Problem Set 5 Solutions

Group Problem 1 [20]: Abstract Syntax Trees
You were given the BINDEX program:

\[ \text{bin} (a \ b \ c) \\
(\times (\text{bind} \ d (\times a \ c)) \\
(\text{bind} \ e (-(d \ b)) \\
(\div (\times b \ d) (+ e \ a))) \\
(\text{bind} \ e (\text{bind} \ b (8 \ a) \\
(-b \ c)) \\
(+ e b))) \]

Suppose that this program is run on the actual parameters \( a = 2, b = 3, \) and \( c = 5. \) An annotated abstract syntax tree for this program has the following structure, where Rand1 Subtree is depicted in Fig. 1 and Rand2 Subtree is depicted in Fig. 2:

```
[a;b;c] forms

Pgm (({}, {}), 42)

body

Mul

Binapp (({a,b,c}, e0, 42)

rand1

Rand1 Subtree (({a,b,c}, e0, 3)

Rand2 Subtree (({a,b,c}, e0, 14)
```

The following abbreviations are used for environments:

<table>
<thead>
<tr>
<th>e0</th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a = 2, b = 3, c = 5}</td>
<td>{a = 2, b = 3, c = 5, d = 10}</td>
<td>{a = 2, b = 3, c = 5}</td>
<td>{a = 2, b = 16, c = 5}</td>
<td>{a = 2, b = 3, c = 5, e = 11}</td>
</tr>
</tbody>
</table>

Group Problem 2 [50]: SIMPREX

a. [10] Give the results of running each of the programs on the argument 3:

i. [1] (simprec \( x \) \( x \) \( \text{simprec} \ 0 \) \( \text{sum} \) \( \times(x) \) \( \text{sum} \)) a))

\[
2 + 2 + 2 + 0 = 6
\]

ii. [2] (simprec \( x \) \( x \) \( \text{simprec} \ 0 \) \( \text{sum} \) \( \times(x) \) \( \text{sum} \)) 4))

\[
4 + 3x + 2x^2 + x^3 \text{ at } x = 3 \\
= 4 + 3 \cdot 3 + 2 \cdot 3^2 + 3^3 \\
= 4 + 9 + 18 + 27 \\
= 58
\]
Figure 1: Rand1 Subtree for the sample BINDEX program.
Figure 2: Rand2 Subtree for the sample BINDEX program.
iii. \[ (\text{simprex} \ (y)) \ (\text{simprec} \ 0 \ (a \ b \ (+ \ b \ (\sigma c \ 1 \ a \ (* \ a \ c)))) \ y)) \]
\((3 \cdot 1 + 3 \cdot 2 + 3 \cdot 3) + (2 \cdot 1 + 2 \cdot 2) + 1 \cdot 1 \)
\(= 18 + 6 + 1 \)
\(= 25 \)

iv. \[ (\text{simprex} \ (n)) \]
\((\text{simprec} \ (\text{simprec} \ (* \ n \ (- \ n \ 3)) \ (q \ r \ r) \ (* \ n \ n)) \)
\((c \ d \ (+ \ d \ (\text{simprec} \ 0 \ (x \ \text{sum} \ (+ \ \text{sum} \ (- \ (* \ 2 \ x) \ 1))) \ c))) \)
\((\text{simprec} \ -5 \ (a \ b \ (+ \ 1 \ b)) \ (* \ n \ n))) \)

In the case where \(n\) is 3, the program is the same as the following expression \(E_{iv}':\)
\((\text{simprec} \ (\text{simprec} \ 0 \ (q \ r \ r) \ 9) \)
\((c \ d \ (+ \ d \ (\text{simprec} \ 0 \ (x \ \text{sum} \ (+ \ \text{sum} \ (- \ (* \ 2 \ x) \ 1))) \ c))) \)
\((\text{simprec} \ -5 \ (a \ b \ (+ \ 1 \ b)) \ 9)) \)

