this version: April 28, 2010

A semantic Turing machine

Arnold Neumaier
Peter Schodl

Fakultät für Mathematik, Universität Wien
Nordbergstr. 15, A-1090 Wien, Austria
email: Arnold.Neumaier@univie.ac.at
email: Peter.Schodl@univie.ac.at
WWW: http://www.mat.univie.ac.at/~neum/FMathL

Abstract

A semantic Turing machine (STM) is a variant of a programable register machine that combines the transparency and simplicity of the action of a Turing machine with a clearly arranged assembler-style programming language and a user-friendly representation of semantic information.
This paper describes the concept of the STM, its memory management and flow control, and shows how a semantic Turing machine can simulate any ordinary Turing machine. Analogous to a universal Turing machine, we give a universal semantic Turing machine (USTM), which is a special STM that can simulate every STM. The USTM serves both as a self-contained semantic explanation of many aspects of the STM, and as a check that an STM implementation works correctly.
Three appendices give the grammar of the STM programming language, tables for character code, and the essential parts of a MATLAB implementation of the STM.

Contents

1 Introduction **2**

2 The semantic Turing machine **4**

3 The STM programming language **6**

4 Flow control **6**

5 Nodes with constant meaning **8**

6 External values and external processors **9**

7 Description of the STM commands **10**

8 Turing machines and their simulation **14**

9 The USTM **19**

A The grammar of the STM programming language **27**

B Tables of nodes with constant meaning **28**

C The MATLAB implementation **32**

1 Introduction

A **Turing machine**, introduced originally in 1936 by Turing [23], is a commonly used abstract model of a simple computer. Informally, we think of a Turing machine (TM) as a reading/writing head that moves along an arbitrary long tape which is divided into cells, each containing one character. The Turing machine is always in some **state**, and it has a list of instructions, usually called the **transition table**. Determined by the character currently read from the tape and the state the TM is currently in, the transition table assigns to the TM some character to write on the tape, to move one cell to the left or the right, and some state to enter. For a rigorous definition and properties, see, e.g., the classic book by Rogers [17] or Aho et al. [3] or almost any other computability book; see also Section 8 below

The concept of a Turing machine is very simple and at the same time very powerful (we remind of Church's Thesis, discussed, e.g., by Odifreddi [11]), but it has two disadvantages that prevent the use of a TM as a device for efficiently performing calculations:

1. The instructions of the TM are too primitive, their formulation is not intuitive in terms of semantically important actions. Given a set of instructions of some TM, it is very laborious to find out what this TM does.
2. The representation of information on the one-dimensional tape is adequate only in some cases. Usually the result of a calculation cannot be interpreted easily.

We alter the concept of a TM concerning those two issues, and the resulting machine is a **semantic Turing machine** (STM):

Concerning item 1, the STM is able to execute an **STM program**, i.e., a sequence of commands written in an assembler-like language. Each command performs a comprehensible action on the memory.

Concerning item 2, the STM represents information by semantic relations between **nodes** represented by a binary operator, the **dot operation**. Using the dot operation, complex relations can be represented in a simple and user-friendly way, either as a directed, labelled graph, or equivalently as a sparse matrix. Thus an STM allows the expression of semantics in a very natural form.

Alltogether, we think of the STM as a machine that performs some basic actions on its memory (the labelled graph or sparse matrix). The STM has random access to this memory, and the actions it performs (like writing, copying, deleting,. . .) are determined by a human-readable program.

That the STM is as least as powerful as an ordinary Turing machine is shown in Section 8, but we give the STM even more power by allowing it to access the capabilities of the physical device it is implemented on: external memory and external processors, see Section 6. This has the consequence that the STM is no longer equivalent to an ordinary Turing machine, or in other words, not every STM program, regarded as a function on the context,

is **Turing computable**. For example, external processors might have access to the system clock etc..

A cornerstone in the creation of the STM is the proof that the STM is powerful enough to simulate itself. This is done by giving an STM program that can simulate every other STM program. Since this is analogous to the role of a universal Turing machine, we call this program the **Universal Semantic Turing Machine** (USTM).

The USTM is a program short and transparent enough to be checked by hand. It has only 232 lines of code, see Section 9 (compare this, e.g., to the reflective interpreter by Jefferson & Friedman in [4], which has 273 lines). The USTM gives us a possibility to check many aspects of the STM for correctness. Once one has convinced oneself of the correctness of the USTM, one can make the implementation of the STM on some physical device also trustworthy by checking empirically (or, in principle, in a formal way) that any STM program executed by the implemented STM produces the same output as in the case when the USTM simulates this program.

All this makes the STM a semantically self-contained, transparent and easy usable tool that can be a trustworthy foundation for any computer system that deals with semantic content. This concept of building up a high-level language step by step, starting from a low-level languages such as the language of the STM is called **bootstrapping**. For details about bootstrapping, see [22].

Currently, we have a tested implementation of the STM in MATLAB, the essential part of which is given in Appendix C. From the point of view of the physical device, this implementation of the STM is a **virtual machine**.

The term 'semantic Turing machine' has been used recently in a paper by Rodriguez & Bollen [16]. However, their concept of a semantic Turing machine is much closer to an ordinary Turing machine, using a transition table, but operating on a semantic web.

Contents. Section 2 describes the STM and its memory in detail, and Section 3 defines the programming language of the STM. The change from the transition table of a TM to an assembler-like language for the STM makes programming much more convenient, but it requires the storage of information for flow control, described in Section 4. The relation between nodes denoting constants and their physical representation is described in Section 5. The features of the STM that allow it to make use of other algorithms on the physical device are discussed in Section 6. In Section 7 we give a description of every command and some simple examples, and in Section 8 we show how the STM can simulate an ordinary TM. The USTM, a special STM program that can simulate every other STM program, is given in Section 9. Appendix A gives a definition of the grammar of the STM programming language, Appendix B gives a list of all nodes with constant meaning, and Appendix C the essential parts of the current implementation of the STM in MATLAB.

Acknowledgements. D. E. Stevenson from Clemson University contributed useful remarks to an earlier version.

Support by the Austrian Science Fund (FWF) under contract number P20631 is gratefully acknowledged.

2 The semantic Turing machine

A **semantic Turing machine** (STM) is a machine manipulating semantic information about the relation between objects called **nodes**. The nodes are taken from a countable infinite set containing the nonnegative integers; they are represented in human-readable programs as alphanumeric text. The nodes representing the nonnegative integers are called **counters**. The relation between nodes is represented as a partial, binary function written as a **dot operation**, as in `a.b=c`. The dot operation assigns to certain pairs of nodes another node. For example, the information 'f is continuous' can be represented by

`f.continuous=true`,

where `f`, `continuous`, and `true` are nodes. The left argument of the dot operation is called the **record**, the right argument is called the **field**.

We say that **the node `a.b` does (not) exist** if `a.b` is (not) defined. The dot operation is written as left associative, hence `a.b.c` means `(a.b).c`. This way, we can store more complicated relations like `a.b.c=d` in a simple matrix: by setting `x=a.b`, the equation can be rewritten as `x.c=d`.

Instead of a partial function, we can equivalently think of the dot operation as an infinite sparse matrix, or as a directed, labeled graph. The representation of the dot operation as a sparse matrix is called the **semantic matrix**, abbreviated SM.

		continuous	
f		true	

Figure 1: `f.continuous=true` as a matrix

The dot operation regarded as a directed labeled graph is called **semantic graph**, abbreviated SG.

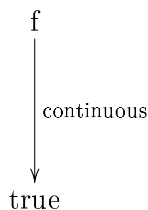

Figure 2: `f.continuous=true` as a graph

In the SM, the record becomes a row and the field becomes a column, while in the SG the record becomes a vertex and the field becomes an edge. One can construct arbitrarily complex records whose content is accessible recursively from their top node. Note that since there are no safeguards to exclude backreferences leading to infinite cycles, records may turn out to be 'infinitely long', in spite of finite memory!

Both representations have advantages and difficulties: The matrix has the disadvantage that it is hard to see which nodes are reachable from one node, which is easy in the directed graph

by simply following the edges. On the other hand, when represented as graph, it might be confusing that a node is used both as a record and a field, since some edges might elsewhere appear as nodes.

The SG is related to the concept of a **semantic network**, introduced by RICHENS [14] in 1956. This and akin concepts are discussed in detail by SOWA [20]. A standardized and widely used example of a semantic network with the aim to be used in the World Wide Web is the Resource Description Framework (RDF), described by MANOLA et al. [7] and specified by LASSILA et al. [6]. Formal details of the semantics of the STM will be given in a separate publication. However, the language of the STM is independent of this interpretation.

Independent of the interpretation of the dot operation either as semantic matrix or as graph, we will refer to it as the **memory** of the STM.

Since there are equivalent formulations of Turing machines which use a 2-dimensional memory instead of the tape (a proof is given by COHEN [2]) the change to a binary operator instead of a tape alone would not go beyond the scope of a Turing machine. But by allowing the STM to manipulate its external environment, the scope of an STM becomes strictly bigger cf. Section 6.

To allow this, nodes can have an **external value**, which is not part of the memory of the STM, but which might be a string, an integer or another data type, stored outside the memory. This external storage is handled exclusively by **external processors**, known to the STM only by name, and the `move` command. The input/output of he STM is also handled via external values. External values are discussed in more detail in Section 6.

