Scheme Computability, Part II

1 Meta-Circular Interpreters

A high point of the Abelson/Sussman text *Structure and Interpretation of Computer Programs* (SICP) is the construction of a Scheme interpreter in Scheme—"meta-circular" interpreter. Of course from a mathematical point of view, such an interpreter is simply circular: it's of no use as a mathematical definition of anything unless Scheme evaluation has been already been independently defined—such as by our Substitution Model. Nevertheless, the construction serves a sound pedagogical purpose as a first illustration of interpreter design for variants and extensions of Scheme.

Meta-circular interpreters play a similar role in computability theory: the use of helpful programming features that are not directly available in Scheme—for example, the ability to evaluate two or more expressions in parallel (cf. Notes 3, Cor. 9.2)—can be justified by extending a meta-circular interpreter for Scheme to simulate these features.

Definition 1.1. A *meta-circular interpreter* for Scheme is a Scheme expression, *Meval*, such that for all S-expressions, M,

$$M\downarrow \quad \text{iff} \quad (\textit{Meval}\ \ 'M)\downarrow,$$

and,

$$\text{output}(M) = \text{output}((\textit{Meval}\ \ 'M)).$$

Let's review some of the several approaches available for constructing meta-circular interpreters. First, there is the basic one in SICP[1]. Next, Abelson/Sussman also exhibit a compiler written in Scheme which compiles Scheme expressions to register machine code formatted as S-expressions. They also show how to write a register machine simulator in Scheme. So this second meta-circular interpreter could essentially be expressed in the form

$$\texttt{(lambda (s) (}\textit{simulate-register-machine}\ \texttt{(}\textit{compile}\ \texttt{s)))}.$$

A third, altogether different meta-circular interpreter is our Scheme implementation of the Substitution Model. Inspection of this interpreter reveals that side-effects are used in inessential ways – mainly to control the interpreter's read-eval-print-loop, global environment, and I/O behavior. The pattern-matching and rewrite-rule portions of the interpreter that implement the kernel Substitution Model interpreter are side-effect free, that is,they are *functional*.

So each of these three approaches leads to a proof that interpreters exist:

[1] The basic Abelson/Sussman interpreter, however, does not handle `call/cc`. It has to be rewritten in tail-recursive style, as in 6.001 Project 3 from Fall '02, to support the call/cc control abstraction

Theorem 1.2. *There is a meta-circular interpreter, Meval, for Scheme.*[2]

A full proof of Theorem 1.2 should not only produce an *Meval* expression, but should also prove that it satisfies Definition 1.1. Developing such a correctness proof is an important exercise in program verification. But as we observed, we are familiar with meta-circular interpreters, and since most of the ideas involved in such a correctness proof are not needed elsewhere in developing Scheme computability theory, we will not actually prove that these interpreters work as specified.

Now we can confirm an important claim made in Part I:

Corollary 1.3. Halts *is recognizable.*

Proof. *Meval* is a recognizer for Halts. □

A meta-circular interpreter implies that there is a single "universal" recognizable set, $\mathcal{U}$, that codes all recognizable sets.

Theorem 1.4.

$$\mathcal{U} ::= \{(M\ \ N) \mid N \in \mathcal{H}_M\}$$

is a recognizable set.

Proof. Given input $(M\ \ N)$, a recognizer for $\mathcal{U}$ need only simulate the application of M to N, which just means evaluating the combination $(M\ \ N)$. So a recognizer for $\mathcal{U}$ would be

```
(lambda (s)
 (if (and (list s) (= (length s) 2))      ; s = (M  N)
     (Meval s)                            ; apply M to N
     Ω0)                                  ; otherwise diverge
```

□

2 Step by Step Interpretation

We claimed earlier that if a set and its complement were both recognizable, then the set is decidable.

The proof of the claim is fairly easy: if H is a recognizer for some set $\mathcal{S}$ and G is a recognizer for its complement, then to decide whether $S \in \mathcal{S}$, just evaluate $(H\ \ 'S)$ and $(G\ \ 'S)$ in parallel. Exactly one of these expressions will halt; if it's the first, then $S \in \mathcal{S}$, otherwise $S \notin \mathcal{S}$.

