1 Substitution Lemmas

In this problem, you will prove two important substitution lemmas that were used in class.

a. (part of Winskel, exercise 6.10) In writing our axiomatic semantics, we relied on a lemma that said that the truth of an assertion remained the same if we substituted into locations mentioned in the assertion the values of those locations in the store. Show that by structural induction on the assertion that this lemma is true:

$$\sigma \models^I B[a/X] \iff \sigma[\mathcal{A}[\![a]\!]\sigma/X] \models^I B$$

Note that here $\mathcal{A}$ is our original semantic function for evaluating arithmetic expressions in IMP, not the extended function that is able to evaluate formula variables; the extended function is written as $\mathcal{A}v$ in Winskel.

b. In proving the soundness of the typing rules for the simply-typed lambda calculus, we relied on a lemma that said we could substitute a term with type τ' for a variable of the same type, without changing the type of the containing expression:

$$(\{x : \tau'\} \vdash e : \tau) \wedge (\emptyset \vdash e' : \tau') \quad \Rightarrow \quad \emptyset \vdash e[e'/x] : \tau$$

Use induction on the structure of e to show that this substitution lemma holds.

2 Axiomatic Semantics

Suppose we want to add integer arrays to the IMP language. We do this by adding two new arithmetic expressions: $\mathsf{X}[a]$ to get the integer stored in array X at the index denoted by the expression a, and $\mathsf{len}(\mathsf{X})$ to find the length of the array. Arrays are zero-indexed, so legal array indices for X are in the range $0 \leq i < \mathsf{len}(\mathsf{X})$. We also need to add a new command for array update: $\mathsf{X}[a_0] = a_1$.

If X is an array, we use the "function-update" notation from class, $\mathsf{X}[i \mapsto j]$, to denote the array X updated so that $\mathsf{X}[i]= j$ (so long as i is in bounds for the array X).

We can now attempt to define the axiomatic semantics for our extended language by adding a new axiom for the array update command:

$$\{B[\mathsf{X}[a_0 \mapsto a_1]/\mathsf{X}]\}\ \mathsf{X}[a_0] = a_1\ \{B\}$$

Unfortunately, this is not enough. As hinted at above, in order to prove the correctness of an assignment to an array, we must make sure that the array index is in bounds. Similarly, the correctness of arithmetic expressions involving array entries (such as $\mathsf{3} + \mathsf{X[4]}$) also depends on bounds checking. We thus need to extend our proof rules to check that the "domain" of arrays is respected.

A similar problem arises when we add integer division to the language. If we naively add the arithmetic expression a_0/a_1 without changing our notion of correctness to account for division by zero, our axiomatic semantics will be wrong.

In this problem you extend the axiomatic semantics to account for these domain restrictions.

The syntax for the extended version of IMP (including arrays and integer division) is given below:

$$\begin{array}{lcl}
a & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1 \mid a_0/a_1 \mid \mathsf{X[a]} \mid \mathsf{len(X)} \\
b & ::= & b_0 \wedge b_1 \mid b_0 \vee b_1 \mid \neg b \mid a_0 = a_1 \mid a_0 \leq a_1 \\
c & ::= & \mathsf{skip} \mid \mathsf{while}\ b\ \mathsf{do}\ c \mid x := a \mid \mathsf{X}[a_0] := a_1 \\
 & & \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \mid c_0; c_1
\end{array}$$

a. Write inductively defined predicates $dom_a(-)$ and $dom_b(-)$ for arithmetic expressions and boolean expressions, respectively. The idea is that that for any arithemetic expression a_0, $\models^I dom_a(a_0)$ if and only if every array index operation and every integer division in a_0 obeys they proper domain restrictions. (Similarly, $dom_b(-)$ asserts that boolean expressions don't have domain violations.)

b. Change the proof rules for the axiomatic semantics of our extended IMP language to use the domain predicates from part (a).

c. The following program in the extended version of IMP performs an insertion sort on the array X.

