Scheme Computability

What can and can't Scheme do? We can begin to pin this question down by asking what functions can be computed by Scheme procedures. In functional Scheme, at least, it is natural to think of procedures as computing functions, since a (closed) functional procedure always returns the same value every time it is applied to the same arguments. But if we try to pin down the domains and ranges of general Scheme procedures, we know that we come up against some challenging mathematical problems because of Scheme's self-applicative values. To avoid such difficulties, we begin by considering functions of one argument on the *integers*[1].

Definition 1. For any integer, n, let $\widehat{n}$ be its Scheme numeral. For example, $\widehat{23}$ is `23`. A partial function $f : \mathbb{Z} \to \mathbb{Z}$ is *Scheme computable* iff there is a Scheme expression F such that

- ev-val(`(`$F\ \widehat{n}$`)`) is defined iff $n \in \text{domain}(f)$, and
- ev-val(`(`$F\ \widehat{n}$`)`) $= \widehat{f(n)}$ for all $n \in \text{domain}(f)$.

Such an expression F is said to *compute* the function f.

For example, the function $f(n) = n^2$ is Scheme computable because the expression

```
(lambda (n) (* n n))
```

computes it. Likewise, the function $t(n)$ which equals a "stack" of exponentiated two's of height n, e.g., $t(1) = 2, t(2) = 2^2, t(3) = 2^{2^2}, \ldots$ is computed by the expression

```
(lambda (n)
  (define (compose f g)
    (lambda (n) (f (g n))))
  (define (repeatedly-compose f)
    (lambda (k)
      (if (<= k 1) f (compose f ( (repeatedly-compose f) (- k 1) )))))
  (let ((double (lambda (m) (* 2 m))))
    (let ((2-to-the (lambda (n)
                     ( ( (repeatedly-compose double) n ) 1 ))))
      ( ( (repeatedly-compose 2-to-the) n ) 1 ))))
```

[1] We don't even want to use Scheme numbers which are not integers, e.g., in MIT Scheme, `(= 1 1.0000000000000001)` returns `#t`.

In fact, the definition of `compose` above shows that Scheme computable integer-valued functions one argument are *closed under composition.* That is, if f, g are Scheme computable functions from integers to integers, then so is their composition $f \circ g$. In fact, we can see that the Scheme computable functions have an endless number of other closure properties—basically any way of combining functions which is programmable in Scheme.

It should be clear that the Scheme computable integer-valued functions are a huge class of functions, containing all the familiar number-theoretic functions, and indeed all the integer-valued functions anyone outside the theory computability is likely to think of. Nevertheless, we will soon see some clearly defined functions which are *not* Scheme computable.

Computability theory is traditionally developed using functions on the integers as above. In our setting with Scheme, it is convenient, natural, and unproblematical to consider computability over the set of *printable values.* These are Scheme expressions specified by the grammar:

⟨printable value⟩ ::= ⟨primitive value⟩ | `(cons` ⟨printable value⟩ ⟨printable value⟩`)`
⟨primitive value⟩ ::= ⟨integer⟩ | ⟨boolean⟩ | ⟨symbol⟩ | `()`
⟨symbol⟩ ::= `(quote` ⟨variable⟩`)` | `(quote` ⟨procedure identifier⟩`)`
| `(quote` ⟨keyword⟩ `)`

Problem 1. Let V_0 be some Scheme value. A V_0-*detector* is a Scheme expression, D, such that ev-val(`(`D E`)`) $=$ `#t`, for any Scheme expression, E, such that ev-val(E) $= V_0$, and ev-val(`(`D E`)`) $=$ `#f` for any E such that ev-val(E) is defined and is not syntactically *identical* to V_0.

1(a) Show that every printable value has a detector.

1(b) Show that no Scheme value which is not a printable value has a detector. *Hint*: You may use the following facts about observational congruence, $\cong$, for Scheme with side-effects:

$$\texttt{(lambda (l) (apply } P \texttt{ l))} \cong P$$

for all procedure identifiers, P.

$$\texttt{((lambda (}x\texttt{) } E[x]\texttt{) } M\texttt{)} \cong E'[M]$$

for all evaluation contexts E, E', variables x, and expressions M such that the occurrence of x at the hole in E is free and E' is a fresh renaming of E.

