Homework 2

Due 15 October

Handout 3
CS242: Autumn 2002
8 October

Reading

1. Read Chapter 3 on Lisp and Sections 4.1–4.2 on syntax and lambda calculus for this homework. Read Sections 4.3–4.4 for the next lecture.

Problems

1. .... Parsing and Precedence

Draw parse tress for the following expressions, assuming the grammar and precedence described in Example 4.2:

(a) $1 - 1 * 1$.

(b) $1 - 1 + 1$.

(c) $1 - 1 + 1 - 1 + 1$, if we give $+$ higher precedence than $-$.

2. .... Lambda Calculus Reduction

Use lambda calculus reduction to find a shorter expression for $(\lambda x.\lambda y.xy)(\lambda x.xy)$. Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

3. .... Lambda Reduction with Sugar

Here is a "sugared" lambda-expression using `let` declarations:

$$\begin{array}{l} \texttt{let}\ compose = \lambda f.\, \lambda g.\, \lambda x.\, f(g\,x)\ \texttt{in} \\ \quad \texttt{let}\ h = \lambda x.\, x + x\ \texttt{in} \\ \quad\quad compose\ h\ h\ 3 \end{array}$$

The "de-sugared" lambda-expression, obtained by replacing each `let` $z = U \texttt{in} V$ by $(\lambda z.\, V)\, U$ is

$$\begin{array}{l} (\lambda compose. \\ \quad (\lambda h.\ compose\ h\ h\ 3)\ \ \lambda x.\, x + x) \\ \quad \lambda f.\, \lambda g.\, \lambda x.\, f(g\ x). \end{array}$$

This is written using the same variable names as the `let`-form in order to make it easier to read the expression.

Simplify the desugared lambda expression using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

4. .... Translation into Lambda Calculus

A programmer is having difficulty debugging the following `C` program. In theory, on an "ideal" machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, since extra space is required every time a function call is made.)

```
int f(int (*g)(...)){ /*  g points to a function that returns an int */
   return g(g);
}
```

```
int main(){
  int x;
  x = f(f);
  printf("Value of x = %d\n", x);
  return 0;
}
```

Explain the behavior of the program by translating the definition of `f` into lambda calculus and then reducing the application `f(f)`. This program assumes that the type checker does not check the types of arguments to functions.

5. .... Denotational Semantics

The text describes a denotational semantics for the simple imperative language given by the grammar

$$P \ ::= \ x := e \mid P_1; P_2 \mid \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \mid \mathbf{while}\ e\ \mathbf{do}\ P$$

Each program denotes a function from *states* to *states*, where a *state* is a function from *variables* to *values*.

(a) Calculate the meaning $\mathcal{C}[\![x := 1; x := x + 1;]\!](s_0)$ in approximately the same detail as the examples given in the text, where $s_0 = \lambda v \in \mathit{Variables}.\, 0$ giving every variable the value 0.

(b) Denotational semantics is sometimes used to justify ways of reasoning about programs. Write a few sentences, referring to your calculation in part (a), explaining why

$$\mathcal{C}[\![x := 1; x := x + 1;]\!](s) \ = \ \mathcal{C}[\![x := 2;]\!](s)$$

for every state s.

6. .... Denotational Semantics and Linux Bugs

This problem will discuss a nonstandard denotational semantics related to finding null-pointer bugs. Dereferencing a `NULL` pointer will cause a segmentation fault, and in operating system kernel code, this will generally cause the machine to reboot. Despite this disastrous effect, the code for Linux may dereference pointers that are potentially `NULL`. Let's take a look at an example in the SCSI driver (`drivers/scsi/hosts.c`):

```
170:  shn = (Scsi_Host_Name*) kmalloc(sizeof(Scsi_Host_Name), GFP_ATOMIC);
171:  shn->name = kmalloc(hname_len + 1, GFP_ATOMIC);
```

Notice that the variable, `shn`, is immediately dereferenced after being assigned the return value from `kmalloc`, the Linux memory allocator. If the call to `kmalloc` fails and returns `NULL`, this code will produce a segmentation fault and cause your machine to reboot. Rebooting is not only inconvenient, but an impediment to debugging since the state of the machine is lost. The code *should* read:

```
170:  shn = (Scsi_Host_Name*) kmalloc(sizeof(Scsi_Host_Name), GFP_ATOMIC);
171:  if (shn != NULL)
172:    shn->name = kmalloc(hname_len + 1, GFP_ATOMIC);
```

Your goal, in this problem, is to complete the definition of a denotational semantics that statically analyzes C code to catch these kinds of null pointer errors in Linux kernel or related code.

