CPSC 311: Definition of Programming Languages: Assignment 4: Stepping

Joshua Dunfield and Khurram A. Jafery
University of British Columbia

November 1, 2016

1 Logistics

You may work in teams of **up to 2** on this assignment. (You *can* work individually. But we recommend that you collaborate, mostly for your benefit, but—I'll be honest—also for ours: there are many students taking 311, but not very many TAs, and working individually means one more assignment to mark. **Repeating this reminder for a4. Also, you can submit as a team even if you complete the assignment separately and meet only to decide on a combined solution. We don't particularly recommend that, but it's still one less assignment to mark, and you'll almost certainly learn something from the other person's solution.**)

You must include a README.txt file based on this template:

`http://www.ugrad.cs.ubc.ca/~cs311/2016W1/assignments/support/README.txt`

For your final submission, be sure you have replaced **all** of the "`TODO`"s in `README.txt`.

handin has not yet been set up for this assignment. When handin has been set up, we will make an announcement on Piazza.

Download `http://www.ugrad.cs.ubc.ca/~cs311/2016W1/assignments/a4.rkt`

1.1 Important! (new in a3, still in a4)

If your code is rejected by "Check Syntax", or the `handin` script can't run your code, you will receive a mark of 0 **for the entire coding portion of the assignment**.

So, if you get stuck on one problem, **comment out the code that doesn't work**, and try to explain in a comment what you were trying to do. **Make sure that your final handin, at minimum, prints test results**—even if all the tests fail!

This assignment is due at 22:00 (10:00pm) on Saturday, November 12, 2016.

1.2 Handin

You must turn in the coding part of the assignment using the **command-line version** of the handin program.

For handin, this assignment is called `a4`. Submit two files:

- `a4.rkt`
- `README.txt`

If you are working in a team, submit only one set of files. If you have both run handin for a4 already, have one person overwrite their handin with a directory containing only a file called `please-mark-aaaaa` where `aaaaa` is the CS ugrad username of your partner.

Just a reminder, late assignments are not accepted (except for the "grace period" of a few minutes, which you shouldn't rely on), and (basically) no excuses will be entertained. So, hand in your assignments early and often!

Avoid using DrRacket comment boxes, because handin is still afraid of them. Comments using "`;`" and "`#|` . . . `|#`" are fine.

2 Overview

- Problems 1 and 2 are about small-step semantics; hand in your solution on paper.
- Problems 3–5 are also about small-step semantics; hand in your solution using `handin` (see above).
- There is a bonus problem.

