $S_{i+1} = \{true, false, 0\} \cup$
     $\{succ\ t_1, pred\ t_1, iszero\ t_1 \mid t_1 \in S_i\}$
     $\cup\ \{if\ t_1\ then\ t_2\ else\ t_3 \mid t_1, t_2, t_3 \in S_i\}$

  We let $S = \cup S_i, \forall i$ be the set of all terms

The definition of S_i has two clauses; thus, there are two cases to consider: (1) i = 0 and (2) i > 0

If i = 0, then $S_0 = \{\}$. Because the empty set is a subset of every set, we are done.

And what about i > 0?

We need to show two facts:

1) S satisfies the conditions that define T
2) Any set that satisfies the conditions in T contains S

Proving Equivalence

Do you remember what is complete induction?

- We assume that $S_j \subseteq S'$ for all $j < i$; thus, we can show that $S \subseteq S'$

- [INDUCTIVE DEFINITION OF TERMS] The set of terms T is the smallest set such that:
  1. {true, false, 0} $\in$ T
  2. if $t_1 \in T$, then {succ t_1, pred t_1, iszero t_1} $\subseteq$ T
  3. if $t_1 \in T$, $t_2 \in T$ and $t_3 \in T$, then if t_1 then t_2 else $t_3 \in T$

[NATURAL DEFINITION OF TERMS] (only S_{i+1})
S_{i+1} = {true, false, 0} $\cup$ {succ t_1, pred t_1, iszero t_1 | $t_1 \in S_i$} $\cup$ {if t_1 then t_2 else t_3 | $t_1, t_2, t_3 \in S_i$}

If $i > 0$, then we can do complete induction on i, assuming that S_i is the union of three sets.

If the terms in S_i are constants, e.g., true, false, or 0, then we are done, because by (1) on the left, this constant meets the condition of T

If the term in S_i is succ t_1 (or pred t_1, or iszero t_1), for some $t_1 \in S_j$, by induction we know that t_1 meets the conditions of T, and by rule (2) we know that succ t_1 also does it.

The case for if t_1 then t_2 else t_3 is similar to the previous case.

We need to show two facts:

1) S satisfies the conditions that define T
2) Any set that satisfies the conditions in T contains S 

Proving Equivalence

- We assume that $S_j \subseteq S'$ for all $j < i$; thus, we can show that $S \subseteq S'$

- [INDUCTIVE DEFINITION OF TERMS] The set of terms T is the smallest set such that:
  1. {true, false, 0} $\in$ T
  2. if $t_1 \in T$, then {succ t_1, pred t_1, iszero t_1} $\subseteq$ T
  3. if $t_1 \in T$, $t_2 \in T$ and $t_3 \in T$, then if t_1 then t_2 else $t_3 \in T$

So, we have just showed that any S_i meets the conditions that define T.

But $S = \cup S_i$. Because $S_j \subseteq S'$, we have that $S \subseteq S'$.

We need to show two facts:

1) S satisfies the conditions that define T
2) Any set that satisfies the conditions in T contains S 

Inductive Definitions

- The recursive nature of syntactic terms gives us the opportunity to define functions on terms inductively.
- We can find the constants appearing in a term t via the following SML definition:

```
datatype Term = TRUE | FALSE | ZERO | SUCC of Term | PRED of Term |
                ISZERO of Term | IF of Term * Term * Term

fun const TRUE = [TRUE]
  | const FALSE = [FALSE]
  | const ZERO = [ZERO]
  | const (SUCC T) = const T
  | const (PRED T) = const T
  | const (ISZERO T) = const T
  | const (IF (T1, T2, T3)) = (const T1) @ (const T2) @ (const T3)
```

```
$> const (IF (TRUE, PRED ZERO, ZERO));
$> val it = [TRUE,ZERO,ZERO] : Term list
```

Inductive Definitions

- We can define the size of a term t via the following SML function:

```
datatype Term = TRUE | FALSE | ZERO | SUCC of Term | PRED of Term |
                ISZERO of Term | IF of Term * Term * Term

fun size TRUE = 1
  | size FALSE = 1
  | size ZERO = 1
  | size (SUCC T) = size T + 1
  | size (PRED T) = size T + 1
  | size (ISZERO T) = size T + 1
  | size (IF (T1, T2, T3)) = (size T1) + (size T2) + (size T3) + 1
```

```
$> size (IF (TRUE, PRED ZERO, ZERO));
$> val it = 5 : int
```

