Formal Semantics III: Designing Rules (Part A)

CAS CS 320: Principles of Programming Languages

Thursday, April 4, 2024

Administrivia

- Homework 8 due Saturday, Apr 6 – not Friday, Apr 5 – by 11:59 pm.
- Project 1 (i.e. Homework 9) posted Friday, Apr 5 (tomorrow), and due Friday, Apr 12.
- Final exam on Wednesday, May 8, 3:00-5:00 pm in STO 50.

REVIEWS FROM PRECEDING LECTURE

(April 2)

Setting Up The Evaluation Rules

High-Level

$$(S, p) \longrightarrow (S', p')$$

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

High-Level

$$\underbrace{(S, \boxed{p})}_{} \longrightarrow (S', \boxed{p'})$$

$$(S, \overset{\text{program}}{\boxed{p}}) \longrightarrow (S', \overset{\text{transformed program}}{\boxed{p'}})$$

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

High-Level

$$\underbrace{\overbrace{(S}_{\text{state}}, \overbrace{p}^{\text{program}})}_{} \longrightarrow (\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})$$

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

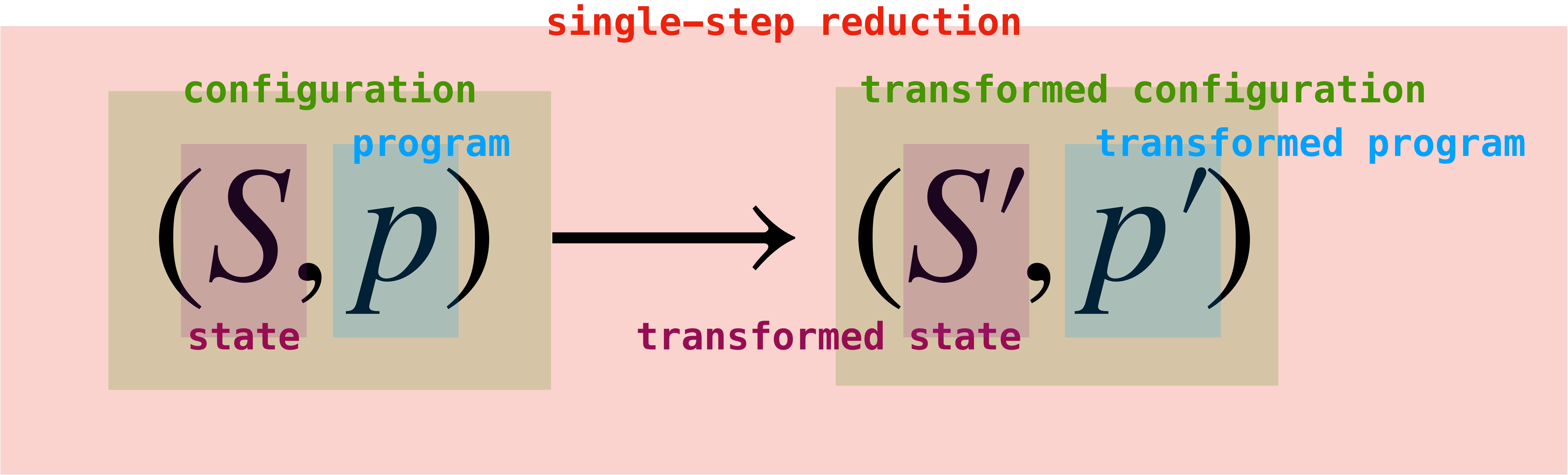
High-Level

$$\overbrace{(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}})}^{\text{configuration}} \longrightarrow \overbrace{(\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})}^{\text{transformed configuration}}$$

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

High-Level



Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

Example: Arithmetic Expressions

program

$$(\underbrace{\varnothing}_{\text{state}}\;,\; 10 \times (2+3)) \longrightarrow (\varnothing\;,\; 10 \times 5) \longrightarrow (\varnothing\;,\; 50)$$

<u>State:</u> none

<u>Program:</u> arithmetic expression

Example: (Fragment of) OCaml

program

$(\varnothing\,,$ `let x = 3 in if x > 10 then 4 else 5`$)$ $\longrightarrow (\varnothing\,,$ `if 3 > 10 then 4 else 5`$)$

state

$\longrightarrow (\varnothing\,,$ `if false then 4 else 5`$)$

$\longrightarrow (\varnothing\,,$ `5`$)$

State: none

Program: OCaml expression

For purely functional languages
there is no state

Example: Stack-Oriented Language

state program

$(\varnothing\ ,\ \texttt{push 2; push 3; add}) \longrightarrow$

$(2 :: \varnothing,\ \texttt{push 3; add}) \longrightarrow$

$(3 :: 2 :: \varnothing,\ \texttt{add}) \longrightarrow$

$(5 :: \varnothing,\ \epsilon)$

State: stack (i.e., list) of values

Program: sequence of commands for manipulating the stack

REVIEWS FROM PRECEDING LECTURE

(April 2)

