Scheme Computability

1 Computable Functions on S-Expressions

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. Computability theory is traditionally developed using functions on the integers, but in our setting with Scheme, it is natural to consider computability over the set of *S-expressions*:

$$\begin{aligned}\langle\text{s-expression}\rangle &::= \langle\text{self-evaluating}\rangle \mid \langle\text{identifier}\rangle \mid \texttt{(}\langle\text{s-expression}\rangle^*\texttt{)}\\ \langle\text{self-evaluating}\rangle &::= \langle\text{integer}\rangle \mid \langle\text{boolean}\rangle \mid \langle\text{string}\rangle\\ \langle\text{identifier}\rangle &::= \text{identifiers which are not } \langle\text{self-evaluating}\rangle\end{aligned}$$

S-expressions are the standard print representations of those list structures that have such representations[1]. For any S-expression, S, Scheme provides a simple way to form an expression whose value would have S as its standard printed representation. Namely, the value of the Scheme expression `(quote` S`)` prints out as S. We shall abbreviate `(quote` S`)` as `'`S. Most Scheme readers and printers also use this abbreviation.

The *printable values* are exactly what we get from desugaring a quoted S-expression. These are the canonical Scheme expressions whose values would print out as S-expressions:

$$\begin{aligned}\langle\text{printable value}\rangle &::= \langle\text{integer}\rangle \mid \langle\text{boolean}\rangle \mid \langle\text{string}\rangle \mid \langle\text{symbol}\rangle \mid \langle\text{printable list-value}\rangle\\ \langle\text{printable list-value}\rangle &::= \langle\text{nil}\rangle \mid \texttt{(list }\langle\text{printable value}\rangle^+\texttt{)}\\ \langle\text{symbol}\rangle &::= \texttt{'}\langle\text{identifier}\rangle\\ \langle\text{nil}\rangle &::= \texttt{'()}\end{aligned}$$

So `'`S desugars into a printable value which would in turn print out in standard Scheme as exactly S again. For example, `'(a b)` has the standard printable representation `(a b)`. Note that our Scheme Substitution Model uses standard *input* representations of values rather than

[1] We restrict the numbers (more precisely, numerals) in printable values to be integers rather than general Scheme numbers to avoid the messy details of floating point arithmetic. For example, in MIT Scheme, `(= 1 1.0000000000000001)` returns `#t`.
For simplicity we also omit "dotted pair" notation for pairs which are not lists.

output representations. This means the expression `'(a b)` evaluates in the Substitution Model to the printable value `(list 'a 'b)`, i.e.,

$$\texttt{'(a b)} \downarrow \texttt{(list 'a 'b)}$$

We let sexp(E) be the standard S-expression representation of the printable value of E, if any.

Definition 1. If E is a Scheme expression, p is a printable value, and S is an S-expression such that

$$E \downarrow p \text{ and } \texttt{(quote } S\texttt{)} \downarrow p$$

then we say that sexp(E) $= S$. If E does not converge to a printable value, then sexp(E) is undefined.

So, for example,

$$\text{sexp}(\texttt{'(a b)}) = \texttt{(a b)} = \text{sexp}(\texttt{(list 'a 'b)}) = \text{sexp}(\texttt{(cons 'a '(b))})$$

It follows from the definition of the Substitution Model and the rules for desugaring quoted S-expressions that sexp(E) is a unique S-expression or is undefined.

Now it is easy to define computable functions on S-expressions:

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

- $\texttt{(}F\ \texttt{'}S_1\ \ldots\ \texttt{'}S_n\texttt{)} \downarrow$ iff $(S_1 \ldots, S_n) \in \text{domain}(f)$, and
- $\text{sexp}(\texttt{(}F\ \texttt{'}S_1\ \ldots\ \texttt{'}S_n\texttt{)}) = \text{sexp}(\texttt{(quote}\ f(S_1, \ldots, S_n)\texttt{)})$ for all $(S_1 \ldots, S_n) \in \text{domain}(f)$.