(a) In denotational semantics, the meaning of each program is a function from states to states. If we want to understand changes in the values of variables, then a state will be a mapping from variables to values. In this problem, we want to keep track of whether a pointer has a been set to something other than `null`. As in the semantics tracking uninitialized variables, we also want to consider a program erroneous if an error appears anywhere in the program. Fill in the blank in the definition of state for the semantics below:

$$\textbf{State} \quad = \quad \{\underline{\hspace{3cm}}\} \cup (\,\mathit{Variables} \rightarrow \{\mathit{unknown}, \mathit{null}, \mathit{notnull}\})$$

Remember that *Variables* here include any pointer that can be dereferenced.

(b) The $+$ operator on states is used to combine possible states that could arise at the same point in the execution of the program. For example, if a program contains `if (...) { ...}; else { ...}` , then we do not know which branch will be taken at run time. Therefore, the semantics will combine state σ_{then} arising from the "then" branch and σ_{else} arising from the "else" branch and use state $\sigma_{then} + \sigma_{else}$ as the program state after the if-then-else.

Fill in the blanks (column 3) of this table:

$\sigma_1(x)$	$\sigma_2(x)$	$(\sigma_1 + \sigma_2)(x)$
unknown	*unknown*	
unknown	*null*	
unknown	*notnull*	
null	*null*	
null	*notnull*	
notnull	*notnull*	

This definition is part of what makes our analysis conservative in detecting errors. Recall that a conservative analysis is one that allows an error to be reported in a program that doesn't necessarily have one, but assures that every program with an error will be reported as such.

(c) Fill in the blanks of the given commands.

$$(1)\mathbf{C}[\![\mathtt{x} = \mathtt{y}]\!]\sigma \quad = \quad \begin{cases} \underline{\hspace{3cm}} & \text{if } \mathbf{E}[\![\mathtt{y}]\!]\sigma = \mathit{error} \\ \text{modify}(\sigma, \mathtt{x}, \underline{\hspace{3cm}}) & \text{otherwise} \end{cases}$$

$$(2)\mathbf{C}[\![\mathtt{x} = \mathtt{kmalloc}(...)]\!]\sigma \quad = \quad \text{modify}(\sigma, \mathtt{x}, \underline{\hspace{3cm}}\,)$$

$$(3)\mathbf{C}[\![*\mathtt{x} = \mathtt{value}]\!]\sigma \quad = \quad \begin{cases} \underline{\hspace{3cm}} & \text{if } \mathbf{E}[\![\mathtt{x}]\!]\sigma = \underline{\hspace{3cm}} \\ & \text{ or } \mathbf{E}[\![\mathtt{x}]\!]\sigma = \underline{\hspace{3cm}} \\ \sigma & \text{otherwise} \end{cases}$$

$$(4)\mathbf{C}[\![P1; P2]\!]\sigma \quad = \quad \begin{cases} \textbf{error} & \text{if } \mathbf{C}[\![P1]\!]\sigma = \underline{\hspace{3cm}} \\ \mathbf{C}[\![P2]\!](\underline{\hspace{3cm}}) & \text{otherwise} \end{cases}$$

$$(5)\mathbf{C}[\![\mathtt{if}(\mathtt{x} == \mathtt{NULL})\ P1\ \mathtt{else}\ P2]\!]\sigma \quad = \quad \begin{cases} \mathbf{C}[\![P1]\!]\ \text{modify}(\sigma, x, \underline{\hspace{3cm}}) \\ +\ \mathbf{C}[\![P2]\!]\ \text{modify}(\sigma, x, \underline{\hspace{3cm}}) \end{cases}$$

(d) Calculate the meaning of:

```
{
   int* x;
```

```
2:     x = kmalloc (4, GFP_ATOMIC);
3:     if (x == NULL)
4:       printf ("no mem\n");
5:     else
6:       *x = 5;
7:   }
```

You may assume that statements and expressions that are not defined in our semantics have no effect on the state (e.g., `printf`) and that the initial state has the variable, `x`, with the value *notnull*. Show the main steps. You may use the line number in the commands instead of writing out the entire code. For example, your first step will start out like this:

$$\mathbf{C}[\![L2; L3-6]\!]\sigma = \ldots \texttt{(Command \#)}$$

where L2 is line 2 (the statement with call to `kmalloc`) and L3-6 is lines 3 through 6 (the `if` statement). Please use the numbers of the commands that we provide in your calculation. When you apply the $+$ operator, indicate that with `(+)` to the right of the calculation. Feel free to rename states at various points in your calculation.

