Problem Set 3 Solutions

Reminder: **QUIZ 1** in class on Monday, 7 October.

Problem 1

(a) `v`, `x` are free in the expression at the end of the body, so are needed. The definition of `w` is not immediate, so `w` is needed, and `w` needs `u`. This leaves `y` and `z` available for garbage collection at the first step.

(b) All the definitions get garbage collected by the end, even with the Lambda Body Restriction. (To see how, run the body using the `smeval` procedure from the rewriting interpreter described in Handout 2.)

(c) —so they certainly all get garbage collected when the restriction is relaxed.

2(a)

$$
\begin{aligned}
\langle\text{simple expression}\rangle &::= \langle\text{self-evaluating}\rangle \mid \langle\text{primitive procedure variable}\rangle \\
&\quad\ \mid \langle\text{combination}\rangle \mid \langle\text{if}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{self-evaluating}\rangle &::= \langle\text{numeral}\rangle \mid \langle\text{boolean}\rangle \\
\langle\text{numeral}\rangle &::= \texttt{0} \mid \texttt{1} \mid \texttt{-1} \mid \texttt{2} \mid \texttt{-2} \mid \ldots \\
\langle\text{boolean}\rangle &::= \texttt{\#t} \mid \texttt{\#f} \\
\langle\text{primitive procedure variable}\rangle &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{=} \mid \texttt{<}
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{combination}\rangle &::= \texttt{(}\, \langle\text{operator}\rangle\, \langle\text{operand}_1\rangle\, \langle\text{operand}_2\rangle\, \texttt{)} \\
\langle\text{operator}\rangle &::= \langle\text{simple expression}\rangle \\
\langle\text{operand}\rangle &::= \langle\text{simple expression}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{if}\rangle &:= \texttt{(if}\ \langle\text{test}\rangle\, \langle\text{consequent}\rangle\, \langle\text{alternative}\rangle\, \texttt{)} \\
\langle\text{test}\rangle &::= \langle\text{simple expression}\rangle \\
\langle\text{consequent}\rangle &::= \langle\text{simple expression}\rangle \\
\langle\text{alternative}\rangle &::= \langle\text{simple expression}\rangle
\end{aligned}
$$

2(b)

Definition. The *meaning*, $[\![s]\!]$, of a simple expression, s, is defined by structural induction on $\langle$simple expression$\rangle$'s as follows:

- $$\begin{aligned} [\![\mathbf{n}]\!] &= n \qquad \text{if } \mathbf{n} \text{ is the numeral for the number } n, \\ [\![\texttt{\#t}]\!] &= \mathit{true}, \\ [\![\texttt{\#f}]\!] &= \mathit{false}. \end{aligned}$$

- $[\![op]\!] = \circ$ when op is **+**, **-**, *****, **=**, or **<**, and $\circ$ is the binary arithmetic operation $+, -, *, =,$ or $<$, respectively. (We treat the binary relations $=$ and $<$ on numbers as binary operations from numbers to Boolean values).

- $[\![(op\ s_1\ s_2)]\!] = [\![s_1]\!] \circ [\![s_2]\!]$, providing $[\![s_1]\!]$ and $[\![s_2]\!]$ are both defined to be numbers, op is a $\langle$primitive procedure variable$\rangle$ and $[\![op]\!] = \circ$.

- $[\![(\texttt{if}\ s_1\ s_2\ s_3)]\!] = \begin{cases} [\![s_2]\!] & \text{if } [\![s_1]\!] = \mathit{true} \text{ and } [\![s_2]\!] \text{ is defined,} \\ [\![s_3]\!] & \text{if } [\![s_1]\!] = \mathit{false} \text{ and } [\![s_3]\!] \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$

- if s is not of one of the forms above, then $[\![s]\!]$ is undefined.

2(c)

If $s_1 \longrightarrow s_2$, then s_2 is a simple expression, and either both of s_1 and s_2 are meaningless, or else $[\![s_1]\!] = [\![s_2]\!]$.

Proof: By structural induction on s_1.

s_1 is $\langle$self-evaluating$\rangle$ or a $\langle$primitive procedure variable$\rangle$: No rule applies to s_1, so there is nothing to prove.

s_1 is of the form $(op\ s_3\ s_4)$: There are three subcases to consider.

1. s_1 is itself the redex of a rule.
   This is possible only if op is a $\langle$procedure variable$\rangle$ and s_3, s_4 are both numerals, in which case s_2 must be $\langle$self-evaluating$\rangle$ and $s_1 \longrightarrow s_2$ by a primitive application rule. Then $[\![s_1]\!] = [\![s_2]\!]$ since the left and righthand sides of primitive application rules obviously have the same meaning.