The memory of the STM contains the **program** to execute, its **context** (i.e. input and output, corresponding to the tape of an ordinary TM), and the information about **flow control** as well, all represented via the dot operation. To enable the processing of more than one program in the same memory (e.g., when one program calls another program as a subroutine), each program has its own **core**, i.e., a record reserved for temporary data. The core of a called program cannot access the core of the calling program (unless it is reachable from other nodes, which should be avoided to have transparent programs). A recursively called program has one core at every level of the recursion.

When denoting nodes, we will use the **hash** `#` in the following way: an arbitrary node can be represented by a hash followed by some suggestive name. For example, instead of writing `a.b.c` together with the information that `a` is a library, `b` is a program and `c` is a process, we simply write `#lib.#program.#process`.

Since the core is the most important record for a program, will simplify the notation for it: If no confusion can arise about the program (and hence the core) under consideration, we use the **caret** `^` to abbreviate reference to the current core. Hence `^a` means `#core.a`, where `#core` is the core of the program under consideration. In any STM program, `^` means always the core of this program. The caret binds stronger than the dot operation, hence `a.^b` means `a.(#core.b)`.

To start processing a program, the STM needs to know the node that contains the context (input and output) of the program, and since we can store the programs in different **libraries**, also the node denoting the library in which the program code can be found is needed. Therefore the call of an STM program has three arguments: the name of the program, the library and the context. If no library is specified, a **standard library** is assumed. For every call of a program, a new node is created and used as core. Note that we pass the node where the context can be found to the called program, and not the context itself.

Hence we always **call by reference**, not by value. Call by value must be simulated by an explicit `copy` command.

3 The STM programming language

The most elementary part of the STM programming language is a command. There are 33 different commands; a list of the commands and their action is given in Section 7. The commands are divided into three groups: commands that structure the program but have no influence on the memory at runtime, commands for flow control, and assignments, which make alterations in the memory of the STM or external values.

STM programs are, compared to transition tables of Turing machines, much less intricate. In fact, the STM programming language is much more akin to an assembler-style language.

Before describing the commands in detail, we first have to say something about the structure of the language, nodes with constant meaning, and external processors and values. This is the content of this and the next two sections.

The STM programming language, fully defined by the grammar in Appendix A, has the reserved names

```
program        process        start
```

for structuring the program, and the reserved names

```
create         move           exist          function
fields         in             if             stop
copy           out            vcopy
of             as             goto
const          clean          external
```

for commands making certain alterations in the memory or in flow control. Other names that should not be used since they have internal meaning are given in Table 14.

The meaning of these nodes will be discussed in Section 4. All other names and more general alphanumeric strings may be used as variables for nodes.

The STM programming language is the lowest level of a fully comfortable programming language that we are in the process of developing.

4 Flow control

This section describes how the information for flow control is represented in the memory of the STM.

A **process** is a sequence of commands, beginning with the command `process(#proc)`. Every process ends with a command that either halts the STM or calls another process. A process can be antered only at its first command, but it is possible to leave a process before its last command line is reached.

An **STM program** is the command `program(#prog)` followed by a sequence of processes.

Each STM-program is represented in the memory by a number of relations of this type:

`#lib.#program.#process.#line.#part=#node`,

where `#lib`, `#program` and `#process` are the names of the library, the program and the process, and `#line` is the number of the line in the process. The node `#part` is one of the following: `commname` refers to the name of the command, `1` to the first argument, `2` to the second argument, and `3` to the third argument.

The node `#lib.#program.start` contains the node referring to the first process, i.e., the process that has to be executed first in the program `#program`.

The (changing) STM command currently executed is called the **focus**. The focus is represented in the node `^process.^line`, where `^process` contains the process currently executed, and `^line` contains a counter. The node `^process.^line` then is the currently executed line in the currently executed process, hence the focus. Incrementing `#line` means to proceed one line forward in the programm. Changing `^process` and setting `^line=1`, as done by the `goto` and `if` command, sets the focus to the first line of another process.

The memory of the STM may contain more than one program: a program (the **caller**) can call another program (the **callee**) as a subroutine. Every STM program that was not called by the user but by another STM program is called a **function**. A program can only modify nodes reachable from its own core and its context node.

Since each program has its own core, context etc., it is not sufficient just to manage the information which command is currently executed, but a collection of information concerning flow control called a **frame** has to be kept in the memory. There is one frame for every program, called the **local frame**, and one special additional frame that enables the STM to jump between programs, called the **global frame**. (Note that in this sense, a program that calls itself as a subroutine are two separate programs with two separate local frames, although the program code is represented only once in the memory of the STM.)

To separate different programs, and to be able to properly return to the caller, the STM enters a new **level** each time one program calls another and returns to the previous level upon completing the called program. The information in which level the STM actually operates is available in `corelist.depth`. The node `corelist.depth` contains a counter, and `corelist.(corelist.depth)` contains the node which is the core of the program on level `corelist.depth`.

So when calling a function via a command of the form `function: #commname(^#context,^#lib)`, the node `corelist.depth` has to be incremented and the caret is set to a newly created node. When leaving a function, `corelist.depth` is decremented and the caret is set back to `corelist.(corelist.depth)`.

	depth	1	2	...
corelist				

Table 1: The structure of the global frame

The local frame of some program can be accessed only by this program itself, and holds all the information about the current state of the program, or of the state the program was in, in the moment another program was called. The local frame can be accessed via the core, so which local frame is used is determined only by the current core. Hence when setting back the caret to the core of the caller of a function, the STM is set to continue executing

the caller, since it now uses the local frame of the caller. In the memory, a local frame looks as illustrated in Table 2, if `#core` is the core of the program currently executed. The local frame contains

- the context, library and the core, written in the nodes `^context`, `^lib` and `^core`;
- the name of the program, `^program` contains the node `#lib.#program`, and is needed to jump to other processes by setting `^process = #lib.#program.#p`, where `#p` is the name of the process to jump to;
- the node `^process` containing `#lib.#program.#process`, and
- the node `^line` containing the counter representing the current line in the process, and needed to obtain the focus via `^process.^line`.

	`process`	`line`	`context`	`lib`	`program`	`core`
`#core`						

Table 2: The structure of a local frame

Confusion might arise from the fact that `^core` always contains the current core.

5 Nodes with constant meaning

Although nodes are semantic entities, on a physical device they always have to be represented by a list of **bytes**. Hence in an implementation of the STM, we have to store the correlation between nodes and bytes in a table.

We write octal numbers with a preceding slash, e.g. $/10 = 8$. The 512 counters whose names are the octal numbers between /0 and /1777 are treated as **constants**, i.e. these nodes are counters that have a fixed semantic interpretation when.

A byte exists of 8 bits, we write it in terms of 3 octals as /xyz with x a digit between 0 and 3, and y and z digits between 0 and 7. This naturally groups the $4 \cdot 64 = 256$ bytes by their first octal x into 4 groups of 64 characters each.

Our coding scheme allows a compact and readable representation of arbitrary text, accommodating up to $3 \cdot 64 = 192$ primary characters quotable without alteration, and a handful of auxiliary characters with a syntactical meaning. We have chosen our set of primary characters, specified in Appendix B, with an eye on being able to represent typical mathematical text, including formulas.

The constants whose names fit a style byte are used as **characters**. Bytes with x=0 are called **ordinary characters (ochar)**, those are the digits and alphabetic characters, together with blank ( ) and newline (↵). In **charcode text**, which is a low-level output format, they will be printed directly by the corresponding symbol. The bytes with x=1 are called **regular characters (rchar)**, these are symbols frequently used in text, such as parenthesis, punctuation marks etc. As charcode text, regular characters are right quotes followed by an ochar. Since two rchars are frequently used in the programs, we want to introduce their charcodes here, namely the node with the meaning *true* ($\top$) with charcode `'T` and *false* ($\bot$) with charcode `'F`. The bytes with x=2 are called **special characters (schar)**, they represent symbols for mathematical typesetting. As charcode text, they are

printed by a double quote followed by an ochar. The remaining 64 characters with x=3 are used as **auxiliary characters (xchar)**, and **keys**: In charcode text, they are a left quote followed by a digit. The auxiliary characters `‘0`–`‘9` are quotes and control characters. All other bytes of the form #3xz represent keys: Keys are operators to the following characters, for example the key `‘n` means ‘make a slash through the next symbol’. For example, `‘n"B` is the charcode text for $\not\Rightarrow$. Other keys are used to change fonts, for unicode characters, for IEEE floating point numbers etc..

Appendix B contains a complete list of the ochars, rchars, schars and xchars (Tables 10 – 13) and not for keys since the identification of keys is still under development. On media where an rchar or schar character prints naturally, the encoding by charcode is optional. Thus it is permitted to write `(xy)` for `’6xy’7`. But xchar characters must always be encoded in charcode to ensure unique decodability. The details of the coding scheme are relevant for defining specific protocols for import and export, and will be described elsewhere.

Since constants are counters, they can be incremented and decremented by the STM. For example, by multiple incrementations, one can change an entry in the memory from `’T` (*true*) to `’F` (*false*). But since the nodes for flow control are not counters, one cannot access vital nodes for flow control in this fashion.

6 External values and external processors

The STM has the ability to access the facilities of the physical device it is implemented on. This may provide the STM with much better performance for tasks it can export, and allows the use of external processors and programs in different programming languages.

Every node can have an **external value**, which is some data associated to this node, but not part of the memory of the STM. Instead, it is managed by the physical device which executes the STM. In descriptions of commands, we refer to the external value of the node `#node` by `VALUE(#node)`.

The values of nodes are directly processed by the physical device. Hence one can benefit from the full computational power of the physical device. A computation on the external values is said to be realized by an **external processor**. External processors have no access to the memory of the STM, but may be called from the STM as the command `external`. From a theoretical point of view, external processors are **oracles** to the STM, as defined by SHOENFIELD [19].

