

CSE 114A: Fall 2023

**Foundations of Programming
Languages**

Formalizing Nano

Owen Arden

UC Santa Cruz

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!

Nano1: Syntax

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

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

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

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

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' ”

Nano1: Operational Semantics

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

[Add-L]
$$\frac{e1 \Rightarrow e1'}{\quad} \quad \text{-- premise}$$
$$e1 + e2 \Rightarrow e1' + e2 \quad \text{-- conclusion}$$

[Add-R]
$$\frac{e2 \Rightarrow e2'}{\quad}$$
$$n1 + e2 \Rightarrow n1 + e2'$$

[Add]
$$n1 + n2 \Rightarrow n \quad \text{where } n == n1 + n2$$

[Let-Def]
$$\frac{e1 \Rightarrow e1'}{\quad}$$
$$\text{let } x = e1 \text{ in } e2 \Rightarrow \text{let } x = e1' \text{ in } e2$$

[Let]
$$\text{let } x = v \text{ in } e2 \Rightarrow e2[x := v]$$

Nano1: Operational Semantics

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

$$\begin{aligned}x[x := v] &= v \\y[x := v] &= y \quad \text{-- assuming } x \neq y \\n[x := v] &= n \\(e1 + e2)[x := v] &= e1[x := v] + e2[x := v] \\(\mathbf{let} \ x = e1 \ \mathbf{in} \ e2)[x := v] &= \mathbf{let} \ x = e1[x := v] \ \mathbf{in} \ e2 \\(\mathbf{let} \ y = e1 \ \mathbf{in} \ e2)[x := v] &= \mathbf{let} \ y = e1[x := v] \ \mathbf{in} \\e2[x := v]\end{aligned}$$

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

Nano1: Operational Semantics

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

$$\begin{array}{c} \text{[Add]} \text{ -----} \\ 1 + 2 \Rightarrow 3 \end{array}$$

$$\begin{array}{c} \text{[Add-L]} \text{ -----} \\ (1 + 2) + 5 \Rightarrow 3 + 5 \end{array}$$

Do we have rules for all kinds of expressions?

Nano1: Operational Semantics

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

[Add-L]
$$\frac{e1 \Rightarrow e1'}{\quad} \quad \text{-- premise}$$
$$e1 + e2 \Rightarrow e1' + e2 \quad \text{-- conclusion}$$

[Add-R]
$$\frac{e2 \Rightarrow e2'}{\quad}$$
$$n1 + e2 \Rightarrow n1 + e2'$$

[Add]
$$n1 + n2 \Rightarrow n \quad \text{where } n == n1 + n2$$

[Let-Def]
$$\frac{e1 \Rightarrow e1'}{\quad}$$
$$\text{let } x = e1 \text{ in } e2 \Rightarrow \text{let } x = e1' \text{ in } e2$$

[Let]
$$\text{let } x = v \text{ in } e2 \Rightarrow e2[x := v]$$

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**

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:

$$\begin{array}{c} [\text{Add}] \text{ -----} \\ 1 + 2 \Rightarrow 3 \end{array}$$

$$\begin{array}{c} [\text{Add-L}] \text{ -----} \\ (1 + 2) + (4 + 5) \Rightarrow 3 + (4 + 5) \end{array}$$

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

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

$$\begin{array}{c} \text{???} \\ [???] \text{ -----} \\ (1 + 2) + (4 + 5) \Rightarrow (1 + 2) + 9 \end{array}$$

QUIZ

If these are the only rules for let bindings, which reductions are valid? *

$e1 \Rightarrow e1'$

[Let-Def] -----
 $\text{let } x = e1 \text{ in } e2 \Rightarrow \text{let } x = e1' \text{ in } e2$

[Let] $\text{let } x = v \text{ in } e2 \Rightarrow e2[x := v]$

- (A) $(\text{let } x = 1 + 2 \text{ in } 4 + 5 + x) \Rightarrow (\text{let } x = 3 \text{ in } 4 + 5 + x)$
- (B) $(\text{let } x = 1 + 2 \text{ in } 4 + 5 + x) \Rightarrow (\text{let } x = 1 + 2 \text{ in } 9 + x)$
- (C) $(\text{let } x = 1 + 2 \text{ in } 4 + 5 + x) \Rightarrow (4 + 5 + 1 + 2)$
- (D) A and B
- (E) All of the above



<http://tiny.cc/cse116-reduce-ind>

QUIZ

If these are the only rules for let bindings, which reductions are valid? *

$e1 \Rightarrow e1'$

[Let-Def] -----
 $\text{let } x = e1 \text{ in } e2 \Rightarrow \text{let } x = e1' \text{ in } e2$

[Let] $\text{let } x = v \text{ in } e2 \Rightarrow e2[x := v]$

- (A) $(\text{let } x = 1 + 2 \text{ in } 4 + 5 + x) \Rightarrow (\text{let } x = 3 \text{ in } 4 + 5 + x)$
- (B) $(\text{let } x = 1 + 2 \text{ in } 4 + 5 + x) \Rightarrow (\text{let } x = 1 + 2 \text{ in } 9 + x)$
- (C) $(\text{let } x = 1 + 2 \text{ in } 4 + 5 + x) \Rightarrow (4 + 5 + 1 + 2)$
- (D) A and B
- (E) All of the above



<http://tiny.cc/cse116-reduce-grp>

Evaluation relation

Like in λ -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

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

Evaluation relation

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

$e \rightsquigarrow e'$ iff

- $e \xrightarrow{*} e'$
- 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)

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.

