A semantic virtual machine

Arnold Neumaier
Peter Schodl

Fakultät für Mathematik, Universität Wien
Nordbergstr. 15, A-1090 Wien, Austria
WWW: http://www.mat.univie.ac.at/∼neum/FMathL

Abstract

A semantic virtual machine (SVM) 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 SVM, its memory management and flow control, and shows how a semantic virtual machine can simulate any ordinary Turing machine. Analogous to a universal Turing machine, we give a universal semantic virtual machine (USVM), which is a special SVM that can simulate every SVM. The USVM serves both as a self-contained semantic explanation of many aspects of the SVM, and as a check that an SVM implementation works correctly.

Contents

1 Introduction

A **Turing machine**, introduced originally in 1936 by TURING [18], 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 [13] or AHO et al. [2] or almost any other computability book; see also Section 9 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 [10]), 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 virtual machine** (SVM):

Concerning item 1, the SVM is able to execute an **SVM 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 SVM represents information by semantic relations between **objects** represented by a binary operator, the **semantic memory**. Using the semantic memory, complex relations can be represented in a simple and user-friendly way, and be visualized as a directed, labelled graph. Thus an SVM allows the expression of semantics in a very natural form.

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

That the SVM is as least as powerful as an ordinary Turing machine is shown in Section 9, but we give the SVM 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 SVM is no longer equivalent to an ordinary Turing machine, or in other words, not every SVM program, regarded as a function on the context, is **Turing computable**. For example, external processors might have access to the system clock etc. However, the main reason for enabling the SVM to call external processors is higher performance and reusability of trusted algorithms. The SVM command that calls external processes is essentially a foreign function interface (FFI) of the SVM.

A cornerstone in the creation of the SVM is the proof that the SVM is powerful enough to simulate itself in a very simple way. This is done by giving an SVM program that can

simulate every other SVM program. Since this is analogous to the role of a universal Turing machine, we call this program the **universal semantic virtual machine** (USVM).

The USVM is a program short and transparent enough to be checked by hand. It has only 166 lines of code, see Section 10 (compare this, e.g., to the reflective interpreter by Jefferson & Friedman in [3], which has 273 lines). The USVM gives us a possibility to check many aspects of the SVM for correctness: Once one has convinced oneself of the correctness of the USVM, one can make the implementation of the SVM on some physical device also trustworthy by checking empirically (or, in principle, in a formal way) that any SVM program executed by the implemented SVM produces the same output as in the case when the USVM simulates this program.

All this makes the SVM a semantically self-contained, transparent and easily usable tool that can be a trustworthy foundation for any computer system that deals with semantic content.

Contents. Section 2 formally defines the framework we use to store information. Section 3 describes the SVM and its memory in detail, and Section 4 defines the programming language of the SVM. Having an assembler-like language for the SVM instead of a transition table like the Turing machines makes programming much more convenient, but it requires the storage of information for flow control, described in Section 5. The features of the SVM that allow it to make use of other algorithms on the physical device are discussed in Section 6. In Section 7 we give an operational semantic of the SVM and in Section 8 we describe the commands in a more informal way and give some examples. In Section 9 we show how the SVM can simulate an ordinary TM. The USVM, a special SVM program that can simulate every other SVM program, is given in Section 10.

2 Definition of the framework

We define the abstract data structure we use to represent mathematics.

It can be regarded as a special case of a **semantic network**, introduced by Richens [11] in 1956. This and akin concepts are discussed in detail by Sowa [17]. Also, it is inspired by, and representable in, the semantic web [5]. A standardized and widely used example of a semantic net with the aim to be used in the World Wide Web is the Resource Description Framework (RDF), described by Manola et al. [6] and specified by Lassila et al. [4].

2.1 The semantic memory

There is an unlimited number of **objects**, but only finitely many of them are represented explicitly in stored memory. Objects can be compared for equality, which is an equivalence relation. On the metalevel, we refer to objects by strings not beginning with a hash (#); different objects are referred to by different strings. `Empty` is an object. **Object variables** are variables in the usual sense, ranging over the set of objects. We refer to object variables

via a string beginning with a hash (#) followed by some alphanumeric string. For example, in the statement

#name.type = `String` for every object #name representing a string,

`type` and `Name` are specific objects, and #name is a variable in the same sense as x is a variable in

x^2 is even for every even integer x.

Usually, we will use suggestive strings for variables, e.g., we use #handle or #h for an object that is intended to be a handle.

A **semantic mapping** (abbreviated SM) assigns to any two objects #h and #f a unique object #h.#f such that

if #f = `Empty` or #h = `Empty` then #f.#h = `Empty`.

A **semantic unit** (short **sem**) is an equation of the form #h.#f = #e with nonempty #h, #f, and #e; we call #h the **handle**, #f the **field**, and #e the **entry** of the sem. The **constituents** of an object #a are the sems in which #a is the handle.

Semantic mappings are used to store mathematics, but to be able to alter the data we need a dynamical framework. The semantic mapping that changes over time (formally, a semantic mapping valued function of time) is called the **semantic memory**.

A **position** is a pair (#h/#f) consisting of two objects #h and #f. We call #h the **handle**, #f the **field** and #h.#f the **entry** of (#h/#f). This position is called **occupied** if #h.#f is not `Empty`.

We say that the sem #d.#e=#f **follows** the sem #a.#b = #c if #d = #c. Using a left-associative notation, we then write #a.#b.#e = #f; thus #a.#b.#e stands for (#a.#b).#e. This notation naturally extends to more dots.