External values can be copied to the memory of the STM, and conversely. This is done by the commands `in` and `out`. The information about how to represent the external value in the memory of the STM is called the **protocol**, and is used as an argument for the commands `in` and `out`. Our current implementation includes protocols for representing

- natural numbers,
- charcode text,
- STM-programs,
- tapes and transition tables for the Turing machine (see Section 8).

There may be an arbitrary number of protocols, as long as the device on which the STM is implemented knows how to interpret them.

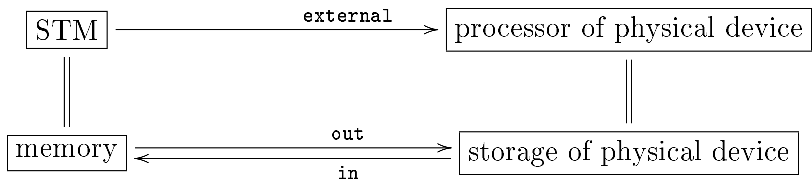

Figure 3: Interaction of the STM with the physical device

7 Description of the STM commands

We now introduce the commands of the STM language and describe their effect. There are four groups of commands: Table 3 describes the commands that are needed to give the program an appropriate structure. Table 4 contains the assignments, i.e., those commands that perform alterations in the memory of the STM. If an assignment refers to an non-existing node, an error is produced. Table 5 gives the commands used for flow control, and Table 6 the commands that establish communication with the physical device, namely call external processes and access external values.

We remind the reader that `'T` stands for *true*($\top$), and `'F` for *false*($\bot$), see Section 5.

STM command	comment
`program #1`	first line of the program `#1`
`process #1`	first line of the process `#1`
`start #1`	start with process `#1`

Table 3: Structuring commands

STM command	comment
^#1=(^#2==^#3)	sets ^#1 to 'T if ^#2 = ^#3, else to 'F
^#1=copy of ^#2	makes a complete copy of the graph with root ^#2 to the node ^#1
^#1=copy of #2	makes a complete copy of the graph with root #2 to the node ^#1
#1=copy of ^#2	makes a complete copy of the graph with root ^#2 to the node #1
^#1.#2=^#3	assigns ^#3 to ^#1.#2
^#1.^#2=^#3	assigns ^#3 to ^#1.^#2
^#1=^#2.#3	assigns ^#2.#3 to ^#1
^#1=^#2.^#3	assigns ^#2.^#3 to ^#1
^#1.#2=const #3	writes #3 to ^#1.#2, but produces an error if the node #3 is not a constant node
^#1.^#2=const ^#3	writes ^#3 to ^#1.^#2, but produces an error if the node #3 is not a constant node
^#1 ++	increments the counter ^#1, but produces an error if ^#1 is not a counter
^#1 --	decrements the counter ^#1, but produces an error if ^#1 is not a counter
create ^#1	assigns some free node to ^#1
^#1=fields of ^#2	assigns the used fields of the record ^#2 to ^#1.1, ^#1.2,...
^#1=exist(#2.#3)	sets ^#1 to 'T if #1.#2 exists, else to 'F
^#1=exist(^#2.^#3)	sets ^#1 to 'T if ^#1.^#2 exists, else to 'F
clean ^#1	deletes the nodes reachable *only* from ^#1

Table 4: Assignment commands

STM command	comment
goto #1	sets the focus to the first line of process #1
goto ^#1	sets the focus to the first line of process ^#1
if ^#1 goto #2	sets the focus to the first line of process #2 if ^#1='T, and to the next line if ^#1='F
function: #1(^#2,^#3)	starts execution of the STM program #1 in library ^#3 with context ^#2
stop	ends a program

Table 5: Commands for flow control

STM command	comment
`external #1(^#2)`	starts execution of the external processor `#1` with context `^#2`
`external ^#1(^#2)`	starts execution of the external processor `^#1` with context `^#2`
`in ^#1 as #2`	imports `VALUE(^#1)` into `^#1` by protocol `#2`
`out ^#1 as #2`	exports `^#1` into `VALUE(^#1)` by protocol `#2`
`in ^#1 as ^#2`	imports `VALUE(^#1)` into `^#1` by protocol `^#2`
`out ^#1 as ^#2`	exports `^#1` into `VALUE(^#1)` by protocol `^#2`
`^#1=‘string‘`	sets `VALUE(^#1)` to `string`
`^#1=vcopy of ^#2`	sets `VALUE(^#1)` to `VALUE(^#2)`

Table 6: Commands for external communication

Example 7.1: A simple loop. To get acquainted with the STM programing language, we give a simple example for an STM program.

This example writes the constant `5` in the node `^five` and the constant `1` in `^x`, increments the node `^x`, and if the two nodes are not equal, loops and increments `^x` again. Hence, when this program halts, it will have performed four iterations of the loop, and `^x` will contain `5`. Lines beginning with a percent sign \% are treated as comments and do not affect the execution of the program.

```
program test
% first line of the program "test"
process setup
% first line of the process "setup"
        ^core.x=const 1
        ^core.five=const 5
        goto loop
process loop
        ^x ++
        ^test=(^x==^five)
        if ^test goto end
        goto loop
process end
        stop
start setup
```

Upon execution of this program the following happens: Because of the line `start first` at the end, the focus is set to the beginning of process `first`. Then the core is filled: `^five` is set to `5`, and `^x` is set to `1` (we remind that `^core.a=^a`, see Section 4). The next command makes the focus jump to the first line of the process `loop`. The node `^x` is incremented, and then `^x` is compared to `^five`. The result of this comparison, i.e., either `'T` (true) or `'F` (false), is written in `^test`. The next command performs a jump to the process `end` if the content of `^test` is `'T`, and proceeds to the next line if `^test` is `'F`. The next line sets the focus back to the first line of the process `loop`.

Example 7.2: Hello, world! This program is an example of how to use an external processor. It simply prints `Hello, world!` on the screen, but since the STM has no direct

access to the screen of the physical device, external values and an external processor have to be used. Hence the display depends on the external processor, which is not described here. Instead, we assume an external processor printvalue, which prints VALUE(x) to the screen when invoked printvalue(x).

To explain certain features of the STM, we give two versions of the Hello, world! program: The first version on the left side writes all characters one by one as constant nodes (see section 5), the second one on the right side makes use of a special STM command that writes an external value while parsing. The reader is invited to create a third equivalent (but less instructive) program, further shortening the second one.

```
program helloworld
process fillcontext
        ^context.0=const 13
        ^context.1=const H
        ^context.2=const e
        ^context.3=const l
        ^context.4=const l
        ^context.5=const o
        ^context.6=const ,
        ^context.7=const 
        ^context.8=const w
        ^context.9=const o
        ^context.10=const r
        ^context.11=const l
        ^context.12=const d
        ^context.13=const !
        goto display
process display
        out ^context as charcode
        external: printvalue(^context)
        stop
start fillcontext
```

```
program helloworld2
process fillvalue
        ^context=‘Hello, world‘
        in ^context as string
        ^toinc=^context.0
        ^toinc ++
        ^context.0=^toinc
        create ^exclmark
        ^exclmark=‘!‘
        in ^exclmark as string
        ^exclmark=^exclmark.1
        ^context.^toinc=^exclmark
        goto display
process display
        out ^context as charcode
        external: printvalue(^context)
        stop
start fillvalue
```

We give short descriptions of the processes:
Due to the line start(fillcontext) / start(fillvalue), process fillcontext / fillvalue is executed first. In the program on the left, process fillcontext writes the characters of the string to the node ^context, one by one, as constants. Since protocol charcode assumes a counter giving the length of the string at position #string.0, ^context.0 is set to 13, the length of the string.

In the program on the right, the process fillvalue first writes at the parse stage the string Hello, world into VALUE(#newnode) for some unused node #newnode. During runtime, this command sets VALUE(^context) = VALUE(#newnode), hence we end up with VALUE(^context) = Hello, world. Then, this external value is imported as a string to the node ^context. Since the exclamation mark ! is missing, we increment ^context.0 (which contains a counter, hence no error is produced) and append the exclamation mark at the end of the string.

In both programs, the process display writes the string Hello, world! to VALUE(^context). Then it calls the external processor printvalue, which prints VALUE(^context) to the

screen. So the result of execution of either program is `Hello, world!` printed on the screen.

8 Turing machines and their simulation

In this section we introduce Turing machines formally and show that the STM is a generalization of a Turing machine, by giving an STM-program that simulates an ordinary Turing machine.

The tape of the Turing machine The cells on the tape of the Turing machine are simulated by the nodes `^context.tape.0`, `^context.tape.1` etc., where `^context.tape.0` is the left end of the tape, and initially holds the delimiting symbol '`>`'. Nodes that do not exist are interpreted by the Turing machine as blank spaces. The alphabet of the Turing machine is arbitrary, but must not contain the pipe `|`. Furthermore, the characters `>` and the blank space are reserved. At the beginning of execution, the head of the TM is assumed to be on `^context.tape.1`, and the TM to be in state 1.

The instructions of the Turing machine The action of the Turing machine is determined by a finite list of **instructions** of the form

$$S \mid R \mid W \mid M \mid S'$$

which applies if S is the state the TM is currently in, and R is the symbol currently read by the TM. In this case, the following actions are performed, in this order:

1. The symbol W is written. If W is the empty string, then nothing is written.
2. The head moves one cell to the right if M = `R` and one cell to the left if M = `L`. If M is the empty string, then no movement is performed.
3. The state of the TM changes to S'.

