Preface

Research is a cumulative enterprise — new results are based on earlier ones. Analysing properties for classes of systems, for example, and deriving algorithms for these purposes, the frontiers of knowledge can be pushed forward. Where is the limit? If enough effort is invested, can these frontiers be pushed indefinitely? Proceeding in the traditional way, one is tempted to assume that there exist no boundaries, no *qualitative leaps* can be reached.

It seems that most surprising results can be found when different kinds of paradigms are combined: It turns out that *computability theory* offers nice ways to attack the fundamental system theoretic problems in a fresh way. One of the most powerful concepts available in computability theory is that of *Gödelian undecidability* — all frameworks that are powerful enough are either *incomplete* or *inconsistent.* And, as it turns out, some classes of systems constitute such frameworks; stability of such systems, for example, cannot be exhaustively solved.

This report studies a specific nonlinear system structure and shows that for this class of systems the undecidability issues become acute. Tools are introduced to utilize this framework, so that systems with special properties can be determined, and after that, different kinds of "killer systems" can be constructed.

A concrete "compiler" for transforming rather high-level descriptions written in a specialized programming language into a discrete-time dynamic state-space system of the form $x(k+1) = g(Ax(k))$ is presented. This compiler has been implemented in `Python` script language and it is available (together with this report, and some sample program files) in Internet at

`http://saato.hut.fi/hyotyniemi/publications/02_report133/`.

I am grateful to Mr. Lauri Kovanen, who acted as a Summer trainee at the Control Engineering Laboratory of HUT in 2002: He implemented the compilation from the $\mathcal{L}$++ language into `Matlab` matrix form using `Python` scripts; he also tested and debugged the codes. I am also grateful to Professors Vincent Blondel and Pekka Orponen for discussions and for their kind interest.

Keywords: Computability, undecidability, dynamic systems, stability

Contents

Chapter 1

Introduction

The behavior of linear systems is exactly known: No qualitative surprises are faced no matter what is the initial state and no matter what is the dimension of the system. On the other hand, if the system is nonlinear, it seems that everything can collapse. It has been proven that very complicated behaviors can emerge, for example, in nonlinear continuous-time systems (see [13]), in hybrid systems ([2]), and even in cellular automata ([16]). What this complexity means — briefly, it can be argued that there exist such systems that will *for ever* defy analysis attempts, no matter how the theories will proceed (these issues will be elaborated on later in this report).

In what follows, we will concentrate on discrete-time, autonomous nonlinear state-space systems of the general form

$$s(k+1) = g(s(k)). \tag{1.1}$$

Of course, complicated functions g can result in complicated behaviors; for example, for sinusoidal function form having an infinite number of changes in the derivative sign, extraordinary behavioral complexity can be reached even if s were scalar (see [11]). On the other hand, when the dimension of the state vector s increases, the behavior can become qualitatively more and more complicated even for simple-looking (vector-valued) nonlinearities (see [1]). Dimensional complexity can be used as a substitute for structural complexity.

Defining the system as a combination of linear and nonlinear parts, so that

$$s(k+1) = g(As(k)), \tag{1.2}$$

where the nonlinearity g is defined elementwise, meaning that

$$g_i(s) = g(s_i), \tag{1.3}$$

one can interpret the resulting system as a recurrent, discrete-time neural network structure, with matrix A determining the "synaptic weights", and g being the "activation function" of the neurons. Because of the current interest on neural networks, the capabilities of such systems have gained special emphasis; it has been shown (see [14] and [15]) that for sigmoidal activation functions various kinds of Gödelian undecidability results apply. Also for piecewise linear (saturating) nonlinearities the same holds.

In this report, a special nonlinear discrete-time dynamic system structure will be concentrated on:

$$s(k+1) = f_{\text{cut}}(As(k)). \tag{1.4}$$

Here the "generalized cut" function $f_{\text{cut}} : \mathcal{R}^{\dim(s)} \to \mathcal{R}_+^{\dim(s)}$ is defined elementwise as[1]

$$f_{\text{cut},i}(s) = \begin{cases} s_i, & \text{if } s_i > 0, \text{ and} \\ 0, & \text{otherwise,} \end{cases} \tag{1.5}$$

for all $i = 1, \ldots, \dim(s)$. This means that for positive values the nonlinearity vanishes, whereas for negative values the values themselves vanish. Even though the nonlinearity is, in a sense, the simplest possible, just cutting off the negative entries, the behavior of the class of such systems is extremely complex, as will be studied later.

There exists an infinite number of different types of nonlinearities — why should one be specially interested in the cut function (1.5)?

One motivation is the extreme simplicity of the nonlinearity — but it also turns out that such *positive systems* with strictly non-negative state variables can efficiently be used as frameworks for natural phenomena. Typically, in the theory one concentrates on positive *linear* systems [5], but, as shown in [9] and [10], the explicit nonlinearity makes it also possible to study applications to *cognitive systems* in the *complex systems* framework.

Earlier studies concerning the system structure (1.4) were presented in [6] and [7]; now a more detailed discussion is presented.

[1] A note on notations: Subindices refer to vector elements, or in the case of matrices, the first index refers to rows and the second one to columns

Chapter 2

Computability with Systems

Running a computer program is a dynamic process; however, in the computer logic the state transitions are determined in very complex ways. From the dynamic systems point of view, such processes typically have too loose structure to be truly interesting. There are exceptions — some computers can be determined in an extremely fixed, rigid framework: It turns out that the simple general system structure (1.4) can emulate the operation of a programmable computer. To recognize the connections between theoretical computer science and dynamic systems analysis, some computability theory is first needed.

2.1 About computability

According to the basic axiom of computability theory, all computable functions can be implemented using a *Turing machine.* There are various ways to realize a Turing machine; it can be shown that if a function is Turing computable, it can be coded using the language $\mathcal{L}$ that only includes four operations, being defined in the Backus-Naur (BNF) form as follows:

`Program`	::=	`Commands`*
`Commands`	::=	*LineNumber* INC `VarName`
	::=	*LineNumber* DEC `VarName`
	::=	*LineNumber* GOTO *LineNumber*
	::=	*LineNumber* IF `VarName` = 0 SKIP 1.

This means that one only needs increment and decrement commands, branching capability and a simple conditional structure; all algorithms can be implemented in that framework when the basic structures are repeated sufficiently. The variables above can only have positive integer values.

The above extremely simple language $\mathcal{L}$ can be made more practical if some additional functionality is included. So, define the extended language $\mathcal{L}$++ as follows:

`Program`	::=	`Variables`* `Commands`*	
`Variables`	::=	*LineNumber* `VarDef`	
`VarDef`	::=	*VarName* = *Constant*	% Initial value
`Commands`	::=	*LineNumber* `Operations`	
	::=	*LineNumber* IF `Condition`*	
		LineNumber THEN `Operations`	
		LineNumber ELSE `Operations`	
`Condition`	::=	*VarName* = 0	% Test
`Operations`	::=	`Modifications`* `Jump`	
`Modifications`	::=	*VarName* ADD *Constant*	% Increment
	::=	*VarName* SUB *Constant*	% Decrement
`Jump`	::=	GOTO *LineNumber*	% Branch

Above, *Constant* $\in \mathcal{N}$ can be any positive value; *VarName* is a character string. All variables are assumed to have non-negative integer values; if an operation would make the variable value negative, zero value is used instead. It is assumed that all lines in the code must have a distinct line number; the line numbers must be successive integers starting from 1. Note that a conditional branch always exhausts *three lines,* so that the line number counter is incremented by three (the reason for this peculiarity is seen later). The Kleene stars "*" mean that there can be an arbitrary number of corresponding constructs.

It can be shown that a the above language $\mathcal{L}$++ is in one-to-many correspondence with the dynamic system structure (1.4).

