CPSC 311: Definition of Programming Languages: Assignment 2: Evaluation

Joshua Dunfield
University of British Columbia

October 3, 2016

1 Logistics

You may work in teams of 2 on this assignment. (You *can* work individually; if an odd number of students do the assignment, someone will have to. But we recommend that you collaborate, mostly for your benefit, but—I'll be honest—also for ours: fewer assignments to mark.)

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`.

We are still setting up automated testing for this assignment, so please hold off on running 'handin' for now. Sorry! This way, you at least get to see what the assignment is, and you can do the first two problems (which are to be handed in on paper).

Download

`http://www.ugrad.cs.ubc.ca/~cs311/2016W1/assignments/a2.rkt`

Experiment with our `do-parse`, `do-parse-interp`, `test-parse`, `test-parse-sugar`, and `test-interp` functions, or write your own! There are a few test cases in `a2.rkt`, which are mostly commented out because an incomplete parser will cause `errors`; you'll need to write more.

*This assignment is due at 22:00 (10pm) on **Friday**, 2016–10–07.*

1.1 Handin

Most of the assignment—except the first two questions—must be turned in using the handin program.

For handin, this assignment is called `a2`.

Submit two files: `a2.rkt` and `README.txt`, with contents as described above. (If you choose to try one or more bonus problems, also include a file `bonus.rkt`). Include your name(s) at the top of each file.

Just a reminder, late assignments are not accepted, and (basically) no excuses will be entertained. So, hand in your assignments early and often!

Please avoid using DrRacket comment boxes; handin will not work properly with them. Comments using "`;`" and "`#| ... |#`" are fine.

2 Overview

In this assignment, you'll take our implementation of the "Fun" language, discussed in the last few lectures, and extend it to implement a larger "Fun++" language.

Most of this assignment will follow this recipe:

1. **Extend the concrete syntax** (EBNF grammar).
2. **Extend the abstract syntax** (**define-type**).
3. **Add evaluation rules.**
4. **Extend the parser**.
5. **Extend the interpreter** (including any necessary definitions, such as substitution).

For each of these steps, you will do an increasing share of the work, as we do a decreasing share:

1. We completely define the concrete syntax.
2. We completely define the abstract syntax.
3. We give you *all* of the evaluation rules.
4. We give you *most* of the parser.
5. We give you *most* of the *subst* function, and *some* of the interpreter itself.

In addition to extending the parser, substitution function, and interpreter, you will implement some features as *syntactic sugar*, inside the parser.

3 Syntax

Familiarize yourself with the syntax in `a2.rkt`.

For Fun++, we have added the following to the abstract syntax:

- `Pair` and `Pair-case`: A `Pair` expression contains two expressions (similar to a tuple in Python). If you have a `Pair` expression, you can see what's inside with `Pair-case`. You can think of `Pair` as something like a (very simple) form of **define-type**, and `Pair-case` as a (very simple) form of **type-case**.
- `Bfalse` and `Btrue`: Like Racket `#f` and `#t`.
- `Ite`: Like Racket `if`.
- `Rec`: Allows writing recursive functions (and other recursive things?).
- `Let*`: Like `let*` in Racket, `Let*` takes a sequence of bindings, rather than a single binding, *and* each binding can refer to identifiers bound earlier in the sequence.

We also *replaced* `Add` and `Sub` with a more general "binary operator" variant, `Binop`, whose first argument is an `Op`, defined in `a2.rkt`: An `Op` is a `Plusop` (corresponding to the old `Add`), a `Minusop` (corresponding to the old `sub`), an `Equalsop`, or a `Lessthanop`. The last two correspond to Racket `=` (on numbers) and Racket `<`.

4 Evaluation semantics

4.1 Values

As in Fun, (Num n) and (Lam x e) are values in Fun++. Additionally, the booleans (Btrue) and (Bfalse) are values, and pairs of values are values.

(The Let* expression doesn't add a new kind of data to Fun—it just allows us to bind several things at once, so it doesn't change our notion of value.)

4.2 Evaluation rules

4.2.1 Rules reused from Fun

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-let} \qquad \frac{}{(\mathsf{Id}\ x)\ \ \mathsf{free\text{-}variable\text{-}error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\mathsf{Lam}\ x\ e1) \Downarrow (\mathsf{Lam}\ x\ e1)}\ \text{Eval-lam} \qquad \frac{e1 \Downarrow (\mathsf{Lam}\ x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

4.2.2 Rule for Binop

This rule replaces Eval-add and Eval-sub. It assumes that, “elsewhere”, we have defined what $v1$ op $v2$ means for all four variants of `Op`. We give you a function `apply-binop` that implements this.

