Scheme Computability, II

1 Meta-Circular Interpreters and Universal Machines

The builtin operator `eval` has become an official part of Scheme only in the latest (R5RS) Scheme specification. It also officially requires an extra environment argument. Since we're doing careful theory, we need to be careful about using `eval` as a half-decider for Halts. Here is what we really have in mind for an "eval" procedure:

Definition 1. A *meta-circular interpreter* for Scheme is a closed Scheme expression, meta-eval, such that for all S-expressions, S, `(meta-eval '`S`)` and S are Scheme-equivalent.

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

Let's review some of the many ways we might construct meta-circular interpreters. First, there is the basic one in Abelson/Sussman textbook. Second, 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 this second meta-circular interpreter could be expressed as

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

Still a third, altogether different meta-circular interpreter is our Substitution Model interpreter based on term rewriting on the course web-page.

A proper proof of Theorem 1 requires that we prove that one or another of these interpreters really is one, i.e., that its behavior according to our formal definition of Scheme by term rewriting satisfies the equivalence condition of the Theorem. This would most easily done with the simplest interpreter, namely, Abelson/Sussman's, but that interpreter does not implement the `call/cc` feature. To handle that feature, one needs a tail-recursive version of the Abelson/Sussman interpreter[1]. Developing a correctness proof even for the simple tail-recursive interpreter is an interesting and informative exercise in program verification. It also is a fairly complex task, involving proofs many more rules for reasoning about Scheme equivalences than we havew developed so far. On the other hand, the existence of a meta-circular interpreter is a very familiar fact, and most of the ideas involved in its correctness proof are not needed elsewhere in developing Scheme computability theory, so we shall omit it.

Now we have really established:

Corollary 1. Halts is half-decidable.

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

[1] There is one available in the Fall '97 6.001 page for pset 9

Problem 1. None of these meta-circular interpreters implement the `(lambda l` $\langle$body$\rangle$`))` form for procedures with varying numbers of arguments. Of course this is not hard to do, as we saw in an earlier assignment. Doing so would allow us to obtain a stronger, more natural version of of Definition 1, namely,

> **Strong Meta-evaluation**. For all Scheme expressions, E, the following equation is a sound Scheme equivalence:
>
> $$\texttt{(meta-eval '}E\texttt{)} = E.$$

In the rest of this exercise we show that the strong meta-evaluation condition does not hold without the varying argument feature.

1(a) Show that if a procedure, meta-eval, satisfies the strong meta-evaluation condition, then it is indeed a meta-circular interpreter according to Definition 1.

1(b) Let E be a Scheme expression without the `(lambda l ...` form, and let n be an upper bound on the number of formal parameters of the lambda expressions in E. Suppose $E \downarrow L$. Argue that n is also an upper bound on the number of formal parameters of any lambda expression in L. Conclude that there is no strong meta-circular evaluator.

1(c) Explain how to obtain a strong meta-circular evaluator if we include the `(lambda l ...` form in kernel Scheme. For simplicity, ignore `call/cc`. *Hint*: Such an interpreter can be obtained by modifying the "analyzing" interpreter described in the Abelson/Sussman text.

Meta-circular interpreters are an example of *universal machine*: one program or machine which is capable of simulating any other. The existence of universal machines was first proved by a researchers in mathematical logic, notably the German Kurt Gödel and the Englishman Alan Turing, in the early 1930's. The possibility of a universal machine stimulated the development of the first general purpose digital computers in the late 1930's and early 40's.

2 Decidable and Half-Decidable Sets

There is another informative way to prove Corollary **??** that $\overline{\text{Halts}}$ is not half-decidable, appealing only to Scott's first rather than second, theorem. Namely, we shall prove that if a set and its complement are *both* half-decidable, then the set is decidable. That's why we chose the terminology "half"-decidable in the first place. Assuming this claim for the moment, then since we know Haltsis not decidable by Scott's first Theorem **??**, but is half-decidable by the Corollary above, it cannot be that case that its complement is also half-decidable.

The proof of the claim is fairly easy: if H_1 is a half-decider for some set $\mathcal{S}$ and H_2 is a half-decider for its complement, then to decide whether $S \in \mathcal{S}$, just evaluate $(H'_1 S)$ and $(H'_2 S)$ in parallel. Exactly one of these expressions will halt; if the application of H_1 halts then $S \in \mathcal{S}$, while if the the application of H_2 halts then $S \notin \mathcal{S}$.