[2] It is cheating a little to appeal to the Abelson/Sussman approaches in justifying Theorem 1.2, because in the Abelson/Sussman interpreter, mutable lists (`set-car!`, `set-cdr!`) are used to implement environments. However, our mathematical model of Scheme, namely the Substitution Model, does not model mutable lists (`set-car!`, `set-cdr!`). Consequently, this interpreter technically does not provide an *Meval* expression in the part of Scheme we have mathematically defined. However, the only lists that need to be mutated by the SICP evaluator are the ones used to represent bindings in an environment frame. The representation of bindings can easily be replaced with a procedural representation described elsewhere in SICP, where `set!` is used to simulate the list mutators. A similar remark applies to the Abelson/Sussman register machine compiler and simulator. So in this way, their constructions would, in fact, yield the required *Meval* expression.

The only problem is that this procedure involves running two Scheme evaluations in parallel, but we know that Scheme has no such run-in-parallel operation. But it can simulate the parallelism by alternating between simulations of the two evaluations.

To manage this, we need the ability to run a metacircular interpreter for a given number of "steps." For definiteness, we'll define a step to be a Substitution Model rule application, and let step be the total function taking an S-expression and a natural number argument such that

$$\text{step}(M,n) = \begin{cases} \texttt{(printable-flag } \text{output}(M)\texttt{)} & \text{if } M \stackrel{\leq n}{\Rightarrow} V \text{ for some } \langle\text{printable-sval}\rangle,\ V, \\ \texttt{(not-printable-flag)} & \text{if } M \stackrel{\leq n}{\Rightarrow} V \text{ for some, } V \notin \langle\text{printable-sval}\rangle \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

Theorem 2.1. *There is a stepping meta-circular interpreter for Scheme.*

Proof. Modify the Substitution Model interpreter to take an extra argument natural number, n, in addition to an expression argument, M. At each rule application increment a counter until M gets rewritten to a final value or there have been n rule applications. □

Now we can complete the proof of:

Theorem 2.2. *$\mathcal{S}$ is decidable iff $\mathcal{S}$ and $\overline{\mathcal{S}}$ are recognizable.*

Proof. We already proved the left to right implication in Part I.

For the other direction, let H be a recognizer for $\mathcal{S}$ and G be a recognizer for its complement. As suggested above, to decide of $S \in \mathcal{S}$, simulate H and G running on S in parallel. Exactly one of them will halt on S, and the answer is true or false according to which halts first. Namely, let *Step* be an expression that computes the step function. Then a decider for $\mathcal{S}$ is:

```
(lambda (s)
  (letrec
    ((try (lambda (n)
            (cond ((Step (list 'H s) n) #t)
                  ((Step (list 'G s) n) #f)
                  (else (try (+ n 1)))))))
     (try 0))
```

Note that correctness of this decider depends on the fact that neither H nor G causes side-effects on lists to which they are applied. This property is guaranteed in our kernel Scheme, since we omitted operations such as `set-car!` with side-effects on lists. [3]

□

Similarly, we can now prove another fact claimed in Part I:

Theorem 2.3. *Every nonempty recognizable set is computably countable.*

[3] A decider modified so that H and G were always applied to fresh copies of their argument would work even if H and G were allowed to side-effect included lists.

Proof. Let $\mathcal{S}$ be a recognizable set with recognizer H, let S_0 be some S-expression in $\mathcal{S}$. We count $\mathcal{S}$ by calculating the nth pair consisting of an expression M and a natural number m. If (`H M`) halts within m steps, we let M count as the nth element of $\mathcal{S}$; otherwise we let S_0 be the nth element.

Namely, let C compute a counting function for the set of all S-expressions. Then a counting function for $\mathcal{S}$ is computed by

```
(lambda (n)
 (let ((Mm (C n)))
  (if (and (list?  Mm)
           (>= (length Mm) 2)
           (step (list 'H (car Mm)) (cdr Mm)))
      (car Mm)
      'S0)))
```

□

Problem 1. Prove that the recognizable sets are closed under union.

Problem 2. **(a)** Let $f : \langle\text{s-expr}\rangle \to \langle\text{s-expr}\rangle$ be a computable partial function. Prove that if $\mathcal{S}$ is recognizable, then so is $f(\mathcal{S})$.

(b) Prove that an infinite set $\mathcal{S}$ is recognizable iff $\mathcal{S} = f(\mathbb{N})$ for some one-to-one (that is,injective) computable function, f, that is total on $\mathbb{N}$.

