Limitation Theorems for Boolean Machines

Jeffrey A. Barnett[*]

August 10, 2022

Abstract

A simple class of abstract computers is defined: *Boolean Machines* (BM) are entities that create and compose Boolean expressions, without quantifiers, from arguments. BM also provide a primitive that determines whether an expression is satisfiable. The Halting Theorem for Turing Machines is adapted to this class of machines and we decide if an old proof is still applicable. Halting is "analogized" to mean that the value of a BM applied to arguments is a constant, where *constant* means either the value or its negation is a tautology. The goals of this investigation are 1) get extra mileage from an existing proof if possible and 2) learn what we can from either success or failure of the venture. A slight riff on Rice's Theorem is part of the conclusion

1 Introduction

Turing Machines (TM) are generally accepted as proper theoretical models of the limits of realizable computational capabilities. *Realizable* is taken to mean doable mechanically without recourse to an oracle or chicanery; though machine size is assumed finite, it is unbounded. A. Turing introduced TM [6] in 1936 though he called them something different. The hypothesis that the model is a good one is strengthened by the facts that many other models of computation have been proven equivalent and that no one has suggested a valid counterexample. It is assumed, herein, that readers have a basic acquaintance with Turing machines; an accessible introduction is afforded in Linz [4].

Perhaps the most interesting and well-studied result in computation theory is the *Halting Theorem* (HT) that simply states: There does not exist a single TM that can take any TM description, plus arguments for it, and determine if that computation would be completed in a finite number of steps. If a computation completes, it is said to halt, otherwise it does not halt. N.B. The fact that a computation does not halt is no indicate that it is inherently flawed; consider a computation that exposes the digits of π one at a time.

HT has been proved in literally dozens of different ways [2, 3, 4, 6]. One proof that seems relatively clever appears for Theorem 12.1, a statement of HT, in Linz [4]. That proof is examined herein by adapting it to an entirely different class of computations to see what happens. What I will do is 1) define a class of Boolean Machines (BM) that

[*]The author can be reached at jbb@notatt.com.

calculate Boolean expressions from arguments, 2) define such a machine/input pair as having a "constant" result if the value, a Boolean expression, reduces to true or false, and not constant otherwise, and 3) define and nominate one such machine as our candidate universal constancy determiner and make an analogy between halting for TM and constancy for BM.

2 Plan of Attack

The discussion in the remainder of this note is summarized here:

Section 3 reproduces the proof of HT for TM mentioned above. It sketches the definitions needed and gives an informal English version of that proof.

Section 4 introduces BM machines and the language that defines them by examples and brief discussions.

Next, Section 5 transplants the proof in Section 3 for the TM HT to the world of BM. The proof steps and terminology used are chosen so that the mapping between this proof and the prior one is easy to spot. An analysis of the proof follows the evaluation of a rather "tricky" case and we find that the proof is based on a hidden assumption. What we learn rules out simulation as a methodology to analyze computational behavior in many cases and some alternatives are briefly discussed.

Finally Section 6, the conclusion of this article, turns new-found insights into a better, i.e., a correct, theorem that properly categorizes what a universal BM constancy decider can do. That theorem will be translated back to the world of TM but the result will be somewhere among pathetic, a little joke, or simply provoke the question "So what exactly did you expect?" A variation on Rice's Theorem [5] is introduced too.

There are three appendices that serve, together, to introduce and formally define BM machines and the language used to specify them:

The syntax description of the BM language is presented in Appendix A along with a brief introduction to the metalanguage used to define that syntax.

Appendix B introduces the formalities of applying a BM to arguments and this necessitates discussion of some nonsyntactic constraints inherent in the definition and applications of these machines.

Appendix C presents a straw man implementation, in Common Lisp, of an interpreter to execute a BM applied to arguments.

Appendix D deals with a halting problem variation for TM and may rightly be considered off topic.

The proofs about BM that appear in the main body of this article rely on Appendices A, B, and C for grounding.

3 A Proof of HT

This section recapitulates the proof of HT, found in Linz [4] as Theorem 12.1. I am not clear who spotted this proof originally but it is similar to one shown me in the mid 1960s. Figure 1 provides some of the material to be discussed below. The proof starts with the assumption that it is possible for a single function—`h` in the figure—with arguments `tm` and

```
Procedure h (tm, args)
     If "tm applied to args halts" Then Go yes Else Go no;
yes: Halt;
no:  Halt;
End h;

Procedure h′ (tm, args)
     If "tm applied to args halts" Then Go yes Else Go no;
yes: Go yes;
no:  Halt
End h′;

Procedure ĥ (tm)
     h′(tm, tm);
End ĥ;
```

Figure 1: Linz 12.1 refutation of TM halting theorem.

`args` to decide whether the computation `tm(args)` halts or not: halting of that computation is signalled by `h` itself halting at the label `yes`; otherwise it halts at the label `no`.

Note that any TM and any argument or arguments to a TM can be represented easily as character strings.[1] The interpretation of a TM-defining string is set by convention and there are an arbitrary number of ways to choose. The convention for interpreting an argument string is instituted while specifying the TM and must be know to whoever or whatever provides the arguments. Selecting these conventions is not part of the HT definition and we take advantage of that here by using simple-to-read forms (as in Figure 1) that should be comprehensible to any computer professional.

The interior of `h` contains the phrase `"tm applied to args halts"` and the substance of HT is that there is no way to implement an `h` using this predicate for all `tm`–`args` pairs. We now assume that it *is possible* and proceed to a contradiction.

If `h` is a realizable Turing machine, so is the `h′` shown next in the figure. When `h′` is applied to a `tm`–`args` pair the result is

If `tm(args)` halts then `h′` loops forever, i.e., `h′` doesn't halt.
If `tm(args)` does not halt then `h′` halts (at label `no`).

Next, `ĥ` is constructed from `h′` as shown in the figure. `ĥ` simply takes its single argument, `tm`, and passes it in both argument positions to `h′`. So when `h′` is applied to the `tm`–`tm` pair the result of `h′` and, hence, `ĥ` is

If `tm(tm)` halts then `ĥ(tm)` doesn't halt.
If `tm(tm)` does not halts then `ĥ(tm)` halts.

Now `ĥ` is itself a TM so it is legitimate to ask about the result of `ĥ(ĥ)`. It is easy to see that

[1]Even a one-character input alphabet will do. The length of the string could be interpreted as the Gödel number of a multiple-object vector in a larger alphabet.

If $\hat{\texttt{h}}(\hat{\texttt{h}})$ halts then $\hat{\texttt{h}}(\hat{\texttt{h}})$ does not halt.
If $\hat{\texttt{h}}(\hat{\texttt{h}})$ does not halts then $\hat{\texttt{h}}(\hat{\texttt{h}})$ halts.

This is of course the rankest nonsense! The only real assumption made here is an `h` with the predicate `"tm applied to args halts"`. Since that assumption leads to a contradiction, we must abandon it as false and this is a proof of the halting theorem. □

4 Brief Introduction to BM with Examples

Since the intent herein is to apply a known proof structure to a new class of computation, one has the right to demand some formality in the available knowledge about that class. Unfortunately, rigor in this area—defining classes of computation and associated programming languages—is boring reading (and writing as well) and in this case, probably not necessary. So here is a compromise:

This section 1) defines a few lexical conventions such as what various font uses indicate, 2) introduces language features of Boolean Machines, BM, such as scoping rules, and 3) provides a few examples to convey a decent impression of "how it all works."

There are three appendices that provide a more rigorous definition of the language including its syntax, context sensitive rules, and its intended method of evaluation. If at any point, what you are reading and your understanding differ, sort it out by leafing through an appropriate appendix.

