Formal Semantics III: Designing Rules (Part C)

CAS CS 320: Principles of Programming Languages

Thursday, April 4, 2024

REVIEWS FROM PRECEDING LECTURE

(April 2)

Applying The Evaluation Rules

To Our “Toy” Language –

called “StackLang” in these slides where we use “p/S” to denote a configuration instead of “(S,p)”

Operational Semantics of StackLang

```
<prog>  ::= <com> ; <prog> | []
<com>   ::= Push <int> | Pop | Swap | Add | Quit
<int>   ::= integers
```

StackLang is a simple stack manipulating language. When designing its operational semantics, we must account for the stack state.

```
<state> ::= <prog>/<stack> | ERROR
<stack> ::= <int> :: <stack> | []
```

Operational Semantics of StackLang

```
<prog>   ::= <com> ; <prog> | []
<com>    ::= Push <int> | Pop | Swap | Add | Quit
<int>    ::= integers

<state> ::= <prog>/<stack> | ERROR
<stack> ::= <int> :: <stack> | []
```

We include the stack as a part of StackLang's reduction relation.

$$P/S \to Q/R$$

This relation states that program P with stack S reduces to program Q with stack R.

Operational Semantics of StackLang

$$\frac{}{\text{Push } n \,;\; p/S \to p/(n :: S)}\text{push}$$

$$\frac{}{\text{Pop} \,;\; p/(n :: S) \to p/S}\text{pop-ok}$$

$$\frac{}{\text{Pop} \,;\; p/[] \to \text{ERROR}}\text{pop-error}$$

$$\frac{}{\text{Swap} \,;\; p/(m :: n :: S) \to p/(n :: m :: S)}\text{swap-ok}$$

$$\frac{}{\text{Swap} \,;\; p/(n :: []) \to \text{ERROR}}\text{swap-error1}$$

$$\frac{}{\text{Swap} \,;\; p/[] \to \text{ERROR}}\text{swap-error0}$$

$$\frac{}{\text{Add} \,;\; p/(m :: n :: S) \to p/(m + n :: S)}\text{add-ok}$$

$$\frac{}{\text{Add} \,;\; p/(n :: []) \to \text{ERROR}}\text{add-error1}$$

$$\frac{}{\text{Add} \,;\; p/\, [] \to \text{ERROR}}\text{add-error0}$$

Operational Semantics of StackLang

$$\frac{}{\text{Push } n\ ;\ p/S \rightarrow\ p/(n\ ::\ S)}\ \text{push}$$

$$\frac{}{\text{Pop}\ ;\ p/(n\ ::\ S) \rightarrow\ p/S}\ \text{pop-ok}$$

$$\frac{}{\text{Pop}\ ;\ p/[] \rightarrow \text{ERROR}}\ \text{pop-error}$$

$$\frac{}{\text{Swap}\ ;\ p/(m\ ::\ n\ ::\ S) \rightarrow p/(n\ ::\ m\ ::\ S)}\ \text{swap-ok}$$

$$\frac{}{\text{Swap}\ ;\ p/(n\ ::\ []) \rightarrow \text{ERROR}}\ \text{swap-error1}$$

$$\frac{}{\text{Swap}\ ;\ p/[] \rightarrow \text{ERROR}}\ \text{swap-error0}$$

$$\frac{}{\text{Add}\ ;\ p/(m\ ::\ n\ ::\ S) \rightarrow p/(m + n :: S)}\ \text{add-ok}$$

$$\frac{}{\text{Add}\ ;\ p/(n\ ::\ []) \rightarrow \text{ERROR}}\ \text{add-error1}$$

$$\frac{}{\text{Add}\ ;\ p/\ [] \rightarrow \text{ERROR}}\ \text{add-error0}$$

Incredibly similar to pattern matching in OCaml!

Operational Semantics of StackLang

Example: reduction of Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] in an empty stack.

We begin by deriving all the necessary single step reductions and labeling their conclusions.

$$(1) \quad \dfrac{}{\text{Push } 1 \;;\; \text{Push } 2 \;;\; \text{Swap} \;;\; \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; [\,] \rightarrow \text{Push } 2 \;;\; \text{Swap} \;;\; \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; (1 :: [\,])}\ \text{push}$$

Operational Semantics of StackLang

Example: reduction of Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] in an empty stack.