Inductive Definitions

- We can define the depth of a term t via the following SML function:

```
datatype Term = TRUE | FALSE | ZERO | SUCC of Term | PRED of Term |
                ISZERO of Term | IF of Term * Term * Term

fun max(i1, i2, i3) = if i1 > i2 andalso i1 > i3 then i1
                      else if i2 > i3 then i2 else i3

fun depth TRUE = 1
  | depth FALSE = 1
  | depth ZERO = 1
  | depth (SUCC T) = depth T + 1
  | depth (PRED T) = depth T + 1
  | depth (ISZERO T) = depth T + 1
  | depth (IF (T1, T2, T3)) = max((depth T1), (depth T2), (depth T3)) + 1
```

```
$> depth (IF (TRUE, PRED ZERO, ZERO));
$> val it = 3 : int
```

Proof by Induction on Terms

PRINCIPLE OF THE STRUCTURAL INDUCTION ON TERMS: If, for each term s, given P(r) for all immediate subterms r of s we can show P(s), then P(s) holds for all s.

- Example: The number of distinct constants in a term t is no greater than the size of t:

```
fun const TRUE = [TRUE]
  | const FALSE = [FALSE]
  | const ZERO = [ZERO]
  | const (SUCC T) = const T
  | const (PRED T) = const T
  | const (ISZERO T) = const T
  | const (IF (T1, T2, T3)) =
    (const T1) @ (const T2) @
    (const T3)
```

```
fun size TRUE = 1
  | size FALSE = 1
  | size ZERO = 1
  | size (SUCC T) = size T + 1
  | size (PRED T) = size T + 1
  | size (ISZERO T) = size T + 1
  | size (IF (T1, T2, T3)) =
    (size T1) + (size T2) +
    (size T3) + 1
```

Can you prove this lemma?

Proof by Induction on Depth

- The number of distinct constants in a term t is no greater than the size of t.
- We assume this property for every subterm of a term, and show it for the term itself.
- **Base**: t has no subterms. In this case, t must be a constant, and we have that the number of distinct constants (NDC) of t is 1, which equals its size.

```
fun const TRUE = [TRUE]
  | const FALSE = [FALSE]
  | const ZERO = [ZERO]
```

```
fun size TRUE = 1
  | size FALSE = 1
  | size ZERO = 1
```

Proof by Structural Induction

- **Inductive step**: we must reason about the terms with subterms. There are two cases:
  - t = succ t_1 (or t = pred t_1, or t = iszero t_1). By the induction hypothesis, $NDC(t_1) \leq SIZE(t_1)$, and by the implementation of `const` and `size`, we are done
  - t = if t1 then t2 else t3. By the induction hypothesis, $NDC(t_i) \leq SIZE(t_i)$, $i \in \{1, 2, 3\}$. Thus, we conclude our proof by looking into the implementation of `const` and `size`.

```
fun const (SUCC T) = const T
  | const (PRED T) = const T
  | const (ISZERO T) = const T
  | const (IF (T1, T2, T3)) =
    (const T1) @ (const T2) @
    (const T3)
```

```
fun size (SUCC T) = size T + 1
  | size (PRED T) = size T + 1
  | size (ISZERO T) = size T + 1
  | size (IF (T1, T2, T3)) =
    (size T1) + (size T2) +
    (size T3) + 1
```

Examples: even numbers

- Consider the definition of numbers in ML:

```
datatype nat = Zero | Succ of nat
```

- We can define the even numbers as follows:

```
fun isEven Zero = true
  | isEven (Succ Zero) = false
  | isEven (Succ (Succ n)) = isEven n
```

- And we can define doubles as follows:

```
fun double Zero = Zero
  | double (Succ n) = Succ (Succ (double n))
```