2.2 The "Turing System"

The state transition matrix A in (1.4) can be defined so that any program in language $\mathcal{L}$++ can be presented as a discrete-time process. Running a program means that the process (1.4) is iterated until the state no more changes, or until a fixed state is found — the final variable values, or the calculation results, can be seen in the resulting state vector. The proof of this is very similar to that presented in [7], and instead of the actual proof, a procedure for implementing the language $\mathcal{L}$++ in the form (1.4) is presented.

To implement a "compiler" for the language $\mathcal{L}$++, an extension of the basic BNF syntax is used, so that the *semantics* of the parsed constructs are simultaneously defined. First, the source code is matched against patterns in the RCL style using depth-first search; operations (indented lines) are executed in order of appearance, immediately after the preceding subsequences

of source code have been parsed. Because of the simple syntactical structure of the language $\mathcal{L}$++, there exist no problems with backtracking.

Before the compilation procedure is started, the data structures are initialized. The dimensions of A and s are dependent of the program complexity, so that there is a separate entry in the "snapshot vector" s corresponding to every program line, and A is defined to have compatible size. All elements in A and $s(0)$ are initially set to zero. In addition the these, there are internal variables that are needed to take care of appropriate control flow in the compiler. Below, subindices refer to matrix (vector) rows and columns, respectively.

`Program` ::= `Variables`* `Commands`*
  If program successfully parsed, then:
    Set $s_{EntryPoint}(0) = 1$

`Commands` ::= `LineNumber Operations`
    For all variables v:
      Set $A_{\#v,CurrentLine} = Vars_{\#v}$
  ::= `LineNumber` "`IF`" `Condition`*
  If recognized, then:
    Set $A_{CurrentLine+1,CurrentLine} = 1$
    Set $A_{CurrentLine+2,CurrentLine} = 1$
  `LineNumber Then-block`
  `LineNumber Else-block`
    For all variables v:
      Set $A_{\#v,CurrentLine-1} = Then_{\#v} - Else_{\#v}$
      Set $A_{\#v,CurrentLine} = Else_{\#v}$

`Operations` ::= `Modifications`* `Jump`
    Set $A_{Jump,CurrentLine} = 1$

`Modifications` ::= $v \in Strings$ "`ADD`" $c \in \mathcal{Z}$
  If recognized, then:
    Set $Vars_{\#v} = Vars_{\#v} + c$
  ::= $v \in Strings$ "`SUB`" $c \in \mathcal{Z}$
  If recognized, then:
    Set $Vars_{\#v} = Vars_{\#v} - c$

`Jump` ::= "`GOTO`" $i \in \mathcal{N}$
  If recognized, then:
    Set $Jump = i$
  ::= "" % If ignored, go to the next line
    Set $Jump = CurrentLine + 1$

`Condition` ::= $v \in Strings$ "$= 0$"
If recognized, then:
Set $A_{CurrentLine+1,\#v} = -1$

`Then-block` ::= "`THEN` `Operations`"
Set $Then = Vars$

`Else-block` ::= "`ELSE` `Operations`"
Set $Else = Vars$
Set $A_{Jump,CurrentLine-1} = A_{Jump,CurrentLine-1} - 1$

`LineNumber` ::= $i \in \mathcal{N}$
If recognized, then:
Set $CurrentLine = i$
For all variables v:
Set $Vars_{\#v} = 0$

`Variables` ::= `LineNumber VarDef`
Set $EntryPoint = CurrentLine + 1$

`VarDef` ::= $v \in Strings$ "$=$" $c \in \mathcal{N}$
If recognized, then:
Define $\#v = CurrentLine$
Set $A_{\#v,\#v} = 1$
Set $s_{\#v}(0) = c$.

To achieve context dependent features in the compilation process, there are some additional compilation-time integer-valued variables ($LineNumber$, $EntryPoint$, $Jump$) and vecors having entries for each of the variables defined in the program ($Vars$, $Then$, and $Else$).

Earlier version of the compiler was presented in [8]; however, the original formulation of the compiler was flawed in that respect that correct operation of the IF–THEN–ELSE structure could be guaranteed only if there existed only one reference to a single variable within a THEN–ELSE structure; the new formulation has no such limitations. On the other hand, the original formulation produced "elegant" matrices having only 1, -1, and 0 valued elements; generally, this cannot be guaranteed any more. The new formulation tends to produce "denser" matrix structures, that is, there are fewer zero entries in the matrix A.

2.3 Sketch of proof

It is the state vector s that contains the program snapshot, including variable values and the program counter, and matrix A determines how this snapshot is changed. First, the basic structure representing a program row, $A_{n+1,n} = 1$, transfers the "program counter" from row n to row $n+1$. On the other hand, looking at the implementation of the variable definitions, $A_{\#V,\#V} = 1$, it is evident that basically one has a discrete-time integrator:

$$s_V(k+1) = s_V(k).$$

References to the variable may change the integrator contents: for example, $A_{\#V,n} = v$ means that v is added to the variable contents if there happens to be 1 in s_n. Subtractions operate like negative additions; because of the cut function, negative values become zeros.

It is evident that various additions can be combined and carried out simultaneously; further, a branch can also be included in the same structure, so that can be accomplished parallel during one step. Indeed, the operation of a combined IF–THEN–ELSE block with various simultaneous manipulations can be integrated. Assume that one has the following code in the program:

n: `IF` $W_1 = 0$ $W_2 = 0$ ...
$n+1$: `THEN` V_1 `ADD` t_1 V_2 `ADD` t_2 ... `GOTO` m_T
$n+2$: `ELSE` V_1 `ADD` e_1 V_2 `ADD` e_2 ... `GOTO` m_E,

and assume that originally the snapshot contains the following entries:

$$\begin{cases} s_n(k) = 1 \\ s_{n+1}(k) = 0 \\ s_{n+2}(k) = 0 \\ \\ s_{W_1}(k) = w_1 \in \{0,1,2,...\} \\ s_{W_2}(k) = w_2 \in \{0,1,2,...\} \\ \vdots \\ \\ s_{V_1}(k) = v_1 \in \{0,1,2,...\} \\ s_{V_2}(k) = v_2 \in \{0,1,2,...\} \\ \vdots \\ \\ s_{m_T}(k) = 0 \\ s_{m_E}(k) = 0. \end{cases}$$

After the next step, the distribution of the values will be

$$\left\{\begin{array}{rcl} s_n(k+1) &=& 0 \\ s_{n+1}(k+1) &=& f_{\text{cut}}\,(1-w_1-w_2-\cdots) \\ s_{n+2}(k+1) &=& 1 \\ \\ s_{W_1}(k+1) &=& w_1 \\ s_{W_2}(k+1) &=& w_2 \\ &\vdots& \\ \\ s_{V_1}(k+1) &=& v_1 \\ s_{V_2}(k+1) &=& v_2 \\ &\vdots& \\ \\ s_{m_T}(k+1) &=& 0 \\ s_{m_E}(k+1) &=& 0, \end{array}\right.$$

and after the next step, there holds

$$\left\{\begin{array}{rcl} s_n(k+2) &=& 0 \\ s_{n+1}(k+2) &=& 0 \\ s_{n+2}(k+2) &=& 0 \\ \\ s_{W_1}(k+2) &=& w_1 \\ s_{W_2}(k+2) &=& w_2 \\ &\vdots& \\ \\ s_{V_1}(k+2) &=& f_{\text{cut}}\,(v_1 + f_{\text{cut}}\,(1-w_1-w_2-\cdots)\cdot(t_1-e_1)+e_1) \\ &=& \left\{\begin{array}{ll} f_{\text{cut}}(v_1+t_1), & \text{if } w_1=w_2=\cdots=0 \\ f_{\text{cut}}(v_1+e_1), & \text{otherwise} \end{array}\right. \\ s_{V_2}(k+2) &=& f_{\text{cut}}\,(v_2 + f_{\text{cut}}\,(1-w_1-w_2-\cdots)\cdot(t_2-e_2)+e_2) \\ &=& \left\{\begin{array}{ll} f_{\text{cut}}(v_2+t_2), & \text{if } w_1=w_2=\cdots=0 \\ f_{\text{cut}}(v_2+e_2), & \text{otherwise} \end{array}\right. \\ &\vdots& \\ \\ s_{m_T}(k+2) &=& f_{\text{cut}}\,(f_{\text{cut}}\,(1-w_1-w_2-\cdots)) \\ &=& \left\{\begin{array}{ll} 1, & \text{if } w_1=w_2=\cdots=0 \\ 0, & \text{otherwise} \end{array}\right. \\ s_{m_E}(k+2) &=& f_{\text{cut}}\,(1-f_{\text{cut}}\,(1-w_1-w_2-\cdots)) \\ &=& \left\{\begin{array}{ll} 0, & \text{if } w_1=w_2=\cdots=0 \\ 1, & \text{otherwise.} \end{array}\right. \end{array}\right.$$

This means that the manipulations have been carried out just as expected.