(e) Now assume that we add a new command:

$$(6)\mathbf{C}[\![\mathtt{memset}(\mathtt{x}, 0, \mathtt{size})]\!]\sigma = \mathbf{C}[\![*\mathtt{x} = 0]\!]\sigma$$

for any value of `size`. Calculate the meaning of the following code, which is taken directly from Linux 2.4.1 in `net/sched/sch_gred.c`):

```
439:  if (table->tab[table->def] == NULL)
      {
440:       table->tab[table->def] = kmalloc(sizeof(struct gred_sched_data),
441:                                        GFP_KERNEL);
442:       if (table->tab[ctl->DP] == NULL)
443:         return -ENOMEM;
444:
445:       memset(table->tab[table->def], 0, (sizeof(struct gred_sched_data)));
      }
```

You may assume that the initial state, σ, has variables mapped to values as follows:

$$\begin{aligned} \sigma(\texttt{table->tab[table->def]}) &= \textit{unknown} \\ \sigma(\texttt{table->tab[ctl->DP]}) &= \textit{unknown} \end{aligned}$$

Again, statements and expressions that are not defined in our semantics have no effect on the state. Show all steps as you did for the last part of the problem.

(f) If we change line 442 to:

```
if (table->tab[table->def] == NULL)
```

the bug in the code would be fixed.

Would your final state in part (e) change?

If your answer is yes, show your recalculation of the meaning. If your answer is no, explain in one sentence how you must change the denotational semantics to demonstrate that the bug has been fixed.

7. .... Lazy Evaluation and Parallelism

In a "lazy" language, a function call f(e) is evaluated by passing the *unevaluated* argument to the function body. If the value of the argument is needed, then it is evaluated as part of the evaluation of the body of f. For example, consider the function g defined by

fun g(x, y) = **if** x = 0
  then 1
  else if x + y = 0
    then 2
    else 3;

In a lazy language, the call g(3, 4 + 2) is evaluated by passing some representation of the expressions 3 and 4 + 2 to g. The test x = 0 is evaluated using the argument 3. If it were true, the function would return 1 without ever computing 4 + 2. Since the test is false, the function must evaluate x + y, which now causes the actual parameter 4 + 2 to be evaluated. Some examples of lazy functional languages are Miranda, Haskell and Lazy ML; these languages do not have assignment or other imperative features with side effects.

If we are working in a pure functional language without side-effects, then for any function call $f(e_1, e_2)$, we can evaluate e_1 before e_2 or e_2 before e_1. Since neither can have side-effects, neither can affect the value of the other. However, if the language is lazy, we might not need to evaluate both of these expression. Therefore, something can go wrong if we evaluate both expressions and one of them does not terminate.

As Backus argues in his Turing Award lecture, an advantage of pure functional languages is the possibility of parallel evaluation. For example, in evaluating a function call $f(e_1, e_2)$ we can evaluate both e_1 and e_2 in parallel. In fact, we could even start evaluating the body of f in parallel as well.

(a) Assume we evaluate $g(e_1, e_2)$ by starting to evaluate g, e_1, and e_2 in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?

(b) Now, suppose the value of e_1 is zero and evaluation of e_2 terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression $g(e_1, e_2)$ will terminate in error. What will happen with lazy evaluation? Parallel evaluation?

(c) Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate $g(e_1, e_2)$ by starting g, e_1, and e_2 in parallel, if the value of e_1 is zero and evaluation of e_2 terminates in an error?

(d) Suppose, now, that the language contains side-effects. What if e_1 is z, and e_2 contains an assignment to z. Can you still evaluate the arguments of $g(e_1, e_2)$ in parallel? How? Or why not?