A short-hand notation for k repetitions ($k = 0, 1, 2, \ldots$) of a field: #a.$\underbrace{\text{#b.}\cdots\text{.#b}}_{k \text{ times}}$.#e is

written as #a.#b^k.#e.

A **path of sems** starting at #h and ending at #e is a sequence of sems such that the first sem has the handle #h, each later sem follows the previous one, and the last sem has entry #e, and no sem has the field `type`. An object #e is **reachable** from a handle #h if there is some path of sems starting at #h and ending in #e. A sem is **reachable** from a handle #h if there is some path of sems starting at #h that contains that sem. A position is **reachable** from a handle #h if the handle of that position is an object reachable from #h.

If the set of sems reachable from an object #h is finite, then the set of sems reachable from #h defines the **record** with handle #h.

Clearly, a SM allows one to construct arbitrarily complex records. In contrast to records in programming languages such as Pascal, records in a SM may contain cycles. Indeed, backreferences are an important part of the design of the type system; for example, they allow labelled context-free grammars to be defined as type systems.

2.2 Illustration by semantic graphs

For graphical illustration of a semantic mapping, we will interpret a sem #a.#b=#c as an edge with label #b from node #a to node #c of a directed labeled graph, called a

semantic graph. Objects may, but need not have **external values**, i.e., data of arbitrary form, associated with the object, but stored outside the semantic memory. We refer to the value of an object #obj by VALUE(#obj). In a semantic graph, objects that have an external value are printed as a box containing that value. For better readability we use dashed edges for edges labeled with type, since these constituents have importance for the typing, and bold edges for edges labeled with next, since this makes linked lists more readable. Different nodes of the semantic graph may represent the same object. For example, the information $\frac{12}{4} = 3$ may be represented as a list of sems as fiven in Figure 1, or equivalently as the semantic graph in Figure 2.

```
$380.type=Binary         $370.type=Fraction       VALUE($244) = 12
$380.lhs=$370            $370.num=$244            VALUE($246) = 3
$380.rhs=$246            $370.denom=$248          VALUE($248) = 4
$380.relation=Equal      $244.type=Integer
$246.type=Integer        $248.type=Integer
```

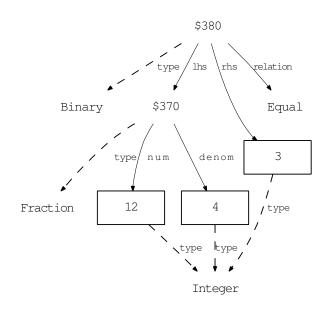Figure 1: A list of sems and values

Figure 2: A semantic graph

3   The semantic virtual machine

A **semantic virtual machine** (SVM) is a machine manipulating semantic information in a semantic memory. Independent of the interpretation of the semantic memory either as semantic mapping or as graph, we will refer to it as the **memory** of the SVM.

Since there are equivalent formulations of Turing machines which use a 2-dimensional memory instead of the tape (a proof is given by COHEN [1]) 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 SVM to manipulate its external environment, the scope of an SVM becomes strictly bigger cf. Section 6.

To allow this, objects can have an **external value**, which is not part of the SM but arbitrary data stored outside the semantic memory (on the physical machine the SVM is implemented on) and associated to the object. This external storage is handled exclusively by **external processors**, i.e., algorithms executed by the physical machine. External values are discussed in more detail in Section 6.

The memory of the SVM contains the **program** to execute, and the information about **flow control** as well, all represented via a semantic mapping. To enable the processing of more than one program in the same memory, each execution of the SVM has its own **core**, i.e., a record reserved for the input, the output and temporary data.

Since the core is the most important record for a program, will simplify the notation for it: We use the **caret ^** to abbreviate reference to the core of the execution under consideration. Hence `^a` means `#core.a`, where `#core` is the core of the execution under consideration. The caret binds stronger than the semantic mapping, hence `a.^b` means `a.(#core.b)`.

To start processing a program, the SVM needs to know the object that contains the program, and the object that serves as the core. Therefore the call of an SVM program has two arguments: the name of the program and the core.

4   The SVM programming language

The most elementary part of the SVM programming language is a command. There are 24 different commands; a list of the commands and their action is given in Section 8. The commands fall into four groups: commands that structure the program but have no influence on the memory at runtime, commands for flow control, assignments, which make alterations in the memory of the SVM, and commands handling or external values.

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

Before describing the commands in detail, we say something about the structure of the language and external processors and values. This is the content of this and the next section.

The SVM programming language has the reserved names

```
program          process          start
```

for structuring the program, and the reserved names

```
check        fields       refset       refin
create       get          set          refout
exist        goto         setconst     unset
existref     if           stop
external     move         in
externalref  refget       out
```

for commands making certain alterations in the memory or in flow control. The meaning of these names will be discussed in Section 5. The name `type` should also not be used as a name in a SVM program to prevent collusion with the typing system [14] which is not a part of the SVM but based on it.

All other names and more general alphanumeric strings may be used as variables for objects.

The SVM programming language is the lowest level of a fully comfortable programming language that we are in the process of developing, see [9] and [15].

4.1 The grammar of the SVM programming language

The complete grammar of the SVM programming language si defined by the following grammar, using 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. The token COMMENT is a string beginning with a percent sign (%) and not containing a newline (\n).

