Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Platforms, Specifications and Decisions

J.A. Bergstra

Preprint nr. 019 March 2000

Artificial Intelligence Preprint Series

ISBN 90-393-2399-2
ISSN 1389-5184
Prof.dr. A. Visser, Editor

Onderwijsinstituut CKI
Utrecht University
Heidelberglaan 8
3584 CS Utrecht
The Netherlands

PLATFORMS, SPECIFICATIONS AND DECISIONS

J.A. Bergstra

February 15, 2000

This document contains as part 2 a revised and extended version of the existing UU CKI report 017, and as part 3 a reformatted version of the UvA PRG report P2000 01.

Value & Assignment

Specification & Test

Compiler & Interpreter

Decidable & Undecidable

Program Algebra with Concrete Data Types

Jan Bergstra*

February 15, 2000

Abstract

Program variables in a call-by-value mode are added to the elementary syntax of program algebra. Only very simple data are considered: finite sequences of bits. The meaning of program variables, data type constants and assignments in a call-by-value regime is given by means of a translation into behaviors or processes. Assignment actions as well as input actions are modeled by means of sequential substitutions for processes frongg behavior extraction.

Contents

*J.A. Bergstra, University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl.

1 Introduction

This paper presumes acquaintance with program algebra and projection semantics as outlined in [2]. The current objective is to extend program algebra with data types, program variables and assignments. The data types to be included are concrete. This means that no abstraction in favor of specification or analysis has been made. Program variables are conventionally used to name abstract registers containing variable elements of a data type.[1] Program variables in a call-by-value setting together with assignments and tests are very characteristic of all classical forms of imperative programming. In program algebra [2] the principles of imperative programs have been outlined with an emphasis on control structures, leaving the lesser important aspect of assignments and values aside. Adding values, variables and assignments to program algebra is not entirely unproblematic.

Decisions have to be made about data types and values, and about a strategy for semantic description. Just as in the case of program algebra without data there fails to be a unique way in which a behavior can be attached to a program. In fact the number of degrees of freedom increases considerably with the introduction of concrete data. An attempt has been made to use a simple formal technology. For instance substitutions have been avoided. The related notion of term evaluation cannot be avoided, however. Only a very simple form of data are covered in detail in the sequel. More complicated data types, for instance floating-point arithmetic, have been ignored in favor of a focus on the very principles of program variables and their manipulation.

We have made a simplification to our discussion by assuming that input actions and output actions are the only means of communication with components present in the context of a running program. A specialized notation is provided for outputs to the user console screen. As is usual in the case of real-life program notations, this facilitates the rendering of examples of programs and their expected executions.[2]

2 Concrete Data Types

Current computers use bits, bytes, long ints, floats and many similar data types. We have simply generalized these to bit groups. A bit group is a sequence of bits of finite and fixed length. For each natural number n the set of bits of length n constitutes a concrete data type denoted with $\mathrm{BG}(n)$.[3]We write BS (bit sequences) for the union of the $\mathrm{BG}(n)$ for all n. $\mathrm{BS}(0)$ denotes the singleton collection with the empty sequence[4] as its only element.

A rather elaborate scheme of notations for elements of BGs and for variables and meta-variables ranging over these will be used below. Most definitions are given in the form of examples admitting obvious generalization.

[1] It should be noticed that program variables do not range over programs although the name suggests that they do. Program variables are variables used in programs and are fixed names of objects. In contrast program references may point to different objects at different times. Program references rather than program variables are critical in object-oriented imperative programming. This paper sheds no light on the use and meaning of program references, however. As program variables are considered more basic these are discussed independently of objects and references. Of course it is also possible to take a program references as a point of departure.

[2] Rather than providing definitions in formal generality we will often use instantiations from which, we hope, the reader to be able to generalize quickly to the adequate level of abstraction. For instance if it is stated that `37bc246` represents a bit sequence with length 37, a reader is expected to generalize this and to understand that `38bc246` denotes a bit sequence of length 38.

[3] Concrete data types are given as collections of values. Abstract data types are implicitly characterized by means of families of operators. Of course operators on concrete data types are essential, as they are for abstract dat types. Nevertheless concrete data types can be defined prior to their most prominent operators.

[4] Empty sequences will be denoted with [].

2.1 Constant values

With 23bc678 we denote the 23-bit-long bit sequence representing the constant number 678 (decimal notation, as a rule used as a default). Because 678 uses fewer than 23 bits in a binary notation[5] additional 0s will be added to the left. In all cases we will have the most significant bit or digit on the left. Similarly 3bc6578 is a 3-bit-long string containing the 3 least significant bits of a binary representation of 6578. Hexadecimal notation[6] is indicated with `h`. For instance 23bch6E78B represents a bit string containing the 23 least significant bits of the binary version of the hexadecimal number 6E78B (using additional 0s to make up for 23 bits in this case). Of course binary constants are also possible: `6bcb001110001` represents `110001` and `10bcb001110001` represents `00001110001`. `For a binary number 100110 the least significant bit is 0 (i.e. the right-most one) whereas 1 is the most significant bit. 1001 is the sequence of its 4 most significant bits. 0110 is the sequence of its 4 least significant bits..` These notations allow us to use binary, decimal and hexadecimal notation freely while having an underlying semantics consisting of bit sequences of an unambiguously defined length. We assume that these examples are sufficiently clear, making definitions phrased in more general terms redundant. We notice that 28bch1789BD56AE is in BG(28) and so on.

2.2 Casting

There is an obvious casting upwards as well as downwards between the various BGs. Casting upwards is done by adding leading 0s, casting downwards is performed by removing the correct number of leading bits. For example `(7b)5bc4 = 0000100` and `(5b)7bc64 = 00000`.

[5]Binary notation works as follows: 0 represents 0, 1 represents 1, 10 represents 2, 11 represents 3, 100 represents 4 101 represents 5, 110 represents 6, 111 represents 7, 1000 represents 8 and so on.

[6]Hexadecimal notation is as follows: (hexadecimal) 0 represents (decimal) 0,..,9 represents 9, A represents 10, B represents 11, C represents 12, D represents 13, E represents 14, F represents 15, 10 represents 16, 11 represents 17,..,1A represents 26, 1B represents 17,.., 20 represents 32 and so on.

2.3 Variables

All variable names are natural numbers, and range over finite bit sequences (i.e. the union of all BG(n). Such names are denoted with binary, decimal (default) or hexadecimal formats. For instance `v5` is variable 5. It is the same variable as `v05` or `vb0101` or `vh5`. Variables are converted to decimal form as follows: a string only consisting of 0s (the empty string included) represents 0. Other strings `x` are considered the binary representation of a natural number (after removing leading 0s). Converting this number to decimal notation gives the required decimal form (denoted with `dval(x)`).

2.4 Long sequences

In case very long sequences are needed exponential notation is used: `e45bd91` denotes the binary form of 91 with enough 0s added to the left so as to obtain a bit sequence of length 2^{45}.

Obviously more flexible notations are possible, e.g. allowing the exponent to be provided in hexadecimal notation etc. Such modifications can be easily designed e.g. `h2Bbcd345` is the constant 345 viewed as a bit sequence of length 2B in hexadecimal notation (i.e. of length 43).

2.4.1 Terminology for large data

A byte is a constant for BG(8), a kilobyte is a constant for BG(2^{13}). A megabyte is a constant for BG(2^{23}) and a gigabyte of information denotes some constant in BG(2^{33}). Terabytes are in BG(2^{43}).

2.5 Expressions

Expressions are best introduced using a context-free grammar in BNF style.[7] To save space we will forget about hexadecimal notation. In the notation below white space plays the role of string concatena-

[7]BNF stands for Backus-Naur Form.

tion.[8] D stands for digit, B for Boolean value (or bit), NN stands for natural number, BS for bit sequence, pos for positive, ∘ represents string concatenation for string expressions. With [] we denote the emply list.

2.5.1 A BNF grammar for data expressions

This grammar generates an ad hoc collection of expressions for data, without any pretentions for usability or practicality. Dpos = 1|2|3|4|5|6|7|8|9,
D = 0 | Dpos,
DSne = D | D DSne,
DS = [] | DSne,
DSEXP = DS | [D] | DSEXP ∘ DSEXP,
DNNpos = Dpos | Dpos DS,
DNN = 0 | DNNpos,
Bpos = 1,
B = 0 | Bpos,
BEXP = `true` | `false`
| `non`(BEXP)
| `or`(BEXP,BEXP)
| `and`(BEXP,BEXP)
| `eqBS`(BSEXP,BSEXP)
| `eqNN`(NNEXP,NNEXP)
| `lessNN`(NNEXP,NNEXP)
| `grNN`(NNEXP,NNEXP)
BSne = B | B BSne,
BS = [] | BSne,
BNNpos = Bpos | Bpos BS,
BNN = 0 | BNNpos,
BSVAR = v DNN | vb BNN,
BSCONST = DNNpos bc DS
| DNNpos bcb BS
| e DNNpos bc DS
| e DNNpos bcb BS,
NNOP2pref = `max`|`minus`|`plus`
|`mult`|`div`|`rem`|`bwand`|`bwor`,
NNOP1 = `2log`|`2exp`|`bwneg`,
NNEXP = DNN | b BNN |
| NNOP1 (NNEXPP)
| NNOP2pref (NNEXP,NNEXP)
| `bs2nn`(BSEXP)
| `lengthOf`(BSEXP),
BSEXP = BS | [BITEXP] | BSVAR
| BSEXP ∘ BSEXP
| (DNNpos b) BSEXP
| (e DNNpos b) BSEXP
| `nn2bs`(NNEXP)
| `tail`(BSEXP)
| (BSEXP)
BITEXP = B | `head`(BSEXP)
| `b2bit`(BEXP).

[8]The white space as a string is written \w when needed, the newline as \n.

2.5.2 Remarks

The notation `[0]` may be useful when it is important to see from the type script the a bit sequence of length 1 is meant. Here the BS expressions[9], `[1]` and `(1)` all denote the same string.

The terminals of this grammar are given by the following set SYMBOLalph (alphabet of symbols)[10]

SYMBOLalph {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, [,], t, r, u, e, f, a, l, s, e, n, o, n, o, r, a, n, d, e, q, B, S, e, q, B, N, l, e, s, s, N, N, g, r, N, N, v, v, b, b, c, e, b, c, b, m, a, x, m, i, n, u, s, p, l, u, s, m, u, l, t, d, i, v, r, e, m, b, w, a, n, d, b, w, o, r, 2, l, o, g, 2, e, x, p, b, w, n, e, g, b, s, 2, n, n, (, ',' ,) ,l, e, n, g, t, h, O, f, n, n, 2, b, s, t, a, i, l, h, e, a, d, b, 2, b, i, t }

In this grammar a space is used as a string concatenation operator (whereas ∘ denotes concatenation at the level of string expressions). When writing strings string concatenation take no space. Thus DS generates for instance [], 3, 34, 345.