Problem 3. For sets $\mathcal{A}, \mathcal{B}$ of S-expressions, let

$$\mathcal{A} \uplus \mathcal{B} ::= \{\texttt{(0 } A\texttt{)} \mid A \in \mathcal{A}\} \cup \{\texttt{(1 } B\texttt{)} \mid B \in \mathcal{B}\}\,.$$

Prove that if $\mathcal{A}$ is not decidable, then neither $\mathcal{A} \uplus \overline{\mathcal{A}}$ nor its complement is recognizable.

3 The Recursion Theorem

The Recursion Theorem synthesizes the self-application techniques used in Part I of these Notes. It guarantees the existence of fixed points, up to observational equivalence, of computable transformations of programs.

Theorem 3.1. *(The Recursion Theorem) Let f be a total computable function on S-expressions. Then there is a Scheme expression,* fix(f)*, such that*

$$\text{fix}(f) \equiv f(\text{fix}(f)).$$

Here is one weird interpretation of what the Recursion Theorem says: many compilers perform "source to source" transformations of programs into equivalent ones that are better suited for compilation. Suppose we try to define an opposite kind of computable program transformation, f, that *never* preserves equivalence. Namely, f is supposed to transform every expression into one that is *not* observationally equivalent to the original. The Recursion Theorem implies there is no such computable f. This is another example of the phenomenon demonstrated by Scott's Rice's Theorems in Part I: nontrivial properties of expression evaluation are undecidable. In this case, the Recursion Theorem implies that no procedure can even determine enough about an arbitrary expression simply to yield a different value.

A simple application of the Recursion Theorem implies the existence of the kind of self-reproducing programs constructed in Part I. Namely, let

$$f(M) ::= \texttt{(quote } M\texttt{)}.$$

Then by the Recursion Theorem, there is a Scheme expression, $\text{fix}(f)$, such that

$$\text{fix}(f) \equiv \texttt{(quote } \text{fix}(f)\texttt{)}. \tag{1}$$

So in particular, the outputs of these expressions, if any, must be the same:

$$\text{output}(\text{fix}(f)) = \text{output}(\texttt{'}\text{fix}(f)). \tag{2}$$

But since

$$\text{output}(\texttt{'}S) = S \tag{3}$$

for all S-expressions, S, we conclude from (2) and (3) that

$$\text{output}(\text{fix}(f)) = \text{fix}(f),$$

that is, $\text{fix}(f)$ is self-reproducing.

Proof. (The Recursion Theorem)

We begin by observing that there is a *total* computable function, g, with the property that

$$g(S) \equiv f(\text{output}(\texttt{(}S\ \texttt{'}S\texttt{)})) \tag{4}$$

for all expressions, S, such that $\text{output}(\texttt{(}S\ \texttt{'}S\texttt{)})$ is defined.

In fact, we can exhibit an expression, G, computing, g. Namely,

$$G ::= \texttt{(lambda (s) (list 'F (list s (list 'quote s))))}.$$

Now since g is total, $g(G)$ is defined. So by definition of G computing g, we have that

$$\text{output}(\texttt{(}G\ \texttt{'}G\texttt{)}) = g(G). \tag{5}$$

In particular, $\text{output}(\texttt{(}G\ \texttt{'}G\texttt{)})$ is defined, so we can let S be G in (4), to obtain

$$\begin{aligned} g(G) &\equiv f(\text{output}(\texttt{(}G\ \texttt{'}G\texttt{)})) && \text{by (4)} \\ &= f(g(G)) && \text{by (5).} \end{aligned} \tag{6}$$

So define $\text{fix}(f) ::= g(G)$. Now (6) implies

$$\text{fix}(f) \equiv f(\text{fix}(f))$$

as required. □

Problem 4. **(a)** Show that if $M \equiv N$, then $\mathcal{H}_M = \mathcal{H}_N$.

(b) Show that there is a "self-recognizing" expression, Q, namely, an expression such that

$$\mathcal{H}_Q = \{Q\}\,.$$

Problem 5. Describe a procedure to compute fix(f) given any expression, F, that computes a total function, f.

4 Minimal Scheme Expressions

You may have heard of the "Liar" paradox: if I say "I am lying," am my lying or not? A slightly less obvious paradox comes out of the following definition:

> Let n be the smallest nonnegative integer that is not definable by an English sentence of fewer than 112 characters.

The paradox here is that this definition of n has only 111 characters.