Show that isEven (double n) = true

1. How could you set up induction in this case?
2. What would be the base case?
3. What about the inductive hypothesis?

```
datatype nat = Zero | Succ of nat

fun double Zero = Zero
  | double (Succ n) = Succ (Succ (double n))

fun isEven Zero = true
  | isEven (Succ Zero) = false
  | isEven (Succ (Succ n)) = isEven n
```

Show that isEven (double n) = true

- **Base**: isEven (double Zero) = isEven Zero = true
- **Inductive step**:
  - isEven (double (Succ n)) = isEven (Succ (Succ (double n)))
  - By induction:
    - isEven (double n) = true
  - Thus:
    - isEven (Succ (Succ (double n))) = true

```
datatype nat = Zero | Succ of nat

fun double Zero = Zero
  | double (Succ n) = Succ (Succ (double n))

fun isEven Zero = true
  | isEven (Succ Zero) = false
  | isEven (Succ (Succ n)) = isEven n
```

Example: Binary Trees

- Consider the definition of numbers in ML:

```
datatype 'a Tree = Empty | Node of 'a Tree * 'a * 'a Tree
```

- We can define the number of vertices as follows:

```
fun numV Empty = 1
  | numV (Node(t1, _, t2)) = 1 + numV t1 + numV t2
```

- And we can define the number of edges as follows:

```
fun numE Empty = 0
  | numE (Node(t1, _, t2)) = 2 + numE t1 + numE t2
```

Show that numVertices t = numEdges t + 1

1. How could you set up induction in this case?
2. What would be the base case?
3. What about the inductive hypothesis?

```
datatype 'a Tree = Empty | Node of 'a Tree * 'a * 'a Tree

fun numV Empty = 1
  | numV (Node(t1, _, t2)) = 1 + numV t1 + numV t2

fun numE Empty = 0
  | numE (Node(t1, _, t2)) = 2 + numE t1 + numE t2
```

Show that numVertices t = numEdges t + 1

Base: numVertices Empty = 1 = numEdges Empty + 1

Inductive step:

numVertices (Node (t1, _, t2)) = 1 + numVertices t1 + numVertices t2

numEdges (Node (t1, _, t2)) = 2 + numEdges t1 + numEdges t2

- *By induction*:

  numVertices t1 = numEdges t1 + 1 *and* numVertices t2 = numEdges t2 + 1

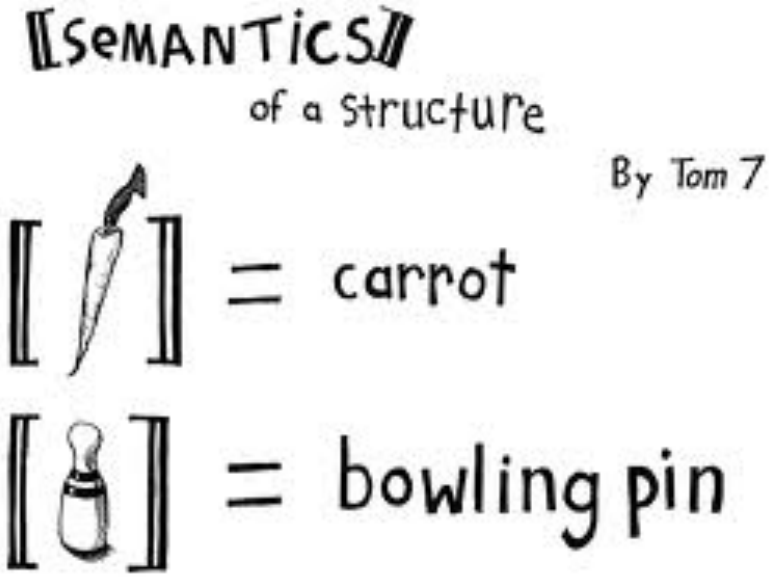
- *Thus*: numVertices (Node (t1, _, t2)) = 1 + numVertices t1 + numVertices t2
  = 1 + numEdges t1 + 1 + numEdges t2 + 1
  = 3 + numEdges t1 + numEdges t2
  = 1 + (2 + numEdges t1 + numEdges t2)
  = 1 + numEdges (Node (t1, _, t2))

```
datatype 'a Tree = Empty | Node of 'a Tree * 'a * 'a Tree

fun numV Empty = 1
  | numV (Node(t1, _, t2)) = 1 + numV t1 + numV t2

fun numE Empty = 0
  | numE (Node(t1, _, t2)) = 2 + numE t1 + numE t2
```