The only problem with this argument is that to show $\mathcal{S}$, we are supposed to describe a *Scheme* procedure which is the decider. The preceding decision procedure involves running two Sheme evaluations in parallel, but Scheme has no such run-in-parallel operation. So we need is to describe how to implement this run-in-parallel operation.

To do this, we 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 sexp(S) is a printable value, the interpreter returns `(cons 'printable` sexp(S)`)`. If S halts in at most n steps and sexp(S) is not a printable value, the interpreter just 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-eval, such that for all S-expressions, S,

1. sexp(`(step-eval '`S $\widehat{n}$`)`) is defined for all $n \in \mathbb{N}$,
2. $S \downarrow$ iff there is an $n \in \mathbb{N}$ such that sexp(`(step-eval '`S $\widehat{n}$`)`) $\neq$ `#f`, and
3. for all $n \in \mathbb{N}$, if sexp(`(step-eval '`S $\widehat{n}$`)`) $\neq$ `#f`, then
   - if sexp(S) is a $\langle$printable value$\rangle$, then

     sexp(`(step-eval '`S $\widehat{n}$`)`) = `(cons 'printable` sexp(S)`)`,

   - if sexp(M) is not a $\langle$printable value$\rangle$, then

     sexp(`(step-eval '`M $\widehat{n}$`)`) = `#t`.

Theorem 2. 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 3. $\mathcal{S}$ is decidable iff $\mathcal{S}$ and $\overline{\mathcal{S}}$ 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 $\mathcal{S}$, and H_2 be a half-decider for $\overline{\mathcal{S}}$. We may assume wlog that whenever an application of H_1 or H_2 halts, it returns `#t`. Then a decider for $\mathcal{S}$ is

```
(lambda (v)
  (define (try n)
    (cond ((step-eval (list 'H1 v) n) #t)
          ((step-eval (list 'H2 v) n) #f)
          (else (try (+ n 1)))))
  (try 0))
```

■

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

Proof: Let $H_{\mathcal{A}}$ be a half-decider for $\mathcal{A}$, and $H_{\mathcal{B}}$ for $\mathcal{B}$. Again assume wlog that whenever an application of these half-deciders halts, it returns `#t`. Then

```
(lambda (v) (begin (HA v) (HB 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-eval (list 'HA v) n)
        (step-eval (list 'HB 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-eval to some arguments could *interfere* with subsequent applications of step-eval to those same, or other, arguments. That is, the first application could have a side-effect which changed the subsequent behavior of step-eval 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 `(meta-eval '`M`)` will always evaluate to the same printable value, if any, no matter what computations have preceded the application, as long as meta-eval 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) (meta-eval (list '`D` qs)))`

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

Problem 3. Prove

Theorem 5. Decidable sets are closed under union, intersection, and set-theoretic difference[3].

Problem 4. Assume without loss of generality that half-deciders return `#t` when they halt. Explain why

`(lambda (v) (or (`$H_{\mathcal{A}}$` v) (`$H_{\mathcal{B}}$` v)))`

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

Problem 5. 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.

3 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:

[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.

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

Definition 3. 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 6. Sketch how to write a Scheme program computing the verifier for the arithmetic equation proof system of Handout 3.

Theorem 6. The set of theorems of any proof system is half-decidable.

Proof: The range of v is half-decidable by Theorem **??**, 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 4. 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 $\texttt{(}M \texttt{ = } N\texttt{)}$ is *valid* iff $M \cong N$. As usual, we use the notation $\models Q$ to mean that equation Q is valid.

Lemma 1. 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 7. Prove Lemma 1. *Hint*: This will take some work.

Lemma 2. The set of *closed* Scheme expressions which do not halt is not a half-decidable set.

Problem 8. Prove Lemma 2. *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 $g : \langle\text{s-expression}\rangle \to \langle\text{s-expression}\rangle$ 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 **??**, $\mathcal{N}$ would be half-decidable as well, contradicting Lemma 2. ■

Definition 5. A proof system, v, for Scheme equations is *sound* iff $\vdash_v\ Q$ implies $\models Q$ for all Scheme equations, Q.

Theorem 7. 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 6 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. ■

4 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 9. 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.)