2.5.3 Examples of expressions

Examples of expressions generated from a number of different non-terminals of the above BNF:

DS `[],0,00102,16550`

[9]Strings generated from the non-terminal BSEXP 1, texttt((1))

[10]In set notation repetitions are allowed: $\{4,1,2,4,4,7\} = \{4,1,2,7\}$ and order does not matter: $\{1,2,3,7\} = \{7,3,2,1\}$. Moreover, the set theory notation uses the comma (,) as its separator. When ',' is considered a set element it will be quoted. The set containing just a comma is denoted with {','} rather than with {,}.

BEXP `true, non(false), and(false,false), eqNN(bs2nn([0]),lengthOf([]∘[7]))`

BSEXP `[0], [0] ∘ [head(0010010)] ∘ 278, (128b)1000 ∘ (e18d)707, (e12b)(1000 ∘ 00110) ∘ (e18b)707`

2.6 Expression evaluation

We will make no distinction between BS and

$$\bigcup_{n=0}^{n=\infty} \mathrm{BG}(n).$$

All expressions will take values in BS and all variables will be instantiated with values in BS as well. Let σ be a partial mapping from BSVAR to BS. Given an expression E such that all variables in E are in the domain of σ, we denote with $\mathtt{eval}_\sigma$(E) the result of evaluating expression E in context (or valuation[11]) σ. The technical definition of `eval` is implicit in the preceding discussion and the mnemonics of the operators. We give some examples only. The rules of calculation for BS expressions are simple but somewhat arbitrary. We list them in arbitrary order:

1. 0 in NN represents `true`, 1 represents `false`.
2. `[0]` evaluates to 0, and `[0]∘[1]` to 01.
3. The head of an empty sequence takes value 0, its tail is the empty sequence.
4. The casting operator adapts the length of a bit sequence by either removing bits from the left or adding 0s at the left.
5. Arithmetic operations (`plus` etc.) are performed with full precision on the natural numbers. This means that arguments are first transformed into natural numbers (remove leading 0s, read a bit string as a binary number representation, in case of constants use binary, decimal or hexadecimal notation and transform [] to 0). Then computations are carried out. Results are converted back to the binary format in a standard way without redundant leading 0s.
6. Division by 0 generates an error `M`. This error propagates throug all operations (boolean and aritmetic). The meaning of a program starting with an action depending on the value `M` is $||\#||$.
7. Subtraction (`minus`) is understood as cut-off subtraction (also termed `monus`). Cut-off subtraction produces 0 where usual subtraction produces negative values. Thus we will assume `5-7 = 0` etc.
8. `lengthOf` returns a bit sequence of minimal length for representing the length of its argument in binary fashion, representing 0 by the sequence [0] (rather than []).
9. nn2bs gives 0 on 0, 1 on 1, 10 on 2 etc.
10. `bs2nn` produces 0 on `[]`, on 0 and on 00, 2 on 010, 3 on 0011 and, 1 on 1 on 01 and so on.
11. Valuations are denoted by sequences of pairs. E.g `[0110/v3,..,100110/v72,00/v75]`, is a valuation assigning the bit sequence 0110 to the program variable v3, the bit sequence 100110 to program variable v72, the bit sequence 00 to program variable v75, and the empty sequence (by default) to all other variables. If σ is a valuation then σ`[011001/v15]` is the valuation that coincides with σ on all variables except for `v15`, where it produces the value 011001 instead. An example:

    ```
    ([010/v2,110/v4,1111/v5,00000/v7]
    [1110/v4])[0/v3] = [010/v2,0/v3,
    1110/v4,1111/v5,00000/v7]
    ```

12. Concerning valuations we mention these additional examples:

    $$\sigma\texttt{[010/v34](v34)} = \texttt{010},$$

    $$\sigma\texttt{[10/v54](v34)} = \sigma\texttt{(v34)}$$

    $$\emptyset\texttt{[10/v54](v34)} = \texttt{[]},$$

13. The empty valuation is denoted with $\emptyset$. It maps each variable `vi` by default to the empty bit sequence.

[11] Sometimes also called environment.

2.6.1 Closed expressions

For closed expressions, i.e. expressions not containing any variables, the evaluation is independent of any valuation σ. We provide some examples with comments.

1. $\texttt{eval}_\sigma(\texttt{2bc5}) = [0] \circ [1] = 01$
2. $\texttt{eval}_\sigma(\texttt{((10b)110011)} = 0000110011$
3. $\texttt{eval}_\sigma(\texttt{((8b)1001110011)} = 01110011$
4. $\texttt{eval}_\sigma(\texttt{((7b)5bcb11111100)} = 0011100$
5. $\texttt{eval}_\sigma(\texttt{((6b)8bcb101))} = 000101$
6. $\texttt{eval}_\sigma(\texttt{nn2bs(0110))} = 110$
7. $\texttt{eval}_\sigma(\texttt{eqNN(bs2nn(0110),0110)} = 0$
8. $\texttt{eval}_\sigma(\texttt{eqBS(nn2bs(0110),0110)} = 1$
9. $\texttt{eval}_\sigma(\texttt{plus(3bc5,2bc11))} =$
   $\texttt{eval}_\sigma(\texttt{plus(3bc5,2bc3))} =$
   $\texttt{eval}_\sigma(\texttt{((3b)4bc8)} = 000$
10. $\texttt{eval}_\sigma(\texttt{ lengthOf(plus(lengthOf(7bc8),}$
    $\texttt{10bc13))))} =$
    $\texttt{eval}_\sigma(\texttt{lengthOf(plus(3bc7,10bc13)))} =$
    $\texttt{eval}_\sigma(\texttt{3bc5}) = 101$

2.6.2 Open expressions

Open expressions are non-closed expressions, i.e. expressions containing variables. Open expressions can only be evaluated if a valuation is provided, mapping the variables occurring in the expressions to BS. We give two examples.

1. $\texttt{eval}_{\texttt{[110/v3,00010/v5]}}(\texttt{v3} \circ$
   $\texttt{nn2bs(plus(bs2nn(v5),lengthOf(v5)))} =$
   $\texttt{110111}$
2. $\texttt{eval}_{\texttt{[[]/v4,0/v7]}}(\texttt{v3} \circ$
   $\texttt{nn2bs(plus(bs2nn(v4),lengthOf(v7)))} =$
   $\texttt{1}$

3 Actions with bit group parameters

In this section, a collection of instructions is outlined. The collection is named Σ_{pv1} (co-service interface using binary data types no. 1). Σ_{pv1} will be used as a vehicle for the explanation of behavior extraction in the presence of program variables. In practical case studies this co-service interface will probably be too small. It can be extended in many directions without invalidating the semantic considerations to be developed below.

When a program starts it is assumed that all of its variables for BGs are initialized with an empty sequence. At any instant of time during the computation of a program all variables and meta-variables have some current value. This value is a bit string in the appropriate BG.

The notation used for instructions below is quite heavy. It should allow combination with other mechanisms for working with variables, references or pointers without generating confusion. In an actual language the names of the actions below, if present in any form at all, are likely to be simplified because general precautions to avoid confusion between different conventions can be taken at a global level of design.

3.1 Data control instructions

Advanced control instructions are processed by the program itself so to speak. Semantically they are taken into account by means of projection semantics. Data control instructions affect program semantics at a far lower level: data control instructions are taken into account by an appropriate behavior extraction. In [2] no mention of data control actions was made. We will do so below, thereby necessitating an extension of the definition of behavior extraction. Data control instructions are not a part of a lower (or co-service) interface of a program, as they do not constitute requests for the service of another system component. When determining a behavior for a program such instructions are taken into account by the behavior extraction operator.

Below `Q` stands for an expression in BSEXP. Further `R` represents a BEXP term (i.e. a boolean expression in this context).[12] Except for the Test instructions all other instructions will in all cases return the boolean value `true` to the program as a default value.

3.1.1 Test instructions

The following test instructions are useful:

- `pvarTest(R)` is an instruction testing the truth of `R`. Execution of this test will leave all values of variables unchanged. For instance `pvarTest(eqNN(e2bcb11,3bch3))` will return `true`. In contrast the test `pvarTest(eqBS(e2bcb11,3bch3)` must return `false` because equality is tested between two constants for BG(4) resp. BG(3).

3.1.2 Assignment instructions

Having described all actions that perform a real test first, we will proceed with a repertoire of actions that will always return `true` by default.

- `pvarAssign(v17 = Q)` is an instruction. The idea is simply that expression `Q` is evaluated (thus denoting some bit sequence) and that exactly that bit sequence is assigned to the variable `v17`.

3.2 Target interface actions

The target interface used in our subsequent discussions allows for screen output and synchronized interactions at a (synchronized) port.

3.2.1 Asynchronous screen output instructions

The screen is a device familiar to most computer users. In our example only very few instructions for screen control are taken into account. We use screen output in order to be able to provide very simple examples of program behavior involving program variables and assignments to program variables. It is plausible to assume that screen output takes place in an asynchronous fashion. There may be some buffering going on after the program's write instruction has been processed and before the screen user notices any effects.

- `pvarWriteNl(Q)` will be executed by writing the numerical value of the bit sequence contained in `Q` in decimal notation, followed by a new line.

3.2.2 Synchronized port interactions

A synchronized port is identified by a natural number indicating a (logical) point where behaviors may interact. In particular a program during execution constitutes a behavior and it may interact with other behaviors at a synchronized port. Complementary to a behavior interacting at a port there needs to be another behavior interacting at the same time in a matching fashion. The other behavior may also be the behavior of a program. This mechanism exists in the program notations OCCAM and Ada.

Synchronized port interactions are so-called blocking actions. A program has to wait (is blocked) until these actions have been sucessfully completed. Because other behaviors are essential for the adequate execution of synchronized port interactions, the amount of time needed to wait will (in the majority of cases) be outside the control of the waiting program. Algorithms using synchronized port interactions are not 'waitfree'.[13]

In the case of synchronized port interaction both input and output are blocking. In the case of asynchronous port interactions, however, only input actions are blocking (assuming that the buffer capac-

[12] We will use instances of variables (e.g. `v17`) where arbitrary variables can occur, inviting the reader to make the obvious generalization him/her self.

[13] A multi-threaded program is waitfree if different threads can always proceed independently towards the completion of their respective tasks. In particular if one thread stagnates all other threads can still run and successfully termination, without any disproportional loss in execution time. If stagnation of a thread leads to a moderate decrease in overal processing speed, while allowing all other thread to proceed in a regular fashion the algorithm being analysed is said to allow graceful degradation (with respect to thread failure). Graceful degradation is always a sign of robustness.

ities of intermediate buffers are sufficiently large, or even infinite). In the practically unavoidable case of limited buffer capacities output actions can be blocking too, even in the asynchronous case. As the buffer itself may exhibit a complicated behavior it is often formally specified with a specification notation admitting a form of behavior extraction. A very useful viewpoint is that at both ends of the asynchronous buffer synchronous port interactions take place.