3 Substitution

$$
\begin{array}{rcl}
[e2/x](\mathsf{Num}\ n) &=& (\mathsf{Num}\ n) \\
[e2/x](\mathsf{Leaf}\ A) &=& (\mathsf{Leaf}\ A) \\
[e2/x](\mathsf{Branch}\ eKey\ eL\ eR) &=& (\mathsf{Branch}\ [e2/x]eKey\ [e2/x]eL\ [e2/x]eR) \\
[e2/x](\mathsf{Tree\text{-}case}\ e\ eLeaf\ x1\ x2\ x3\ eBranch) &=& (\mathsf{Tree\text{-}case}\ [e2/x]e\ [e2/x]eLeaf\ x1\ x2\ x3\ eBranch) \\
&& \text{if } x = x1 \text{ or } x = x2 \text{ or } x = x3 \\
[e2/x](\mathsf{Tree\text{-}case}\ e\ eLeaf\ x1\ x2\ x3\ eBranch) &=& (\mathsf{Tree\text{-}case}\ [e2/x]e\ [e2/x]eLeaf\ x1\ x2\ x3\ [e2/x]eBranch) \\
&& \text{if } x \neq x1 \text{ and } x \neq x2 \text{ and } x \neq x3 \\
[e2/x](\mathsf{Id}\ x) &=& e2 \\
[e2/x](\mathsf{Id}\ y) &=& (\mathsf{Id}\ y) \quad \text{if } x \neq y \\
[e2/x](\mathsf{Lam}\ x\ A\ eB) &=& (\mathsf{Lam}\ x\ A\ eB) \\
[e2/x](\mathsf{Lam}\ y\ A\ eB) &=& (\mathsf{Lam}\ y\ A\ [e2/x]eB) \\
&& \text{if } x \neq y \\
[e2/x](\mathsf{App}\ eFun\ eArg) &=& (\mathsf{App}\ [e2/x]eFun\ [e2/x]eArg) \\
[e2/x](\mathsf{Binop}\ op\ eL\ eR) &=& (\mathsf{Binop}\ op\ [e2/x]eL\ [e2/x]eR) \\
[e2/x](\mathsf{Pair}\ eL\ eR) &=& (\mathsf{Pair}\ [e2/x]eL\ [e2/x]eR) \\
[e2/x](\mathsf{Inl}\ eL) &=& (\mathsf{Inl}\ [e2/x]eL) \\
[e2/x](\mathsf{Inr}\ eR) &=& (\mathsf{Inr}\ [e2/x]eR) \\
[e2/x](\mathsf{Bfalse}) &=& (\mathsf{Bfalse}) \\
[e2/x](\mathsf{Btrue}) &=& (\mathsf{Btrue}) \\
[e2/x](\mathsf{Ite}\ e\ eThen\ eElse) &=& (\mathsf{Ite}\ [e2/x]e\ [e2/x]eThen\ [e2/x]eElse) \\
[e2/x](\mathsf{Let}\ x\ e\ eB) &=& (\mathsf{Let}\ x\ [e2/x]e\ eB) \\
[e2/x](\mathsf{Let}\ y\ e\ eB) &=& (\mathsf{Let}\ y\ [e2/x]e\ [e2/x]eB) \\
&& \text{if } x \neq y \\
[e2/x](\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eB) &=& (\mathsf{Pair\text{-}case}\ [e2/x]e\ x1\ x2\ eB) \\
&& \text{if } x = x1 \text{ or } x = x2 \\
[e2/x](\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eB) &=& (\mathsf{Pair\text{-}case}\ [e2/x]e\ x1\ x2\ [e2/x]eB) \\
&& \text{if } x \neq x1 \text{ and } x \neq x2 \\
[e2/x](\mathsf{Rec}\ x\ A\ eB) &=& (\mathsf{Rec}\ x\ A\ eB) \\
[e2/x](\mathsf{Rec}\ y\ A\ eB) &=& (\mathsf{Rec}\ y\ A\ [e2/x]eB) \qquad \text{if } x \neq y \\
[e2/x](\mathsf{Unit}) &=& (\mathsf{Unit}) \\
[e2/x](\mathsf{Par}\ eL\ eR) &=& (\mathsf{Par}\ [e2/x]eL\ [e2/x]eR) \\
[e2/x](\mathsf{Choose}\ eL\ eR) &=& (\mathsf{Choose}\ [e2/x]eL\ [e2/x]eR) \\
[e2/x](\mathsf{Sum\text{-}case}\ e\ xL\ eL\ xR\ eR) &=& (\mathsf{Sum\text{-}case}\ [e2/x]e\ xL\ eL'\ xR\ eR') \\
&& \text{where } \begin{array}{ll} eL' = eL & \text{if } x = xL \\ eL' = [e2/x]eL & \text{if } x \neq xL \\ eR' = eR & \text{if } x = xR \\ eR' = [e2/x]eR & \text{if } x \neq xR \end{array} \\
[e2/x](\mathsf{Inl\text{-}case}\ e\ xL\ eL) &=& (\mathsf{Inl\text{-}case}\ [e2/x]e\ xL\ eL') \\
&& \text{where } \begin{array}{ll} eL' = eL & \text{if } x = xL \\ eL' = [e2/x]eL & \text{if } x \neq xL \end{array} \\
[e2/x](\mathsf{Inr\text{-}case}\ e\ xR\ eR) &=& (\mathsf{Inr\text{-}case}\ [e2/x]e\ xR\ eR') \\
&& \text{where } \begin{array}{ll} eR' = eR & \text{if } x = xR \\ eR' = [e2/x]eR & \text{if } x \neq xR \end{array}
\end{array}
$$

Definitions of values and evaluation contexts:

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & (\mathsf{Num}\ n) \\
 & & \mid & (\mathsf{Lam}\ x\ e) \\
 & & \mid & (\mathsf{Pair}\ v\ v) \\
 & & \mid & (\mathsf{Leaf}) \\
 & & \mid & (\mathsf{Branch}\ v\ v\ v) \\
 & & \mid & (\mathsf{Unit}) \\
 & & \mid & (\mathsf{Inl}\ v) \\
 & & \mid & (\mathsf{Inr}\ v)
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Evaluation contexts} & \mathcal{C} & ::= & [\,] \\
 & & \mid & (\mathsf{Binop}\ \mathsf{op}\ \mathcal{C}\ e) \\
 & & \mid & (\mathsf{Binop}\ \mathsf{op}\ v\ \mathcal{C}) \\
 & & \mid & (\mathsf{App}\ \mathcal{C}\ e) \\
 & & \mid & (\mathsf{App}\ v\ \mathcal{C}) \\
 & & \mid & (\mathsf{Let}\ x\ \mathcal{C}\ e) \\
 & & \mid & (\mathsf{Ite}\ \mathcal{C}\ e\ e) \\
 & & \mid & (\mathsf{Pair}\ \mathcal{C}\ e) \\
 & & \mid & (\mathsf{Pair}\ v\ \mathcal{C}) \\
 & & \mid & (\mathsf{Pair\text{-}case}\ \mathcal{C}\ x\ x\ e) \\
 & & \mid & (\mathsf{Branch}\ \mathcal{C}\ e\ e) \\
 & & \mid & (\mathsf{Branch}\ v\ \mathcal{C}\ e) \\
 & & \mid & (\mathsf{Branch}\ v\ v\ \mathcal{C}) \\
 & & \mid & (\mathsf{Tree\text{-}case}\ \mathcal{C}\ e\ x\ x\ x\ e) \\
 & & \mid & (\mathsf{Inl}\ \mathcal{C}) \\
 & & \mid & (\mathsf{Inr}\ \mathcal{C}) \\
 & & \mid & (\mathsf{Sum\text{-}case}\ \mathcal{C}\ x\ e\ x\ e) \\
 & & \mid & (\mathsf{Inl\text{-}case}\ \mathcal{C}\ x\ e) \\
 & & \mid & (\mathsf{Inr\text{-}case}\ \mathcal{C}\ x\ e) \\
 & & \mid & (\mathsf{Par}\ \mathcal{C}\ e) \\
 & & \mid & (\mathsf{Par}\ e\ \mathcal{C})
\end{array}
$$

Figure 1 **Values and evaluation contexts**

$\boxed{e1 \longrightarrow e2}$ Expression $e1$ steps to $e2$

Reduction rules:

$$\frac{v1 \text{ op } v2 = v}{(\mathsf{Binop}\ \mathsf{op}\ v1\ v2) \longrightarrow v}\ \text{Step-binop} \qquad \frac{}{(\mathsf{Let}\ x\ v1\ e2) \longrightarrow [v1/x]\,e2}\ \text{Step-let}$$

$$\frac{}{(\mathsf{App}\ (\mathsf{Lam}\ x\ eB)\ v) \longrightarrow [v/x]\,eB}\ \text{Step-app-value} \qquad \frac{}{(\mathsf{Rec}\ u\ e) \longrightarrow [(\mathsf{Rec}\ u\ e)/u]\,e}\ \text{Step-rec}$$

$$\frac{}{(\mathsf{Ite}\ (\mathsf{Btrue})\ eThen\ eElse) \longrightarrow eThen}\ \text{Step-ite-true} \qquad \frac{}{(\mathsf{Ite}\ (\mathsf{Bfalse})\ eThen\ eElse) \longrightarrow eElse}\ \text{Step-ite-false}$$

$$\frac{}{(\mathsf{Pair\text{-}case}\ (\mathsf{Pair}\ v1\ v2)\ x1\ x2\ eBody) \longrightarrow [v2/x2]\,[v1/x1]\,eBody}\ \text{Step-pair-case}$$

$$\frac{}{(\mathsf{Tree\text{-}case}\ (\mathsf{Leaf})\ eLeaf\ xK\ xL\ xR\ eBranch) \longrightarrow eLeaf}\ \text{Step-tree-case-leaf}$$

$$\frac{}{(\mathsf{Tree\text{-}case}\ (\mathsf{Branch}\ vK\ vL\ vR)\ eLeaf\ xK\ xL\ xR\ eBranch) \longrightarrow [vR/xR]\,[vL/xL]\,[vK/xK]\,eBranch}\ \text{Step-tree-case-branch}$$

$$\frac{}{(\mathsf{Inl\text{-}case}\ (\mathsf{Inl}\ v)\ xL\ eL) \longrightarrow [v/xL]\,eL}\ \text{Step-inl-case} \qquad \frac{}{(\mathsf{Inr\text{-}case}\ (\mathsf{Inr}\ v)\ xR\ eR) \longrightarrow [v/xR]\,eR}\ \text{Step-inr-case}$$

$$\frac{}{\begin{array}{l}(\mathsf{Sum\text{-}case}\ (\mathsf{Inl}\ v)\ xL\ eL\ xR\ eR)\\ \longrightarrow [v/xL]\,eL\end{array}}\ \text{Step-sum-case-left} \qquad \frac{}{\begin{array}{l}(\mathsf{Sum\text{-}case}\ (\mathsf{Inr}\ v)\ xL\ eL\ xR\ eR)\\ \longrightarrow [v/xR]\,eR\end{array}}\ \text{Step-sum-case-right}$$