If no instruction applies, the TM halts.

Example 8.1: Replacement This is an example of a very simple Turing machine, which simply replaces in a string of `a`'s and `b`'s every occurring `a` by `c`. It has only one state, and runs through the tape from left to right, replacing every `a` it runs along. When it reaches the end of the string, it reads a blank, and no instruction applies, hence the Turing machine halts.

The two instructions are:

```
1|b||R|1
1|a|c|R|1
```

Thus, the tape with initial content

`> b b b a b b a b b a`

has, when the Turing machine halts, the content:

`> b b b c b b c b b c`

Example 8.2: Division with remainder This is a more complicated example of a Turing machine that performs a division with remainder. The number of `a`'s at the beginning of the tape is divided by the number of the `b`'s following. The result is represented as the number of `q`'s for the quotient, and the number of `r`'s for the remainder, when the Turing machine halts.

For example, if we want to perform 8 divided by 3, the tape initially looks like this:

`> a a a a a a a a b b b`

where the eight `a`'s represent the dividend and the three `b`'s the divisor.

When the Turing machine halts, the tape contains:

`> A A A A A A A A b b b q q r r`

which tells us that the quotient is 2 (represented by the two `q`'s), and the remainder is also 2 (represented by the two `r`'s). Note that the `a`'s have changed to `A`'s in order for the processed `a`'s to be distinguishable from those not yet processed.

This is the transition table of the Turing machine that performs a division with remainder:

```
1|a||R|1     3|a||R|3     4|b||L|2     6|B|b|L|6    7|q||R|7     9|B||R|9
1|b||L|2     3|A||R|3     4|q|||5      6|A|||2      7| ||L|8     9|b||R|9
2|B||L|2     3|B||R|3     4| |||5      6|a|||2      8|q||L|8     9|q||R|9
2|A||L|2     3|b|B|R|4    5|q||R|5     7|A||R|7     8|b||L|8     9|r||R|9
2|a|A||3     3| |||5      5| |q||6     7|B||R|7     8|r||L|8     9| |r||8
2|>||R|7     3|q|||5      6|q||L|6     7|b||R|7     8|B|b||9
```

This Turing Machine performs the division by the following steps:

State 1 just brings the head in the right position to start:
The head moves to the right until the first `b` is read, then moves one cell to the left and enters state 2.

State 2, 3, 4 and 5 determine the quotient, i.e., the number of `q`'s on the tape: For every `b`, an `a` is replaced by an `A`, and the `b` is replaced by a `B`. If there is no more `b` on the tape, one `q` is written and the `B`'s are replaced by `b`'s:
In **state 2**, the head is moved to the left, until the rightmost `a` is reached. If there is no `a` on the tape (because all `a`'s have been replaced by `A`'s to mark them as processed), the TM changes to state 7. Else the rightmost `a` is changed to `A`, and the TM enters state 4. **State 3** then replaces the leftmost `b` by a `B` and changes to state 4. If there is no `b` on the tape (every `b` has been replaced by a `B`), the TM changes to state 5. **State 4** is only a case distinction: if the `b` just replaced was the last one, then change to state 5, and if there is still a `b` on the tape, then change to state 2, i.e., perform a loop. **State 5** moves the head to the right until it reaches an empty cell, writes a `q`, and changes to state 6.

State 6 changes all `B`'s back to `b`'s and puts the head on the rightmost cell containing either `A` or `a`. The TM is then set to state 2 again and this is repeated (and every time a `q` is written) until there are no more `a`'s on the tape, and state 7 is entered.

State 7 then simply puts the head to the last nonempty cell of the tape and changes to state 8.

States 8 and 9 determine the remainder of the division (i.e., `b`'s which has not been replaced by `B`'s in state 3) and write the corresponding number of `r`'s to the tape:
The head is put to the rightmost `B` by **state 8**, and before entering state 9, this `B` is replaced by a `b`. If there is no `B` on the tape, then no instruction applies and the TM halts. In **state**

9, the head is put to the first empty cell of the tape, an `r` is written there, and the TM enters state 8 again, hence loops.

Since their introduction, Turing machines have been subject of intensive study. Besides serving as a theoretical basis of all programming languages (λ-calculus being another), Turing machines have many interesting applications, reaching from problems in logic, e.g., the halting problem c.f. Odifreddi [11], to formal languages (see Cohen [2]). Turing machines have been applied to biology by Tanomaru [21] for the study of mutation, and a special Turing machine is Langton's Ant, introduced in 1986 by Langton [5] and generalized by Beuret & Tomassini [1]. Turing Machines have even been considered extensively in cognitive sciences and philosophy, prominently by Penrose [12], and more recently by Scheuz [18].

There has been put much effort in constructing smaller and smaller **universal Turing machines**, i.e., special Turing machines that can simulate every other Turing machine, from the 1950's until today[1]. Small universal Turing machines were studied, e.g., in the influential paper [9] by Minsky, by Robinson [15], and recently by Neary & Woods [10].

Also there is research going on searching for Turing machines that write as many characters as possible without looping forever. This problem was introduced in 1962 by Rado in [13], and such a Turing machine is called a **busy beaver**. For most classes of busy beavers, there exists only a current 'champion', i.e., the best known version, while maximality is not proven. E.g., the 2-symbol, 5-state Busy Beaver introduced 1990 by Marxen & Buntrock [8] writes 4098 characters before halting, making it champion of its class.

Very little effort was put in making universal Turing machines user-friendly while keeping them small, which would be related to the goal of our work.

The STM-code of a universal TM

To get further acquainted with the STM programming style we now specify the STM-program that simulates an arbitrary Turing machine. The reader should interpret it with the help of Tables 3 – 6 .

The instructions of the Turing machine are represented in the memory of the STM as follows:

`^context.state.`n	contains S of instruction n
`^context.reading.`n	contains R of instruction n
`^context.towrite.`n	contains W of instruction n
`^context.tomove.`n	contains M of instruction n
`^context.tostate.`n	contains S' of instruction n

Note that if W in the instruction is the empty string, `#nr.towrite` is set to `#nr.reading`, hence no alteration is done by writing. If M in the instruction is the empty string, `#nr.tomove` is set to `X`, just to distinguish it from `L` and `R`. The position of the head is stored in `^position`, and the state of the Turing Machine is stored in `^state`.

The first program `LOADANDCALL` is not part of the universal Turing machine itself, but imports the instructions to simulate, the tape to work on, and the STM program `UTM`. To demonstrate how to import an STM program as an external value, we create a new library in `^lib2` and import the program `UTM` there. Then `LOADANDCALL` calls `UTM` as a function.

[1] In October 2007, Alex Smith claimed to have found the smallest Universal Turing Machine possible, having 2 states and 3 symbols, see `www.wolframscience.com/prizes/tm23/solved.html`

```
program LOADANDCALL

process loadtm
% imports the tape "divide14by3" and the transition table "division"
        create ^tapein
        ^context.tape=^tapein
        ^tapein=‘divide14by3.tape‘
        in ^tapein as tm_tape
        ^context=‘division.tm‘
        in ^context as tm_instructions
        goto loadutm
process loadutm
% imports the STM program in the file "utm" into the library in ^lib2
        create ^utmprog
        create ^lib2
        create ^progname
        ^progname=‘utm‘
        ^utmprog.lib=^lib2
        ^utmprog.name=^progname
        in ^utmprog as stmprogram
        goto call
process call
% starts execution of the STM program UTM
        function: UTM(^context,^lib2)
        out ^tapein as tm_tape
        stop
start loadtm
```

The next STM-program `UTM` essentially searches for a command that applies, then performs the instructions, and then loops. In more detail,

`init`	initially sets up the records so that processing can begin,
`nextcommand`	resets the records and replaces an empty cell on the tape by a blank.
`trynext`	brings the next instruction to consideration, and halts if there is no next instruction.
`checkstate`	and
`checksymbol`	compares the state of the Turing Machine with S in the instruction, and the symbol currently read on the tape with R in the instruction, respectively. In other words, these two processes check if the instruction under consideration applies.
`executecommand`	performs the actions given in the instruction, except for moving the head to the left or the right.
	Since `^position` contains a counter representing the position of the head on the tape,
`left`	moves the head to the left by decrementing `^position`, and
`right`	moves the head to the right by incrementing `^position`.

Correctness is straightforward to prove.

```
program UTM

process init
% makes the necessary nodes available in the core
        ^core.position=const 1
        ^core.state=const 1
        ^core.cR=const R
        ^core.cL=const L
        ^commandstate=^context.state
        ^commandreading=^context.reading
        ^commandtowrite=^context.towrite
        ^commandtomove=^context.tomove
        ^commandtostate=^context.tostate
        ^taperef=^context.tape
        goto nextcommand

process nextcommand
% initializes the comparing, reads the tape, replaces empty node
% by a blank space
        ^core.trycommand=const 1
        ^symbolontape=^taperef.^position
        ^notreadingblank=exist(^taperef.^position)
        if ^notreadingblank goto checkstate
        ^core.symbolontape=const
        goto checkstate

process trynext
% read the next command, halts the TM if there are no instructions left
        ^trycommand ++
        ^therearemorecommands=exist(^commandstate.^trycommand)
        if ^therearemorecommands goto checkstate
        stop

process checkstate
% checks if the state of the TM is equal to the state in the command
        ^stateincommand=^commandstate.^trycommand
        ^samestate=(^state==^stateincommand)
        if ^samestate goto checksymbol
        goto trynext

process checksymbol
% checks if the symbol read by the TM is equal to
% the symbol in the command
        ^symbolincommand=^commandreading.^trycommand
        ^samesymbol=(^symbolincommand==^symbolontape)
        if ^samesymbol goto executecommand
        goto trynext

process executecommand
```

```
% executes the instructions in the command
        ^towrite=^commandtowrite.^trycommand
        ^tomove=^commandtomove.^trycommand
        ^tostate=^commandtostate.^trycommand
        ^taperef.^position=^towrite
        ^core.state=^tostate
        ^moveleft=(^tomove==^cL)
        ^moveright=(^tomove==^cR)
        if ^moveleft goto left
        if ^moveright goto right
        goto nextcommand

process left
% moves the head to the left
        ^position --
        goto nextcommand

process right
% moves the head to the right
        ^position ++
        goto nextcommand

start init
```

The STM program above simulates an arbitrary TM and does not use external storage. Since an ordinary TM has no external storage and it is not specified how an external processor should behave, it is impossible to give an ordinary TM that simulates an arbitrary STM program.