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

Another way to say that F computes f is to say that

$$\texttt{(}F\ \texttt{'}S_1 \ldots \texttt{'}S_n\texttt{)} = \texttt{'}f(S_1, \ldots, S_n)$$

is a valid Scheme equation for all S-expressions $S_1, \ldots, S_n$.

Problem 1. Give an example of a Scheme expression F such that F computes the identity function on S-expressions, but $\texttt{(}F\ \texttt{x)} = \texttt{x}$ is not a valid Scheme equation.

For practice with quoting, and in 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 evaluation of R returns a printable value syntactically identical to R. That is, $\text{sexp}(R) = R$!

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

$$\text{sexp}(\texttt{(}\text{make-selfapply}\ \texttt{'}S\texttt{)}) = \texttt{(}S\ \texttt{'}S\texttt{)} \qquad (*)$$

For example, we could define

make-selfapply ::= `(lambda (s) (list s (list 'quote s)))`

Now substituting make-selfapply for S in (*) above yields

$$\text{sexp}(\texttt{(}\text{make-selfapply}\ \texttt{'}\text{make-selfapply}\texttt{)}) = \texttt{(}\text{make-selfapply}\ \texttt{'}\text{make-selfapply}\texttt{)}$$

In other words, we can choose R to be (make-selfapply 'make-selfapply).

Problem 2.

2(a) Check that this last R is self-reproducing by evaluating it in real Scheme.

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 make-selfapply looks like as long as it satisfies the specification (*).

2(c) An expression D is *doubly self-reproducing* iff $\text{sexp}(R) = \texttt{(}R\ \ R\texttt{)}$. 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.

2 Scott's Rice's Theorem

Definition 3. A set $\mathcal{S}$ is Scheme *decidable* iff its membership function is Scheme computable, i.e., there is a Scheme expression, D, called a *decider* for $\mathcal{S}$, such that

$$\text{sexp}(\texttt{(}D\ \ \texttt{'}S\texttt{)}) = \texttt{\#t} \text{ for } S \in \mathcal{S}, \text{ and}$$
$$\text{sexp}(\texttt{(}D\ \ \texttt{'}S\texttt{)}) = \texttt{\#f} \text{ for } S \notin \mathcal{S}.$$

Let

$$\text{Halts} ::= \{S \in \langle\text{s-expression}\rangle \mid S \downarrow \}.$$

We aim to prove

Corollary 1. Halts is not Scheme decidable.

This famous result will follow from a slightly more general result of Dana Scott.

Definition 4. Two sets $\mathcal{A}$, $\mathcal{B}$ of S-expressions are *Scheme separable* iff there is a Scheme expression, T, called a *separator* for $\mathcal{A}$, $\mathcal{B}$, with the following three properties:

1. If $S \in \mathcal{A}$, then sexp$(\texttt{(}T\ \texttt{'}S\texttt{)}) = \texttt{\#t}$,
2. If $S \in \mathcal{B}$, then sexp$(\texttt{(}T\ \texttt{'}S\texttt{)}) = \texttt{\#f}$, and
3. for all S-expressions, S, sexp$(\texttt{(}T\ \texttt{'}S\texttt{)})$ is defined and equals either `#t` or `#f`.

$\mathcal{A}$ and $\mathcal{B}$ are *Scheme inseparable* iff they are not Scheme separable.

For any set $\mathcal{S}$ of S-expressions, let $\overline{\mathcal{S}}$ be the complement of $\mathcal{S}$, i.e., the set of S-expressions *not* in $\mathcal{S}$. A decider for a set of S-expressions is, by definition, precisely a separator for $\mathcal{S}$ and $\overline{\mathcal{S}}$. So a set is Scheme decidable iff it and its complement are Scheme separable.

Definition 5. A set $\mathcal{S}$ of S-expressions is *submodel-invariant* providing $S_1 \underset{\text{submodel}}{\longrightarrow} S_2$ implies that $S_1 \in \mathcal{S}$ iff $S_2 \in \mathcal{S}$, for all S-expressions S_1, S_2.