The expression \((\text{simprec} \ 0 \ (q \ r \ r) \ 9)\) ignores the numbers 9 down to 1 and simply returns 0. The expression \((\text{simprec} \ -5 \ (a \ b \ (+ \ 1 \ b)) \ 9)\) adds one for each of the nine numbers, starting with -5, to yield 4. So \(E_{iv}\) can be simplified to \(E_{iv}':\)
\((\text{simprec} \ 0 \)
\((c \ d \ (+ \ d \ (\text{simprec} \ 0 \ (x \ \text{sum} \ (+ \ \text{sum} \ (- \ (* \ 2 \ x) \ 1))) \ c))) \)
4\)

This expression calculates \(f(4, f(3, f(2, f(1, 0))))\), where
\[ f(c,d) = d + \sum_{x=1}^{c} (2 \cdot x - 1) \]

It turns out that \(\sum_{x=1}^{c} (2 \cdot x - 1) = x^2\), so that \(f(c,d) = d + c^2\), and \(E_{iv}\) and \(E_{iv}'\) have the value \(4^2 + 3^2 + 2^2 + 1^2 = 16 + 9 + 4 + 1 = 30\).

a. \[10\] The solution to this part is presented in Fig. 3.

b. \[22\]

i. \[2\] Extend the definition of \text{sexpToExp} in Simprex.ml to correctly parse \text{simprec} expressions.

\[ (* \ val \ \text{sexpToExp} : \text{Sexp.sexp} -> \text{exp} *) \]
\[ \text{and} \ \text{sexpToExp} \ \text{sexp} = \]
\[ \text{match} \ \text{sexp} \ \text{with} \]
\[ \vdots \]
\[ | \text{Seq} \ [\text{Sym} \ "\text{simprec}"; \ \text{zero}; \ \text{Seq} \ [\text{Sym} \ n; \ \text{Sym} \ a; \ \text{comb}]; \ \text{arg}] \rightarrow \]
\[ \text{Simprec}(\text{sexpToExp} \ \text{zero}, \ n, \ a, \ \text{sexpToExp} \ \text{comb}, \ \text{sexpToExp} \ \text{arg}) \]
Figure 3: Wiring diagram for the sample SIMPREX expression.
ii. [2] Extend the definition of `expToSexp` in `Simprex.ml` to correctly unpars the `simprec` expressions.

```ml
(* val expToSexp : exp -> Sexp.sexp *)
and expToSexp e =
  match e with
  : |
  | Simprec(zero,n,a,comb,arg) ->
    Seq [Sym "simprec";
         expToSexp zero;
         Seq [Sym n; Sym a; expToSexp comb];
         expToSexp arg]
```

iii. [3] Extend the definition of `freeVarsExp` in `Simprex.ml` to correctly determine the free variables of a `simprec` expression.

```ml
(* val freeVarsExp : exp -> S.t *)
(* Returns the free variables of an expression *)
and freeVarsExp e =
  match e with
  : |
  | Simprec(zero,n,a,comb,arg) ->
    S.union (freeVarsExps [zero;arg])
      (S.diff (freeVarsExp comb) (listToSet [n;a]))
```

iv. [6] Extend the definition of `eval` in `SimprexEnvInterp.ml` to correctly evaluate `simprec` expressions using the environment model.

```ml
(* val eval : Simprex.exp -> int Env.env -> int *)
and eval exp env =
  match exp with
  : |
  | Simprec(zero,n,a,comb,arg) ->
    foldr (fun i ans ->
      eval comb (Env.bindall [n;a] [i;ans] env))
      (eval zero env)
      (gen (fun x -> x - 1)
          (fun y -> y <= 0)
          (eval arg env))
```

The `gen` expression returns a list of the integers from the value of `arg` down to 1. An alternative way to generate this list is via

```ml
rev (range 1 (eval arg env))
```