In such a case the needed complementary behavior is that of an asynchronous buffer capable of storing incoming or outgoing messages. The buffer may be part of some hardware. At the other side of that buffer a second program may be interacting at a second port. In this fashion two program behaviors can interact in an asynchronous fashion. The interaction is mediated by hardware (or software of course) in between. Both programs interact in a synchronous fashion with that intermediate hardware. At a higher level of abstraction the interaction between the two programs can be understood in terms of asynchronous communication, forgetting all details of the intermediate mechanisms.[14] Ports are denoted as follows: p267 denotes port number 267. When communicating with a port a behavior exchanges a bit sequence with another behavior. The length of the sequence is a parameter for the input action as well as for the output action. We assume that ports are numbered in a uniform way and that port names are unique with respect to a platform at least. Between different platforms synchronized interaction is a meaningless abstraction.

- `pvarSynchInput(45bp17,v29)` is an action that is executed by means of taking an input along synchronized port number 17, casting the value received to length 45 and then assigning it to variable `v29`. In case no matching behavior is present at port 17 this input action is executed by waiting until an appropriate partner behavior has appeared. If that does not happen a deadlock has occurred.

- `pvarSynchOutput(8bp5,Q)` is an action that is executed by means of producing the value of `Q` as an output along synchronized port number 5, (casting the value before output to length 8). In case no matching behavior is present at port 5 this output action is executed by waiting until an appropriate partner behavior has appeared. If that does not happen a deadlock has occurred.

  At port 9 the following actions can interact synchronously (or synchronize or communicate) for instance: `pvarSynchOutput(11b9,Q)` and `pvarSynchInput(11b9,v5)`. The effect of this synchronisation is the same as the effect of thge assignment `pvarAssign(v5 = (11b)Q)`.

  If the message size (11 in this case) or the port name (9 in this case) differ,[15] no communication is possible.

3.3 Advanced control instructions

We will not make use of any advanced control instructions here. In case more expressive program notations are needed such instructions are easily added. Projection semantics will prescribe a transformation into PGLA programs removing the advanced control instructions in favor of PGLA control instructions and (in most cases) instructions from the reply co-service interface needed to access the service of a coprogram (the stack in the case that advanced control instructions for recursion have been removed, the thread vector in the case that advanced control instructions for multi-threading were used and so on). We will not make use of any advanced control instructions here. In case more expressive program notations are needed such instructions can be simply added.

3.4 Reply co-service interface actions

We will not make use of a particular coprogram offering a reply service to the programs discussed. If, however, the program notation is extended for instance from PGLApv to PGLDgmt+pv (PGLD with goto's, multi-threading, and program variables) the actions of the service interface of the thread vector coprogram

[14] In Java Programming in terms of sockets and pipes makes use of this abstraction.

[15] For instance `pvarSynchOutput(11b9,Q)` and `pvarSynchInput(12b9,v5)`.

(or any more appropriate modification of it depending on the particular form of multi-threading needed) have to be included in a reply co-service interface.

4 Programs over Σ_{pv1}

PGLA(Σ_{pv1}) is the family of PGLA programs (see [2]) using atomic instructions from the interface Σ_{pv1}. Except for the tests and assignments this interface can be considered a lower interface.[16] Here are some example programs.

1. $x1 =$

```
pvarAssign(v5 = 4bc7);
pvarWriteNl(v5);
pvarAssign(v2 = lengthOf(lengthOf(v5)));
pvarWriteNl(v2);
pvarAssign(v2 = plus(v2,div(v5,2bcb10)));
pvarWriteNl(v2);
pvarWriteNl(v4);
+pvarTest(eqNN(v2,v5));
pvarWriteNl(6bc3);
!
```

2. $x2 =$

```
pvarAssign(v1 = 5bc0);
+pvarTest(eqBS(v1,4bc0));
#3;
pvarWriteNl(v2);
pvarWriteNl(mult(v1,10bcb0010));
pvarAssign(v2 =
(div(2exp(plus(v2,1)),mult(v3,1));
pvarWriteNl(minus(v2,2bc3));
pvarAssign(v1 = mult(v1,v2));
-pvarWriteNl(v1);
#3;
-pvarWriteNl(v4);
#;
!
```

3. $x3 =$

```
pvarAssign(v0 = minus(v0,10));
#4;
(
pvarAssign(v0 = plus(v0,31));
-pvarTest(lessNN(7bc29,plus(v0,3)));
!;
pvarWriteNl(v0);
pvarAssign(v0 = v4)
)ω
```

4.1 Intelligent Artificiality

Artificiality is what machines can do, intelligent artificiality (IA) takes place if people perform or learn tasks particularly suited for machines. A prominent example of IA is school arithemetic. Children in school are taught to perform computational tasks which may be hard for them but are trivial for most modern computers. To determine the output of a (computer) program with pen and paper is a matter of IA as well.

It is a reasonable viewpoint that in order for a person to claim the (mechanical) understanding of a program notation he/she must be able to execute its programs by hand (in principle). Any actual program written in the program notation (which is given of course together with its semantics) poses an IA problem. This is called the implicit IA problem of a program notation.[17] The ability to solve the implicit IA problem for a program notation constitutes an operational understanding of that program notation.

Projection semantics followed by behavior extraction and other semantic techniques can be understood as general techniques for solving the 'program execution by hand' problem for a program notation (in principle).

The implicit IA problem posed by PGLA(Σ_{pv1}) is non-trivial. Working with an instance of PGLA already further projections are not necessary. Behavior extraction suffices to solve the implicit IA problem.

[16] In [2] the notation $\mathrm{PGLA}_{\Sigma_{pv1}}$ has been used. In the present context brackets are preferable over subscripts, however. Upper and lower interfaces were discussed in [3]). Instructions in a lower interface should be considered requests for other system components. It is assumed that all instructions will return a boolean value upon completion of their execution.

[17] It should be noticed that the ability to write programs for a given program notation by no means constitutes an IA problem. Currently program writing is mainly a matter of either human intelligence or compiler execution. A compiler, however needs a program as its input.

One may criticize the definitions below for being too 'formal'. Taking the intention to solve the implicit IA problem for PGLA(Σ_{pv1}) as the key objective, competing stories will have to provide a more efficient explanation on how to solve the implicit IA problem in this case. Currently the author is unaware of the existence of such more efficient explanations.

5 Behaviors of PGLA programs with data

5.1 Behavior extraction operators

The behavior of a PGLA(Σ_{pv1}) program x depends on a valuation σ because x may contain program variables without prior initialization to some constant value. With

$$|x|_{pglaPv(\sigma)}$$

we denote the behavior of x given valuation σ. On the assumption that by default PGLA(Σ_{pv1}) programs have their variables initialized at the empty sequence we may write

$$|x|_{pglaPv} = |x|_{pglaPv(\emptyset)}.$$

The parametrized behavior extraction operator $|-|_{pglaPv(\sigma)}$ is defined by means of a collection of equations (or better: rewrite rules). For brevity of notation we will write $|x|_\sigma$ instead of $|x|_{pglaPv(\sigma)}$.

The parametrized behavior extraction operator $|-|_{pglaPv(\sigma)}$ serves as an auxiliary operator for the definition of its non-parametrized instantiation $|-|_{pglaPv}$.[18] The equations below should be understood in the context of program algebra. The program object equations of [2] are needed to turn a program into a sequence of instructions first. That justifies the absence of equations explicitly addressing the behavior of programs with repetition.

The atomic instructions are grouped int three containers: `pvarTests` is the collection of atomic instructions of the form `pvarTest(R)`, `pvarAssignments` is the collection of actions of the form `pvarAssign`, `pvarSynComs` denotes the input actions as well as the ouput actions, and `pvarWrites` contains the instructions of the form `pvarWriteNl(q)`.

5.2 Behavior extraction equations

With `i` we denote a silent step. This is a behavior which enacts no change in state or any externally visible effect during its execution. Of course the execution can take some time.

5.2.1 Semantic equations for termination instructions

$$|!|_\sigma = \mathtt{i}$$

$$|!;X|_\sigma = \mathtt{i}$$

$$|\#|_\sigma = M$$

$$|\#;X|_\sigma = M$$

5.2.2 Semantic equations for the jump instructions

The case of the jump instructions requires a case distinction on the counter of the jump. In case that counter is zero, a divergence will occur. In case that counter is one, at least one further instruction should be performed, otherwise an error occurs. In case the counter exceeds two, the program should contain at least two subsequent instructions; otherwise the program will produce an error. In these equations k ranges over the natural numbers.[19]

$$|\#0|_\sigma = D$$

$$|\#0;X|_\sigma = D$$

$$|\#1|_\sigma = |\#|_\sigma$$

[18] It is an interesting question whether and how the use of an auxiliary behavior extraction operator for carrying a valuation can be avoided. We have concluded that it can be avoided in several ways, but only at the cost of readability.

[19] One may argue that the equations for #0 are about termination as well, considering divergence as yet another form of termination. m the point of view of program algebra the nature of the constant D is not important. It is some constant in the theory of behaviors one uses, suitable for denoting a system that can wait forever, not performing any externally observable actions.

$$|\#1; X|_\sigma = |X|_\sigma$$

$$|\#k+2|_\sigma = |\#|_\sigma$$

$$|\#k+2; u|_\sigma = |\#|_\sigma$$

$$|\#k+2; u; X|_\sigma = |\#k+1; X|_\sigma$$

5.2.3 Semantic equations for atomic tests

Because tests may contain aritmetical expressions and division by 0 cannot be ruled out beforehand, the case of a test leading to an error must be dealt with explicitly.

$$|+a|_\sigma = |-a|_\sigma = |a|_\sigma = |\#|_\sigma$$

$$|a; X|_\sigma = |X|_\sigma$$

$$|+a; u|_\sigma = |+a; u; \#|_\sigma$$

$$|-\texttt{pvarTest(R)}; X|_\sigma = |+\texttt{pvarTest(neg(R))}; X|_\sigma$$

$$|+\texttt{pvarTest(R)}; u; X|_\sigma = |u; X|_\sigma \lhd \texttt{eval}_\sigma(\texttt{R}) \rhd |X|_\sigma$$

$$P_1 \lhd \texttt{M} \rhd P_2 = ||\#||$$

$$P_1 \lhd \texttt{true} \rhd P_2 = P_1$$

$$P_1 \lhd \texttt{false} \rhd P_2 = P_2$$

5.2.4 Semantic equations for atomic assignments

In the equations of this group a ranges over `pvarAssignments`

$$|+a|_\sigma = |-a|_\sigma = |a|_\sigma = |\#|_\sigma$$

$$|+a; X|_\sigma = |a; X|_\sigma$$

$$|-a; u|_\sigma = |\#|_\sigma$$

$$|-a; u; X|_\sigma = |a; X|_\sigma$$

$$|\texttt{pvarAssign(v48 = Q)}; X|_\sigma = |X|_{\sigma[\texttt{eval}_\sigma(\texttt{Q})/\texttt{v48}]}$$

(provided $\texttt{eval}_\sigma(\texttt{Q}) \neq \texttt{M}$, $||\#||$ otherwise)

5.2.5 Semantic equations for synchronous communications and console output actions

In this group of equations a ranges over `pvarSynComs` $\cup$ `pvarWrites`.