Printable values are traditionally printed out by Scheme implementations without the `cons`'s, `quote`'s on symbols, or nil's at the end of lists. For example, the printable value

```
(cons #f (cons (cons 1 (cons 2 ())) (cons (quote a) ())))     (*)
```

would print out as

$$\texttt{(\#f (1 2) a).}$$

Such parenthesized output expressions are called *S-expressions*; they are defined by a simple grammar:

$$\langle\text{s-expression}\rangle ::= \langle\text{identifier}\rangle \mid \texttt{(}\langle\text{s-expression}\rangle^{*}\texttt{)}$$

Scheme provides a standard way, namely `quote`'ing, to convert an S-expression into the printable-value which would produce it as output. For example, `(quote (#f (1 2) a))` is a legitimate Scheme expression, whose value will be the printable value (*) above.

In our substitution model for Scheme, we treat the `quote` of an S-expression as sugar. First, we extend Scheme grammar:

$$\langle\text{derived expression}\rangle ::= \ldots \mid \langle\text{quoted}\rangle$$
$$\langle\text{quoted}\rangle ::= \texttt{(quote}\ \langle\text{s-expression}\rangle\texttt{)}$$

Next, we describe how to desugar quoted S-expressions into printable values, namely:

- If V is an $\langle\text{integer}\rangle$, $\langle\text{boolean}\rangle$, or `()`, then (quote V) desugars to V.
- If `(quote` V`)` is a symbol, then `(quote` V`)` desugars to itself.
- `(quote (`$\langle\text{identifier}_1\rangle \ldots \langle\text{identifier}_n\rangle$`))` desugars recursively to

$$\texttt{(cons (quote}\ \langle\text{identifier}_1\rangle\texttt{) (quote (}\langle\text{identifier}_2\rangle\ \ldots\ \langle\text{identifier}_n\rangle\texttt{))).}$$

The Scheme expression `(quote` S`)` is traditionally abbreviated as `'`S.

These definitions ensure that `'`S desugars into a printable value which traditional implementations would print out as "S." Now it is easy to define computable functions on S-expressions:

Definition 2. A partial function $f : \langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$ is *Scheme computable* iff there is a Scheme expression F such that

- ev-val(`(`F `'`S`)`) is defined iff $S \in \text{domain}(f)$, and
- ev-val(`(`F `'`S`)`) = ev-val(`'`$f(S)$) for all $S \in \text{domain}(f)$.

Such an expression F is said to *compute* the function f.

For practice with quoting and preparation for the non-computability arguments in the next section, we consider how to make "self-reproducing" Scheme expressions. A Scheme expression, R, is self-reproducing iff ev-val(R) = ev-val(`'`R). So in real Scheme, evaluation of R would yield output "R"!

Here's a way to make one: let `make-self-apply` be an expression such that

$$\text{ev-val}(\texttt{(make-self-apply '}S\texttt{)}) = \text{ev-val}(\texttt{'(}S\ \texttt{'}S\texttt{)}).$$

For example, we could use the Scheme definition:

```
(define (make-self-apply qs) (list qs (list 'quote qs)))
```

Then if we evaluated `(make-self-apply 'make-self-apply)` in Scheme, we would get the output `(make-self-apply 'make-self-apply)`!

This is very nearly self-reproducing, but it hinges on the prior definition of `make-self-apply`. To get an entirely self-contained self-reproducing expression, we incorporate the definition into the expression. Let, ⟨perverse⟩ be the expression

```
( (lambda ()
    (define (make-self-apply qs)
      (list qs (list 'quote qs)))
    make-self-apply) )
```

Now we have that, even with no prior definitions,

$$\text{ev-val}(\texttt{(}\langle\text{perverse}\rangle\ \texttt{'}S\texttt{)}) = \text{ev-val}(\texttt{'(}S\ \texttt{'}S\texttt{)}), \qquad (**)$$

for all S-expressions S. Let R be `(`⟨perverse⟩ `'`⟨perverse⟩`)` to obtain a really self-reproducing expression.