We begin by deriving all the necessary single step reductions and labeling their conclusions.

$$\text{(1)}\quad \frac{}{\text{Push } 1 \text{ ; Push } 2 \text{ ; Swap ; Add ; Quit ; [] / []} \rightarrow \text{Push } 2 \text{ ; Swap ; Add ; Quit ; [] / (1 :: [])}}\ \text{push}$$

$$\text{(2)}\quad \frac{}{\text{Push } 2 \text{ ; Swap ; Add ; Quit ; [] /(1 :: [])} \rightarrow \text{Swap ; Add ; Quit ; [] / (2 :: 1 :: [])}}\ \text{push}$$

Operational Semantics of StackLang

Example: reduction of Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] in an empty stack.

We begin by deriving all the necessary single step reductions and labeling their conclusions.

(1) $$\frac{}{\text{Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] / []} \to \text{Push 2 ; Swap ; Add ; Quit ; [] / (1 :: [])}}\ \text{push}$$

(2) $$\frac{}{\text{Push 2 ; Swap ; Add ; Quit ; [] /(1 :: [])} \to \text{Swap ; Add ; Quit ; [] / (2 :: 1 :: [])}}\ \text{push}$$

(3) $$\frac{}{\text{Swap ; Add ; Quit ; [] / (2 :: 1 :: [])} \to \text{Add ; Quit ; [] / (1 :: 2 :: [])}}\ \text{swap}$$

Operational Semantics of StackLang

Example: reduction of $\text{Push 1 ; Push 2 ; Swap ; Add ; Quit ; []}$ in an empty stack.

We begin by deriving all the necessary single step reductions and labeling their conclusions.

(1) $$\dfrac{}{\text{Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] / []} \to \text{Push 2 ; Swap ; Add ; Quit ; [] / (1 :: [])}}\ \text{push}$$

(2) $$\dfrac{}{\text{Push 2 ; Swap ; Add ; Quit ; [] /(1 :: [])} \to \text{Swap ; Add ; Quit ; [] / (2 :: 1 :: [])}}\ \text{push}$$

(3) $$\dfrac{}{\text{Swap ; Add ; Quit ; [] / (2 :: 1 :: [])} \to \text{Add ; Quit ; [] / (1 :: 2 :: [])}}\ \text{swap}$$

(4) $$\dfrac{}{\text{Add ; Quit ; [] / (1 :: 2 :: [])} \to \text{Quit ; [] / (3 :: [])}}\ \text{add}$$

Operational Semantics of StackLang

Example: reduction of Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] in an empty stack.

We begin by deriving all the necessary single step reductions and labeling their conclusions.

(1) $$\dfrac{}{\text{Push } 1 \;;\; \text{Push } 2 \;;\; \text{Swap} \;;\; \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; [\,] \rightarrow \text{Push } 2 \;;\; \text{Swap} \;;\; \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; (1 :: [\,])}\;\text{push}$$

(2) $$\dfrac{}{\text{Push } 2 \;;\; \text{Swap} \;;\; \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/ (1 :: [\,]) \rightarrow \text{Swap} \;;\; \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; (2 :: 1 :: [\,])}\;\text{push}$$

(3) $$\dfrac{}{\text{Swap} \;;\; \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; (2 :: 1 :: [\,]) \rightarrow \text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; (1 :: 2 :: [\,])}\;\text{swap}$$

(4) $$\dfrac{}{\text{Add} \;;\; \text{Quit} \;;\; [\,] \;/\; (1 :: 2 :: [\,]) \rightarrow \text{Quit} \;;\; [\,] \;/\; (3 :: [\,])}\;\text{add}$$

Cannot reduce further.

Operational Semantics of StackLang

Example: reduction of Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] in an empty stack.

Compose together single step reductions via the transitive rule for multi-step.

reflexive

(4) Quit ; [] / (3 :: []) $\to^*$ Quit ; [] / (3 :: [])

transitive

(3) Add ; Quit ; [] / (1 :: 2 :: []) $\to^*$ Quit ; [] / (3 :: [])

transitive

(2) Swap ; Add ; Quit ; [] / (2 :: 1 :: []) $\to^*$ Quit ; [] / (3 :: [])

transitive

(1) Push 2 ; Swap ; Add ; Quit ; [] /(1 :: []) $\to^*$ Quit ; [] / (3 :: [])

transitive

Push 1 ; Push 2 ; Swap ; Add ; Quit ; [] / [] $\to^*$ Quit ; [] / (3 :: [])

(THIS PAGE INTENTIONALLY LEFT BLANK)