Scheme Computability, II

1 Meta-Circular Interpreters and Universal Machines

Definition 1. A *meta-circular interpreter for Scheme* is a closed Scheme expression, ⟨smeval⟩, such that for all Scheme expressions, M,

1. `(`⟨smeval⟩ `'`M`)` halts iff M halts, and
2. if ev-val(M) is a ⟨printable value⟩, then ev-val(`(`⟨smeval⟩ `'`M`)`) = ev-val(M).

Theorem 1. There is a meta-circular interpreter for Scheme.

Proof: The Abelson/Sussman textbook exhibits a meta-circular interpreter. ■

Abelson/Sussman also exhibit a compiler written in Scheme which compiles Scheme bodies to register machine code formatted as S-expressions. They also show how to write a register machine simulator in Scheme. So another meta-circular interpreter for Scheme is

`(lambda (qs) (`⟨simulate-register-machine⟩ `(`⟨compile⟩ `qs)))`.

A third, very different meta-circular interpreter based on term rewriting is available on the course handouts web-page in the directory `/rewriting-scheme-code`.

Problem 1. Explain why none of these meta-circular interpreters satisfy a stronger and more natural version of of Definition 1, namely,

> **Strong Interpretation.** For all Scheme expressions, M,
>
> `(`⟨smeval⟩ `'`M`)` $\cong M$.

Problem 1.5 Show that if a procedure, ⟨smeval⟩, satisfies the Strong Interpretation condition, then it is indeed a meta-circular interpreter according to Definition 1.

Procedure 1.6

(a) Let p be a Scheme procedure, and Procs(p) be the set of lambda expressions in the set {ev-val(`(`p `'`S`)`) | S is an S-expression}. Show that the arities (number of formals) of the lambda expressions in Procs(p) is bounded.

(b) Conclude that there is no "strong" meta-circular interpreter, i.e., procedure satisfying the Strong Interpretation condition.

Problem 1.7 Extend kernel Scheme with lambda-expressions of varying numbers of arguments. These expressions are of the form

$$\texttt{(lambda } \langle\text{variable}\rangle \; \langle\text{body}\rangle\texttt{)}$$

(notice that the variable foloowing the `lambda` is not parenthesized), and their behavior is determined by the rewrite rule:

$$\texttt{((lambda } l \;\; B\texttt{)} \;\; V_1 \ldots V_n\texttt{)} \longrightarrow \texttt{((lambda (}l\texttt{)} \;\; B\texttt{) (list } V_1 \ldots V_n\texttt{))}$$

Show that now there *is* a "strong" meta-circular interpreter. *Hint*: Such an interpreter can be obtained by modifying the "analyzing" interpreter described in the Abelson/Sussman text.

A meta-circular interpreter, or more typically some compiled version of it, is known as a "universal machine": one program or machine which is capable of simulating any other. The existence of universal machines was an insight of a community of researchers in mathematical logic, notably the German Kurt Gödel and the Englishman Alan Turing, in the early 1930's. This idea stimulated the development of the first general purpose digital computers in the late 1930's and early 40's.

Corollary 1. Halts is half-decidable.

Proof: Any meta-circular interpreter is a half-decider for Halts. ■

Theorem 2. Decidable sets are closed under union, intersection, and set-theoretic difference[1].

Proof: If $D_{\mathcal{A}}$ is a decider for $\mathcal{A}$, and $D_{\mathcal{B}}$ is a decider for $\mathcal{B}$, then

$$\texttt{(lambda (v) (or (}D_{\mathcal{A}}\texttt{ v) (}D_{\mathcal{B}}\texttt{ v)))}$$

$$\texttt{(lambda (v) (and (}D_{\mathcal{A}}\texttt{ v) (}D_{\mathcal{B}}\texttt{ v)))}$$

$$\texttt{(lambda (v) (and (}D_{\mathcal{A}}\texttt{ v) (not (}D_{\mathcal{B}}\texttt{ v))))}$$

are deciders for $\mathcal{A} \cup \mathcal{B}$, $\mathcal{A} \cap \mathcal{B}$, and $\mathcal{A} - \mathcal{B}$, respectively. ■

[1] The set-theoretic difference, $A - B$ of sets A, B is $\{a \in A \mid a \notin B\}$.

It will now be useful to modify any of the meta-circular interpreters described above to take an additional, nonnegative integer argument indicating how many evaluation steps to perform. With arguments S and n, the modified interpreter tries only to simulate n evaluation steps of Scheme expression S. If S does not halt within n steps, the interpreter returns `#f`. If S halts within n steps and ev-val(S) is a printable value, the interpreter returns `(cons 'printable` ev-val(S)`)`. If S halts in at most n steps and ev-val(S) is not a printable value, the interpreter returns `#t`. The following definition abstracts the properties of this modified interpreter which we need.