STMPROGRAM	→	HEADER lines of PROCESS STARTPROCESS
HEADER	→	`program(` 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(` NAME `)`
NC	→	NAME = `check(` NAME `,` NAME `)`
NC	→	`create(` NAME `)`
NC	→	NAME =`exist(` NAME `,` NAME `)`
NC	→	NAME =`existref(` NAME `,` NAME `)`
NC	→	NAME =`external(` NAME `,` NAME `)`
NC	→	NAME =`externalref(` NAME `,` NAME `)`
NC	→	NAME =`fields(` NAME `)`
NC	→	NAME =`get(` NAME `,` NAME `)`
NC	→	`goto(` NAME `)`
NC	→	`if(` NAME `,` NAME `)`
NC	→	`move(` NAME `)`
NC	→	NAME =`refget(` NAME `,` NAME `)`
NC	→	`(` NAME `,` NAME `)`=`refset(` NAME `)`
NC	→	`(` NAME `,` NAME `)`=`set(` NAME `)`
NC	→	`(` NAME `,` NAME `)`=`setconst(` NAME `)`
NC	→	NAME =`in(` NAME `,` NAME `)`
NC	→	NAME =`out(` NAME `,` NAME `)`

$$
\begin{array}{rcl}
\text{NC} & \rightarrow & \text{NAME =refin(NAME , NAME)} \\
\text{NC} & \rightarrow & \text{NAME =refout(NAME , NAME)} \\
\text{NC} & \rightarrow & \text{unset(NAME , NAME)} \\
\text{SC} & \rightarrow & \text{stop}
\end{array}
$$

4.2 Representation of SVM programs in the SM

A **process** is a sequence of commands, beginning with the command `process(#proc)`. Every process ends with a command that either halts the SVM or calls another process.

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

Each process is represented in the memory by a linked list of commands, and the first command of each process is accessible by:

```
#program.#processname = #process
```

where `#program` is the record containing the program, `#processname` is the name of the process, and `#process` is a linked list of the commands of process `#processname`.

Each command is represented in the memory by a record `#command` with

```
#command.#part = #object
```

```
#command.next = #nextcommand
```

and `#nextcommand` is the command in the line below this command. The object `#part` is one of the following: `comm` refers to the name of the command, `arg1` to the first argument, `arg2` to the second argument, and `arg3` to the third argument. Since not all commands have three arguments, some of these may be empty.

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

The SVM is untyped. However, to further specify the representation of SVM programs in the SM, we give typesheets (see [14]) for SVM programs:

```
SVM::

SvmProgram:
  allOf> start = Object
         processes = Processes
  nothingElse>

Processes:
  someOfType> Object = SvmCommand

CommandName:
  atomic> check, create, exist, existref, external, externalref
  atomic> fields, get, goto, if, move, refget, refset, set
  atomic> setconst, transportin, transportout, transportrefin
```

8

```
    atomic> transportrefout, unset, stop

SvmCommand:
  allOf> comm = CommandName
  optional> arg1 = Object
            arg2 = Object
            arg3 = Object
            next = SvmCommand
  nothingElse>
```

The semantic graph in Figure 3 is the record that represents the SVM program `copyFields` as given in text form in Section 8, page 16. Note that for transparency, the sems with field `type` are not printed.

Figure 3: Representation of an SVM program in the SM

5 Flow control

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

The SVM command currently executed is called the **focus**, it may change after each program execution. A process can be entered only at its first command, but it is possible to leave a process before its last command line is reached.

During runtime, the focus is represented in the object `^focus`. Setting the entry of `^focus`

to `^focus.next` means to proceed one line forward in the programm. Setting the entry of `^focus` to `#program.#process` as done by the `goto` and `if` command, sets the focus to the first line of the process `#process`.

It is assumed that `^core` always contains the current core, i.e., `#core.core=#core`. This allows us to reduce the number of different commands.

6   External values and external processors

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

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

The values of objects 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 SVM, but may be called from the SVM as the command `external`. From a theoretical point of view, external processors are **oracles** to the SVM, as defined by SHOENFIELD [16].

External values can be imported into the memory of the SVM, and conversely. This is done by the commands `transportin`, `transportout`, `transportrefin` and `transportrefout`. The information about how to represent the external value in the memory of the SVM is called the **protocol**, and is used as an argument for the transport-commands. Figure 4 displays the interactions between the SVM and the physical device, and the SVM commands involved. Our current implementation includes protocols for representing

- strings,

- SVM programs (see Section 4),

- tapes and transition tables for Turing machines (see Section 9),

- Concise programs (see [15]).

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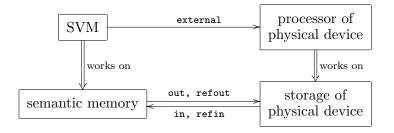

Figure 4: Interaction of the SVM with the physical device

7 An operational semantic for the SVM

We formally define the action of the SVM, in particular of every SVM command, by giving an operational semantic.

Let V the set of values.

A **state** is a triple (s, v, O) such that O is a sets of objects, $s : O \times O \to O$ is a semantic mapping, and $v : O \to V$ is a partial mapping that associates to some objects in O an external value. We define $\mathfrak{S}$ to be the set of states.

The object $p \in O$ denotes the record that contains the program to execute, and $c \in O$ denotes the core to use. In the following, we define $f := s(c, \texttt{focus})$.

