Administrative

Talk in the Programming Language Seminar on Monday:

Classes = Objects + Data Abstraction
Kathleen Fisher, Stanford University

We probably won't get very far on the ML paper today, but will cover it next Tuesday.

Postfix Operational Semantics

$$< N.Q,\ S > \ \rightarrow\ < Q,\ N.S >$$

$$< (Q_{exec}).Q_{rest},\ S > \ \rightarrow\ < Q_{rest},\ (Q_{exec}).S >$$

$$< \texttt{pop}.Q,\ V.S > \ \rightarrow\ < Q,\ S >$$

$$< \texttt{swap}.Q,\ V_1.V_2.S > \ \rightarrow\ < Q,\ V_2.V_1.S >$$

$$< \texttt{if}.Q_{rest},\ V_{false}.V_{true}.0.S > \ \rightarrow\ < Q_{rest},\ V_{false}.S >$$

$$< \texttt{if}.Q_{rest},\ V_{false}.V_{true}.N.S > \ \rightarrow\ < Q_{rest},\ V_{true}.S >$$

$$< \texttt{exec}.Q_{rest},\ Q_{exec}.S > \ \rightarrow\ < Q_{exec}@Q_{rest},\ S >$$

Postfix Operational Semantics (con't)

$$< A.Q,\ N_1.N_2.S > \ \rightarrow\ < Q,\ N_{result}.S >$$
where $(A\ N_2\ N_1)$ evaluates to N_{result}

$$< R.Q,\ N_1.N_2.S > \ \rightarrow\ < Q,\ 1.S >$$
where $(R\ N_2\ N_1)$ evaluates to true

$$< R.Q,\ N_1.N_2.S > \ \rightarrow\ < Q,\ 0.S >$$
where $(R\ N_2\ N_1)$ evaluates to false

Alternate Rules for Exec

Last time we had two inference rules:

$$\frac{< Q_{exec},\ S > \ \rightarrow\ < Q'_{exec}, S' >}{< \texttt{exec}.Q_{rest}, Q_{exec}.S > \ \rightarrow\ < \texttt{exec}.Q_{rest}, Q'_{exec}, S' >}$$

$$< \texttt{exec}.Q_{rest}, ().S > \ \rightarrow\ < Q_{rest}, S >$$

The following single axiom was suggested in class:

$$< \texttt{exec}.Q_{rest},\ Q_{exec}.S > \ \rightarrow\ < Q_{exec}@Q_{rest},\ S >$$

Are these two formulations of `exec` semantics equivalent?

Example of exec Semantics using 1 Axiom

$$< ((2\ (3\ \texttt{mul})\ \texttt{exec})\ 1\ \texttt{swap exec add}),\ \ []>$$

$$\rightarrow\ < (1\ \texttt{swap exec add}),\ \ [(2\ (3\ \texttt{mul})\ \texttt{exec})]>$$

$$\rightarrow\ < (\texttt{swap exec add}),\ \ [1\ (2\ (3\ \texttt{mul})\ \texttt{exec})]>$$

$$\rightarrow\ < (\texttt{exec add}),\ \ [(2\ (3\ \texttt{mul})\ \texttt{exec})\ 1]>$$

$$\rightarrow\ < (2\ (3\ \texttt{mul})\texttt{exec add}),\ \ [1]>$$

$$\rightarrow\ < ((3\ \texttt{mul})\texttt{exec add}),\ \ [2\ 1]>$$

$$\rightarrow\ < (\texttt{exec add}),\ \ [(3\ \texttt{mul})\ 2\ 1]>$$
$$\rightarrow\ < (3\ \texttt{mul add}),\ \ [2\ 1]>$$
$$\rightarrow\ < (\texttt{mul add}),\ \ [3\ 2\ 1]>$$
$$\rightarrow\ < (\texttt{add}),\ \ [6\ 1]>$$
$$\rightarrow\ < (),\ \ [7]>$$

Example of exec Semantics using 2 Rules

$< ((2\ (3\ \texttt{mul})\ \texttt{exec})\ 1\ \texttt{swap exec add}),\ [] >$

$\rightarrow\ < (1\ \texttt{swap exec add}),\ [(2\ (3\ \texttt{mul})\ \texttt{exec})] >$

$\rightarrow\ < (\texttt{swap exec add}),\ [1\ (2\ (3\ \texttt{mul})\ \texttt{exec})] >$

$\rightarrow\ < (\texttt{exec add}),\ [(2\ (3\ \texttt{mul})\ \texttt{exec})\ 1] >$

$\rightarrow\ < (\texttt{exec add}),\ [((3\ \texttt{mul})\ \texttt{exec})\ 2\ 1] >$