$$\frac{e1 \Downarrow v1 \qquad e2 \Downarrow v2 \qquad v1\ \mathrm{op}\ v2\ =\ v}{(\mathsf{Binop}\ \mathrm{op}\ e1\ e2) \Downarrow v}\ \text{Eval-binop}$$

4.2.3 Rules for pairs

$$\frac{e1 \Downarrow v1 \qquad e2 \Downarrow v2}{(\mathsf{Pair}\ e1\ e2) \Downarrow (\mathsf{Pair}\ v1\ v2)}\ \text{Eval-pair} \qquad \frac{ePair \Downarrow (\mathsf{Pair}\ v1\ v2) \qquad [v2/x2][v1/x1]eB \Downarrow v}{(\mathsf{Pair\text{-}case}\ \ ePair\ \ x1\ \ x2\ \ eB) \Downarrow v}\ \text{Eval-pair-case}$$

4.2.4 Rules for booleans

$$\frac{}{(\mathsf{Btrue}) \Downarrow (\mathsf{Btrue})}\ \text{Eval-btrue} \qquad \frac{}{(\mathsf{Bfalse}) \Downarrow (\mathsf{Bfalse})}\ \text{Eval-bfalse}$$

$$\frac{e \Downarrow (\mathsf{Btrue}) \qquad eThen \Downarrow v}{(\mathsf{Ite}\ e\ eThen\ eElse) \Downarrow v}\ \text{Eval-ite-true} \qquad \frac{e \Downarrow (\mathsf{Bfalse}) \qquad eElse \Downarrow v}{(\mathsf{Ite}\ e\ eThen\ eElse) \Downarrow v}\ \text{Eval-ite-false}$$

4.2.5 Rule for `Rec`

$$\frac{[(\mathsf{Rec}\ u\ e)/u]e \Downarrow v}{(\mathsf{Rec}\ u\ e) \Downarrow v}\ \text{Eval-rec}$$

4.2.6 Rules for `Let*`

$$\frac{eB \Downarrow v}{(\mathsf{Let^*}\ ()\ eB) \Downarrow v}\ \text{Eval-let*-empty} \qquad \frac{e1 \Downarrow v1 \qquad [v1/x1](\mathsf{Let^*}\ bindings\ eB) \Downarrow v}{(\mathsf{Let^*}\ ((x1\ e1)\ bindings)\ eB) \Downarrow v}\ \text{Eval-let*}$$

4.3 Substitution

Substitution for Fun++ abstract syntax

$$\begin{aligned}
[e2/x](\text{Num } n) &= (\text{Num } n)\\
[e2/x](\text{Id } x) &= e2\\
[e2/x](\text{Id } y) &= (\text{Id } y) \quad \text{if } x \neq y\\
[e2/x](\text{Lam } x\ eB) &= (\text{Lam } x\ eB)\\
[e2/x](\text{Lam } y\ eB) &= (\text{Lam } y\ [e2/x]eB)\\
&\quad \text{if } x \neq y\\
[e2/x](\text{App } eFun\ eArg) &= (\text{App } [e2/x]eFun\ [e2/x]eArg)\\
[e2/x](\text{Binop op } eL\ eR) &= (\text{Binop op } [e2/x]eL\ [e2/x]eR)\\
[e2/x](\text{Pair } eL\ eR) &= (\text{Pair } [e2/x]eL\ [e2/x]eR)\\
[e2/x](\text{Bfalse}) &= (\text{Bfalse})\\
[e2/x](\text{Btrue}) &= (\text{Btrue})\\
[e2/x](\text{Ite } e\ eThen\ eElse) &= (\text{Ite } [e2/x]e\ [e2/x]eThen\ [e2/x]eElse)\\
[e2/x](\text{Let } x\ e\ eB) &= (\text{Let } x\ [e2/x]e\ eB)\\
[e2/x](\text{Let } y\ e\ eB) &= (\text{Let } y\ [e2/x]e\ [e2/x]eB)\\
&\quad \text{if } x \neq y\\
[e2/x](\text{Let* } ()\ eB) &= (\text{Let* } ()\ [e2/x]eB)\\
[e2/x](\text{Let* } ((x\ e)\ bindings)\ eB) &= (\text{Let* } ((x\ [e2/x]e)\ bindings)\ eB)\\
[e2/x](\text{Let* } ((y\ e)\ bindings)\ eB) &= (\text{Let* } ((y\ [e2/x]e)\ bindings')\ eB')\\
&\quad \text{if } x \neq y\\
&\quad \text{and } [e2/x](\text{Let* } bindings\ eB) = (\text{Let* } bindings'\ eB')\\
[e2/x](\text{Pair-case } e\ x1\ x2\ eB) &= (\text{Pair-case } [e2/x]e\ x1\ x2\ eB)\\
&\quad \text{if } x = x1 \text{ or } x = x2\\
[e2/x](\text{Pair-case } e\ x1\ x2\ eB) &= (\text{Pair-case } [e2/x]e\ x1\ x2\ [e2/x]eB)\\
&\quad \text{if } x \neq x1 \text{ and } x \neq x2\\
[e2/x](\text{Rec } x\ eB) &= (\text{Rec } x\ eB)\\
[e2/x](\text{Rec } y\ eB) &= (\text{Rec } y\ [e2/x]eB)\\
&\quad \text{if } x \neq y
\end{aligned}$$

5 Problems—CPSC 311 2016W1 a2

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.

