COMPSCI 3MI3 - Principles of Programming Languages

Towards the Lambda Calculus

J. Carette

McMaster University

Fall 2023

Adapted from "Types and Programming Languages" by Benjamin C. Pierce and Nick Moore's material.

Language Safety

"Informally [...], safe languages can be defined as ones that make it impossible to shoot yourself in the foot while programming." (Pierce, 2002)

- Safety refers to whether or not a language protects its own abstractions.
  - In Haskell, lists can only be accessed in the normal way.
  - In C, pointer manipulation can be used to violate the bounds of arrays to read adjacent data.
- Language safety can be enforced either statically or dynamically, though often a combined approach is used.
  - Haskell, for example, checks array bounds dynamically.

	Statically Checked	Dynamically Checked
Safe	ML, Haskell, Java	Lisp, Scheme, Perl, Postscript
Unsafe	C, C++	

- "In modern languages, the type system is often taken as the foundation of the language's design, and the organizing principle, in light of which every other aspect of the design is considered." (Pierce, 2002)

EBNF Examples

The following grammar describes how we write the integers.

$\langle integer \rangle ::= [-] \langle digit \rangle \{ \langle digit \rangle \}$

$\langle digit \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- Here, we have two rules. The top level rule, `<integer>`, and a sub-rule, `<digit>`.
- "|" denotes a set of options. For example, `<digit>` can be any of the numbers indicated but no others.
- "[]" denote something which is optional. For example, the negative sign denoting a negative integer may be absent.
- "{}" denote zero or more repetitions of the contents. For example, zero or more digits may follow the first.

EBNF Examples

The following grammar is for writing hexadecimal integers.

⟨*Hex Integer*⟩ ::= [-]⟨*hex digit*⟩{⟨*hex digit*⟩}

⟨*hex digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

Note that the grammar describing decimal integers is a **subset of** the grammar for hexadecimal integers.

- Although every `Integer` is also a `Hex Integer`, this does not imply anything about how either group would be interpretted.
- EBNF describes *syntax*, not *semantics*!

Death and Syntaxes!

Here are a few more example grammars:

- A Python list: items in quotes are taken literally.

$\langle List \rangle$ ::= '[' $\langle Object\ List \rangle$ ']'

$\langle Object\ List \rangle$ ::= $\langle Object \rangle$, $\langle Object\ List \rangle$ | $\langle Object \rangle$

- A Python function.

$\langle Function \rangle$ ::= def $\langle Identfier \rangle$ ($\langle Argument\ List \rangle$) : '\n\t' $\langle Statement\ List \rangle$

Untyped Arithemetic Expressions (UAE) - Syntax

$$
\begin{array}{rcl}
\langle t \rangle & ::= & \text{true} \\
& | & \text{false} \\
& | & \text{if } \langle t \rangle \text{ then } \langle t \rangle \text{ else } \langle t \rangle \\
& | & 0 \\
& | & \text{succ } \langle t \rangle \\
& | & \text{pred } \langle t \rangle \\
& | & \text{iszero } \langle t \rangle
\end{array}
$$

- `t` is a **metavariable**.
- EBNF is a **metalanguage** (a language which describes languages), and `t` is a variable of that language.

Define Your Terms!

Equivalent descriptions formalisms:

- Defining Terms Inductively
- Defining Terms Using Inference Rules
- Defining Terms Using Set Theory

Inductive Definition

The set of *terms* is the smallest set $\mathcal{T}$ such that:

$$\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T} \tag{1}$$

$$t_1 \in \mathcal{T}, \implies \{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1\} \subseteq \mathcal{T} \tag{2}$$

$$t_1, t_2, t_3 \in \mathcal{T} \implies \{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\} \subseteq \mathcal{T} \tag{3}$$

Terms By Rules of Inference

Similar to rules of *natural deduction* used in the presentation of logical systems.

