Problem Set 1 Solutions

Group Problem 1 [25]: Interpreters and Compilers

As seen in Lecture #2, there are two basic reasoning rules involving interpreters and translators (a.k.a. compilers):

1. The Interpreter Rule (I)

   \[
   \text{P-in-L program; L interpreter machine} \rightarrow \text{P machine} \quad (I)
   \]

2. The Translator Rule (T)

   \[
   \text{P-in-S program; S-to-T translator machine} \rightarrow \text{P-in-T program} \quad (T)
   \]

In practice, we will often elide the word “machine” for interpreter machines and translator machines. E.g., we will refer to an “L interpreter machine” as an “L interpreter”, and an “S-to-T translator machine” as an “S-to-T translator”. We will also often elide the word “program”; e.g., we will refer to a “P-in-L program” as “P-in-L”.

a. [10] You were given:

- a Java-to-C-in-Scheme compiler (i.e., a Java-to-C compiler written in Scheme);
- a Scheme-in-Pentium interpreter (i.e., a Scheme interpreter written in Pentium code);
- a C-to-Pentium-in-Pentium compiler (i.e., a C-to-Pentium compiler written in Pentium code);
- a Pentium-based computer (i.e., a Pentium interpreter machine).

Using the rules given above, here is a “proof” of how to execute a given program P-in-Java:

\[
\begin{align*}
\text{P-in-Java;} & \quad \text{Java-to-C-compiler-in-Scheme;} \\
& \quad \text{Scheme-interpreter-in-Pentium;} \\
& \quad \text{Pentium interpreter} \\
& \quad (I) \\
\hline
\text{P-in-Java;} & \quad \text{Java-to-C compiler;} \\
& \quad \text{Scheme interpreter} \\
& \quad (I) \\
\hline
\text{P-in-C} & \quad \text{C-to-Pentium-compiler-in-Pentium;} \\
& \quad \text{Pentium interpreter} \\
& \quad (I) \\
\hline
\text{P machine} & \quad (I)
\end{align*}
\]
b. [15] Suppose you are given the following:

- a OCAML-in-MIPS interpreter;
- a OCAML-to-MIPS-in-OCAML compiler;
- a MIPS-based computer.

i Here is how we can generate a OCAML-to-MIPS-in-MIPS compiler:

\[
\begin{array}{c}
\text{OCAML-to-MIPS-compiler-in-OCAML;}
\hline
\text{OCAML-interpreter-in-MIPS;}
\text{MIPS interpreter}
\end{array}
\]

\[
\begin{array}{c}
\text{OCAML-to-MIPS-compiler-in-OCAML;}
\hline
\text{OCAML interpreter}
\end{array}
\]

\[
\begin{array}{c}
\text{OCAML-to-MIPS-compiler-in-MIPS}
\end{array}
\]

The interesting part of this process is that the OCAML-to-MIPS-compiler-in-OCAML is used twice: (1) as a program to be executed and (2) as an input to the executed program (which, since it is an OCAML compiler, can accept any ocaml program as an input).

ii Suppose we accidentally delete the OCAML-in-MIPS interpreter after generating the OCAML-to-MIPS-compiler-in-MIPS. Then we can still execute any program P-in-OCAML as follows:

\[
\begin{array}{c}
P\text{-in-OCAML;}
\hline
\text{OCAML-to-MIPS-compiler-in-MIPS;}
\text{MIPS interpreter}
\end{array}
\]

\[
\begin{array}{c}
\text{P\text{-in-MIPS ;}
\text{MIPS interpreter}
\end{array}
\]

\[
\begin{array}{c}
P\text{ machine}
\end{array}
\]

Group Problem 2 [25]: Trusting Trust

a. [10]

In Stage II, two languages are considered: (1) the usual C programming language and (2) C+\v— an extension to C in which ‘\v’ is treated as the vertical tab character (which has ASCII value 11).