2. s_2 is of the form (op s_3' s_4) where $s_3 \longrightarrow s_3'$.

   If $[\![s_3]\!]$ is undefined, then by definition, so is $[\![s_1]\!]$. But then by induction, $[\![s_3']\!]$ is also undefined, and by definition, so is $[\![s_2]\!]$. Thus both $[\![s_1]\!]$ and $[\![s_2]\!]$ are undefined.

   If $[\![s_3]\!]$ is defined, then by induction, $[\![s_3]\!] = [\![s_3']\!]$. If $[\![s_3]\!]$ is not a number, then both $[\![s_1]\!]$ and $[\![s_2]\!]$ are undefined by definition. If $[\![s_3]\!]$ is a number, then

   $$[\![s_1]\!] = [\![s_3]\!] \circ [\![s_4]\!] = [\![s_3']\!] \circ [\![s4]\!] = [\![s_2]\!],$$

   where $[\![op]\!] = \circ$.

3. s_2 is of the form (op s_3 s_4') where $s_4 \longrightarrow s_4'$. As in the previous subcase.

s_1 is of the form (`if` s_3 s_4 s_5): This time there are four subcases.

1. s_1 is itself the redex of a rule. This is possible if s_3 is `#t` and s_2 is s_4, in which case the meanings of s_1 and s_2 are both undefined or are equal by the definition of $[\![s_1]\!]$. The only other way s_1 can be a redex is if s_3 is `#f`, and we similarly conclude that $[\![s_1]\!]$ and $[\![s_2]\!]$ are both undefined or equal.

2. s_2 is of the form (`if` s_3' s_4 s_5) where $s_3 \longrightarrow s_3'$.

   If $[\![s_3]\!]$ is undefined, then by definition, so is $[\![s_1]\!]$. But then by induction, $[\![s_3']\!]$ is also undefined, and by definition, so is $[\![s_2]\!]$. Thus both $[\![s_1]\!]$ and $[\![s_2]\!]$ are undefined.

   If $[\![s_3]\!]$ is defined, then by induction, $[\![s_3]\!] = [\![s_3']\!]$. If $[\![s_3]\!]$ is not a Boolean value, then both $[\![s_1]\!]$ and $[\![s_2]\!]$ are undefined by definition. Finally, if $[\![s_3]\!]$ is a Boolean, then by definition of the meaning of a conditional, the meanings of s_1 and s_2 are both undefined or are equal.

3. s_2 is of the form (`if` s_3 s_4' s_5) where $s_4 \longrightarrow s_4'$. Similar to the previous case.

4. s_2 is of the form (`if` s_3 s_4 s_5') where $s_5 \longrightarrow s_5'$. Similar to the previous case.

■

2(d)

Define the *unit* expressions to be ⟨self-evaluating⟩ or ⟨primitive procedure variable⟩'s. We first prove that the only terminated simple expressions whose meaning is defined are unit expressions. In other words, if s_1 is a meaningful simple expression which is *not* a unit, then s_1 will contain a redex of a rewriting rule. (The redex may be s_1 itself.) The proof is by structural induction on s_1.

If s_1 is a unit, there is nothing to prove.

Suppose s_1 is of the form `(if` s_3 s_4 s_5`)`. In this case, since s_1 is meaningful, $[\![s_3]\!]$ must be defined to be Boolean value. If s_3 is not a unit, then by induction, s_3 contains a redex, hence so does s_1. Otherwise, s_3 must be `#t` or `#f`, since these are only units with Boolean values, and in that case s_1 itself is a redex.

The final case, when s_1 is a combination, follows similarly.

Now by 2(c), rewriting preserves simplicity of expressions. Moreover, the only rewrite rules applicable to simple expressions are primitive applications and conditional rules, and these rules always *decrease* the size of the rewritten expression. So it is impossible to successively rewrite a simple expression more times than the length (number of syntactic tokens) of the expression. Hence, every simple expression eventually rewrites to a terminated expression.

Also, by 2(c), rewriting preserves meaning of simple expressions. So a meaningful simple expression must eventually rewrite to a unit with the same meaning. Since distinct units have different meanings, we conclude that every meaningful simple expression must eventually rewrite to a *unique* unit.