9 The USTM

The USTM is a special STM program capable of 'simulating' the processing of an other STM program `P` in the following sense: The context of the USTM contains the STM program `P` and the context of `P`. When the USTM has finished, the USTM has produced the same changes in the context as `P` would have produced when called directly.

When the USTM is started, nodes for `program`, `context` and `library` have to be passed to the USTM as part of its core. It is assumed that this information is stored in the nodes `^sim_prog`, `^sim_context` and `^sim_lib` before calling the USTM.

The following table gives an overview which process in the USTM program simulates which command in the STM program. The commands `program`, `process` and `start` do not have to be simulated since they only structure the program, and their effect is already covered by the way the program to be simulated is represented in the memory.

Command	Process
program #program	–
process #process	–
start #process	–
^#1=^#2.#3	get
^#1=^#2.^#3	refget
^#1.#2=^#3	set
^#1.^#2=^#3	refset
^#1.#2=const #3	setconst
^#1.^#2=const ^#3	setconstref
^#1=copy of ^#2	copy
#1=copy of ^#2	copyfromcore
^#1=copy of #2	copytocore
create ^#newnode	create
^#isequal=(^#left==^#right)	check
^#counter ++	inc
^#counter --	dec
^#fieldlist=fields of ^#record	fields
goto #process	goto
goto ^#process	move
if ^#cond goto #process	if
function: #program(^#context,^#library)	function
stop	stop
external: #program(^#input)	external
external: ^#program(^#input)	externalref
clean ^#node	clean
in ^#node as #protocol	transportin
out ^#node as #protocol	transportout
in ^#node as ^#protocol	transportrefin
out ^#node as ^#protocol	transportrefout
^#node=exist(#record,#field)	exist
^#node=exist(^#record,^#field)	existref
^#1='string1'	string
^#1=vcopy of ^#2	vcopy

Table 8: The 33 STM commands and their USTM processes

The STM code of the USTM

The example program below implements a simulator for the STM, which shows that the STM programming language is universal.

```
program USTM

process init
% initialize nodes, initialize local and global frame
        create ^ustmcore
        ^sim_context=^context.context
        ^sim_prog=^context.prog
```

```
        ^sim_lib=^context.lib
        ^sim_libprog=^sim_lib.^sim_prog
        ^sim_startproc=^sim_libprog.start
        create ^corelist
        ^corelist.depth=const 1
        ^corelist.1=^ustmcore
        ^ustmcore.process=^sim_startproc
        ^ustmcore.core=^ustmcore
        ^ustmcore.line=const 1
        ^ustmcore.context=^sim_context
        ^ustmcore.program=^sim_libprog
        ^ustmcore.programcore=^simulcore
        ^ustmcore.lib=^sim_lib
        goto load

process next
% proceed to the next command to simulate
        ^depth=^corelist.depth
        ^ustmcore=^corelist.^depth
        ^toinc=^ustmcore.line
        ^toinc ++
        ^ustmcore.line=^toinc
        goto load

process load
% load the information about the command to simulate to the core
        ^depth=^corelist.depth
        ^ustmcore=^corelist.^depth
        ^sim_process=^ustmcore.process
        ^sim_line=^ustmcore.line
        ^sim_focus=^sim_process.^sim_line
        ^sim_comm=^sim_focus.1
        ^arg1=^sim_focus.2
        ^arg2=^sim_focus.3
        ^arg3=^sim_focus.4
        goto ^sim_comm

process external
% external: #program(^#input)
        ^data=^ustmcore.^arg2
        external: ^arg1(^data)
        goto next

process externalref
% external: ^#program(^#input)
        ^prog=^ustmcore.^arg1
        ^data=^ustmcore.^arg2
        external: ^prog(^data)
        goto next
```

```
process transportin
% move in ^#node as #protocol
        ^totransport=^ustmcore.^arg1
        in ^totransport as ^arg2
        goto next

process transportout
% move out ^#node as #protocol
        ^totransport=^ustmcore.^arg1
        out ^totransport as ^arg2
        goto next

process transportrefin
% move in ^#node as ^#protocol
        ^totransport=^ustmcore.^arg1
        ^protocol=^ustmcore.^arg2
        in ^totransport as ^protocol
        goto next

process transportrefout
% move out ^#node as ^#protocol
        ^totransport=^ustmcore.^arg1
        ^protocol=^ustmcore.^arg2
        out ^totransport as ^protocol
        goto next

process copy
% ^#1=copy of ^#2
        ^thecopy=^ustmcore.^arg1
        ^theoriginal=^ustmcore.^arg2
        ^thecopy=copy of ^theoriginal
        goto next

process copyfromcore
% #1=copy of ^#2
        ^theoriginal=^ustmcore.^arg2
        ^arg1=copy of ^theoriginal
        goto next

process copytocore
% ^#1=copy of #2
        ^thecopy=^ustmcore.^arg1
        ^thecopy=copy of ^arg2
        goto next

process string
% ^#1=`string1`
        ^lhsvcopy=^ustmcore.^arg1
```

```
        ^lhsvcopy=vcopy of ^arg2
        goto next

process vcopy
% ^#1=vcopy of ^#2
        ^lhsvcopy=^ustmcore.^arg1
        ^rhsvcopy=^ustmcore.^arg2
        ^lhsvcopy=vcopy of ^rhsvcopy
        goto next

process setconst
% ^#1.#2=const #3
        ^setleft=^ustmcore.^arg1
        ^setleft.^arg2=const ^arg3
        goto next

process setconstref
% ^#1.^#2=const ^#3
        ^setleft=^ustmcore.^arg1
        ^setright=^ustmcore.^arg2
        ^toset=^ustmcore.^arg3
        ^setleft.^setright=const ^toset
        goto next

process create
% create ^#newnode
        ^toassign=^ustmcore.^arg1
        create ^toassign
        ^ustmcore.^arg1=^toassign
        goto next

process clean
% clean ^#node
        ^toclean=^ustmcore.^arg1
        unlink ^toclean
        goto next

process fields
% ^#fieldlist=fields of ^#record
        ^writeto=^ustmcore.^arg1
        ^lookfor=^ustmcore.^arg2
        ^writeto=fields of ^lookfor
        goto next

process function
% function: #program(^#context,^#library)
        ^toinc=^corelist.depth
        ^toinc ++
        ^corelist.depth=^toinc
```

```
        create ^newcore
        ^corelist.^toinc=^newcore
        ^contextofcallee=^ustmcore.^arg2
        ^newcore.context=^contextofcallee
        ^newcore.line=const 1
        ^libcalled=^ustmcore.^arg3
        ^libprog=^libcalled.^arg1
        ^newcore.program=^libprog
        ^proctoexec=^libprog.start
        ^newcore.process=^proctoexec
        ^newcore.lib=^arg3
        ^newcore.core=^newcore
        ^newcore.programcore=^simulcore
        goto load

process check
% ^#isequal=(^#left==^#right)
        ^checkleft=^ustmcore.^arg2
        ^checkright=^ustmcore.^arg3
        ^checked=(^checkleft==^checkright)
        ^ustmcore.^arg1=^checked
        goto next

process exist
% ^#node=exist(#record,#field)
        ^exleft=^core.arg2
        ^exright=^core.arg3
        ^copytoex=exist(^exleft.^exright)
        ^answer=^core.arg1
        ^ustmcore.^answer=^copytoex
        goto next

process existref
% ^#node=exist(^#record,^#field)
        ^exleft=^ustmcore.^arg2
        ^exright=^ustmcore.^arg3
        ^copytoex=exist(^exleft.^exright)
        ^answer=^core.arg1
        ^ustmcore.^answer=^copytoex
        goto next

process goto
% goto #process
        ^currentprogram=^ustmcore.program
        ^gotoprocess=^currentprogram.^arg1
        ^ustmcore.process=^gotoprocess
        ^ustmcore.line=const 1
        goto load
```

```
process refset
% ^#1.^#2=^#3
        ^refsetleft=^ustmcore.^arg1
        ^refsetright=^ustmcore.^arg2
        ^refsetto=^ustmcore.^arg3
        ^refsetleft.^refsetright=^refsetto
        goto next

process set
% ^#1.#2=^#3
        ^setleft=^ustmcore.^arg1
        ^setto=^ustmcore.^arg3
        ^setleft.^arg2=^setto
        goto next

process refget
% ^#1=^#2.^#3
        ^refgetleft=^ustmcore.^arg2
        ^refgetright=^ustmcore.^arg3
        ^towrite=^refgetleft.^refgetright
        ^ustmcore.^arg1=^towrite
        goto next

process get
% ^#1=^#2.#3
        ^getleft=^ustmcore.^arg2
        ^towrite=^getleft.^arg3
        ^ustmcore.^arg1=^towrite
        goto next

process if
% if ^#cond goto #process
        ^core.zero=const 0
        ^core.one=const 1
        ^tr=(^one==^one)
        ^fal=(^zero==^one)
        ^tojump=^ustmcore.^arg1
        ^applies=(^tojump==^tr)
        ^appliesnot=(^tojump==^fal)
        if ^applies goto ifapplies
        if ^appliesnot goto ifappliesnot
        stop
process ifapplies
        ^currentprogram=^ustmcore.program
        ^gotoprocess=^currentprogram.^arg2
        ^ustmcore.process=^gotoprocess
        ^ustmcore.line=const 1
        goto load
process ifappliesnot
```

```
        goto next

process move
% goto ^#process
        ^target=^ustmcore.^arg1
        ^currentprogram=^ustmcore.program
        ^movetoprocess=^currentprogram.^target
        ^ustmcore.process=^movetoprocess
        ^ustmcore.line=const 1
        goto load

process inc
% ^#counter ++
        ^toinc=^ustmcore.^arg1
        ^toinc ++
        ^ustmcore.^arg1=^toinc
        goto next

process dec
% ^#counter --
        ^todec=^ustmcore.^arg1
        ^todec --
        ^ustmcore.^arg1=^todec
        goto next

process stop
% stop
        ^todec=^corelist.depth
        ^calleecore=^corelist.^todec
        ^core.one=const 1
        ^finish=(^todec==^one)
        ^todec --
        ^corelist.depth=^todec
        if ^finish goto stopprogram
        goto returntocaller
process stopprogram
        unlink ^calleecore
        stop
process returntocaller
        ^callercore=^corelist.^todec
        ^resulttocopy=^calleecore.result
        ^placetocopy=^calleecore.context
        ^callercore.^placetocopy=^resulttocopy
        unlink ^calleecore
        goto next
start init
```