Suppose we are given the following:

- a C-to-Pentium-compiler (this is just a “black box”; we don’t care where it comes from);
- a C+\v-to-Pentium-compiler-in-C (Figure 2.3 in Thompson’s paper);
- a C+\v-to-Pentium-compiler-in-C+\v(source code to be Figure 2.2 in Thompson’s paper);
- a Pentium-based computer;

Our goal is to use the C+\v-to-Pentium-compiler-in-C+\vsource code to create a C+\v-to-Pentium-compiler-in-Pentium binary. This cannot be done directly using the C-to-Pentium compiler, but it can be accomplished via the following bootstrapping sequence:
Although C+\textbackslash v-to-Pentium-compiler-in-Pentium\textsubscript{1} and C+\textbackslash v-to-Pentium-compiler-in-Pentium\textsubscript{2} are functionally identical, the second one is generated by compiling a compiler written in C+\textbackslash v itself. The C+\textbackslash v-to-Pentium-compiler-in-C+\textbackslash v program doesn’t explicitly indicate the ASCII value associated with ‘\textbackslash v’. Where is that knowledge stored? In both copies of the C+\textbackslash v-to-Pentium-compiler-in-Pentium binary! The lesson here is that the compiler binary can contain knowledge that is not in the compiler source code but is preserved when the binary is used to compile the compiler source code to produce a new binary.

The above analysis assumes that a C-to-Pentium-compiler is given. Where did it come from? The very first C-to-Pentium compiler could not have been written in C. There must have been some sort of bootstrapping sequence, perhaps similar to the one above. For example, it could have been written in some other language X for which an interpreter existed. However, once the first C-to-Pentium compiler was created, it would then be possible to use it to compile a C-to-Pentium-in-C compiler (such as Figure 2.1 in Thompson’s article). Once this step is achieved, the compiler can “compile itself”.

b. [15]
The goal of Stage III is to show how a compiler generated from source code without a Trojan Horse can still insert a Trojan Horse into a login program. For this stage assume the following parts:

- a C-to-Pentium-compiler (again, just a “black box” we are given);
- a C-to-Pentium-compiler-in-C without Trojan Horses (Figure 3.1 in Thompson’s paper);
- a C-to-Pentium-compiler\textsubscript{TH}-in-C with two Trojan Horses (Figure 3.3 in Thompson’s paper);
- a login-in-C program with no Trojan Horse;
- a Pentium-based computer;

We shall use the subscript TH to indicate a that program contains a Trojan Horse. A C-to-Pentium-compiler\textsubscript{TH} has the “feature” that it can insert Trojan Horses when compiling source code that is an untrojaned login program or an untrojaned compiler program. That is, if P is a login or compiler program, it is as if there is a new rule:

The Trojan Horse Rule (TH)

\[
\frac{P\text{-in-C program; C-to-Pentium-compiler}_{TH}}{P_{TH}\text{-in-Pentium}} \quad (TH)
\]

We see again from the following proof that binary code can preserve knowledge that is not in the source code! In particular, once the C-to-Pentium\textsubscript{TH}-compiler-in-Pentium is created the first time, we can delete the C-to-Pentium\textsubscript{TH}-compiler-in-C source code. Remarkably, a new copy of C-to-Pentium\textsubscript{TH}-compiler-in-Pentium can be created by using the existing copy to compile a trojanless C-to-Pentium-in-C compiler.
Group Problem 3 \([20]\): OCAML Evaluation

Fig. 1 shows the result of evaluating the expressions and declarations in sequence. Here are a few notes:

- Many arithmetic and boolean binary operators in OCAML, such as `+`, `−`, `*`, `/`, `mod`, `&&`, and `||`, are *infix* operators: they must be placed *between* their two operand expressions. Such infix operators cannot be used directly as first-class function values. However, they can be converted to first-class functions by syntactically wrapping them in parentheses. For example:

```ocaml
# let app_3_5 f = f 3 5;;
val app_3_5 : (int -> int -> 'a) -> 'a = <fun>
# app_3_5 +;;
Characters 9-11: app_3_5 +;; (* can’t use an infix operator as a function value *)
```

```
# app_3_5 (+);; (* can use a parenthesis-wrapped operator as a function value *)
- : int = 8
```