The `rev` here is necessary to get the elements in the order expected by the `foldr`. 
v. [3] Extend the definition of subst in Simprex.ml to correctly perform substitutions into simprec expressions. To avoid variable capture is is necessary to rename the two bound variables (n and a) in their scope (comb).

\[
(* \text{val subst : exp \rightarrow exp Env.env \rightarrow exp} *)
let rec subst exp env =
  match exp with
  |
  |
  Simprec(zero,n,a,comb,arg) ->
    let n' = StringUtils.fresh n
    and a' = StringUtils.fresh a in
    Simprec(subst zero env, n', a', subst (renameAll [n;a] [n';a']) comb env, subst arg env)
\]

vi. [6] Extend the definition of eval in SimprecSubstInterp.ml to correctly evaluate simprec expressions using the substitution model.

\[
(* \text{val eval : Simprex.exp \rightarrow int} *)
and eval exp =
  match exp with
  |
  |
  Simprec(zero,n,a,comb,arg) ->
    foldr (fun i ans ->
        eval (subst comb (Env.make [n;a] [Lit i; Lit ans]))
    (eval zero)
    (gen (fun x -> x - 1)
         (fun y -> y <= 0)
         (eval arg)))
\]

As in part (iv), the gen can be replaced by the reversal of an invocation of range.

c. [8] The goal of this problem is to desugar sigma into simprec.
Let us first consider the case in which \(E_{lo}\) is 1. In this case,

\[
(\sigma_{\sigma} \ I_{\sigma} 1 \ E_{hi} \ E_{body}) \rightarrow
(\text{simprec} 0 \ (I_{\sigma} I_{\sigma} (\ I_{\sigma} \ + \ I_{\sigma} \ E_{body}) \ E_{hi}) \ ; \ I_{\sigma} \ fresh)
\]

But what if \(E_{lo}\) isn’t 1? A simple, but potentially inefficient and incorrect, solution is based on the observation that in many cases,

\[
(\sigma_{\sigma} E_{lo} \ E_{hi} \ E_{body})
\]

is equivalent to

\[
(- (\sigma_{\sigma} 1 \ E_{hi} \ E_{body}) \ (\sigma_{\sigma} 1 \ E_{lo} \ E_{body}))
\]

This is potentially inefficient because it evaluates \(E_{body}\) (twice) for all values in the range from 1 to \((E_{lo} - 1)\). These evaluations are not necessary to compute the result, and can potentially swamp the computation (e.g., suppose that \(E_{lo}\) is 1,000,000 and \(E_{hi}\) is 1,000,001). Moreover, this is potentially incorrect because \(E_{body}\) might be defined at all values between \(E_{lo}\) and \(E_{hi}\), but might not be defined at some values between 1 and \((E_{lo} - 1)\). For example, in
(\(\sum_{i=2}^{5} (\frac{1}{i+10} - \frac{1}{i+1})\)), the body expression will raise an exception if evaluated at \(i = 1\).

We seek a desugaring that works in all cases. The structure of \texttt{simpred} requires us to develop an \(n\) and \(c\) such that the desired result can be calculated via:

\[
c(n, c(n-1), c(n-2), \ldots c(2, c(1, 0)))
\]

We assume that the base case is 0, since we're performing a sum. What should \(n\) be? It should be the number of values for which we want to calculate \(E_{\text{body}}\). Let \(hi\) be the value of \(E_{hi}\) and \(lo\) be the value of \(E_{lo}\). Then \(n\) should be \((hi - lo) + 1 = hi - (lo - 1)\). Presumably \(c\) should have the form

\[
c(i, sum) = \text{sum} + ???
\]

Let's assume that the \(E_{\text{body}}\) from \texttt{sigma} is parameterized over \(i\). The ??? cannot simply be the value of \(E_{\text{body}}\), because \(i\) should range from \(lo\) to \(hi\), but in fact ranges from 1 to \(hi - (lo - 1)\). To evaluate \(E_{\text{body}}\) at the correct indices, it is necessary to add \((lo - 1)\) to each index. So:

\[
c(i, \text{sum}) = \text{sum} + (E_{\text{body}} \text{evaluated at } i = i + (lo - 1))
\]