Definition 2. A *stepping meta-circular interpreter for Scheme* is a closed Scheme expression, ⟨step-smeval⟩, such that for all Scheme expressions, M,

1. `(`⟨step-smeval⟩ `'`M $\widehat{n}$`)` halts for all $n \in \mathbb{N}$,
2. M halts iff there is an $n \in \mathbb{N}$ such that ev-val(`(`⟨step-smeval⟩ `'`M $\widehat{n}$`)`) $\neq$ `#f`, and
3. for all $n \in \mathbb{N}$, if ev-val(`(`⟨step-smeval⟩ `'`M $\widehat{n}$`)`) $\neq$ `#f`, then
   - if ev-val(M) is a ⟨printable value⟩, then

     ev-val(`(`⟨step-smeval⟩ `'`M $\widehat{n}$`)`) = `(cons 'printable` ev-val(M)`)`,
   - if ev-val(M) is not a ⟨printable value⟩, then

     ev-val(`(`⟨step-smeval⟩ `'`M $\widehat{n}$`)`) = `#t`.

Theorem 3. There is a stepping meta-circular interpreter for Scheme.

Problem 2. Pick one of the meta-circular interpreters described above and explain how to modify it to obtain a stepping version.

Theorem 4. The half-decidable sets are closed under union and intersection.

Proof: If $H_{\mathcal{A}}$ is a half-decider for $\mathcal{A}$, and $H_{\mathcal{B}}$ for $\mathcal{B}$, then

```
(lambda (v) (begin (H_A v) (H_B v)))
```

is a half-decider for $\mathcal{A} \cap \mathcal{B}$.

A half-decider for $\mathcal{A} \cup \mathcal{B}$ is

```
(lambda (v)
  (define (try n)
    (or (⟨step-smeval⟩ (list 'H_A v) n)
        (⟨step-smeval⟩ (list 'H_B v) n)
        (try (+ n 1))))
  (try 0)))
```

■

There is an additional assumption implicit in the above proof. Note that it is possible that application of ⟨step-smeval⟩ to some arguments could *interfere* with subsequent applications of ⟨step-smeval⟩ to those same, or other, arguments. That is, the first application could have a side-effect which changed the subsequent behavior of ⟨step-smeval⟩ so that it no longer acted as a stepping interpreter. So the expression above for the $\mathcal{A} \cup \mathcal{B}$ half-decider is correct only because *we have tacitly assumed that there is no unexpected interference.*

We will continue to make this "no interference" assumption in our subsequent constructions. We can justify this assumption by observing that all three meta-circular interpreters we described above do not interfere with themselves, as long as they are successively applied only to Scheme *expressions*[2]. That is, if M is a Scheme expression, then `(`⟨smeval⟩ `'`M`)` will always evaluate to the same printable value, if any, no matter what computations have preceded the application, as long as ⟨smeval⟩ was previously applied only to expressions. The stepping meta-circular interpreter we obtain by modifying one of the meta-circular interpreters described above will also be interference-free.

For any decider, D, for example, the expression

```
(lambda (qs) (⟨smeval⟩ (list 'D qs)))
```

is decider for the same set, but will interfere neither with itself nor any other evaluation.

Problem 3. Assume without loss of generality that half-deciders return `#t` when they halt. Explain why

```
(lambda (v) (or (H_A v) (H_B v)))
```

is a half-decider for $\mathcal{A} \cup \mathcal{B}$ iff $\mathcal{B} \subseteq \mathcal{A}$.

Theorem 5. $\mathcal{A}$ is decidable iff $\mathcal{A}$ and $\overline{\mathcal{A}}$ are half-decidable.

Proof: It is easy to turn a decider into a half-decider, so the proof of the left to right implication is straightforward.

For the other direction, let H_1 be a half-decider for $\mathcal{A}$, and H_2 be a half-decider for $\overline{\mathcal{A}}$. Then a decider for $\mathcal{A}$ is

```
(lambda (v)
  (define (try n)
    (cond ((⟨step-smeval⟩ (list 'H_1 v) n) #t)
          ((⟨step-smeval⟩ (list 'H_2 v) n) #f)
          (else (try (+ n 1)))))
  (try 0))
```

■

[2]But the Abelson/Sussman interpreters do not behave the same after being applied to *bodies*, because evaluation of a body may install definitions in a global environment which carries over to subsequent evaluations.

Problem 4. For two sets $\mathcal{A}, \mathcal{B}$ of S-expressions, let

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

Suppose $\mathcal{A}$ is undecidable, and let $\mathcal{S} = \mathcal{A} \uplus \overline{\mathcal{A}}$. Prove that neither $\mathcal{S}$ nor $\overline{\mathcal{S}}$ is half-decidable.

