Assignment 2

Joshua Dunfield
due: Tuesday, 5 February 2019

Name: Estimated time spent:

Submit your modified a2.rkt by email (joshuad@cs.queensu.ca) as a2-yourname.rkt.
Please fill in the “Estimated time spent” (above). Your answer will not change your mark, but helps me to design reasonable assignments.

1 Language Extension

Consider the following language, defined by a grammar and a big-step evaluation judgment. The big-step evaluation given is incomplete, in the informal sense that it has no rules saying how to evaluate \((\text{Abs } e)\).

integers \ n
values \ v ::= \ n
expressions \ e ::= \ n
| \ (+ \ e_1 \ e_2)
| \ (\text{Abs } e)

\(e \downarrow v\) expression \ e \) evaluates to value \ v

\[ \begin{align*}
    n \downarrow n & \quad \text{eval-const} & & \quad \text{eval-add} \\
    e_1 \downarrow n_1 & & e_2 \downarrow n_2
\end{align*} \]

Question 1(a). Roughly following the structure of eval-add, design a rule “eval-abs” such that \((\text{Abs } e)\) will compute the absolute value of \(e\). Similar to how we used the standard notation for addition, \(n_1 + n_2\), in eval-add, you may use the notation \(|n|\) for the absolute value of \(n\).

\[ \begin{align*}
    (\text{Abs } e) \downarrow & \quad \text{eval-abs}
\end{align*} \]

Question 1(b). Extend the Racket code a2.rkt (which is nearly identical to the expr1.rkt file that I showed in lecture) to support the construct \((\text{Abs } e)\):

(i) Add a “variant” (also called a “constructor”) for \((\text{Abs } e)\) to the \texttt{define-type} declaration.

This will cause Racket to yell at you, because \texttt{EXPR} will then have three variants instead of two, which makes the \texttt{type-case} in \texttt{big-step} become incomplete. The simplest fix (until you extend \texttt{big-step}, below) is to add an \texttt{else} branch to the \texttt{type-case} that uses \texttt{error}.

(ii) Extend the procedure \texttt{parse} so that it accepts S-expressions that match the production \((\text{Abs } e)\) in our grammar, and returns your Abs variant. Write at least two test cases for \texttt{parse} that involve absolute value.
(iii) Extend the procedure big-step to support absolute value, following your rule eval-abs. Note that the Racket standard library has a procedure abs. Write at least three test cases for big-step that involve absolute value.
§1 Language Extension

2 Proof techniques

These questions are not about complete proofs. In some of the questions, the conjecture is not even true, or you have not been given enough information to do a complete proof. Instead, they ask you to make progress on several different proofs by using a specific proof technique.

In all of these questions, the grammar of expressions is the extended grammar (Section ??), and the system of rules deriving $e \Downarrow v$ includes the three rules eval-const, eval-add, eval-abs.

**Question 2(a).** Using the extended grammar of expressions (Section ??), list the cases produced by case analysis on $e$. The cases must correspond to the grammar. Do not attempt to complete the cases to show $e' = e''$.

**Conjecture.**

For all expressions $e$, $e'$ and $e''$,

if $e \rightarrow e'$ and $e \rightarrow e''$

then $e' = e''$.

**Proof.** Consider cases of $e$.

- Case

**Question 2(b).** In this question, your goal is to derive

$$(+ 0 (+ e_{21} e_{22})) \Downarrow 0$$

The following are given. Use equations (and the fact that $0 + 0 = 0$) and apply the rules eval-const, eval-add to derive the goal.

- $e_{21} \Downarrow n_{21}$  Given
- $e_{22} \Downarrow n_{22}$  Given
- $n_{21} = 0$  Given
- $n_{22} = 0$  Given
§2 Proof techniques

**Question 2(c).** In this question, use *inversion*: write down all the facts given by inverting on rule eval-abs. (I can’t give you a specific goal because what you get depends on your rule, eval-abs.)

\[(\text{Abs } e_1) \downarrow n_1 \quad \text{Given}\]

**Question 2(d).** This question uses notation you have never seen, but it can still be answered.

**Conjecture.**
For all \(C, M_1, M_2\) and \(D_1\) such that \(D_1\) derives \(M_1 \rightarrow_R M_2\),
there exists \(D_2\) such that \(D_2\) derives \(C[M_1] \rightarrow_R C[M_2]\).

- Suppose that I suggest you use structural induction on \(D_1\). Write the appropriate induction hypothesis:

- Suppose that I suggest you use structural induction on \(M_1\). Write the appropriate induction hypothesis (using the subexpression ordering \(\prec\) for \(M_1\)): 
§2 Proof techniques

§3 Typing

\[
\text{types } \quad \Lambda ::= \text{int}
\]

<table>
<thead>
<tr>
<th>expression e has type ( \Lambda )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e : \Lambda )</td>
</tr>
<tr>
<td>type-const ( n : \text{int} )</td>
</tr>
<tr>
<td>type-abs ( (\text{Abs } e) : \text{int} )</td>
</tr>
<tr>
<td>type-add ( e_1 : \text{int} ), ( e_2 : \text{int} )</td>
</tr>
<tr>
<td>( (+ e_1 e_2) : \text{int} )</td>
</tr>
</tbody>
</table>

This is not a terribly interesting type system: every possible expression has the same type, \( \text{int} \).

Prove the following conjecture:

**Conjecture 3.1.**

*For all expressions \( e \), it is the case that \( e : \text{int} \).*

**Proof.** By structural induction on \( e \). [The only thing given is \( e \); we are trying to construct the derivation of \( e : \text{int} \), but it doesn’t exist yet. So we have to induct on \( e \).

**Induction hypothesis:** For all expressions \( e' \) such that \( e' \prec e \), it is the case that \( e' : \text{int} \).