Assignment 2 Q&A

15-312 Foundations of Programming Languages
Kevin Watkins ⟨kw@cmu.edu⟩

February 7, 2005

Q1. Student A writes:

> For [the] evaluator, once we call `evalPrimop`, do we know that the returned expression is [a] value? Or should we check if it is, and step further when it's not?

A1. First, looking at the rule *OpVals*,

$$\frac{\text{(by primop } o\text{)}}{o(v_1, ..., v_n) \mapsto v}\ \mathit{OpVals},$$

we see that the result must be a value. Although it wasn't explicitly stated, you can assume that the entire operational semantics, and the rule *OpVals* in particular, is deterministic. So if $e \mapsto v$ by rule *OpVals* for some v, then that v is unique.

With this in mind, regarding `evalPrimop`, you can assume that it satisfies the following specification:

> If $e \mapsto v$ by rule *OpVals*, then `evalPrimop`$(e) = v$.
>
> If e does not step to v by rule *OpVals* for any such v, then `evalPrimop`$(e) =$ `raise PrimopStuck`.

Knowing that `evalPrimop` satisfies this specification should allow you to use it in a correct way within your implementation of `step`.

Q2. Student A continues:

> Also, for compatibility rules, for example `if` e `then` e_1 `else` e_2 `end`, I step on e if it's not already `true` or `false`. So it looks like:
>
> ```
> | step (If(e, e1, e2)) = step (If (step e, e1, e2))
> ```
>
> But how do I know that `(step e)` will evaluate to `true` or `false`, as it may require multiple steps on e alone to evaluate to [a] value?

A2. As stated in the assignment, you must implement `step` in a clear, correct way, such that whenever e is a *closed* de Bruijn term, it satisfies the specification

> If $e \mapsto e'$ for some (unique) e', then `step`$(e) = e'$.
>
> If $e \mapsto e'$ does not hold for any e', then `step`$(e) =$ `raise NoStep`.

In deciding how to implement `step` for e_{if} = `if` e `then` e_1 `else` e_2 `end`, you should consider the specification carefully, look at the possibilities for $e_{if} \mapsto e'_{if}$, and write code to implement that case of `step` accordingly. It may be that the code in Student A's question satisfies the specification, or it may be that it doesn't. Note that the specification completely determines the behavior of `step` on closed de Bruijn terms because the relation $e \mapsto e'$ is deterministic.

Q3. Student A continues:

> Also for [the implementation of `step` on] `Primop`(...), is there a simple way to know if we should use [rule] *OpArg* or [rule] *OpVals*? I'm using some list operation[s] to determine [... coding strategy omitted ...]. I was wondering if there's a cleaner way to do it.

A3. The model solution I wrote uses list operations in this case. In accordance with our grading criteria, I tried to write correct code that was as clear as possible. You should strive to do the same. It may be that there is a clearer solution using some technique other than list operations; if you think of one, you should use it. The one thing you shouldn't do is make the code for this case *less* clear in a misguided attempt to make it more efficient or something.

Q4. Student A writes again:

> When we're stepping on `Int`(3), since this is already a value, there is no step to take. Does this mean that the evaluator should raise [the] `NoStep` [exception]? This would disallow an expression like 3, which is valid in MinML.

A4. I am confident that you can answer this question for yourself with a little careful thought about the implications of the specification for `step` given in the assignment statement.