For a semantic mapping s and a set $R \subseteq O \times O \times O$ we define $\text{replace}(s, R)$ as a semantic mapping s' with

$$s'(h, f) := \begin{cases} e & \text{if } (h, f, e) \in R \\ s(h, f) & \text{otherwise} \end{cases}$$

For a mapping $v : O \to V$ and a set $R \subseteq O \times V$ we define $\text{replace}(s, R)$ as mapping $v' \in O \to V$ with

$$v'(o) := \begin{cases} e & \text{if } (o, e) \in R \\ v(o) & \text{otherwise} \end{cases}$$

The execution of the SVM is a (possibly infinite) sequence of states $(S_1, S_2, \ldots, S_i, \ldots)$ determined by an initial state $S_0 = (s_0, v_0, O_0)$ and by $S_1 := \text{start}(S_0)$ and $S_{t+1} := \text{step}(S_t)$ for $t = 1, 2, \ldots$. The function $\text{start} : \mathfrak{S} \to \mathfrak{S}$ is defined by $\text{start}(s, v, O) = (\text{replace}(s, R), v, O)$ with $R := \{(c, \texttt{focus}, s(p, \texttt{start})), (c, \texttt{core}, c)\}$. The function $\text{step} : \mathfrak{S} \to \mathfrak{S}$ is defined by a case distinction over $s(f, \texttt{comm})$, as follows:

check: If $s(f, \texttt{comm}) = \texttt{check}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{True})\}$$

if $s(f, \texttt{arg2}) = s(f, \texttt{arg3})$ and

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{False})\}$$

if $s(f, \texttt{arg2}) \neq s(f, \texttt{arg3})$

create: If $s(f, \texttt{comm}) = \texttt{create}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O \cup \{o\})$ with $o \notin O$, and $R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), o)\}$.

exist: If $s(f, \texttt{comm}) = \texttt{exist}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{True})\}$$

if $s(s(c, s(f, \texttt{arg2})), s(f, \texttt{arg3})) \neq \texttt{Empty}$ and

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{False})\}$$

if $s(s(c, s(f, \texttt{arg2})), s(f, \texttt{arg3})) = \texttt{Empty}$

existref: If $s(f, \mathtt{comm}) = \mathtt{existref}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v, O)$ with

$$R = \{(c, \mathtt{focus}, s(f, \mathtt{next})), (c, s(f, \mathtt{arg1}), \mathtt{True})\}$$

if $s(s(c, s(f, \mathtt{arg2})), s(c, s(f, \mathtt{arg3}))) \neq \mathtt{Empty}$ and

$$R = \{(c, \mathtt{focus}, s(f, \mathtt{next})), (c, s(f, \mathtt{arg1}), \mathtt{False})\}$$

if $s(s(c, s(f, \mathtt{arg2})), s(c, s(f, \mathtt{arg3}))) = \mathtt{Empty}$

external: If $s(f, \mathtt{comm}) = \mathtt{external}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v', O)$ with
$R = \{(c, \mathtt{focus}, s(f, \mathtt{next}))\}$
and $v'(s(c, s(f, \mathtt{arg1})))$ is the value calculated by the external process associated to $s(f, \mathtt{arg2})$
with input $v(s(c, s(f, \mathtt{arg3})))$ and $v' = v$ for all other objects.

externalref: If $s(f, \mathtt{comm}) = \mathtt{external}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v', O)$ with
$R = \{(c, \mathtt{focus}, s(f, \mathtt{next}))\}$
and $v'(s(c, s(f, \mathtt{arg1})))$ is the value calculated by the external process associated to $s(c, s(f, \mathtt{arg2}))$
with input $v(s(c, s(f, \mathtt{arg3})))$ and $v' = v$ for all other objects.

fields: Let $(e_1, \ldots, e_n)$ a sequence containing every field of $s(c, s(f, \mathtt{arg2}))$. If $s(f, \mathtt{comm}) = \mathtt{fields}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v, O \cup \{o_1, \ldots, o_n\})$ with
$o_i \notin O$ for all $i = 1, \ldots, n$, and

$$\begin{aligned}
R = &\{(c, \mathtt{focus}, s(f, \mathtt{next})), (s(s(c, s(f, \mathtt{arg1})), \mathtt{next}), o_1)\} \\
&\cup \{(o_i, \mathtt{entry}, e_i) \mid i = 1, \ldots, n\} \\
&\cup \{(o_i, \mathtt{next}, o_{i+1}) \mid i = 1, \ldots, n-1\}
\end{aligned}$$

get: If $s(f, \mathtt{comm}) = \mathtt{get}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v, O)$ with
$R = \{(c, \mathtt{focus}, s(f, \mathtt{next})), (c, s(f, \mathtt{arg1}), s(s(c, s(f, \mathtt{arg2})), s(f, \mathtt{arg3})))\}$.

goto: If $s(f, \mathtt{comm}) = \mathtt{goto}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v, O)$ with
$R = \{(c, \mathtt{focus}, s(p, s(f, \mathtt{arg1})))\}$.

if: If $s(f, \mathtt{comm}) = \mathtt{if}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v, O)$ with
$R = \{(c, \mathtt{focus}, s(p, s(f, \mathtt{arg2})))\}$ if $s(c, s(f, \mathtt{arg1})) = \mathtt{True}$
$R = \{(c, \mathtt{focus}, s(f, \mathtt{next}))\}$ if $s(c, s(f, \mathtt{arg1})) = \mathtt{False}$
For every other value of $s(c, s(f, \mathtt{arg1}))$ the SVM stops and issues an error.

move: If $s(f, \mathtt{comm}) = \mathtt{move}$, then $\mathrm{step}(s, v, O) = (\mathrm{replace}(s, R), v, O)$ with
$R = \{(c, \mathtt{focus}, s(p, s(f, s(c, \mathtt{arg1}))))\}$.

