

# CSE 114A: Fall 2021

## Foundations of Programming Languages

### *Formalizing Nano*

Owen Arden  
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

---

## Formalizing Nano

Goal: we want to guarantee properties about programs, such as:

- evaluation is deterministic
- all programs terminate
- certain programs never fail at run time
- etc.

To prove theorems about programs we first need to define formally

- their *syntax* (what programs look like)
- their *semantics* (what it means to run a program)

Let's start with Nano1 (Nano w/o functions) and prove some stuff!

2

---

## Nano1: Syntax

We need to define the syntax for *expressions* (*terms*) and *values* using a grammar:

```
e ::= n | x           -- expressions
    | e1 + e2
    | let x = e1 in e2
```

```
v ::= n               -- values
```

where  $n \in \mathbb{N}$ ,  $x \in \text{Var}$

3

## Nano1: Operational Semantics

Operational semantics defines how to execute a program step by step

Let's define a *step relation* (reduction relation)  $e \Rightarrow e'$

- “expression  $e$  makes a step (reduces in one step) to an expression  $e'$ ”

4

## Nano1: Operational Semantics

We define the step relation *inductively* through a set of *rules*:

```
[Add-L]      e1 => e1'      -- premise
             -----
e1 + e2 => e1' + e2      -- conclusion

[Add-R]      e2 => e2'
             -----
n1 + e2 => n1 + e2'

[Add]        n1 + n2 => n      where n == n1 + n2

[Let-Def]     e1 => e1'
             -----
let x = e1 in e2 => let x = e1' in e2

[Let]        let x = v in e2 => e2[x := v]
```

5

## Nano1: Operational Semantics

Here  $e[x := v]$  is a value substitution:

```
x[x := v]      = v
y[x := v]      = y      -- assuming x != y
n[x := v]      = n
(e1 + e2)[x := v] = e1[x := v] + e2[x := v]
(let x = e1 in e2)[x := v] = let x = e1[x := v] in e2
(let y = e1 in e2)[x := v] = let y = e1[x := v] in
e2[x := v]
```

Do not have to worry about capture, because  $v$  is a value (has no free variables!)

6

## Nano1: Operational Semantics

A reduction is *valid* if we can build its **derivation** by “stacking” the rules:

```
[Add] -----
      1 + 2 => 3
[Add-L] -----
      (1 + 2) + 5  =>  3 + 5
```

Do we have rules for all kinds of expressions?

7

## Nano1: Operational Semantics

We define the step relation *inductively* through a set of *rules*:

```
          e1 => e1'      -- premise
[Add-L]  -----
      e1 + e2 => e1' + e2  -- conclusion

          e2 => e2'
[Add-R]  -----
      n1 + e2 => n1 + e2'

[Add]    n1 + n2 => n      where n == n1 + n2

          e1 => e1'
[Let-Def] -----
      let x = e1 in e2 => let x = e1' in e2

[Let]    let x = v in e2 => e2[x := v]
```

8

## 1. Normal forms

There are no reduction rules for:

- $n$
- $x$

Both of these expressions are *normal forms* (cannot be further reduced), however:

- $n$  is a *value*
  - intuitively, corresponds to successful evaluation
- $x$  is *not* a value
  - intuitively, corresponds to a run-time error!
  - we say the program  $x$  is **stuck**

9

## 2. Evaluation order

In  $e1 + e2$ , which side should we evaluate first?

In other words, which one of these reductions is valid (or both)?

1.  $(1 + 2) + (4 + 5) \Rightarrow 3 + (4 + 5)$
2.  $(1 + 2) + (4 + 5) \Rightarrow (1 + 2) + 9$

Reduction (1) is *valid* because we can build a **derivation** using the rules:

[Add] -----  
           $1 + 2 \Rightarrow 3$   
[Add-L] -----  
           $(1 + 2) + (4 + 5) \Rightarrow 3 + (4 + 5)$

Reduction (2) is *invalid* because we cannot build a derivation:

- there is *no rule* whose conclusion matches this reduction!