Theorem 1. (**Scott's Rice's First Theorem.**) Let $\mathcal{A}$, $\mathcal{B}$ be nonempty, submodel-invariant sets of S-expressions. Then $\mathcal{A}$ and $\mathcal{B}$ are computably inseparable.

Notice that Halts is obviously a submodel-invariant, nonempty set of S-expressions, and so is its complement. Therefore Halts and its complement are inseparable, and hence Halts is not Scheme decidable. This proves Corollary 1 above.

Now we proceed to the proof of Theorem 1:

Proof:

By contradiction. Suppose there was a separator, T, for $\mathcal{A}$ and $\mathcal{B}$. Assume for simplicity that T does not contain `call/cc` or `abort`.

Since $\mathcal{A}$ and $\mathcal{B}$ are nonempty, there are S-expressions $A \in \mathcal{A}$ and $B \in \mathcal{B}$. Let `s` be a variable not free in T, A or B, and define the "perverse" expression, P, to be:

$$P ::= \texttt{(lambda(s) (if (}T\texttt{ (list s (list `quote s))) }B\ A\texttt{))}.$$

Now supppose S is an S-expression such that $\text{sexp}(\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))}) = \texttt{\#t}$. This implies that `(`$T$ `(list s (list ‘quote s)))` $\downarrow$ `#t`, so by definition of P above,

$$\texttt{(}P\ \texttt{'}S\texttt{)} \xrightarrow[\text{submodel}]{*} B.$$

Hence by submodel-invariance,

$$\texttt{(}P\ \texttt{'}S\texttt{)} \in \mathcal{B}.$$

Now by separator-condition 2,

$$\text{sexp}(\texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))}) = \texttt{\#f}.$$

So we have shown that if

$$\text{sexp}(\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))}) = \texttt{\#t},$$

then

$$\text{sexp}(\texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))}) = \texttt{\#f}.$$

Similarly, if

$$\text{sexp}(\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))}) = \texttt{\#f},$$

then

$$\text{sexp}(\texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))}) = \texttt{\#t}.$$

That is, for every S-expression, S,

$$\text{sexp}(\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))}) = \texttt{\#t} \text{ iff } \text{sexp}(\texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))}) = \texttt{\#f}. \qquad (**)$$

Now let S be P and C be `(`P `'`P`)`. Then the equivalence (**) above yields

$$\text{sexp}(\texttt{(}T\ \texttt{'}C\texttt{)}) = \texttt{\#t} \text{ iff } \text{sexp}(\texttt{(}T\ \texttt{'}C\texttt{)}) = \texttt{\#f}.$$

which is only possible if $\text{sexp}(\texttt{(}T\ \texttt{'}C\texttt{)})$ is not defined, contradicting separator-condition 3. ■

Problem: (a) Suppose T did contain `abort`. What step of the above argument would no longer be justified? *Hint*: Suppose T was simply of the form `(abort` T'`)`.

(b) Justify the assumption that the separator does not contain `call/cc` or `abort`. *Hint*: Argue that if there were a separator which did contain occurrences of `call/cc` or `abort`, then there would also be one which did not.

Definition 6. A set $\mathcal{S}$ of S-expressions is *half-decidable* iff there is a Scheme expression H such that for all S-expressions S,

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

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

Theorem. (**Scott's Rice's Second Theorem.**) A set of S-expressions is said to be *nontrivial* iff neither it nor its complement is empty. Let $\mathcal{S}$ be a nontrivial, submodel-invariant set of S-expressions containing $\overline{\text{Halts}}$. Then $\mathcal{S}$ is not half-decidable.

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

Corollary 2. $\overline{\text{Halts}}$ is not half-decidable.

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

These two theorems reveal that *no* nontrivial fact about the value of a Scheme expression can reliably be predicted by any procedure which could be expressed in Scheme. Since we know that Scheme can simulate any other programming language, we can simply say that no computational procedure of any kind can reliably determine any nontrivial property of the values of Scheme expressions.