$$|+a|_\sigma = |-a|_\sigma = |a|_\sigma = |a; \#|_\sigma$$

$$|+a; X|_\sigma = |a; X|_\sigma$$

$$|-a; u|_\sigma = |a; \#|_\sigma$$

$$|-a; u; X|_\sigma = |a; X|_\sigma$$

$$|\texttt{pvarWriteNl(Q)}; X|_\sigma =$$

$$\texttt{screenWriteNl(bs2nn(eval}_\sigma\texttt{(Q)))} \cdot |X|_\sigma$$

(provided $\texttt{eval}_\sigma(\texttt{Q}) \neq \texttt{M}$, $||\#||$ otherwise)

$$|\texttt{pvarSynchOutput(45b18p,Q)}; X|_\sigma =$$

$$\texttt{s}_{18}\texttt{((45b)eval}_\sigma\texttt{(Q))} \cdot |X|_\sigma$$

(provided $\texttt{eval}_\sigma(\texttt{Q}) \neq \texttt{M}$, $||\#||$ otherwise)

$$|\texttt{pvarSynchInput(45b18p,v7)}; X|_\sigma =$$

$$\sum_{w \in \texttt{BG(45)}} \texttt{r}_{18}(w) \cdot |X|_{\sigma[w/\texttt{v7}]}$$

In these equations the action $\texttt{s}_n(w)$ (for a bit sequence w) denotes a synchronous send action at port n, whereas $\texttt{r}_n(w)$ denotes a sychronous read action at port n. The summation $\sum_{w \in \texttt{BG}(k)}(X_w)$ offers a choice between any of the behaviors X_w. In most cases the context of a behavior will decide which choce is going to be made during a run of the behavior.[20]

It should be noticed that there is no real difference between the equations for synchronous output and the asynchrounous write actions. The difference is noticed only when the resulting behaviors are placed in an appropriate context.

[20]The summation over all $w \in \texttt{BG}(m)$ is best viewed as a shorthand for a very big but finite sum of alternatives. Behavior extraction equations are written at the level of informal mathematics, allowing the status of variables to be formally unanalysed. Of course the equations can be written within an entirely formlized syntax as well. For the pupose of semantic clarification that is not essential, however. This interpretation follows standard practice in process algebra in the style of ACP of [1].

5.2.6 Divergence for non-trivial loops

The above equations should be used to obtain successive steps of the behavior of a program object X. The equations may be applied infinitely often without ever generating a piece of atomic behavior. In that case the program has a non-trivial loop. In this case it will be identified with D. By doing so we obtain for instance: $|(\#1)^\omega|_\sigma = |\#0|_\sigma$ and $|\texttt{pvarAssign(v2 = [])};(\#2;\texttt{pvarWriteNl(v2)})^\omega|_\sigma = |\#0|_\sigma$.

If this identification of behaviors exhibiting a non-trivial loop with divergence is omitted as a transformation rule, the set of all equations mentioned so far constitutes an orthogonal term-rewrite system.

5.2.7 Remark

One may criticize the behavior extraction equations as given above for the use of subscripts in the extraction operator, the use of coprograms having been advocated for similar purposes in program algebra elsewhere. Two comments on this critique are possible: (i) in other cases coprograms can be removed in favor of the use of extraction operator subscripts as well; the coprograms can serve as a systematic foundation for the use of these subscripts. (ii) If the theory of behavior is taken from process algebra, the so-called process prefix operator is able to capture exactly the combination of alternative composition parametrized by a set of data and the modification of a valuation inside a behavior expression used in case of input actions. It follows that by using more process algebra (in the role of behavior theory) the use of subscripts in behavior extraction equations can be entirely avoided while keeping the equations very close to the original equations for PGLA.

Process algebra fails to have the flexibility, however, to incorporate a stack (for projecting returning jump instructions and return instructions) or a thread vector (arising in the projection of multitheading instructions). Therefore it is not the case that the use of coprograms can be removed in favor of a more sophisticated use of process algebra.

6 Cumulative console output

Behavior extraction provides an abstract view of the meaning of a program. It is sufficiently abstract to merit the title 'semantics'. It is far from sufficiently abstract for many conceivable purposes. We provide a very simple case of a more abstract semantics based on behavior extraction. Consider the subnotation obtained from our PGLA instance by disallowing synchronized input and output. All behavior of a program is now shown via console output actions (`pvarWriteNl(q)`). The extracted behavior of a program is then a finite or infinite sequence of print actions for natural numbers in digital notation. This follows from the fact that all decisions (tests) can be executed independently of the execution history of the programs. Cumulated console output semantics simply turns this sequence into a list of DNN's (digital NN expressions, see 2.5) separated by comma's. We write $|x|_{pglaPvCco}|$ for this sequence. $|x|_{pglaPvCco}$ is easily obtained from $|x|_{pglaPv}$:

$$|x|_{pglaPvCco} = ||x|_{pglaPv}|_{Cco}$$

$$|\texttt{i}|_{Cco} = []$$

$$|\texttt{screenWriteNl}(w)|_{Cco} = [w]$$

$$|\texttt{screenWriteNl}(w) \cdot P|_{Cco} = [w] \circ |P|_{Cco}$$

The sequence $[w1] \circ [w2] \circ ... \circ [wk]$ is then written as w1,w2,..,wk, thus obtaining a finite or infinite comma separated sequence.

6.1 Examples

When calculating the semantics of specific examples the following notation on program objects: `Del(k,`x`)` removes the first `k` instructions from the prgram object x. For instance `Del(5,`x`)` denotes the program object obtained from x by deleting the first 5 actions.[21] If x has length 5 or less deletion produces the program #. The programs of section 4 are useful candidates for generating cumulative console output. Two cases are considered in detail, calculating $|x3|_{Cco}$ is left as an exercice for the arduous reader.

[21] When deleting actions it may be necessary to unfold repetitions.

6.1.1 Program $x1$

Now $|x1|_{Cco} = ||x1|_{pglaPv}|_{Cco} = ||x1|_{\emptyset}|_{Cco} =$

$$||\mathtt{Del(1,}x1\mathtt{)}|_{\mathtt{[0111/v5]}}|_{Cco} =$$

$$|\mathtt{screenWriteNl(111)}\cdot|\mathtt{Del(2,}x1\mathtt{)}|_{\mathtt{[0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ ||\mathtt{Del(2,}x1\mathtt{)}|_{\mathtt{[0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ ||\mathtt{Del(3,}x1\mathtt{)}|_{\mathtt{[10/v2,0111/v5]}}|_{Cco} = \mathtt{[7]} \circ$$

$$|\mathtt{screenWriteNl(10)}\cdot|\mathtt{Del(4,}$$

$$x1\mathtt{)}|_{\mathtt{[10/v2,0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ ||\mathtt{Del(4,}x1\mathtt{)}|_{\mathtt{[10/v2,0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ ||\mathtt{Del(5,}x1\mathtt{)}|_{\mathtt{[10/v2,}}$$

$$\mathtt{0111/v5]\,[101/v2]}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ ||\mathtt{Del(5,}x1\mathtt{)}|_{\mathtt{[101/v2,0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ \mathtt{[5]} \circ ||\mathtt{Del(6,}x1\mathtt{)}|_{\mathtt{[101/v2,0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ \mathtt{[5]} \circ \mathtt{[0]} \circ$$

$$||\mathtt{Del(7,}x1\mathtt{)}|_{\mathtt{[101/v2,0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ \mathtt{[5]} \circ \mathtt{[0]} \circ$$

$$||\mathtt{Del(9,}x1\mathtt{)}|_{\mathtt{[101/v2,0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ \mathtt{[5]} \circ \mathtt{[0]} \circ ||!|_{\mathtt{[101/v2,0111/v5]}}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ \mathtt{[5]} \circ \mathtt{[0]} \circ |\mathtt{i}|_{Cco} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ \mathtt{[5]} \circ \mathtt{[0]} \circ \mathtt{[]} =$$

$$\mathtt{[7]} \circ \mathtt{[2]} \circ \mathtt{[5]} \circ \mathtt{[0]} = 7250$$

6.1.2 Program $x2$

Now $|x2|_{Cco} = ||x2|_{pglaPv}|_{Cco} = ||x2|_{\emptyset}|_{Cco} =$

$$||\mathtt{Del(1,}x2\mathtt{)}|_{\mathtt{[00000/v1]}}|_{Cco} =$$

$$|\mathtt{screenWriteNl(0)}\cdot|\mathtt{Del(4,}x2\mathtt{)}|_{\mathtt{[00000/v1]}}|_{Cco} =$$

$$\mathtt{[0]} \circ ||\mathtt{Del(4,}x2\mathtt{)}|_{\mathtt{[00000/v1]}}|_{Cco} =$$

$$\mathtt{[0]} \circ |\mathtt{screenWriteNl(0)}\cdot|\mathtt{Del(5,}$$

$$x2\mathtt{)}|_{\mathtt{[00000/v1]}}|_{Cco} =$$

$$\mathtt{[0]} \circ \mathtt{[0]} \circ ||\#||_{Cco} = 00$$

7 The Halting problem

The subset $\mathrm{PGLA}(\Sigma_{pv1be})$ of $\mathrm{PGLA}(\Sigma_{pv1})$ is obtained by considering bounded numerical expressions in assignments only, and refraining from the use of the concatenation $\circ$ in assignments. Bounded numerical expressions have the form `(kb)nn2bs(Q)` or `((ekb)nn2bs(Q))` with $\mathtt{k} \in \mathrm{NN}$. The effect of these drastic restrictions is that the number of values that each individual variable can have during a computation is bounded. This implies that a program can be in finitely many states only. As the state space can be systematically searched it becomes decidable whether a $\mathrm{PGLA}(\Sigma_{pv1be})$ program will terminate.

In the general case, of $\mathrm{PGLA}(\Sigma_{pv1})$ the situation is entirely different. This can be seen in several ways. For instance is it possible to simulate register machines in this language. For register machines the halting problem is known to be undecidable. Another way to prove the fact is to consider diophantine equations. One easily writes in $\mathrm{PGLA}(\Sigma_{pv1})$ a program for making an exhaustive search for the solution of a diophantine equation, failing to terminate if there is no such solution. This problem is known to be undecidable.

8 References

[1] J.A. Bergstra and J.-W. Klop. Process algebra for synchronous communication. *Information and Control*, 60 (1/3):109–137, 1984.

[2] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998. To appear in Formal Aspects of Computing.

[3] J.A. Bergstra and M.E. Loots. Abstract data type coprograms. Technical report, Department of Philosophy, Utrecht University, 1999. In: Bergstra/Loots, Programs, Interfaces and Components, Artificial Intelligence Preprint Series 006.2.

A Exercises

1. Determine the following values in BS:

   (a) `eval`$_{\texttt{[1010/v3,00010/v8]}}$`(v3 ∘ nn2bs(mult(bs2nn(v8), lengthOf(v4 ∘ v3))))`

   (b) `eval`$_{\texttt{[[]/v2,0/v9]}}$`(v3 ∘ nn2bs(minus(bs2nn(v9),lengthOf(v2))))`

2. Give a completely precise definition of term evaluation for terms of sort BEXP over the BNF given in this paper. (Required is a so-called inductive definition. All other sorts are needed as well.)