2 Integer Representations

In the context of Scheme computability, it was natural to work with sets of printable values, and Scott's Rice's Theorems identify an extensive class of noncomputable functions and undecidable sets of printable value representations of S-expressions.

But instead of translating an S-expression, S, into the Scheme expression, $\texttt{'}S$, we can choose to translate S-expressions into Scheme expressions in very different ways. This will allow us to discover undecidable sets of familiar objects like nonnegative integers, as well as sets of more exotic objects like Scheme procedures.

So let us consider translating an arbitrary S-expression, S, into a Scheme expression which we denote as $\texttt{"}S$. A minimum requirement on the translation is that the value of $\texttt{"}S$ should exist and uniquely determine S, i.e.,

$$S_1 \neq S_2 \text{ implies ev-val}(\texttt{"}S_1) \neq \text{ev-val}(\texttt{"}S_2)$$

for all S-expressions, S_1, S_2.

At this point we can take the definitions of computability, inseparability, decidability, and half-decidability from Handout 15, as well as the definitions of meta-circular interpreters above, with all occurrences of $\texttt{'}\langle\text{phrase}\rangle$ replaced by $\texttt{"}\langle\text{phrase}\rangle$. To avoid ambiguity, we will refer to these modified definitions as "-computability, "-inseparability, etc.

For example, we will say that the translation has the *self-apply property* if there is a Scheme expression, $\langle\text{rep-self-apply}\rangle$, which computes the translations of self-applications, namely,

$$\text{ev-val}\ (\texttt{(}\langle\text{rep-self-apply}\rangle\ \texttt{"}S\texttt{)}) = \text{ev-val}\ (\texttt{"(}S\ \texttt{"}S\texttt{)}) \qquad \text{(self-apply property)}$$

for all S-expressions, S.

Now, for *any* translation with the self-apply property, the proof of Scott's Rice's First Theorem holds with no change other than modifying the definition of $\langle\text{perverse}\rangle$ to be:

```
(lambda(rep-s)
  (if (⟨separator⟩ (⟨rep-self-apply⟩ rep-s))
      B0
      A0)),
```

where `rep-s` is a variable not free in A_0 or B_0.[3] So we conclude that Halts is "-undecidable for any translation with the self-apply property.

[3] There is no need to require that `rep-s` not occur free in $\langle\text{separator}\rangle$ or $\langle\text{rep-self-apply}\rangle$. These expressions are specified to work at "top-level" before any variables have been defined, so, without loss of generality, we may assume they are closed expressions.

Problem 5. Suppose we choose to translate S into the Scheme expression `(cons 1 '`S`)` if S halts, and into `(cons 0 '`S`)` otherwise. Show that Halts is "-decidable and "-separable from $\overline{\text{Halts}}$. Why doesn't Scott's Rice's Theorem apply in this case?

Problem 6. We consider a Scheme definition, $\langle$code-def$\rangle$, of a variable `code` so that

$$\text{ev-val}(\texttt{(code '}S\texttt{)}) = \texttt{(lambda () (code '(}S\texttt{ '}S\texttt{)))}.$$

Here is such a definition:

```
(define (code pv)
  (lambda () (code (list pv (list 'quote pv)))))
```

Now define "S to be

$$\texttt{((lambda (qs)}\,\langle\text{code-def}\rangle\;\texttt{(lambda () (code qs)) '}S\texttt{)}$$

6(a) Verify that translating S to "S has the required uniqueness property.

6(b) Verify that this translation has the self-apply property. *Hint*: Let $\langle$rep-self-apply$\rangle$ be `(lambda (rep-s) (rep-s))`.

6(c) Show that *no* non-trivial set of S-expressions is "-half-decidable.

6(d) Conclude that the property in Handout 15, Problem 4(a) fails for "-half-decidability.

Problem 6 shows that the conditions on translations are not strict enough to avoid pathological cases. One straightforward way to find satisfactory nonstandard translations is to ensure that there is a computable correspondence between the nonstandard translation and the standard one using `quote`.

Definition 3. A translation from S-expressions, S, to Scheme expressions, "S, will be called *good* iff there are Scheme expressions, $\langle$to-quote$\rangle$ and $\langle$from-quote$\rangle$, such that

$$\text{ev-val}(\texttt{(}\langle\text{to-quote}\rangle\;\texttt{"}S\texttt{)}) = \text{ev-val}(\texttt{'}S), \text{ and}$$

$$\text{ev-val}(\texttt{(}\langle\text{from-quote}\rangle\;\texttt{'}S\texttt{)}) = \text{ev-val}(\texttt{"}S).$$

It is easy to verify that good translations have the uniqueness, self-apply, and equality properties above, so all the results of Handout 15 will apply to them, as will essentially all the results we will establish below about the standard `quote` translation.