  For example, if a rule has two premises, you should probably have two `let`-bindings, and use the same names for the results of evaluation (for example, $v1$ in Eval-pair).

  The goal is to make it easy to look at your code alongside the rules, and be confident that your code follows the rules—even if you don't fully understand how the rules work.

  Thus, even if you have a great idea for a better, faster way of interpreting Fun++ programs, *you can't use that idea on this assignment*, because that would make it harder to convince a reader that your code really does implement the rules. For instance, there are much more efficient ways to interpret function application than substitution, but your code for this assignment must use substitution anyway.

- You must write test cases for the features you're implementing; you should also write test cases for code that we gave you, to improve your understanding of concepts such as evaluation rules. Implementations of programming languages are especially tricky to debug; if your interpreter returns a surprising value, or raises an error when it shouldn't, or *doesn't* raise an error when it should, it could be

  - a bug in your interpreter (or one of the functions it calls, such as `subst`), or
  - a bug in the test program (for example, you're not sure what a Fun++ program *should* do, and you wrote something that you didn't intend), or
  - a bug in your understanding of the rules.

  It could also be a problem in our specification, such as a typo in this handout. If you believe you have found such a problem, let us know; if it is a problem, we will announce a correction, and will endeavour not to deduct marks for implementing the mistaken semantics.

Having just made you read all of that, the first two problems are not coding problems. Turn them in on paper (exact method to be determined; handing in at the start of lecture is acceptable). You can print this sheet and write on it, or write out your answers on a separate sheet. **Be sure to write your name, student number, and CS userid.**

Name:		Student ID#:							
CS userid:									

Problem 1: Derivations

Following the "method of hope" from `lec-functions.pdf`, complete the following two evaluation derivations. The rules are given above in Section 4.

1(a).

I've already started this one. I searched in the evaluation rules for a rule whose conclusion looked like (Let ... ...) ⇓ ..., and I found Eval-let, so I matched up the meta-variables $e1$ and $e2$ to (Num 4) and (Lam q (Id s)), respectively.

The dashed line on the top, to the right of the =, is for you to write in the result of substituting something (not known yet, since you haven't derived the first premise (Num 4) ⇓ ···) for s in (Lam q (Id s), but **before** evaluating (since it's to the *left* of ⇓).

As you write horizontal lines, write the rule name next to it, as I did with Eval-let.

$$\dfrac{(\mathsf{Num}\ 4) \Downarrow \_\_\_\_\_\_ \qquad \overset{=\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_}{\big[\_\_\_\_\_\_ /s\big](\mathsf{Lam}\ q\ (\mathsf{Id}\ s))} \Downarrow \_\_\_\_\_\_\_\_}{\big(\mathsf{Let}\ \underbrace{s}_{x}\ \underbrace{(\mathsf{Num}\ 4)}_{e1}\ \underbrace{(\mathsf{Lam}\ q\ (\mathsf{Id}\ s))}_{e2}\big) \Downarrow \_\_\_\_\_\_\_\_\_\_\_\_}\ \text{Eval-let}$$

1(b). Hint: Do 1(a) first. The lecture notes mention a convention of writing a checkmark above a judgment if you've already derived it elsewhere; do that if possible.

$$\dfrac{}{\Big(\mathsf{App}\ \big(\mathsf{Let}\ s\ (\mathsf{Num}\ 4)\ (\mathsf{Lam}\ q\ (\mathsf{Id}\ s))\big)\ \big(\mathsf{Num}\ 9\big)\Big) \Downarrow \_\_\_\_\_\_\_\_\_\_\_\_}\ \_\_\_\_\_\_\_\_$$

↑write the rule name↑

Problem 2 is on the next page.

