CSCI 3155: Homework Assignment 6

Due Friday, October 30, 2009

For this assignment, you will again work with a partner. You will write up and turn in this assignment in pairs. Choose one of you to upload your write up to the moodle. Please name the file

`hw6-YourIdentiKey-YourPartnersIdentiKey.pdf`

Also, please include both of your PL-Detective user ids on your write-up and indicate which one should be considered the submission id. As always, you are welcome to discuss in larger groups. Just be sure to acknowledge those with which you discussed.

**Bookkeeping**

**Exercise 1**: Indicate in a sentence or two how much time you spent on this homework, how difficult you found it subjectively, and what you found to be the hardest part. Any non-empty answer will receive full credit.

If you would like share something about yourself that I do not already know, please do so. And if your opinions have changed since the last submission, indicate one thing you like about the class so far and one thing you would change about it.

**Data Types**

**Exercise 2**: Skill 11.1. Give a concrete and realistic example that uses a subrange type. Explain why your example demonstrates a good use in a few sentences.

**Exercise 3**: Skill 11.2. Give a concrete and realistic example that uses an enumeration type. Explain why your example demonstrates a good use in a few sentences.

**Pointers and Memory Management**

**Exercise 4**: Synthesis. If a language wants to support garbage collection, the language must be designed appropriately. In other words, one cannot design a language while being oblivious to garbage collection and hope that it will “just work.” In particular, the language must be designed such that the garbage collector is able to do the following:

- find all the pointers in memory; and
- identify which object each pointer points to.
For the exercise, answer the following:

1. Explain the need for the two requirements given above.

2. Does C++ support these two requirements? Support your answer either with quotes from the C++ language definition or with examples and their output (much like how you use the PL Detective) from a C++ compiler/run-time system.

Exercise 5: Consider the following program in C++ (really a C program except that we use `new/delete` instead of `malloc/free`):

```c
1 unsigned int* max(unsigned int a[5]) {
2    unsigned int* maxptr = new unsigned int;
3    for (int i = 0; i < 5; ++i) {
4        if (*maxptr < a[i]) *maxptr = a[i];
5    }
6    return maxptr;
7 }
8
9 int main() {
10   unsigned int a1[5] = {2,7,4,5,5};
11   unsigned int a2[5] = {2,7,4,11,5};
12   unsigned int* t;
13   t = max(a1);
14   printf("max of a1: %d\n", *t);
15   t = max(a2);
16   printf("max of a2: %d\n", *t);
17 }
```

1. **Skill 12.1.** Does the above program suffer from memory leak or dangling pointer bugs? If so, indicate where and why.

2. **Skill 12.2.** Rewrite the program above to eliminate the memory leak or dangling pointer bugs.

Exercise 6: Consider the following program in C++ (really a C program except that we use `new/delete` instead of `malloc/free`):

```c
1 int* g;
2 // make the location pointed to by g contain the same value as p
3 void f(int* p) {
4    g = p;
5 }
6
7 int main() {
8    int* x = new int;
9    *x = 5;
```
10   f(x);
11   delete (x);
12   printf("g is: %d\n", *g);
13 }

1. **Skill 12.1.** Does the above program suffer from memory leak or dangling pointer bugs? If so, indicate where and why.

2. **Skill 12.2.** Rewrite the program above to eliminate the memory leak or dangling pointer bugs.

**Evaluation Order**

**Exercise 7: Skill 13.3.** Calls are amongst the most complicated expression constructs in programming languages. There are many, many different ways of supporting calls and different languages often differ in their support for calls. In this question, we will consider only the operand evaluation order issues with calls; later in the semester we will consider other issues.

Let’s consider a call schematically $F(A_1, \ldots, A_n)$. $F$ is the procedure being called. While we are used to calling a procedures directly (e.g., `print()`), most languages, including MYSTERY, allow complicated expressions for $F$. For example, if $a$ is an array of procedures, one can do $a[2](5)$ in MYSTERY to call the procedure at $a[2]$ with argument $5$. MYSTERY also allows a procedure to return a procedure (i.e., has higher-order functions). Thus, if a procedure $f$ returns a procedure, then one could write $f() (x)$. This calls the procedure returned by $f$ with the argument $x$. Needless to say, most languages allow one to pass arbitrarily complicated arguments (i.e., $A_i$).

1. Come up with and describe in detail a compelling example of why would one want to return a procedure from a procedure.