$???$   
[???] -----  
           $(1 + 2) + (4 + 5) \Rightarrow (1 + 2) + 9$

10

## Evaluation relation

Like in  $\lambda$ -calculus, we define the **multi-step reduction** relation  $e \Rightarrow^* e'$ :

$e \Rightarrow^* e'$  iff there exists a sequence of expressions  $e_1, \dots, e_n$  such that

- $e = e_1$
- $e_n = e'$
- $e_i \Rightarrow e_{i+1}$  for each  $i$  in  $[0..n)$

Example:

$(1 + 2) + (4 + 5)$   
 $\Rightarrow^* 3 + 9$

because

$(1 + 2) + (4 + 5)$   
 $\Rightarrow 3 + (4 + 5)$   
 $\Rightarrow 3 + 9$

11

## Evaluation relation

Now we define the **evaluation relation**  $e \rightsquigarrow e'$ :

$e \rightsquigarrow e'$  iff

- $e \Rightarrow^* e'$
- $e'$  is in normal form

Example:

$(1 + 2) + (4 + 5)$   
 $\rightsquigarrow 12$

because

$(1 + 2) + (4 + 5)$   
 $\Rightarrow 3 + (4 + 5)$   
 $\Rightarrow 3 + 9$   
 $\Rightarrow 12$

and  $12$  is a *value* (normal form)

12

# Theorems about Nano1

---

Let's prove something about Nano1!

1. Every Nano1 program terminates
2. Closed Nano1 programs don't get stuck
3. *Corollary (1 + 2)*: Every closed Nano1 program evaluates to a value

How do we prove theorems about languages?

By induction.

13

---

## Mathematical induction in PL

---

14

---

## 1. Induction on natural numbers

---

To prove  $\forall n. P(n)$  we need to prove:

- *Base case*:  $P(0)$
- *Inductive case*:  $P(n + 1)$  assuming the *induction hypothesis* (IH): that  $P(n)$  holds

Compare with inductive definition for natural numbers:

```
data Nat = Zero      -- base case
         | Succ Nat -- inductive case
```

No reason why this would only work for natural numbers...

In fact we can do induction on *any* inductively defined mathematical object (= any datatype)!

- lists
- trees
- programs (terms)
- etc

15

## 2. Induction on terms

$e ::= n \mid x$   
 $\mid e1 + e2$   
 $\mid \text{let } x = e1 \text{ in } e2$

To prove  $\forall e. P(e)$  we need to prove:

- *Base case 1:*  $P(n)$
- *Base case 2:*  $P(x)$
- *Inductive case 1:*  $P(e1 + e2)$  assuming the IH:  
that  $P(e1)$  and  $P(e2)$  hold
- *Inductive case 2:*  $P(\text{let } x = e1 \text{ in } e2)$  assuming the IH:  
that  $P(e1)$  and  $P(e2)$  hold

16

## 3. Induction on derivations

Our reduction relation  $\Rightarrow$  is also defined *inductively*!

- Axioms are base cases
- Rules with premises are inductive cases

To prove  $\forall e, e'. P(e \Rightarrow e')$  we need to prove:

- *Base cases:*  $[Add]$ ,  $[Let]$
- *Inductive cases:*  $[Add-L]$ ,  $[Add-R]$ ,  $[Let-Def]$  assuming the IH:  
that  $P$  holds of their premise

17

## Theorem: Termination

**Theorem I [Termination]:** For any expression  $e$  there exists  $e'$  such  
that  $e \Rightarrow^* e'$ .

Proof idea: let's define the *size* of an expression such that

- size of each expression is positive
- each reduction step strictly decreases the size

Then the length of the execution sequence for  $e$  is *bounded* by the size of  $e$ !

size  $n$  = ???  
size  $x$  = ???  
size  $(e1 + e1)$  = ???  
size  $(\text{let } x = e1 \text{ in } e2)$  = ???

18

## Theorem: Termination

Term size:

```
size n           = 1
size x           = 1
size (e1 + e2)   = size e1 + size e2
size (let x = e1 in e2) = size e1 + size e2
```

Lemma 1: For any  $e$ ,  $\text{size } e > 0$ .

Proof: By induction on the term  $e$ .

- Base case 1:  $\text{size } n = 1 > 0$
  - Base case 2:  $\text{size } x = 1 > 0$
  - Inductive case 1:  $\text{size } (e1 + e2) = \text{size } e1 + \text{size } e2 > 0$  because  $\text{size } e1 > 0$  and  $\text{size } e2 > 0$  by IH.
  - Inductive case 2: similar.
- QED.