$$\frac{}{(\mathsf{Par}\ v1\ e2) \longrightarrow v1}\ \text{Step-par-left} \qquad \frac{}{(\mathsf{Par}\ e1\ v2) \longrightarrow v2}\ \text{Step-par-right}$$

$$\frac{}{(\mathsf{Choose}\ e1\ e2) \longrightarrow e1}\ \text{Step-choose-left} \qquad \frac{}{(\mathsf{Choose}\ e1\ e2) \longrightarrow e2}\ \text{Step-choose-right}$$

Context rule:

$$\frac{e \longrightarrow e'}{\mathcal{C}[e] \longrightarrow \mathcal{C}[e']}\ \text{Step-context}$$

Figure 2 **Small-step semantics**

4 Problems

IMPORTANT: Read this before you start coding!

- Don't worry about making your code fast; clarity and correctness are much more important.

  You must implement code that follows the rules; you should also make your code similar to the rules.

Problems 1–2 are not coding problems. Turn them in on paper in the box in room X235. You can print this part of the assignment and write on it, or write out your answers separately. **Be sure to write your name, student number, and CS userid.** If you are working with someone else, turn in **one** solution.

	Name	Student ID#	CS userid
Student #1:			
Student #2:			

Problem 1: Small steps to big steps

The small-step semantics in Figure 2 has two reduction rules for Sum-case: Step-sum-case-left and Step-sum-case-right. Write the corresponding **big-step** rule(s) for Sum-case below. You do not need to name the rule(s).

$\boxed{e \Downarrow v}$ Expression e evaluates to value v

Problem 2: One Step Up

For each of the following expressions, attempt to evaluate it using the small-step semantics provided above.

- It is possible that the expression **converges** to a value, i.e. after taking a finite number of steps, the expression reduces to a value.
- It is possible that the expression **diverges**, i.e. no progress is made by taking any further steps as the size of the program never reduces—in this case, stop after a few steps and briefly explain why the expression diverges; or
- It is possible that stepping gets **stuck**, i.e., the expression is not a value, and there is no rule that you could apply to reduce the expression any further—in this case, stop when no further steps can be taken and briefly explain why.

Underline the subexpression being currently reduced in a given step, and beneath the line write the name of the reduction rule you're applying. While reducing the expression, you may need to substitute; it is best to write only the final result of substitution within the step.

Part 2a:

$$\left(\mathsf{App}\ \left(\mathsf{Lam}\ x\ \left(\mathsf{App}\ (\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Par}\ (\mathsf{Unit})\ (\mathsf{Lam}\ y\ (\mathsf{Id}\ x))))\ (\mathsf{Num}\ 2))\ (\mathsf{Id}\ x)\right)\right)\ \left(\mathsf{Num}\ 1\right)\right)$$

$\longrightarrow$

Part 2b:

$$\left(\mathsf{Rec}\ x\ (\mathsf{Rec}\ y\ (\mathsf{Choose}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ y)))\right)$$

$\longrightarrow$

Problem 2: One Step Up, continued

Part 2c:

$$\left(\mathsf{Inr\text{-}case}\ \left(\mathsf{Sum\text{-}case}\ (\mathsf{Inr}\ (\mathsf{Btrue}))\ x1\ \left(\mathsf{Inr}\ (\mathsf{Id}\ x1)\right)\ x2\ \left(\mathsf{Inr}\ (\mathsf{Inr}\ (\mathsf{Id}\ x2))\right)\right)\ x\ \left(\mathsf{Id}\ x\right)\right)$$

$\longrightarrow$

Problem 3: Tangled Up in Sums

In `a4.rkt` we have implemented a small-step interpreter and added several new features: "angelic nondeterminism" Par (better known as parallelism), "demonic nondeterminism" Choose, **sum types**, and a **unit type**. Recall that a Pair in Fun is like a **define-type** that has one variant, with two arguments; sum types are like **define-type**s that have two variants, each with one argument.

Sums are **introduced** by the expressions Inl and Inr. These correspond to the variant names in a **define-type**.

Sums can be **eliminated** in several ways:

- The expression Sum-case is like a **type-case** with two branches, one for Inl and one for Inr.
- The expression Inl-case is somewhat like a **type-case** with only **one** branch, for Inl. Trying to step an Inl-case on an Inr will get stuck.
- The expression Inr-case is somewhat like a **type-case** with only **one** branch, for Inr. Trying to step an Inr-case on an Inl will get stuck.