2(e)

We claim that simple expressions satisfy the *strong diamond property*: If an expression s can rewrite in one step to s_1, in symbols, $s \longrightarrow s_1$; and also $s \longrightarrow s_2$, then there is an s_3 such that s_1 and s_2 can both be rewritten to s_3 in *at most* one step. We saw in class that strong diamond implies unique termination.

We prove the strong diamond property by structural induction on s. Suppose s $s \longrightarrow s_1$ and $s \longrightarrow s_2$ with $s_1 \neq s_2$. Since a rewrite rule applies to s, it must be a combination or a conditional.

If s is a conditional, it is of the form `(if` s_4 s_5 s_6`)`. Now we have three subcases.

1. s_1 and s_2 arise from applying rules to redexes in different components of the conditional. For example, $s_1 =$ `(if` s'_4 s_5 s_6`)` because the test, s_4 rewrites to s'_4, and $s_2 =$ `(if` s_4 s_5 s'_6`)` because the alternate branch, s_6 rewrites to s'_6. Then we can let s_3 be `(if` s'_4 s_5 s'_6`)`.

2. s_1 and s_2 arise from applying rules to redexes in the same component of the conditional. For example, $s_1 =$ `(if` s'_4 s_5 s_6`)` because the test, s_4 rewrites to s'_4, and $s_2 =$ `(if` s''_4 s_5 s_6`)` because the test also rewrites to s''_4. Then by induction there is an s_7 such that s'_4 and s''_4 each rewrite to s_7 in at most step. Then we can let s_3 be `(if` s_7 s_5 $s6$`)`.

3. s_1 arises from applying a rule with all of s as the redex. This means the test, must be a $\langle$boolean$\rangle$, say `#t`, and s is of the form `(if #t` s_1 s_6`)`. Also, s_2 must arise from applying a rule to a branch of s. If s_2 arose from rewriting a redex

in the other branch, then s_2 is of the form `(if #t` s_1 s'_6`)` where $s_6 \longrightarrow s'_6$. So we let s_3 be s_1 since $s_2 \longrightarrow s_1$. If s_2 arose from a redex in the branch s_1, then s_2 is `(if #t` s'_1 s_6`)` where $s_1 \longrightarrow s'_1$. Then let s_3 be s'_1.

3(a)

The function $replace(B, x, y)$ is defined by structural induction on the Scheme body B.

Suppose B is of the form

`(define` v_1 V_1`)` ... `(define` v_n V_n`)` M,

where M is an expression. If $x \in \{v_1, \ldots, v_n\}$, then $replace(B, x, y) = B$, otherwise

$replace(B, x, y) =$ `(define` v_1 V'_1`)` ... `(define` v_n V'_n`)` M'

where $V'_i = replace(V_i, x, y)$, and $M' = replace(M, x, y)$.

Suppose B is of the form `(`M_1 ... M_n`)`. Then $replace(B, x, y) =$ `(`M'_1 ... M'_n`)`, where $M'_i = replace(M_i, x, y)$.

The other cases are similar.

3(b)

The decision procedure for α-equivalence is defined by structural induction on B_1.

The case for B_1 a lambda expression was given in the hint for expressions with two formals; the extension to an arbitrary number of formals is straightforward.

Suppose B_1 of the form

`(define` v_1 V_1`)` ... `(define` v_n V_n`)` M.

Let $u_1, \ldots, u_n$ be distinct "fresh" variables, i.e., variables which do not occur in B_1, B_2. Let

$$r_1(N) = replace(replace(\ldots(replace(N, v_1, u_1), v_2, u_2)\ldots v_{n-1}, u_{n-1}), v_n, u_n),$$

Then B_1 and B_2 are $=_\alpha$ iff B_2 is of the form

`(define` v'_1 V'_1`)` ... `(define` v'_n V'_n`)` M',

$r_1(M) =_\alpha r_2(M')$ and $r_1(V_i) =_\alpha r_2(V'_i)$ for $i = 1, \ldots, n$, where

$$r_2(N) = replace(replace(\ldots(replace(N, v'_1, u_1), v'_2, u_2)\ldots v'_{n-1}, u_{n-1}), v'_n, u_n).$$

The other cases are similar.

Problem 5

By these rules,

```
(equal? (lambda (x) (+ 1 2)) (lambda (x) (+ 1 2))) ⟶ #t.
```

But also,

```
(equal? (lambda (x) (+ 1 2)) (lambda (x) (+ 1 2)))
    ⟶ (equal? (lambda (x) (+ 1 2)) (lambda (x) 3))
    ⟶ #f
```