Applying The Evaluation Rules

Reduction Rules

premise premise premise

$$\frac{P_1 \qquad P_2 \qquad \cdots \qquad P_k}{C \to C'}\ \texttt{red-rule}$$

conclusion name

Reduction Rules

$$\frac{\overset{\texttt{premise}}{\boxed{P_1}} \qquad \overset{\texttt{premise}}{\boxed{P_2}} \qquad \cdots \qquad \overset{\texttt{premise}}{\boxed{P_k}}}{\underset{\texttt{conclusion}}{\boxed{C \to C'}}} \ \overset{\texttt{name}}{\boxed{\texttt{red-rule}}}$$

Then general form of a reduction rule has a collection of **premises** and a **conclusion**, which is a shape of a reduction

Reduction Rules

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \dots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C \to C'}}\ \overset{\text{name}}{\texttt{red-rule}}$$

Then general form of a reduction rule has a collection of **premises** and a **conclusion,** which is a shape of a reduction

Think of a "shape" like an OCaml "pattern"

Reduction Rules

$$\frac{\overset{\texttt{premise}}{P_1} \qquad \overset{\texttt{premise}}{P_2} \qquad \dots \qquad \overset{\texttt{premise}}{P_k}}{\underset{\texttt{conclusion}}{C \to C'}}\ \overset{\texttt{name}}{\texttt{red-rule}}$$

Then general form of a reduction rule has a collection of **premises** and a **conclusion**, which is a shape of a reduction

Think of a "shape" like an OCaml "pattern"

A premise may be another reduction "shape" or a **trivial condition** (we will see examples in the next slide)

Example

premise

$$\frac{e_1 \longrightarrow e_1'}{\texttt{add}\ e_1\ e_2 \longrightarrow \texttt{add}\ e_1'\ e_2}\ \texttt{add-left}$$

conclusion

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int> | ERROR
<op>   ::= add | sub  | eq
<bool> ::= true | false
<int>  ::= ...
```

<u>Example Programs:</u>

```
(add 2 3)
(add (add 2 3) 5)
(eq (add 2 3) (sub 7 2))
(add true 2)
```

Example

premise

$$\frac{e_1 \longrightarrow e_1'}{\texttt{add}\ e_1\ e_2 \longrightarrow \texttt{add}\ e_1'\ e_2}\ \texttt{add-left}$$

conclusion

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int> | ERROR
<op>   ::= add | sub  | eq
<bool> ::= true | false
<int>  ::= ...
```

<u>Example Programs:</u>

```
(add 2 3)
(add (add 2 3) 5)
(eq (add 2 3) (sub 7 2))
(add true 2)
```

If e_1 *reduces to* e_1' *in one step, then* add e_1 e_2 *reduces to* add e_1' e_2 *in one step*

Example

premise

$$\frac{e_1 \longrightarrow e_1'}{\texttt{add}\ e_1\ e_2 \longrightarrow \texttt{add}\ e_1'\ e_2}\ \texttt{add-left}$$

conclusion

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int> | ERROR
<op>   ::= add | sub  | eq
<bool> ::= true | false
<int>  ::= ...
```

<u>Example Programs:</u>

```
(add 2 3)
(add (add 2 3) 5)
(eq (add 2 3) (sub 7 2))
(add true 2)
```

If e_1 *reduces to* e_1' *in one step, then* $\texttt{add}\ e_1\ e_2$ *reduces to* $\texttt{add}\ e_1'\ e_2$ *in one step*

In this case, the premise is another reduction

Another Example

$$\frac{n_1 \texttt{ is a number} \qquad n_2 \texttt{ is a number}}{\texttt{add}\ n_1\ n_2 \longrightarrow n_1 + n_2}\ \texttt{add-ok}$$

If n_1 and n_2 are numbers then `add` n_1 n_2 *reduces in one step to* ***the number*** $n_1 + n_2$

In this case, the premises are trivial conditions (it should be easy to determine if something is a number)

Rules for Addition

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int> | ERROR
<op>   ::= add | sub  | eq
<bool> ::= true | false
<int>  ::= ...
```

$$\frac{e_1 \longrightarrow e_1'}{\texttt{add}\ e_1\ e_2 \longrightarrow \texttt{add}\ e_1'\ e_2}\ \texttt{add-left} \qquad \frac{e_2 \longrightarrow e_2'}{\texttt{add}\ e_1\ e_2 \longrightarrow \texttt{add}\ e_1\ e_2'}\ \texttt{add-right}$$

$$\frac{v\ \texttt{is a Bool or Error}}{\texttt{add}\ v\ e \longrightarrow \texttt{Error}}\ \texttt{add-left-error} \qquad \frac{v\ \texttt{is a Bool or Error}}{\texttt{add}\ e\ v \longrightarrow \texttt{Error}}\ \texttt{add-right-error}$$

error handling

$$\frac{n_1\ \texttt{is a number} \qquad n_2\ \texttt{is a number}}{\texttt{add}\ n_1\ n_2 \longrightarrow n_1 + n_2}\ \texttt{add-ok}$$

(THIS PAGE INTENTIONALLY LEFT BLANK)