We observed in the Handout on arithmetic equations that there is no single true theorem which is intrinsically unprovable, because if it's true, one can simply include it as axiom in some sound proof system. Similarly, there is no single Scheme expression and property of its value which cannot be detected by some procedure – one can simply include a test for exactly this expression in a detection procedure. But if a procedure tries to detect a nontrivial, submodel-invariant property of every S-expression to which it may be applied, then it must give a wrong answer, or fail to give any answer, for *some* S-expression argument.

Problem 4.

4(a) Two sets are said to be *almost equal* iff there are only finitely many elements in one but not the other. Show that if $\mathcal{S}_1$ is half-decidable and $\mathcal{S}_2$ is a set of S-expressions which is almost equal to $\mathcal{S}_1$, then $\mathcal{S}_2$ is also half-decidable.

4(b) Let $\mathcal{A} = \text{Halts}$ and $\mathcal{B} = \overline{\text{Halts}}$. Conclude that any Scheme procedure T must violate the separator conditions of Scott's Rice's Theorem for *infinitely many* S-expressions, S.

Note that Halts itself *is* half-decidable: if the variable `eval` is bound to a meta-circular evaluator, then `(eval '`S`)` halts iff $S \in \text{Halts}$, so `eval` would be a half-decider for Halts.

3 The Hard Part of Scheme

The proof that Halts is undecidable does not provide much information about which part of Scheme is the "hard part." For example, we have noted that Hilbert's 10th Problem is undecidable. Since Scheme has integers and can define procedures for evaluating any polynomial, this alone would lead to the halting problem being undecidable. But in fact the presence of integers in Scheme is not an essential cause of undecidability. The Halting Problem is still undecidable for a small fragment of Kernel Scheme, which not only has neither integers, symbols, nor lists, it doesn't even have conditionals!

To show this, we explain how to "simulate" two-counter machines with a small fragment of Scheme. This implies that the Halting problem for the small Scheme fragment will be as hard as the Halting Problem for two-counter machines. But we have observed earlier that all of Scheme can be "compiled" into two-counter machines, so the Halting problem for two-counter machines is as hard as the Halting problem for all of Scheme. Since the full Scheme halting problem is undecidable, so are the Halting problems for two-counter machines and hence for the small fragment of Scheme.

For any two-counter machine M, a triple, (n_0, n_1, n_2), of natural numbers describes a "configuration" or "complete state" at any step in a computation of M: n_0 designates the value of the program counter, and n_1, n_2 the contents of the two counters. Let $next_M(n_0, n_1, n_2)$ be the triple describing the next complete state of M after one instruction execution. For example, if the seventh instruction of M was `dec2` — decrement counter two — then $next_M(7, 5, 111) = (8, 5, 110)$, indicating that the program counter next points to the eighth instruction, counter one is unchanged, and counter two has decreased by one. For definiteness, we specify that $next_M(n_0, n_1, n_2) = (0, n_1, n_2)$ if n_0 is zero or greater than the number of instructions in M.

Suppose $next_M$ is Scheme computable. Then M halts starting in configuration $(1, 0, 0)$ iff the following Scheme expression halts:

```
(letrec ((try (lambda (tripl)
                (if (zero? (first trpl))
                    0     ;it doesn't matter what value is returned here
                    (try (nextm (first trpl)
                                (second trpl)
                                (third trpl)))))))
  (try (triple 1 0 0)))
```

where `nextm` is bound to a procedure for computing $next_M$. (We assume wlog that `nextm` is an expression without `set!`, `call/cc` or `abort`. This ensures that `nextm` computes $next_M$ every time it is applied, and that an application of `nextm` returns a value without aborting.)

Actually, to allow the conditional above to be simulated below by a procedure `if-proc`, we modify the expression above so `try` returns a *delayed* result:

```
(letrec ((try (lambda (tripl)
                (if (zero? (first trpl))
                    (lambda () 0)
                    (lambda ()
                      ((try
                       (nextm (first trpl)
                              (second trpl)
                              (third trpl))))))))))
  ((try (triple 1 0 0))))
```

Now to "simulate" two-counter machines using only a small part of Scheme, we represent each natural number, n, by a lambda expression $\widehat{n}$. In the above expression, we would replace 0 and 1 by their procedure representations $\widehat{0}, \widehat{1}$, and also replace `zero?` and `if` by expressions which work on number representations instead of numbers. We know how to represent a pair of values as a procedure, and we similarly define procedures `triple`, `first`, `second`, and `third` for forming procedural representations of any triple of values and for selecting components of a represented triple (see below). This reduces the two-counter halting problem to the halting problem for the part of Scheme we used to handle the number and pair representations, along with `letrec` and combinations. So now we examine what part of Scheme is needed for these number and pair representations.