Without blank lines and comment lines, the USTM contains 232 lines.

A The grammar of the STM programming language

We give the complete grammar of the STM programming language with partially labelled, BNF like productions. A line beginning with a percent sign `%` is treated as a comment without any effect on the program. To ease readability, white spaces at the beginning of a line are ignored.

We define the following macros in the grammar:

```
: macro(lines of $1)
macro: $1 | macro newline $1

: macro(string of $1)
macro: $1 | macro $1
```

The tokens `BLANK`, `CHARACTER` and `ALPHANUMERIC` in the grammar stand for a blank space, any character and any alphanumeric character respectively.

```
                STMPROGRAM = HEADER lines of PROCESS STARTPROCESS
                HEADER = program BLANK NAME
                NAME = string of ALPHANUMERIC
                PROCESS = PROCESSHEADER lines of COMMAND PROCESSEND
                PROCESSHEADER = process NAME
                COMMAND = NC | GC | SC | string of BLANK COMMAND | COMMENT
                PROCESSEND = GC | SC | string of BLANK PROCESSEND | COMMENT
                STARTPROCESS = start BLANK NAME
                COMMENT = % string of CHARACTER
create:         NC =  create BLANK ^ NAME
inc:            NC = ^ NAME BLANK ++
dec:            NC = ^ NAME BLANK --
copy:           NC = ^ NAME =copy BLANK of BLANK ^ NAME
copyfromcore:   NC =  NAME =copy BLANK of BLANK ^ NAME
copytocore:     NC = ^ NAME =copy BLANK of BLANK NAME
fields:         NC = ^ NAME =fields BLANK of BLANK ^ NAME
external:       NC =  external: BLANK NAME (^ NAME )
externalref:    NC =  external: BLANK ^ NAME (^ NAME )
if:             NC =  if BLANK NAME BLANK goto BLANK NAME
get:            NC = ^ NAME =^ NAME . NAME
refget:         NC = ^ NAME =^ NAME .^ NAME
check:          NC = ^ NAME =(^ NAME ==^ NAME )
exist:          NC = ^ NAME =exist( NAME . NAME )
existref:       NC = ^ NAME =exist(^ NAME .^ NAME )
set:            NC = ^ NAME . NAME =^ NAME
refset:         NC = ^ NAME .^ NAME =^ NAME
setconst:       NC = ^ NAME . NAME =const BLANK NAME
setconstref:    NC = ^ NAME .^ NAME =const BLANK ^ NAME
clean:          NC =  clean BLANK  ^ NAME
function:       NC =  function: BLANK NAME (^ NAME ,^ NAME )
transportin:    NC =  in BLANK ^ NAME BLANK as BLANK NAME
```

transportout:	NC =	`out BLANK ^ NAME BLANK as BLANK NAME`
transportrefin:	NC =	`in BLANK ^ NAME BLANK as BLANK ^ NAME`
transportrefout:	NC =	`out BLANK ^ NAME BLANK as BLANK ^ NAME`
string:	NC =	`^ NAME=‘ NAME ‘`
vcopy:	NC =	`^ NAME =vcopy BLANK of BLANK ^ NAME`
goto:	GC =	`goto BLANK NAME`
move:	GC =	`goto BLANK ^ NAME`
stop:	SC =	`stop`

B Tables of nodes with constant meaning

In these tables, the column ‘symbol’ contains the semantic meaning, i.e., the node, the ochar-, rchar-, schar- and xchar-column are the representation of the nodes in charcode text, and the column ‘octal’ contains the byte. We use the rchars `'T`=⊤ and `'F`=⊥ for the logical constants ‘true’ and ‘false’

To keep the charcode readable, the charcodes /144, /177, /244, /277, /344 and /377 are not used.

symbol	ochar	octal
0	0	/000
1	1	/001
2	2	/002
3	3	/003
4	4	/004
5	5	/005
6	6	/006
7	7	/007
8	8	/010
9	9	/011
A	A	/012
B	B	/013
C	C	/014
D	D	/015
E	E	/016
F	F	/017
G	G	/020
H	H	/021
I	I	/022
J	J	/023
K	K	/024
L	L	/025
M	M	/026
N	N	/027
O	O	/030
P	P	/031
Q	Q	/032
R	R	/033
S	S	/034
T	T	/035
U	U	/036
V	V	/037

symbol	ochar	octal
W	W	/040
X	X	/041
Y	Y	/042
Z	Z	/043
newline(↵)	↵	/044
a	a	/045
b	b	/046
c	c	/047
d	d	/050
e	e	/051
f	f	/052
g	g	/053
h	h	/054
i	i	/055
j	j	/056
k	k	/057
l	l	/060
m	m	/061
n	n	/062
o	o	/063
p	p	/064
q	q	/065
r	r	/066
s	s	/067
t	t	/070
u	u	/071
v	v	/072
w	w	/073
x	x	/074
y	y	/075
z	z	/076
blank( )	␣	/077

Table 10: The ordinary characters

symbol	rchar	octal
,	'0	/100
.	'1	/101
:	'2	/102
;	'3	/103
!	'4	/104
?	'5	/105
(	'6	/106
)	'7	/107
-	'8	/110
_	'9	/111
Ä	'A	/112
Å	'B	/113
Ç	'C	/114
\	'D	/115
É	'E	/116
⊥	'F	/117
&	'G	/120
†	'H	/121
^	'I	/122
Ø	'J	/123
È	'K	/124
Ł	'L	/125
□	'M	/126
Ñ	'N	/127
Ö	'O	/130
£	'P	/131
≠	'Q	/132
€	'R	/133
$	'S	/134
⊤	'T	/135
Ü	'U	/136
Ê	'V	/137

symbol	rchar	octal
⌊	'W	/140
⌈	'X	/141
¥	'Y	/142
§	'Z	/143
	'↵	/144
ä	'a	/145
å	'b	/146
ç	'c	/147
/	'd	/150
é	'e	/151
<	'f	/152
>	'g	/153
ħ	'h	/154
ı	'i	/155
ø	'j	/156
è	'k	/157
ł	'l	/160
−	'm	/161
ñ	'n	/162
ö	'o	/163
+	'p	/164
=	'q	/165
%	'r	/166
ß	's	/167
*	't	/170
ü	'u	/171
ê	'v	/172
⌋	'w	/173
⌉	'x	/174
~	'y	/175
©	'z	/176
	'␣	/177

Table 11: The regular characters

symbol	schar	octal
$\emptyset$	“0	/200
$\vert$	“1	/201
$\langle$	“2	/202
$\rangle$	“3	/203
$\{$	“4	/204
$\}$	“5	/205
$[$	“6	/206
$]$	“7	/207
∞	“8	/210
$\Vert$	“9	/211
$\forall$	“A	/212
$\Rightarrow$	“B	/213
$\Leftarrow$	“C	/214
$\Leftrightarrow$	“D	/215
$\exists$	“E	/216
$\equiv$	“F	/217
$\gg$	“G	/220
$\succ$	“H	/221
$\urcorner$	“I	/222
$\doteq$	“J	/223
$\approx$	“K	/224
$\ll$	“L	/225
$\simeq$	“M	/226
∇	“N	/227
$\cong$	“O	/230
$\prod$	“P	/231
$\uparrow$	“Q	/232
$\downarrow$	“R	/233
$\mapsto$	“S	/234
$\bigotimes$	“T	/235
$\bigoplus$	“U	/236
$\bigvee$	“V	/237

symbol	schar	octal
$\bigwedge$	“W	/240
$\times$	“X	/241
$\bigcup$	“Y	/242
$\bigcap$	“Z	/243
	“↵	/244
$\rightarrow$	“a	/245
$\leftarrow$	“b	/246
$\leftrightarrow$	“c	/247
$\in$	“d	/250
$\ni$	“e	/251
$\notin$	“f	/252
$\geq$	“g	/253
$\prec$	“h	/254
$\int$	“i	/255
$\subseteq$	“j	/256
$\supseteq$	“k	/257
$\leq$	“l	/260
$\cdot$	“m	/261
$\neg$	“n	/262
$\circ$	“o	/263
$\pm$	“p	/264
$\mp$	“q	/265
∂	“r	/266
$\sum$	“s	/267
$\otimes$	“t	/270
$\oplus$	“u	/271
$\vee$	“v	/272
$\wedge$	“w	/273
$\times$	“x	/274
$\cup$	“y	/275
$\cap$	“z	/276
	“␣	/277

Table 12: The special characters

use	xchar	octal
left quote(`)	‘0	/300
right quote(')	‘1	/301
double quote(")	‘2	/302
hash(#)	‘3	/304
at(@)	‘4	/305
tabulator	‘5	/306
end of file	‘6	/307

Table 13: The auxiliary characters

C The MATLAB implementation

In our MATLAB implementation the STM memory is a sparse matrix, zero entries `SM(x,y)` of the SM correspond to nonexisting nodes `x.y`. Counters are represented by an index. The node representing the number n has the index $2n+1$.