refget: If $s(f, \texttt{comm}) = \texttt{refget}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), s(s(c, s(f, \texttt{arg2})), s(c, s(f, \texttt{arg3}))))\}$

refset: If $s(f, \texttt{comm}) = \texttt{refset}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(c, s(f, \texttt{arg1})), s(c, s(f, \texttt{arg2})), s(c, s(f, \texttt{arg3})))\}$

set: If $s(f, \texttt{comm}) = \texttt{set}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(c, s(f, \texttt{arg1})), s(f, \texttt{arg2}), s(c, s(f, \texttt{arg3})))\}$.

setconst: If $s(f, \texttt{comm}) = \texttt{setconst}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(c, s(f, \texttt{arg1})), s(f, \texttt{arg2}), s(f, \texttt{arg3}))\}$.

stop: If $s(f, \texttt{comm}) = \texttt{stop}$, then $\text{step}(s, v, O)$ is not defined and the sequence of states ends.

in: If $s(f, \texttt{comm}) = \texttt{in}$, then $\text{step}(s, v, O) = (s', v, O \cup \{o_1, \ldots, o_n\})$ with
$s'(c, s(f, \texttt{arg1}))$ is the semantic mapping that represents the value $v(s(c, s(f, \texttt{arg3})))$ in a record with handle $s(c, s(f, \texttt{arg1}))$ according to the protocol associated with $s(f, \texttt{arg2})$, using objects $\{o_1, \ldots, o_n\} \notin O$, and $s'(c, \texttt{focus}) = s(f, \texttt{next})$.

refin: If $s(f, \texttt{comm}) = \texttt{refin}$, then $\text{step}(s, v, O) = (s', v, O \cup \{o_1, \ldots, o_n\})$ with
$s'(c, s(f, \texttt{arg1}))$ is the semantic mapping that represents the value $v(s(c, s(f, \texttt{arg3})))$ in a record with handle $s(c, s(f, \texttt{arg1}))$ according to the protocol associated with $s(c, s(f, \texttt{arg2}))$, using objects $\{o_1, \ldots, o_n\} \notin O$, and $s'(c, \texttt{focus}) = s(f, \texttt{next})$.

out: If $s(f, \texttt{comm}) = \texttt{out}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), \text{replace}(v, Q), O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next}))\}$
$Q = \{(s(c, s(f, \texttt{arg1})), e)\}$ and e is the value that represents the record with handle $s(c, s(f, \texttt{arg3}))$ according to the protocol associated with $s(f, \texttt{arg2})$

refout: If $s(f, \texttt{comm}) = \texttt{refout}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), \text{replace}(v, Q), O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next}))\}$
$Q = \{(s(c, s(f, \texttt{arg1})), e)\}$ and e is the value that represents the record with handle $s(c, s(f, \texttt{arg3}))$ according to the protocol associated with $s(c, s(f, \texttt{arg2}))$

unset: If $s(f, \texttt{comm}) = \texttt{unset}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(f, \texttt{arg1}), s(f, \texttt{arg2}), \texttt{Empty})\}$.
For every other value of $s(f, \texttt{comm})$, the execution of the SVM stops and issues an error.

7.1 Garbage collection

For some state $S = (s, v, O) \in \mathfrak{S}$ we define the relation $o_1 \rhd_S o_2$ if there exists an object $x \neq \texttt{Empty}$ with either $s(o_1, x) = o_2$ or $s(o_1, o_2) = x$.

We define $o_1 \rhd\rhd_S o_2$ if there exists a sequence $(x_1, \ldots, x_n)$ of objects with $o_1 \rhd_{s_t} x_1 \rhd_S \ldots \rhd_{s_t} x_n \rhd_S o_2$.

For a set X of objects, we define $X \rhd\rhd_S o$ if there exists an $x_i \in X$ with $x_i \rhd\rhd_S o$.

Theorem (Garbage collection) For given state S and $p, c \in O$, we define $O^{\text{ess}} = \{o \in O | \{c, p\} \rhd\rhd_S o\}$.

Let s^{ess} and v^{ess} the restrictions of s and v to O^{ess}, let $S^{\text{ess}} = (s^{\text{ess}}, v^{\text{ess}}, O^{\text{ess}})$.

Then

$$\text{step}(S) = \text{step}(S^{\text{ess}})$$

Proof: It is easy to check that in all cases of the definition of step, the result step(S) only depends on objects o with $\{c, p\} \rhd\rhd_S o$. ∎

This theorem allows us to perform garbage collection after every step of the SVM, i.e., delete every sem `a.b=c` with `a` $\notin O^{\text{ess}}$, and delete every object `o` $\notin O^{\text{ess}}$, and its value.

8 Description of the SVM commands

We now introduce the commands of the SVM language and describe their effect. There are four groups of commands: Table 2 describes the commands that are needed to give the program an appropriate structure. Table 3 contains the assignments, i.e., those commands that perform alterations in the memory of the SVM. Table 4 gives the commands used for flow control, and Table 5 the commands that establish communication with the physical device, namely call external processes and access external values.