2. Discover whether MYSTERY evaluates the procedure being called before it evaluates its arguments. For example, with $f() (x)$ does MYSTERY call the function $f$ first or does it evaluate $x$ first (i.e., looks up the value of $x$). You may execute up to 4 PRINT statements.

3. Discover whether or not MYSTERY evaluates arguments to a call from left-to-right. You may execute up to 4 PRINT statements.

Use the following link to submit programs:

http://www-plan.cs.colorado.edu/diwan/pldeval.htm

You should provide evidence that supports your argument. You will lose 5% of the points for this question for each additional executed PRINT.

**Induction and Type Safety**

**Exercise 8: Skill 8.4.** This exercise is similar to the one on Homework Assignment 5 and provides additional practice with structural induction. Consider the following SML **datatype** for the abstract syntax of a simple language of arithmetic and boolean expressions, including addition and if-then-else.
datatype va =
  Bool of bool
 | Int of int

datatype expr =
  V of va
 | Plus of expr * expr
 | IfThenElse of expr * expr * expr

There are two kinds of values in our language (given as SML values of type \textit{va}): booleans and integers.

With this language, we can create intuitively ill-typed expressions (given as SML values of type \textit{expr}). For example, the addition of two booleans:

\[
\text{Plus (V (Bool true), V (Bool false))}
\]

If we try to evaluate the above expression, it should raise an error or crash. To prevent such cases, we can define a type checker that rules out such expressions.

In this exercise, we consider the idea of \textit{type safety}, that is, if the type checker says the expression is well-typed, then evaluation will not “go wrong.” This property is quite cool: it says we can write a checker that is guaranteed to rule out certain \textbf{bad executions before execution}. This language is small to minimize the number of cases you need to prove (thankfully, right?) and thus not particularly realistic, but you could think about how extending the language might affect the proof you write.

Our language of types consists of only the base types for booleans (\textit{TBool}) and integers (\textit{TInt}):

\[
\text{datatype typ = TBool | TInt}
\]

We define a type inference as an SML function \textit{ityp} by recursion over \textit{expr}.

\[
\begin{align*}
(* \textit{ityp : expr} \rightarrow \text{typ option} *) \\
\text{fun ityp (V (Bool \_)) = SOME (TBool)} \\
| \text{ityp (V (Int \_)) = SOME (TInt)} \\
| \text{ityp (Plus (e1, e2)) =} \\
| \hspace{1em} \text{(case (ityp e1, ityp e2) of} \\
| \hspace{2em} (SOME TInt, SOME TInt) \Rightarrow \text{SOME TInt} \\
| \hspace{2em} _ \Rightarrow \text{NONE}) \\
| \text{ityp (IfThenElse (e1, e2, e3)) =} \\
| \hspace{1em} \text{(case (ityp e1, ityp e2, ityp e3) of} \\
| \hspace{2em} (SOME TBool, SOME t2, SOME t3) \Rightarrow \text{if } t2 = t3 \text{ then SOME t2 else NONE} \\
| \hspace{2em} _ \Rightarrow \text{NONE})
\end{align*}
\]

The \textit{ityp} function returns a \textit{typ option} where \textit{SOME t} means the \textit{expr} is well-typed (and infers the \textit{typ} of the \textit{expr}) and \textit{NONE} indicates the \textit{expr} is ill-typed. The first two cases say that boolean or integer values have boolean and integer type, respectively. For \textit{Plus}, both subexpressions must have type \textit{TInt} and the result has type \textit{TInt}. For \textit{IfThenElse(e1, e2, e3)}, the guard \textit{e1} must have type \textit{TBool} and the branches \textit{e2} and \textit{e3} must have the same type.

In this assignment, we define another style of evaluator. We define an SML function \textit{step} that conceptually takes one step of evaluation.
fun step (Plus (V (Int i1), V (Int i2))) = V (Int (i1 + i2))
| step (Plus (V (Int i1), e2)) = Plus (V (Int i1), step e2)
| step (Plus (e1, e2)) = Plus (step e1, e2)
| step (IfThenElse (V (Bool true), e2, e3)) = e2
| step (IfThenElse (V (Bool false), e2, e3)) = e3
| step (IfThenElse (e1, e2, e3)) = IfThenElse (step e1, e2, e3)