Problem 2.

2(a) Check that this last R is self-reproducing by evaluating it in real Scheme. (Use `pp` or `pretty-print` so the output is readable.)

2(b) Exhibit two other self-reproducing expressions and check them in real Scheme. Turn in your pretty-printed output. *Hint*: : It doesn't matter what ⟨perverse⟩ looks like as long as it satisfies the specification (**).

2(c) An expression D is *doubly self-reproducing* iff ev-val(R) = ev-val(`'(`R R`)`). Exhibit a doubly self-reproducing expression and check it in real Scheme. Turn in your output.

2(d) *Optional* Adapt these ideas to construct a self-reproducing program in your favorite non-Lisp programming language. Check your program by running it, and turn in your output.

Scott's Rice's Theorem

Definition 3. A set $\mathcal{S}$ of S-expressions is *evaluation-invariant* if $S_1 \underset{\text{eval}}{\longrightarrow} S_2$ implies that $S_1 \in \mathcal{S}$ iff $S_2 \in \mathcal{S}$ for all S-expressions S_1, S_2.

$\mathcal{S}$ is *non-trivial* iff $\mathcal{S}$ is nonempty and does not contain all S-expressions.

Two sets $\mathcal{A}$, $\mathcal{B}$ of S-expressions are *computably inseparable* iff there is no Scheme expression, $\langle$separator$\rangle$, with the following three properties:

1. If $A \in \mathcal{A}$, then ev-val(($\langle$separator$\rangle$ 'A)) = `#t`.
2. If $B \in \mathcal{B}$, then ev-val(($\langle$separator$\rangle$ 'B)) = `#f`.
3. For all S-expressions, S, ev-val(($\langle$separator$\rangle$ 'S)) is defined and equals either `#t` or `#f`.

Theorem. Scott's Rice's First Theorem. Let $\mathcal{A}$, $\mathcal{B}$ be nontrivial, evaluation-invariant sets of S-expressions. Then $\mathcal{A}$ and $\mathcal{B}$ are computably inseparable.

Proof:

By contradiction. Suppose there was a $\langle$separator$\rangle$ expression as above. Since $\mathcal{A}$ and $\mathcal{B}$ are nontrivial, there are S-expressions $A_0 \in \mathcal{A}$ and $B_0 \in \mathcal{B}$.

Let `pv` be a variable not free in $\langle$separator$\rangle$, A_0 or B_0, and define $\langle$perverse$\rangle$ to be the Scheme expression:

```
(lambda(pv)
  (if (⟨separator⟩ (list pv (list 'quote pv)))
      B0
      A0)).
```

Suppose ev-val(($\langle$separator$\rangle$ '(S 'S))) = `#t` for some S-expression S. Then by definition of $\langle$perverse$\rangle$ above, ($\langle$perverse$\rangle$ 'S) $\underset{\text{eval}}{\longrightarrow}^* B_0$. So by evaluation-invariance, ($\langle$perverse$\rangle$ 'S) $\in \mathcal{B}$. Finally, by $\langle$separator$\rangle$-condition 2, ev-val(($\langle$separator$\rangle$ '($\langle$perverse$\rangle$ 'S))) = `#f`.

So we have shown that, if

ev-val(($\langle$separator$\rangle$ '(S 'S))) = `#t`,

then

ev-val(($\langle$separator$\rangle$ '($\langle$perverse$\rangle$ 'S))) = `#f`.

Similarly, if

$$\text{ev-val}(\texttt{(}\langle\text{separator}\rangle\ \texttt{'(}S\ \texttt{'}S\texttt{))}) = \texttt{\#f},$$

then

$$\text{ev-val}(\texttt{(}\langle\text{separator}\rangle\ \texttt{'(}\langle\text{perverse}\rangle\ \texttt{'}S\texttt{))}) = \texttt{\#t}.$$