So now, if we want an undecidable set, say, of *nonnegative integers*, we need only find a *good* translation from quoted S-expression values into nonnegative integers.

However, there *isn't any* such good translation using our current Scheme kernel. The crux of the problem is that there is no procedure which can produce an infinite number of symbols starting from non-symbols.

Problem 7.

7(a) Prove that if $B_1 \underset{\text{eval}}{\longrightarrow} B_2$, then every symbol in body B_2 must occur in body B_1.

7(b) Conclude that for any kernel Scheme expression, F, $\{\text{ev-val}(\texttt{(}F\ \widehat{n}\texttt{)}) \mid n \in \mathbb{N}\}$ contains only finitely many symbols.

So we need to extend Scheme to allow definition of a *symbol enumerator*, namely, a total function from nonnegative integers *onto* the set of symbols.

Real Scheme includes string and string-to-symbol procedures which allow the definition of symbol enumerators. We will not bother introducing string procedures into our kernel, but henceforth include a new, primitive procedure identifier

$$\langle \text{primitive procedure id} \rangle ::= \ldots \mid \texttt{int->sym}$$

such that `int->sym` is specified to compute some symbol enumerator

$$\text{int-to-sym} : \mathbb{N} \to \langle \text{symbol} \rangle .$$

Once we have `int->sym`, we can also translate symbols back into integers:

Lemma 1. There is a computable, necessarily one-to-one, total function

$$\text{sym-to-int} : \langle \text{symbol} \rangle \to \mathbb{N}$$

such that

$$\text{int-to-sym}(\text{sym-to-int}(\sigma)) = \sigma$$

for all symbols σ.

Proof:

```
(define (sym->int symb)
  (define (search n)
    (if (eq? symb (int->symb n)) n (search (+ n 1))))
  (search 0))
```

■

Now it will be easy to define a good translation from S-expressions to nonnegative integers. We first choose an integer *pairing* function, pair, which defines a computable one-to-one map from pairs of nonnegative integers to nonnegative integers. For example, we can take pair(m, n) to be $2^m 3^n$.

Problem 8. Show that there are Scheme computable functions, first, and second, such that

$$\text{first}(\text{pair}(n, m)) = n \text{ and } \text{second}(\text{pair}(n, m)) = m.$$

Now we can define $\langle \text{from-quote} \rangle_{\mathbb{N}}$ to be

```
(lambda (pv)
  (define (pair n m) (* (expt 2 n) (expt 3 m)))
  (define (first n) ...)
  (define (second n) ...)
  (define (sexp->int qs)
    (cond
     ((boolean? qs)
      (if qs (pair 0 1) (pair 0 0)))
     ((integer? qs) (pair 1 qs))
     ((symbol? qs) (pair 2 (sym->int qs)))
     ((null? qs) (pair 3 0))
     (else
      (pair 4                           ;qs = '(S1...Sk) for k>=1
            (pair (sexp->int (car qs))
                  (sexp->int (cdr qs))))))))
  (sexp->int pv))
```

Note that $\langle \text{to-quote} \rangle_{\mathbb{N}}$ above actually defines a one-to-one total function

$$\text{sexp-to-int} : \langle \text{s-expression} \rangle \to \mathbb{N}.$$

Problem 9. Exhibit a corresponding definition, $\langle\text{to-quote}\rangle_{\mathbb{N}}$, of a total function

$$\text{int-to-sexp} : \mathbb{N} \rightarrow \langle\text{s-expression}\rangle$$

onto the set of S-expressions.

Now we get a set of nonnegative integers which is undecidable, because by Scott's Rice's First Theorem for the $\langle\text{from-quote}\rangle_{\mathbb{N}}$ translation, Halts is "-undecidable. In other words,

Definition 4. Let

$$\text{Halts}_{\mathbb{N}} ::= \{n \mid \widehat{n} = \text{ev-val } ((\langle\text{from-quote}\rangle_{\mathbb{N}} \texttt{ '}S)) \text{ for some } S \in \text{Halts}\}$$

Theorem 6. $\text{Halts}_{\mathbb{N}}$ is undecidable.

Problem 10. Let $f(n) = \text{pair}(4, \text{pair}(n, \text{pair}(1, n))))$, and let F be a Scheme expression computing f. Verify that we can use F as $\langle\text{rep-self-apply}\rangle$ for the $\langle\text{from-quote}\rangle_{\mathbb{N}}$ translation.