The resolution of the paradox is easy: we should have rejected this alleged definition of n from the start because it refers to ill-defined concepts such as "English sentence" and definability by such sentences. But an interesting theorem arises from this paradox when we replace reference to English sentences by reference to some given infinite set of Scheme expressions, and replace reference to the meaning of a sentence by reference to the value of the expression.

Namely, if we had a way to generate the expressions in a given infinite set, then we could write a program that searched for, and then evaluated, the first generated expression bigger than the program we write. The Recursion Theorem will justify our writing a program that contains its own size as a known constant. It follows that our program has the same value, if any, as one of the generated expressions, but our program is smaller. Another way to say this is that there can't be a generating procedure for an infinite set of minimum-size expressions.

Definition 4.1. Define the *size*, size(S), of an S-expression, S, to be the number of occurrences of characters, including blanks, in S. A closed Scheme expression, M, is *minimal* iff output(M) is defined, and for any closed expression, N, if size$(N) <$ size(M), then output$(N) \neq$ output(M). Let $\text{MIN}_{\text{Scheme}}$ be the set of minimal expressions.

Definition 4.2. A set, $\mathcal{S}$, of S-expressions is *immune* iff it is infinite but has no infinite recognizable subset.

Theorem 4.3. *MIN_{Scheme} is immune.*

Proof. We begin with the observation that Scheme's builtin procedures for converting symbols and numbers to strings make it easy to define a procedure for mapping any S-expression to its representation as a string. Since there is also a builtin `string-length` procedure, it follows that the function, size$(.)$, is computable.

Now let $\mathcal{H}$ be any infinite recognizable set of S-expressions. We will show that there is an expression in $\mathcal{H}$ that is not minimal. It follows that $\mathcal{H} \not\subseteq \text{MIN}_{\text{Scheme}}$, which proves that $\text{MIN}_{\text{Scheme}}$ is immune.

To find the non-minimal expression in $\mathcal{H}$, define $m(S)$ to be the first expression in $\mathcal{H}$ that is bigger than S. That is, letting h be a computable counting function for $\mathcal{H}$,

$$m(S) ::= h(\min\{n \in \mathbb{N} \mid \text{size}(h(n)) > \text{size}(S)\}).$$

Now since h and size$(.)$ are computable, so is m. Also, by definition of m,

$$m(S) \in \mathcal{H} \tag{7}$$

and

$$\text{size}(m(S)) > \text{size}(S) \tag{8}$$

for all S-expressions, S.

If $m(S)$ does not converge to a printable value, then $m(S)$ is a non-minimal expression in $\mathcal{H}$, and we are done. So we may assume that output$(m(S))$ is defined for all S-expressions, S.

Now by the Recursion Theorem, there is a Scheme expression $R ::= \text{fix}(m)$ such that

$$R \equiv m(R),$$

and since output$(m(R))$ is defined, we have

$$\text{output}(R) = \text{output}(m(R)). \tag{9}$$

But size$(m(R)) >$ size(R) by (8), and we conclude from (7) and (9) that $m(R) \in \mathcal{H}$ is not minimal. □

All the undecidable sets we identified before $\text{MIN}_{\text{Scheme}}$ could be proved undecidable by many-one-reducing Halts, or its complement, to them. This is not the case for any immune set, as demonstrated in the following problems.

Problem 6. **(a)** Prove that if $\mathcal{S}_1 = f^{-1}(\mathcal{S}_2)$, for some partial function, f, then $\mathcal{S}_1 = f^{-1}(f(\mathcal{S}_1))$.

(b) Conclude that if $\mathcal{S}_1$ is a recognizable set that is undecidable and $\mathcal{S}_1 \leq_{\text{m}} \mathcal{S}_2$, then $\mathcal{S}_2$ has a recognizable *sub*set that is undecidable.

(c) Conclude that Halts $\not\leq_{\text{m}} \text{MIN}_{\text{Scheme}}$.

Problem 7. **(a)** Show that every productive set has an infinite recognizable subset. *Hint:* Start by applying the witness function to a recognizer for the empty set.

(b) Conclude that $\overline{\text{Halts}} \not\leq_{\text{m}} \text{MIN}_{\text{Scheme}}$.

