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 ::= n | x* (expressions)
  - e1 + e2
  - *let x = e1 in e2*

- *v ::= n* (values)

where *n ∈ ℕ, x ∈ 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 \( e' \)”

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

\[
\begin{align*}
e1 & \Rightarrow e1' \quad \text{-- premise} \\
\text{[Add-L]} \quad e1 + e2 & \Rightarrow e1' + e2' \quad \text{-- conclusion} \\
\text{[Add-R]} \quad n1 + e2 & \Rightarrow n1 + e2' \\
\text{[Add]} \quad n1 + n2 & \Rightarrow n \quad \text{where } n = n1 + n2 \\
\text{[Let-Def]} \quad \text{let } x = e1 \text{ in } e2 & \Rightarrow \text{let } x = e1' \text{ in } e2 \\
\text{[Let]} \quad \text{let } x = v \text{ in } e2 & \Rightarrow e2[x := v]
\end{align*}
\]

Here \( e[x := v] \) is a value substitution:

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

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:

\[
\begin{align*}
\text{Add} & : \quad 1 + 2 \Rightarrow 3 \\
\text{Add-L} & : \quad (1 + 2) + 5 \Rightarrow 3 + 5
\end{align*}
\]

Do we have rules for all kinds of expressions?

Nano1: Operational Semantics

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

\[
\begin{align*}
\text{e1} & \Rightarrow \text{e1'} \quad \text{-- premise} \\
\text{Add-L} & : \quad \text{e1} + \text{e2} \Rightarrow \text{e1'} + \text{e2'} \quad \text{-- conclusion} \\
\text{e2} & \Rightarrow \text{e2'} \\
\text{Add-R} & : \quad \text{n1} + \text{e2} \Rightarrow \text{n1} + \text{e2'} \\
\text{Add} & : \quad \text{n1} + \text{n2} \Rightarrow \text{n} \quad \text{where} \quad \text{n} = \text{n1} + \text{n2} \\
\text{e1} & \Rightarrow \text{e1'} \\
\text{Let-Def} & : \quad \text{let} \ x = \text{e1} \ \text{in} \ \text{e2} \Rightarrow \text{let} \ x = \text{e1'} \ \text{in} \ \text{e2} \\
\text{Let} & : \quad \text{let} \ x = \text{v} \ \text{in} \ \text{e2} \Rightarrow \text{e2}[x := \text{v}]
\end{align*}
\]

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 \( e_1 + e_2 \), 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:

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

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

- there is no rule whose conclusion matches this reduction!

\[
\text{???}
\]

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

QUIZ

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

\[
e_1 \Rightarrow e_1'
\]

\[
\text{[Let-Def]} \quad \begin{array}{c}
\text{let } x = e_1 \text{ in } e_2 \Rightarrow \text{let } x = e_1' \text{ in } e_2 \\
\text{[Let]} \quad \begin{array}{c}
\text{let } x = v \text{ in } e_2 \Rightarrow e_2[x := v]
\end{array}
\end{array}
\]

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

http://tiny.cc/cmps112-reduce-ind

QUIZ

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

\[
e_1 \Rightarrow e_1'
\]

\[
\text{[Let-Def]} \quad \begin{array}{c}
\text{let } x = e_1 \text{ in } e_2 \Rightarrow \text{let } x = e_1' \text{ in } e_2 \\
\text{[Let]} \quad \begin{array}{c}
\text{let } x = v \text{ in } e_2 \Rightarrow e_2[x := v]
\end{array}
\end{array}
\]

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

http://tiny.cc/cmps112-reduce-grp
Evaluation relation

Like in \( \lambda \)-calculus, we define the multi-step reduction relation \( e \Rightarrow e' \):

\[ e \Rightarrow e' \quad \text{iff there exists a sequence of expressions } e_1, \ldots, e_n \text{ such that} \]

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

Example:

\[(1 + 2) + (4 + 5)\]
\[\Rightarrow 3 + 9\]

because

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

Evaluation relation

Now we define the evaluation relation \( e \Rightarrow* e' \):

\[ e \Rightarrow* e' \quad \text{iff} \]

- \( e \Rightarrow e' \)
- \( e' \) is in normal form

Example:

\[(1 + 2) + (4 + 5)\]
\[\Rightarrow* 12\]

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.
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  |  x
      |  e1 + e2
      |  let x = e1 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: $P(let x = e1 in e2)$ assuming the IH: that $P(e1)$ and $P(e2)$ hold
3. Induction on derivations

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

- Axioms are 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

Theorem: Termination

**Theorem 1 [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 tables:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>???</td>
</tr>
<tr>
<td>$x$</td>
<td>???</td>
</tr>
<tr>
<td>$e_1 + e_2$</td>
<td>???</td>
</tr>
<tr>
<td>$\text{let } x = e_1 \text{ in } e_2$</td>
<td>???</td>
</tr>
</tbody>
</table>

Lemma 1: For any $e$, size $e > 0$.

Proof: By induction on the term $e$.