Appendix A defines a simple BNF-like language that is used to specify the syntax of the BM "programming" language. That language specification is arranged to give hints as to the intent of what is being defined. For example, `<bvar>` is a Boolean variable and `<mvar>` is a motor-valued variable, where a `<motor>` is a function definition. However, a context-free grammar cannot actually enforce that such variables always have proper values.

This is a good place to comment on the use of `typewriter` font: it is used throughout when presenting program fragments such as `(AND a b)` and metavariables such as `<bexp>`. In the later case the metavariable can be used as a name or determiner to designate a piece of code or example in context such as 'then `<bvar>` is evaluated'. Further, in examples of code, words in the language, such as `AND`, are capitalized while user-chosen names, such as `b`, are not.

Appendix B presents context-sensitive rules that supplement areas under-defined by the syntax specification alone. The rules for evaluating forms are elaborated in detail too.

Finally, Appendix C presents a complete interpreter for BM machines/programs. The interpreter is written in standard Common Lisp. It has not been tested or debugged so read with an open mind. That code is commented fairly well so there is some redundancy to support understanding.

4.1 What kind of language is it?

The BM language is described here in terms of some of its properties. Even though BM are discussed as a class of abstract machines, we only can know them by examples of their descriptions as programs in a programming language. So do not let descriptions that play on the machine/program duality confuse.

Property 1 (Side effects)**.** No operation in BM has side effects. √

Property 2 (Control statements)**.** There is no concept of conditional execution, transfer of control, or iteration in BM. √

Property 3 (BM has a Lisp-like syntax)**.** The BM language has two logical constants, `!TRUE` and `!FALSE`, and `<var>`, a syntax category for variables. In addition to these atomic forms, we have forms that resemble `(operator $ operand)`, where `operator` is a functional form that can be applied to arguments, `$` indicates that zero or more of the following term are expected, and `operand` is an atomic or list expression. So all expressions or forms are self terminating because of parentheses. √

Property 4 (Value types)**.** There are only two types of values that may result from evaluation: Boolean expressions and closures. A Boolean expression resulting from evaluation will consist of Boolean constants, free Boolean variables (`<bvar>`), and list expression with `AND`, `OR`, or `NOT` operators. These Boolean expressions resulting from evaluation are called simplified Boolean expressions (`<bexp>`). A closure is a pair of a function specification (`<motor>`) and an initial context in which to locate the values of visible bound variables. √

Property 5 (Boolean expressions before evaluation)**.** Boolean expressions in the BM language have more possible sorts of contents than the simplified `<bexp>` introduced above. In particular, they may contain `<satp>`, bound `<bvar>`, and `<subst>`. A `<satp>` form is `(SATP <bexp>)`. The form's value is `!TRUE` if the `<bexp>` simplified by evaluation can be made true by some substitution of Boolean constants for its free variables; its value is `!FALSE` otherwise. A `<subst>` form applies a closure to evaluated arguments; the argument values are bound to variable names and may be referenced within the lexical scope of those bindings. `<subst>` are further described next. √

Property 6 (Function definitions)**.** A function definition is called a `<motor>`; an evaluated `<motor>` is called a closure (in 5 above); and a `<subst>` applies a closure to arguments. A syntax definition of a `<motor>` resembles: `(`λ` ($vars) <bexp>)`, where λ serves as a special operator to introduce this function definition. That is followed by a list of zero or more variables and a Boolean expression. When this functional is "called" parameters are evaluated, bound to these variables, and the imbedded `<bexp>` is evaluated in a context where those bindings are visible: `(`λ` (a b c) (AND a (OR b c)))` is a example. If the parameters passed are `ax`, `bx`, and `!FALSE` where `ax` and `bx` are unbound variables, the result is `(AND ax (OR bx !FALSE))` which might or might not be simplified to `(AND ax bx)`. So functions simply substitute parameter values for references to variables that bind them. √

Property 7 (Applying a closure to arguments)**.** The form that actually applies a function to arguments is `(APPLY closure $ <arg> ())`. In this article, *apply* is meant as in apply cosine to angle. √

Property 8 (Variable scopes are lexical)**.** This is compatible with newer Lisps that allow both lexical and dynamic scope regimes. Note, that the use of lexical scoping is different than the regimes in the majority of Lisps that exist today. N.B. Though BM code resembles both Lisp and the λ calculus, it is neither: Computational capabilities are distinctly less. √

4.2 Language examples

The top-level form defined in the language is called `<run>` and it is simply a form with the operator `RUN` and the operands are a mixture of Boolean machine definitions with names (`<bm-def>`) and test cases (`<test>`). Each form is evaluated in a context where only the names of the previously defined `<bm-def>` are visible. A `<bm-def>` is simple `(DEFINE <bm-name> <motor>)` and its evaluation just makes a closure of `<motor>` in the context of the previously defined `<bm-name>`. A `<test>` is just an `<apply>`, see Property 7.

Let's look at some examples. Assume that there is a system that accepts the `<run>` operands, evaluates forms one by one, and prints the `<bm-name>` when a `<bm-def>` is encountered and the value returned when a `<test>` is evaluated. The first example is the definition of a `<bm>` named `implies` that takes two arguments, `b1` and `b2`, and returns a form that is the logical way of saying that `b1` implies `b2`:

```
→  (DEFINE implies (λ (b1 b2) (OR (NOT b1) b2)))
←  implies DEFINED
```

The system simply acknowledges the definition. We next input a `<test>` form that uses the newly defined `implies` as the operator.

```
→  (APPLY (BM implies) x (OR x y) ())
←  (OR (NOT x) (OR x y))
```

As is obvious, the two argument, `x` and `(OR x y)`, are simply substituted, respectively, for references to the variables `b1` and `b2` while evaluating the body of `implies`. Ignore the `()` in these examples; the reason for their appearance will be explained later.

Now a new BM is defined that calculates whether a `<bexp>` is a constant—see the discussion introducing Eq. 1, page 8. The `<bm-def>` for `c?` captures that idea:

```
→  (DEFINE c? (λ (b) (NOT (AND (SATP b) (SATP (NOT b))))))
←  c? DEFINED
→  (APPLY (BM c?) (APPLY (BM implies) x (OR x y) ()) ())
←  (NOT (AND !TRUE !FALSE))
```

This example is more complicated. A `<subst>` is used to calculate the form that logically means that `x` implies `(OR x y)` as shown above and *that* form is the one passed to `c?` where the value is generated. It is permissable for an implementation to simplify what is shown here to `!TRUE` but it isn't required.

These few examples are here to give a hint as to the type of language defined and its intended operational model. We now turn our attention to details of argument passing and partially explain those '`()`' in more detail. Consider the following:

```
→  (DEFINE foo (λ (x f !REST args) (AND x (APPLY f args))))
←  foo DEFINED
→  (APPLY (BM foo) g (BM implies) c d)
←  (AND g (OR (NOT c) d))
```

In this case, `foo` is defined as a function that takes at least two arguments: the first should be a `<bexp>` that is evaluated and bound to `x` and the second should be a `<motor>` (known because `f` is applied to arguments) that is evaluated and bound to `f`. Additional arguments, if any, are evaluated and a list of the values is bound to the rest variable `args`. Notice that when `f` is applied, there is no `()` there, rather a rest variable is. Note that `()` is the empty list so the notation is quite consistent. Note that the only place that a rest variable may be referenced (as opposed to bound) is as the last argument to `APPLY`.

It should be obvious from these few examples that the only real behaviors that may be programmed in BM is substitution, some optional simplification, and determining if an evaluated `<bexp>` is satisfiable.

5 Halting Theorem Analogy for BM

A conjecture about BM that is similar to HT for TM is formulated, a proof proposed, and the result analyzed. The conjecture is formalized in Section 5.1 then a proof is sketched in Section 5.2 that is similar to the one that confirms the HT for TM presented in Section 3. The proof for BM is analyzed and an unwarranted assumption is uncovered. Finally, Section 5.4 discusses some techniques that might be applied to determine properties of programs in cases where a universal determiner or even faithful simulation is not productive.