19

## Theorem: Termination

Lemma 2: For any  $e, e'$  such that  $e \Rightarrow e'$ ,  $\text{size } e' < \text{size } e$ .

Proof: By induction on the derivation of  $e \Rightarrow e'$ .

Base case [Add].

- Given: the root of the derivation is  
[Add]:  $n1 + n2 \Rightarrow n$  where  $n = n1 + n2$
- To prove:  $\text{size } n < \text{size } (n1 + n2)$
- $\text{size } n = 1 < 2 = \text{size } (n1 + n2)$

20

## Theorem: Termination

Lemma 2: For any  $e, e'$  such that  $e \Rightarrow e'$ ,  $\text{size } e' < \text{size } e$ .

Inductive case [Add-L].

- Given: the root of the derivation is [Add-L]:

$e1 \Rightarrow e1'$

-----  
 $e1 + e2 \Rightarrow e1' + e2$

- To prove:  $\text{size } (e1' + e2) < \text{size } (e1 + e2)$
- IH:  $\text{size } e1' < \text{size } e1$

```
size (e1' + e2)
= -- def. size
  size e1' + size e2
< -- IH
  size e1 + size e2
= -- def. size
  size (e1 + e2)
```

Inductive case [Add-R]. Try at home

21

## Theorem: Termination

Lemma 2: For any  $e, e'$  such that  $e \Rightarrow e'$ ,  $\text{size } e' < \text{size } e$ .

Base case [Let].

- Given: the root of the derivation  
is [Let]:  $\text{let } x = v \text{ in } e2 \Rightarrow e2[x := v]$
- To prove:  $\text{size } (e2[x := v]) < \text{size } (\text{let } x = v \text{ in } e2)$

```
size (e2[x := v])
= -- auxiliary lemma!
  size e2
< -- lemma
  size v + size e2
= -- def. size
  size (let x = v in e2)
```

Inductive case [Let-Def]. Try at home

QED.

22

## Nano2: adding functions

23

## Syntax

We need to extend the syntax of expressions and values:

```
e ::= n | x           -- expressions
    | e1 + e2
    | let x = e1 in e2
    | \x -> e         -- abstraction
    | e1 e2           -- application

v ::= n               -- values
    | \x -> e         -- abstraction
```

24



## Operational semantics

We need to extend our reduction relation with rules for abstraction and application:

$$\text{[App-L]} \quad \frac{e1 \Rightarrow e1'}{e1 \ e2 \Rightarrow e1' \ e2}$$

$$\text{[App-R]} \quad \frac{e \Rightarrow e'}{v \ e \Rightarrow v \ e'}$$

$$\text{[App]} \quad (\lambda x \rightarrow e) \ v \Rightarrow e[x := v]$$

25

## Evaluation Order

$$\begin{aligned} & ((\lambda x \ y \rightarrow x + y) \ 1) \ (1 + 2) \\ \Rightarrow & (\lambda y \rightarrow 1 + y) \ (1 + 2) & \text{-- [App-L], [App]} \\ \Rightarrow & (\lambda y \rightarrow 1 + y) \ 3 & \text{-- [App-R], [Add]} \\ \Rightarrow & 1 + 3 & \text{-- [App]} \\ \Rightarrow & 4 & \text{-- [Add]} \end{aligned}$$

Our rules define **call-by-value**:

1. Evaluate the function (to a lambda)
2. Evaluate the argument (to some value)
3. “Make the call”: make a substitution of formal to actual in the body of the lambda

The alternative is **call-by-name**:

- do not evaluate the argument before “making the call”
- can we modify the application rules for Nano2 to make it call-by-name?

26

## Theorems about Nano2

Let’s prove something about Nano2!

1. Every Nano2 program terminates (?)
2. Closed Nano2 programs don’t get stuck (?)

27

## Theorems about Nano2

---

1. Every Nano2 program terminates (?)

What about  $(\lambda x \rightarrow x \ x) \ (\lambda x \rightarrow x \ x)$ ?

2. Closed Nano2 programs don't get stuck (?)

What about  $1 \ 2$ ?

Both theorems are now false!

To recover these properties, we need to add *types*:

1. Every *well-typed* Nano2 program terminates
2. *Well-typed* Nano2 programs don't get stuck