So far we have not specified any particular primitive rewrite rules for `int->sym`. This makes it possible to include arbitrary new "decidable" sets of integers in Scheme extended with `int->sym`. Namely, let Z be some set of integers, i.e., $Z \subseteq \mathbb{Z}$. Using rules

$$\texttt{(int->sym } \widehat{2n}\texttt{)} \longrightarrow \texttt{'true} \qquad if n \in Z,$$

$$\texttt{(int->sym } \widehat{2n}\texttt{)} \longrightarrow \texttt{'false} \qquad if n \notin Z,$$

for all $n \in \mathbb{N}$, will imply that Z is decided by the expression

```
(lambda (n) (if (eq? 'true (int->sym (* 2 n))) #t #f)) .
```

The remaining rewrite rules for application of `int->sym` to odd numbers can then be chosen to ensure that `int->sym` is a symbol enumerator.

Problem 11.

11(a) Define primitive rewrite rules for `int->sym` so that it is a symbol enumerator and so that the same sets of integers are decidable after extending Scheme with `int->sym` as before.

11(b) Describe some simple general conditions on the rewrite rules for `int->sym` which are sufficient to ensure the property of part 11(a). How about necessary and sufficient? *Hint*: Consider simulating a Scheme-with-`int->sym` decider for Z by a Scheme expression which contains no symbols or occurrences of `int->sym`.

Problem 11.5 Define

$$\text{church}(0) ::= \texttt{(lambda(x) x)},$$
$$\text{church}(n+1) ::= \texttt{(lambda (f) (lambda (x) (f ((}\text{church}(n)\texttt{ f) x)))).}$$

Show that translating S into church(sexp-to-int(S)) is a good translation of S-expressions into Scheme expressions.

3 Enumerability

Definition 5. A *range set* is the range of a Scheme computable partial function from $\mathbb{N}$ to S-expressions.

In other words, $\mathcal{S}$ is a range set iff there is a Scheme expression, F, such that

$$\{\text{ev-val}(\texttt{(}F\ \widehat{n}\texttt{)}) \mid n \in \mathbb{N} \text{ and } \texttt{(}F\ \widehat{n}\texttt{)} \text{ halts}\} = \{\text{ev-val}(\texttt{'}S) \mid S \in \mathcal{S}\}.$$

Theorem 7. A set of S-expressions is a range set iff it is half-decidable.

Proof: To show that range sets are half-decidable, suppose F computes a function, f, from $\mathbb{N}$ to S-expressions, then

```
(lambda (qs)
  (define (first n) ... )
  (define (second n) ... )
  (define (equal? s t) ... )
  (define (in-range? s n)
    (or (equal? s (⟨step-smeval⟩ (list 'F (first n)) (second n))
        (in-range? (+ n 1))))
  (in-range? qs 0))
```

is a half-decider for the range of f.

Conversely, suppose H is a half-decider for a set $\mathcal{S}$ of S-expressions. The empty set is the range of the function on $\mathbb{N}$ with empty domain, so we may assume there is some $S_0 \in \mathcal{S}$. Then $\mathcal{S}$ is the range of the function computed by:

```
(lambda (n)
  (let ((s-exp (int->sexp (first n))))
    (if (⟨step-smeval⟩ (list 'H s-exp) (second n))
        s-exp
        S0)))
```

■

Problem 12. Write a Scheme definition of `equal?` such that

$$\text{ev-val}(\texttt{(equal? '}S\texttt{ '}S\texttt{))} = \texttt{\#t},$$
$$\text{ev-val}(\texttt{(equal? '}S\texttt{ '}T\texttt{))} = \texttt{\#f} \text{ if } S \neq T,$$

for all S-expressions S, T.

Theorem 8. Every infinite half-decidable set of S-expressions is the range of a one-to-one, total computable function on the $\mathbb{N}$.

Problem 13. Prove Theorem 8.

Definition 6. A partial function $f : \mathbb{N} \to \mathbb{N}$ is *strictly increasing* if, for all $n < m \in \mathbb{N}$,

$$m \in \text{domain}(f) \quad \text{implies} \quad n \in \text{domain}(f) \text{ and } f(n) < f(m).$$

Theorem 9. A set of S-expressions is decidable iff it is the range of (int-to-sexp $\circ f$) for some strictly increasing $f : \mathbb{N} \to \mathbb{N}$.

Problem 14. Prove Theorem 9. *Hint*: Treat finite and infinite sets of S-expressions as separate cases.

Definition 7. For any partial function $g : \langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$ and set $\mathcal{S}$ of S-expressions, let

$$g(\mathcal{S}) ::= \{g(S) \mid S \in \mathcal{S} \cap \text{domain}(g)\}.$$

Theorem 10. If $\mathcal{S}$ is half-decidable and $g : \langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$ is computable, then $g(\mathcal{S})$ is also half-decidable.