Problem 2: From code to a rule

This problem is "backwards" from the rest of the assignment: instead of writing code that follows a rule, you'll write a rule that follows the code. Suppose we added a language feature called Map-pair, and wrote this branch of `interp`:

```
(define-type E
      ; ... variants of E ...
      [Map-pair (function-exp E?) (pair-exp E?)]
      )

(define (interp e)   ; interp : E −> E
  (type-case E e
      [Num (n)  (Num n)]

      ; ... branches for other variants of E ...

      [Map-pair (eFun ePair)
                (let ([vFun (interp eFun)]
                      [vPair (interp ePair)])
                     (type-case E vFun
                       [Lam (x body)
                            (type-case E vPair
                              [Pair (v1 v2)
                                    (let ([fv1 (interp (subst v1 x body))]
                                          [fv2 (interp (subst v2 x body))])
                                      (Pair fv1 fv2))]
                              [else (error "not a pair")])]
                       [else (error "not a lam")]))
       ]
```

Write premises to make the rule correspond to the code above.

$$\frac{}{(\text{Map-pair}\ eFun\ ePair) \Downarrow \ \underline{\qquad\qquad\qquad\qquad}}\ \text{Eval-map-pair}$$

Problem 3: Parsing

Extend our parser, `parse`, to handle the new binary operators = and <.

One of the reasons we're introducing the `Binop` variant is to reduce redundant code, but *someone* seems to have left redundant code in the parser for + and -, so you should improve on that!

We already handle parsing of `Rec`, `Pair`, `Pair-case`, `Bfalse`, `Btrue`, `Let*` and `Ite`, so you're free to do Problems 4 and 6 before you finish this problem.

Problem 4: Pairs

In the interpreter function, `interp`, write code for `Pair` and `Pair-case`. Try to follow the evaluation rules closely.

We have already extended the substitution function, `subst`, to handle pairs.

Problem 5: Syntactic sugar

The BNF grammar in `a2.rkt` includes two productions, for `{Fst <E>}` and `{Snd <E>}`, that you should handle differently from the others.

Extend the parser to recognize these productions, but treat them as *syntactic sugar*: when your parser sees `{Fst <E>}` or `{Snd <E>}`, it should actually return a Pair-case. Design the Pair-case so that it will evaluate to the value you would get if we added the following two rules:

$$\frac{e \Downarrow (\mathsf{Pair}\ v1\ v2)}{(\mathsf{Fst}\ e) \Downarrow v1} \qquad\qquad \frac{e \Downarrow (\mathsf{Pair}\ v1\ v2)}{(\mathsf{Snd}\ e) \Downarrow v2}$$

If you're not sure what Pair-case your parser should return, you might want to try to write a derivation tree for Pair-case, leaving the *body* of the Pair-case blank, and seeing what the body needs to be to make the Pair-case evaluate to the same value as the first rule above.

(Once you handle `Fst`, it should be clear how to handle `Snd`.)

Because this is syntactic sugar, you don't need to (and must not) do anything in `subst` or `interp`!

Problem 6: `Ite`

- Add cases for Bfalse, Btrue, and Ite to the `subst` function, following the definition above.
- In the interpreter function, `interp`, write code for `Ite`. Your code will need to figure out which of the two `Ite` rules to apply.

Problem 7: `Let*`

In the interpreter function, `interp`, write code for `Let*`.

Hint: Look at the `Let*` branch of `subst`.

Do **not** treat `Let*` as syntactic sugar! But, after you've done it "for real" in this problem, you're welcome to try bonus problem X1.

Bonus problems on the next page. . .

6 Bonus problems

If you choose to complete a bonus, submit a separate `bonus.rkt` file with your bonus work in it. (If your bonus work passes all of our handin tests, this can just be a copy of your regular submission, but it's conceivable that your bonus will violate our tests! Plus, this way you can solve the main assignment, set that work aside, and then try for the bonus without endangering your main grade.) Also note that we reserve the right to summarily give no credit to ill-documented or ill-tested bonus submissions.

Warning! Some bonus problems might be unsolvable.

Note: All three bonus problems involve modifying `parse`; if you do more than one, you can do it in a single parser, or write multiple parsers if necessary. (Don't change the parser that's used for the main problems! Submit your modified `parse` in `bonus.rkt` only.)

Bonus problem X1: `Let*` as syntactic sugar

Write a different parser that treats `Let*` as syntactic sugar by turning `Let*` into a nested sequence of `Let`s.

Also, add a note to your `README.txt` file that says whether you think this is a better or worse way to add `Let*` to the language than what you had to do in Problem 7, and explains why. (A couple of sentences are fine, but if you want to say more, feel free.)

Bonus problem X2: `Let` as syntactic sugar

Same as X1, but treat `Let` itself as syntactic sugar. Hint: `Let` evaluates one argument, and substitutes into another. What does `App` do?

Also, add a note to your `README.txt` file that says whether you think this is a better or worse way to add `Let` to the language than to implement it the way we did.

Bonus problem X3: integers as syntactic sugar

Like X1, but treat `Num` and `+` as syntactic sugar! Is that even possible? (See the warning above!) You only need to support integers (*not* floats, rationals, etc.) and you can ignore `-`, `=` and `<`, but your parser must not produce any `Num` expressions whatsoever. You have to build up integers from... something else. (Extra-special bonus: do it without pairs.)