COMPSCI 3MI3 - Principles of Programming Languages

Topic 4 - Operational Semantics

NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

Semantic Styles

Evaluation of Booleans via Small Step Operational Semantics

Proof of Determinacy

Normal Forms

Termination of Booleans

Arithmetic Semantics

Arguing Semantics

In the last topic we examined various strategies for formally specifying language syntax. By specifying the **semantics** of our Untyped Arithmetic Expressions language, we will have a complete and working model of a programming language that's ready for implementation[1]!

In general there are three major semantic styles.

- Operational Semantics
  - Small-Step
  - Big-Step
- Denotational Semantics
- Axiomatic Semantics

[1] By you! During an assignment!

Operational Semantics

Under operational semantics, we define how a language behaves by specifying an **abstract machine** for it.

- An abstract machine is abstract because it operates on the terms of the language themselves.
  - This is in contrast to a regular machine, which must first translate the terms to instructions in the computer processor's instruction set.
- For simple languages (such as UAE), the *state* of this abstract machine is simply a term of the language.
- The machine's behaviour is specified by a **transition function**.
- The *meaning* of a term is the final state of the abstract machine at the point of halting.

Operational Semantics (cont.)

Small Step

For each state, gives either the results of a single simplification, or indicates the machine has halted.

Big Step

A single transition within the abstract machine evaluates the term to its final result.

Sometimes, we will use two different operational semantics for the same language. For example:

- One might be abstract, on the terms of the language as used by the programmer.
- Another might represent the structures used by the compiler/interpreter.

In the above case, proving correspondance between these two machines is *proving the correctness of an implementation of the language*, i.e., proving the correctness of the compiler itself.

Denotational Semantics

Rather than viewing meaning as a sequence of machine states, under denotational semantics, the meaning of a term is taken to be a *mathematical object*, such as a function or a number. We need two things:

- **Semantic Domains** $\rightarrow$ A collection of sets of mathematical objects which we can map terms to.
- **Interpretation Function** $\rightarrow$ A mapping between the terms of our language and the elements of our semantic domains.

For example:

- In the previous topic, we said we would sometimes simplify expressions, like writing $\mathtt{succ}(\mathtt{succ}(\mathtt{succ}0))$ as 3.
  - $\mathbb{N}$ is the semantic domain.
  - The interpretation function maps terms of UAE to $\mathbb{N}$.
    - Nominally, by counting the number of succ invokations.

Denotational Semantics (cont.)

The search for semantic domains to model language features is an area of computer science research known as **domain theory**, and is especially applicable to functional programming languages.

Denotational semantics have a number of advantages

- It avoids messy implementation details
- Properties of the semantic domains can be used to reason about program behaviours. For example:
  - Proving two programs have the same behaviour.
  - Proving a program meets some specification.
  - Proving that certain behaviours are impossible, be they desireable or undesirable.

Axiomatic Semantics

- Operational and Denotational Semantics start with a language behaviour, and then derive laws of reasoning from these definitions.
- **Axiomatic Semantics** work in the reverse direction. We take the laws themselves as the definition of the language.
- This means that the meaning of a term is precisely *that which can be proved about it*.
- This normally takes the form of assertions about the modification of program states made by program instructions.
- This approach is closely related to **Hoare Logic**.

Semantic Cage Match

During the early development of these semantic styles (1960s-70s), operational semantics were generally regarded as inferior to the more abstract styles.

- Denotational semantics as a style has a tough time dealing with concurrency and nondeterminism, which is fatal in the internet age.
- Axiomatic semantics run into a similar issue with procedures.

In reality, each of these styles made contributions to the others, but operational semantics eventually won out because of its simplicity. *In this course, we will be using Operational Semantics exclusively.*

Operational Semantics of Booleans

Let us first consider the small step semantics of the boolean elements of UAE.

$\mathbb{B}$ *(untyped)*

Syntax

t	::=		*terms:*
		true	*constant true*
		false	*constant false*
		if t then t else t	*conditional*
v	::=		*values:*
		true	*true value*
		false	*false value*

Evaluation $\boxed{t \longrightarrow t'}$

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

Let's Break it Down...