5.1 A conjecture about BM

Above, the concept of a *constant-valued* Boolean expression has been bandied about. It is now time to make that concept more precise.

Definition 1 (Constant-valued Boolean expression)**.** A *constant-valued Boolean expression* is one that always evaluates to true or always evaluates to false. The evaluation must be identical no matter what values are substituted for the free variables in the expression. N.B. the same value, true or false, must be substituted for all instances of a single variable.

In the BM language, `!TRUE` and `!FALSE` are constants as are forms such as `(AND v (NOT v))` and `(OR v (NOT v))`. On the other hand, a single free variable, e.g., `v`, and forms such as `(OR v !FALSE)` are not constants. The problem of determining whether quantifier-free Boolean forms are constants is trivial though the computations to determine this fact can be expensive.

The language provides a `<satp>` primitive that can be used to calculate constancy of a quantifier free Boolean expression. This primitive, however, simply determines if an expression is satisfiable:

Definition 2 (Satisfiability)**.** Let $[v_1 \ldots v_n]$ be a vector of all the unique free Boolean variables that occur in the Boolean expression, x. Then x is *satisfiable* if and only if there exists a vector, $[b_1 \ldots b_n]$ where 1) each b_i is either true or false and 2) x evaluates identically true when b_i is substituted for every occupance of v_i in x.

Now note that every quantifier free Boolean expression will evaluate either true or false when such a substitution is done. The expression is constant if and only if all 2^n possible substitutions produce the same value. Therefore, for a Boolean expression to not be a constant there must exist some substitution vector that makes it true and some other vector that makes it false. Ergo, as the primitive form `<satp>` determines if a vector exists that makes an argument form true, the criterion that a Boolean expression, b, *not* be a constant is,

```
(AND (SATP b) (SATP (NOT b)))
```

because if `(SATP (NOT b))` is true, then the confirming substitution will make `b` false. So, the criterion that b is constant is

```
(NOT (AND (SATP b) (SATP (NOT b)))).          (1)
```

We can now state the conjecture that we want to prove.

Conjecture 1 (Constancy theorem for BM)**.** There does not exist a single universal constancy determiner, a BM, that takes as arguments a `<mexp>`, `m`, and `<arg-list>`, `a`, and determines if `(APPLY m a)` is a constant. A determiner must always return a value: `!TRUE` if a constant and `!FALSE` otherwise.

5.2 Proof of conjecture

We now construct a proof, similar to the one in the Linz book, that shows the constancy conjecture for BM is, indeed, a theorem. The difference here is that we are not checking halting; rather, the check is whether a `<motor>` applied to an argument list returns a constant Boolean expression. See Figure 1 and the surrounding discussion for a proof of HT. Figure 2 shows similar structures for the proof of the BM Constancy Theorem. N.B. All references

```
(DEFINE c? (LAMBDA (bexp)
             (NOT (AND (SATP bexp) (SATP (NOT bexp))))))

(DEFINE h (LAMBDA (motor !REST rargs)
            (APPLY (BM c?) (APPLY motor rargs) ())))

(DEFINE h′ (LAMBDA (motor !REST rargs)
             (AND (APPLY (BM h) motor rargs) h′--new)))

(DEFINE ĥ (LAMBDA (motor)
            (APPLY (BM h′) motor motor ())))

(APPLY (BM ĥ) (BM ĥ) ())
```

Figure 2: Run components for proof of BM Constancy Theorem.

in the `<bm-def>` are to other `<bm-def>` or via locally bound variables so resolving object references and scope issues is trivial.

We start by defining a *constancy predicate*, `c?`, that uses Eq. (1). We know that BM `c?` evaluates to `!TRUE` if its argument is a constant and `!FALSE` otherwise. Note that `c?` is used as the BM equivalent of the predicate "`tm applied to args halts`" in Figure 1.

The arguments to `h` are the `<motor>` of some arbitrary Boolean machine and arguments for it. That `<motor>` is applied to the specified arguments and the result is passed to `c?`; the constancy predicate then decides matters.

If `h` is a realizable Boolean machine, so is the `h′` shown next in Figure 2. When `h′` is applied, the result is

If `motor(args)` is constant, the value of `h′` is a free variable, `h′--new`.
If `motor(args)` is not constant, the value of `h′` is `!FALSE`.

N.B. A Boolean expression that consist solely of a single free variable (`h′--new` is nowhere bound) is surely not constant but `!FALSE` is. Note the parallelism with `h′` in Figure 1.

Next, `ĥ` is constructed from `h′` as shown. `ĥ` simply takes its single argument and passes it in both argument positions to `h′`. So when `h′` is applied to the pair, the result of `h′` and, hence, `ĥ` is

If `motor(motor)` is constant, the result is not constant.
If `motor(motor)` is not constant, the result is the *constant* `!FALSE`.

Now `ĥ` is itself a BM so it is legitimate to ask about the result of `ĥ(ĥ)`. It is easy to see that

If `ĥ(ĥ)` is constant then `ĥ(ĥ)` is not constant.
If `ĥ(ĥ)` is not constant, `ĥ(ĥ)` is a constant.

This is of course the same nonsense seen in the proof of HT! The only real assumption made here is the existence of `h` as a decider. Since that assumption leads to a contradiction, we must assume abandon that assumption as invalid. □

5.3 Proof analyses

If one examines the run sequence shown in Figure 2, it quickly reveals an infinitely deep, recursive evaluation chain; in other words, evaluation would not halt. Figure 3 provides a trace of the evaluation to "establish" the BM constancy theorem. Two-tiered boxes in the

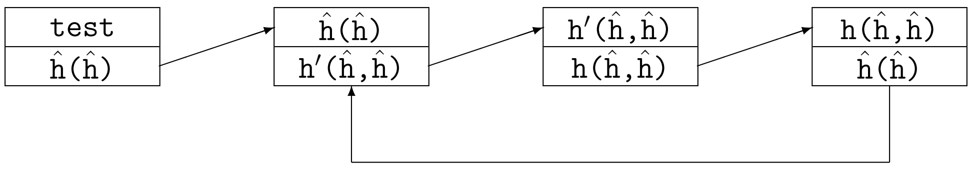

Figure 3: Trace of test run.

figure show a computational site: the top represents a place such as a test or an evaluation "at" a `<bm-def>` where that evaluation causes the evaluation shown in the bottom of the box to be done immediately; that inner evaluation causes an entry to another `<bm-def>`. That form will be the top of the following box. Let's walk through the trace. The first box occurs as a `<test>` evaluation in the `<run>` and the form evaluated is $\hat{\texttt{h}}$(`ĥ`). The second box shows that calculation carried out by $\hat{\texttt{h}}$ which needs to evaluate `h′(ĥ,ĥ)`.

We now arrive at the third box and see that the `h′` calculation evaluates `h(ĥ,ĥ)`. Now `h` applies its first argument to the rest of its arguments, i.e., it needs to evaluate `ĥ(ĥ)` as shown in the fourth box. And this evaluation takes us back to box two, `ĥ`, where the infinite loop is now readily apparent. Note that none of the evaluations in the loop are context dependent other than references to defined BM and local variables whose bindings are BM plus arguments for them that are never actually used. The same sort of loop must appear within any decider based solely on evaluation or jointly on tracing and simulation.