SVM 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 2: Structuring commands

8.1 Example: shallow copy

The following SVM program performs a shallow copy from the record passed to the SVM in `#core.copyfrom` to the object in `#core.copyto`.

```
program(copyFields)
process(getfields)
  fieldstack=fields(copyfrom)
  goto(checkfornext)
process(checkfornext)
  thereismore=exist(fieldstack,next)
```

SVM command	comment
#1=check(#2,#3)	sets ^#1 to True if ^#2 = ^#3, else to False
create(#1)	assigns some free object to ^#1
#1=exist(#2,#3)	sets ^#1 to True if ^#1.#2 exists, else to False
#1=existref(#2,#3)	sets ^#1 to 'T if ^#1.^#2 exists, else to False
#1=fields(#2)	creates a linked list with handle ^#1 containing all fields of ^#2
#1=get(#2,#3)	assigns ^#2.#3 to ^#1
#1=refget(#2,#3)	assigns ^#2.^#3 to ^#1
(#1,#2)=refset(#3)	assigns ^#3 to ^#1.#2
(#1,#2)=set(#3)	assigns ^#3 to ^#1.^#2
(#1,#2)=setconst(#3)	writes #3 to ^#1.#2
unset(#1,#2)	deletes the position (^#1,^ #2), i.e., sets it to Empty

Table 3: Assignment commands

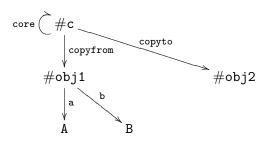| SVM command	comment
goto(#1)	sets the focus to the first line of process #1
move(#1)	sets the focus to the first line of process ^#1
if(#1,#2)	sets the focus to the first line of process #2 if ^#1=True, and to the next line if ^#1=False
stop	ends a program

Table 4: Commands for flow control

SVM command	comment
#1=external(#2,#3)	calls external processor #2 with input VALUE(^#3) and output VALUE(^#1)
#1=externalref(#2,#3)	calls external processor ^#2 with input VALUE(^#3) and output VALUE(^#1)
#1=in(#2,#3)	imports VALUE(^#2) into ^#1 by protocol #3
#1=out(#2,#3)	exports record ^#2 into VALUE(^#1) by protocol #3
#1=refin(#2,#3)	imports VALUE(^#2) into ^#1 by protocol ^#3
#1=refout(#2,#3)	exports record ^#2 into VALUE(^#1) by protocol ^#3

Table 5: Commands for external communication

```
    if(thereismore,copynext)
    stop
process(copynext)
    fieldstack=get(fieldstack,next)
    fieldtocopy=get(fieldstack,entry)
    entrytocopy=refget(copyfrom,fieldtocopy)
    (copyto,fieldtocopy)=refset(entrytocopy)
    goto(checkfornext)
start(getfields)
```

For example, if the SVM is called with the program above and core `#c` and a semantic memory containing the following sems:
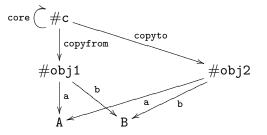

Upon execution of this program the following happens:

First, the process `getfields` is entered (due to the line `start(getfields)`), and this process generates a linked list beginning in `^fieldstack` such that each field `#f` of `^copyfrom` is represented as `^fieldstack.next`^{k}`.entry=#f` for some $k \in 0, 1, 2, \ldots$. Then, process `checkfornext` is entered.

The process `checkfornext` checks if `^fieldstack.next` is nonempty. If this is the case, process `copynext` is entered; otherwise the SVM halts.

Process `copynext` first sets `^fieldstack` to `^fieldstack.next`, i.e., we advance one link in the linked list that contains all the fields of `^copyfrom`. Then the field of `^copyfrom` (stored in `^fieldstack.entry`) and the entry `^copyfrom.^fieldstack.entry` are stored in `^fieldtocopy` and `^entrytocopy` respectively. Finally this field and entry are written into a new sem with handle `^copyto`.

The result after execution of the SVM is a semantic memory containing:

9   Turing machines and their simulation

In this section we formally introduce Turing machines and show that the SVM is Turing complete by giving an SVM program that simulates any ordinary Turing machine.

9.1   The tape of the Turing machine

The cells on the tape of the Turing machine are simulated by a linked list in `^context.tape` where `^context.tape.entry` is the left end of the tape, and initially holds the delimiting symbol '>'. Objects 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.next`, and the TM to be in state '1'.

9.2   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.

9.3   Example: 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

9.4   Example: 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.

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 [7] by MINSKY, by ROBINSON [12], and recently by NEARY & WOODS [8].

Appearently no effort was put in making universal Turing machines user-friendly while keeping them small, which would be related to the goal of our work.

9.5 The SVM-code of a universal TM

To prove Turing completeness we now specify a SVM program that simulates an arbitrary Turing machine. The reader should interpret it with the help of Tables $2-5$.

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

`^transtable`	contains the linked list of instructions
`^transtable.next`^{k}`.state`	contains S of the kth instruction
`^transtable.next`^{k}`.reading`	contains R of the kth instruction
`^transtable.next`^{k}`.towrite`	contains W of the kth instruction
`^transtable.next`^{k}`.tomove`	contains M of the kth instruction
`^transtable.next`^{k}`.tostate`	contains S' of the kth instruction
`^transtable.next`^{k}`.next`	contains the $k+1$th instruction

Note that if W in the instruction is the empty string, `#obj.towrite` is set to `#obj.reading`, hence no alteration is done by writing. If M in the instruction is the empty string, `#obj.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`.

```
program(UTM)
process(init)
% makes the necessary nodes available in the core
    tape=in(tapename,tm_tape)
    transtable=in(tablename,tm_instructions)
    position=get(core,tape)
    (core,state)=setconst(1)
    (core,cR)=setconst(R)
    (core,cL)=setconst(L)
    (core,blank)=setconst( )
    goto(nextcommand)
process(nextcommand)
% initializes the comparing, reads the tape
    trycommand=get(core,transtable)
    reading=get(position,entry)
    goto(checkstate)
process(trynext)
% halts the TM if there are no instructions left
    therearemorecommands=exist(trycommand,next)
    if(therearemorecommands,increase)
    goto(endprocess)
```