$j = 1$;
while $j <$ len(X) do
  $x =$ X[j];
  $i = j-1$;
  while $(i \geq 0) \wedge (\mathsf{X}[i] < x)$ do
    X[i+1] = X[i];
    $i = i - 1$
  X[i+1] $= x$
  $j = j + 1$

The desired post-condition, P, of the above loop specifies that X is sorted in descending order:

$$P = \{\forall i.0 \leq i < (\mathsf{len}(\mathsf{X}) - 1) \Rightarrow \mathsf{X}[i] \geq \mathsf{X}[i+1]\}$$

1. Write down the loop invariants for the two while loops in the above program using the assertion language for axiomatic semantics discussed in class.
2. If A_0 is the loop invariant for the outer loop, prove that $A_0 \wedge \neg(j < \mathsf{len}(\mathsf{X})) \Rightarrow P$. That is, if the outer loop terminates with loop invariant A_0, then the post-condition holds.
3. Let A_1 be the loop invariant for the inner while loop and let c be the body $\mathsf{X}[i+1] = \mathsf{X}[i];\ i = i-1$. Prove that that A_1 is actually an invariant, that is the following judgement is derivable in the extended axiomatic semantics:

$$\{A_1 \wedge ((i \geq 0) \wedge (\mathsf{X}[i] < x))\}\ c\ \{A_1\}$$

3 Type checking

Suppose we extend the simply-typed lambda calculus with a new type constructor list(τ), representing a finite list of elements all of the same type τ. This type will support the following operations:

- The expression (list $e_1\ \ldots\ e_n$) constructs a list containing the elements e_1 through e_n in sequence.
- The expression (hd l) returns the first element in the list l.
- The expression (tl l) returns a list contains the second and following elements in the list l.
- The expression null yields an empty list.
- The expression (null? l) tests whether the list l is empty, returning true or false accordingly.

a. Extend the syntax and static semantics of $\lambda^{\rightarrow}$ to support the new kind of type.

b. Using domain equations, define a domain in the meta-language that can be used to represent values of this type. Extend the function $\mathcal{T}$ that maps a type expression τ into its corresponding domain, accordingly.

4 Implementation

In this problem you will write an interpreter for a trimmed-down, call-by-value version of the Lambda language used in assignment two. Source code for the lexer, parser, and pretty-printer, as well-as the framework for the interpreter are found in `lambdaval.tar.gz` available from the course web page. The only file you need to modify for this homework is `interpretation.sml`.

Some old features of the Lambda language have been dropped: functions take only one argument, application is thus limited to one argument, the list notation using square brackets has been removed, the operators have been separated into a separate syntactic classes (binops and unops) and their application has been restricted appropriately. Furthermore, since this version is call-by-value, letrec has been restricted to functions.

A few new features have also been added to the language: (print e) causes the value of e to be printed (if it is "printable", that is, not a function). Two additional constructs (for x e e) and (yield e) are described below.

The new syntax is given below, where x and f range over variables and n ranges over integers. (Source code implementing this syntax is given in `ast.*`.)

$$
\begin{array}{rcl}
\textit{binop} & ::= & + \mid * \mid - \mid \textsf{cons} \mid \textsf{and} \mid \textsf{or} \\
\textit{unop} & ::= & \textsf{hd} \mid \textsf{tl} \mid \textsf{zero?} \\
e & ::= & x \mid \#\textsf{t} \mid \#\textsf{f} \mid n \mid (\textsf{fn}\ (x)\ e) \mid (e\ e) \mid (\textsf{if}\ e\ e\ e) \mid \\
 & & (\textit{binop}\ e\ e) \mid (\textit{unop}\ e) \mid (\textsf{letrec}\ (f\ (\textsf{fn}\ (x)\ e))\ e) \\
 & & (\textsf{print}\ e) \mid (\textsf{for}\ x\ e\ e) \mid (\textsf{yield}\ e)
\end{array}
$$

The interpreter you write will be modeled on the continuation-style denotational semantics discussed in class. That is, SML functions will be used to interpret the $\rightarrow$ constructor of the domain equations. Thus, the `interpretation.*` files contain SML datatypes (which correspond to the domain equations as seen in class). In particular, semantic domains for values (which include booleans, numbers, cons cells, and functions), continuations, environments, and computations are provided. The bulk of the interpreter is the semantic function `C : expr -> Computation`, which corresponds to the $\mathcal{C}[\![-]\!]$ we use in class.

The file `interpretation.sml` contains the skeleton of an implementation of `C`, as well as a bunch of utility functions for evaluating binary and unary operators, and printing values. The function `extend`, of (SML) type `Env -> var -> value -> Env`, updates an environment to bind a new value to a variable. The initial environment `rho0` and initial continuation `K0` are also defined.