There are several important observations to emphasize at this point:

Remark 1 (It's not the predicate's fault)**.** We have not proven that an implementation must enter an infinite recursive loop; that's not what has been shown. What has actually been demonstrated is that there is no consistent value possible for `h` to return. $\surd$

Remark 2 (Nature's method)**.** Perhaps non halting is Nature's natural way to avoid exposing contradictory behavior! $\surd$

Remark 3 (Evaluated `<bexp>`)**.** When a `<bexp>` is evaluated, the result, if any, has reduced syntactic complexity. There are three differences: 1) no `<satp>` remain, 2) no `<subst>` remain, and 3) all remaining variables are free and unbound. So we have only expressions that are built from `<constant>`, free `<bvar>`, and forms with `AND`, `OR`, and `NOT` operators. Note that it is possible that evaluation of a `<bexp>` not produce a result because of a language violation, e.g., a syntax error, or infinite loop; but if evaluation does complete it will be of the reduced complexity described here.[2] $\surd$

Remark 4 (`<satp>`)**.** A working constancy predicate, `c?`, is constructible from `<satp>` as has been discussed earlier. The fact that its argument has been evaluated and is in simplified form makes that rather easy to see or prove with an induction argument. However, it was just shown that a test case can be constructed such that there is an infinite loop, *the constancy predicate is never applied*, and there is no consistent value if it were. $\surd$

Remark 5 (Halting Theorem)**.** The proof of the TM halting theorem faces a double gotcha. The first, like the BM problem, is that one can easily concocted a test case where the halting predicate will never be applied while an infinite evaluation chain develops. The second problem is that there is no universal halting predicate, period, as has been proved elsewhere in a variety of ways. $\surd$

5.4 If not simulation, then what?[3]

The use of simulation and/or interpretation are key technologies in the above discussions. In re Turing Machines, the terms typically denote a *Universal Turing Machine* (UTM), that may be presented with an encoding of a TM, and arguments for it, then faithfully

[2]Easy to prove by induction

[3]The contents of this Section are slightly off topic and can be skipped. However, they may be of interest to those seeking alternative methods to characterize computations.

simulate behavior state transition by state transition. Most models of computation have no such universal capability. The BM world does have this capability only because it has been endowed with the `APPLY` primitive as well as the possibility of using `<rest-var>`.

To paraphrase some of what was learned above: The possibility within a language to have a universal interpreter can provide new capabilities as well as introduce classes of unsolvable problems. The natural question to ask is whether there are alternatives to such simulation-style approaches that may resolve some otherwise intractable problems. That is the objective of the brief discussion in this section

There are some standard analytic techniques and tools to resolve infinite recursive definitions in the service of numerical iteration. Take, for example, the iterative definition $x_{n+1} = x_n^2 - 2x_n + 2$ along with an x_0. An attempt to solve for an x by performing this iteration is doomed to almost always go on forever. What is the stopping criteria? Of course, no competent numerical analyst would have a problem.

If this is to converge to some x value, that x must satisfy $x = x^2 - 2x + 2$, i.e., $x = 1$ or $x = 2$. The rest of the solution must determine which, if either solution will be reached for a given x_0. The possible solutions to this form of equation are called *fixed points* and their properties have been well-studied both in theory and for many specific applications.

What is not so well known[4] is that sets of Boolean equations may have multiple solutions that are fixed point of the sort of iterative substitutions that are used to solve these equations. Lets look at an example: the two simultaneous equations $a = b \wedge A$ and $b = a \vee B$, where a and b are variables and A and B are constants in the Boolean algebra hosting this exercise. The simultaneous solutions here are all of the form $a = A \wedge (B \vee \theta)$ and $b = B \vee (A \wedge \theta)$, where θ is any expression definable in the algebra.

If one imagines these solutions in a Venn diagram, we note that the solution where $\theta = \text{true}$ covers the maximum area possible and the one where $\theta = \text{false}$ covers the least. They are called, in the jargon, the *maximum* and *minimum fixed points* respectively.

If one is going to take the next step beyond simulation for the TM HT or the BM constancy problem, fixed point algorithms seems a logical methodology to study. No single algorithm will resolve all cases in either domain, but might make some progress. One should note that much of the proofs of program correctness or calculations of algorithmic time complexity involve similar techniques.

The recursion shown in Figure 3 above likely has no fixed point solution based on considerations of data types. A Boolean value is demanded but all involved data objects are closures of `<motor>`.[5] And you simply can't make a silk purse out of a pig's ear, or so the conventional wisdom says.

6 Comments on Limits of Computation Theorems

Since the BM formalism provides sufficient power to 1) do a universal interpretation of a `<motor>` applied to arguments and 2) can determine if a `<bexp>` simplified by evaluation is

[4]This branch of knowledge is only known if your career decisions have been interesting and you took a wrong turn along the way.

[5]The exception is `h′--new` but the embedding `AND` never completes; the recursion is through its first argument.

a constant (see Eq. 1, pp. 8), a positive capability theorem follows:

Theorem 3 (BM Constancy)**.** `h(motor, rargs)`, correctly decides if `motor(args)` is a constant if and only if `motor(args)` halts. See Figure 2.

Proof. Since `c?` is effective on evaluated `<bexp>`: If `motor(args)` halts and produces a value, so does `h(motor, rargs)`; If it does not halt, no value is produced. □

As has been pointed out above, there are two problems with the halting theorem: 1) lack of an effective halting predicate and 2) the possibility of an infinite loop and never applying the halting predicate. However, there is still room to squeeze a modest positive halting theorem result with similarity to the above:

Theorem 4. There exist a TM with arguments `tm` and `args` that correctly decides halting of `tm(args)` if and only if `tm(args)` halts.

Proof. Let the decider be `Func haltp (tm, args) tm(args) return true`. □

This may seem rather silly but it's the direct complement to Theorem 3 above. The only reason for the head jerk is that halting plays two roles in this theorem: 1) it's a condition on `haltp` and 2) it's a property of the computation `tm(args)`. These two meanings become entangled.

Finally, we consider a more general theorem for classes of computation that permit a universal machine to exist. All we know is that there is a set, S, whose members are representable in the field. Further we are given specific $p \in S$ and $q \notin S$. The natural question along these lines is whether there is a decider, that given a machine, m, and arguments, a, for it can determine if $m(a) \in S$. Given this set up, the result is a decided no! This will be our second reuse of an old proof:

Theorem 5 (Rice Krispies)**.** Assume that a computational class, defined by a representation language, provides 1) the "universal" capability to pass functional arguments as well as argument strings and 2) the capability to combine functions. Assume also, there is a set, S, a collection of objects of the sort manipulated within this computation class with a definite object $p \in S$ and another definite object $q \notin S$. Then whether or not the predicate $x \in S$ is effectively computable, there is no universal decider that, given any m and a can decide if $m(a) \in S$.

Proof. Figure 4 shows the usual construction that has been used for both the halting theorem

```
FUNC $h(f,a)$  $f(a) \in S$;

FUNC $h'(f,a)$ IF $h(f,a)$ THEN $q$ ELSE $p$;

FUNC $\hat{h}(f)$  $h'(f,f)$;

TEST $\hat{h}(\hat{h})$
```

Figure 4: Rice Krispies counterexample.

and the BM constancy theorem. If one traces the test case, precisely the same loop exhibited in Figure 3 is evident. Further, unravelling the return values develops the following, punch line in an appropriate form.

$$\begin{array}{l} \hat{h}(\hat{h}) \text{ returns } q \text{ if and only if } \hat{h}(\hat{h}) \in S. \\ \hat{h}(\hat{h}) \text{ returns } p \text{ if and only if } \hat{h}(\hat{h}) \notin S. \end{array}$$

Therefore, even if $x \in S$ is effectively computable, the decider is simply not a decider. $\square$

This Theorem is just a variation of one called Rice's Theorem [5]. That theorem also shows that there are restrictions, for TM, on deciding facts about behavior including halting. The only slight extension of Rice's result, herein, is applicability to other classes of computation such as BM. However, I will concede that while interesting to me, there appears little interest in such classes. Rather, attention is focused more on programming languages which are more or less Turing complete (up to memory limitations) and easy of accomplishing important implementations with them.

