SWEN430 - Compiler Engineering

Lecture 5 - Operational Semantics I

Dr David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What are “Operational Semantics”?

“The meaning of a program in the **strict language** *is explained in terms of a* **hypothetical computer** *which performs the set of actions which constitute the elaboration of that program”*

– Report on the Algorithmic Language ALGOL 68.

Operational Semantics

"**Operational semantics** *describes the meaning of a programming language by specifying how it executes on an abstract machine.*

– Winskel'93

"**Operational semantics** *are classified in two categories: structural operational semantics (or* **small-step** *semantics) formally describe how the individual steps of a computation take place in a computer-based system; by opposition natural semantics (or big-step semantics) describe how the overall results of the executions are obtained.*"

–Wikipedia

Notation

$$\frac{\mathtt{A}}{\mathtt{B}}\ (\mathit{Name})$$

- To prove `B`, must be able to show `A`.

Rules of Inference

A rule of inference, inference rule or transformation rule is a logical form consisting of a function which takes premises, analyzes their syntax, and returns a conclusion (or conclusions).

–Wikipedia

The λ-calculus (a minor variation of)

$$
\begin{array}{lll}
t & ::= & \text{(Terms)} \\
 & \mid \quad x & \text{(Variables)} \\
 & \mid \quad v & \text{(Values)} \\
 & \mid \quad t\ t & \text{(App)} \\
 & & \\
v & ::= & \text{(Values)} \\
 & \mid \quad \lambda x.t & \text{(Function)} \\
 & \mid \quad c & \text{(Integer)}
\end{array}
$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(R-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(R-App2)}$$

$$\frac{}{(\lambda x.t_1)\ v_1 \longrightarrow t_1[x \mapsto v_1]} \quad \text{(R-App3)}$$

- A **simple** language — *syntax* and *semantics* on one slide!
- Useful starting point for **formalising** programming languages
- Term $t_1[x \mapsto t_2]$ is t_1 with all occurences of x replaced with t_2

Example λ-calculus programs

- $(\lambda x.x)\ (\lambda y.1) \longrightarrow (\lambda y.1)$
- $\Big(\big(\lambda x.(\lambda y.x\ y)\big)\big(\lambda z.z\big)\Big)\ 1 \longrightarrow \Big(\lambda y.\big((\lambda z.z)\ y\big)\Big)\ 1$
  $\longrightarrow (\lambda z.z)\ 1 \longrightarrow 1$
- $(\lambda x.x\ x)(\lambda y.y\ y) \longrightarrow ?$
- $\Big(\big(\lambda x.(\lambda y.x\ y)\big)\ \big((\lambda z.z)\ 1\big)\Big)\ 1 \longrightarrow ?$

Lambda Calculus in Java (Terms)

```
interface Term { }

class Var implements Term { String var; ... }
class App implements Term { Term lhs, rhs; ... }

interface Value extends Term {}
class Int implements Value { int val; ... }
class Fun implements Value {
    String param;
    Term body;
    ...
}
```

Lambda Calculus in Java (Substitution)

```
Term replace(Term t1, String v, Term t2) {
   if(t1 instanceof Var) {
      Var v1 = (Var) t1;
      return v.equals(v1.var) ? t2 : t1;
   } else if(t1 instanceof App) {
      App a = (App) t1;
      Term l = replace(a.lhs, v, t2);
      Term r = replace(a.rhs, v, t2);
      return new App(l, r);
   } else if(t1 instanceof Fun) {
      Fun f = (Fun) t1;
      if(f.param.equals(v)) { return t1; }
      else {
         Term b = replace(f.body, v, t2);
         return new Fun(f.param, b);
      }
   } else { return t1; }
}
```

