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!

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

Nano1: Operational Semantics

Operational semantics defines how to execute a program step by step

Let's define a *step relation* (*reduction relation*) `e => 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*:

```
             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]
```


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!)

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?

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]
```


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) => 3 + (4 + 5)`
2. `(1 + 2) + (4 + 5) => (1 + 2) + 9`

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

```
          [Add] ----------
                1 + 2 => 3
[Add-L] ----------------------------------
        (1 + 2) + (4 + 5)  =>  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)  =>  (1 + 2) + 9
```


Evaluation relation

Like in λ-calculus, we define the **multi-step reduction** relation `e =*> e'`:

`e =*> e'` iff there exists a sequence of expressions `e1, ..., en` such that

- `e = e1`
- `en = e'`
- `ei => e(i+1)` for each `i in [0..n)`

Example:

```
    (1 + 2) + (4 + 5)
=*> 3 + 9
```

because

```
   (1 + 2) + (4 + 5)
=> 3       + (4 + 5)
=> 3       + 9
```


Evaluation relation

Now we define the **evaluation relation** `e =~> e'`:

`e =~> e'` iff

- `e =*> e'`
- `e'` is in normal form

Example:

```
    (1 + 2) + (4 + 5)
=~> 12
```

because

```
   (1 + 2) + (4 + 5)
=> 3       + (4 + 5)
=> 3       + 9
=> 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 | 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 `=>` 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 =~> 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 (let x = e1 in e2) = ???
```

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`, `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 (e1 + e2) = size e1 + size e2 > 0` because `size e1> 0` and `size e2 > 0` by IH.
- *Inductive case 2:* similar.

QED.

Theorem: Termination

Lemma 2: For any `e, e'` such that `e => e'`, `size e' < size e`.

Proof: By induction on the *derivation* of `e => e'`.

Base case `[Add]`.

- Given: the root of the derivation is

  `[Add]: n1 + n2 => n` where `n = n1 + n2`
- To prove: `size n < size (n1 + n2)`
- `size n = 1 < 2 = size (n1 + n2)`

Theorem: Termination

Lemma 2: For any `e, e'` such that `e => e'`, `size e' < size e`.

Inductive case `[Add-L]`.

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

```
     e1 => e1'
--------------------------
e1 + e2 => e1' + e2
```

- **To prove**: `size (e1' + e2) < size (e1 + e2)`
- **IH:** `size e1' < 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

Theorem: Termination

Lemma 2: For any `e, e'` such that `e => e'`, `size e' < size e`.

Base case `[Let]`.

- Given: the root of the derivation is `[Let]: let x = v in e2 => e2[x := v]`
- To prove: `size (e2[x := v]) < size (let x = v in e2)`

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

Inductive case `[Let-Def]`. Try at home

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:

```
           e1 => e1'
[App-L] ----------------
        e1 e2 => e1' e2

          e => e'
[App-R] ------------
        v e => v e'

[App]   (\x -> e) v => e[x := v]
```


Evaluation Order

```
   ((\x y -> x + y) 1) (1 + 2)
=> (\y -> 1 + y) (1 + 2)        -- [App-L], [App]
=> (\y -> 1 + y) 3              -- [App-R], [Add]
=> 1 + 3                        -- [App]
=> 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 (?)

Theorems about Nano2

1. Every Nano2 program terminates (?)

   What about `(\x -> x x) (\x -> 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