On the other hand, there is a natural sense in which the halting problem and the minimality problem for Scheme expressions are "computationally equivalent." Namely, if we had an "oracle" procedure available to decide membership in Halts, then we could define a Scheme procedure that decides $\text{MIN}_{\text{Scheme}}$ by making calls to the oracle, and vice-versa. Of course we know an oracle-procedure for Halts can't be a Scheme procedure, but we needn't be concerned with how the oracle works as long as it is guaranteed to provide correct answers to membership queries.

Formally, we add a new procedure constant, `oracle?`, to the grammar for Scheme expressions to obtain the *oracle-Scheme* expressions. Now let $\mathcal{S}$ be any set of S-expressions. We can extend the Substitution Model to apply to oracle-Scheme expressions by adding two simple rules for the oracle:

$$\texttt{(oracle?}\ \ V\texttt{)} \to \texttt{\#t} \qquad \text{if print}(V) \in \mathcal{S}, \qquad (\mathcal{S}\text{-\#t})$$

$$\texttt{(oracle?}\ \ V\texttt{)} \to \texttt{\#f} \qquad \text{otherwise.} \qquad (\mathcal{S}\text{-\#f})$$

If an oracle-Scheme expression computes a function, f, when the Substitution Model is extended with the $\mathcal{S}$ oracle rules above, then f is said to be computable *relative to* $\mathcal{S}$, or $\mathcal{S}$*-computable*, for short. We define $\mathcal{S}$*-decidability*, $\mathcal{S}$*-recognizability*, *etc.*, similarly.

Another way to say that $\mathcal{S}_1$ is decidable relative to $\mathcal{S}_2$ is to say that $\mathcal{S}_1$ is *Turing-reducible* to $\mathcal{S}_2$, indicated with the notation

$$\mathcal{S}_1 \leq_{\text{T}} \mathcal{S}_2.$$

So the remark above that Halts and $\text{MIN}_{\text{Scheme}}$ are "computationally equivalent" has the precise meaning that each of these sets is Turing-reducible to the other.

Problem 8. **(a)** Prove that $\text{MIN}_{\text{Scheme}} \leq_{\text{T}} \text{Halts}$.

(b) Show conversely that $\text{Halts} \leq_{\text{T}} \text{MIN}_{\text{Scheme}}$.

Problem. There is a recognizable set, $\mathcal{C}$, whose complement is immune. To generate successive elements of $\mathcal{C}$, start by generating all pairs `(`M N`)` such that $M \downarrow N$ and $2\,\text{size}(M) \leq \text{size}(N)$. Then filter this sequence so that no two pairs have the same M.

Prove that $\overline{\mathcal{C}}$ is immune. *Hint:* Don't forget to prove $\overline{\mathcal{C}}$ is infinite.

5 Builtin `eval` [Optional]

An `eval` procedure has become an official part of Scheme in the Revised[5] Scheme Manual. It requires an extra environment argument, but the choice of environment is essentially limited to being the Scheme initial environment, so we shall omit it in our discussion. Note that it is not included in the Scheme kernel used in the Substitution Model.

The new `eval` operator is a builtin Scheme procedure officially satisfying a specification, that, at least for closed expressions, is simpler to express, but a little more demanding to achieve, than Definition 1.1 for meta-circular interpreters. Namely,

$$\texttt{(eval '}M\texttt{)} \equiv M \qquad \text{(espec)}$$

for all *closed* Scheme expressions, M.

The meta-circular interpreters described above satisfy this condition only when the value of M is printable—and when M diverges—but not when the value is non-printable. The reason is that each of these interpreters uses its own representation of procedures and environments, and needs a corresponding `apply` procedure adapted to those representations. But the condition (espec) above requires that if M converges to a procedure value, then `(eval '`M`)` must return an *actual Scheme procedure* equivalent to M and have caused the same side-effects on the environment as evaluation of M.

But it is actually simple to modify an SICP-style evaluator to use actual procedures instead of lists to represent procedure values. This allows the builtin `eval` to be treated as syntactic sugar. Specifically, the basic SICP `meval` uses a constructor `make-procedure` to represent a compound procedure as a list of three things: the procedure parameters, the procedure body, and the procedure environment. If we simply modify the constructor definition:

```
(define (make-procedure params body-expr env)
  (lambda vals
     (meval body-expr (extend-environment params vals env)))),
```

then the ordinary Scheme `apply` procedure can be used instead of the interpreter's `m-apply`. The result is a defined `eval` procedure that satisfies (espec).