In this appendix we give the essential part of the MATLAB implementation of the STM, i.e., the part that reflects the USTM. Commands that are not simulated by the USTM but are just called via the USTM are displayed only in abbreviated form. Printing these routines in full would not improve understanding of the STM since nothing in the USTM corresponds to them.

A few variables that occur in the MATLAB code shall be described: `SM` is the MATLAB representation of the semantic matrix, a sparse matrix of arbitrary size. `LINKS` is a matrix of the same size as `SM`, needed for garbage collection. `LABEL` is a cell array containing the names of each node. In particular, it contains all constant nodes, and the nodes with fixed meaning for execution, given in Table 14. The nodes with index 2 up to 18 in the table below are reserved for flow control, the nodes with index greater or equal 30 are reserved for command names.

index	label
2	depth
4	context
6	core
8	corelist
10	lib
12	line
14	program
16	process
18	result
20	true
22	false
24	node24
26	node26
28	node28
30	external
32	externalref
34	move
36	goto
38	fields
40	copy
42	copytocore
44	copyfromcore
46	vcopy
48	set
50	get

index	label
52	refset
54	refget
56	stop
58	create
60	if
62	check
64	exist
66	existref
68	setconst
70	setconstref
72	inc
74	dec
76	start
78	function
80	clean
82	string
84	transportin
86	transportout
88	transportrefin
90	transportrefout
92	node92
94	node94
96	node96
98	node98
100	node100

Table 14: Nodes with fixed meaning

C1. Functions simulated by the USTM:

```
function run(var1,var2,var3)
    % var1=programm, var2=context, var3=lib
    global SM actual_core errorflag LABEL  oldsecond n_process n_lib;
    global n_corelist n_line n_start n_program n_depth n_core
    global n_context n_result n_true n_false;
    errorflag=0;
    n_depth = 2 ;
    n_context = 4 ;
    n_core = 6 ;
    n_corelist = 8 ;
    n_lib = 10 ;
    n_line = 12 ;
    n_program = 14 ;
    n_process = 16 ;
    n_result = 18 ;
    n_true = 20;
    n_false = 22;
    n_external = 30 ;
    n_externalref = 32 ;
    n_move = 34 ;
    n_goto = 36 ;
    n_fields = 38 ;
    n_copy = 40 ;
    n_copytocore = 42 ;
    n_copyfromcore = 44 ;
    n_vcopy = 46 ;
    n_set = 48 ;
    n_get = 50 ;
    n_refset = 52 ;
    n_refget = 54 ;
    n_stop = 56 ;
    n_create = 58 ;
    n_if = 60 ;
    n_check = 62 ;
    n_exist = 64 ;
    n_existref = 66 ;
    n_setconst = 68 ;
    n_setconstref = 70 ;
    n_inc = 72 ;
    n_dec = 74 ;
    n_start = 76 ;
    n_function = 78 ;
    n_clean = 80 ;
    n_string = 82 ;
    n_transportin = 84 ;
    n_transportout = 86 ;
    n_transportrefin = 88 ;
    n_transportrefout = 90 ;
    semset(labsearch(var3),labsearch(var1),labsearch([var3 '.' var1]));
    actual_prog=SM(labsearch(var3),labsearch(var1));
    actual_core=nextfree;
    depth=3;
    oldsecond=0;
% global frame
    semset(n_corelist,depth,actual_core);
    semset(n_corelist,n_depth,depth);
```

```
% local frame
    semset(actual_core,n_context,labsearch(var2));
    semset(actual_core,n_program,actual_prog);
    semset(actual_core,n_lib,labsearch(var3));
61  semset(actual_core,n_core,actual_core);
    semset(actual_core,n_line,3);
    if (SM(actual_prog,n_start) ~= 0)
        semset(actual_core,n_process,SM(actual_prog,n_start));
    else
66      semset(actual_core,n_process,actual_prog);
    end
while (SM(n_corelist,n_depth) >= 3 && errorflag==0)
        depth=SM(n_corelist,n_depth);
        actual_core=SM(n_corelist,depth);
71      process=SM(actual_core,n_process);
        progline=SM(actual_core,n_line);
        focus=SM(process,progline);
    if (focus == 0)
        disp('ERROR called by run.m: focus=0 !');
76      errorflag=1;
        return;
    end
disp(labask(focus));
    comm=SM(focus,3);
81  actual_prog=SM(actual_core,n_program);
    actual_lib=SM(actual_core,n_lib);
    switch comm
        case n_external
        stm_external(SM(focus,5),SM(focus,7));
86      case n_externalref
        stm_externalref(SM(focus,5),SM(focus,7));
        case n_transportin
        stm_transportin(SM(focus,5),SM(focus,7),actual_lib);
        case n_transportout
91      stm_transportout(SM(focus,5),SM(focus,7),actual_lib);
        case n_transportrefin
        stm_transportrefin(SM(focus,5),SM(focus,7),actual_lib);
        case n_transportrefout
        stm_transportrefout(SM(focus,5),SM(focus,7),actual_lib);
96      case n_fields
        stm_fields(SM(focus,5),SM(focus,7));
        case n_copytocore
        stm_copytocore(SM(focus,5),SM(focus,7));
        case n_copyfromcore
101     stm_copyfromcore(SM(focus,5),SM(focus,7));
        case n_goto
        stm_goto(SM(actual_prog,SM(focus,5)));
        case n_refset
        stm_refset(SM(focus,5),SM(focus,7),SM(focus,9));
106     case n_refget
        stm_refget(SM(focus,5),SM(focus,7),SM(focus,9));
        case n_stop
        stm_stop();
        case n_create
111     stm_create(SM(focus,5));
        case n_if
```

```
            jumpmark=SM(SM(actual_core,n_program),SM(focus,7));
            stm_if(SM(focus,5),jumpmark);
            case n_check
116         stm_check(SM(focus,5),SM(focus,7),SM(focus,9));
            case n_existref
            stm_exist(SM(focus,5),SM(actual_core,SM(focus,7)),...
                SM(actual_core,SM(focus,9)));
            case n_set
121         stm_set(SM(focus,5),SM(focus,7),SM(focus,9));
            case n_get
            stm_get(SM(focus,5),SM(focus,7),SM(focus,9));
            case n_setconst
            stm_setconst(SM(focus,5),SM(focus,7),SM(focus,9));
126         case n_setconstref
            stm_setconstref(SM(focus,5),SM(focus,7),SM(focus,9));
            case n_inc
            stm_inc(SM(focus,5));
            case n_exist
131         stm_exist(SM(focus,5),SM(focus,7),SM(focus,9));
            case n_copy
            stm_copy(SM(focus,5),SM(focus,7));
            case n_function
            stm_function(SM(focus,5),SM(focus,7),SM(focus,9));
136         case n_dec
            stm_dec(SM(focus,5));
            case n_move
            stm_move(SM(actual_prog,SM(actual_core,SM(focus,5))));
            case n_clean
141         stm_clean(SM(focus,5));
            case n_vcopy
            stm_vcopy(SM(focus,5),SM(focus,7));
            case n_string
            stm_string(SM(focus,5),SM(focus,7));
146         otherwise
            disp(['ERROR called by run.m: unknown command ' LABEL{comm} ]);
            errorflag=1;
            return;
        end
151 end
```

```
  function stm_function(v2,v3,v4)
  % ^v2 = programm, ^v3 = context, ^v4 = lib
3 global SM n_line n_depth actual_core n_core n_context
  global n_start n_process n_program errorflag n_corelist n_lib
  incfocus()
  var2=v2;
  var3=SM(actual_core,v3);
8 var4=SM(actual_core,v4);
  % global frame
  depth=SM(n_corelist,n_depth);
  depth=depth+2;
  semset(n_corelist,n_depth,depth);
13 actual_core=nextfree;
  semset(n_corelist,depth,actual_core);
  % local frame
  semset(actual_core,n_context,var3);
```

```
   semset(actual_core,n_lib,var4);
18 semset(actual_core,n_core,actual_core);
   if (var2==0)
   disp('ERROR called by stm_function.m: program not found!');
   errorflag=1;
   return;
23 end
   actual_prog=SM(var4,var2);
   if (actual_prog==0)
   disp('ERROR called by stm_function.m: ^arg2 does not exist!');
   errorflag=1;
28 return;
   end
   semset(actual_core,n_program,actual_prog);
   if (SM(actual_prog,n_start)~=0)
       semset(actual_core,n_process,SM(actual_prog,n_start));
33 else
       semset(actual_core,n_process,actual_prog);
   end
   semset(actual_core,n_line,3);
   disp([' >>> starting function: ' labask(actual_prog) ...
38     '  in core: ' labask(actual_core) ]);
```

```
 1 function stm_set(var1,var2,var3)
   global SM actual_core
   var4=actual_core;
   semset(SM(var4,var1),var2,SM(var4,var3));
   incfocus();
```

```
   function stm_get(var1,var2,var3)
   global SM actual_core
 3 var4=actual_core;
   semset(var4,var1,SM(SM(var4,var2),var3));
   incfocus();
```

```
   function stm_move(var1)
   global actual_core n_process n_line
 3 semset(actual_core,n_process,var1);
   semset(actual_core,n_line,3);
```

```
   function stm_stop()
   global SM n_depth n_corelist n_result n_context
   depth=SM(n_corelist,n_depth);
   coreofcallee=SM(n_corelist,depth);
 5 depth=depth-2;
   semset(n_corelist,n_depth,depth);
   if depth>3
       coreofcaller=SM(n_corelist,depth);
       outcore=SM(coreofcallee,n_result);
10     if outcore ~=0
           nameofout=SM(coreofcallee,n_context);
           semset(coreofcaller,nameofout,outcore);
       end
   end
15 incfocus()
   io_coredelete(coreofcallee);
```

```
1 function stm_string(v1,v2)
   global VALUE SM actual_core;
      var1= SM(actual_core,v1);
      VALUE{var1}=VALUE{v2};
      incfocus();

  function stm_copyfromcore(var1,var2)
      global SM actual_core
      v2=SM(actual_core,var2);
4     io_copy(var1,v2);

  function stm_copytocore(var1,var2)
      global SM actual_core
      v1=SM(actual_core,var1);
4     io_copy(v1,var2);

  function stm_external(var2,v3)
                  %program, context
      global errorflag actual_core SM
4     var3=SM(actual_core,v3);
      if (var2==0)
      disp('ERROR called by stm_external.m: ^arg2 does not exist!')
      errorflag=1;
      return;
9     end
  prog=labask(var2);
  disp(['=> ' prog ' called']);
  feval(prog,var3);
  incfocus();
```