You can think of Inl-case and Inr-case as a **type-case** with a "missing else". Racket/PLAI does not let you write such a thing, but many languages with features similar to **type-case** (including Haskell, Standard ML, and OCaml) do allow this.

The **unit type** is a type with only one value: (Unit).

Small-step semantics was introduced in `lec-smallstep`, with a few more features added in `lec-smallstep-2`:

http://www.ugrad.cs.ubc.ca/~cs311/2016W1/notes/lec-smallstep.pdf

http://www.ugrad.cs.ubc.ca/~cs311/2016W1/notes/lec-smallstep-2.pdf

In the function `reduce`, implement the rules Step-sum-case-inl, Step-sum-case-inr, Step-inl-case, and Step-inr-case from Figure 2.

As in previous assignments, you may find it helpful to look at how we have implemented similar rules, such as Step-pair-case and Step-tree-case-branch.

We already wrote code in `step` that looks inside evaluation contexts for sum expressions (Inl, Inr, Sum-case, Inl-case, Inr-case).

Problem 4: Angels and Demons

This problem is about nondeterminism. ("Demonic nondeterminism" already made an appearance on the practice midterm.)

In the function `reduce`, implement the rules Step-par-left, Step-par-right, Step-choose-left, and Step-choose-right; see Figure 2.

The rules for Choose overlap, so you have a choice about which half of a Choose to step. To resolve this choice, call the Racket function `random` with argument 2. This will (pseudo-)randomly return 0 or 1. If it returns 0, do Step-choose-left; if it returns 1, do Step-choose-right.

You're now done with Choose! However, you aren't done with this problem, because the rule Step-context depends on the definition of evaluation contexts, which includes productions for Par. So:

In the function `step`, implement the two additional possibilities for $\mathcal{C}$. Similar to Choose, resolve overlapping possibilities by calling `random`. It's often a good idea to try to follow the pattern of the existing branches, and that should hold for this problem as well.

Problem 5: A Problem Has No Name

Search for "Problem 5" in `a4.rkt` to see where to write your solutions.

Problem 5a: Find Fun expressions $e1$ and $e2$ such that stepping

$$(\mathsf{Par}\ e1\ e2)$$

always converges (results in a value), but repeatedly stepping

$$(\mathsf{Choose}\ e1\ e2)$$

does not always converge. **Or,** explain why no such expressions exist.

Problem 5b: Find Fun expressions $e3$ and $e4$ such that repeatedly stepping

$$(\mathsf{Choose}\ e3\ e4)$$

always converges (results in a value), but repeatedly stepping

$$(\mathsf{Par}\ e3\ e4)$$

does not always converge. **Or,** explain why no such expressions exist.

5 Bonus problems

This bonus problem is to be handed in **on paper**, attached to your solution for Problems 1–2.

Warning! Some bonus problems may be unsolvable.

Bonus problem X1: Seems random...

In Problem 4, the rules and definitions didn't require you to call `random`; you had to do that because we told you to, in the problem text.

Since definitions in English can be ambiguous, it would be advantageous to specify this using **precise** techniques such as rules, equations (like the definition of substitution), and BNFs (like the definitions of values and evaluation contexts).

Modify the dynamic semantics of the language in this assignment so that any implementation that satisfies your semantics **must** randomly choose between alternatives (for Choose and Par).

For simplicity, you can ignore pairs, trees, and sums.

Hint: Just as we assumed some definition of addition $n1 + n2$ when we defined the dynamic semantics of Add, you can assume you have a random number generator function RNG such that

$$\mathsf{RNG}(m) \;=\; \langle m', b\rangle$$

where m is an integer *seed*, m' is the "new" seed, and b is a random bit (either 0 or 1). The idea is that, starting from some m_0, you feed the new seed back into the random number generator:

- The first time you need to randomly choose, call $\mathsf{RNG}(m_0)$, which gives $\langle m_1, b_1\rangle$. Use b_1 to decide which alternative to choose.
- The second time you need to randomly choose, call $\mathsf{RNG}(m_1)$, which gives $\langle m_2, b_2\rangle$. Use b_2 to decide which alternative to choose.
- The third time you need to randomly choose, call $\mathsf{RNG}(m_2)$, which gives $\langle m_3, b_3\rangle$...

Since RNG is a mathematical function, for any given m you will always get the same pair of m' and b back, but different m will (pseudo-randomly) give different b.

In addition to being allowed to change the rules, you can change the judgment forms themselves! For example, you could add meta-variables to $e1 \longrightarrow e2$.