Compilers Laboratory

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

Operational Semantics



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

Operational Semantics

- Operational Semantics specifies the behavior of a programming language by defining a simple *abstract machine* for it.
- This machine is abstract because it interprets the terms of the language directly, instead of via micro-instructions, like an ordinary hardware does.
- The machine's behavior is defined by a transition function, that for each term and a state, tells us the next abstract state that can be achieved.
- The meaning of a term t is the final state that the machine reaches when started with t as its initial state.

Evaluation Rules for Boolean Expressions

Syntax	*Semantics*	
t ::=		
true	if true then t_2 else t_3 → t_2	[E-IfTrue]
| false		
| if t then t else t	if false then t_2 else t_3 → t_3	[E-IfFalse]
v ::=		
true	$\dfrac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$	[E-If]
| false		

- We read t → t' as *t evaluates to t' in one step*. If the original state of the abstract machine is t, then after the evaluation, this state is t'

Evaluation Rules for Boolean Expressions

Syntax	*Semantics*	
t ::=		
true	if true then t_2 else t_3 → t_2	[E-IfTrue]
| **false**		
| **if t then t else t**	if false then t_2 else t_3 → t_3	[E-IfFalse]
v ::=		
true	$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$	[E-If]
| false		

- Usually we define the evaluation rules with regards to a syntax. **Some elements** of this syntax are already in a normal form, i.e., cannot be transformed further. For the **others**, we need evaluation rules.

Evaluation Rules for Boolean Expressions

Syntax	*Semantics*	
t ::= true | false | if t then t else t	if true then t_2 else t_3 → t_2	[E-IfTrue]
	if false then t_2 else t_3 → t_3	[E-IfFalse]
v ::= true | false	$\dfrac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$	[E-If]

- Some rules are called **axioms**. They are always applicable, once we have a term in the right format. Other rules are called **theorems**, because they require some **premises**.

Evaluation Rules for Boolean Expressions

Syntax	*Semantics*	
t ::=		
true	if true then t_2 else t_3 → t_2	[E-IfTrue]
| false		
| if t then t else t	if false then t_2 else t_3 → t_3	[E-IfFalse]
v ::=		
true	$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$	[E-If]
| false		

- Given these rules, how can the expression below be evaluated?

```
if true then (if false then false else false) else true
```

The Evaluation Order

Semantics

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

- There is only one order in which the expression below can be evaluated.
- The rule [E-If] forces the evaluation of the conditional, before any internal term can be evaluated.

Could we have a rule that lets us evaluate internal parts of the then or else branches before evaluating the conditional?

```
if true then (if false then false else false) else true
```

The Evaluation Order

Semantics

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

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

- There is only one order in which the expression below can be evaluated.
- The rule [E-If] forces the evaluation of the conditional, before any internal term can be evaluated.

Could we have a rule that lets us evaluate **internal** parts of the then or else branches before evaluating the conditional?

```
if true then (if false then false else false) else true
```

Induction on Derivations

- The derivation rules let us build *derivation trees* for terms.
- The fact that these trees have a well-defined depth lets us use a proof technique called induction on derivations.

$$\dfrac{\dfrac{\dfrac{}{\text{if true then false else false} \rightarrow \text{false}}\,[\text{E-IfTrue}]}{\text{if } \textbf{if true then false else false} \text{ then true else true} \rightarrow \text{if } \textbf{false} \text{ then true else true}}\,[\text{E-If}]}{\text{if } \textbf{if if true then false else false then true else true} \text{ then false else false} \rightarrow \text{if } \textbf{if false then true else true} \text{ then false else false}}\,[\text{E-If}]$$

Principle of The Induction on Derivation: suppose P is a predicate on derivations of evaluation statements. If, for each derivation D, given P(C) for all immediate subderivations C we can show P(D), then P(D) holds for all D.

Determinacy

- [DETERMINACY OF ONE-STEP EVALUATION] if $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$

The proof of this theorem is by induction on the derivation tree. We must, for each derivation rule, show that it satisfies the theorem.