$\rightarrow\ < (\texttt{exec add}),\ [(\texttt{exec})\ (3\ \texttt{mul})\ 2\ 1] >$

$\rightarrow\ < (\texttt{exec add}),\ [(\texttt{exec})\ (\texttt{mul})\ 3\ 2\ 1] >$
$\rightarrow\ < (\texttt{exec add}),\ [(\texttt{exec})\ ()\ 6\ 1] >$
$\rightarrow\ < (\texttt{exec add}),\ [()\ 6\ 1] >$
$\rightarrow\ < (\texttt{add}),\ [6\ 1] >$
$\rightarrow\ < (),\ [7] >$

Proof (sketch) that Both are Equivalent

Given any configuration of the form $<\texttt{exec}Q, S>$, show that both reduction relations give the same result: either they both produce a stuck configuration or they both lead to the same result.
1) If the stack S is empty, the configuration is:

$$<\texttt{exec}\ Q,\ []>$$

which is stuck in either case.
2) If the stack S is non-empty, but the first element is not parenthesized, the configuration has the form:

$$<\texttt{exec}\ Q,\ V\ .\ S_1>$$

which is also stuck.
3) If the stack is nonempty and the top element is parenthesized but empty expressions, the configuration is:

$$<\texttt{exec}\ Q,\ ()\ .\ S_1>$$

Proof continued

but this reduces to:

- $< Q,\ S_1 >$ using the @ axiom, by definition of @ on an empty list.
- $< Q,\ S_1 >$ using the 2 rules, by the exec-done rule.

4) If the stack is nonempty and the top element is a parenthesize non-empty expression sequence, the configuration has the form:

$$< \texttt{exec}\ Q,\ (Q_1\ .\ Q_2)\ .\ S_1 >$$

- Using the @ axiom, this produces:

$$< Q_1\ .\ Q_2\ @\ Q,\ S_1 >$$

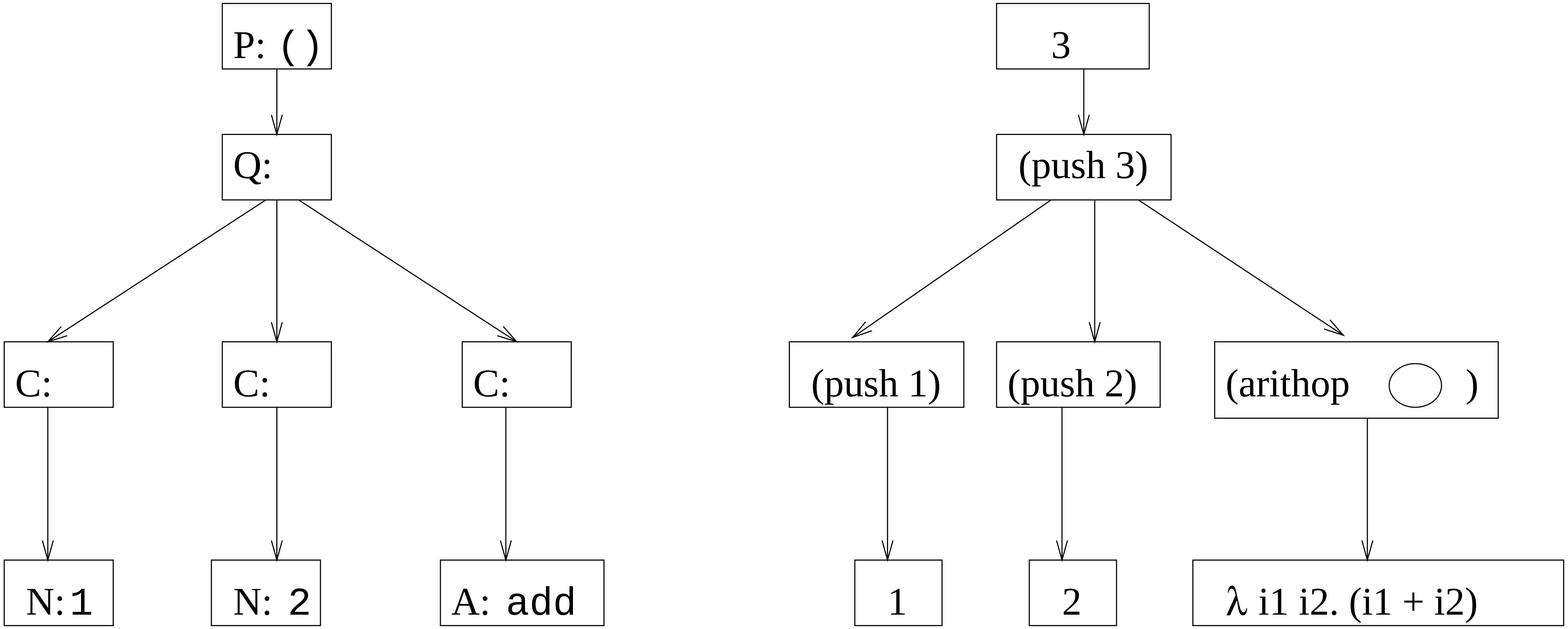
  Now either this configuration is stuck or there is some $Q_1'\ S_1'$:

$$< Q_1'\ .\ Q_2\ @\ Q,\ S_1' >$$

- Using the exec-progress rule, either the entire the configuration is stuck or there exists a $Q_1\ S_1$:

$$< \texttt{exec}\ Q,\ (Q_1'\ .\ Q_2),\ S_1' >$$

Postfix Denotational Semantics



The valuation function is a homomorphism defined on the structure of programs. Technicall, there are a set of functions that define the meaning, one for each “type” of expression: P, Q, C, N, R, A.

Postfix Semantic Domains

The kinds of objects that exist in the denotational semantics are given by the *semantic domain* for the language.

$$t \in StackTransform = Stack \rightarrow Stack$$

$$s \in Stack = Value * + Error$$

$$v \in Value = Int + StackTransform$$

$$a \in Answer = Value + Error$$

$$Error = \{error\}$$

$$i \in Int = \{..., 2, -1, 0, 1, 2, ...\}$$

$$b \in Bool = \{true, false\}$$

The variable names t, s, v, a, i, b will be used (with subscripts) to denote some value in the given domain.

Applying the Valuation Function to a Program

Informally, the effect of executing a sequence Q is to "execute each item in Q in turn." Making this formal gives us

$$\mathcal{Q}[\![]\!] = Id_{Stack} = \lambda\ s.s$$
$$\mathcal{Q}[\![C.Q]\!] = \mathcal{Q}[\![Q]\!] \circ \mathcal{Q}[\![C]\!]$$

(where $(f \circ g)(x) = f(g(x))$ denotes function composition).

This is the definition of a particular piece of the $[\![\ldots]\!]$ valuation function. Note that we have omitted the environment η, because there are not variables in Postfix programs.

Auxiliary Functions

Things get a bit complex when we take into account all the possible error conditions. To keep things neat, we introduce auxiliary functions, defined separately. For example,

$$\mathcal{C}[\![\mathtt{exec}]\!] = \lambda s.\ (\mathit{transform}\ (\mathit{top}\ s)\ (\mathit{pop}\ s))$$

The auxiliary function *top* returns either the top (first element) of its argument, or *error* if the argument is null. Likewise, *pop* returns either the tail of its stack argument, or *error* if given an empty sequence. The function *transform* returns either its argument, if that is a function, or *error*.

Postfix Denotational Semantics

P [[(Q)]] = *(top (Q* [[(Q)]] *empty-stack))*

C [[N]] = (*push N*[[N]])

C [[(Q)]] = (*push Q*[[Q]])

C [[A]] = (*arithop A*[[A]])

A [[div]] = λ *n1:nat* .λ *n2: nat* . **if** *(i2 = 0)* **then** *error-answer*

else *i1 / i2*

Q [[C . Q]] = C [[C]] $\circ$ Q [[Q]]

Q [[[]]] = λ s . s

N [[N]] = the integer value of N

Postfix Denotational Semantics

arithop: (Int->Int->Answer)->Stack-Transform

= λ *f s* . **case** *(top s)*

$\triangleright$ *(Int->Value i1)* $\Box$ **case** *(top (pop s))*

$\triangleright$ *(Int->Value i2)* $\Box$ *(push (f i1 i2) (pop (pop s)))*

$\triangleright$ **else** *error-stack* **endcase**

$\triangleright$ **else** *error-stack* **endcase**

What does this tell us about arithmetic operators in Postfix?

Postfix Denotational Semantics

pop: Stack-Transform

= λ *s* . **case** *s*

▷ *(Value*->Stack v*)* ⇒ **if** *(empty? v*)* **then** *error-stack*

else *(Value*->Stack (tail v*))*

▷ **else** *error-stack* **endcase**

top: Stack->Answer

= λ *s* . **case** *s*

▷ *(Value*->Stack v*)* ⇒ **if** *(empty? v*)* **then** *error-answer*

else *(Value*->Stack (head v*))*

▷ **else** *error-answer* **endcase**