2.4 Implementation aspects

The "pseudocode compiler" above can be readily implemented in different environments; for example, a simple version for `Matlab` already exists, and a more polished `Python` version [17] has been constructed (indeed, `Perl`, etc., would do just as well; however, general-purpose programming languages like `C`, etc., are unnecessarily heavy tools for this). Some general implementation guidelines are given below.

Simplification of syntax. There are some pragmatic aspects to be taken care of to make programming easier for the end-user. First, it is not reasonable to ask the user to keep track of the actual line numbers — these can be automatically completed by the compiler. Correspondingly, branches with absolute addresses are extremely cumbersome in practice, and rather than asking the programmer to use absolute line numbers, some labeling mechanism is needed; for example, as an example of possible syntax,

```
Label:      ...
            GOTO Label
            ...
```

This means that the compilation becomes a two-step process: During the first step the symbol table is constructed and references resolved (and also the size of the data structures, the variable n, is determined), and only after that in the step two the final compilation takes place.

In addition to the mnemonic names, there are various minor enhancements that can make the life easier for the programmer. For example, the "empty" ELSE branches (if only the THEN portion is needed in the code) can be completed automatically by the compiler. Label `End` denotes termination; there is no need to implement such a row, `GOTO End` just means that the program counter vanishes and the program halts. The GOTO commands can be neglected; in such a case control flow continues from the next line.

Additionally, it is assumed that comments (preceded by "%") can be placed on any line; all characters after this are ignored on that line.

Macros. If the proposed language is really to be used for some practical task, yet another pragmatic shorthand can be introduced in the language: Because simple tasks (like copying a variable, or adding a value of a variable to another) consist of a large number of elementary operations, a mechanism like *macro expansion* might be useful. Assume that the code

```
            ...
            Add(Var1 Var2)
Continue:   ...
```

is mechanically substituted with the code

```
            ...
Add:        IF Var2 = 0
            THEN GOTO AddBack
            ELSE Var1 ADD 1 Var2 SUB 1 VarAdd ADD 1 GOTO Add
AddBack:    IF VarAdd = 0
            THEN GOTO Continue
            ELSE Var2 ADD 1 VarAdd SUB 1 GOTO AddBack
Continue:   ...
```

considerable simplifications in the code outlook can be reached. The macros are expanded during the preprocessing of the code (first pass of the compiler). Note that there may exist various invocations of the same macros, and the labels have to be made uniquely distinguishable in the compilation phase.

If there are hidden variables within a macro code, the rows implementing the variables are added *within the code:* This makes resolving the variables simpler during the first pass of the compilation, but it must be remembered that these lines are not executable and they are skipped by the program counter. Note that to make a macro re-entrant, the original values of the hidden variables need to be restored (in the above code `VarAdd` is automatically reset to 0 after execution). The next row number after the macro is resolved by the compiler; references to `End` in the macro are automatically substituted by this number. All these modifications are explicitly shown in the `.log` file. When implementing the macros, the only extension to the programming language syntax is

```
Commands            ...
                ::= LineNumber MacroName "(" v* ")" Jump

MacroName       ::= m ∈ Strings
```

that is, the macro invocation with the arguments listed between parentheses is given as a single operation on one line. The `Jump` can be omitted; but if it is included, references to `End` within the macro code are substituted by this label.

When a macro call has been detected in a code, the predefined macro folder is searched for a file with the macro name. The idea is that the macro files have exactly the same format as the main routines themselves — this means that the macros can be run also as stand-alone programs and debugged separately. The key point is that the variables having no initial values

defined in the code are interpreted as input parameters; on the main routine level, these variables are queried from the user, but within a macro, they are interpreted as input arguments (in the given order). Yet another point needs to be mentioned: If there are external `Matlab` commands within a file, these lines are collected in the set of all external commands; but these lines are only executed if the control is within the corresponding block. This means that one needs to keep track of the invocation order and test it before running the external codes (this means that macro externals are only executed while running that macro; but the main program externals are *not* executed there!).

Extensions of control structures.[1] There is no reason why the logical THEN and ELSE branches could not contain more than just one set of operations. Indeed, the only problem that emerges is that there is need to keep track of program flow, that is, jumping to the actual beginning of the ELSE branch, rather than simply to "two rows below". Nesting the IF – THEN – ELSE structures may introduce ambiguous structures; one way to circumvent these if some kind of ENDIF is introduced in the end of all conditional structures, but in that case the principle "one row in program, one dimension in matrix" has to be compromized. Another possibility is to impose indentations, etc., in an organized way: The originally ambiguous structure

```
IF ...              % Outer IF structure
THEN ...
ELSE ...
IF ...              % Outside the ELSE branch
THEN ...
ELSE ...
```

becomes unambiguous after indentation (the latter IF–THEN–ELSE being inside the outer ELSE branch):

```
IF ...              % Outer IF structure
THEN ...
ELSE ...
     IF ...         % Within the ELSE branch
     THEN ...
     ELSE ...
```

Note that the matrix-form evaluation of program structures makes it easy to extend the standard sequential control flow: Parallel, simultaneous processes

[1] These extensions have not yet been implemented in the current compiler

can be carried out at the same time. However, the program syntax is not extended here in that direction (special syntax is needed to define branching of the control flow, and merging of different flows).

Note that in the sieve code in [9], some syntactical extensions were utilized to reach lower-dimensional structures: There it is allowed to write conditions like "IF $A - B = 0$". Looking closer at the compiler it is easy to see how such an extension can be implemented — A and B just contribute in the opposite directions — but because it would be difficult to implement such a structure in a consistent always-working way, it is left out from the general syntax description.

About compilation and execution processes. The compiler reads an ASCII file containing the program code. An ASCII log file with the same name, but with extension "`.log`" can be created in the same folder; in this file the program code is printed, as seen by the compiler — that is, the "completed" program code with explicit line numbers is listed — and the warnings and errors are printed next to the lines where they were detected. The compilation results are also written into an ASCII file with the same name, but with extension "`.m`". This means that `Matlab`-form data structures are constructed, so that they can readily be loaded into the `Matlab` environment. The produced `Matlab` file consists of the following parts:

1. The definition of the matrix A (because of its sparsity, it is reasonable to represent this as `A = zeros(n,n); A(i,j) = c`, etc.).

2. *Some kind of* user interface for determining the initial values that are omitted in the code, and a mechanism for constructing the vector $s(0)$ out of them.

3. Implementation of the loop $s(k+1) = g(As(k))$ and appropriate monitoring mechanisms for maintaining the computing process (see below).

Normally, the iteration continues until convergence is reached, that is, until the program counter vanishes, and this should be detected by the system. But there are also other possible ways of running programs, and these should be supported: For example, as shown in [9], to implement "emergent phenomena", the process can continue infinitely, some external operation just being carried out if some special condition is fulfilled — say, a text is printed on the screen if some variable has some specific value, or if some line in program code has been reached.