Our final desugaring is:

\[
(\sum I_{\sigma} E_{hi} E_{body}) \rightarrow
(bind I_{lo} E_{lo}; I_{lo} fresh)
\]

\[
(bind I_{hi} E_{hi}; I_{hi} fresh)
\]

\[
(bind I_{lodec} (- I_{lo} 1); I_{lodec} fresh)
\]

\[
(\text{simpred } 0)
\]

\[
(I_{\sigma} I_{\text{ans}}; I_{\text{ans}} fresh)
\]

\[
(+ I_{\text{ans}} (bind I_{\sigma} (+ I_{\sigma} I_{lodec} \text{ } E_{body})))
\]

\[
(- I_{hi} I_{lodec})))))
\]

It's not really necessary to introduce fresh names \(I_{lo}\) and \(I_{hi}\) in this desugaring, since \(E_{lo}\) and \(E_{hi}\) are evaluated only once. But introducing the fresh name \(I_{lodec}\) is important for avoiding duplicating computation, and \(I_{\text{ans}}\) must be fresh to avoid variable capture in \(E_{\text{body}}\). This desugaring is expressed in OCaml as:

```ocaml
| Seq [Sym "sigma"; Sym name; lox; hix; bodyx] ->
  let loVar = StringUtils.fresh "lo"
  and hiVar = StringUtils.fresh "hi"
  and loDecVar = StringUtils.fresh "lodec"
  and ansVar = StringUtils.fresh "ans" in
  Bind(loVar, sexpToExp lox,
    Bind(hiVar, sexpToExp hix,
      Bind(loDecVar, BinApp(Sub, Var loVar, Lit 1),
        Simprec(Lit 0, (*** Ezero ***)
          name,
        ansVar,
          (*** Ecomb ***)
        BinApp(Add,
          Var ansVar,
          Bind(name,
            BinApp(Add, Var name, Var loDecVar),
              sexpToExp bodyx)),
          (*** Earg ***)
        BinApp(Sub, Var hiVar, Var loDecVar))))))
```
Group Problem 3 [15]: Extending Valex with New Primitive Operators

Adding the following four elements to the primops list extends Valex with the four new primitive operators specified in the problem:

(* absolute value *)
Primop("abs", checkOneArg checkInt (fun i -> Int (abs i)));

(* integer sqrt *)
Primop("sqrt", checkOneArg checkInt (fun i -> Int(int_of_float(sqrt(float_of_int i)))));

(* list range *)
Primop("between", checkTwoArgs (checkInt,checkInt)
  (fun i1 i2 -> List (map (fun i -> Int i) (range i1 i2))));

(* list reversal *)
Primop("reverse", checkOneArg checkList (fun vs -> List(rev vs)));

The only tricky aspect to these definitions is guaranteeing that the values returned by the primop are tagged with an appropriate constructor for the valu data type.

Group Problem 4 [20]: Desugaring classify

Whenever implementing a desugaring, it is helpful to first consider the desugaring of some concrete examples. In the case of the classify construct we begin with the first example from the assignment:

\[
\begin{align*}
\text{hofl} (\text{grade}) \\
\text{(classify grade} \\
\quad ((90 \ 100) \ 'A') \\
\quad ((80 \ 89) \ 'B') \\
\quad ((70 \ 79) \ 'C') \\
\quad ((60 \ 69) \ 'D') \\
\quad \text{(otherwise} \ 'F'))))))
\end{align*}
\]