Proof: If $\mathcal{S}$ is the range of a computable enumerator $h : \mathbb{N} \to \langle\text{s-expression}\rangle$, then $g(\mathcal{S})$ is the range of the computable enumerator $g \circ h$, and so is half-decidable by Theorem 7. ■

4 Incompleteness

Any proof system has some set of things called *proofs*. One key property of a proof is that "understanding" is not needed to verify that it is a proof—the proof can be checked mechanically. That is, we require that the set of proofs be *decidable*. In addition, there must be a mechanical way to extract from the proof the theorem it proves.

For example, we consider proof systems for *Scheme equations*: S-expressions of the form (M = N) where M, N are Scheme expressions. We can safely assume that proofs are represented as S-expressions. So any proof system is completely determined by a computable function $v : \langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$, called a *verifier*, which detects whether an S-expression is a proof, and if it is, extracts the theorem proved. Formally, we may as well define the verifer to *be* the proof system:

Definition 8. A *proof system for Scheme equations* is a computable function, $v : \langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$, such that for every S-expression, S,

1. $v(S) = \texttt{\#f}$, or
2. $v(S) = Q$ for some Scheme equation, Q.

An S-expression, S, such that $v(S) = Q$ for some equation, Q, is called a *v-proof of Q*. An equation which has a v-proof is called a *v-theorem* and is said to be *v-provable*; we use the notation $\vdash_v Q$ to indicate that Q is v-provable.

Problem 15. Sketch how to write a Scheme program computing the verifier for the arithmetic equation proof system of Handout 3.

Theorem 11. The set of theorems of any proof system is half-decidable.

Proof: The range of v is half-decidable by Theorem 10, hence so is $\text{range}(v) - \{\texttt{\#f}\}$; this set is the set of v-theorems, by definition of v-theorem. ■

Recall from Handout 13:

Definition 9. Two Scheme bodies are *observationally distinguishable* iff one evaluates to a printable value, and the other does not evaluate to exactly that printable value.

Bodies M, N are *observationally congruent*, in symbols, $M \cong N$, iff there is *no* Scheme context, C, such that $C[M]$ and $C[N]$ are observationally distinguishable.

An equation (M = N) is *valid* iff $M \cong N$. As usual, we use the notation $\models Q$ to mean that equation Q is valid.

Lemma 2. Let $\langle \text{spin} \rangle$ be a closed nonhalting expression. For any closed Scheme expression, M,

$$M \text{ does not halt iff } \models \texttt{(}M \texttt{ = } \langle \text{spin} \rangle\texttt{)}.$$

Problem 16. Prove Lemma 2. *Hint*: This will take some work.

Lemma 3. The set of *closed* Scheme expressions which do not halt is not a half-decidable set.

Problem 17. Prove Lemma 3. *Hint*: Either adapt the proof of Scott's Rice's Second Theorem (cf. Handout 15), or show that if the closed nonhalting expressions were half-decidable, then Halts would be too.

Corollary 2. The set of valid Scheme equations is not half-decidable.

Proof: Let $g : \langle \text{s-expression} \rangle \rightarrow \langle \text{s-expression} \rangle$ be defined by

$$g\ (S) = \begin{cases} M, & \text{if } S \text{ is } \texttt{(}M \texttt{ = } \langle \text{spin} \rangle\texttt{)} \text{ where } M \text{ is a closed Scheme expression;} \\ \langle \text{spin} \rangle\,, & \text{otherwise.} \end{cases}$$

Let $\mathcal{V}$ be the set of valid Scheme equations, and $\mathcal{N}$ be the set of closed, nonhalting Scheme expressions. So by definition, $g(\mathcal{V}) = \mathcal{N}$. Obviously g is computable. So, if $\mathcal{V}$ was half-decidable, then by Theorem 10, $\mathcal{N}$ would be half-decidable as well, contradicting Lemma 3. ■

Definition 10. A proof system, v, for Scheme equations is *sound* iff $\vdash_v Q$ implies $\models Q$ for all Scheme equations, Q.

Theorem 12. Incompleteness. For *any* sound proof system for Scheme equations, there is a valid equation which is not provable.

Proof: By definition of soundness, the provable equations are a subset of the valid equations. By Theorems 11 and Corollary 2, the set of the provable equations and the set of valid equations are not equal. So there must be a valid equation which is not provable. ■

We show in Section 6, Problem 20 how, given a Scheme expression computing the verifier for a sound proof system, to explicitly construct an unprovable valid equation.

5 Computability in Different Programming Languages

Are more functions computable in Scheme with side-effects than in functional Scheme?—in Scheme extended with "parallel" operators like `parallel-or` compared to regular "sequential" Scheme?–in Fortran or C compared to any of these versions of Scheme?—compared to register machines? Briefly, the answer is "No" for the familiar reason that there is a simulator or compiler for each of these language dialects in any of the others.

