Determining the **Limitations** of Computation

Prof. Albert R. Meyer
MIT Laboratory for Computer Science
meyer@theory.lcs.mit.edu

December 9, 1997

LANGUAGE WARS

Is Scheme as POWERFUL as C (or Java, or C++, or...)?

Is C (or Java, or C++, or...)? as powerful as Scheme?

What is it that Scheme CAN'T do?

LIMITATIONS BY **DESIGN**

- PROCEDURES ARE "BLACK BOXES"

For example, it is impossible to define a Scheme procedure `BIG-BODY?` such that

```
(BIG-BODY? P)
```

returns `#t` iff the body of procedure `P` longer than 100.

It is even impossible to define a Scheme procedure `PRIMITIVE?` such that

```
(PRIMITIVE? P)
```

returns `#t` iff procedure `P` is primitive.

This is as it should be: we $WANT$ the only possible uses of procedures to be

- APPLYING them to arguments,
- passing them around,
- storing them in data structures.

MORE LIMITATIONS BY DESIGN

- SCHEME IS "SEQUENTIAL"

It is impossible to define a Scheme procedure `PARALLEL-RETURN?` such that

$$\begin{array}{l}\texttt{(PARALLEL-RETURN?}\\ \quad\texttt{(lambda ()}\ < E_1 >\texttt{)}\\ \quad\texttt{(lambda ()}\ < E_2 >\texttt{))}\end{array}$$

returns `#t` **iff** at least one of the expressions

$$\texttt{(lambda ()}\ < E_1 >\texttt{)}$$

and

$$\texttt{(lambda ()}\ < E_2 >\texttt{)}$$

returns a value.

MORE LIMITATIONS BY DESIGN

Scheme uses a `cons` cells for storage. The garbage collector can freely rearrange cells without changing how they behave. That's why we want:

- It is impossible to define a Scheme procedure which sets absolute memory location `00017A8` to zero. (This is the sort of thing you **can** do in C.)

LIMITATIONS BECAUSE OF **APPROXIMATIONS**

In MIT Scheme,

- `(= 1 1.0000000000000001)` returns `#t`.

Limitations by Design can be Overcome by RE-DESIGN

Modify the Scheme intepreter in various (not always tasteful) ways. For example,

- `BIG-BODY?` is easy to add to `MEVAL` as a builtin procedure because `MEVAL` includes the body of a compound procedure in its representation of the procedure; `PRIMITIVE?` is similarly easy to add in.

- `PARALLEL-RETURN?` can also be added as a builtin, but it's trickier: define a loop which does alternate steps of the evaluation of `(lambda ()` $< E_1 >$`)` and `(lambda ()` $< E_2 >$`)`.
- Avoid the `(= 1 1.0000000000000001)` problem by keeping EXACT numerical values.
- Access absolute memory locations by letting Scheme do a Remote Procedure Call to a `C` program with access to absolute memory locations. (Olin Shivers adds such RPC's to Scheme as part of a project to develop Scheme as a systems programming language).

LIMITATIONS FROM OUR **IGNORANCE**

Scheme 48 reports a warning **before** evaluating

```
> (1 (- 2 2))
Warning: non-procedure in operator position
         (1 (- 2 2))
         (procedure: #{Type :exact-integer #f #f})
```

but evaluates

```
> (/ 1 (- 2 2))
Error: rational division by zero
       1
```

The Java compiler reports a divide-by-zero error **before** evaluating
`1 /(2 - 2)`,
but does evaluate
`1 /(x - x)`.

We don't expect *any* compiler to detect the divide-by-zero error in

```
( / 1
    (- (fib (* 2 n))
    (ceiling (* (sqrt 5)
                (square (fib n))))))}
```

Are we sure that the compiler can't handle Fibonacci identities? It might seem theoretically possible to build a theorem-prover into the compiler and deduce that

$$\text{fib}(2n) = \left\lceil \sqrt{5}(\text{fib}(n))^2 \right\rceil$$

for all n `>` 2.

In any case, now that we know that we're supposed to detect this error, we could build an *ad hoc* test for it into `MEVAL`.

