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) \textbf{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 \textbf{Exec}(P,arg) can have arbitrary or undefined behavior, or can “bomb.”

(b) \textbf{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 \(arg_1, arg_2\), \(P'(arg_1)\) behaves like \(P(arg_1, arg_2)\). If \(P\) is anything else, \textbf{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 \textbf{Subst} to substitute for the first input variable(s) instead of the last.

\textbf{Theorem 1.1} Turing machines provide an acceptable programming environment.

\textbf{Proof}. Any universal Turing machine can play the role of \textbf{Exec}. For \textbf{Subst}, if \(P\) is a Turing machine, let \textbf{Subst}(arg2,P) be a Turing machine \(P'\) such that on any input \(x\), \(P'\) uses an opening routine of states that have \(arg_2\) “hard-wired” into them to change the tape to \(x\#arg_2\), and then connects to the start state of \(P\).

It is important to bear in mind that the output of the \textbf{Subst} function is the code for \(P'\), and has \textit{nothing} to do with the operation of \(P'\) itself. I have actually done the \textbf{Subst} function for Turing machines several times in lecture, without making much of a fuss about it. You should picture the \textbf{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 \textbf{Subst} function is called the \(S-m-n\) \textit{Theorem} for Turing machines.

For programming languages with \textit{named variables} there are two main ways to implement \textbf{Subst}. In languages like Lisp and ML that grew out of Church’s \textit{lambda calculus}, the body of \(P'\) can be something like \texttt{lambda \textit{x} \Rightarrow P(x,\textit{arg2})}. This is called \textit{taking a closure} of the function \(P\). (Note that the Turing machine \(P'\) above essentially does this. Lisp also has an explicit \texttt{eval} function that works like \textbf{Exec}. ML and the \textit{Scheme} dialect of Lisp do not, but programmers sometimes hack around the lack by writing the string \(P\) or \((P \ \texttt{arg})\) to a file \texttt{P.txt} and then calling \texttt{use(“P.txt”)}; or \texttt{(load “P.txt”).} The other way is to do a \textit{textual}
substitution on the named input variable that will hold the second argument. Then one changes the \texttt{Subst} function to

(b') \texttt{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 \( \text{arg} \) for every occurrence of \( z \) in the body of \( P \). If \( P \) does not meet these conditions, \texttt{Subst(arg,"z",P)} can return garbage.

Here we may picture \texttt{Subst} as having three \texttt{String} arguments and returning the code of \( P' \) as a \texttt{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 \textit{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 \texttt{const} parameter in C++ or an \texttt{in} parameter in Ada, or any \texttt{non-ref} variable in ML. (In C/C++ lingo one says that \( z \) is used only as an \texttt{rvalue}.)

In my research I use a dialect of C called \texttt{Singular} that provides an explicit \texttt{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 language that uses the basic C++ \texttt{iostream} library for input and output. The C++ statement \texttt{cin \gg x \gg z;} reads two strings from standard input and assigns them to the variables \( x \) and \( y \). The statement \texttt{cout \ll x \ll \text{5};} outputs the value of \( x \) and then the literal number \( \text{5} \). I will assume that the only \texttt{cin} statement comes at the beginning of a program. Then \texttt{Subst(arg,"z",P)} is easy to implement by changing the \texttt{" \gg z;"} at the end of the first statement to \texttt{;} and substituting \( \text{arg} \) for every other occurrence of \( z \). I will also assume that my “mini-C++” also comes with a ready-made \texttt{Exec} command.

2 \hspace{1em} \textbf{The Recursion Theorem}

This theorem holds in \textit{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.

\textbf{Theorem 2.1} \textit{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 \texttt{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 \texttt{cout \ll 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 \texttt{mycode}, but the first step in the proof takes care of that.

\textbf{Proof.} First, alter your program \( P \) by appending \texttt{mycode;} in place of the \texttt{;} in your opening \texttt{cin} statement, and let \( P \) represent the \texttt{String} encoding of this program. Let us use names for the following literal \texttt{Strings}:

\begin{verbatim}
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);"
\end{verbatim}

Here I intend that the literal text of your \( P \) is substituted directly into the string \texttt{eff}, and that \texttt{ker} and \texttt{eff} are literally inserted into \texttt{arg}, and finally that \texttt{ker} and \texttt{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 \( \theta \) 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;
- \( \text{Subst}(w, "u", \text{ker}) \) 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

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

Then \( P = \text{"cin} >> x >> \text{mycode}; \text{cout} << \text{mycode}; \text{return Rest_of_P(x);"} \) 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:

```c++
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:

```c++
    = 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

```c++
    = 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 \( a == \text{mycode} \), 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 \( P(x, \text{mycode}) \)” above is the same as the text’s “\( \psi(x, e_0) \)” on page 57, with its two arguments interchanged.

3