We want this to desugar into something like:

\[
\begin{align*}
\text{hofl} (\text{grade}) \\
\quad (\text{if} \ (\&\& \ (\leq 90 \text{ grade}) \ (\leq \text{ grade} 100)) \\
\quad \ 'A' \\
\quad \ (\text{if} \ (\&\& \ (\leq 80 \text{ grade}) \ (\leq \text{ grade} 89)) \\
\quad \ 'B' \\
\quad \ (\text{if} \ (\&\& \ (\leq 70 \text{ grade}) \ (\leq \text{ grade} 79)) \\
\quad \ 'C' \\
\quad \ (\text{if} \ (\&\& \ (\leq 60 \text{ grade}) \ (\leq \text{ grade} 69)) \\
\quad \ 'D' \\
\quad \ 'F'))))))
\end{align*}
\]

The above example is simple in the sense that none of the discriminant, the low values, or the high values are non-trivial expressions. An example that better illustrates the general case is:

\[
\begin{align*}
\text{hofl} (a \ b \ c \ d) \\
\text{(classify} \ (* \ c \ d) \\
\quad ((a \ (- (/ \ (+ a \ b) \ 2) \ 1)) \ (* \ a \ c)) \\
\quad ((+/ (/ \ (+ a \ b) \ 2) \ 1) \ b) \ (* \ b \ d)) \\
\quad \text{(otherwise} \ (- \ d \ c))))
\end{align*}
\]

We expect the desugaring for this second example to be:
We want to name the value of \((\ast\ c\ d)\) so that we don’t have to recalculate it. Here, \(I_{disc}\) is a fresh name chosen so as not to conflict with any other names that appear in the example.

Based on the two examples, we develop the following three high-level desugaring rules for classify:

1. \((classify\ I_{disc}\ (otherwise\ E_{\text{default}}))\) desugars to \(E_{\text{default}}\). Note that the discriminant in this case is required to be a variable \(I_{disc}\) rather than an arbitrary expression \(E_{disc}\). This is intentional: the specification requires the evaluation of \(E_{disc}\), and if \(I_{disc}\) were replaced by \(E_{disc}\), the above desugaring would not evaluate \(E_{disc}\) in the case where the only clause in the classify is an otherwise clause.

2. \((classify\ I_{disc}\ ((E_{lo}\ E_{hi})\ E_{\text{body}})\ \ldots)\) desugars to
   \[
   (if\ (&& (<= \ E_{lo} \ I_{disc})
            (<= \ I_{disc} \ E_{hi}))
    \ E_{body}
    (classify\ I_{disc}\ \ldots))
   \]
   Again, we assume the discriminant is an identifier \(I_{disc}\) rather than an arbitrary expression \(E_{disc}\). Otherwise, it would be evaluated multiple times, and we only want to evaluate it once.

3. The final rule handles introducing a name \(I_{disc}\) when \(E_{disc}\) is not already a variable:
   \((classify\ E_{disc}\ \ldots)\) desugars to: \((bind\ I_{disc}\ E_{disc}\ (classify\ I_{disc}\ \ldots))\), where \(I_{disc}\) is a fresh variable. Introducing a the variable \(I_{disc}\) in this rule enables the other two rules.

The final step of desugaring is to translate the high-level rules developed above into OCAML code in the desugarRules function:

```ocaml
(* Desugar classify *)
| Seq [Sym "classify"; Sym var; Seq [Sym "otherwise"; default]] -> default
| Seq (Sym "classify" :: Sym var :: Seq [Seq [lo; hi]; body] :: clauses) ->
  Seq [Sym "if";
    Seq [Sym "&&";
      Seq [Sym "<="; lo; Sym var];
      Seq [Sym "<="; Sym var; hi]]; 
    body;
    Seq (Sym "classify" :: Sym var :: clauses)]
| Seq (Sym "classify" :: disc :: clauses) -> (* For non-variable discriminants *)
  let discFresh = StringUtils.fresh("disc") in
  Seq [Sym "bind"; Sym discFresh; disc; 
    Seq(Sym "classify" :: Sym discFresh :: clauses)]
```