Computer Science & Engineering 505 – Midterm – Answer Key

November 9, 2001
Open book & notes – 50 minutes – 10 points per question
60 points total

Name: _____

Please use the back of the page if necessary for long answers.

1. Consider the following CLP($\mathcal{R}$) rules.

```
twice(2*N) :- smallodd(N).
smallodd(1).
smallodd(3).
smallodd(5).
```

   Show the simplified derivation tree for the following goal.

```
twice(A), A>5.
```

   I don't want to try and convince latex to lay out a tree, so here's a description. (We'll also put a scanned image on the web.)

   The root of the tree is the goal

$$\langle \text{twice(A)}, A > 5 \mid \text{true} \rangle$$

   From the root node there is a single branch R_1, where R_1 is the "twice" rule.

$$\langle \text{smallodd(A/2)}, A > 5 \mid \text{true} \rangle$$

   Under this there are three branches, R_2, R_3, and R_4, where R_2, R_3, and R_4 are the three "smallodd" rules.

   Here is the R_2 branch.

$$\langle \blacksquare \mid \text{false} \rangle$$

   Here is the R_3 branch:

$$\langle \square \mid A = 6 \rangle$$

   Here is the R_4 branch:

$$\langle \square \mid A = 10 \rangle$$

2. HAL includes support for CHRs (constraint handling rules). The rules for `and` in the boolean solver, as given in the HAL paper, are:

```
true(X)  \ and(X,Y,Z) <=> Y=Z
false(X) \ and(X,Y,Z) <=> false(Z)
true(Y)  \ and(X,Y,Z) <=> X=Z
false(Y) \ and(X,Y,Z) <=> false(Z)
true(Z)  \ and(X,Y,Z) <=> true(X), true(Y)
false(Z) \ and(X,Y,Z) <=> notboth(X,Y).
```

The first of these rules says that if we match these two constraints in the current constraint store:

```
true(X)
and(X,Y,Z)
```

then we remove `and(X,Y,Z)` from the store and add `Y=Z`.

(a) Give CHRs to handle `not`.

```
true(X)  \ not(X,Y) <=> false(Y)
false(X) \ not(X,Y) <=> true(Y)
true(Y)  \ not(X,Y) <=> false(X)
false(Y) \ not(X,Y) <=> true(X)
```

(b) Give CHRs to handle `xor`. (You can use the `not` predicate if you want.)

```
true(X)  \ xor(X,Y,Z) <=> not(Y,Z)
false(X) \ xor(X,Y,Z) <=> Y=Z
true(Y)  \ xor(X,Y,Z) <=> not(X,Z)
false(Y) \ xor(X,Y,Z) <=> X=Z
true(Z)  \ xor(X,Y,Z) <=> not(X,Y)
false(Z) \ xor(X,Y,Z) <=> X=Y
```

3. Suppose that the following Haskell code has been loaded.

```
my_const k x = k

my_map f [] = []
my_map f (x:xs) = f x : my_map f xs

my_map2 f [] [] = []
my_map2 f (x:xs) (y:ys) = f x y : my_map2 f xs ys

double x = 2*x

mystery = 2 : my_map double mystery
```

What is the result of evaluting the following Haskell expressions? Remember that `:type x` asks Haskell to print the type of `x`, so in that case give just the type. Otherwise just give the value. If there is a compile-time type error, or a non-terminating computation, say so. If the result is an infinite data structure, give some initial parts of it. If Haskell would give an error of some sort rather than producing output, say so.

(a)
```
:type my_const
a -> b -> a
```

(b)
```
:type my_map (my_const "ho")
[a] -> [String]
```

(c)
```
my_const "ho" (1/0)
"ho"
```

(d)
```
mystery
[2,4,8,16,32,....
```

(e)
```
:type my_map2
(a -> b -> c) -> [a] -> [b] -> [c]
```

4. Suppose that we had a version of Haskell (say Vaskell) that uses call-by-value semantics. Compare Haskell and Vaskell. Are there expressions that evaluate to different values in Haskell and Vaskell? If there are, give an example.