- Parentheses-wrapped operators are *prefix* functions: they must be placed *before* their operands:

```ocaml
# 1 + 2;; (* addition operator is infix *)
- : int = 3
# (+) 1 2;; (* parenthesis-wrapped operator is a prefix function *)
```

```
# (++) 1 2;; (* parenthesis-wrapped operator is a prefix function *)
```

```
# (+) 1 2;; (* prefix function cannot be used in an infix position. *)
```

```
# (+) 1 2;; (* prefix function cannot be used in an infix position. *)
```

This expression is not a function, it cannot be applied

- Note that when the `*` operator is converted to a function value, it is necessary to leave at least one space between the operator and either paren. This is because OCAML interprets `(*)
as the “begin comment” symbol and *) as the “end comment” symbol. So the multiplication function can be written as (*), but cannot be written as (*), (*), or (*).

- Many students had trouble understanding the evaluation of the following expression:

```
app_3_5 (fun x y -> if x < y then (+) else (-));;
```

This is equivalent to

```
(fun x y -> if x < y then (+) else (-)) 3 5;;
```

which is in turn equivalent to

```
if 3 < 5 then (+) else (-);;
```

which simplifies to

```
if true then (+) else (-);;
```

which evaluates to

```
(+)
```

So the final result of this expression is the addition function! Note that this addition function is not applied to any numbers; in particular, it is not applied to 3 and 5.

To see that the value is indeed the addition function, we can apply the original expression to two integers. For example:

```
# (app_3_5 (fun x y -> if x < y then (+) else (-))) 10 20;;
- : int = 30
```

**Group Problem 4 [30]: OCAML Functions**

This problem involved translating the following JAVA class method into related OCAML functions:

```java
public static int f (int n) {
    if (n <= 2) {
        return n;
    } else {
        return n + f(n/2) + f(n/3);
    }
}
```

a. [10]: Translating f to OCAML

```
let rec f n =
    if n <= 2 then
        n
    else
        n + f(n/2) + f(n/3)
```

b. [20]: Counting nodes

Here is one way to express g:
Figure 1: A sequence of OCAML declarations and expressions.
let rec g n = 
  if n <= 2 then 
    (n,1) 
  else 
    let (a1,b1) = g(n/2) 
    and (a2,b2) = g(n/3) 
    in (n + a1 + a2, 1 + b1 + b2)

The key in this function definition is using `let` with pattern matching to “deconstruct” the pairs returned from the recursive calls of `g`. Alternatively, we could use the pair selection operators `fst` and `snd` directly to extract the components from the pairs:

```ocaml
let rec g n = 
  if n <= 2 then 
    (n,1) 
  else 
    let p1 = g(n/2) 
    and p2 = g(n/3) 
    in (n + (fst p1) + (fst p2), 1 + (snd p1) + (snd p2))
```

Indeed, undermine the hood OCaml “desugars” the `let`-based pattern matching idiom for pairs from the first definition of `g` to something like the form in the second definition of `g`.

It’s worth noting that you should avoid the following definition of `g`:

```ocaml
let rec g n = 
  if n <= 2 then 
    (n,1) 
  else 
    (n + (fst (g (n/2))) + (fst (g (n/3))), 1 + (snd (g (n/2))) + (snd (g (n/2))))
```

Although this version of `g` returns the correct answer, it takes significantly more time to do so than the first two versions. It is often the case that duplicating a function call in a recursive function can inflate the running time of the function by an exponential factor. For instance, it can turn a linear-time function into an exponential time function, and can turn a logarithmic-time function into a linear-time function. This explosion can be avoided by naming the result of the recursive function call and refering to the same name twice rather than making the same function invocation twice. This is the strategy we used in the first two versions of `g` above.

Let’s consider one more version of `g` — one that uses pattern matching on `n`:

```ocaml
let rec g n = 
  match n with 
  1 -> (1,1) 
  2 -> (2,1) 
  _ -> let (a1,b1) = g(n/2) 
    and (a2,b2) = g(n/3) 
    in (n + a1 + a2, 1 + b1 + b2)
```

Is this a correct definition of `g`? No. Although it gives the correct answer for positive values of `n`, it does not give the correct answer for 0 and negative values of `n`. Pattern matching is a wonderful feature, but sometimes an old-fashioned conditional is more appropriate.