Definition 7. Let

B ::= `(make-pair (lambda (l r) (lambda (z) (z l r))))`
`(true (lambda (l r) l))`
`(false (lambda (l r) r))`
`(left (lambda (pr) (pr true)))`
`(right (lambda (pr) (pr false)))`
`(triple (lambda (v1 v2 v3) (make-pair v1 (make-pair v2 v3))))`
`(first (lambda (tripl) (left tripl)))`
`(second (lambda (tripl) (left (right tripl))))`
`(third (lambda (tripl) (right (right tripl))))`
`(zero?-proc (lambda (n) ... ))`
`(if-proc (lambda (t c a) ... ))`
`(inc-proc (lambda (n) ... ))`
`(dec-proc (lambda (n) ... ))`

$\widehat{0}$ ::= `(make-pair true true)`

$\widehat{n+1}$ ::= `(make-pair false` $\widehat{n}$`)` for $n \in \mathbb{N}$.

Problem 5.

5(a) Complete the definition of the binding for `zero?-proc` so that

$$\texttt{(letrec (}B\texttt{) (zero?-proc } \widehat{0}\texttt{))} \downarrow \texttt{\#t}$$

and

$$\texttt{(letrec (}B\texttt{) (zero?-proc } \widehat{n+1}\texttt{))} \downarrow \texttt{\#f}.$$

5(b) Complete the definition of the binding for `if-proc` so that

$$\texttt{(letrec (}B\texttt{) (if-proc true } C\ A\texttt{))}$$

is equivalent to `(letrec (`B`)` C`)`, and

$$\texttt{(letrec (}B\texttt{) (if-proc false } C\ A\texttt{))}$$

is equivalent to `(letrec (`B`)` A`)` for any Scheme expressions C, A such that *both* `(letrec (`B`)` C`)` $\downarrow$ and `(letrec (`B`)` A`)` $\downarrow$.

5(c) Complete the definition of the binding for `inc-proc` so that

$$\texttt{(letrec (}B\texttt{) (inc-proc } \widehat{n}\texttt{))]}$$

is equivalent to

$$\texttt{(letrec (}B\texttt{) } \widehat{n+1}\texttt{)}$$

for all natural numbers n.

5(d) Complete the definition of the binding for `dec-proc` so that

$$\texttt{(letrec (}B\texttt{) (dec-proc } \widehat{n+1}\texttt{))}$$

is equivalent to

$$\texttt{(letrec (}B\texttt{) } \widehat{n}\texttt{)}$$

for all natural numbers n.

5(e) Suppose a sequence of instructions defining a two-counter machine, M, is given. Sketch how to define, in a small fragment of Scheme, an expression N computing a version of $next_M$ using the procedural representations of number triples. Namely, if $next_M(n_0, n_1, n_2) = (m_0, m_1, m_2)$, then

$$\texttt{(letrec (}B\texttt{) (}N\ \widehat{n_0}\ \widehat{n_1}\ \widehat{n_2}\texttt{))}$$

is equivalent to

$$\texttt{(letrec (}B\texttt{) (triple } \widehat{m_0}\ \widehat{m_1}\ \widehat{m_2}\texttt{)).}$$

5(f) To summarize, we have indicated why the halting problem remains undecidable even for a small fragment of Scheme. Exhibit a BNF grammar defining this small fragment. *Hint*: Since none of the bindings in B are recursive, the `letrec` of the multiple bindings B can be eliminated, leaving the binding of `try` as the sole `letrec` binding needed. (There is even a trick for eliminating the sole remaining `letrec`, leaving only combinations of lambda expressions and variables, but we won't go into that here.)