   No, there aren't any — all expressions that evaluate successfully will evaluate to the same value in Haskell and Vaskell.

   Are there expressions whose evaluation performs differently in one or the other (for example, terminating with an error or getting into an infinite recursion in one language and not the other)? If there are, give an example.

   Yes. For example, the expression

   ```
   my_const "ho" (1/0)
   ```

   (as given above) evaluates successfully in Haskell, but gives a divide-by-zero error in Vaskell. Generally, any expression that evaluates without error in Vaskell will also evaluate without error in Haskell, but not vice versa.

5. Tacky but easy-to-grade true/false questions! (OK, even tackier: we're only grading questions 5a-5e.)

   This answer key includes some explanation — but you didn't need to give any explanation in your answer.

   (a) The variables in an answer returned by CLP($\mathcal{R}$) must always be a subset of the variables in the original goal. **False.** There may be some additional variables. Consider the `length` rule, and the goal goal `length(X,100)`. The answer will include 100 new variables not in the original goal.

   (b) If CLP($\mathcal{R}$) had a complete solver, it would never return an answer of "maybe". **True** (by the definition of "complete").

   (c) If Haskell used call-by-name semantics instead of lazy evaluation, there would be certain expressions which could no longer be evaluated, since they would terminate with an error or get into an infinite recursion in the call-by-name version, even though they used to work correctly in standard Haskell. **False**. Lazy evaluation is the same as call-by-name, except that the first time an expression is evaluated, the value is cached and just returned if the expression is ever evauated again.

   (d) The monadic function `putStr` in Haskell has the type `String -> IO ()`. So the expression `putStr "ho"` denotes the action, that if it is ever performed, will print `ho`. **True**

   (e) Difference lists in CLP($\mathcal{R}$) allow programmers to append an item to a list in constant time. If we used an ordinary list representation, rather than a difference list, the time would be proportional to the length of the list. **True**

   (f) In the $\lambda\lambda\lambda$ fraternity, as part of a long-standing hazing ritual, pledges are required to apply the Y combinator to the lazily-evaluated recursive `keg` function to compute the total yearly consumption at the frat. (Unfortunately, nobody has ever finished solving the problem, but that hasn't stopped the festivities.) **Whatever . . .**

6. Suppose that we extend the untyped lambda calculus with a let-expression. The new syntax is

e	::=	x	variable
		$\lambda x.e$	abstraction (function)
		$e_1\ e_2$	application (function call)
		let $x = e_1$ in e_2	(let expression)

   The semantics are that the system computes the value of e_1, and then evaluates e_2 with that value substituted for x (so we're using call-by-value semantics).

(a) Show the the operational semantics rule(s) that need to be added to handle `let`.
Here is a straightforward pair of rules:

$$\frac{e_1 \longrightarrow e_1'}{\text{let } x = e_1 \text{ in } e_2 \longrightarrow \text{let } x = e_1' \text{ in } e_2} \text{ (E-Let1)}$$

$$\frac{}{\text{let } x = v \text{ in } e \longrightarrow [x \mapsto v]e} \text{ (E-Let2)}$$

A perhaps more elegant alternative is to use a single rule that converts a let into a function abstraction and application. (Note that we don't need a separate rule to reduce e_1 to a value — the call-by-value semantics for function application takes care of that.)

$$\frac{}{\text{let } x = e_1 \text{ in } e_2 \longrightarrow (\lambda x.e_2)e_1} \text{ (E-Let)}$$

(b) What are the new values in the language (if any)? Here is the grammar for values without "let":

v ::= $\lambda x.e$

There aren't any.

(c) What are the new stuck expressions in the language (if any)? Here is the grammar for stuck expressions without "let":

stuck ::= x
| stuck e
| v stuck

If we use the first version of the operational semantics, add this rule:

stuck ::= let $x =$ stuck in e

If we use the second version, we don't need any new rules.