References

[1] Church, A. (1941)*The Calculi of Lambda-Conversion* (ISBN 978-0-691-08394-0).

[2] Davis, M. (1958) *Computability and Unsolvability* McGraw-Hill Book Company.

[3] Davis, M. (1965) *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions* Dover Publications, Inc.

[4] Linz, P. (2019) *An Introduction to Formal Languages and Automata, Sixth Edition* Jones and Bartlet Learning, an Ascend Learning Company.

[5] Rice, H. G. (1953) Classes of Recursively Enumerable Sets and Their Decision Problems, *Transactions of the American Mathematical Society*, v74 pp 358–366.

[6] Turing, A.M. (1936) On computable numbers, with an application to the Entscheidungsproblem, *Proceeding of the London Mathematical Society* Ser. 2, Vol. 42 (pp 230–265).

A Syntax of the Boolean Machine Language

The syntax specification for the class of BM is specified here using a form of BNF. The use of special characters in the metalanguage is described in Figure 5. Figure 6 presents the syntax of the BM language using these conventions. The character classes, `<alpha>` and

`<...>`	Content is a metavariable name
{...}	Parentheses in metalanguage
[...]	Contents are optional
`|`	Separates alternatives
`$`	Following metaterm will appear 0 or more times
`:=`	Metavariable on left defined by meta expression on right

Figure 5: Metalanguage syntax forms.

`<numeric>`, are the usual. I have used upper case for symbols that are part of the language and lower case for meta terms and user-supplied language fragments throughout to improve

```
      <run> ::= (RUN ${<bm-def>|<test>})
   <bm-def> ::= (DEFINE <bm-name> <motor>)
     <test> ::= <subst>
    <motor> ::= ({λ|LAMBDA} <var-list> <bexp>)
 <var-list> ::= ($<var> [!REST <rest-var>])
     <bexp> ::= <constant>|<bvar>|<lform>|<satp>|<subst>
 <constant> ::= !TRUE | !FALSE
    <lform> ::= <and>|<or>|<not>
      <and> ::= (AND $<bexp>)
       <or> ::= (OR $<bexp>)
      <not> ::= (NOT <bexp>)
     <satp> ::= (SATP <bexp>)
    <subst> ::= (APPLY <mexp> <arg-list>)
 <arg-list> ::= $<arg> {<rest-var>|()}
      <arg> ::= <bexp>|<mexp>
     <mexp> ::= <bm-ref>|<mvar>|<motor>
   <bm-ref> ::= (BM <bm-name>)
  <bm-name> ::= <sym>
 <rest-var> ::= <sym>
      <var> ::= <bvar>|<mvar>
     <bvar> ::= <sym>
     <mvar> ::= <sym>
      <sym> ::= <alpha>${<alpha>|<numeric>|-}
```

Figure 6: BM syntax specification.

readability. However, the defined language is intended to be case insensitive and include accented characters for expositions. Note, the following appendix will describe the context sensitive and "semantic" aspects of the language that the syntax specification cannot easily capture. Table 1 gives a suggestive phrase for each metavariable to help understand the

Table 1: Intended meaning of metavariables.

Variable	Intention	Variable	Intention
`<run>`	A complete test case.	`<satp>`	Is arg satisfiable?
`<bm-def>`	Define a named function.	`<subst>`	Apply function to args.
`<test>`	Top-level `<subst>`.	`<arg-list>`	Arguments to function.
`<motor>`	Function definition.	`<arg>`	An argument.
`<var-list>`	A function's variables.	`<mexp>`	Function valued form.
`<var>`	A variable.	`<bm-ref>`	Reference to a BM.
`<bexp>`	Boolean-valued expression.	`<bm-name>`	Name of BM.
`<constant>`	Boolean constant.	`<rest-var>`	Variable with list of values.
`<lform>`	A logical form.	`<bvar>`	Boolean variable.
`<and>`	Form with `and` operator.	`<mvar>`	Function valued variable.
`<or>`	Form with `or` operator.	`<sym>`	A user-supplied name.
`<not>`	Form with `not` operator.		

intended connotation of its name.

It will be obvious to many readers that the syntax defined is very Lisp like. All non-atomic expressions are bound by parentheses and the expression operator[6] is always the first element. What I am calling a motor is just a lambda-like form that one might encounter in all old Lisps and most modern implementations as well as the Church Lambda Calculus [1]. Note, however, resemblance is superficial; the chosen syntax greatly simplifies development of supporting software. The use of lambda expressions will be discussed in detail below but a hint is given here:

A `<motor>` is the function part of a functional invocation: it binds variables and creates a context in which those variables are visible. A `<subst>` applies a `<motor>` to a list of values that are bound to its variables. Use of those variables within the scope of the `<motor>` (evaluation of the embedded `<bexp>`) causes substitution of the bound values.

The top-level syntactic construct is a `<run>`, a mixed sequence of `<bm-def>` and `<test>`, where the former are named definitions of BM and the latter are test cases. A `<bm-def>` names a `<motor>` that can be used as the function portion of a `<subst>`.

A `<motor>` is comprised of three parts: 1) an operator, λ or `lambda`, 2) a `<var-list>` that specifies the number of parameters that are expected and the local names given the values passed, and 3) a Boolean expression, `<bexp>`, that is evaluated in the context of the variables bound by this `<motor>` and any other containing `<motor>` as explained in the next section. The evaluation must produce a `<bexp>` value.

The rest of the syntax specification is straightforward if one recalls that 1) Lisp is basically a Polish prefix language where forms are enclosed in balanced parentheses with the form operator first in the list and 2) that the BM language is claimed to use Lisp syntax.

[6]What is called a function or operator herein means a piece of computation defined in a programming language. There is no appeal to set theory, et al, involved in this definition.

B Language Rules

This section makes some of the nonsyntactic definitions and constraints more explicit. In addition, the way a BM is applied to arguments for evaluation is elaborated. The discussion is broken up into smallish blocks that are numbered. Some are labeled "Rule" and others "Observation." Sometimes it wont be clear why a particular label was chosen. Therefore, blocks are numbered using the same counter to make cross referencing easier. The basic topics to be discussed below include types of objects, variable types, context of evaluation rules, and how `<subst>` apply their "functional" part to given arguments. The explanations of these topics are interdependent (also somewhat circular) so some topics start and resume later. Appendix C is an implementation consistent with these rules.

Rule 1 (Name restrictions)**.** Two `<var>` with the same name (a `<sym>`) should not appear in the same `<var-list>` and `nil` is not a `<sym>`. A `<sym>` with two consecutive "`-`" characters is reserved for a special purpose: `h--x` should only explicitly appear in the `<bexp>` of a `<bm-def>` named `h`. This is a way to guarantee a private name. No two `<bm-def>` should have the same name in a single `<run>`.

Observation 2. (Evaluation) The term *evaluation* is used, herein, in the same sense as an interpreter (or code output by a compiler) executing a language form to produce a value and/or side effect. Table 2 summarizes evaluation for each language form.

Table 2: Brief form evaluation descriptions.

Atom	**Evaluation Description**
Constant	Idempotent.*
Variable	Retrieve bound value or assume free.*
Operator	**Form Evaluation**
`RUN`	Evaluate embedded arguments left to right.
`DEFINE`	Extend top-level context with closure.
λ, `LAMBDA`	Close form in current context.*
`AND, OR, NOT`	Replace subforms with their evaluation; optional optimizations.*
`BM-REF`	Retrieve closure, type `bm`, from context.*
`SATP`	Is argument satisfiable?*
`APPLY`	Eval `<motor>`, eval `<arg-list>` forms, extend context with `<var-list>` entries bound to argument evals, eval `<bexp>` from `<motor>`.*
* Return the value retrieved or produced to parent form (caller).	