$\mathcal{T}$ shall be the set of terms defined by the following rules:

$$\frac{}{\texttt{true} \in \mathcal{T}} \qquad \frac{}{\texttt{false} \in \mathcal{T}} \qquad \frac{}{0 \in \mathcal{T}}$$

$$\frac{t \in \mathcal{T}}{\texttt{succ}\ t \in \mathcal{T}} \qquad \frac{t \in \mathcal{T}}{\texttt{pred}\ t \in \mathcal{T}} \qquad \frac{t \in \mathcal{T}}{\texttt{iszero}\ t \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T} \qquad t_3 \in \mathcal{T}}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \in \mathcal{T}}$$

Terms, Set-Theoretically

For each natural number i, define the set S_i as follows:

$$
\begin{aligned}
S_0 &= \varnothing \\
S_{i+1} &= \{true, false, 0\} \\
&\quad \cup \{t \in S_i \mid \texttt{succ}\ t, \texttt{pred}\ t, \texttt{iszero}\ t\} \\
&\quad \cup \{t_1, t_2, t_3 \in S_i \mid \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\}
\end{aligned}
$$

And let

$$S = \bigcup_i S_i$$

Does $\mathcal{T} = S$?

Yes. By induction – prove $\mathcal{T} \subseteq S$ and also $S \subseteq \mathcal{T}$.

Size of a Term

The size of a term t, written $size(t)$, is written as follows:

$$
\begin{array}{lcl}
size(\texttt{true}) & = & 1 \\
size(\texttt{false}) & = & 1 \\
size(\texttt{0}) & = & 1 \\
size(\texttt{succ}\ t_1) & = & size(t_1) + 1 \\
size(\texttt{pred}\ t_1) & = & size(t_1) + 1 \\
size(\texttt{iszero}\ t_1) & = & size(t_1) + 1 \\
size(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) & = & size(t_1) + size(t_2) + size(t_3) + 1
\end{array}
$$

Depth of a Term

The depth of a term t, written $depth(t)$ is written as follows:

$$
\begin{aligned}
depth(\texttt{true}) &= 1 \\
depth(\texttt{false}) &= 1 \\
depth(\texttt{0}) &= 1 \\
depth(\texttt{succ}\ t_1) &= depth(t_1) + 1 \\
depth(\texttt{pred}\ t_1) &= depth(t_1) + 1 \\
depth(\texttt{iszero}\ t_1) &= depth(t_1) + 1 \\
depth(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) &= max(depth(t_1), depth(t_2), \\
&\quad depth(t_3)) + 1
\end{aligned}
$$

Induction on Size and Depth

With these new definitions, we can now introduce three exciting new forms of induction!

[Induction on Size]

- If, for $s \in \mathcal{T}$
  - Given $P(r)$ for all r such that $size(r) < size(s)$
  - we can show $P(s)$
- We may conclude $\forall s \in \mathcal{T} \mid P(s)$

[Induction on depth]

- If, for $s \in \mathcal{T}$
  - Given $P(r)$ for all r such that $depth(r) < depth(s)$
  - we can show $P(s)$
- We may conclude $\forall s \in \mathcal{T} \mid P(s)$

These two forms of induction are derived from Complete Induction over $\mathbb{N}$

Structural Induction over Terms

[Structural Induction Over Terms]

- If, for $s \in \mathcal{T}$
  - We can show $P(c)$ for the language constants, and
    - Given $P(r)$ for all immediate subterms of r of s
    - we can show $P(s)$
  - We may conclude $\forall s \in \mathcal{T} \mid P(s)$

These methods of induction are equivalent to each other, but using one or the other can *simplify our proofs*.

- Formally, these three forms of induction are interderivable.
- As a matter of style, we will often use structural induction:
  1. Because it is a bit more intuitive.
  2. To avoid having to detour into numbers.

Semantics Styles

Once we have 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

Meaning of a term is a *mathematical object*.