The specification (espec) does not apply in the presence of free variables because it implies that `(eval '`M`)` should reference the dynamic evaluation environment instead of the top-level user environment. Evaluation of quoted free variables is problematic in other ways, *e.g.*, the property that

$$\texttt{(lambda (}x\texttt{) }M\texttt{)} \equiv \texttt{(lambda (}y\texttt{) }M[x := y]\texttt{)}$$

for y not free in M, no longer holds. For example, M might be:

```
(lambda (xx) (eval '(string->symbol (string-append "x" "x")))).
```

Problem 9. Notice that the new `make-procedure` procedure above uses the `(lambda val ...)` form to define a procedure taking a variable number of arguments. Prove that a meta-circular evaluator satisfying (espec) for closed expressions, M, could not be written in the sublanguage of kernel Scheme that does not include this form. *Hint:* Prove that the maximum number of parameters of any lambda expressions appearing in an expression is preserved by the Substitution Model rewrite rules.

Problem 10. **(a)** Prove that no meta-circular interpreter can satisfy (espec) for all open expressions M. *Hint:* Observe that if $M \to N$, then the free variables of N are a subset of the free variables of M.

(b) Explain how, for any finite set of variables, to construct a meta-circular interpreter satisfying (espec) for all Scheme expressions, M, whose free variables are contained in the finite set.

6 Second Incompleteness Theorem [Optional]

Draft

The First Incompleteness Theorem states that any sound proof system is incapable of proving some true assertion; there is even a way to contruct such an unprovable truth given the rules for the system. On the other hand, the "missing truth" is contrived just for the purpose of witnessing

incompleteness, and there is no reason to suppose that it is of any independent interest. The Second Incompleteness Theorem addresses this concern by identifying an "interesting" truth that can't be proved. Namely, no sound proof system can prove its own soundness!

Actually, the Second Incompleteness Theorem is even stronger: there is a weaker condition than soundness called "consistency," and the Second Theorem shows that no consistent proof system can even prove its own consistency.

Definition 6.1. A set of S-expressions is called *inconsistent* iff the equation $(N \;=\; \Omega_0)$ is in the set for some closed expression N such that $N{\downarrow}$. The set is *consistent* iff it is not inconsistent.

A proof system for Scheme equivalences is called consistent iff its set of provable equations is consistent.

An inconsistent proof system is certainly unsound. Conversely, if a system is consistent, and $(N \;=\; \Omega_0)$ is provable for some closed expression, N, then N must diverge, and so the equation must be true. In other words, a consistent system is sound for equations of the restricted form $(N \;=\; \Omega_0)$ (though it need not be sound in general).

Now given the recognizer, P, for the provable assertions in some proof system, there is a way to detect when the system is inconsistent. Namely,

Lemma 6.2.

$$\mathcal{I} ::= \{P \mid \mathcal{H}_P \text{ is inconsistent}\}$$

is recognizable.

Proof. To see if $\mathcal{H}_P$ is inconsistent, test in parallel all expressions N to see if both N converges and $(N \;=\; \Omega_0)$ is provable. This parallel process can be simulated by enumerating all the S-expressions, looking for one of the form $(N \;\; l \;\; m)$ such that the equation $(N \;=\; \Omega_0)$ is provable in l steps and N converges in m steps.

□

So, letting I be the recognizer for $\mathcal{I}$,

$$\begin{array}{lll} \mathcal{H}_P \text{ is consistent} & \text{iff} & P \notin \mathcal{I} \\ & \text{iff} & (I \;\; 'P){\uparrow} \\ & \text{iff} & (I \;\; 'P) \equiv \Omega_0. \end{array}$$

In other words, the assertion that the proof system given by P is consistent can be expressed by the single equation

$$((I \;\; 'P) \;=\; \Omega_0).$$

Let's call this equation *consis-eq*(P).

Let $\mathcal{P}_0$ be the equations of the form $(M \;=\; \Omega_0)$ in $\mathcal{H}_P$. Note that there is a total computable function, g, such that $\mathcal{P}_0 = \mathcal{H}_{g(P)}$. We leave it to the reader to exhibit a Scheme expression, G, computing g.

Now suppose *consis-eq*(P) is true. Since the proof system described by P is consistent, it follows that $\mathcal{P}_0$ is a subset of $\mathcal{E}_0$, the valid Scheme equivalences of the form $(M \;=\; \Omega_0)$.