The first case says that Plus of two integers can be added, while the next two cases take a step in a subexpression of Plus. The first two cases for IfThenElse follow the appropriate branch, while the last case takes a step in the guard e1. Notice that this function is not exhaustive. We have left out the ill-typed cases, so if step is passed an ill-typed expr (or a va, that is, V v where v : va), the SML match exception will be raised. Then, to fully evaluate, we can repeatedly apply step until we reach a va:

fun eval (V v) = v
| eval e = eval (step e)

Type safety says that if an expr is well-typed (as determined by ityp), then we can repeatedly apply step without failing until possibly reaching a va. We can break this property down into two parts:

- If an expr is well-typed, then we can take one step without failing (i.e., we can make progress). In this exercise, you will prove this part (Theorem 1).
- If an expr is well-typed and we can make progress, then the result of the step has the same type (i.e., types are preserved). We do not yet have the technical ability to show this part, but you can check out CSCI 5535 next semester to learn more.

For this exercise, prove the Progress Theorem. A template for a proof is given below as a suggestion.

**Theorem 1 (Progress).** For all expr values e, typ values t, if

\[ \text{ityp } e \rightarrow^* \text{ SOME } t \]

then

either \( e = V v \) for some \( v : \text{va} \) or \( \text{step } e \rightarrow^* e' \) for some \( e' : \text{expr} \).

**Proof.** By induction on \__________\.

**Base Case:** \( e = V v \)

\ldots

**Inductive Case:** \( e = \text{Plus}(e_1, e_2) \)

1. \( \text{ityp } (\text{Plus}(e_1, e_2)) \rightarrow^* \text{ SOME } t \) hypothesis
2. \( \text{ityp}(\text{Plus}(e_1, e_2)) \)
   \[ \rightarrow \text{(case (ityp e_1, ityp e_2) of}
   \]
   \[ (\text{SOME TInt, SOME TInt}) \rightarrow \text{SOME TInt} \mid _\rightarrow \text{NONE}) \]
   \[ \text{by def. of ityp} \]

3. \( t = \text{SOME TInt} \)

4. \( \text{ityp } e_1 \rightarrow^* \text{SOME TInt} \) and \( \text{ityp } e_2 \rightarrow^* \text{SOME TInt} \)

5. ... 

**Inductive Case:** \( e = \text{IfThenElse}(e_1, e_2, e_3) \)

1. \( \text{ityp}(\text{IfThenElse}(e_1, e_2, e_3)) \rightarrow^* \text{SOME } t \)
   \[ \text{hypothesis} \]

2. \( \text{ityp}(\text{IfThenElse}(e_1, e_2, e_3)) \)
   \[ \rightarrow \text{(case (ityp e_1, ityp e_2, ityp e_3) of}
   \]
   \[ (\text{SOME TBool, SOME } t_2, \text{SOME } t_3) \rightarrow
   \]
   \[ \text{if } t_2 = t_3 \text{ then SOME } t_2 \text{ else NONE}
   \]
   \[ \mid _\rightarrow \text{NONE} \) \]
   \[ \text{by def. of ityp} \]

3. \( t = \text{SOME } t_2 \)

4. \( \text{ityp } e_1 \rightarrow^* \text{SOME TBool}, \text{ityp } e_2 \rightarrow^* \text{SOME } t_2, \) and \( \text{ityp } e_3 \rightarrow^* \text{SOME } t_2 \)

5. ... 

\[ \square \]

The template essentially provides the steps to show that if \( \text{ityp } e \) evaluates to \( \text{SOME } t \), then the subexpressions of \( e \) evaluate to \( \text{SOME} \) of appropriate \text{typ}s. Recall that we write

\[
\begin{align*}
\text{mle} & \rightarrow \text{mle}' \quad \text{for one step of SML evaluation} \\
\text{mle} & \rightarrow^* \text{mle}' \quad \text{for some number of steps of SML evaluation}
\end{align*}
\]

To finish the proof, you may assume the following basic lemma that says that values of each type have particular forms.

**Lemma 2** (Canonical Forms).

1. For all \( va \) values \( v \), if \( \text{infertyp } (Vv) \rightarrow^* \text{SOME TInt} \), then \( v = \text{Int } i \) (for some \( i : \text{int} \)).

2. For all \( va \) values \( v \), if \( \text{infertyp } (Vv) \rightarrow^* \text{SOME TBool} \), then \( v = \text{Bool } i \) (for some \( i : \text{int} \)).

The SML code for this exercise is available on the course website for your experimentation (along with two example \text{expr}s).