- **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:

- writing succ(succ(succ0)) as 3.
  - $\mathbb{N}$ is the semantic domain.
  - The interpretation function maps terms of UAE to $\mathbb{N}$.
    - ⋆ How? By counting the number of succ invokations.

Axiomatic Semantics

- **Axiomatic Semantics** give laws of behaviour of terms.
- 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**.

Operational Semantics of Booleans

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 $t \longrightarrow t'$

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

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

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

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.
- Demonstrate the derivability of a given pair using **formal derivation**.

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:

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

(See evaluation on board)

Example Evaluation 2


```
if (if true then false else true) then true else false   (6)
```


$$t = \texttt{if true then false else true} \quad (7)$$

(rest also on board)

Induction on Derivations

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

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

[Induction on Derivations]

- 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.

Determinacy

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.

By induction on the derivation of $t \to t'$.

Normal Form

Language designers care a lot about *how* an expression is evaluated.
Programmers care more about *what* it evaluates to.

- A term t is in **Normal Form** if no evaluation rule applies to it.
- For untyped booleans, the normal form of all terms is `true` or `false`.

THEOREM : [Every Value is in Normal Form]
`true` and `false` are in normal form.

Become highly non-trivial later!

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

THEOREM :

[If t is in Normal Form, t is a Value]

Proof:

- Suppose t is in Normal Form but *not* a value.
- IH: Subterms of t that are not a value are not in normal form.
- t not a value, must be of form

$$\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \tag{9}$$

- Whether it can be evaluated depends on what t_1 is. There are three possibilities:

  - t_1 is `true`. E-IfTrue applies.
  - Similarly for $t_1 =$ `false` and E-IfFalse.
  - Thus t_1 is not a value, and by IH, not in normal form. But this means E-If applies, to t is not normal.

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

- evaluation **terminates** when it reaches some normal form.
- in a finite number of steps,
- without repeating "states" (which would cause non-termination).

But what about the the **halting problem** ? Rough answer: it's easy to use types to keep it at bay.

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

multi-step evalution relation $\rightarrow^*$:

- the **reflexive, transitive closure** of one-step evaluation.

In other words, the smallest relation such that:

$$t \rightarrow t' \implies t \rightarrow^* t' \tag{10}$$

$$\forall t \in \mathcal{T} \mid t \rightarrow^* t \tag{11}$$

$$t \rightarrow^* t' \wedge t' \rightarrow^* t'' \implies t \rightarrow^* t'' \tag{12}$$

i.e. contains $\rightarrow$, is reflexive and is transitive.

Uniqueness of Normal Form

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

$$t \twoheadrightarrow^* u \wedge t \twoheadrightarrow^* u' \implies u = u' \tag{13}$$

Proof idea:

- single-step is unique
- can't have multiple normal forms in a path of $\twoheadrightarrow^*$ (by definition, normal forms don't evaluate).

Termination of Evaluation

THEOREM : [Termination of Evaluation] (for Booleans)

$$\forall t \in \mathcal{T}\ \exists t' \in \mathcal{N} | t \rightarrow^* t' \tag{14}$$