In the left-hand column we see two syntactic specifications.

- The first, is a re-stating of the terms we are interested in, particularly those relating to booleans and boolean operations.
- The second defines a subset of terms, called **values**, which are the possible final results of evaluation.

The right-hand column defines an **evaluation relation** on terms, written $t \implies t'$

- We pronounce this "t evaluates to t' in one step."
- This relation is defined by the following three rules of inference.

Inference Rules

Let's examine our rules of inference in detail, starting with:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \rightarrow t_2 \tag{1}$$

If our conditional expression is a literal truth value, we can simplify the expression to whatever term is represented by t_2.

Similarily for the second rule:

$$\texttt{if false then } t_2 \texttt{ else } t_3 \rightarrow t_3 \tag{2}$$

A literal false value can simplify the if statement to just the else-branch term.

Inference Rules (cont.)

The third evaluation rule (E-If) is less straightforward:

$$\frac{t_1 \rightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad (3)$$

- If we can evaluate t_1 to t_1', we can substitute the guard term of an if-expression to t_1', assuming it was t_1 originally.
- In fact, we can use a *different* abstract machine to establish $t_1 \implies t_1'$.
  - You can think of this as the abstract machine spawning an instance of itself in order to evaluate the guard expression.
- Note the absence of similar rules for the true and false branches of this if-expression.
- We've baked an **order of evaluation** into our semantics, because the if-expression *must* be evaluated *before* branches can be evaluated.

The One-Step Evaluation Relation

Our evaluation relation $\rightarrow$ is defined as the smallest binary relation on terms satisfying the three rules given.

- When a pair (t, t') is in our evaluation relation, we say that "the evaluation statement $t \rightarrow t'$ is **derivable**."
- By smallest, we mean that the relation contains no pairs other than those derived from instances of our inference rules.
  - Since there are an infinite number of terms, there are also an infinite number of instances of the inference rules, and an infinite number of pairs in our evaluation relation.
- We can demonstrate the derivability of a given pair using **formal derivation**.
  - The textbook uses inference tree style proofs, but here I'm using natural deduction style proofs because they are easier to follow and fit on the slides better[2].

[2] I tried both.

Example Evaluation

Consider the following expression in UAE:

```
if true then (if false then false else false) else true
```
(4)

Let's set t to the inner if expression so the derivation tree fits on the page:

$$t = \texttt{if false then false else false} \tag{5}$$

We prove the results of the evaluation as follows:

(1)	`if true then` t `else true`	
(2)	t	E-IfTrue on (1)
(3)	*false*	E-IfFalse on (2)
(4)	`if true then` t `else true` $\rightarrow$ `false`	E-intro on (1) to (3)

Example Evaluation 2

$$\texttt{if (if true then false else true) then true else false} \tag{6}$$

$$t = \texttt{if true then false else true} \tag{7}$$

(1)	`if` t `then true else false`	
(2)	t	Assume
(3)	`false`	E-IfTrue on (2)
(4)	$t \rightarrow$ `false`	E-intro on (2) and (3)
(5)	`if false then true else false`	E-If on (1) and (4)
(6)	`if` t `then true else false` $\rightarrow$ `false`	E-Intro on (1) to (5)

Observations

Note that we have no ability to evaluate the inner if expression before the outer one, because there is no evaluation rule which gives us that ability!

- If we had rules for evaluating the branches of an if statement, we would have to make a choice about which rule to follow to evaluate the statement.
- This would make the evaluation relation **ambiguous** and **non-deterministic**.
- The way our semantics are currently arranged, our evaluation relation is **deterministic**, providing a clear **evaluation strategy**.

Induction on Derivations

Fact: $t \to t'$ is derivable iff there is a derivation proof with $t \to t'$ as its conclusion.

- This fact can be used to reason about the properties of the evaluation relation.

[Induction on Derviations]

- If we can show that
  - Given that the same property holds for all sub-derivations,
  - The property must necessarily hold for the super-derivation,
- We may conclude that the property holds for all possible derivations.