Problem 18. The `smeval` procedure in the course web-page is a "nearly" functional interpreter for Scheme with assignments. If it were really functional, it would be a demonstration that any function computable with assignments was already computable in the functional kernel.

The one place `smeval` uses a nonfunctional feature is with the procedure `getfresh`, which generates a "fresh" new symbol every time it is applied. Explain how to modify `smeval` to be a truly functional Scheme procedure once the kernel is extended with the `int->sym` procedure identifier. (This is a tricky programming problem.)

6 The Recursion Theorem

The undecidability of the halting problem, and more generally Scott's Rice's Theorems, revealed the impossibility of having an automatic procedure to detect any important property of expression evaluation. Another reflection of the general incomprehensibility of Scheme programs is Kleene's Recursion Theorem. This time, given any Scheme expression, we simply try to change it so it will produce a *different* result. This too is impossible.

Theorem 13. Kleene's Recursion Theorem. Let change : $\langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$ be a Scheme computable function. Then there is a Scheme expression, F, such that either

- ev-val(F) and ev-val(change(F)) are both undefined, or
- ev-val(F) = ev-val(change(F)).

Proof: Let h be the partial function on S-expressions such that $h(S) = \text{ev-val}(\text{change}(\texttt{(}S\ \texttt{'}S\texttt{)}))$. Assuming change is computable by some Scheme expression, C, we observe that h is computable by the expression:

$$H ::= \texttt{(lambda (qs) (}\langle\text{smeval}\rangle\ \texttt{(}C\ \texttt{(list qs (list 'quote qs)))))}$$

where `qs` is a fresh variable.

So for every S-expression, S, either

- ev-val(`(`H `'`S`)`) and ev-val(change(`(`S `'`S`)`)) are both undefined, or
- ev-val(`(`H `'`S`)`) = ev-val(change(`(`S `'`S`)`)),

by definition of H and ⟨smeval⟩.

Letting S be H above, we see that we can define F to be `(`H `'`H`)`. ■

Let fix be the function mapping the expression C to the expression F in the above proof. The construction of F from C shows that fix is a computable, total function from S-expressions to S-expressions, where by definition

$$\text{ev-val}(\text{fix}(C)) = \text{ev-val}(\text{change}(\text{fix}(C)))$$

whenever either expression has a value.

As a simple application of the Recursion Theorem, let change(S) = `(quote` S`)`. So the expression F given in the Recursion Theorem will satisfy ev-val(F) = ev-val(`'`F); in other words, F is yet another self-reproducing expression. We can similarly obtain a doubly self-reproducing expression (cf. Handout 15, problem 1(c)), by letting change(S) = `'(`S S`)`.

Problem 19.

19(a) Assume our kernel Scheme is extended with a rule-defined procedure identifier `smeval` which is a meta-circular evaluator. Write out the self-reproducing and doubly self-reproducing expressions described above.

19(b) Check that your expressions work by evaluating them in real Scheme. (Note that if you use MIT Scheme's builtin `eval` to act as the rule-defined `smeval`, you must supply it with a second argument which is an environment; so you should first evaluate the definition

```
(define smeval (lambda (qs) (eval qs user-initial-environment))).
```

Alternatively, you may use the `smeval` procedure from the course web-page to serve as the evaluator, modified so it *returns* a printable value instead of pretty-printing it.)

The Recursion Theorem yields elegant proofs of many results in computability theory. For example:

Definition 11. Let $\mathcal{A}, \mathcal{B}$ be distinct sets of S-expressions. An element in the symmetric difference of $\mathcal{A}$ and $\mathcal{B}$ is called a *witness* that $\mathcal{A} \neq \mathcal{B}$, i.e., a witness is an element, W, such that

$$W \in \mathcal{A} \quad \text{iff} \quad W \notin \mathcal{B}.$$

For any Scheme expression, H, let $\mathcal{W}_H$ be the set of S-expressions for which H is a half-decider. That is,

$$\mathcal{W}_H ::= \{S \in \langle\text{s-expression}\rangle \mid \texttt{(}H\ \texttt{'}S\texttt{)}\ \text{halts}\}.$$

A set $\mathcal{A}$ is *productive* if there is a total computable function, $w : \langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$, such that $w(H)$ is a witness that $\mathcal{A} \neq \mathcal{W}_H$ for all S-expressions, H. The function w is called a *productive function* for $\mathcal{A}$.

So productive sets are not half-decidable in a notably demonstrable way.

Theorem 14. NotHalts is productive.