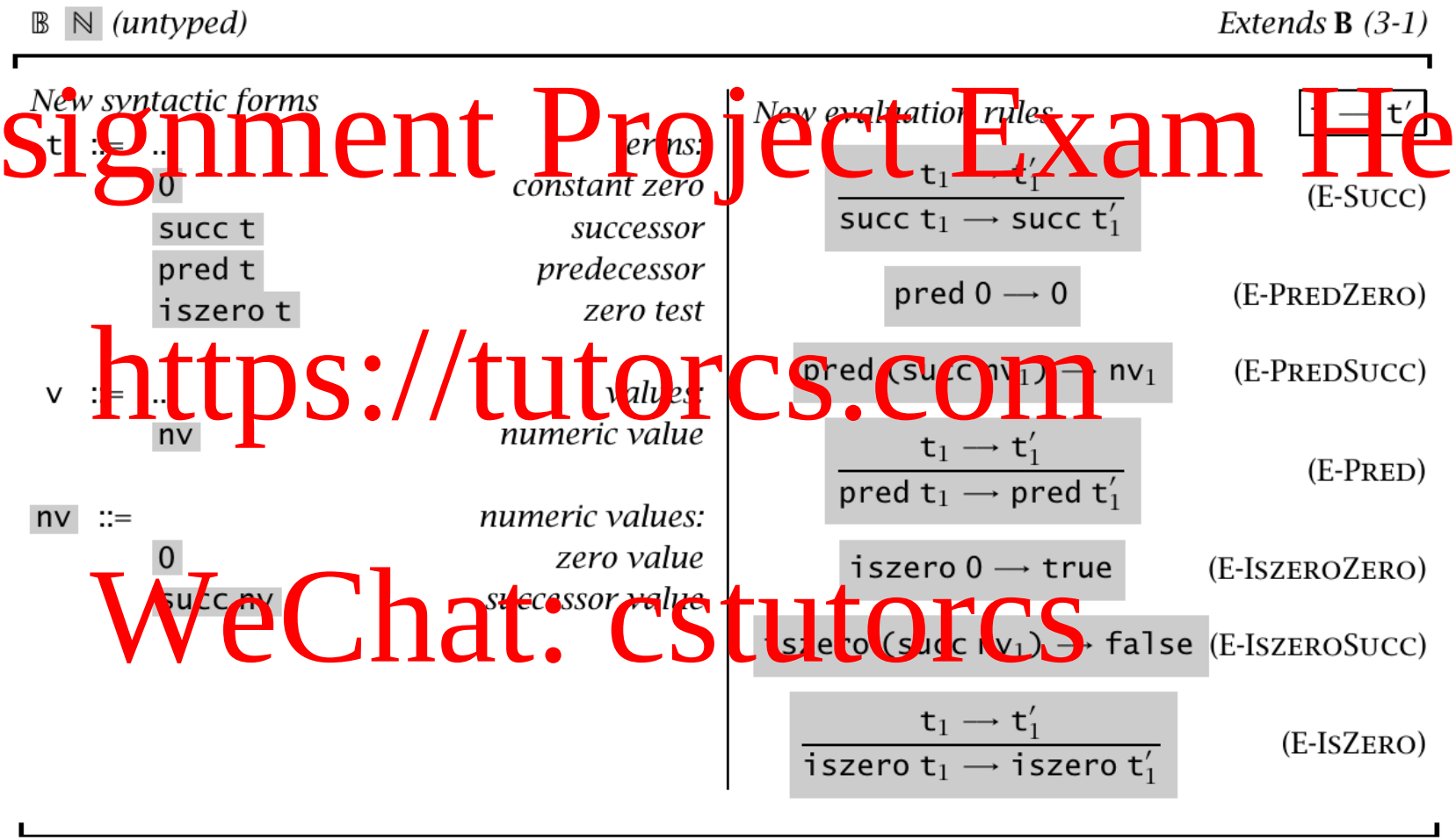
Where $\mathcal{N}$ is the set of normal forms of $\mathcal{T}$.

Useful: **LEMMA :**

$$s \rightarrow s' \implies size(s) > size(s'). \tag{15}$$

So single-step evaluation reduces size, and there are no infinite descending chains of naturals.

Extended UAE Semantics



When a term is in normal form but not a value, it is **stuck**.

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...

$$\texttt{iszero}\ (\texttt{succ}\ \mathsf{nv}_1) \longrightarrow \texttt{false} \quad \text{(E-IsZeroSucc)}$$

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

To Resolve Ambiguity!

E-IsZero!

- We can't use E-IsZeroSucc, because `succ pred succ 0` is not a value.
- 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...

$$t \to t' \wedge t \to t'' \text{ no longer implies } t' = t'' \tag{16}$$

This is bad language design, even if multi-step semantics might still turn out fine.