The MATLAB implementation corresponding to the USTM has 247 lines, compared to the 232 lines of the USTM. But in this comparison, consider that the MATLAB implementation covers error messages and stores information for garbage collection. Also consider that the MATLAB implementation contains 9 lines that display information on the screen, which is not part of the action of a STM.

C2. Functions not simulated by the USTM:

```
function stm_check(var1,var2,var3)
% sets ^var1="true" if ^var2=^var3, and ^var1="false" otherwise
% 14 lines, calls subroutines semset and incfocus

function stm_refget(var1,var2,var3)
% sets ^var1=^var2.^var3
% 5 lines of code, calls subroutines semset and incfocus

function stm_refset(var1,var2,var3)
% sets ^var1.^var2=^var3
% 5 lines of code, calls subroutines semset and incfocus

function stm_goto(var1)
% sets the focus to the forst line of process var1
% 4 lines of code, calls subroutine semset

function stm_externalref(v2,v3)
```

```
% calls the external processor ^v2 with context ^v3
% 13 lines of code

function stm_transportin(var1,var2)
% imports VALUE(^var1) into var1 using protocol var2
% 5 lines of code, calls subroutines according to the protocol

function stm_transportout(var1,var2)
% exports var1 into VALUE(^var1) using protocol var2
% 5 lines of code, calls subroutines according to the protocol

function stm_transportrefin(var1,var2)
% imports VALUE(^var1) into var1 using protocol ^var2
% 6 lines of code, calls subroutines according to the protocol

function stm_transportrefout(var1,var2)
% exports var1 into VALUE(^var1) using protocol ^var2
% 6 lines of code, calls subroutines according to the protocol

function stm_setconst(var1,var2,var3)
% writes var3 to ^var1.var2 and produces error if var3 is not a constant
% 11 lines of code, calls subroutine labask, charsearch and semset

function stm_setconstref(var1,var2,var3)
% writes ^var3 to ^var1.^var2 and produces error if ^var3 is not a constant
% 15 lines of code, calls subroutine labask, charsearch and semset

function stm_copy(var1,var2)
% makes copy of the DAG with root var1 in the node var2
% 5 lines of code, calls subroutine io_copy and incfocus

function stm_inc(var1)
% increases ^var1 by 1
% 5 lines of code, calls subroutines semset and incfocus

function stm_dec(var1)
% decreases ^var1 by 1
% 10 lines of code, calls subroutines semset and incfocus

function stm_exist(var1,var2,var3)
% writes "true" to ^var1 if var2.var3 exists, and "false" otherwise
% 9 lines of code, calls subroutines semset and incfocus

function stm_if(var1,var2)
% sets focus to process var2 if ^var1="true", proceeds one line if ^var1="false"
% 11 lines of code, calls subroutine semset

function stm_clean(arg1)
% deletes all edges reachable only from var1
% 4 lines of code, calls subroutines io_coredelete and incfocus

function stm_create(var1)
% ^var1 is assigned a new node
% 7 lines of code, calls subroutines nextfree (12 lines), labput and incfocus

function stm_fields(v1,var2)
```

```
% writes the edges from ^var2 into ^v1.2, ^v1.3, ^v1.4 ... ^v1.n and n into ^v1.1
% 15 lines of code, calls subroutines semset and incfocus

function stm_vcopy(var1,var2)
% sets VALUE(^var1)=VALUE(^var2)
% 6 lines of code, calls subroutine incfocus

function erg=labsearch(var1)
% erg is the integer corresponding to the label var1.
% 15 lines of code, calls subroutine labput

function labput(var1,var2)
% integer var1 gets label var2
% 4 lines of code

function erg=labask(var1)
% erg is the label of the integer var1
% 3 lines of code

function erg=charsearch(var1)
% erg is the index of the node with charcode var1
% 14 lines of code

function io_copy(var1,var2)
% makes a copy of the DAG with root var1 to var2
% 126 lines of code

function io_coredelete(var1)
% deletes all nodes reachable only from var1
% 35 lines of code

function semset(var1,var2,var3)
% sets var1.var2=var3 and takes care of garbage collection in the SM
% 12 lines of code

function erg=nextfree
% creates a node and returns the number representing this node
% 12 lines of code

function incfocus()
% sets the focus to the next line
% 3 lines of code
```

The MATLAB implementation of functions not simulated by the STM has about 390 lines.

There are auxiliary functions for prettyprinting, protocols, parsing the STM program and writing it into the SM, displaying the SM, setting up the environment etc. altogether about 820 lines. So the complete MATLAB implementation of the whole STM adds up to about 1500 lines without comments and blank lines.

References

[1] O. Beuret and M. Tomassini. Behaviour of Multiple Generalized Langton's Ants. In *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*. MIT Press, 1997.

[2] Daniel I. Cohen. *Introduction to computer theory.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[3] J.E. Hopcroft, J.D. Ullman, and A.V. Aho. *The design and analysis of computer algorithms.* Addison-Wesley, Boston, MA, USA, 1975.

[4] S. Jefferson and D.P. Friedman. A simple reflective interpreter. *LISP and symbolic computation*, 9(2):181–202, 1996.

[5] C. Langton. Studying artificial life with cellular automata. *Physica*, D 22:120–149.

[6] O. Lassila, R.R. Swick, et al. Resource Description Framework (RDF) Model and Syntax Specification. 1999.

[7] F. Manola, E. Miller, et al. RDF Primer. *W3C Recommendation*, 10, 2004.

[8] H. Marxen and J. Buntrock. Attacking the Busy Beaver 5. *Bulletin of the EATCS*, 40:247–251, 1990.

[9] Marvin Minsky. Size and structure of universal Turing machines using tag systems. *Proceedings of Symposia in Pure Mathematics*, 5.

[10] T. Neary and D. Woods. Small fast universal Turing machines. *Theoretical Computer Science*, 362(1–3):171–195, 2006.

[11] Piergiorgio Odifreddi. *Classical Recursion Theory.* North Holland, Amsterdam, New York, Oxford, 1999.

[12] R. Penrose and M. Gardner. *The emperor's new mind.* Oxford Univ. Press, 1989.

[13] T. Rado. On non-computable functions. *The Bell System Technical Journal*, 41(3):877–884, 1962.

[14] R. H. Richens. Preprogramming for mechanical translation. *Mechanical Translation*, 3(1):20–28, 1956.

[15] Raphael M. Robinson. Minsky's Small Universal Turing Machine. *International Journal of Mathematics*, 2(5):551–562.

[16] M.A. Rodriguez and J. Bollen. Modeling Computations in a Semantic Network. *CoRR, abs/0706.0022*, 2007.

[17] Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967.

[18] Matthias Scheutz, editor. *Computationalism: New Directions.* MIT Press, Cambridge, MA, 2002.

[19] J.R. Shoenfield. *Recursion theory.* Springer-Verlag New York, 1993.

[20] J.F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* MIT Press, 2000.

[21] Julio Tanomaru. Evolving Turing Machines from Examples. *Lecture Notes In Computer Science*, pages 167–182, 1998.

[22] P.D. Terry. *Compilers and compiler generators: an introduction with C++*. International Thomson Computer Press, 1997.

[23] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42(2):230–265.