Here, **sub-derivation** is a derivation occuring *after* the derivation at hand. So, in example evaluation 1 above, if E-IfTrue is the derivation, E-IfFalse is it's sub-derivation.

Determinacy

As an excercise in induction on derivations, let us consider the following theorem.

THEOREM :
[Determinacy of One-Step Evaluation]

$$t \to t' \wedge t \to t'' \implies t' = t'' \tag{8}$$

That is to say, if a term t evaluates to t', and the same term evaluates to t'', t' and t'' must be the same term.

Proof of Determinacy

We will prove the above theorem by induction on a derivation of $t \to t'$.

- We will assume the desired result for all smaller derivations.
- We will then perform case analysis over the three available evaluation rules.

Note that this is *not* induction over the *length* of a derivation sequence. This technique bears more resemblance to **structural induction** than to **complete induction**.

We're going to go through this one in quite a bit of detail, so buckle up!

Proof of Determinacy I

Just a reminder of the thing we're trying to prove...

$$t \to t' \wedge t \to t'' \implies t' = t'' \tag{9}$$

We can make certain determinations about our cases based on the derivation used in $t \to t'$. That derivation must be one of the following:

- E-IfTrue, E-IfFalse, E-If

So let's start with E-IfTrue.

- If E-IfTrue was the last derivation, we know that t must be of the form:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \tag{10}$$

Proof of Determinacy II

In order for t_1 to not equal t_2, we would need to apply a different rule to it, other than E-IfTrue, so let's examine the possibilities.

- E-IfFalse $>>$ This strategy may only apply if $t_1 = \texttt{false}$. Since $t_1 = \texttt{true}$, this is not a legal strategy.
- E-If $>>$ This strategy can only be applied if t_1 can be evaluated. Since $t_1 \in V$ (that is, it is a member of our set of values), further evaluation is impossible.

Therefore, the $\rightarrow$ in $t \rightarrow t''$ must be E-IfTrue. Therefore, $t' = t''$ in this case. We can make a symmetrical argument in the case

where we start with E-IfFalse.

Proof of Determinacy III

Now, let us consider the case of E-If. We know that:

$$t = \texttt{if}\ \ s_1\ \ \texttt{then}\ \ s_2\ \ \texttt{else}\ \ s_3 \tag{11}$$

And that, via E-If,

$$t' = \texttt{if}\ \ s_1'\ \ \texttt{then}\ \ s_2\ \ \texttt{else}\ \ s_3 \tag{12}$$

And we have the premise:

$$s_1 \to s_1' \tag{13}$$

Proof of Determinacy IV

We can apply the same argument from the E-IfTrue case to this as well. We know that the only valid rule to apply to $t \rightarrow t''$ is E-If, so we know that:

$$t'' = \texttt{if } s_1'' \texttt{ then } s_2 \texttt{ else } s_3 \tag{14}$$

And we also have access to the premise:

$$s_1 \rightarrow s_1'' \tag{15}$$

However, this is insufficient to prove our final case!

- We have $s_1 \rightarrow s_1'$ and $s_1 \rightarrow s_1''$, but in order for $t' = t''$, we have to show that $s_1' = s_1''$, since that is the only point of difference between t' and t''.

Proof of Determinacy V

We haven't needed the induction hypothesis so far, but here it is:

- We assume, for all sub-derivations of $t \to t'$, that:

$$s \to s' \wedge s \to s'' \implies s' = s'' \tag{16}$$

And there we have it!

- Since we know $s_1 \to s_1'$ and $s_1 \to s_1''$, we may conclude $s_1' = s_1''$!
- $s_1' = s_1''$, so both t' and t'' must be the same term!

Therefore, the inductive step holds!

Proof of Determinacy VI

For completeness, we should also address our base case briefly.

- In this case, the base case is the final derivation, which will yield a value, rather than another expression.
- The only derivation rules which can yeild a value are E-IfTrue and E-IfFalse.
- A very similar argument may be employed to show that $t \rightarrow t'$ cannot use a different rule than $\rightarrow t''$, because the guard term of t:
  1. Must be a value (i.e., must be irreducible), because this is the final evaluation step.
  2. Can be either true or false, but not both, via our language definition.

