

CS 162 Programming languages

Lecture 7: Operational Semantics I

Yu Feng
Winter 2023

What does a program mean?

- We have learned how to specify syntax.
 - Example: `let x = lambda lambda` is not a valid λ^+ program
 - But we have not yet talked about what the meaning of a program is.
- First question: What is the meaning of a program in λ^+ ?
 - Answer: The value the program evaluates to
 - Example: `let x=3 in x` Value:3

How to specify meaning of programs

- Option 1: Don't worry too much
- Developer of language has some informal concept of the intended meaning, implement a compiler/interpreter that does whatever the language designers believe to be reasonable.
- Then, declare the meaning to be whatever the compiler produces
- A terrible idea

How to specify meaning of programs

- Why is this such a bad idea?
- This approach promotes bugs/inconsistencies to expected behavior.
- Hides specification of language in many implementation details
- Makes it almost impossible to implement another compiler that accepts the same language
- Unfortunately, this is (still) a very common approach
- Languages designed this way: C, C++ (to some extent), Perl, PHP, JavaScript, ...

How to specify meaning of programs

- Option 2: Try to write out precisely the meaning of each language construct in documentation, then follow this description in implementation
- Example: Describe the meaning of $e_1 + e_2$ in the λ^+ language:
- First attempt: “This evaluates to the sum of e_1 and e_2 ”
- What if e_1 or e_2 is not a number?
- Second attempt: “This evaluates to the sum if both e_1 and e_2 evaluate to numbers, and is stuck if either of them evaluate to a list”
- What if e is lambda?...

How to specify meaning of programs

- Written language is, by nature, ambiguous. It is very difficult to fully specify the meaning of all language constructs this way
- Easy to miss cases
- Results in long, complicated and difficult to understand specifications, but an improvement over no specification

Written specification in practice

- Let's look at the ISO C++ standard: page 34:

©ISO/IEC

N4582

² A declaration is a *definition* unless it declares a function without specifying the function's body (8.4), it contains the **extern** specifier (7.1.1) or a *linkage-specification*²⁵ (7.5) and neither an *initializer* nor a *function-body*, it declares a static data member in a class definition (9.2, 9.4), it is a class name declaration (9.1), it is an *opaque-enum-declaration* (7.2), it is a *template-parameter* (14.1), it is a *parameter-declaration* (8.3.5) in a function declarator that is not the *declarator* of a *function-definition*, or it is a **typedef** declaration (7.1.3), an *alias-declaration* (7.1.3), a *using-declaration* (7.3.3), a *static_assert-declaration* (Clause 7), an *attribute-declaration* (Clause 7), an *empty-declaration* (Clause 7), a *using-directive* (7.3.4), an explicit instantiation declaration (14.7.2), or an explicit specialization (14.7.3) whose *declaration* is not a definition.

[*Example:* all but one of the following are definitions:

```
int a;                // defines a
extern const int c = 1; // defines c
int f(int x) { return x+a; } // defines f and defines x
struct S { int a; int b; }; // defines S, S::a, and S::b
struct X {            // defines X
    int x;             // defines non-static data member x
    static int y;      // declares static data member y
    X(): x(0) { }      // defines a constructor of X
};
int X::y = 1;          // defines X::y
enum { up, down };     // defines up and down
namespace N { int d; } // defines N and N::d
namespace N1 = N;      // defines N1
X anX;                 // defines anX
```

whereas these are just declarations:

```
extern int a;          // declares a
extern const int c;    // declares c
int f(int);            // declares f
struct S;              // declares S
typedef int Int;        // declares Int
extern X anotherX;     // declares anotherX
using N::d;            // declares d
```

Machine model

- To study the operational semantics, we must understand what our machine model will require.
- Need to know when our machine is “done” executing a program: the final expressions are **values**.

Values in λ^+

- We define a value in an inductive way:
 - Any integer i is a value.
 - Boolean constants `true` and `false` are values
 - Any lambda expression `lambda x. e` is a value.
 - `Nil` is a value.
 - If v_1, v_2 are values, then $v_1::v_2$ are values.
 - No other expression is a value.

Values in λ^+

- Those expressions are values:

10

$\text{lambda } x. 1 + 2$

true

Nil

$10 :: \text{lambda } y. y$

- Those expressions are NOT values:

$1 + 2$

$(\text{lambda } x. 1 + 2)10$

if Nil then 10 else 20

$(1 + 2) :: \text{Nil}$

Inference rules

$$\frac{\begin{array}{c} \text{Hypothesis 1} \\ \dots \\ \text{Hypothesis N} \end{array}}{\vdash \text{Conclusion}}$$

- This means “given hypothesis 1, ..., N, the conclusion is provable”

$$\frac{\begin{array}{c} \text{Mitem 1 grade} \geq 70 \\ \dots \\ \text{Final grade} \geq 140 \end{array}}{\vdash \text{Final grade: A}}$$

Operational Semantics

- Operational semantics: define how program states are related to final values
- The ***big-step*** evaluation relation asserts that *we can prove for any expression of the form e that the meaning of this expression will evaluate to v*

$$e \Downarrow v$$

Operational Semantics

$$\frac{}{i \Downarrow i} \text{INT}$$

Any integer constant i will evaluate to itself

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2}{e_1 + e_2 \Downarrow i_1 + i_2} \text{ADD}$$

if e_1 and e_2 both evaluate to integers, then $e_1 + e_2$ evaluates to the sum of those integers

$$\frac{\frac{\text{INT} \frac{}{1 \Downarrow 1} \quad \text{INT} \frac{}{2 \Downarrow 2}}{\text{ADD} \frac{1 + 2 \Downarrow 3}} \quad \frac{}{4 \Downarrow 4} \text{INT}}{\text{ADD} \frac{(1 + 2) + 4 \Downarrow 7}}$$

Operational Semantics

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2 \quad i_1 \odot i_2 \text{ holds}}{e_1 \odot e_2 \Downarrow \text{true}} \text{ PREDTRUE}$$

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2 \quad i_1 \odot i_2 \text{ does not hold}}{e_1 \odot e_2 \Downarrow \text{false}} \text{ PREDFALSE}$$

The predicate operators $\odot \in \{=, <, >\}$ evaluate to false if the predicate does not hold and to true otherwise

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \text{ IFTRUE}$$

$$\frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \text{ IFFALSE}$$

Operational Semantics

$$\frac{}{\text{lambda } x. e \Downarrow \text{lambda } x. e} \text{ LAMBDA}$$

Lambda abstractions just evaluate to themselves

$$\frac{e_1 \Downarrow \text{lambda } x. e'_1 \quad e_2 \Downarrow v \quad [x \mapsto v]e'_1 \Downarrow v'}{(e_1 \ e_2) \Downarrow v'} \text{ APP}$$

To evaluate the application $(e_1 \ e_2)$, we first evaluate the expression e_1 . The operational semantics “**get stuck**” if e_1 is not a lambda abstraction. This notion of “getting stuck” in the operational semantics corresponds to a **runtime error**. Assuming the expression e_1 evaluates to a lambda expression, and e_2 evaluates to a value v , we evaluate the application expression by binding v to x and then evaluating the expression $[x \mapsto v]e'_1$ as in β -reduction in lambda calculus.

Operational Semantics

$$\frac{e_1 \Downarrow v_1 \quad [x \mapsto v_1]e_2 \Downarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v_2} \text{ LET}$$

First evaluate the initial expression e_1 , which yields value v_1 .

Next, we substitute x with v_1 in e_2 , and evaluate it to v_2 , which becomes the result of evaluating the entire let expression.