To interpret the letrec expression, we need to be able to take a fixed point of an appropriate function. In a call-by-value setting, it is not always possible to guarantee that such fixed points are values (i.e. don't diverge), which is why we restrict letrec to operate on functions. (This guarantees that the possible divergence has been "delayed" until the function f is applied, and so produces a value.) Due to this restriction, we can use a more limited version of the fix operator, which instead of having type $(A \rightarrow A) \rightarrow A$ as usual, only works on functions, and so has type $((A \rightarrow B) \rightarrow (A \rightarrow B)) \rightarrow (A \rightarrow B)$. Thus, in SML, the fix we need is: `fun fix f = fn x => f (fix f) x`. This function is also provided in `interpretation.sml`.

As usual, the file `top.sml` provides code for loading source files and running your interpreter. There are a number of `.lam` files which contain valid (hopefully self-explanatory) source. You should be able to compile the entire program by typing `CM.make()` at the SML command prompt. Mail your `interpretation.sml` file to `zdance@cs.cornell.edu`.

a. Write the `C` function using the denotational semantics in class to guide you. It may be useful to recall that the SML syntax for an anonymous function like $\lambda x.e$ is `fn x => e`. Also, you will have to insert the `K` constructor to turn a function (of the correct type) into a "semantic" continuation of type `Cont`. That is, if `f` is an SML function of type `value -> value`, then `K f` is a member of the `Cont` domain.

   The (print e) expression should evaluate e, call `printVal` on the resulting value, and continue with `BoolV true` as its result.

b. In this question, we add a new syntactic construct for iteration to the language. There will be two new kinds of expressions: (for x e_i e_b) and (yield e). The idea is that a for expression evaluates the "iterator" e_i. If a (yield e_v) expression is encountered during the evaluation of e_i, then the body e_b is evaluated in an environment where x is bound to the result of evaluation e_v. After evaluating e_b, control is transferred back into e_i at the point of the yield and evaluation continues as though the yield had resulted in e_b. A yield expression not in the scope of any for expression should raise a runtime error. Note that the variable x is bound in e_b but not e_i.

For example: `(for x (yield 3) (print x))` Should have the effect of printing "3" and then halting with the value true.

As another example:

```
(for x (letrec (f (fn (y) ((fn (x) (f (+ y 1))) (yield (+ y 1))))) (f 0))
     (print x))
```

Should loop infinitely, printing the number "1", "2", "3", etc. The following example shows how to construct an iterator that walks over a tree of cons cells recursively, and yields all the leaf cells. The leaf cells are identified by having a head that is zero; their tail contains the actual value of the leaf. For example, the following code traverses a tree and prints the integers from 1 to 5:

```
(letrec (walk (fn (cell)
               (if (zero? (hd cell))
                 (yield (tl cell))
                 (seq (for left (walk (hd cell)) (yield left))
                      (for right (walk (tl cell)) (yield right))))))
 (let (tree
     (cons (cons (cons 0 1) (cons 0 2))
          (cons (cons 0 3) (cons (cons 0 4) (cons 0 5)))))
   (for (n (walk tree) (print n)))
 ))
```

Note that we don't have let and seq constructs in the language, but they can be desugared (identically!):

$$(\mathsf{let}\ (x\ e_1)\ e_2) \longrightarrow ((\mathsf{fn}\ x\ e_2)\ e_1)$$

$$(\mathsf{seq}\ e_1\ e_2) \longrightarrow ((\mathsf{fn}\ x\ e_2)\ e_1) \qquad (x \notin FV[\![e_2]\!])$$

(You don't need to add them to the language; these desugarings are provided for the purpose of understanding the code above.) The file treewalk.lam contains the desugared version of this code, which you can use as a test case. Other test cases are also included.

Give a denotational semantics for each of the new constructs (you shouldn't need to change any domain equations), and implement them in your interpreter. To support the ability to yield values, the type Computation has already been changed for you from `Cont -> Env -> value` to `Cont -> Cont -> Env -> value`, where the second Cont is the continuation used by yield. The constant Kco is a top-level continuation that raises an error when a yield is performed outside the scope of an iterator. One subtlety is that the yield continuation expects a pair consisting of the continuation that jumps back into the iterator and the value to be yielded.