That is, for every S-expression, S,

$$\text{ev-val}(\texttt{(}\langle\text{separator}\rangle\ \texttt{'(}S\ \texttt{'}S\texttt{))}) = \texttt{\#t} \ \text{ iff } \ \text{ev-val}(\texttt{(}\langle\text{separator}\rangle\ \texttt{'(}\langle\text{perverse}\rangle\ \texttt{'}S\texttt{))}) = \texttt{\#f}.$$

Now let S be $\langle$perverse$\rangle$ and C be $\texttt{(}\langle\text{perverse}\rangle\ \texttt{'}\langle\text{perverse}\rangle\texttt{)}$. Then the equivalence above instantiates to

$$\text{ev-val}(\texttt{(}\langle\text{separator}\rangle\ \texttt{'}C\texttt{)}) = \texttt{\#t} \ \text{ iff } \ \text{ev-val}(\texttt{(}\langle\text{separator}\rangle\ \texttt{'}C\texttt{)}) = \texttt{\#f}.$$

which contradicts $\langle$separator$\rangle$-condition 3. ■

Definition 4. For any set $\mathcal{S}$ of S-expressions, define the *complement* of $\mathcal{S}$, in symbols, $\overline{\mathcal{S}}$, to be the set-theoretic complement of $\mathcal{S}$ relative to the set of S-expressions. In other words, $\overline{\mathcal{S}}$ is the set of S-expressions *not* in $\mathcal{S}$. The set $\mathcal{S}$ is Scheme *decidable* iff its membership function is Scheme computable, i.e., there is a Scheme expression, D, such that ev-val$(\texttt{(}D\ \texttt{'}S\texttt{)}) = \texttt{\#t}$ for $S \in \mathcal{S}$, and ev-val$(\texttt{(}D\ \texttt{'}S\texttt{)}) = \texttt{\#f}$ for $S \in \overline{\mathcal{S}}$.

Corollary 1. The set

$$\text{Halts} ::= \{\, S \in \langle\text{s-expression}\rangle \mid S \text{ is a Scheme expression and ev-val}(S) \text{ is defined}\,\}.$$

is not Scheme decidable.

Proof: If it were, the expression computing its membership function would be a separator for the set $\mathcal{A} = \text{Halts}$ and $\mathcal{B} = \overline{\text{Halts}}$. ■

Definition 5. A set $\mathcal{S}$ of S-expressions is *half-decidable* iff there is an Scheme expression H such that for all S-expressions S,

$$\texttt{(}H\ \texttt{'}S\texttt{)} \text{ halts } \ \text{iff} \ \ S \in \mathcal{S}.$$

Such an H is called a *half-decider* or *demon* for $\mathcal{S}$. Half-decidable sets are usually called *recursively enumerable* (r.e.) sets.

Theorem. Scott's Rice's Second Theorem. Let $\mathcal{S}$ be a non-trivial, evaluation-invariant set of S-expressions containing all the non-halting Scheme expressions. Then $\mathcal{S}$ is not half-decidable.

The proof of the Second Theorem is similar to that of the First.

Corollary 2. The set

$$\text{NotHalts} ::= \{\, S \in \langle\text{s-expression}\rangle \mid S \text{ is a Scheme expression and ev-val}(S) \text{ is not defined}\,\}$$

is not half-decidable.

Problem 3. Prove Scott's Rice's Second Theorem.

These two theorems reveal that there is *no* property of Scheme expressions depending only on how they evaluate, that can be detected in an *automatic*, completely reliable, way. Of course, just as there is no single unprovable theorem, there is no single program which cannot be demonstrated to have some particular property. But any *procedure* that tries to detect a non-trivial, evaluation-invariant property must give a wrong answer, or fail to give a required answer, for *some* input expression.

Problem 4.

4(a) Show that if $\mathcal{S}_1$ is half-decidable, and

$$\mathcal{S}_2 = (\mathcal{S}_1 - \mathcal{F}_1) \cup \mathcal{F}_2$$

for some finite sets $\mathcal{F}_1, \mathcal{F}_2$, then $\mathcal{S}_2$ is half-decidable.

4(b) Conclude that any Scheme procedure which tries to decide whether expressions have a non-trivial, evaluation-invariant property must, give wrong answers, or fail to give an answer when it should, for an *infinite number* of input expressions.