- Base case 1: size $n = 1 > 0$
- Base case 2: size $x = 1 > 0$
- Inductive case 1: size $e_1 + e_2 = \text{size } e_1 + \text{size } e_2 > 0$ because size $e_1 > 0$ and size $e_2 > 0$ 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]: $n_1 + n_2 \Rightarrow n$ where $n = n_1 + n_2$
- To prove: $\text{size } n < \text{size } (n_1 + n_2)$
- $\text{size } n = 1 < 2 = \text{size } (n_1 + n_2)$

Inductive case [Add-L].

- Given: the root of the derivation is [Add-L]:
  $e_1 \Rightarrow e_1'$
- $e_1 + e_2 \Rightarrow e_1' + e_2$
- To prove: $\text{size } (e_1' + e_2) < \text{size } (e_1 + e_2)$
- IH: $\text{size } e_1' < \text{size } e_1$
  - $\text{size } (e_1' + e_2) = \text{--- def. size } e_1'$
  - $\text{size } e_1' + \text{size } e_2$
  - $\text{--- IH }$ $\text{size } e_1 + \text{size } e_2 = \text{--- def. size } e_1 + e_2$
  - $\text{--- IH }$ $\text{size } e_1 + e_2$

Inductive case [Add-R]. Try at home

Base case [Let].

- Given: the root of the derivation
  is [Let]: $\text{let } x = v \text{ in } e_2 \Rightarrow e_2[x := v]$
- To prove: $\text{size } (e_2[x := v]) < \text{size } (\text{let } x = v \text{ in } e_2)$

  $\text{size } (e_2[x := v]) = \text{--- auxiliary lemma } e_2$
  - $\text{--- IH }$ $\text{size } v + \text{size } e_2$
  - $\text{--- def. size } e_2$

Inductive case [Let-Def]. Try at home

QED.
QUIZ

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

\[ e_1 \Rightarrow e_1' \]

[Let-Def] ...........................

\[ \text{let } x = e_1 \text{ in } e_2 \Rightarrow \text{let } x = e_1' \text{ in } e_2 \]

- (A) \( e_1 \Rightarrow e_1' \)
- (B) size \( e_1' \leq \text{size } e_1 \)
- (C) size (let \( x = e_1 \text{ in } e_2 \)) \leq \text{size (let } x = e_1' \text{ in } e_2 \)

http://tiny.cc/cmps112-induct-ind

QUIZ

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

\[ e_1 \Rightarrow e_1' \]

[Let-Def] ...........................

\[ \text{let } x = e_1 \text{ in } e_2 \Rightarrow \text{let } x = e_1' \text{ in } e_2 \]

- (A) \( e_1 \Rightarrow e_1' \)
- (B) size \( e_1' \leq \text{size } e_1 \)
- (C) size (let \( x = e_1 \text{ in } e_2 \)) \leq \text{size (let } x = e_1' \text{ in } e_2 \)

http://tiny.cc/cmps112-induct-grp

Nano2: adding functions
### Syntax

We need to extend the syntax of expressions and values:

\[
\begin{align*}
  e & ::= n \mid x \quad \text{-- expressions} \\
  & \quad \mid e_1 + e_2 \\
  & \quad \mid \text{let } x = e_1 \text{ in } e_2 \\
  & \quad \mid \lambda x \to e \quad \text{-- abstraction} \\
  & \quad \mid e_1 e_2 \quad \text{-- application}
\end{align*}
\]

\[
\begin{align*}
  v & ::= n \quad \text{-- values} \\
  & \quad \mid \lambda x \to e \quad \text{-- abstraction}
\end{align*}
\]

### Operational semantics

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

- \( e_1 \Rightarrow e_1' \) [App-L]
- \( e_1 e_2 \Rightarrow e_1' e_2 \) [App]
- \( e \Rightarrow e' \) [App-R]
- \( v e \Rightarrow v e' \) [App]
- \( (\lambda x \to e) v \Rightarrow e[x := v] \) [App]

### QUIZ

With rules defined above, which reductions are valid? *

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

http://tiny.cc/cmps112-reduce2-ind
## Evaluation Order

\[
((\lambda x \ y \to \ x + y) \ 1 \ 2) \\
\Rightarrow \ (\lambda y \to \ 1 + y) \ (1 + 2) \ \ \ \ \ \ \ \ \ \ [\text{App-L}], [\text{App}] \\
\Rightarrow \ (\lambda y \to \ 1 + y) \ 3 \ \ \ \ \ \ \ \ \ \ [\text{App-R}], [\text{Add}] \\
\Rightarrow \ 1 + 3 \ \ \ \ \ \ \ \ \ \ [\text{App}] \\
\Rightarrow \ 4 \ \ \ \ \ \ \ \ \ \ [\text{Add}]
\]

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 (?)
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

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

Theorems about Nano2

1. Every Nano2 program terminates (?)
   
   What about \( \lambda x \rightarrow x \rightarrow x \)?

2. Closed Nano2 programs don't get stuck (?)
   
   What about \( \lambda x \rightarrow x \rightarrow x \) and 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