This inclusion of extra code also offers possibilities for implementing easy debugging facilities. For example, the `Matlab` command `keyboard` within a function temporarily transfers the command to outside, letting the user test the values of the variables, etc.

The achieve this extended functionality, the program syntax has to be slightly augmented:

`Program` ::= `Variables`* `Commands`* `Externals`*

...

`Externals` ::= *Any* `Matlab` *code.*

In the external `Matlab` code, references to symbols (line labels or variables) are possible; the appropriate data structures are substituted. The `Matlab` code needs to be parsed through by the compiler; if references to program variables or labels are detected, they are substituted with the references to appropriate vector s elements. It seems to be reasonable to define yet another variable that is available within `Matlab`: Variable *ProgramCounter* contains the number of the line where the control currently is (so that constructs like "`if ProgramCounter == <label> ...`" are possible). These external operations are carried out before **each** program iteration.

Chapter 3

Universality and Beyond

Universal machines are Turing machines that are capable of *emulating* other Turing machines, that is, if the program code and the inputs are given, the same final state and outputs will be found. It needs to be noted, however, that the dynamics is typically very different: Interpreting other programs is a time-consuming process.

Applying the tools developed in the previous chapter, one can construct a compiler for the language $\mathcal{L}$ using the language $\mathcal{L}$++. In what follws, this kind of compiler construction is first carried out, and, after that, this compiler is applied for implementing algorithms with pathological decidability properties. Because of the correspondence between the algorithms and systems of the form (1.4), the same undecidability phenomena apply also to such systems.

3.1 Coding the programs

To make it possible for algorithms to "speak about each other" that is, to construct algorithms that can analyze other algorithms the program structures need to be collapsed — without losing their information content!

A unique, compact representation for programs presented in $\mathcal{L}$ needs to be determined, so that the program (and the variable values) can be coded as a *single number* for handy manipulation of programs. This process of enumerating the programs is called *Gödel numbering,* and, based on the properties of primes, the Gödel number G of a program can be constructed of parts, so that the total Gödel number is the product of the components, $G = G_1 \cdot \cdots G_n$. First, let us study the representation of variables:

- For each variable `V`, introduce a unique odd prime, say, $p_{\mathtt{V}}$.
- If `V` has value v, the contribution of this variable is $G_{\mathtt{V}} = p_{\mathtt{V}}^v$.

Second, let us study the representation of program lines:

- For each line `L`, introduce a unique odd prime, say, $p_{\texttt{L}}$.
- Depending on the contents of the line, apply some of the following:
  - If the line reads `INC V`, the contribution of the line is $G_{\texttt{L}} = p_{\texttt{L}}^{p_{\texttt{V}}}$.
  - If the line reads `DEC V`, the contribution of the line is $G_{\texttt{L}} = p_{\texttt{L}}^{2 \cdot p_{\texttt{V}}}$.
  - If the line reads `IF V=0 SKIP 1`, the contribution of the line is $G_{\texttt{L}} = p_{\texttt{L}}^{2^2 \cdot p_{\texttt{V}}}$.
  - If the line reads `GOTO N`, the contribution of the line is $G_{\texttt{L}} = p_{\texttt{L}}^{2^3 \cdot p_{\texttt{N}}}$.

Now, given some Gödel number G, the program can be restored by calculating how many times a specific prime is included in G, that is, how many times the prime divides the Gödel number. For simplicity, in what follows, it is assumed that successive primes are applied in order of appearance; with no loss of benerality, it is also assumed that there is only one output `Y` (so that $p_{\texttt{Y}} = 3$) and one input `X` (so that $p_{\texttt{X}} = 5$). This means that the first executable program line corresponds to $p_{\texttt{L}} = 7$. Below, some examples of Gödel numbering are presented; the numbers typically grow extremely fast!

- Implementation of "`INC Y`":

  ```
  Y = 0
  X = x
  INC Y
  ```

  This means that the Gödel number is $3^0 \cdot 5^x \cdot 7^3$; at its simplest, defining the (dummy) $x = 0$, one has 343.

- Implementation of "`INC Y / INC Y`":

  ```
  Y = 0
  X = x
  INC Y
  INC Y
  ```

  This means that the Gödel number becomes $3^0 \cdot 5^x \cdot 7^3 \cdot 11^3$; at its simplest, having (dummy) $x = 0$, one receives 456533.

- Implementation of *signum,* or "`Y = sign`(x)"

  ```
  Y = 0
  X = x
  IF X = 0 SKIP 1
  INC Y
  ```

  This means that the Gödel number becomes $3^0 \cdot 5^x \cdot 7^{2^2 \cdot 5} \cdot 11^3$,
  or $106203506442121573331 \cdot 5^x$.

Applying the above notations, not all numbers G represent valid programs; for example, using the above coding, no even Gödel numbers can exist.

3.2 Interpreting the codes

Next, a straightforward implementation of the universal machine based on the above Gödel numbering is presented. Below, the algorithm is presented as pseudocode; in Appendix B, an explicit formulation is presented using the language $\mathcal{L}$++, and employing the help functions given in Appendix A:

Variables:	`Y` = Resulting output of the interpreted program `X` = Gödel number of the program to be interpreted (with initial values) `PC` = Program counter in the interpreted code `ROW` = Code for the current row
`Start:`	Set `ROW` = How many times `PC` divides `X` If no times, end of program If `ROW` cannot be divided by 2 at all, go to `Add` If `ROW` can be divided by 2 exactly once, go to `Sub` If `ROW` can be divided by 2 exactly twice, go to `Skip` Else go to `Goto`
`Add:`	Multiply `X` by the remaining value of `ROW` Change `PC` to point to the next line Return to `Start`
`Sub:`	If divisible, divide `X` by `ROW` Change `PC` to point to the next line Return to `Start`
`Skip:`	If `X` is divisible by `ROW` add `PC` by two Otherwise change `PC` to point to the next line Return to `Start`
`Goto:`	Move remaining value of `ROW` to `PC` Return to `Start`

The behavior of the universal machine, given some simple programs, is illustrated below:

- Assume that the input to the universal machine is

  $\mathtt{X}_{\text{univ}} = 343 = 3^0 5^0 7^3$;
  this means "`INC Y`" with `Y` = **0** and `X` = 0 (`PC` = 7).

  After 53431 steps (see Fig. 3.1) this converges in

  $\mathtt{X}_{\text{univ}} = 1029 = 3^1 5^0 7^3$;
  this means "`INC Y`" with `Y` = **1** and `X` = 0 (`PC` = 11).

  Interpretation: Starting from `Y` = 0 the program `INC Y` halts in a state where `Y` = 1.

- Assume that the input to the universal machine is

  $\texttt{X}_{\texttt{univ}} = 1029 = 3^1 5^0 7^3$;
  this means "`INC Y`" with `Y` = **1** and `X` = 0 (`PC` = 7).

  After 160929 steps this converges in

  $\texttt{X}_{\texttt{univ}} = 3087 = 3^2 5^0 7^3$;
  this means "`INC Y`" with `Y` = **2** and `X` = 0 (`PC` = 11).

  Interpretation: Starting from `Y` = 1 the program `INC Y` halts in a state where `Y` = 2.

- Assume that the input to the universal machine is

  $\texttt{X}_{\texttt{univ}} = 117649 = 3^0 5^0 7^{2\cdot 3}$;
  this means "`DEC Y`" with `Y` = **0** and `X` = 0 (`PC` = 7).

  After 7520543 steps this converges in

  $\texttt{X}_{\texttt{univ}} = 117649 = 3^0 5^0 7^{2\cdot 3}$;
  this means "`DEC Y`" with `Y` = **0** and `X` = 0 (`PC` = 11).

  Interpretation: Starting from `Y` = 0 the program `DEC Y` finally halts with no changes.