So we can overcome limitations from ignorance by

GETTING SMART

LOGICALLY UNAVOIDABLE LIMITATIONS

As a warmup for the logic, let `<make-self-app>` be an expression such that

```
(<make-self-app> '<expr>) ;input expression
```

evaluates to

```
(<expr> '<expr>)           ;output notation
```

Namely,

```
(define (make-self-app quoted-expr)
  (list quoted-expr
        (list 'quote quoted-expr)))
```

Then evaluate

```
(make-self-app 'make-self-app)
```

and get the output

```
;Value: (make-self-app 'make-self-app)
```

SELF-REPRODUCTION!?!

THE SCHEME HALTING PROBLEM

A Scheme expression is said to HALT if it returns a value when it is evaluated (exactly once, in the Scheme initial environment).

Is there a single Scheme expression, `<halts?>`, which decides whether an expression halts? Namely, can it be that for *every* Scheme expression, `<expr>`, we have

CONTRACT:

> `(<halts?> '<expr>)` returns `#f`
> **IFF** `<expr>` does NOT halt,

NOTE: Like an intepreter or compiler, `<halts?>` gets the *quoted* `<expr>` as its argument, and so it can "study" `<expr>` without necessarily evaluating it.

So *suppose* there was a <halts?>.

Let <perverse> be the lambda expression:

```
(lambda (expr)
  (if (<halts?>
       (list expr (list `quote expr)))
      (/ 1 0)     ;ERROR DOES NOT "HALT"
      #t))
```

Then for EVERY expression <expr>, we have by definition
(<expr> '<expr>) does *not* halt iff
(<halts?> '(<expr> '<expr>)) returns #f
iff (<perverse> '<expr>) returns #t
iff (<perverse> '<expr>) *does* halt.

Summary: for every expression `<expr>`,

`(<expr> '<expr>)` does *not* halt iff
`(<perverse> '<expr>)` *does* halt.

In particular, choose `<expr>` be the expression `<perverse>`, so

`(<perverse> '<perverse>)` doesn't halt
iff it does halt.

CONTRADICTION!

We have proved by contradiction:

> THEOREM: It is **logically impossible** to write a `<halts?>` expression in Scheme. In other words, no Scheme program can detect the "Not-Halting" property of Scheme expressions.

SLOGAN:

> The *Halting Problem* for Scheme expressions is not *solvable* by a Scheme program.

But halting is hardly interesting by itself: real programs had better do more than just halt. Maybe unsolvability is isolated to uninteresting properties?

NO! Halting is the tip of the iceberg: essentially ALL properties which depend only on the printable output behavior of Scheme expressions are similarly *unsolvable*.

So SCHEME can't solve the Scheme not-halting problem, but maybe a `C` program could solve the Scheme problem?

NO! Scheme could SIMULATE any `C` solution, and this Scheme simulation would then be a Scheme procedure solving the problem.

It is really **impossible** to have a `<halts?>` expression for Scheme in ANY programming language!

Maybe Scheme halting is difficult to determine because its uses of *Integers* are hard to understand?

NO! Integers can be represented by "pure" number-free procedures, so there is not even a `<halts?>` which handles just the number-free Scheme expressions.

Maybe Scheme halting is difficult to determine because *Recursively* defined procedures are hard to understand?

NO! Recursive definitions can be simulated by "pure" applicative expressions, so nonrecursive expressions are just as hard to understand. (*SICP* Ex 4.21)

FIXED POINTS INSTEAD OF DEFINE'S

```
(define (factdef fact)
      (lambda (n)
         (if (<= n 0)
             1
             (* n (fact (- n 1))))))

(define fact0 (factdef (lambda (n) 0)))

(define fact1 (factdef fact0))

(define fact2 (factdef fact1))
```

We want a procedure `fix` such that

$$\texttt{(fix d)} = \texttt{(d (fix d))}.$$

Then could

```
(define fact (fix factdef))
```

```
(define (fix d)
 ((lambda (x) (d (lambda (z) ((x x) z))))
  (lambda (x) (d (lambda (z) ((x x) z))))))
```

Technicality:
`(fix d)` = `(lambda (n) ((d (fix d)) n))`
is what we can actually accomplish; that's good enough.

Verify that this works by a substitution-model calculation (or by running it in Scheme).

To LEARN MORE THEORY, consider:

6.044 *Computability Theory Of and With Scheme*, Instructor: Prof. Albert R. Meyer,

`ftp://theory.lcs.mit.edu/pub/classes/6.044/homepage.html`

6.045 Theory of Computation

6.821 Programming Languages