---

[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`

```
process(increase)
% go to next command
    trycommand=get(trycommand,next)
    goto(checkstate)
process(checkstate)
% checks if the state of the TM is equal to the state in the command
    stateincommand=get(trycommand,state)
    samestate=check(state,stateincommand)
    if(samestate,checksymbol)
    goto(trynext)
process(checksymbol)
% checks if the symbol read by the TM is equal to
% the symbol in the command
    symbolincommand=get(trycommand,reading)
    samesymbol=check(symbolincommand,reading)
    if(samesymbol,executecommand)
    goto(trynext)
process(executecommand)
% executes the instructions in the command
    state=get(trycommand,tostate)
    towrite=get(trycommand,towrite)
    (position,entry)=set(towrite)
    tomove=get(trycommand,tomove)
    moveleft=check(tomove,cL)
    moveright=check(tomove,cR)
    if(moveleft,left)
    if(moveright,right)
    goto(nextcommand)
process(left)
% moves the head to the left
    notleftend=exist(position,last)
    if(notleftend,goleft)
    create(newfield)
    (position,last)=set(newfield)
    (newfield,entry)=set(blank)
    (newfield,next)=set(position)
    position=get(position,last)
    goto(nextcommand)
process(goleft)
    position=get(position,last)
    goto(nextcommand)
process(right)
% moves the head to the right
    notrightend=exist(position,next)
    if(notrightend,goright)
    create(newfield)
    (position,next)=set(newfield)
    (newfield,entry)=set(blank)
    (newfield,last)=set(position)
```

```
        position=get(position,next)
        goto(nextcommand)
process(goright)
        position=get(position,next)
        goto(nextcommand)
process(endprocess)
        tapeasvalue=out(tape,tm_tape)
        copyoftape=external(valuecopyname,tapeasvalue)
        filename=external(writetofilename,copyoftape)
        stop
start(init)
```

When invoked, it expects to have the following objects in its core:

`^tapename`	is an object that has a tape as external value,
`^tablename`	is an object that has a transistion table as external value,
`^valuecopyname`	is an object associated to an external processor that makes a copy of the external value,
`^filename`	is an object that has a valid filename as external value,
`^writetofilename`	is an object associated to an external processor that writes a type of into a file

The SVM-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, including loading the tape (stored as external value in `^tapename`) and the transistion table (stored as external value in `^tablename`)
`nextcommand`	resets the records and replaces an empty cell on the tape by a blank.
`trynext`	halts if there is no next instruction and else calls the process
`increase`	which brings the next instruction into consideration.
`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`	and
`goleft`	move the head to the left (if the head is not already on the leftmost cell), while
`right`	and
`goright`	move the head to the right.
`endprocess`	then writes the result first as the value of `^tapeasvalue`, then makes a copy of that value to `^copyoftape` (which is redundant, but serves for an example for an external processor), and then the value is written into the file that is the value of `^filename`.

Correctness is straightforward to prove.

The SVM 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 SVM program.

10   The USVM

A universal SVM (USVM) teh semantic analogon to a universal Turing machine. It is a special SVM program capable of 'simulating' the processing of an other SVM program `P` in the following sense: The context of the USVM contains the SVM program `P` and the context of `P`. When the USVM has finished, the USVM has produced the same changes in the context as `P` would have produced when called directly.