This means that if w is the witness function for $\mathcal{E}_0$, then $w(g(P))$ will be in $\mathcal{E}_0$ but not in $\mathcal{H}_P$. So we have shown that if *consis-eq*(P) is true, then $w(g(P))$ is an equation of the form $(M_P \texttt{ = } \Omega_0)$ that is true (because it's in $\mathcal{E}_0$) and not provable in the proof system given by P. In fact, given P, we can actually find this expression M_P. In other words, we have proved

Lemma 6.3. *There is a total computable function, f, such that if $\mathcal{H}_P$ is consistent, then the equation $(f(P) \texttt{ = } \Omega_0)$ is true but not in $\mathcal{H}_P$.*

Now here's the wonderful observation (originally due to the Logician Kurt Gödel, for a class of proof systems for logical assertions about integers rather than Scheme equivalences): if we have a *strong enough* proof system for proving equations, then equations that we can prove by mathematical reasoning like that in the previous paragraph could also be proved formally in the system. In other words, if the equation *consis-eq*(P) was not only true, but was also a provable equation of the system described by P, then the reasoning we used above to conclude that the equation $(f(P) \texttt{ = } \Omega_0)$ is true could be mirrored by a proof in the system. Therefore, $(f(P) \texttt{ = } \Omega_0)$ would also be provable, contradicting the fact that it is not provable. So we conclude that *consis-eq*(P) cannot be provable:

Theorem 6.4 (Second Incompleteness Theorem for Scheme equivalence). *If a proof system is consistent and strong enough, then it cannot prove its own consistency. That is, if P is a recognizer for a strong enough, consistent set of Scheme equations, then consis-eq(P) is true, but is not in $\mathcal{H}_P$,*

Of course we still need to define "strong enough."

Definition 6.5. A set, $\mathcal{P}$, of equations between Scheme expressions is *strong enough* providing:

- if $M \xrightarrow{*} N$ then $(M \texttt{ = } N) \in \mathcal{P}$,
- $\mathcal{P}$ is closed under equational reasoning, including the congruence rule that for any context C, if $(M \texttt{ = } N) \in \mathcal{P}$ then $(C[M] \texttt{ = } C[N]) \in \mathcal{P}$.
- $\texttt{((begin } \Omega_0 \texttt{ } M\texttt{) = } \Omega_0) \in \mathcal{P}$,
- $\texttt{((begin (}\textit{Meval}\texttt{ '}M\texttt{) 1) = (begin } M \texttt{ 1))} \in \mathcal{P}$,
- the equation (valid-eq) below is in $\mathcal{P}$.

Lemma 6.6. *The the following equation describes a valid observational equivalence:*

$$\texttt{((begin (}I\texttt{ '}P\texttt{) (}\textit{Meval}\texttt{ (}F\texttt{ '}P\texttt{)) 1) = (begin (}\textit{Meval}\texttt{ (}F\texttt{ '}P\texttt{)) 1))}, \quad \text{(valid-eq)}$$

where F computes the total computable function, f, of Lemma 6.3.

Proof. There are two cases:

First, suppose that $P \in \mathcal{I}$, that is, $\texttt{(}I\texttt{ '}P\texttt{)}\downarrow$. Hence,

$$\texttt{(begin (}I\texttt{ '}P\texttt{) (}\textit{Meval}\texttt{ (}F\texttt{ '}P\texttt{)) 1)} \xrightarrow{*} \texttt{(begin (}\textit{Meval}\texttt{ (}F\texttt{ '}P\texttt{)) 1)},$$

so (valid-eq) is will be true since Substitution Model rewriting preserves observational equivalence.

Second, suppose that $P \notin \mathcal{I}$, that is, $(I\ 'P)\uparrow$. But this means that $\mathcal{H}_P$ is consistent, so $(f(P)\ =\ \Omega_0)$ is true, which means that $f(P)\uparrow$ and therefore $(\mathit{Meval}\ \ (F\ 'P))\uparrow$. Hence both sides of (valid-eq) diverge, and again (valid-eq) will be true.

□

Problem 11. Complete the proof by contradiction of the Second Incompleteness Theorem, by proving that if $\mathcal{H}_P$ is strong enough, and $\mathit{consis\text{-}eq}(P) \in \mathcal{H}_P$, then $(f(P)\ =\ \Omega_0) \in \mathcal{H}_P$. **Warning:** The definition of "strong enough" above is probably not strong enough `:-)`. I'm not even sure if it's on the right track.