3. Design an extension of the BNF taking into account that constants may be written in hexadecimal notation as well. Then provide 2 examples of closed expressions using hexadecimal notation (as well as the evaluation results of these expressions).

4. Calculate $|x3|_{Cco}$, with the program text $x3$ taken from 4.

5. Write PGLA(Σ_{pr1}) program y computing the sum of the first k squares (for $k = 1$ return 1, for $k = 2$ return 5 etc). The program should write all sums in increasing order to console. Check that the first 3 console outputs are correct by calculating $|y|_{Cco}$. (Hint: write the program in PGLDg first, then translate it into PGLA.)

6. Design a set of semantic equations for PGLDg(Σ_{pr1}). Avoid the transformations via PGLD, PGLC and PGLB in favor of a direct description of behavior extraction on this dialect of PGLDg.

Basic Multi-Competence Programming

Jan A. Bergstra[1,2] Inge Bethke[1] Alban Ponse[1]

[1] *University of Amsterdam, Programming Research Group*
http://www.wins.uva.nl/research/prog/

[2] *Utrecht University, Department of Philosophy*
http://www.phil.uu.nl/eng/home.html

Version 2.0

Abstract. BMCP (Basic Multi-Competence Programming) is a paradigm for teamwork for simple programming tasks. BMCP is simplified in the sense that programming tasks require no more than implementing a given functional specification. BMCP uses CVN (Conditional Value Notation) as a format for its functional specifications. BMCP displays seven competences, this collection being derived from an underlying family of four so-called program assessment perspectives. BMCP requires fixed agreements on interfacing, input and output. This version (2.0) replaces the earlier version 1.0 of this document.

Contents

1 Introduction

Multi-Competence Programming (MCP) is a programming framework meant for larger teams. Its objective is to get a firm grip on quality control supplementary to, and as much as possible simultaneously with, program construction. MCP is primarily an educational framework, delineated along 17 fresh acronyms. Taking the program algebra (see [5]) definition of what a program is as a point of departure, four *program assessment perspectives* emerge, each of which gives rise to a quality control *competence*. In terms of program algebra, a program is viewed as a text. The algorithmic content of this text is to be found via a translation (projection) to a text in a low level language (e.g., PGLA from [5]). The latter text can be viewed as a sequence of elementary instructions.

According to this view a program is a description of a sequence of instructions, the description perhaps given in a high level language which must be projected

down to a low level notation in order to determine its (algorithmic) meaning. This view mainly matters if semantic issues are at stake. When programming in a known program notation, all semantic issues having been resolved already, the program algebra view must be complemented with other views of a pragmatic nature. Four of these views, or program assessment perspectives, are

> the *black box test* perspective,
> the *white box test* perspective,
> the *white box review* perspective,
> and
> the *white box verification* perspective.

Each of these perspectives gives rise to a competence. The competence refers to the (proven, accredited) ability to assess software from that perspective. It also refers to a role that a person or team can play in a software development activity. We get then the following acronyms, referring to the perspectives mentioned above:

BBTC:	Black Box Test Competence
WBTC:	White Box Test Competence
WBRC:	White Box Review Competence
WBVC:	White Box Verification Competence

These four perspectives and competences are further discussed in Sections 2 and 3. In a program development project based on MCP each of these competences is put in the hands of a different team. Because these competences are so totally different in practice, each having its own expert community, journals, guru's etc., we use the phrase multi-competence programming for a project planning involving all or most of these.

Having established the quality control related competences, three more competences are needed to keep a project up and running:

PCC:	Program Construction Competence
PMC:	Project Management Competence
KMC:	Knowledge Management Competence

Altogether seven competences emerge. The teams responsible for these competences are denoted with:

PCT:	Program Construction Team
PMT:	Program Management Team
KMT:	Knowledge Management Team
BBTT:	Black Box Test Team
WBTT:	White Box Test Team
WBRT:	White Box Review Team
WBVT:	White Box Verification Team

The MCP method has immediate implications for a class room setting. When a group of students is constructing programs in order to develop and/or demonstrate their programming abilities, the assessment of this output requires some care.

In fact each student ought to be role aware. When produced programs are judged it must be known:

- from what perspective(s) this judgement will be made;
- how the persons responsible for making the judgement will approach their task.

MCP in its simplest form poses a programming problem in terms of a functional specification that has to be implemented. The program construction activity produces *proposed implementations*. Only the management role can upgrade a proposed implementation to the level of an implementation. This will always require some form of quality control. All major perspectives on product oriented (not: process oriented) quality control are represented in the scheme of four perspectives discussed above.

MCP uses CVN (Conditional Value Notation) as its format for functional specifications. This notation is introduced in Section 4. The advantages of CVN are:

- very concise and compact (though limited in its usability),
- captures the main error conditions: abrupt program abortion (leading to M, the meaningless state) and program divergence (leading to D, the divergent state),
- it uses conditions and quantifiers in a form which easily deals with D and M as mentioned above.

When programming tasks are simplified to the implementation of a given functional specification, avoiding any feedback to the specification itself, the framework is termed BMCP, Basic MCP. Otherwise it is called AMCP for Adaptive MCP. AMCP views the production of the functional specification as a competence as well. The specification production goes through various repetitions, including a range of program construction and assessment activities, which in turn lead to a next phase of the specification. This involves yet another competence, the *requirements analysis competence*. AMCP can only take place in a significantly more involved setting where the programs are used to support in reaching further objectives.

BMCP is considered a useful "educational" approximation of AMCP, even if BMCP is not so practical because AMCP happens to occur more in industrial projects. As an educational paradigm we first concentrate on BMCP and only thereafter consider projects involving AMCP. BMCP contains the bare essentials of programming and of programming related quality control.

There is a huge literature on software management describing numerous tools, techniques, methodologies and standards. We mention [1], [7], [9], [10], [13] and [18].

2 Four Perspectives on a Program

The description of 'what a program is' in program algebra [5] by no means exhausts the valid perspectives one may have on programs. Accepting the viewpoint that a program is a text which is given algorithmic content by a projection into a low level program notation, still four different perspectives on a program can be determined:

Black box (behavioral) perspective. A program is a black box (e.g., a compiled binary, not a "Black box", which is orange). All one can do is to run it and

to perform black box tests to assess its behavior. (See e.g., [17].)

White box (behavioral) perspective. Program P is a known text. Properties of P are taken from its specification. These properties are assessed by means of black box tests and white box tests. White box tests take advantage of knowledge of the program text in order to spot difficulties more efficiently than in black box (random) mode.

White box literary perspective. The program text is to be assessed by means of informal but systematic judgement of the text (code walkthrough etc., see e.g., [12]).

White box verification perspective. The program is a text characterized by properties which have to be demonstrated in a rigorous (formal) fashion. Preferably, proofs are machine readable and machine checked. (Cf. [2].)

Each of these has its value in practice. Some implications of these perspectives are the following activities, mainly used for quality control:

Black box testing. Black box tests are generated from the specification (possibly automatically). A test environment is produced in which many tests can be run (automatically) on any proposed implementation of the specification. It is essential that proposed implementations strictly adhere to boundary conditions regarding interfaces.

White box testing. White box test can often also be generated automatically. It is important that adequate coverage is achieved, i.e., as much as possible all branches of the program must be visited by at least one test run.

Program verification. Proof generation and proof checking (if possible computer supported or automatically).

White box critical review. Here the program is understood as a pseudocode containing the solution of a problem (i.e., how to best implement the specification). In this perspective the fundamental merits of the proposed solution are considered. Speed of calculation, memory usage, clarity, independence of problematic language features, exploitation of interesting language/machine options etc.

3 Competence partitioning in BMC

In a BMC project seven different roles are to be distinguished. These roles are performed by a team. A team may consist of just one person, it is even possible that a person performs the work of different teams at the same time. The following teams can exist:

KMT (the knowledge management team) investigates the following questions:

- What kind of algorithm can be used; what can be expected/achieved?
- Which program constructions will be needed?
- What kind of verification techniques can used? how difficult will this be?
- How can BB/WB tests be derived in the case at hand?

KMT will not only produce some rationale for a proposed implementation but also a context in which the level

of ambition of the proposal can be assessed.

PCT (the program construction team) generates a program and applies modifications if the interaction with any of the other teams makes that plausible.

BBTT (the black box test team) produces test beds for proposed implementations (given the specification of the required functionality).

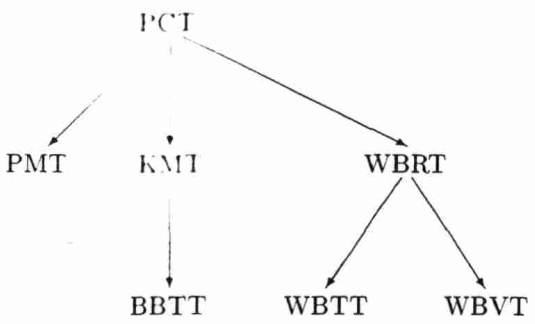
WBTT (the white box test team) delivers white box tests, preferably also in a largely automated setting.

WBRT (the white box review team) produces assessment of proposed implementations based on code inspections of various kinds.

WBVT (the white box verification team) produces correctness proofs.

PMT (the project management team) surveys the entire operation and manufactures the final deliverable containing programs as well as results of BBTT, WBTT, WBRT, WBVT, and KMT.

Amongst these teams, we discern the following hierarchy:



As suggested above, PCT takes the lead. PCT members intend to own the software. PCT uses KMT to produce background knowledge. KMT can manage BBTT as this requires specification expertise only. PCT uses WBRT for permanent assessment. WBRT in turn will guide both WBTT and WBVT (using insights of KMT). PCT uses the service of PMT to obtain better overall performance. For this reason PMT needs some power. This power is handed over by PCT in order to optimize PMT's role.

4 Conditional Value Notation

CVN, Conditional Value Notation, is a simple set of conventions to denote data values in the presence of possible errors. For each data type (data domain) V the presence of two exceptional values is assumed:

> D, a divergent value (the value of a diverging computation);
>
> M, a meaningless value (the value of an aborting computation).