Observation 3 (Context objects and references)**.** At all points during the evaluation of elements in a `<run>`, there is an object called the *current context*, aka, the *context of evaluation.* That context provides access to the definitions of `<bm-def>` when they are referenced as well as the value of variables that are visible in the current context. The pair of a name associated with its value is called a *binding* of that name.

Observation 3.1 (Context object types)**.** The context is a list of context objects. There are three types: `bm`, `var`, and `rst`. When a name is used to reference an object, the object's type is checked and its value must be as expected; else, the program is not well formed. The syntax tries to indicate when a Boolean, motor, or rest is appropriate but a context free grammar can not express such constraints adequately. The rules are clarified here.

Rule 3.2 (Context objects are evaluated)**.** Any object on a context list has been *evaluated* and these values are never reevaluated. When a `<sym>` is used as a key to search a context list, its context type is already constrained. So a `<bm-ref>` searches for an object of the given name of context type `bm`; a `<rest-var>` in a `<subst>` form `<var-list>`, searches for an object of context type `rst`; and a `<mvar>` or `<bvar>` reference (a `<var>` appearing anywhere other than in a `<var-list>`) searches for a context object of type `var`.

Rule 3.3 (Failed search 1)**.** If the search for a `bm` or `rst` context object finds nothing with the given name, an error is signalled.

Rule 3.4 (Failed search 2)**.** Failing to find a `var` context object with a given name is an error or not *depending on how that variable is used.* If the variable is used as a `<bvar>` (it is part of a Boolean expression) and it is not found, it appears as it's own value, i.e., it is simply a free variable with the given name. On the other hand, a search for a `<mvar>` that fails, is an error. The value of a `<mvar>` must evaluated to a closure—see Rule 5.

Rule 3.5 (Evaluate only once)**.** Any evaluation satisfied by searching for and returning a value from an existing binding will never be evaluated again. In particular, any further appearances of a free variable will remain free. Further, an evaluation of a `<motor>` is just the production of a closure. The embedded `<bexp>` will be evaluated every time the `<motor>` is applied since the evaluation context, an environmental variable, is not fully formed until arguments are evaluated and bound.

Rule 4 (Order of evaluation)**.** After a form operator is noted and evaluated if necessary, arguments are evaluated left to right. As part of an argument's evaluation, its subforms are evaluated and so on. Then the form operator (or the `<motor>` in a `<subst>`) is applied to its evaluated arguments.

Rule 4.1 (Exception to evaluation order)**.** When a form such as `<and>` or `<or>` is evaluated, it is permissible to stop evaluation as soon as the form's value is determined. For example, when the form `(OR a b c)` is evaluated: assume the value of `a` is indeterminate but the value of `b` is `!TRUE`. In this case the evaluator may, as an optimization, immediately determine the value of the disjunction to be `!TRUE` and omit the evaluation of `c`. All optimizations of this sort are allowed but not required.[7]

Rule 5 (Closures)**.** A closure is the result when a `<motor>`-defining expression is evaluated, where the *closure* is a pair of the `<motor>` and the context in which its definition was found. When that closure is applied to arguments, its `<bexp>` body "sees" locally bound `<var>` as well as objects bound in the original context of the `<motor>`, etc.

[7]N.B. These optimizations can increase the class of language instances that might be computable. Consider evaluating the form `(OR a b)`, where `a` evaluates as `!TRUE` and `b` is an ill-formed expression whose evaluation would signal an error. This evaluation strategy is a portion of what has been called *evaluate by need* or *lazy evaluation*.

Rule 6 (Name duplication)**.** When the current context list is search, the most recently added binding, if any, with the specified name and type is found.

Rule 7 (Top-level context)**.** When a `<run>` form is evaluated, the current context is set to the empty list. When a `<bm-def>` is encountered, the embedded `<motor>` is made a closure in the current context. The current context is then extended with that closure as a `bm` object with the `<bm-name>`. This has the effect that a `<bm-def>` cannot reference itself by name. When a test form is evaluated, it is evaluated in the current context. Thus, a `<bm-def>` or `<test>` form can only explicitly reference, by name, previously defined `<bm-def>`.

Rule 8 (`<subst>` evaluation)**.** It is time to complete the description of how a *closure* is applied to its passed argument values when `<subst>` are evaluated. Assume `m` is the motor part of the closure (the evaluation of the first argument to `apply`) and that $\mathtt{a}_1 \ldots \mathtt{a}_n$`[()|r]` is the `<arg-list>` that will be evaluated to produce the values passed to `m`.

Rule 8.1 (Closing `m`)**.** When a `<subst>` is evaluated, the first thing done is to determine the context of `m`. If `m` is specified by a `<bm-ref>`, the value retrieved from the current context list is a closure. If `m` is specified by a `<mvar>`, i.e., one that binds a closure as value, then that closure is used. So in these cases, a closure is found. The other possibility is that `m` is an explicit `<motor>`: in this case it is simply closed in the current context.

Rule 8.2 (Evaluating `<arg-list>` items)**.** The next step is to evaluate the arguments in the current context: First, the $\mathtt{a}_1 \ldots \mathtt{a}_n$ are evaluated in order and the values are collected into the list $\mathtt{e} = (\mathtt{e}_1 \ldots \mathtt{e}_n)$; Second, if `r` is given it *must* reference a bound `rst` in the current context in which cases its value, a list of evaluations, is retrieved and appended to the end of list `e`. If `()` appears instead of an `r`, no appending is necessary as () is the empty list isn't it? N.B. further processing described next can not determine nor is it relevant if, for example, e_3 was passed as the evaluation of $\mathtt{a}_3$ or was an item in a retrieved rest list.

Rule 8.3 (The variable list)**.** Let $\mathtt{v}_1 \ldots \mathtt{v}_s$ be the names of the ordinary variables in the `<var-list>` and note that the same variable name may not appear twice. If there is a `<rest-var>`, let $\mathtt{v}_{s+1}$ be its name.

Rule 8.4 (Number of argument constraints)**.** At this point we have a closure for the motor and arguments that have been evaluated. The argument values when substituted are not subject to further substitution or evaluation; in a sense, they have transitioned from "type" program text to "type" value. We now must pair the values with the appropriate variable and create bindings. The first rule is that s, the number of normal variables, must equal the length of `e`, if `!rest` is not in the `<var-list>`; if `!rest` does appear, the length of `e` must be at least s and could be greater.

Rule 8.5 (Initiate creation of new bindings)**.** The value of the current context pointer is saved—it will be restored when processing of this `<subst>` is completed. The current context pointer is now set to the one in the closure of `m`.

Rule 8.6 (Making new bindings)**.** Each $\mathtt{v}_i$ is paired with e_i, where $1 \leq i \leq s$, and is tacked on to the front of the current context list as a `var` binding. If there is a `<rest-var>`, a list of the remaining unconsumed elements of e are paired with $\mathtt{v}_{s+1}$ and that context object, a `rst`, is tacked onto the current context. It is possible that it will be the empty list. Note that the program type of all the ordinary variables that were just bound are determined by what their value is: `<mvar>` if value is a closure and `<bvar>` otherwise. N.B. The order in which the variable bindings are created and joined to the new context is unimportant since there shall be no duplicate names bound by a single `<subst>`.

Rule 8.7 (Apply closure to arguments)**.** Next, the `<bexp>` is retrieved from the `<motor>` in the closure, evaluated in the newly created context, the value is captured, and the context restored to the one that existed when processing of the `<subst>` commenced. Finally, the captured value is returned.