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

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

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

[E-IFTRUE] If the rule used in the derivation $t \rightarrow t'$ is [E-IFTRUE], then t necessarily must have the form "if true then t_2 else t_3". But, in this case, there is only one possibility for $t \rightarrow t''$, and we know that $t' = t'' = t_2$

[E-IFFALSE] The reasoning in this case is analogous to that seen in [E-IFTRUE] .

[E-IF] We know that $t =$ if t_1 then t_2 else t_3 and that $t_1 \rightarrow t_1'$.

How can we apply induction in this last case, [E-IF]?

Determinacy

- [DETERMINACY OF ONE-STEP EVALUATION] if $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$

The proof of this theorem is by induction on the derivation tree. We must, for each derivation rule, show that it satisfies the theorem.

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

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

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

[E-IFTRUE] If the rule used in the derivation $t \rightarrow t'$ is [E-IFTRUE], then t necessarily must have the form "if true then t_2 else t_3". But, in this case, there is only one possibility for $t \rightarrow t''$, and we know that $t' = t'' = t_2$

[E-IFFALSE] The reasoning in this case is analogous to that seen in [E-IFTRUE] .

[E-IF] We know that t = if t_1 then t_2 else t_3 and that $t_1 \rightarrow t_1'$. If $t_1 \rightarrow t_1'$, then we know that t_1 is not a value. Thus, the rule that give us $t \rightarrow t''$ must be [E-IF], and we have that if t_1 then t_2 else $t_3 \rightarrow$ if t_1'' then t_2 else t_3. We then apply induction on $t_1 \rightarrow t_1'$ and $t_1 \rightarrow t_1''$, to conclude that $t_1' = t_1''$. Therefore, if t_1' then t_2 else t_3 = if t_1'' then t_2 else t_3.

Normal Form

- A term t is in *normal form* if no evaluation rule applies to it. In other words, there is no t' such that t → t'.
- It is easy to see in our system of conditional tests that every value is in normal form.
- The converse, i.e., every normal form is a value, is not necessarily true, although it is true for that system.
  - As an example, had we a zero value, then "if 0 then true else false" would be in normal form, but this term is not a value

Prove that, in the system on the right, if t is in normal form, then t is a value.

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

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

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

Normal Form

- If a term t in the system below is in normal form, then this term is a value.

The proof is by structural induction on t. If t is not a value, then it must have the structure "if t_1 then t_2 else t_3" for some t_1, t_2 and t_3. Let go over the possible forms of t_1.

Case t_1 = true, then by Rule [E-IFTRUE] we have a step, contradicting our initial assumption that t, e.g., "if t_1 then t_2 else t_3" is in normal form.

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

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

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

The case t_1 = false is analogous to the previous case.

If t_1 is neither true nor false, then t_1 is not a value. The induction hypothesis lets us conclude that t_1 is not in normal form. Thus, there exists t_1' such that $t_1 \rightarrow t_1'$. By Rule [E-IF] we can take a step, contradicting our initial assumption.

Multi-Step Evaluation

- The multi-step evaluation relation →* is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that
  1. if t → t' then t →* t'
  2. t →* t for all t
  3. if t → t' and t' →* t", then t →* t"
- In other words:

Could you rewrite these three definitions as inference rules?

Multi-Step Evaluation

- The multi-step evaluation relation $\rightarrow^*$ is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that
  - if $t \rightarrow t'$ then $t \rightarrow^* t'$
  - $t \rightarrow^* t$ for all t
  - if $t \rightarrow t'$ and $t' \rightarrow^* t''$, then $t \rightarrow^* t''$
- In other words:

$$\frac{t \rightarrow t'}{t \rightarrow^* t'} \qquad t \rightarrow^* t \qquad \frac{t \rightarrow t' \quad t' \rightarrow^* t''}{t \rightarrow^* t''}$$

What do you think: if $t \rightarrow^* u'$ and $t \rightarrow^* u''$, where u' and u'' are normal forms, then is it the case that $u' = u''$?