Proof: Let change(S) ::= $\texttt{(}H\ \texttt{'}S\texttt{)}$. By the Recursion Theorem, there is an S_0 such that

$$S_0 \text{ halts} \quad \text{iff} \quad \texttt{(}H\ \texttt{'}S_0\texttt{)}\ \text{halts}.$$

Hence,

$$S_0 \in \text{NotHalts iff } \texttt{(}H\ \texttt{'}S_0\texttt{)} \in \text{NotHalts iff } S_0 \notin \mathcal{W}_H,$$

so S_0 is a witness that $\mathcal{W}_H \neq$ NotHalts. Therefore, the function mapping H to S_0 is a productive function for NotHalts; it is computed by the expression

```
(lambda (h) (⟨fix⟩ (lambda (s) (list h (list 'quote s)))))
```

where ⟨fix⟩ is a Scheme expression computing fix. ■

Problem 20. Use Theorem 14 to prove that there is a total computable function, unprov, such that if ⟨verifier⟩ is a Scheme expression computing the verifier of a sound proof system for Scheme equations, then unprov(⟨verifier⟩) is a valid equation which is not a theorem of the proof system.

Another elegant example of the use of the Recursion Theorem is due to Manuel Blum:

Definition 12. A set of S-expressions is *immune* if it is infinite but contains no infinite half-decidable subset.

So not only is there no "demon" which detects the elements in an immune set, but there isn't even a demon which merely detects an infinite number of sample elements from the set.

Definition 13. A Scheme expression, S, is called *minimal* iff it halts, and there is no Scheme expression, T, such that

$$\text{ev-val}(T) = \text{ev-val}(S) \text{ and sexp-to-int}(T) < \text{sexp-to-int}(S).$$

Theorem 15. The set of minimal Scheme expressions is immune.

Proof: By Theorem 8, we need only show that if $f : \mathbb{N} \to \langle\text{s-expression}\rangle$ is a Scheme computable, total function with infinite range, then there is some S-expression $S \in \text{range}(f)$ which is not minimal. If there is an expression in range(f) which is not a Scheme expression or does not halt, then that expression is not minimal. So we may assume that every expression in range(f) is a Scheme expression which halts.

Let change(S) $= f(k)$ where k is the least nonnegative integer for which sexp-to-int(S) $<$ sexp-to-int($f(k)$). Since f is computable, so is the function change. Since f has infinite range, change is a total function with the properties that

$$\text{sexp-to-int}(S) < \text{sexp-to-int}(\text{change}(S)) \text{ and change}(S) \in \text{range}(f),$$

for every S-expression, S.

Let F be the Scheme expression satisfying the Recursion Theorem for the function change. Since change(F) $\in$ range(f), it is a Scheme expression that halts. So by the Recursion Theorem, ev-val(F) $=$ ev-val(change(F)). But since sexp-to-int(F) $<$ sexp-to-int(change(F)) by definition of change, we conclude that change(F) $\in$ range(f) is not minimal. ■

Problem 21. For any total function $g : \mathbb{N} \to \mathbb{N}$ and S-expressions S, T, say that S is *at most g-larger* than T iff sexp-to-int(S) $\leq g$(sexp-to-int(T)). Call a Scheme expression *g-minimal* iff it halts and is at most g-larger than any other Scheme expression with the same value. (So the minimal expressions are the same as the g-minimal expressions for $g(n) = n$.)

Show that the set of g-minimal Scheme expressions is immune for any Scheme computable total function g such that $g(n) \geq n$ for all $n \in \mathbb{N}$.

Problem 22. Show that for any Scheme computable total function $g : \mathbb{N} \to \mathbb{N}$, there is an S-expression, S_g, such that sexp-to-int(ev-val(S_g)) $> g$(sexp-to-int(S_g)).

Problem 23.

23(a) Explain why the set of non-minimal Scheme expressions which halt is half-decidable.

23(b) Call a Scheme expression *except-one-minimal* iff it halts, and there is at most one smaller size Scheme expression with the same value. Show that the set of except-one-minimal Scheme expressions is immune.

Problem 24. Show that for any Scheme computable total function $g : \mathbb{N} \to \mathbb{N}$, there is a finite set $\mathcal{N}_g$ of nonnegative integers, and a half-decider, H, for $\mathcal{N}_g$, such that $\text{sexp-to-int}(M) > g(\text{sexp-to-int}(H))$ for any expression, M, computing the *membership* function of $\mathcal{N}_g$. *Hint*: Evaluation of `(`H n`)` should dovetail the evaluation of `(`S n`)` for all those Scheme expressions, S, numbered less than $g(\text{sexp-to-int}(H))$ which have not already been *hit at* k for some $k < n$. It halts as soon as one of the expressions `(`S n`)` halts with the value `#f`. This S, if any, is said to be *hit at* n.