Observation 9 (These rules are effective)**.** There is no difficulty, theoretical or practical, implementing an interpreter for this language that captures and signals all violations of the above stated rules. See Appendix C for an example of such an implementation.

Observation 10 (Raison d'être `<rest-var>`)**.** The rationale for `<rest-var>` in this language is simply so that one can write a `<subst>` that will apply a `<mexp>` to zero or more arguments without the coder of the `<subst>` knowing intimate details of the `<mexp>`. Said another way: The inclusion of `<rest-arg>` make writing a "Universal BM" trivial.

C Implementation

The following is an implementation of the above described facility written in Common Lisp. This code has not been tested, it is merely typeset using a LaTeX editor as part of the preparation of this document.

```
(defmacro RUN (&rest runitems)
  ;; Input and execute a run test without needing quotes.
  `(dorun ',runitems))

(defconstant true '!TRUE "Representation of Boolean true")
(defconstant false '!FALSE "Representation of Boolean false")

;; A ctx is an object in the lexical context chain with objects
;; linked thru the nxt slot. The types are :bm, :var, and :rst. nm is the
;; name of the object. The val of a :var is simply its value evaluated
;; in the context where it was bound. The :val of a :bm (top-level)
;; or explicit in a <subst>) is a cls struct that specifies the motor
;; and the lexical ctx where it was defined and will execute.
(defstruct ctx typ nm val nxt)
(defstruct cls mtr ctx)

(defun add2ctx (typ name value ctx)
  ;; Adds a binding to ctx and returns the new context object.
  (make-ctx :typ typ :nm name :val value :nxt ctx))

(defun findctx (typ nm ctx &optional err)
  ;; Search for and return a ctx object with specified type and name in the
  ;; given ctx. If not found signal an error if err, otherwise return nil.
  (loop unless ctx
          do (if err (error "~s ~a not found" typ nm)
                 (return nil))
        when (and (eq typ (ctx-typ ctx)) (eq nm (ctx-nm ctx)))
          do return ctx
        do (setq ctx (ctx-nxt ctx))))

(defun findbm (nm ctx &optional err)
  ;; Search for a bm in ctx. Signal an error if not found as err.
  (findctx :bm nm ctx err))

(defun findvar (nm ctx &optional err)
  ;; Search for a var in ctx. Signal an error if not found as err.
  (findctx :var nm ctx err))

(defun findrst (nm ctx &optional err)
  ;; Search for a rst variable.
  (findctx :rst nm ctx err))
```

```
(defun dorun (runners &aux (ctx nil))
  ;; Deal with items, one at a time, in a run list. ctx is initially nil.
  ;; An items is added for each bm definition processed.
  (loop for r in runners
        do (case (car r)
             (DEFINE (setq ctx (eat-bm r ctx)))
             (APPLY (eat-tst r ctx))
             (otherwise (error "~a found in run" r))))
  (format t "~%Run sequence completed. Congratulations!~%")
  (values))

(defun eat-bm (r ctx)
  ;; A BM definition from run. We first close it in the current top-level
  ;; context then add the new closure to that context and return it.
  (destructuring-bind (DEFINE nm mtr) r
    (declare (ignore DEFINE))
    (when (findbm nm ctx)
       (error "~a is duplicate bm name" nm))
    (prog1 (add2ctx :bm nm (make-cls :mtr mtr :ctx ctx) ctx)
           (format t "~%(bm ~a) defined~%~%" nm))))

(defun eat-tst (r ctx)
  ;; A test form is in run. We execute it to produce a Boolean then
  ;; print the form and its value. This is done as two prints so the
  ;; test form will be seen even if exeb blows.
  (format t "~%Test form ~a;~%" r)
  (format t "~a~%~%" (exe-subst r ctx)))

(defun exeb (bexp ctx)
  ;; Evaluate bexp in context ctx. Some value simplification will be done
  ;; but the amount isn't guaranteed. Cases will be syntactically separated
  ;; and farmed out to individual form handlers.
  (typecase bexp
    (symbol
      ;; The possibilities are !TRUE, !FALSE, and a variable. In the latter
      ;; case we search context for a definition. If nothing is found, the
      ;; name itself is the value. It is an error, however, if the symbol is
      ;; a rest var. If a bound value is a closure, there is an error - it
      ;; can't be used as a bexp; i.e., if mvar si, bvar no.
      (cond((eq bexp true) true)
           ((eq bexp false) false)
           ((null bexp) (error "nil or () is not bexp"))
           (t (let((vc (findvar bexp ctx nil)))
                (cond(vc (if (typep (ctx-val vc) 'cls)
                             (error "Motor value of ~a is not bexp" bexp)
```

```
                              (ctx-val vc)))
                      ((findrst bexp ctx t)
                       (error "~a is rstvar, not a bexp" bexp))
                      ;; If not bound, var or rstvar, it is its own value.
                      (t bexp))))))
    (cons
      ;; Sort the cases out as a function of the form operator.
      (case (car bexp)
            (AND (exe-and bexp ctx))
            (OR (exe-or bexp ctx))
            (NOT (exe-not bexp ctx))
            (SATP (exe-satp bexp ctx))
            (APPLY (exe-subst bexp ctx))
            (otherwise (error "~a isn't a bexp" bexp))))
    (t (error "~a is not a bexp" bexp))))

(defun exe-and (bexp ctx)
  ;; Process the embedded bexp and do a mild amount of optimization.
  (let((iargs (cdr bexp)) (oargs nil))
    (loop
      (cond((null iargs)
            (setq oargs (remove-duplicates oargs :test #'equal))
            (return (cond((null oargs) true)
                         ((null (cdr oargs)) (car oargs))
                         (t (cons 'AND (reverse oargs))))))
          (t (let((v (exeb (pop iargs) ctx)))
               (cond((eq v false) (return false))
                    ((eq v true))
                    (t (push v oargs)))))))))

(defun exe-or (bexp ctx)
  ;; Process the embedded bexp and do a mild amount of optimization.
  (let((iargs (cdr bexp)) (oargs nil))
    (loop
      (cond((null iargs)
            (setq oargs (remove-duplicates oargs :test #'equal))
            (return (cond((null oargs) false)
                         ((null (cdr oargs)) (car oargs))
                         (t (cons 'OR (reverse oargs))))))
          (t (let((v (exeb (pop iargs) ctx)))
               (cond((eq v true) (return true))
                    ((eq v false))
                    (t (push v oargs)))))))))

(defun exe-not (bexp ctx)
  (let((bdy (exeb (second bexp) ctx)))
    (cond((eq bdy true) false)
         ((eq bdy false) true)
         ((and (consp bdy) (eq (car bdy) 'NOT)) (cadr bdy))
         (t (list 'NOT bdy)))))

(defun exe-satp (bexp ctx &aux (bx (exeb (second bexp) ctx)))
  ;; After exeb finishes its work, we will have a form that consists of free
  ;; variables, Boolean constants, and forms with operators and, or, or not.
  ;; Our task is to see if any assignment of true/false to the variables
  ;; makes the form true. Our value is true or false as the form is
  ;; satisfiable or not. We start by evaluating the form then running
  ;; through it to compiling a list of variables in it. By the way,
  ;; exponential time complexity only hurts for a little while; or so I
  ;; am told.
  (let((vlist nil))
    (labels((findvs (x)
              ;; Finds all variables in x and adds them to vlist but only
              ;; once.
              (typecase x
                (symbol (unless (or (eq x true) (eq x false))
                          (pushnew (list x nil) vlist :key #'car)))
                (cons (loop v in (cdr x) do (findvs v)))))

            (search (vl)
              ;; vl is vlist or one of its tails. We assign a value, false
              ;; then true, to the value at the head of the list and recur
              ;; on the tail of vl for each value. When we have a complete
              ;; assignment, we test for true evaluation with eve.
              (cond((null vl) (when (eq (eve bx) true)
                                (return from exe-satp true)))
                   (t (setf (cadar vl) false)
                      (search (cdr vl))
                      (setf (cadar vl) true)
                      (search (cdr vl)))))

            (eve (x)
              ;; Mini eval for simple Boolean expressions. Variable values
              ;; maintained in vlist.
              (cond((eq x true) true)
                   ((eq x false) false)
                   ((symbolp x) (cadr (find x vlist :key #'car)))
                   (t (case (car x)
                        (NOT (if (eq (eve (cadr x)) true) false true))
                        (AND (loop for a in (cdr x)
                                   when (eq (eve a) false)
                                     return false
                                   finally return true))
                        (OR (loop for a in (cdr x)
                                  when (eq (eve a) true)
                                    return true
                                  finally return false))
                        (otherwise (error "~a found in satp!" x)))))))

      ;; When completed, vlist is a list of all variables, no duplicates.
      (findvs bx)
      ;; Try all combinations of true/false assignments to variables.
      (search findvs)
      ;; If we get here, no value assignments to the variables makes bexp
      ;; true so we report failure.
      false)))

(defun exe-subst (subst ctx)
  ;; This is where we deal with subst forms - the application of a lambda
  ;; form to arguments.
  (destructuring-bind (APPLY mexp &rest exps) subst
    (declare (ignore APPLY))
    ;; The first task is to close the mexp form if it isn't already. That
    ;; task is performed by exem. The resulting closure context, ectx, is
    ;; context where the lambda body, bexp, is eventually executed. vlist is
    ;; the lambda's variables list. eae will end up being a list of the
    ;; passed arguments evaluated and in original order.
    (let*((cls (exem mexp ctx)) (ectx (cls-ctx cls)) (mtr (cls-mtr cls))
          (vlist (cadr mtr)) (bexp (caddr mtr))
          ;; The loop evaluates the expressions passed as arguments in this
          ;; subst. The only hocus pocus is that the last argument is either
          ;; () or a reference to a rest variable bound in ctx. When all is
          ;; complete, we'll have the evaluated expressions and the rest
          ;; variables lists combined.
          (eae (loop for (exp . tail) on exps
                     unless tail
                       return (if exp (append eae
                                              (ctx-val (findrst exp ctx t)))
                                    eae)
                     collect (evea exp ctx) into eae)))
      ;; If the lambda has no arguments, then none can be passed.
      (if (null vlist) (when eae (error "Too many arguments."))
          ;; Variable names are paired with values and a rest variable,if
          ;; any, gets whatever is leftover. These parings become context
          ;; objects that are tacked on to the front of ectx, the context
          ;; in which the subst body will actually execute.
          (loop for (var . tail) on vlist
                when (eq var '!REST)
                  do (setq ectx (make-ctx :typ :rst :name (car tail)
                                          :value eae :ctx ectx)
                           eae nil)
                     (loop-finish)
                unless eae
                  do (error "Too few arguments")
                do (setq ectx (make-ctx :typ :var :name var :value (pop eae)
                                        :ctx ectx))))
      (if eae (error "Too many arguments"))
      ;; bexp is the lambda body, a Boolean expression by rule, and ectx is
      ;; the context where the lambda was closed extended by the variable
      ;; bindings we just created. So it's execution time!
      (exeb bexp ectx))))

(defun exem (mexp ctx)
  ;; A motor is expected. If a closure is found, it is simply returned;
  ;; otherwise it should be an explicit motor and we close it in the context
  ;; represented by ctx.
  (typecase mexp
    ;; Just return a closure. Can we find one here??
    (cls mexp)
    ;; A symbol should be a variable bound to a closure. Let's find it. Note
    ;; we are baring nil as a variable.
    (symbol (or mexp (error "nil is not a mexp"))
            (let((val (ctx-val (findvar mexp ctx t))))
              ;; Search for var blows if not found. If found it must be
              ;; a closure.
              (if (typep val 'cls) val
                  (error "~a is not a motor" mexp))))
     ;; If mexp is a cons, the operator must be bm or lambda to end up
     ;; with a closure.
    (cons (case (car mexp)
            ;; If a bm-ref, we find the closure for the name or signal
            ;; an error.
            (BM (ctx-val (findbm (cadr mexp) ctx t)))
            ;; If it's an explicit motor, we close it in current context.
            ((LAMBDA λ) (make-cls :mtr mexp :ctx ctx))
            ;; Report programmer error.
            (t (error "~a is not a motor" mexp))))
    (t (error "~a is not a motor" mexp))))

(defun exeae (ae ctx &aux t1 t2)
  ;; ae is an argument expression that is to be evaluated for a subst form
  ;; where the context is ctx. The only hoop to jump here is that we have no
  ;; indication whether a Boolean or motor is expected. A rest value is not
  ;; legal at this point.
  (typecase ae
    (symbol (case ae
              (!TRUE !TRUE)
              (!FALSE !FALSE)
              (otherwise (if (setq t1 (findvar ae ctx nil))
                             (ctx-val t1) ae))))
    (cons (case (car ae)
            ((BM LAMBDA λ) (exem ae ctx))
            (otherwise (exeb ae ctx))))
    (otherwise (error "~a is illegal argument" ae))))

(defun exer (rexp ctx)
  ;; rexp must be the final argument that appears in the lisp of parameters
  ;; being passed to a motor. If it is (), we just return the empty lisp as
  ;; is appropriate. Otherwise we search for a rest list with the given name
  ;; in ctx. Since it's easy to do, we will signal and error if appropriate.
  ;; If we return at all, we return a list of processed expressions.
  (if (null rexp) () (ctx-val (find-rst rexp ctx t))))
```

