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 (Abs e).

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

\[
e \Downarrow v \text{ expression } e \text{ evaluates to value } v
\]

\[
\begin{array}{c}
\text{n } \Downarrow n \\
\hline
\text{eval-const}
\end{array}
\quad
\begin{array}{c}
\text{e_1 } \Downarrow n_1 \\
\text{e_2 } \Downarrow n_2 \\
\hline
\text{eval-add}
\end{array}
\quad
\begin{array}{c}
\text{(+ e_1 e_2) } \Downarrow n_1 + n_2
\end{array}
\]

Question 1(a). Roughly following the structure of eval-add, design a rule “eval-abs” such that (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.

\[
e \Downarrow n \quad (\text{Abs } e) \Downarrow |n| \text{ eval-abs}
\]

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 (Abs e):

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

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

(ii) Extend the procedure parse so that it accepts S-expressions that match the production (Abs e) in our grammar, and returns your Abs variant. Write at least two test cases for 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.
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 1), 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 1), 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 $e = n$
- Case $e = (+ e_1 e_2)$
- Case $e = (\text{Abs } e_1)$

[Writing $e = (\text{Abs } e)$ doesn’t make sense: it’s like saying that $x + 1 = x$. The name $e$ can’t be used for two different things at the same time. We need to rename the $e$ from the grammar so it doesn’t clash with the $e$ mentioned in the conjecture.]

□
§2 Proof techniques

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.

\[e_{21} \Downarrow n_{21} \quad \text{Given}\]
\[e_{22} \Downarrow n_{22} \quad \text{Given}\]
\[n_{21} = 0 \quad \text{Given}\]
\[n_{22} = 0 \quad \text{Given}\]

\[
\begin{array}{|c|c|}
\hline
0 \Downarrow 0 & \text{By eval-const} \\
(+ e_{21} e_{22}) \Downarrow n_{21} + n_{22} & \text{By eval-add} \\
(+ 0 (+ e_{21} e_{22})) \Downarrow 0 + (n_{21} + n_{22}) & \text{By eval-add} \\
\hline
n_{21} = 0 & \text{Above} \\
n_{22} = 0 & \text{Above} \\
0 + (n_{21} + n_{22}) = 0 & \text{By arithmetic} \\
(+ 0 (+ e_{21} e_{22})) \Downarrow 0 & \text{By above equation} [0 + (n_{21} + n_{22}) = 0] \\
\hline
\end{array}
\]
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.)

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

\[n_1 = |n| \quad \text{By inversion on rule eval-abs}\]

\[e_1 \Downarrow n \quad "\]

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 \(M_1 \vdash_R M_2\), there exists \(D_2\) such that \(D_2\) derives \(C[M_1] \vdash_R C[M_2]\).

- Suppose that I suggest you use structural induction on \(D_1\). Write the appropriate induction hypothesis:
  [Refer to the three-step process given in a1sol.pdf.]

  For all \(C', M'_1, M'_2\) and \(D'_1\) such that \(D'_1 \prec D_1\) and \(D'_1\) derives \(M'_1 \vdash_R M'_2\), there exists \(D'_2\) such that \(D'_2\) derives \(C'[M'_1] \vdash_R C'[M'_2]\).

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

  For all \(C', M'_1, M'_2\) and \(D'_1\) such that \(M'_1 \prec M\) and \(D'_1\) derives \(M'_1 \vdash_R M'_2\), there exists \(D'_2\) such that \(D'_2\) derives \(C'[M'_1] \vdash_R C'[M'_2]\).

  [The derivation names don’t actually matter here, so they could safely be omitted:
  For all \(C', M'_1\) and \(M'_2\) such that \(M'_1 \prec M\) and \(M'_1 \vdash_R M'_2\), \(C'[M'_1] \vdash_R C'[M'_2]\).]
3 Typing

\[ e : A \] expression e has type A

\[
\begin{align*}
\text{type-const} & : n : \text{int} \\
\text{type-abs} & : (\text{Abs } e) : \text{int} \\
\text{type-add} & : (e_1 + e_2) : \text{int}
\end{align*}
\]

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

**Consider cases of e.**

- **Case: \( e = n \)**
  - \( n : \text{int} \) By applying rule type-const
  - \( e : \text{int} \) By above equation \([e = n]\)

- **Case: \( e = (\text{Abs } e_1) \)**
  - \( e_1 \prec e \) [By above equation \( e = (\text{Abs } e_1) \)]
  - \( e_1 : \text{int} \) By IH [with \( e_1 \) as \( e' \)]
  - \( (\text{Abs } e_1) : \text{int} \) By applying rule type-abs
  - \( e : \text{int} \) By above equation \([e = (\text{Abs } e_1)]\)

- **Case: \( e = (e_1 + e_2) \)**
  - \( e_1 \prec e \) [By above equation \( e = (e_1 + e_2) \)]
  - \( e_1 : \text{int} \) By IH [with \( e_1 \) as \( e' \)]
  - \( e_2 \prec e \) [By above equation \( e = (e_1 + e_2) \)]
  - \( e_2 : \text{int} \) By IH [with \( e_2 \) as \( e' \)]
  - \( (e_1 + e_2) : \text{int} \) By applying rule type-add
  - \( e : \text{int} \) By above equation \([e = (e_1 + e_2)]\)