- Assume that the input to the universal machine is

  $\texttt{X}_{\texttt{univ}} = 352947 = 3^1 5^0 7^{2\cdot 3}$;
  this means "`DEC Y`" with `Y` = **1** and `X` = 0 (`PC` = 7).

  After 18971679 steps this converges in

  $\texttt{X}_{\texttt{univ}} = 117649 = 3^0 5^0 7^{2\cdot 3}$;
  this means "`DEC Y`" with `Y` = **0** and `X` = 0 (`PC` = 11).

  Interpretation: Starting from `Y` = 1 the program `DEC Y` finally halts in a state where `Y` = 0.

3.3 Eternal mysteries

As shown by Gödel, all frameworks that are powerful enough are either *incomplete* or *inconsistent.* One way to put this is that in such environments one can implement the *liar's paradox* (expressed in some technically sound way). It will now be shown that in the case of dynamic systems, the paradox takes the form *the system is not stable if and only if it can be shown to be stable.* The only possibility is that *there can never exist a method for determining the stability for such systems.*

Assume that the system structure (1.4) can be exhaustively analyzed by some procedure or algorithm G, and assume that this algorithm returns "1" if some property applies, and "0" otherwise. The idea in the paradox construction is to tailor the system and its input so that, for example, it halts (converges) if and only if the algorithm says it would not:

```
Halt:       Universal(Y,G(Halt))
            IF Y = 0
            THEN GOTO End
            ELSE GOTO Halt.
```

Indeed, to implement the paradox, two additional simplifications are available:

1. The algorithm only needs to analyze one single fixed system.
2. The algorithm only needs to analyze this single fixed system for one single fixed input; this input is the Gödel number of this algorithm itself[1].

In each case, a logical contradiction is found: For example, the system remains bounded only if the algorithm says it does not, and vice versa — the only possibility is that such an analysis algorithm cannot exist.

It has been known that such system exist, and they have been constructed, but it has been estimated that such systems would have dimension somewhere around 1000 (actually, this has been constructed for a saturating nonlinearity) — but, as shown here, this estimate is too high at least for f_{cut}. It turns ot that the mortality problem "Given an arbitrary initial state, does the system converge to origin or not" is undecidable for a 338×338 dimensional system matrix A (see Appendix C and Fig. 3.2). Some 10% of the dimension could rather easily be dropped if more optimized (but fragile) program structures were introduced in the language $\mathcal{L}$++. The understandability of the universal machine implementation could also be compromized to bring the dimension down. In addition to this, some explicit extra states are now introduced in the codes only to keep the structure of A simple; in this implementation, only three alternative values (-1, 0, and 1) are employed in the matrix.

One can still strengthen the result: There does not only exist a single pathological system, but there exists an infinite number of counterexamples. To prove this, one lemma is needed:

> *The state vectors s can be freely scaled by some (positive) factor α; this does not qualitatively alter the behavior of the system.*

[1] Of course, this number, even though it is fixed, is not known before the algorithm is implemented; this means that, in practice, one has to define a general algorithm, capable of analysing all inputs for the given system, and only after this general algorithm is implemented, the input can be fixed to attack the simpler problem — this means that Simplification #2 cannot really be utilized in the presented way. However, the key point is that no inputs are any more needed, and the problem of "infinite recess" (having as input the code with the same input) can be avoided

Proof. Assume that a system is defined as $s(k+1) = f_{\text{cut}}(As(k))$ for some A and state sequence $s(k)$. If one defines another vectors as $s_\alpha = \alpha \cdot s$, it turns out that

$$\begin{aligned} f_{\text{cut}}(As_\alpha(k)) &= f_{\text{cut}}(\alpha \cdot As(k)) \\ &= \alpha \cdot f_{\text{cut}}(As(k)) \\ &= s_\alpha(k+1). \end{aligned} \tag{3.1}$$

It turns out that during each state transition, the parameter α simply traverses through the formula, no matter how many times the formula is iterated; this means that, finally, it is the end state of the original process that is multiplied by α when the modified process is iterated. The states can thus be freely scaled — still the number of steps exhausted by the system dynamics remains intact.

The above result means that if one is capable of finding one state $s(0)$ for which some undecidability result applies, one can instantly determine an infinite number of other initial states $s_\alpha(0)$ for which the same result applies (if $\alpha \neq 1$, no exact match with the original algorithm form any more exists).

3.4 "Star Systems"

It is a well-known fact that the state-space representations of systems are not unique; interchanging of state variables does not alter the system dynamics, if all references to these variables are also interchanged. This means that the visual outlook of the system matrix A can be altered by permuting its rows and columns (so that horizontal permutations are always followed by corresponding vertical permutations).

Note that in the "prime sieve" structure in [9] the rows and columns are shuffled to reach more homogeneous looking matrix structure; this means that, after constructing the $n \times n$ dimensional matrix A, and the n dimensional vector $s(0)$, some permutation of the numbers from 1 to n is determined, and the rows in $s(0)$ and rows and columns in A are reordered correspondingly. It turns out that the dynamics of the process remains intact also after this modification.

Because of the nature of the control flow in the original program, some of the state variables are more tightly coupled to other variables than other ones are in A; the state variables corresponding to normal program lines typically are loosely coupled, whereas state variables standing for variables in the algorithm are often referred to from other locations. This means that there typically exist a few state variables inducing "dense" horizontal and vertical lines in A. To alter the overall outlook of the matrix, these lines can be relocated.

For example, one can collect the horizontal and vertical lines in the center of the matrix, and distribute the other dots along the matrix diagonals (of course, this cannot be accomplished exactly; some kind of iterative optimization is needed). This kind of rearrangement means that some kind of "star-like" system form emerges (see 3.3).

All programs can be transformed to this kind of star form — for complex programs there typically exist various more or less good-looking alternatives. Each dot is essential, otherwise the star collapses (into a "neuron star"?)!

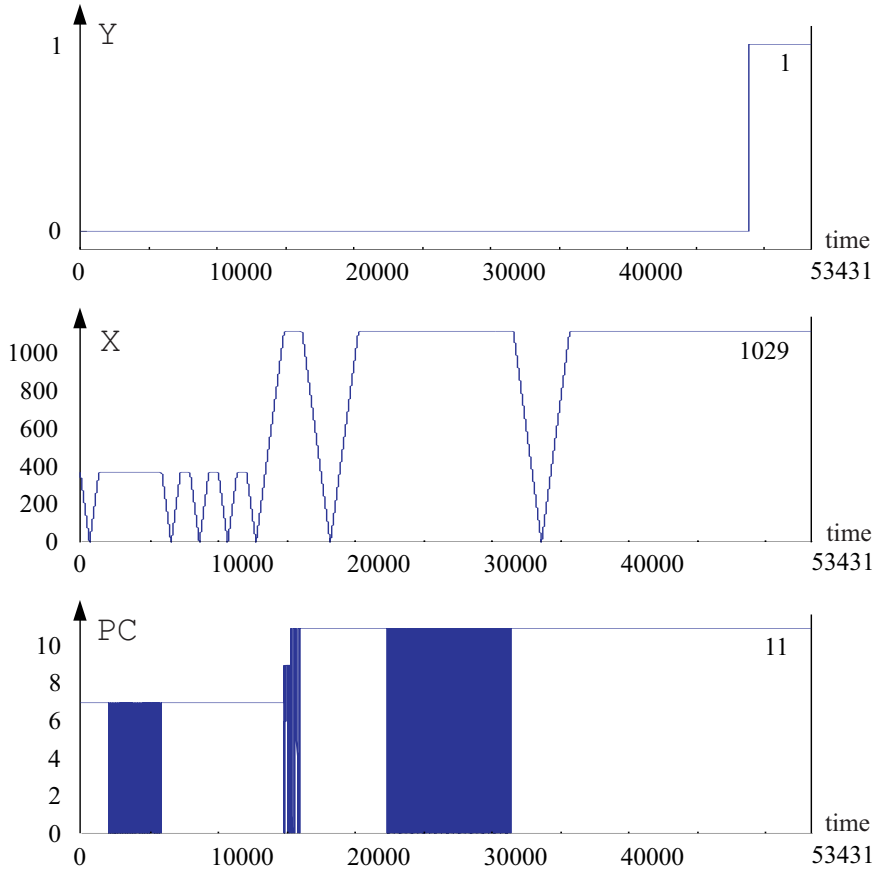

Figure 3.1: Behavior of the system emulating the operation "`INC Y`". The universal machine starting from the state where $X = 343 = 7^3$, it finally ends up in state where $X = 1029 = 3^1 \cdot 7^3$, meaning that variable Y has been incremented from 0 to 1. Seen from a distance, the behaviors of the variables resemble the operation of registers in real digital computers. Even this trivial operation takes a long time, over 50000 steps