Data values different from these exceptional values are called *good*. The predicate $g(\_)$ determines the good objects, i.e., $g(X)$ "holds" for some X in V if X is not D or M.[1] Thus

$$V = V_g \cup \{D, M\}$$

where $V_g = \{X \in V \mid g(X) \text{ holds}\}$. On V the symbol $=$ denotes equality, and $\neq$ denotes non-equality. With the special exception values D and M it often is simpler to use an asymmetric type of equality,

[1] Instead of regarding a predicate as determining a subset, one might consider it as a function to $\{T, F\}$ and set $g(X) = T$ if $g(X)$ holds, and $g(X) = F$ otherwise.

called *left sequential equality* with notation $\circ\!\!=$. The idea is that $X \circ\!\!= Y$ gets its meaning by interpretation from left to right. For example, $M \circ\!\!= X$ equals M because equating the meaningless value with anything else is "meaningless". On $V = V_g \cup \{D, M\}$ left sequential equality is defined by:

$D \circ\!\!= X$	$=$	D	
$M \circ\!\!= X$	$=$	M	
$X \circ\!\!= D$	$=$	D	if $g(X)$ holds
$X \circ\!\!= M$	$=$	M	if $g(X)$ holds
$X \circ\!\!= X$	$=$	T	if $g(X)$ holds
$X \circ\!\!= Y$	$=$	F	if $g(X)$ and $g(Y)$ hold, and $X \neq Y$

For $X \circ\!\!= Y = F$ we also write $X \circ\!\!\neq Y$. Furthermore, in the case that other relations are defined over V_g, these can always be extended to V (including the error values) in a similar way. For example, in the domain $\mathbb{N}$ of the natural numbers the relation $<$ ("smaller than") can be extended to a relation $\propto$ over $V = \mathbb{N} \cup \{D, M\}$ as follows:

$D \propto X$	$=$	D	
$M \propto X$	$=$	M	
$X \propto D$	$=$	D	if $g(X)$ holds
$X \propto M$	$=$	M	if $g(X)$ holds
$X \propto Y$	$=$	T	if $g(X)$ and $g(Y)$ hold, and $X < Y$
$X \propto Y$	$=$	F	if $g(X)$ and $g(Y)$ hold, and $X \geq Y$

It is evident how addition $(+)$ should be extended to $\circ\!\!+$ in $\mathbb{N} \cup \{D, M\}$. It is perhaps less evident how to extend subtraction $(-)$ to $\circ\!\!-$ in $\mathbb{N} \cup \{D, M\}$. Here the typical difference with $\circ\!\!+$ is caused by subtraction being a *partial* function on $\mathbb{N}$, i.e., $X - Y$ is not defined if $X < Y$. This problem is solved in $\mathbb{N} \cup \{D, M\}$ by the following equations:

$$X \circ\!\!- Y = M$$

if $g(X)$ and $g(Y)$ hold, and $X < Y$

$$X \circ\!\!- Y = X - Y$$

if $g(X)$ and $g(Y)$ hold, and $X \geq Y$. Consequently, $\circ\!\!-$ is not a partial function on $\mathbb{N} \cup \{D, M\}$.

As a convention we shall further use the common relation and operation symbols if we are sure that their arguments cannot be error values. (However, as the last example showed, the result value can be an error.)

Exercises.

1. Spell out the complete definition for subtraction on $\mathbb{N} \cup \{D, M\}$.

2. A different view on subtraction on $\mathbb{N}$ is to consider this operation as a partial one that may *diverge*. The idea is that

   $$\begin{aligned} psub(X, Y) \quad = \quad & \\ & \textbf{if } X \circ\!\!= Y \textbf{ then } 0 \\ & \textbf{else } S(psub(X, S(Y))). \end{aligned}$$

   (This example is taken from [3].) Here S is the *successor* function: $S(X) = X + 1$. Thus $psub(X, Y) = X - Y$ if $X \geq Y$. Describe how *psub* can be defined on $\mathbb{N} \cup \{D, M\}$ if we set $S(D) = D$ and $S(M) = M$.

Four-Valued Conditions. In particular for the truth values T and F, instead of working with $V = \{T, F\}$ CVN proposes the use of $\mathbb{T}_4 = \{T, F, D, M\}$ (notations from [4]). We shortly elaborate on this setting, as to introduce the underlying logic over $\mathbb{T}_4$ that is useful for CVN. The logical

connectives relevant for CVN are:

$\neg$	(negation),
$\mathbin{\wedge\!\!\!\circ}$	(left sequential conjunction),
$\mathbin{\circ\!\!\!\vee}$	(left sequential disjunction),
$\circ\!\!\rightarrow$	(left sequential implication).

The (binary) connectives all have *right sequential* counterparts, e.g., $\wedge_\circ$ is called right sequential conjunction and is defined by $X \wedge_\circ Y = Y \mathbin{\wedge\!\!\!\circ} X$.

So-called *truth tables* for $\neg$, $\mathbin{\wedge\!\!\!\circ}$ and $\mathbin{\circ\!\!\!\vee}$ are

X	$\neg X$
M	M
T	F
F	T
D	D

$\mathbin{\wedge\!\!\!\circ}$	M	T	F	D
M	M	M	M	M
T	M	T	F	D
F	F	F	F	F
D	D	D	D	D

$\mathbin{\circ\!\!\!\vee}$	M	T	F	D
M	M	M	M	M
T	T	T	T	T
F	M	T	F	D
D	D	D	D	D

These tables provide the definition of the logical connectives in $\mathbb{T}_4$, e.g., $M \mathbin{\wedge\!\!\!\circ} F = M$ and $F \mathbin{\wedge\!\!\!\circ} M = F$. Notice that $\mathbin{\circ\!\!\!\vee}$ can be defined by *dualization*:

$$X \mathbin{\circ\!\!\!\vee} Y = \neg(\neg X \mathbin{\wedge\!\!\!\circ} \neg Y),$$

where the "=" is plain equality over $\mathbb{T}_4$. In this style the connective $\circ\!\!\rightarrow$ is defined by $X \circ\!\!\rightarrow Y = \neg X \mathbin{\circ\!\!\!\vee} Y$. As another example of the use of this logic consider $\mathbb{N} \cup \{D, M\}$: the relation $\gtrdot$ can now be defined by

$$X \gtrdot Y = \neg(X \lessdot Y).$$

Exercises.

3. Determine the values of
   (a) $T \circ\!\!\rightarrow (M \mathbin{\circ\!\!\!\vee} T)$.
   (b) $(T \mathbin{\wedge\!\!\!\circ} M) \mathbin{\circ\!\!\!\vee} D)$.
   (c) $T \mathbin{\wedge\!\!\!\circ} \neg(M \mathbin{\circ\!\!\!\vee} F)$.

4. Show that the connectives $\mathbin{\wedge\!\!\!\circ}$ and $\mathbin{\circ\!\!\!\vee}$ are *associative*, i.e. $X \mathbin{\wedge\!\!\!\circ} (Y \mathbin{\wedge\!\!\!\circ} Z) = (X \mathbin{\wedge\!\!\!\circ} Y) \mathbin{\wedge\!\!\!\circ} Z$, and $X \mathbin{\circ\!\!\!\vee} (Y \mathbin{\circ\!\!\!\vee} Z) = (X \mathbin{\circ\!\!\!\vee} Y) \mathbin{\circ\!\!\!\vee} Z$. (So we need not use brackets in repeated applications of each of these connectives.)

5. Let $V = V_g \cup \{D, M\}$ and consider the predicate g as a function from V to $\{T, F\}$. Let furthermore equality $(=)$ be given on V_g. Argue whether the expression

$$(\neg g(X) \mathbin{\wedge\!\!\!\circ} X) \mathbin{\circ\!\!\!\vee} (g(X) \mathbin{\wedge\!\!\!\circ} \neg g(Y) \mathbin{\wedge\!\!\!\circ} Y) \mathbin{\circ\!\!\!\vee} (X = Y)$$

defines left-sequential equality.

If_then and If_else. For the domain V (containing D, M) two operators are proposed by CVN:

$$p \cdot\!\!\rightarrow X \quad (p \text{ a four-valued condition}, X \in V)$$

$$X \mathrel{+\!\!\!\rightarrow} Y \quad (X, Y \in V)$$

The expression $p \cdot\!\!\rightarrow X$ is pronounced "if p then X", and $X \mathrel{+\!\!\!\rightarrow} Y$ is pronounced "if X else Y".

The operators $\cdot\!\rightarrow$ and $\twoheadrightarrow$ are determined by the following equations:

$$\begin{array}{|lcl|}\hline T\cdot\!\rightarrow X & = & X \\ F\cdot\!\rightarrow X & = & M \\ D\cdot\!\rightarrow X & = & D \\ M\cdot\!\rightarrow X & = & M \\ \hline\end{array}$$

and

$$\begin{array}{|lcl|}\hline X \twoheadrightarrow Y & = & X \text{ if } g(X) \text{ holds} \\ D \twoheadrightarrow X & = & D \\ M \twoheadrightarrow X & = & X \\ \hline\end{array}$$

So, $X \twoheadrightarrow Y$ equals X if X is a good value or if X diverges, otherwise it equals Y. We notice that $\twoheadrightarrow$ is *associative*, i.e.

$$X \twoheadrightarrow (Y \twoheadrightarrow Z) = (X \twoheadrightarrow Y) \twoheadrightarrow Z.$$

Therefore we can omit parentheses in repeated applications. Furthermore, to enhance readability we assume that $\mathbin{\mathrlap{\wedge}\vee}$, $\mathbin{\mathrlap{\vee}\wedge}$ bind stronger than $\cdot\!\rightarrow$, which in turn binds stronger than $\twoheadrightarrow$, so

$$\neg p \mathbin{\mathrlap{\wedge}\vee} q \cdot\!\rightarrow X \twoheadrightarrow Y = ((\neg p) \mathbin{\mathrlap{\wedge}\vee} q) \cdot\!\rightarrow X) \twoheadrightarrow Y,$$

and the brackets in $p \cdot\!\rightarrow (X \twoheadrightarrow Y)$ cannot be omitted.

The following identities follow easily by considering all possible values of p and q, and will be used in the sequel:

1. $p \cdot\!\rightarrow (X \twoheadrightarrow Y) = p \cdot\!\rightarrow X \twoheadrightarrow p \cdot\!\rightarrow Y$,
2. $p \cdot\!\rightarrow (q \cdot\!\rightarrow X) = p \mathbin{\mathrlap{\wedge}\vee} q \cdot\!\rightarrow X$.

With the operators $\cdot\!\rightarrow$ and $\twoheadrightarrow$, a two-sided conditional can be found as follows:

$$X \lhd p \rhd Y = p \cdot\!\rightarrow X \twoheadrightarrow \neg p \cdot\!\rightarrow Y.$$

(Cf. the notation **if** p **then** X **else** Y.)

Let V be given, and let $\Phi : V_g \rightarrow \mathbb{T}_4$ be a predicate on V_g (taking errors into account). There are two *quantifiers* that we want to use in CVN:

$\forall X \in V_g(\Phi(X))$ ("$\Phi(X)$ for all $X \in V_g$")
$\exists X \in V_g(\Phi(X))$ ("there exists $X \in V_g$ for which $\Phi(X)$")

We describe their values as follows:

$\forall X \in V_g(\Phi(X))$:
if $\Phi(X) = T$ for all $X \in V_g$, then T,
else, if $\Phi(X) = M$ for some $X \in V_g$, then M,
else, if $\Phi(X) = F$ for some $X \in V_g$, then F,
else, D.

$\exists X \in V_g(\Phi(X))$:
if $\Phi(X) = M$ for some $X \in V_g$, then M,
else, if $\Phi(X) = T$ for some $X \in V_g$, then T,
else, if $\Phi(X) = D$ for some $X \in V_g$, then D,
else, F.