Multi-Step Evaluation

- The multi-step evaluation relation $\rightarrow^*$ is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that
  - if $t \rightarrow t'$ then $t \rightarrow^* t'$
  - $t \rightarrow^* t$ for all t
  - if $t \rightarrow t'$ and $t' \rightarrow^* t''$, then $t \rightarrow^* t''$
- In other words:

$$\frac{t \rightarrow t'}{t \rightarrow^* t'} \qquad t \rightarrow^* t \qquad \frac{t \rightarrow t' \quad t' \rightarrow^* t''}{t \rightarrow^* t''}$$

- This theorem, e.g., **if $t \rightarrow^* u'$ and $t \rightarrow^* u''$, where u' and u'' are normal forms, then $u' = u''$**, is a corollary of Determinacy, e.g., if $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$

Termination

- Proving that a program terminates is undecidable in general; however, if the programming language is very simple, then we can show that their programs always terminate.
- Our toy language is, indeed, very simple.
- [TERMINATION OF EVALUATION] For every term t, there exists some normal form t', such that $t \rightarrow^* t'$

Can you prove this theorem?

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

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

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

Termination

- [TERMINATION OF EVALUATION] For every term t, there exists some normal form t', such that t →* t'

The proof is by complete induction on the size of t, e.g., size(t).

Base: size(t) = 1. Thus, t must be either true or false, and t→*t already leads to normal form.

Induction step: size(t) = n. The term t must be such that t = if t_1 then t_2 else t_3. Lets look into t_1.

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

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

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

If t_1 = true, then by rule [E-IFTRUE], we have that if t_1 then t_2 else t_3 → t_2. By induction on t_2, we know that it reaches a normal form. The case for t_1 = false is similar.
If t_1 = if t_1' then t_2' else t_3', then we can apply the induction hypothesis on t_1. Thus, we have that t_1 →* u_1, and u_1 is in normal form, thus, size(u_1) = 1 < size(t_1). We finish by applying induction on if u_1 then t_2 else t_3.

Arithmetic Rules

Syntax

t ::= …
  0
  | succ t
  | pred t
  | iszero t

v ::= …
  nv

nv ::=
  0
  | succ nv

Why do you think we have this weird rule?

And why this new syntactic category?

Semantics

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

pred 0 → 0 [E-PREDZERO]

pred(succ nv_1) → nv_1 [E-PREDSUCC]

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

iszero 0 → true [E-ISZEROZERO]

iszero (succ nv1) → false [E-ISZEROSUCC]

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

Stuck Evaluation

- A term t is stuck if t is in normal form, but t is not a value.

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]

Can you give examples of terms that are stuck?

Stuck Evaluation

- A term t is stuck if t is in normal form, but t is not a value.

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]

pred 0 → 0 [E-PREDZERO]

pred(succ nv_1) → nv_1 [E-PREDSUCC]

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

iszero 0 → true [E-ISZEROZERO]

iszero (succ nv1) → false [E-ISZEROSUCC]

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

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

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

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

In our language of **booleans**, every term would evaluate to a value. Now we have normal forms that are not values. What is the key factor that arithmetic expressions have added to our evaluation rules?

Big Step Evaluation

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

$\text{not true} \rightarrow \text{false}$ [E-NotTrue]

$\text{not false} \rightarrow \text{true}$ [E-NotFalse]

$$\frac{t \rightarrow t'}{\text{not } t \rightarrow \text{not } t'}$$ [E-Not]

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

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

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

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

These are two different styles of semantic description. On the left we have the so called "small step" style. On the right we have the "big step" style. Why do they have these names?

Proving Operational Equivalences

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

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

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

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

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

Proving Operational Equivalences

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

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

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

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

if t_1 then t_2 else t_3
$\equiv$
if not t_1 then t_3 else t_2?

We must do a case analysis on the derivation rules. For each way to evaluate an if clause, we must show equivalence.

In how many different ways can we evaluate an if clause?

Proving Equivalence if we assume [B-TRUE]

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

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

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

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

if t_1 then t_2 else t_3

$\equiv$

if not t_1 then t_3 else t_2?

If we use the rule [B-TRUE], then we know that (i) $t_1 \Downarrow$ true, we know that (ii) $t_2 \Downarrow v$, and we know that (iii) if t_1 then t_2 else t_3 evaluates to v.

