CSE596 Handout Fall 2004

Supplement to Homer-Selman, Sections 3.6–3.7

1 Acceptable Programming Environments

Say that a programming language or environment is "acceptable" if there are two programs in the language that carry out the following respective tasks:

(a) `Exec(P,arg)`: If `P` is the string code of a legal program P that expects exactly one argument, return the result of executing $P(arg)$. If `P` is anything else, the call `Exec(P,arg)` can have arbitrary or undefined behavior, or can "bomb."

(b) `Subst(arg2,P)`: Suppose `P` is the code of a program P with two input variables. Return the code of a program P' with one input variable such that for all arguments $arg1$, $P'(arg1)$ behaves like $P(arg1, arg2)$. If `P` is anything else, Subst(arg2,P) can return an arbitrary "garbage" output.

There is a more general case where P expects $n + m$ input variables and you substitute $arg_1, \ldots, arg_m$ for the last m of them—and this is where the name "S-m-n" comes from. However, only the above "S-1-1" case is really needed. It makes no great difference if you define `Subst` to substitute for the first input variable(s) instead of the last.

Theorem 1.1 *Turing machines provide an acceptable programming environment.*

Proof. Any universal Turing machine can play the role of `Exec`. For `Subst`, if P is a Turing machine, let `Subst(arg2,P)` be a Turing machine P' such that on any input x, P' uses an opening routine of states that have $arg2$ "hard-wired" into them to change the tape to $x\#arg2$, and then connects to the start state of P. □

It is important to bear in mind that the output of the `Subst` function is the code for P', and has *nothing* to do with the operation of P' itself. I have actually done the `Subst` function for Turing machines several times in lecture, without making much of a fuss about it. You should picture the `Subst` function as being computed by some C or Java code R that manipulates Turing machines, not as being (computed by) a Turing machine itself. (Likewise, picture the "f" in a reduction as being computed by a C or Java routine—to spare yourself confusion with the Turing machines being operated on—and as proof that f is a total recursive function, sketch how the routine edits the TM code and explain that the routine always finishes the edits and halts.) Of course R can be converted into an equivalent Turing machine, but there's no reason you should need to be conscious about that. The fact that Turing machines have a `Subst` function is called the *S-m-n Theorem* for Turing machines.

For programming languages with *named variables* there are two main ways to implement `Subst`. In languages like Lisp and ML that grew out of Church's *lambda calculus*, the body of P' can be something like `lambda x => P(x,arg2)`. This is called *taking a closure* of the function P. (Note that the Turing machine P' above essentially does this. Lisp also has an explicit `eval` function that works like `Exec`. ML and the *Scheme* dialect of Lisp do not, but programmers sometimes hack around the lack by writing the string `P` or `(P arg)` to a file `P.txt` and then calling `use("P.txt");` or `(load "P.txt")`.) The other way is to do a *textual*

substitution on the named input variable that will hold the second argument. Then one changes the `Subst` function to

(b′) `Subst(arg,"z",P)`: If `z` is the second of two input parameters of `P`, and if `z` is not on the left side of an assignment in the body of `P`, return the code of the program P' obtained by removing `z` as an input parameter and substituting `arg` for every occurrence of `z` in the body of `P`. If `P` does not meet these conditions, `Subst(arg,"z",P)` can return garbage.

Here we may picture `Subst` as having three `String` arguments and returning the code of P' as a `String`. It is taken for granted that `z` is a token that can be parsed in `P`. This was the general form of the original *Gödel substitution function*, except that for Gödel, `P` was a formula of logic rather than a program. The restriction on `z` is explained by saying that `z` is a read-only input variable, like a `const` parameter in C++ or an `in` parameter in Ada, or *any* non-`ref` variable in ML. (In C/C++ lingo one says that `z` is used only as an *rvalue*.)

In my research I use a dialect of C called `Singular` that provides an explicit `Exec` function and some Perl-like functions for doing this kind of variable substitutions. Most implementations of C itself allow one to do these things via low-level system calls. For what follows, however, I will imagine a langauge that uses the basic C++ `iostream` library for input and output. The C++ statement `cin >> x >> z;` reads two strings from standard input and assigns them to the variables `x` and `y`. The statement `cout << x << 5;` outputs the value of `x` and then the literal number 5. I will assume that the only `cin` statement comes at the beginning of a program. Then `Subst(arg,"z",P)` is easy to implement by changing the " `>> z;`" at the end of the first statement to `;` and substituting `arg` for every other occurrence of `z`. I will also assume that my "mini-C++" also comes with a ready-made `Exec` command.