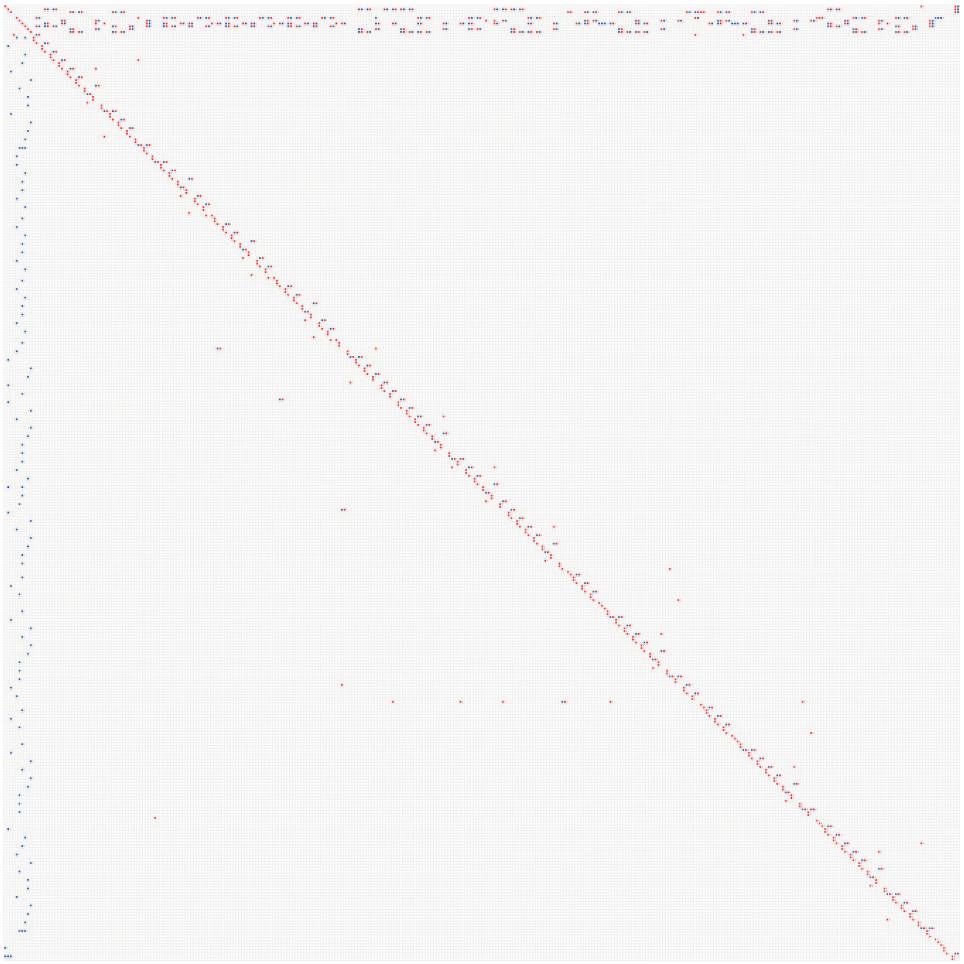

Figure 3.2: The matrix A representing the system corresponding to the `Bomb3` algorithm (see Appendix C). There exist only numbers -1, 0, and 1 in the above system matrix; these values are illustrated so that entries with -1 are shown in blue color, entries with 1 in red, whereas entries with 0 are left blanc

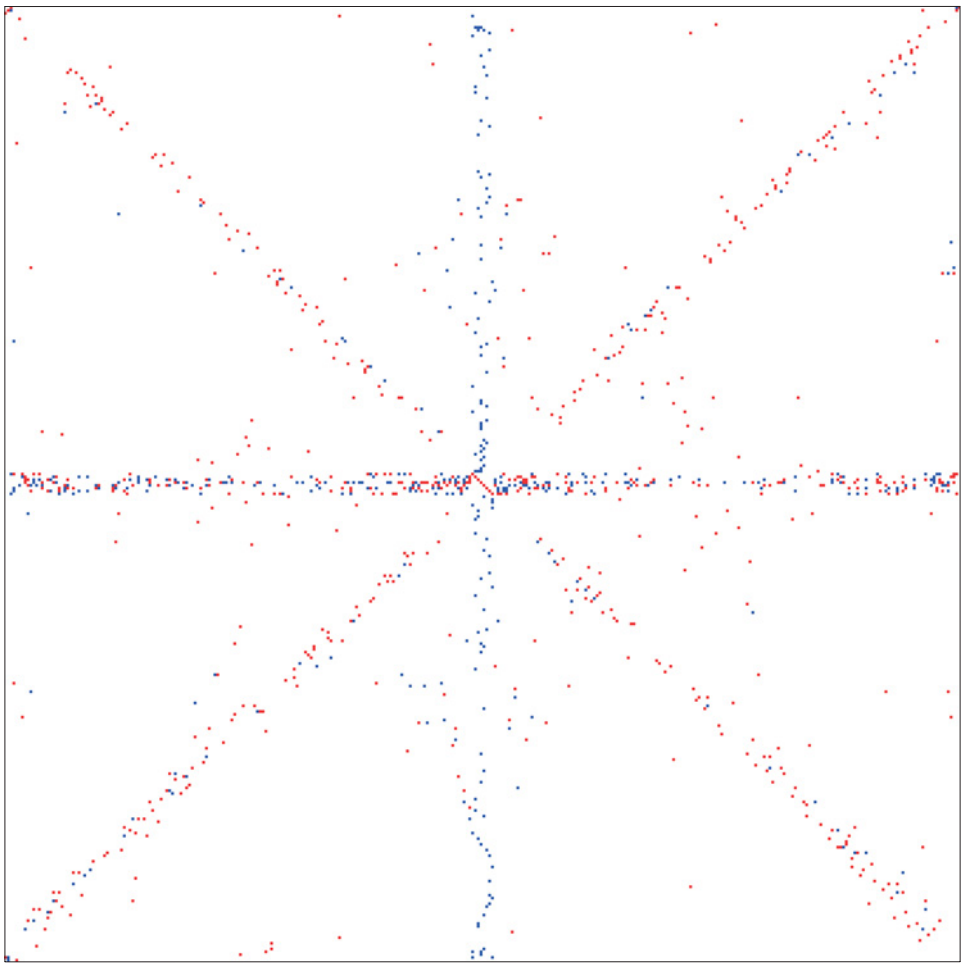

Figure 3.3: The system of Fig. 3.2 when the rows and columns have been appropriately permuted, thus revealing a "star form" representation A^* (one of various alternatives) of the original `Bomb3` system. The state variable s_2 standing for input x has been changed to s^*_{166}; s_3, or the contents of PC (originally 7) has been moved to s^*_{173}; and s_4, or the universal machine program counter has been moved to s^*_{286}, so that $s^*_{286}(0) = 1$. All other entries in the "star state" s^* are originally zeros

Bibliography

[1] Blondel, V.D. and Tsitsiklis, J.N.: A survey of computational complexity results in systems and control. *Automatica,* Volume 36, Issue 9, September 2000, pp. 1249–1274.