We can therefore conclude that both our base case and inductive step hold.

Quod Erat Demonstrandum


You FOOLS!!
This isn't even my NORMAL FORM!!!!
imgflip.com

Normal Form

As programmers, *how* and expression is evaluated is much less interesting than *what* it evaluates to.

- A term t is in it's **Normal Form** if no evaluation rule applies to it.
- That is, there is no t' such that $t \rightarrow t'$
- For the present system of untyped booleans, we know intuitively that the normal form of all terms is `true` or `false`.

Values Are In Normal Form!

THEOREM :
[Every Value is in Normal Form]
Our set of values is in this case just `true` and `false`.

- Given that a term is in normal form if no evaluation rule applies to it, and
- Given that we have no rules for evaluating `true` and `false`,
- We can conclude that they are in normal form.

This is somewhat trivial to demonstrate, but we need to be careful as we add terms to UAE not to disrupt it.

If It's In Normal Form... It's a Value!

THEOREM :
[If t is in Normal Form, t is a Value]
Proof:

- Let's say that t is *not* a value. We seek to show that t can't be in a state where an evaluation rule does not apply to it.
- Let's also introduce an induction hypothesis, and assume that subterms of t are either in normal form or can be evaluated.
- If t is not a value, for the present subset of UAE, there is only one term that is not a value. t must therefore have the form:

$$\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \tag{17}$$

If It's In Normal Form... It's a Value! II

- Whether or not any evaluation rule applies to it depends on what t_1 is. There are three possibilities:
  - t_1 is the `true` value. If this is the case, E-IfTrue applies.
  - Similarly for $t_1 = \texttt{false}$ and E-IfFalse.
  - Since the two above cases take care of the cases where the subterm t_1 is a value, via the induction hypothesis, t_1 must be able to be evaluated. This means that E-If applies to t.
- Therefore, if t is not a value, it can be evaluated, meaning it is not in normal form.
- Therefore, values are the only thing in normal form.
- Therefore, if it's in normal form, it's a value!

We will see later on that this isn't necessarily the case for all languages. *succ* 0 cannot be evaluated, but is also not a value.

Termination

A property of extreme importance, when it comes to programs in various programming languages, is the property of **termination**.

- In our context, evaluation **terminates** when the term we are evaluating reaches some normal form.
- In order for a language like UAE to have any practical utility whatsoever, evaluation must terminate.
- In practical terms, it should be impossible for a finite term to take an infinite amount of time to terminate.
  - i.e., we want to make sure no evaluation rule returns us to a previous state, thus causing an *infinite loop* in evaluation.

Although the **halting problem** is undecidable for larger languages, UAE is small enough that we can prove termination unequivocally.


ALL TERMS
WILL TERMINATE
OR BE TERMINATED
imgflip.com

Multi-step Evaluation ($\rightarrow^*$)

Let's define the **multi-step evalution relation** $\rightarrow^*$

- Multi-step Evaluation is the **reflexive, transitive closure** of one-step evaluation. That is, it is the smallest relation such that:

$$t \rightarrow t' \implies t \rightarrow^* t' \tag{18}$$

$$\forall t \in \mathcal{T} \mid t \rightarrow^* t \tag{19}$$

$$t \rightarrow^* t' \wedge t' \rightarrow^* t'' \implies t \rightarrow^* t'' \tag{20}$$

This last property is critical, because it means that, if you start with t_0, and you end up with t_n through a series of applications of single-step evaluation, multi-step evaluation can jump directly from t_0 to t_n, or any point in-between with one application of $\rightarrow^*$.

Uniqueness of Normal Form

THEOREM :
[Uniqueness of Normal Forms]
Consider $t, u, u' \in \mathcal{T}$, where u and u' are normal forms.

$$t \rightarrow^* u \wedge t \rightarrow^* \implies u = u' \tag{21}$$

That is to say, if u and u' are both the results of multi-step evaluation of t, *and* are in normal form, they must be the same thing!

- This is a fancy way of saying that a term may not eventually evaluate to more than one normal form.
- This is a fancy way of saying that fully evaluated terms must have only one solution.