D Three Valued Halting Predicate

Recent, circa 2022, discussions in the USENET forum, comp.theory, have entertained the idea that cases like the example lifted from the HT proof in Linz are some how *pathological*, and should be excluded from HT's requirement to decide halting status. In other words, the idea is to make the halting predicate three valued: `H` halting, `L` looping, and `P` pathological. All of this without defining pathological! Let's assume the `H`, `L`, and `P` portions of the domain are each nonempty, that `H` and `L` are disjoint, and that `P` contains some cases that would, according to our prior understandings, be classified `H` and other cases `L`. Figure 7 is a code sketch that follows the usual pattern to setup a contradiction to our assumptions. The usual

```
FUNCTION h(tm,a)
  ;; Values are H, L, or P.
  RETURN correct value for tm(a)
  END h;

FUNCTION h′(tm,a)
  ;; Inverts the meaning of H and L.
  SELECT h(tm,a)
    WHEN H THEN loop:GOTO loop;
    WHEN P THEN RETURN H;
    WHEN L THEN RETURN H;
    END h′;

FUNCTION ĥ(tm)
  RETURN h′(tm,tm);
  END ĥ;

Evaluate ĥ(ĥ);
```

Figure 7: Three-valued halt predicate counterexample.

trace of the example of ĥ applied to ĥ unearths that contradiction. So there is no solvable three-valued halt problem either.

```
If tm(a) halts Then h′(tm,a) loops forever;
If tm(a) is pathological Then h′(tm,a) halts (returning H);
If tm(a) loops forever Then h′(tm,a) halts (returning H).

If tm(tm) halts Then ĥ(tm) loops forever;
If tm(tm) is pathological Then ĥ(tm) halts (returning H);
If tm(tm) loops forever Then ĥ(tm) halts (returning H).

If ĥ(ĥ) halts Then ĥ(ĥ) loops forever;
If ĥ(ĥ) is pathological Then ĥ(ĥ) halts (returning H);
If ĥ(ĥ) loops forever Then ĥ(ĥ) halts (returning H).
```