[2] Branicky, M.S.: Universal Computation and Other Capabilities of Hybrid and Continuous Dynamical Systems. *Theoretical Computer Science,* Vol. 138, No. 1, 1995 (Special issue on Hybrid Systems).

[3] Davis, M. and Weyuker, E.: *Computability, Complexity, and Languages—Fundamentals of Theoretical Computer Science.* Academic Press, New York, 1983.

[4] DeCarlo, R.A., Branicky, M.S., Pettersson, S., and Lennartson, B.: Perspectives and Results on the Stability and Stabilizability of Hybrid Systems. *Proceedings of the IEEE,* Vol. 88, No. 7, July 2000, pp. 1069–1082.

[5] Farina, L. and Rinaldi, S.: *Positive Linear Systems: Theory and Applications.* John Wiley & Sons, New York, 2000.

[6] Hyötyniemi, H.: Turing Machines are Recurrent Neural Networks. *STeP'96—Genes, Nets and Symbols,* Proceedings of the Finnish Artificial Intelligence Conference, Vaasa, Finland, August 20–23, 1996, pp. 13–24.

[7] Hyötyniemi, H.: On Unsolvability of Nonlinear System Stability. In the *Proceedings of the European Control Conference (ECC'97),* Brussels, Belgium, July 1–5, 1997.

[8] Hyötyniemi, H.: *Mental Imagery — Unified Framework for Associative Representations.* Helsinki University of Technology, Control Engineering Laboratory, Report 111, August 1998.

[9] Hyötyniemi, H.: *Complex Systems — Searching for Gold.* Arpakannus 2/2002, special issue on Complex Systems, pp. 29–34.

[10] Hyötyniemi, H.: *Studies on Emergence and Cognition — Parts 1 & 2.* Finnish Artificial Intelligence Conference (STeP'02), December 16–17, 2002, Oulu, Finland.

[11] Koiran, P. and Moore, C.: Closed-form Analytic Maps in One and Two Dimensions Can Simulate Universal Turing Machines. *Theoretical Computer Science,* 210 (1999), pp. 217–223.

[12] Moore, C.: Unpredictability and Undecidability in Dynamical Systems. *Physical Review Letters,* Vol. 64, No. 20, May 1990, pp. 2354–2357.

[13] Orponen, P.: A survey of continuous-time computation theory. In Du, D.-Z. and Ko, K.-I. (eds.): *Advances in Algorithms, Languages, and Complexity.* Kluwer Academic Publishers, Dordrecht, 1997, pp. 209–224.

[14] Siegelmann, H.T. and Sontag, E.: On the Computational Power of Neural Nets. *Journal of Computer and Systems Science,* Vol. 50, 1995, pp. 132–150.

[15] Recurrent neural networks: Some systems-theoretic aspects. In Karny, M., Warwick, K., and Kurkova, V. (eds.): *Dealing with Complexity: a Neural Network Approach.* Springer-Verlag, London, 1997, pp. 1–12.

[16] Wolfram, S.: *A New Kind of Science.* Wolfram Media, Champaign, Illinois, 2002.

[17] Information on the `Python` parser, the ready-to-install code and the manuals in electronic form are available at

http://www.python.org/.

Appendix A

Simple Samples

In what follows, some simple functions obeying the syntax defined in Chapter 1 are presented. These functions are used later as macros to implement the universal machine (see Appendix B), and, after that, the "Bomb Systems" (see Appendix C). There are a few points that need to be explained first:

- It needs to be noted that local variables are intentionally avoided in the functions (macros) to reach lower dimensional structures; the variables are assumed to be defined and initialized on the higher level and are passed to macros as parameters.
- Notations are kept consistent as far as possible; the function outputs are denoted normally as `Y` (or `Y1`, `Y2`, etc., in the multi-output cases), and inputs as `X`. The local variables are written in lowercase letters.
- For debugging purposes, the assumed variable bindings are shown as comments next to the variable definitions; that is, on the left hand side of the arrow it is shown what the value *must be* before the macro invocation, and on the right hand side it is shown what the value *will be* after the execution is completed.
- All of the functions are available in Internet, as well as the `Python` form compiler; after installation, they can readily be compiled and tested.

Zero

First, there are three simple functions to initialize variables, `Zero`, `Zero2`, and `Zero3`, zeroing one, two, and three variables, respectively.

```
            var Y         % y -> 0

Zero:       IF Y = 0
            THEN GOTO end
            ELSE Y SUB 1 GOTO Zero
```

Zero2

```
            var Y1        % y1 -> 0
            var Y2        % y2 -> 0

Zero2:      IF Y1 = 0 Y2 = 0
            THEN GOTO end
            ELSE Y1 SUB 1 Y2 SUB 1 GOTO Zero2
```

Zero3

```
            var Y1        % y1 -> 0
            var Y2        % y2 -> 0
            var Y3        % y3 -> 0

Zero3:      IF Y1 = 0 Y2 = 0 Y3 = 0
            THEN GOTO end
            ELSE Y1 SUB 1 Y2 SUB 1 Y3 SUB 1 GOTO Zero3
```

Move

Destructively move the contents of the variable `X` into the variable `Y`, zeroing the source. If the destination is not originally zero, the old and new contents are added together.

```
            var Y         % y -> y + x
            var X         % x -> 0

Move:       IF X = 0
            THEN GOTO end
            ELSE Y ADD 1 X SUB 1 GOTO Move
```

Copy

Copy the contents of the variable `X` into the variable `Y`, without (permanently) affecting the source. If the destination is not originally zero, the old and new contents are added together.

```
            var Y         % y -> y + x
            var X         % x -> x
            var a         % 0 -> 0

Copy:       IF X = 0
            THEN GOTO Reset
            ELSE X SUB 1 Y ADD 1 a ADD 1 GOTO Copy
Reset:      IF a = 0
            THEN GOTO end
            ELSE X ADD 1 a SUB 1 GOTO Reset
```

Subtract

Subtract two variables `X1` and `X2` from each other, simultaneously zeroing the latter. The result will be returned in `Y`. Of course, negative results are cut to zero.

```
            var Y        % 0 -> max(x1-x2,0)
            var X1       % x1 -> x1
            var X2       % x2 -> 0
            var a        % 0 -> 0

Subtract:   Copy(Y X1 a)
Begin:      IF X2 = 0
            THEN GOTO end
            ELSE Y SUB 1 X2 SUB 1 GOTO Begin
```

Mult

Multiply two integers `X1` and `X2`, simultaneously zeroing the latter. The result will be returned in `Y`.

```
            var Y        % 0 -> x1*x2
            var X1       % x1 -> x1
            var X2       % x2 -> 0
            var a        % 0 -> 0
            var b        % 0 -> 0

Mult:       IF X2 = 0
            THEN GOTO end
            Copy(a X1 b)
Back:       IF a = 0
            THEN X2 SUB 1 GOTO Mult
            ELSE Y ADD 1 a SUB 1 GOTO Back
```

Divisible

Test if the variable `X2` exactly divides `X1`. The result in `Y` will be zero if there is no remainder, otherwise `Y` is *something* else.

```
            var Y        % 0 -> 0 if divisible, otherwise > 0
            var X1       % x1 -> x1
            var X2       % x2 -> x2
            var a        % 0 -> 0
            var b        % 0 -> 0

Divisible:Copy(a X1 b)
Begin:    Copy(Y X2 b)
Back:     IF a = 0
          THEN GOTO end
          IF Y = 0
          THEN GOTO Begin
          ELSE a SUB 1 Y SUB 1 GOTO Back
```