From (i) plus [B-NOTTRUE], we know that (iv) not $t_1 \Downarrow$ false.

By rule [B-FALSE], plus (iv), plus (ii), we know that if not t_1 then t_3 else t_2 evaluates to v

Proving Equivalence if we assume [B-FALSE]

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

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

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

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

if t_1 then t_2 else t_3

$\equiv$

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 $t_2 \Downarrow v$

Proving Equivalences on the Small Step Style

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

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

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

$$\text{not true} \rightarrow \text{false} \qquad \text{[E-NotTrue]}$$

$$\text{not false} \rightarrow \text{true} \qquad \text{[E-NotFalse]}$$

$$\frac{t \rightarrow t'}{\text{not } t \rightarrow \text{not } t'} \qquad \text{[E-Not]}$$

If we try to show the same equivalence, e.g., "if t_1 then t_2 else $t_3 \equiv$ if not t_1 then t_3 else t_2", then we will have a bit more of work. First of all, we have three different ways to evaluate an "if" term, whereas before we had only two. Second, we will have to apply induction on the derivation rules. Furthermore, we must rely on the fact that if $t \rightarrow t'$, then t and t' are semantically equivalent.

Proving Equivalence if we assume [E-IfTrue]

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

if true then t_2 else $t_3 \rightarrow t_2$ [E-IfTrue]

if false then t_2 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]

not true → false [E-NotTrue]

not false → true [E-NotFalse]

$$\frac{t \rightarrow t'}{\text{not } t \rightarrow \text{not } t'}$$ [E-Not]

If we use the rule [E-IfTrue], then we know:
(i) t_1 = true
(ii) if t_1 then t_2 else $t_3 \rightarrow t_2$

By rule [E-NotTrue] plus (i), we know that:
(iii) not t_1 = false

By rule [E-IfFalse] plus (iii), we know that if not t_1 then t_3 else $t_2 \rightarrow t_2$

Proving Equivalence if we assume [E-IFFALSE]

$$\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \equiv \text{if not } t_1 \text{ then } t_3 \text{ else } t_2?$$

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

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

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

$$\text{not true} \rightarrow \text{false} \quad \text{[E-NOTTRUE]}$$

$$\text{not false} \rightarrow \text{true} \quad \text{[E-NOTFALSE]}$$

$$\frac{t \rightarrow t'}{\text{not } t \rightarrow \text{not } t'} \quad \text{[E-NOT]}$$

If we use the rule [E-IFFALSE], then we know:

(i) t_1 = false

(ii) if t_1 then t_2 else $t_3 \rightarrow t_3$

By rule [E-NOTFALSE] plus (i), we know that:

(iii) not t_1 = true

By rule [E-IFTRUE] plus (iii), we know that if not t_1 then t_3 else $t_2 \rightarrow t_3$

Proving Equivalence if we assume [E-IF]

This is the complicated case, because we must use induction, combined with the fact that if $t \rightarrow t'$, then $t \equiv t'$

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

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

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

$$\text{not true} \rightarrow \text{false} \quad \text{[E-NOTTRUE]}$$

$$\text{not false} \rightarrow \text{true} \quad \text{[E-NOTFALSE]}$$

$$\frac{t \rightarrow t'}{\text{not } t \rightarrow \text{not } t'} \quad \text{[E-NOT]}$$

If we use the rule [E-IF], then we know that:

(i) $\exists\ t_1'$, such that $t_1 \rightarrow t_1'$

(ii) if t_1 then t_2 else $t_3 \rightarrow$ if t_1' then t_2 else t_3

By rule [E-NOT] plus (i), we know that:

(iii) not $t_1 \rightarrow$ not t_1'

We know that not t_1 is bigger than not t_1' in the derivation, due to (iii). Thus, we can apply induction. We know that, by induction, (A) if t_1' then t_2 else t_3 is equivalent to (B) if not t_1' then t_3 else t_2 (iv). But we know that if t_1 then t_2 else t_3 is equivalent to (A), due to (i), and we know that if not t_1 then t_2 else t_3 is equivalent to (B) by (iii)

Big-Step vs Small-Step: a Comparison

if true then t_2 else t_3 → t_2

if false then t_2 else t_3 → t_3

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

not true → false

not false → true

$$\frac{t \rightarrow t'}{\text{not } t \rightarrow \text{not } t'}$$

Small-step style

The **big-step** style tends to have less inference rules. Hence, proving properties about programs may be easier in this style. Furthermore, because terms evaluate directly to values, in general induction is not necessary on these proofs.