Uniqueness of Normal Form (Proof)

From the determinacy of single-step evaluation, we already know that:

$$t \to t' \wedge t \to t'' \implies t' = t'' \tag{22}$$

This reduces what we need to prove from:

- Terms can't have multiple solutions

To...

- Terms can't have multiple normal forms in a given chain of single-step evaluations.

Here, we use the definition of the normal form:

- A term t is in normal form if there are no evaluation rules which apply to it.

There can't be multiple normal forms along our evaluation path, because something in normal form can't be evaluated further.

QED!

Termination of Evaluation

THEOREM :
[Termination of Evaluation]

$$\forall t \in \mathcal{T}\, \exists t' \in \mathcal{N} | t \rightarrow^* t' \tag{23}$$

Where $\mathcal{N}$ is the set of normal forms of $\mathcal{T}$.

To prove this, let's make some observations about single-step evaluations.

- Previously, we had defined the size of a term analogously to the size of a tree.
- Let's introduce a lemma...

Does the Size Reduce?

LEMMA :

$$s \to s' \implies size(s) > size(s'). \tag{24}$$

Proving this lemma shouldn't be difficult. Because s is being evaluated, we know it's not in normal form, meaning it must be an if-expression.

- According to the definition of *size*:

$$size(\texttt{if}\ s_1\ \texttt{then}\ s_2\ \texttt{else}\ s_3) = size(s_1) + size(s_2) + size(s_3) + 1 \tag{25}$$

Yes it does!

- Using structural induction, we will assume that our lemma applies to subterms of s.
- We know that s will evaluate to either s_2 or s_3, or that s_1 will be evaluated to some s_1'.
  - Niether $size(s_2)$ nor $size(s_3)$ can be greater than $size(s_1) + size(s_2) + size(s_3) + 1$ because *that's not how addition works!*
  - Our induction hypothesis states that $size(s_1) > size(s_1')$, so...

$$size(s_1) + size(s_2) + size(s_3) + 1 > size(s_1') + size(s_2) + size(s_3) + 1 \tag{26}$$

We can therefore conclude that the size of s must be less than the size of s'.

Decreasing Chains!

We have therefore demonstrated that evaluation represents a *decreasing chain with respect to size*. Now, we need an argument for whether this chain is **finite** or `infinite`.

- This can be determined quite easily, by deciding if the domain of size is well founded.
- There are no sizes possible which are less than 1.
  - You could make the argument that the size of $\varnothing$ is 0, though we haven't defined it that way.
- The domain of *size* is therefore $\mathbb{N}$, which is **well founded**.
- Therefore, evaluation chains can't be infinite. They therefore terminate!

More Semantics!

So far, we have only been concerned with the Boolean terms of UAE. Time to make things more complicated!

- As we construct our semantics for arithmetic operations, we are also interested in the problem of nonsense programs.
  - With just the boolean terms, it was impossible to create nonsense.
- Although this is still an untyped expression language, terms have limited applicability. Applying a term to an incompatible term will create nonsense.
- When a term is in normal form, but is not a value, we say that the term is **stuck**.

Extended UAE Semantics

$\mathbb{B}$ $\mathbb{N}$ *(untyped)* — *Extends* **B** *(3-1)*

New syntactic forms

t	::=	...	*terms:*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=	...	*values:*
		nv	*numeric value*
nv	::=		*numeric values:*
		0	*zero value*
		succ nv	*successor value*

New evaluation rules — $t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \quad \text{(E-Succ)}$$

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

$$\texttt{pred (succ } nv_1\texttt{)} \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \quad \text{(E-Pred)}$$

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

$$\texttt{iszero (succ } nv_1\texttt{)} \longrightarrow \texttt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero } t_1 \longrightarrow \texttt{iszero } t_1'} \quad \text{(E-IsZero)}$$

Numeric Values

In order to properly define our semantics, we need to be able to distinguish between boolean and numeric values.