Notice that quantification ranges over V_g, and that

$$\forall X \in V_g(\Phi(X)) = \neg \exists X \in V_g(\neg \Phi(X)).$$

Furthermore we shall sometimes use 'bounded' quantification over V_g. As an example, let $V_g = \mathbb{N}$ and $i \in \mathbb{N}$, then

$$\exists j \; 0 \leq j < i(\Phi(j))$$

expresses that there is a value j such that $0 \leq j$, $j < i$ and $\Phi(j)$ holds. (This is not a real extension: use of bounded quantification can be avoided at cost of readability).

The strength of CVN is its explicit dealing with the two fundamental error cases: abrupt abortion of a computation (leading to M) and computational divergence (modeled by D). CVN can be used to specify programs. This will work in a minority of cases, but this restricted area is of im-

portance.[2] In Sections 5 and 6 we consider two examples of program specification with CVN.

Exercises. Let $V_g = \mathbb{N}$ and let the relation $<$ (smaller than) be given.

6. Determine whether the following expressions are well-defined:
   (a) $T \cdot \rightarrow 6$
   (b) $3 \cdot \rightarrow 5$
   (c) $T \cdot \rightarrow T$
   (d) $(3 \underline{=}_{\sigma} 5) \cdot \rightarrow 2 \mapsto D$
7. Determine the *type* (data type, data domain) of the following expressions:
   (a) $T \cdot \rightarrow 6$
   (b) $F \cdot \rightarrow D$
   (c) $3 \underline{=}_{\sigma} 5$
   (d) $(3 \underline{=}_{\sigma} 5) \cdot \rightarrow T$
   (e) $(3 \underline{=}_{\sigma} 5) \cdot \rightarrow 2 \mapsto 7$
8. Argue whether the following holds:

$$\begin{aligned} X \propto Y \quad = \quad & g(X) \mathbin{\underline{\wedge}}_{\sigma} g(Y) \cdot \rightarrow (X < Y) \mapsto \\ & g(X) \mathbin{\underline{\wedge}}_{\sigma} \neg g(Y) \cdot \rightarrow Y \mapsto \\ & \neg g(X) \mathbin{\underline{\wedge}}_{\sigma} g(Y) \cdot \rightarrow X \mapsto \\ & \neg g(X) \mathbin{\underline{\wedge}}_{\sigma} \neg g(Y) \cdot \rightarrow X. \end{aligned}$$

   Can you find a shorter CVN expression that defines $\propto$?

Motivation for CVN. BCMP strongly depends on the availability of a functional specification of a piece of software beforehand. The task of the BCMP project is to obtain an implementation of that specification.

Clearly such specifications must be expressed in a precise notation. There is a very rich literature about specification notations for software. Notoriously lacking, however, are simple techniques that can deal with less elegant aspects such as errors, divergencies and partiality of operators. Algebraic specification use a non-monotonic logic, but can in practice be based on monotonic term rewriting (see [6]), pseudo-algebraic specifications (present for instance in ASF + SDF [11]) use non-monotonic term rewriting instead. The logic of non-monotonic term rewriting is far from trivial unfortunately. Industrial strength methods like VDM use a three valued logic (see [16, 14]).

The notation CVN, as introduced in the present paper, can be used for the description of simple program functionalities. At the same time it deals with two kinds of errors: divergence (D) and abortion (M). CVN has been stripped from all aspects needed to make it applicable for more complex examples. No form of modularity is supported at all. This reflects the authors opinion that error handling cannot be viewed as an 'add on feature', whereas modularity does constitute an 'add on feature'. In particular the addition of error mechanisms to a complex specification notation may introduce intractable semantic questions. CVN addresses such questions right from the start. The price paid being that there is some 'overkill' in view of simple examples. Having secured a solution for error handling the development of more expressive formalisms can proceed in a systematic and uninterrupted way on top of CVN (when needed).

Two extensions of CVN. We end this section by introducing two 'natural' extensions of CVN. Let $\Phi : \mathbb{N} \rightarrow \mathbb{T}_4$ be a predicate on $\mathbb{N}$ (taking errors into account). The

[2] Of course, many natural extensions of CVN are conceivable (e.g., with recursive or higher order ingredients). Some proposals are discussed later in the paper.

minimalisation operator μ ("mu") applied to Φ selects the (smallest) natural number n such that $\Phi(n) = T$ and $\forall k < n(\Phi(k) \in \{F, M\})$. If no such n exists, the minimalisation of Φ yields value D. In the latter case, minimalisation characterises either an infinite computation (for all $n \in \mathbb{N}$, $\Phi(n) \in \{F, M\}$), or one that breaks down ($\Phi(n) = D$ for certain n). As an example, let the predicate *odd* represent the odd numbers, then $\mu(odd) = 1$. A particular example is the case that $\Phi = \lambda n.M$, i.e. Φ is the predicate that yields value M for all $n \in \mathbb{N}$: now we find $\mu\Phi = D$. Taking S the *successor function* (i.e., $S(X) = X + 1$), minimalisation of Φ can be formally defined as follows:

$$\mu\Phi = \Phi(0) \cdot\!\!\rightarrow 0 \mathbin{+\!\!+} \mu(\lambda n.\Phi(S(n))).$$

Below we provide an example in which the minimalisation operator μ is used.

As a second extension of CVN, we introduce repeated conjunctions and disjunctions. For predicate $\Phi : \mathbb{N} \to \mathbb{T}_4$, we define:

$$\bigwedge_i^k \Phi = T \qquad \text{if } i > k$$

$$\bigwedge_i^k \Phi = \Phi(i) \mathbin{\&} \bigwedge_{i+1}^k \Phi \qquad \text{if } i \leq k$$

and by dualization:

$$\bigvee_i^k \Phi = F \qquad \text{if } i > k$$

$$\bigvee_i^k \Phi = \Phi(l) \mathbin{\mathrm{v}} \bigvee_{i+1}^k \Phi \qquad \text{if } i \leq k$$

In the following example, both minimalisation and repeated conjunction occur.

Example. In the following we describe a prototypical, generic occurrence of D caused by partiality. Consider the partial subtraction function *psub* as introduced in Exercise 2, and recall our notation $p \cdot\!\!\rightarrow X \mathbin{+\!\!+} \neg p \cdot\!\!\rightarrow Y$ for **if** p **then** X **else** Y. So *psub* is defined by

$$psub(X,Y) = (X \mathbin{\&=} Y) \cdot\!\!\rightarrow 0 \mathbin{+\!\!+} \neg(X \mathbin{\&=} Y) \cdot\!\!\rightarrow S(psub(X, S(Y))).$$

In the case that at least one of X, Y is in $\{D, M\}$, this yields the expected value of $psub(X,Y)$. In the other case, so for $X, Y \in \mathbb{N}$, we give a further analysis of the definition of *psub*. We define $S^0(X) = X$ and $S^{k+1}(X) = S(S^k(X))$, and we further write $S(0) = 1$, $S^2(0) = 2$, etc. Unfolding the recursive definition of *psub* once, we obtain

$$\begin{aligned}
psub(X,Y) = \\
&(X \mathbin{\&=} Y) \cdot\!\!\rightarrow 0 \mathbin{+\!\!+} \\
&\neg(X \mathbin{\&=} Y) \cdot\!\!\rightarrow \\
&\qquad [X \mathbin{\&=} S(Y) \cdot\!\!\rightarrow 1 \mathbin{+\!\!+} \\
&\qquad S^2(psub(X, S^2(Y)))] \\
= \quad &(X \mathbin{\&=} Y) \cdot\!\!\rightarrow 0 \mathbin{+\!\!+} \\
&\neg(X \mathbin{\&=} Y) \cdot\!\!\rightarrow [X \mathbin{\&=} S(Y) \cdot\!\!\rightarrow 1] \mathbin{+\!\!+} \\
&\neg(X \mathbin{\&=} Y) \cdot\!\!\rightarrow \\
&\qquad [\neg(X \mathbin{\&=} S(Y)) \cdot\!\!\rightarrow \\
&\qquad S^2(psub(X, S^2(Y)))] \\
= \quad &(X \mathbin{\&=} Y) \cdot\!\!\rightarrow 0 \mathbin{+\!\!+} \\
&[\neg(X \mathbin{\&=} Y) \mathbin{\&} X \mathbin{\&=} S(Y)] \cdot\!\!\rightarrow 1 \mathbin{+\!\!+} \\
&[\neg(X \mathbin{\&=} Y) \mathbin{\&} \neg(X \mathbin{\&=} S(Y))] \cdot\!\!\rightarrow \\
&\qquad S^2(psub(X, S^2(Y))).
\end{aligned}$$

Let

$$\phi_i^k(X,Y) = \bigwedge_i^k \neg(X \mathbin{\&=} S^{i-1}(Y))$$

and

$$\psi_i^k(X,Y) = \phi_i^k(X,Y) \mathbin{\&} X \mathbin{\&=} S^k(Y).$$

Extending the unfolding above, we can infer for any $k > 1$ that

$$\begin{aligned}
psub(X,Y) \quad = \quad &\psi_1^0(X,Y) \cdot\!\!\rightarrow \\
&0 \mathbin{+\!\!+} \psi_1^1(X,Y) \cdot\!\!\rightarrow \\
&1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \psi_1^k(X,Y) \cdot\!\!\rightarrow \\
&k \mathbin{+\!\!+} \psi_1^{k+1}(X,Y) \cdot\!\!\rightarrow \\
&S^{k+1}(psub(X, S^{k+1}(Y))).
\end{aligned}$$

It is not hard to see that $\psi_1^k(X,Y) = X \dot{=} S^k(Y)$ for X, Y ranging over $\mathbb{N}$, so

$$\begin{aligned} psub(X,Y) \quad = \quad & \\ & (X \dot{=} Y) \cdot \to 0 \mapsto \\ & (X \dot{=} S(Y)) \cdot \to 1 \mapsto \\ & (X \dot{=} S^2(Y)) \cdot \to 2 \mapsto \ldots \end{aligned}$$

Now set $f = \lambda n.X \dot{=} S^n(Y)$, then it follows that

$$psub(X,Y) = \mu f.$$

As an example, $psub(1,0) = \mu f(1,0) = 1$. Furthermore, by the fact that $0 \dot{=} S^{k+1}(0) = F$ for all $k \in \mathbb{N}$, we obtain

$$psub(0,1) = D,$$

which reflects the infinite computation resulting from the recursive definition of *psub*.

Note that μf can be taken as the *definition* of *psub* on $\mathbb{N} \times \mathbb{N} \to \mathbb{N} \cup \{D\}$, and indeed reflects the partiality of $psub(X,Y)$ by a divergent computation when $Y > X$. Furthermore, μf also provides a correct definition for the extension of *psub* to $\mathbb{N} \cup \{D\} \times \mathbb{N} \cup \{D\} \to \mathbb{N} \cup \{D\}$. However, on $\mathbb{N} \cup \{D, M\}$, *psub* and μf define different operations: e.g., $psub(M,0) = M$ and $\mu f(M,0) = D$. This ends our example on the extensions of CVN.