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

The SVM code of the USVM

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

```
program(USVM)

%%%%%%%%%%%%%%%%%%%%%%% control handling %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

process(init)
% initialize nodes, initialize local and global frame
    (simcore,core)=set(simcore)
    (simcore,program)=set(simprog)
    simprocesses=get(simprog,processes)
    startfocus=get(simprog,start)
    simfocus=refget(simprocesses,startfocus)
    goto(load)

process(next)
% proceed to the next command to simulate
    simfocus=get(simfocus,next)
    goto(load)

process(load)
% load the information about the command to simulate to the core
    sim_comm=get(simfocus,comm)
    move(sim_comm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
process(move)
% goto ^#process
    processname=get(simfocus,arg1)
    process=refget(simcore,processname)
    simfocus=refget(simprocesses,process)
    goto(load)

process(goto)
% goto #process
    process=get(simfocus,arg1)
    simfocus=refget(simprocesses,process)
    goto(load)

process(if)
% if ^#cond goto #process
    criterionname=get(simfocus,arg1)
    criterion=refget(simcore,criterionname)
    if(criterion,ifapplies)
    goto(ifappliesnot)

process(ifapplies)
    process=get(simfocus,arg2)
    simfocus=refget(simprocesses,process)
    goto(load)

process(ifappliesnot)
    goto(next)

process(stop)
% stop
    stop


%%%%%%%%%%%%%%%%%%%% internal handling %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

process(create)
% create ^#newnode
    toassign=get(simfocus,arg1)
    create(newobj)
    (simcore,toassign)=refset(newobj)
    goto(next)

process(fields)
% ^#fieldlist=fields of ^#record
    writetoname=get(simfocus,arg1)
    getfromname=get(simfocus,arg2)
    getfrom=refget(simcore,getfromname)
    stackoffields=fields(getfrom)
    (simcore,writetoname)=refset(stackoffields)
    goto(next)
```

```
process(check)
% ^#isequal=(^#left==^#right)
    leftname=get(simfocus,arg2)
    left=refget(simcore,leftname)
    rightname=get(simfocus,arg3)
    right=refget(simcore,rightname)
    result=check(left,right)
    writeto=get(simfocus,arg1)
    (simcore,writeto)=refset(result)
    goto(next)

process(exist)
% ^#node=exist(#record,#field)
    leftname=get(simfocus,arg2)
    left=refget(simcore,leftname)
    right=get(simfocus,arg3)
    result=existref(left,right)
    writeto=get(simfocus,arg1)
    (simcore,writeto)=refset(result)
    goto(next)

process(existref)
% ^#node=exist(^#record,^#field)
    leftname=get(simfocus,arg2)
    left=refget(simcore,leftname)
    rightname=get(simfocus,arg3)
    right=refget(simcore,rightname)
    result=existref(left,right)
    writeto=get(simfocus,arg1)
    (simcore,writeto)=refset(result)
    goto(next)

process(setconst)
% ^#1.#2=const #3
    handle=get(simfocus,arg1)
    sethandle=refget(simcore,handle)
    field=get(simfocus,arg2)
    setto=get(simfocus,arg3)
    (sethandle,field)=refset(setto)
    goto(next)

process(refset)
% ^#1.^#2=^#3
    handle=get(simfocus,arg1)
    sethandle=refget(simcore,handle)
    fieldname=get(simfocus,arg2)
    field=refget(simcore,fieldname)
    entry=get(simfocus,arg3)
```

```
      setto=refget(simcore,entry)
      (sethandle,field)=refset(setto)
      goto(next)

process(set)
% ^#1.#2=^#3
    handle=get(simfocus,arg1)
    sethandle=refget(simcore,handle)
    field=get(simfocus,arg2)
    entry=get(simfocus,arg3)
    setto=refget(simcore,entry)
    (sethandle,field)=refset(setto)
    goto(next)

process(refget)
% ^#1=^#2.^#3
    addressname=get(simfocus,arg1)
    handlename=get(simfocus,arg2)
    handle=refget(simcore,handlename)
    fieldname=get(simfocus,arg3)
    field=refget(simcore,fieldname)
    towrite=refget(handle,field)
    (simcore,addressname)=refset(towrite)
    goto(next)

process(get)
% ^#1=^#2.#3
    addressname=get(simfocus,arg1)
    handlename=get(simfocus,arg2)
    handle=refget(simcore,handlename)
    field=get(simfocus,arg3)
    towrite=refget(handle,field)
    (simcore,addressname)=refset(towrite)
    goto(next)

process(unset)
% unset(^#1.^#2)
    firstname=get(simfocus,arg1)
    secondname=get(simfocus,arg2)
    first=refget(simcore,firstname)
    second=refget(simcore,secondname)
    unset(first,second)
    goto(next)


%%%%%%%%%%%%%%%%%%%%%% external handling %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

process(external)
% external: #program(^#input)
    inputname=get(simfocus,arg3)
```

```
        processname=get(simfocus,arg2)
        outputname=get(simfocus,arg1)
        inputnode=refget(simcore,inputname)
        processobj=refget(simcore,processname)
        outputnode=refget(simcore,outputname)
        outputnode=external(processobj,inputnode)
        goto(next)

process(in)
% move in ^#node as #protocol
        transto=get(simfocus,arg1)
        transfrom=get(simfocus,arg2)
        protocol=get(simfocus,arg3)
        objtotransfrom=refget(simcore,transfrom)
        transtotemp=refin(objtotransfrom,protocol)
        (simcore,transto)=refset(transtotemp)
        goto(next)

process(out)
% move out ^#node as #protocol
        transto=get(simfocus,arg1)
        transfrom=get(simfocus,arg2)
        protocol=get(simfocus,arg3)
        objtotransfrom=refget(simcore,transfrom)
        transtotemp=refout(objtotransfrom,protocol)
        (simcore,transto)=refset(transtotemp)
        goto(next)

process(refin)
% move in ^#node as ^#protocol
        transto=get(simfocus,arg1)
        transfrom=get(simfocus,arg2)
        protocolname=get(simfocus,arg3)
        protocol=refget(simcore,protocolname)
        objtotransfrom=refget(simcore,transfrom)
        transtotemp=refin(objtotransfrom,protocol)
        (simcore,transto)=refset(transtotemp)
        goto(next)

process(refout)
% move out ^#node as ^#protocol
        transto=get(simfocus,arg1)
        transfrom=get(simfocus,arg2)
        protocolname=get(simfocus,arg3)
        protocol=refget(simcore,protocolname)
        objtotransfrom=refget(simcore,transfrom)
        transtotemp=refout(objtotransfrom,protocol)
        (simcore,transto)=refset(transtotemp)
        goto(next)
```

```
% info to start program:
start(init)
```

Without blank lines and comment lines, the USVM contains 166 lines.

References

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

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

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

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

[5] T.B. Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific American*, 284(5):34–43, 2001.

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

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

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

[9] A. Neumaier and P. Schodl. A Framework for Representing and Processing Arbitrary Mathematics. In J. Filipe and J.L.G. Dietz, editors, *Proc. Int. Conf. Knowledge Engineering and Ontology Development*, pages 476–479. SciTePress, 2010. An ealier version is available at `http://www.mat.univie.ac.at/~schodl/pdfs/IC3K_10.pdf`.

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

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

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

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

[14] P. Schodl and A. Neumaier. The FMathL type system. Manuscript, available at `http://www.mat.univie.ac.at/~neum/FMathL.html#TypeSystem`, 2010.

[15] P. Schodl, A. Neumaier, K.Kofler, F. Domes, and H. Schichl. Towards a Self-reflective, Context-aware Semantic Representation of Mathematical Specifications. In J. Kallrath, editor, *Modeling Languages in Mathematical Optimization.* Springer, to appear.

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

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

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