- To this end, we create two new syntactic categories: $\mathcal{V}$ and $\mathcal{NV}$, for (Boolean) values and Numeric values.
- Items of both categories have all the properties of values, such as correspondance to normal form and uniqueness.
- Note the inclusion of `succ nv` in `nv`.
  - This means that, if we have an unbroken chain of `succ` terms terminating in zero, this constitues a value.
  - This asserts the natural numbers as being values!

Why are Numeric Values Necessary?

Consider the following expression in UAE:

```
iszero succ pred succ 0
```

Which rule applies?

1. E-IsZeroSucc
2. E-IsZero
3. Both
4. Neither

Those rules again...

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false} \quad \text{(E-IsZeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero}\ t_1 \longrightarrow \texttt{iszero}\ t_1'} \quad \text{(E-IsZero)}$$

To Resolve Ambiguity!

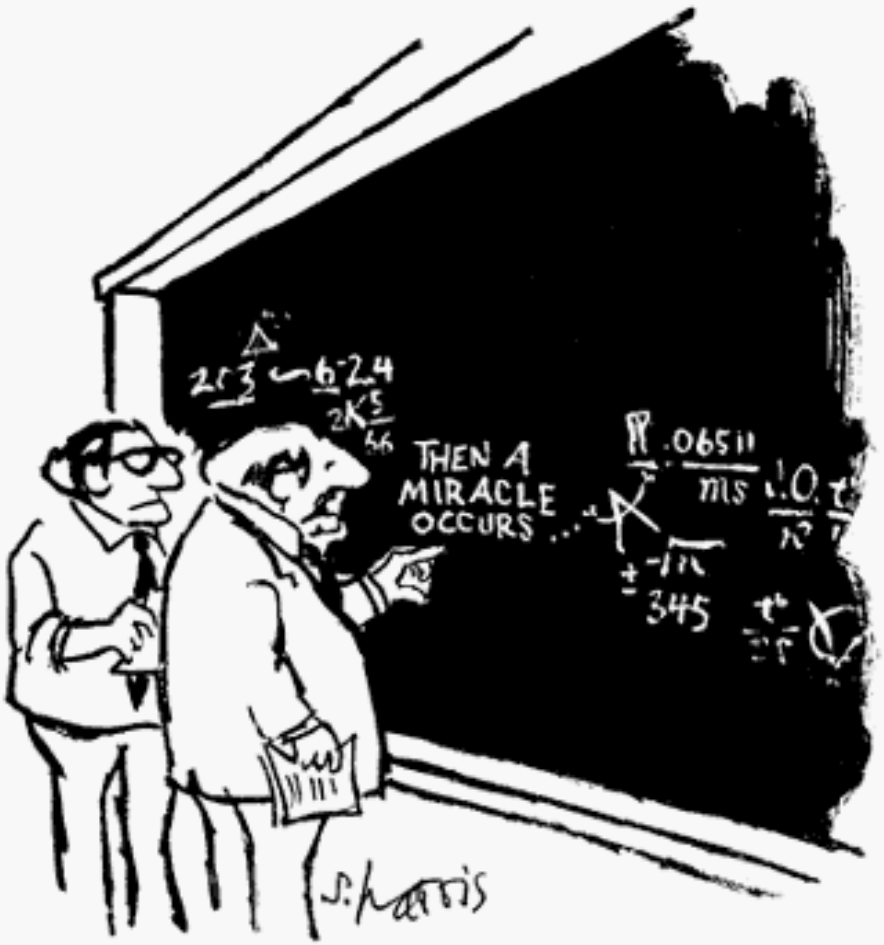
The answer is E-IsZero!

- We can't use E-IsZeroSucc, because `succ pred succ 0` is not a value. *It can be evaluated!*
- If E-IsZeroSucc did not require a numeric value, the rule would also apply to evaluatable `succ` terms.
- This would cause a rather nasty **ambiguity**, destroying our language's determinacy! Basically...

$$t \to t' \wedge t \to t'' \text{ no longer implies } t' = t'' \tag{27}$$

This is something to avoid at all costs, even if we could show that it doesn't break the determinacy of mult-step semantics.

Last Slide Comic



"I think you should be more explicit here in step two."