2 The Recursion Theorem

This theorem holds in *any* acceptable programming environment—via the proof in the text—but for sake of intuition, I will prove and illustrate it for my "mini-C++" system.

Theorem 2.1 *For any program P that you write with an uninitialized read-only variable* `mycode`*, there is an efficient and straightforward way to massage P into a program P' that works exactly the way you intended P to work, in which* `mycode` *is initialized to the string encoding of P'.*

Note that `mycode` is assigned not the code `P` of your P—which would be no great shakes—but the code of the P' you obtain. If the last line of P is `cout << mycode;`, then P' will print its own code. Technically your original P may not be a legal program if the compiler catches your not having initialized the variable `mycode`, but the first step in the proof takes care of that.

Proof. First, alter your program P by appending `>> mycode;` in place of the `;` in your opening `cin` statement, and let `P` represent the `String` encoding of this program. Let us use names for the following literal `String`s:

```
ker  =  "cin >> y >> u; return Exec(Exec(u,u), y);"
eff  =  "cin >> e; return Subst(e,"mycode",P);"
arg  =  "cin >> w; return Exec(eff,Subst(w,"u",ker));"
Ppr  =  "Subst(arg,"u",ker);"
```

Here I intend that the literal text of your `P` is substituted directly into the string `eff`, and that `ker` and `eff` are literally inserted into `arg`, and finally that `ker` and `arg` are inserted to give

you a final (pretty long) literal string `Ppr`. The internal quotes around `u` and `mycode` can be escaped via `\"` or `""` as appropriate. In relation to the Homer-Selman text:

- `ker` is the θ function on page 56, and works a little like a "micro-kernel."
- `eff` is the code of the total recursive function f in Theorem 3.10 on page 56;
- `Subst(w,"u",ker)` returns the value of the function g in the proof of Theorem 3.10 on page 56;
- `arg` is the code of the composition $f \circ g$ and equals k in the text; and
- `Ppr` will evaluate to $g(k)$ on p56, which the text calls n on page 56 and e_0 on page 57.

I claim that `Ppr` *is* the text of the desired program P'. For a classic (but general enough) "proof by example," let us suppose that your original program was

```
cin >> x;
cout << mycode;
return Rest_of_P(x);
```

Then `P = "cin >> x >> mycode; cout << mycode; return Rest_of_P(x);"` after the modification. To prove that the string `Ppr` gives the code of a P' that works equivalently to your original P with `mycode` assigned the code of P' (i.e., `Ppr` itself is assigned to `mycode`), let us trace the execution of P' on some argument `a`. We can use `Exec(Ppr,a)` itself to do this trace:

```
Exec(Ppr,a) = Exec(Subst(arg,"u",ker), a)
            = Exec("cin >> y; return Exec(Exec(arg,arg), y);", a)
```

In the crucial next step, executing the string on the argument `a` results in `a` being substituted for `y`. Note that `Exec` behaves much like `Subst` here. This leaves:

```
 = Exec(Exec(arg,arg), a)
 = Exec(Exec("cin >> w; return Exec(eff, Subst(w,"u",ker));", arg), a)
 = Exec(Exec(eff, Subst(arg,"u",ker)), a)
 = Exec(Exec("cin >> e; return Subst(e,"mycode",P);", Subst(arg,"u",ker)), a)
 = Exec(Subst( Subst(arg,"u",ker), "mycode", P), a)
 = Exec(Subst( Ppr, "mycode", P), a).
```

The last line tells you in full generality that what you get by running `Ppr` on input `a` is the same as what you get by running your original P on input `a` but with `Ppr` itself substituted for `mycode`, which is exactly what you wanted! In this case, you get

```
 = Exec(Subst( Ppr, "mycode", "cin >> x >> mycode; cout << mycode;
                               return Rest_of_P(x);"), a)
 = Exec("cin >> x; cout << Ppr; return Rest_of_P(x);", a).
```

This prints `Ppr` and then calls `Rest_of_P(a)`, whose body may contain further references to `mycode`. That's enough for the proof! □

If you make `Rest_of_P(a)` accept `a` if and only if `a == mycode`, then you have a program P' that accepts only its own code. In the text's traditional notation, this gives you n such that $W_n = \{ n \}$. The "modified program `P(x,mycode)`" above is the same as the text's "$\psi(x, e_0)$" on page 57, with its two arguments interchanged.