RawDiv

Assuming that `X2` exactly divides `X1`, carry out the division.

```
            var Y        % 0 -> x1/x2 (assumed divisible)
            var X1       % x1 -> 0
            var X2       % x2 -> x2
            var a        % 0 -> 0
            var b        % 0 -> 0

RawDiv:   Copy(a X2 b)
Back:     IF X1 = 0 a = 0
          THEN Y ADD 1 GOTO end
          IF a = 0
          THEN Y ADD 1 GOTO RawDiv
          ELSE X1 SUB 1 a SUB 1 GOTO Back
```

Divide

Divide `X1` by `X2`, giving out the integer part in `Y` and the remainder in `REM`.

```
            var Y        % 0 -> int(x1/x2)
            var REM      % 0 -> rem(x1/x2)
            var X1       % x1 -> 0
            var X2       % x2 -> x2
            var a        % 0 -> 0
            var b        % 0 -> 0

Divide:     Copy(a X2 b)
Back:       IF X1 = 0
            THEN GOTO Reset
            IF a = 0
            THEN Y ADD 1 GOTO Divide
            ELSE X1 SUB 1 a SUB 1 GOTO Back
Reset:      IF a = 0
            THEN Y ADD 1 GOTO end
            Subtract(REM X2 a b)
```

DivBy2

Special function for implementing division by 2. Even numbers in `YX` are divided in place, whereas odd numbers are left intact. The remainder (0 or 1) is returned in `REM`.

```
            var REM      % 0 -> rem(x/2)
            var YX       % x -> x, or x/2 if divisible
            var a        % 0 -> ?

DivBy2:     Copy(REM YX a)
Back:       IF REM = 0
            THEN GOTO Even
            ELSE REM SUB 1
            IF REM = 0
            THEN GOTO Zero
            ELSE REM SUB 1 a ADD 1 GOTO Back
Even:       Zero(YX)
            Move(YX a) GOTO end
Zero:       REM ADD 1
```

CalcPower

Find out how many times `PC` can divide `X` without leaving a remainder. This function can be used to find out the prime decomposition of natural numbers; this is needed to resolve the unique Gödel numbering of programs (see Appendix B).

```
            var ROW       % 0 -> R
            var PC        % r -> r
            var X         % r^R*z -> r^R*z
            var a         % 0 -> 0
            var b         % 0 -> 0
            var c         % 0 -> 0
            var d         % 0 -> 0
            var e         % 0 -> 0

CalcPower:Copy(b X c)
Forward:  Move(a b)
          Divide(b c a PC d e)
          IF c = 0
          THEN ROW ADD 1 GOTO Forward
Reset:    IF a = 0 b = 0 c = 0
          THEN GOTO End
          ELSE a SUB 1 b SUB 1 c SUB 1 GOTO Reset
```

FindNextP

Find the next prime larger than a given (odd) prime in `P`; the result is calculated in place.

```
            var P        % p -> succp(p)
            var a        % 0 -> 0
            var b        % 0 -> 0
            var c        % 0 -> 0
            var d        % 0 -> 0

FindNextP:P ADD 1
            P ADD 1
            Zero(a)
            Copy(a P b)
Back:       a SUB 1
            a SUB 1
            a SUB 1
            IF a = 0
            THEN GOTO end
            ELSE a ADD 1
            Divisible(b P a c d)
            IF b = 0
            THEN GOTO FindNextP
            Zero(b) GOTO Back
```

Appendix B

Universal Machine

Below, a program written in the $\mathcal{L}$++ language that is capable of emulating any program written in the simpler $\mathcal{L}$ language is presented. The Gödel number of the program code, together with the original variable bindings (see Chapter 2), is given as the input `X`. The variable `PC` contains the current row number being executed (presented in the "prime form"; see Chapter 2). The (decoded) output of the program is finally given in `Y`, the final row is in `PC` and the program state is in `X`.

```
            var Y        ?  Output
            var X        ?  Input:  Program to be interpreted
            var PC       ?  Program counter
            var ROW = 0
            var A = 0    % Some additional local variables
            var B = 0
            var C = 0
            var D = 0
            var E = 0

Start:      Zero2(ROW C)
                         % Decode the current line:
            CalcPower(ROW PC X A B C D E)
            IF ROW = 0   % If row does not exist
            THEN GOTO Output

                         % Powers of 2 determine the command:
            DivBy2(B ROW C)
            IF B = 0
            THEN GOTO NoAdd
            ELSE B SUB 1 GOTO Add
```

```
NoAdd:    DivBy2(B ROW C)
          IF B = 0
          THEN GOTO NoSub
          ELSE B SUB 1 GOTO Sub
NoSub:    DivBy2(B ROW C)
          IF B = 0
          THEN GOTO Goto
          ELSE B SUB 1 GOTO Skip

                       % "Add one":
Add:      Mult(B X ROW D E)
          Zero(X)
          Move(X B) GOTO Next
                       % "Subtract one (if possible)":
Sub:      Divisible(B X ROW D E)
          IF B = 0
          THEN GOTO Forward
          Zero(B) GOTO Next
Forward:  RawDiv(A X ROW B D)
          Move(X A) GOTO Next
                       % "Skip one line":
Skip:     Divisible(B X ROW D E)
          IF B = 0
          THEN GOTO StepTwo
          ELSE GOTO Next
StepTwo:  FindNextP(PC B A D E) GOTO Next
                       % "Go to line":
Goto:     Zero(PC)
          Move(PC ROW) GOTO Start

                       % "Continue with next line":
Next:     FindNextP(PC B A D E) GOTO Start

                       % Finally, resolve the value of Y:
Output:   ROW ADD 1
          ROW ADD 1
          ROW ADD 1
          CalcPower(Y ROW X A B C D E)
          ROW SUB 1
          ROW SUB 1
          ROW SUB 1
```

Appendix C

Bomb Systems

Next the conclusion: Using the presented guidelines it is possible to construct fixed systems of the form (1.4) having predefined matrices A that will defy analysis attempts for ever. Below, some such unsolvability results are shown — and new ones could easily be found following similar ideas!

The input values x below are undefined while the system is being compiled; its value is dependent of the proposed analysis method. What is sure is that for any analysis algorithm, some x can be found so that the system beats the algorithm.

Unsolvability of boundedness

If there existed an algorithm that could tell for all initial states whether the system of the form (1.4) with the matrix A defined by `Bomb1` below would for ever remain bounded or not, there would exist an infinite number of initial states for which the system would diverge *if and only if* the algorithm would say it would *not* do that, and vice versa:

```
          VAR Y = 0
          VAR X = x    % Unsolvable for some x
          VAR PC = 7

Bomb1:    Universal(Y X PC)
          IF Y = 0
          THEN GOTO End
          ELSE GOTO ToInf
ToInf:    Y ADD 1 X ADD 1 PC ADD 1 GOTO ToInf
```

Unsolvability of convergence

If there existed an algorithm that could tell for all initial states whether the system of the form (1.4) with the matrix A defined by `Bomb2` below would finally converge to some constant state or not, there would exist an infinite number of initial states for which the system would converge *if and only if* the algorithm would say it would *not* do that, and vice versa:

```
          VAR Y = 0
          VAR X = x    % Unsolvable for some x
          VAR PC = 7

Bomb2:    Universal(Y X PC)
ForEver:  if Y = 0
          THEN GOTO End
          ELSE GOTO ForEver
```

Unsolvability of stability

If there existed an algorithm that could tell for all initial states whether the system of the form (1.4) with the matrix A defined by `Bomb3` below would finally go to zero or not, there would exist an infinite number of initial states for which the system would go to zero *if and only if* the algorithm would say it would *not* do that, and vice versa (this could also be rephrased as the "mortality problem", or convergence to origin):

```
          VAR Y = 0
          VAR X = x    % Unsolvable for some x
          VAR PC = 7

Bomb3:    Universal(Y X PC)
          IF Y = 0
          THEN GOTO ToZero
          ELSE GOTO End
ToZero:   Zero3(Y X PC)
```