Mathematical induction in PL

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

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

3. Induction on derivations

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

- Axioms are bases 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

Theorem: Termination

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

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 $(\mathbf{let} \ x = e1 \ \mathbf{in} \ e2)$ = ???

Theorem: Termination

Term size:

$$\text{size } n = 1$$

$$\text{size } x = 1$$

$$\text{size } (e1 + e2) = \text{size } e1 + \text{size } e2$$

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

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.

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: $\text{size } n < \text{size } (n1 + n2)$
- $\text{size } n = 1 < 2 = \text{size } (n1 + n2)$

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: $\text{size } e1' < \text{size } e1$

$$\begin{aligned} & \text{size } (e1' + e2) \\ = & \text{-- def. size} \\ & \text{size } e1' + \text{size } e2 \\ < & \text{-- IH} \\ & \text{size } e1 + \text{size } e2 \\ = & \text{-- def. size} \\ & \text{size } (e1 + e2) \end{aligned}$$

Inductive case [Add-R]. Try at home

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.

QUIZ

What is the IH for the inductive case [Let-Def]? *

$e1 \Rightarrow e1'$

[Let-Def] -----
 $\text{let } x = e1 \text{ in } e2 \Rightarrow \text{let } x = e1' \text{ in } e2$

- (A) $e1 \Rightarrow e1'$
- (B) $\text{size } e1' < \text{size } e1$
- (C) $\text{size } (\text{let } x = e1 \text{ in } e2) < \text{size } (\text{let } x = e1' \text{ in } e2)$



<http://tiny.cc/cse116-induct-ind>

QUIZ

What is the IH for the inductive case [Let-Def]? *

$e1 \Rightarrow e1'$

[Let-Def] -----
 $\text{let } x = e1 \text{ in } e2 \Rightarrow \text{let } x = e1' \text{ in } e2$

- (A) $e1 \Rightarrow e1'$
- (B) $\text{size } e1' < \text{size } e1$
- (C) $\text{size } (\text{let } x = e1 \text{ in } e2) < \text{size } (\text{let } x = e1' \text{ in } e2)$



<http://tiny.cc/cse116-induct-grp>

Nano2: adding functions

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
```

Operational semantics

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

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

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

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

QUIZ

With rules defined above, which reductions are valid? *

- (A) $(\lambda x y \rightarrow x + y) 1 (1 + 2) \Rightarrow (\lambda x y \rightarrow x + y) 1 3$
- (B) $(\lambda x y \rightarrow x + y) 1 (1 + 2) \Rightarrow (\lambda y \rightarrow 1 + y) (1 + 2)$
- (C) $(\lambda y \rightarrow 1 + y) (1 + 2) \Rightarrow (\lambda y \rightarrow 1 + y) 3$
- (D) $(\lambda y \rightarrow 1 + y) (1 + 2) \Rightarrow 1 + 1 + 2$
- (E) B and C



<http://tiny.cc/cse116-reduce2-ind>

QUIZ

With rules defined above, which reductions are valid? *

- (A) $(\lambda x y \rightarrow x + y) 1 (1 + 2) \Rightarrow (\lambda x y \rightarrow x + y) 1 3$
- (B) $(\lambda x y \rightarrow x + y) 1 (1 + 2) \Rightarrow (\lambda y \rightarrow 1 + y) (1 + 2)$
- (C) $(\lambda y \rightarrow 1 + y) (1 + 2) \Rightarrow (\lambda y \rightarrow 1 + y) 3$
- (D) $(\lambda y \rightarrow 1 + y) (1 + 2) \Rightarrow 1 + 1 + 2$
- (E) B and C



<http://tiny.cc/cse116-reduce2-grp>

Evaluation Order

$((\lambda x y \rightarrow x + y) 1) (1 + 2)$
 $\Rightarrow (\lambda y \rightarrow 1 + y) (1 + 2)$ -- *[App-L], [App]*
 $\Rightarrow (\lambda y \rightarrow 1 + y) 3$ -- *[App-R], [Add]*
 $\Rightarrow 1 + 3$ -- *[App]*
 $\Rightarrow 4$ -- *[Add]*

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?

Theorems about Nano2

Let's prove something about Nano2!

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

QUIZ

Let's prove something about Nano2!

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

Are these theorems still true? *

- (A) Both true
- (B) 1 is true, 2 is false
- (C) 1 is false, 2 is true
- (D) Both false



<http://tiny.cc/cse116-nano2-ind>

QUIZ

Let's prove something about Nano2!

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

Are these theorems still true? *

- (A) Both true
- (B) 1 is true, 2 is false
- (C) 1 is false, 2 is true
- (D) Both false



<http://tiny.cc/cse116-nano2-grp>

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