Nevertheless, the small-step style is more often used than the big-step style. What are the advantages of the small-step style?

$$\frac{t_1 \Downarrow \text{true} \quad t_2 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v}$$

$$\frac{t_1 \Downarrow \text{false} \quad t_3 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v}$$

$$\frac{t \Downarrow \text{true}}{\text{not } t \Downarrow \text{false}}$$

$$\frac{t \Downarrow \text{false}}{\text{not } t \Downarrow \text{true}}$$

Big-step style

Big-Step vs Small-Step: a Comparison

$$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2$$

$$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3$$

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

$$\text{not true} \rightarrow \text{false}$$

$$\text{not false} \rightarrow \text{true}$$

$$\frac{t \rightarrow t'}{\text{not } t \rightarrow \text{not } t'}$$

Small-step style

The big-step style has problems with programs that do not terminate. Because it does not show intermediate steps in the evaluation, nothing can be said about these programs.

Additionally, the big step style makes it difficult to establish the evaluation order of terms. On the other hand, given its step-by-step nature, the **small-step** style gives the semanticist total control over this ordering.

$$\frac{t_1 \Downarrow \text{true} \quad t_2 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v}$$

$$\frac{t_1 \Downarrow \text{false} \quad t_3 \Downarrow v}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v}$$

$$\frac{t \Downarrow \text{true}}{\text{not } t \Downarrow \text{false}}$$

$$\frac{t \Downarrow \text{false}}{\text{not } t \Downarrow \text{true}}$$

Big-step style

And yet, for languages whose programs always terminate, big-step yields simpler interpreters….

A Prolog Interpreter in the Big-Step Style

```
eval(zero, zero).

eval(true, true).

eval(false, false).

eval(if(T1, T2, _), V) :- eval(T1, true), eval(T2, V).

eval(if(T1, _, T3), V) :- eval(T1, false), eval(T3, V).

eval(succ(T), succ(V)) :- eval(T, V).

eval(pred(T), zero) :- eval(T, zero).

eval(pred(T), V) :- eval(T, succ(V)).

eval(iszero(T), true) :- eval(T, zero).

eval(iszero(T), false) :- eval(T, succ(_)).
```

Which queries can we perform on this Prolog program?

A Prolog Interpreter in the Big-Step Style

```
eval(zero, zero).

eval(true, true).

eval(false, false).

eval(if(T1, T2, _), V) :- eval(T1, true), eval(T2, V).

eval(if(T1, _, T3), V) :- eval(T1, false), eval(T3, V).

eval(succ(T), succ(V)) :- eval(T, V).

eval(pred(T), zero) :- eval(T, zero).

eval(pred(T), V) :- eval(T, succ(V)).

eval(iszero(T), true) :- eval(T, zero).

eval(iszero(T), false) :- eval(T, succ(_)).
```

```
?- eval(succ(zero), V).
V = succ(zero).

?- eval(pred(succ(zero)), V).
V = zero.

?- eval(iszero(pred(succ(zero))), V).
V = true.

?- eval(if(false, zero, succ(zero)), V).
V = succ(zero).

?- eval(if(iszero(zero), zero, succ(zero)), V).
V = zero.

?- eval(if(iszero(pred(succ(zero))), zero, succ(zero)), V).
V = zero.
```

A Bit of History

- The style of operational semantics that we have used is due to Plotkin.
- The notion of structural induction seem to have been introduced in computer science by Burstall.
- There are many ways to describe the semantics of programming languages: denotational, axiomatic and operational seem to be the best known methods.

- Plotkin, G, "A Structural Approach to Operational Semantics", Technical Report, Aarhus University, Denmark (1981)
- Burstall, M. "Proving Properties of Programs by Structural Induction", The Computer Journal, 12(1):41-48 (1969)