Lambda Calculus in Java (Reduction)

```
Term execute(Term t) {
    if (t instanceof App) {
        App a = (App) t;
        if (!(a.lhs instanceof Value)) {
            // R-App1
            return new App(execute(a.lhs), a.rhs);
        } else if (!(a.rhs instanceof Value)) {
            // R-App2
            return new App(a.lhs, execute(a.rhs));
        } else if (a.lhs instanceof Fun) {
            // R-App3
            Fun fn = (Fun) a.lhs;
            return replace(fn.body, fn.param, a.rhs);
        }
    }
    return t;
}
```

More on the λ-calculus

$$(\lambda y.(\lambda y.y\ 1))\ (\lambda x.x)$$

- **Variable capture**. *How does above reduce?*
  - Assume λ parameters have unique names!
  - Can rename parameters in body of λ term

- **Currying** gives functions with multiple arguments!
  - $\lambda x, y.(\ldots)$ is equivalent to $\lambda x.(\lambda y.(\ldots))$

- **Control-structures** (e.g. `if`, `while`) can be implemented in λ-Calculus!

See: *Types and Programming Languages*, by Ben Pierce!

Big Step versus Small Step Semantics

- Presentation of λ-calculus on previous slides in **small-step** style
- Equivalent presentation in **big-step** style:

$$\frac{}{t \rightsquigarrow t} \qquad \text{(R-Reflex)}$$

$$\frac{t_1 \rightsquigarrow \lambda x.t_3 \quad t_2 \rightsquigarrow v_1}{t_1\ t_2 \rightsquigarrow t_3[x \mapsto v_1]} \qquad \text{(R-App)}$$

- Big-step presentation can often be **shorter**!
- But, harder to prove **strong properties** regarding type soundness

The While Language Calculus ($\lambda_{\mathbb{W}}$)

- **Objective:** to define **operational semantics** of WHILE
- **Problem:** WHILE is **very complex** (e.g. vs λ-calculus)
- **Therefore:** we develop a (significantly) simplified **calculus**, and hope that this characterises the interesting bits
- **Unfortunately:**
  - Need lots of **simplifications** to make calculus manageable!
  - As a result, it barely resembles **original language**
  - But, there's not much else we can do in the space available!

Syntax for $\lambda_{\mathbb{W}}$

p	::=	$f_1 \ldots f_n\ e$	*programs*
f	::=	$T_1\ m(T_2\ n)\{\ \overline{s}\ \}$	*functions*
e	::=		*expressions*
	|	v	*constants*
	|	n, m	*names*
	|	$e_1\ op\ e_2$	*binary*
	|	$e_1(e_2)$	*application*
	|	$\overline{s}$	*block*
s	::=	$n = e \mid \texttt{return}\ e$	*statements*
v	::=	null | true | false | ... | -1 | 0 | 1 | ... | $\lambda x.s$	*values*
T	::=	void | bool | int | null | $T_1 \vee T_2$ | $T_1 \rightarrow T_2$	*types*
op	::=	'!=' | '==' | '<' | '<=' | '>=' | '>'	*operators*
		| '+' | '−' | '*' |	

List of simplifications in $\lambda_{\mathtt{W}}$

- Functions accept **one parameter** and always **return something**
- No **variable declarations** (other than for parameters)
- No **unary operators**, and only a few **binary operators**
- No **compound data types** (i.e. records, arrays) in calculus core
- Only **assignments** and **returns** in calculus core
- Every **program** is an expression (rather than have `main()` function)
- Syntax for **Union types** given as $\mathtt{T}_1 \vee \mathtt{T}_2$

Example $\lambda_{\mathtt{W}}$ Programs

- **Valid** Programs:

  1) `1 + 1`

  2) `int f(int x){ x = x + 1; return x } f(1)`

  3) `int f(int x){ return x + 1 }`
     `int g(bool x){ return 1 }`
     `f(g(true))`

- **Invalid** (but syntactically correct) Programs:

  1) `1 + true`

  2) `int f(int x){ x = true; return x } f(1)`

  3) `int f(int x){ return x } f(true)`

References

- *The Formal Semantics of Programming Languages*, by Glynn Winskel (MIT Press, 1993).

- *Types and Programming Languages*, by Benjamin Pierce.