Exercise.

8. Let $f = \mu.k \exists l,m,n((k+1)^{n+3} + (l+1)^{n+3} = (m+1)^{n+3} \mapsto 3)$. Argue—in a fashion similar to the argument given in the example—that $\mu f = D$. (Hint: Consult `http://www.netspace.org/herald/issues/092995/wiles.f.html`.)

5 Example A

In this section we consider the first of two examples, example **A**, of program specifications using CVN. In this and the second example we will use the notion of an *array*, i.e., a collection of indexed data. We also provide the description of a program (or better, a function) for the example and a black box tester both written in the programming language C designed by Dennis Ritchie at Bell Laboratories in about 1972 (see e.g. [15], [8]).

Input:

$max \in \mathbb{N}$,
integer array $a[0], ..., a[max]$.

We view $\{0, 1, 2, ..., max\}$ as V_g, and use notation

$$[0, max]$$

for this set. More precisely, $[i,j]$ represents the empty set if $i, j \in \{0, 1, 2, ..., max\}$ and $i > j$, and $\{i\} \cup [i+1, j]$ otherwise.

Output in CVN:

$$\begin{aligned} & (max < 50) \cdot \to 2 \mapsto \\ & (\forall i \in [25, max]\ (a[49] \geq a[i])) \cdot \to \\ & \qquad (a[1] + a[2]) \mapsto \\ & (\forall i \in [0, max]\ (\forall j \in [0, max] \\ & \qquad (a[i] - a[j] \neq 83))) \cdot \to \\ & \qquad 12 \mapsto 0 \end{aligned}$$

We remark that this functionality in itself is not an interesting one; it might occur in a larger collection of functionalities.

Table 1 exhibits a straightforward translation of the CVN specification into C code. The tester in Table 2 generates in

main an array $a[0], a[1], \ldots, a[max]$ of random integers which — with the exception of $a[49]$ — are all within the interval [-bound,bound]. The length of the array, i.e., the number of integers generated, the upper and lower bound, and — in case max ≥ 50 — $a[49]$ are determined by the user. The produced array is written to the file `arrayA.txt`; the result computed by the function `ExampleA` is send to the standard output stream. Note that, since C integers are not arbitrary large, bound and $a[49]$ have to be in the range provided by the C compiler.

An exhaustive test set with `BlackboxA.c` will now contain a session similar to the one listed in Table 1. The first test yields clearly the expected result: since the generated array consists of 11 elements, the function returns 2. Inspection of the file `arrayA.txt` is in this case superfluous. Also the second test shows the right performance: the array has more than 50 elements and, since $a[49]$ coincides with the upper bound, it is an upper bound to the elements $a[25], \ldots, a[max]$. However, now we have to consult `arrayA.txt` in order to check that $-677 = a[1] + a[2]$. In the last test we generate a sufficiently long array with elements — with the exception of $a[49]$ — within the range $[-10, 10]$. $a[49]$ is assigned the value -11. $a[49]$ is therefore not an upper bound of the elements $a[25], \ldots, a[max]$, and the difference between the various elements does not exceed 21. The expected functionality is thus the return of the value 12, which is indeed the case.

Exercise.

9. The test set depicted in Table 1 is not yet exhaustive, i.e., not all possible cases are taken into consideration.

```
typedef enum {false, true} bool;

int ExampleA ( int , int * );

int ExampleA ( int max , int *a )
{
  int i, j, result;
  bool cond;

  if ( max < 50 )
     result = 2;
  else {
      cond = true;
      i=25;
      while ( cond && i <= max ){
         if ( a[i] <= a[49] )
            i++;
         else
            cond = false;
      }
      if ( cond )
          result = a[1] + a[2];
      else {
          i = 0;
          cond =  true;
          while ( cond && i <= max ){
              j = 0;
              while ( cond && j <= max ){
                  if ( a[i] - a[j] != 83 )
                     j++;
                  else
                     cond = false;
              }
              i++;
          }
          if ( cond )
             result = 12;
          else
             result = 0;
      }
  }
  return result;
}
```

Table 1: `ExampleA.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ExampleA.c"

int main ( void )
{
  int i, b, max, bound, *a;
  FILE *output;

  printf( "Enter max: " );
  scanf( "%d", &max );
  printf( "Enter bound: " );
  scanf( "%d", &bound );
  srand( time(NULL) );
  output = fopen( "arrayA.txt", "w" );
  a = malloc(sizeof(i));
  for ( i=0; i<= max; i++ ){
    if ( i==49 ){
      printf( "Enter a[49]: " );
      scanf( "%d", a );
    } else {
      b = rand() % 2;
      if ( b == 0 ){
        *a = rand() % (bound + 1);
      } else {
        *a = - (rand() % (bound + 1));
      }
    }
    fprintf( output, "a[%d]=%6d\n" ,
       i, *a );
    a++;
  }
  fclose( output );
  a = a - ( max + 1 );
  printf( "Result: %d\n",
    ExampleA ( max, a ) );
  return 0;
}
```

Table 2: `BlackboxA.c`

```
Comment: with \-  -\ we indicate a line
break inserted in machine printout for
text editing purposes.

(bmcp@model 74) gcc -Wall -o \-
    -\BlackboxA BlackboxA.c
(bmcp@model 75) ./BlackboxA
Enter max: 10
Enter bound: 374
Result: 2
(bmcp@model 76) ./BlackboxA
Enter max: 460
Enter bound: 2000
Enter a[49]: 2000
Result: -677
(bmcp@model 77) more arrayA.txt
a[0]=  -927
a[1]=   928
a[2]= -1605
a[3]=   240
a[4]=  1809
.
.
.
.
(bmcp@model 78) ./BlackboxA
Enter max: 50
Enter bound: 10
Enter a[49]: -11
Result: 12
(bmcp@model 79)
```

Figure 1: Typical test session for `ExampleA.c`

State the remaining case and design an appropriate test.

6 Example B

This section deals with the second example, Example B. We proceed along the same lines as in the previous section.

Input:

$max \in \mathbb{N}$,
integer array $a[0], ..., a[max]$,
integer array $b[0], ..., b[max]$.

Again we set $V_g = \{0, 1, 2, ..., max\}$.

Output: We are interested in a program that will read $a[0], b[0]$, then produce $c[0]$, thereafter read $a[1], b[1]$ and produce $c[1]$ and so on till $c[max]$ has been generated. CVN is used to express $c[i]$ in $a[j], b[j]$ for $j \leq i$. The intuition is that $b[i]$ represents an instruction:

$$\begin{aligned} b[i] &= 0 \quad \text{represents insert,} \\ b[i] &= 1 \quad \text{represents delete,} \\ 0 \neq b[i] &\neq 1 \quad \text{represents test,} \end{aligned}$$

and $a[i]$ contains the value to be inserted, deleted or for which membership is tested. Initially the set which the program will maintain is empty. In CVN:

$$\begin{aligned} &((b[i] = 0) \,\mathbin{?\!\vee}\, (b[i] = 1)) \cdot \rightarrow T \twoheadrightarrow \\ &(i = 0) \cdot \rightarrow F \twoheadrightarrow \\ &(\exists j \; 0 \leq j < i \; ((a[i] = a[j]) \,\mathbin{?\!\wedge}\, (b[j] = 0) \,\mathbin{?\!\wedge} \\ &\qquad \forall k \; j < k < i \; ((b[k] = 1) \multimap \\ &\qquad\qquad (a[k] \neq a[i])))) \\ &\qquad\qquad \cdot \rightarrow T \twoheadrightarrow F \end{aligned}$$

Remark: again this functionality in itself is not an interesting one; it might occur in a larger collection of functionalities.

Table 3 contains the C code for Example B. The tester in Table 4 generates now in `main` two arrays $a[0], a[1], \ldots, a[max]$ and $b[0], b[1], \ldots, b[max]$ of random integers which are all within the interval [-bounda,bounda] and [0,boundb], respectively. The length of the arrays, the upper and lower bounds, and the index `i` of an element in the array $c[0], c[1], \ldots, c[max]$ are determined by the user. The produced arrays are written to the file `arrayB.txt`; the value of the element c[`i`] is send to the standard output stream. In Table 5 we listed a few example tests some of which require inspection of the generated arrays.

Exercise.

10. Design a complete series of tests for `ExampleB.c`

```
typedef enum {false, true} bool;
typedef struct {
  int a;
  int b;
} AB;

char ExampleB ( int , AB * );

char ExampleB ( int i , AB *ab )
{
  int j, k;
  bool cond;
  char result;

  if ( ab[i].b == 0 ||
       ab[i].b == 1 ) {
     result = 'T';
  } else {
    if ( i == 0 ){
      result = 'F';
    } else {
      j = 0;
      cond = false;
      while ( !cond && j < i ){
        if ( ab[i].a == ab[j].a &&
             ab[j].b == 0 ){
          k = j+1;
          cond = true;
          while ( cond && k < i ){
            if ( ab[k].b != 1 ||
            ab[k].a != ab[i].a ){
              k++;
            } else cond = false;
          }
        }
        j++;
      }
      if ( cond ) result = 'T';
      else result = 'F';
    }
  }
  return result;
}
```

Table 3: ExampleB.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ExampleB.c"

int main ( void )
{
  int i, n, max, bounda, boundb;
  AB  *ab;
  FILE *output;

  printf( "Enter max:" );
  scanf( "%d", &max );
  printf( "Enter bounda:" );
  scanf( "%d", &bounda );
  printf( "Enter boundb:" );
  scanf( "%d", &boundb );
  srand( time(NULL) );
  output = fopen( "arrayB.txt", "w" );
  ab = malloc(sizeof(AB));
  for ( i = 0; i <= max; i++ ){
    n = rand() % 2;
    if ( n == 0 ){
      (*ab).a = rand() % (bounda + 1);
    } else {
      (*ab).a = - rand() % (bounda + 1);
    }
    (*ab).b = rand() % (boundb + 1);
    fprintf( output,
      "a[%d]=%6d \t b[%d]=%6d\n",
      i, (*ab).a, i, (*ab).b );
      "a[%d]=%6d \t b[%d]=%6d\n", i,
        (*ab).a, i, (*ab).b );
    ab++;
  }
  fclose( output );
  ab = ab - (max + 1);
  printf( "Enter i: " );
  scanf( "%d", &i );
  printf( "c[%d]=%c\n", i, ExampleB(i,ab) );
  return 0;
}
```

Table 4: BlackboxB.c

```
(bmcp@model 14) gcc -Wall -o \-
    -\BlackboxB BlackboxB.c
(bmcp@model 15) ./BlackboxB
Enter max:23
Enter bounda:41
Enter boundb:2
Enter i: 17
c[17]=T
(bmcp@model 16) ./BlackboxB
Enter max:34
Enter bounda:957
Enter boundb:1102
Enter i: 0
c[0]=F
(bmcp@model 17) more arrayB.txt
a[0]=   401      b[0]=   745
a[1]=   946      b[1]=    23
a[2]=   664      b[2]=   387
a[3]=  -892      b[3]=   478
a[4]=  -431      b[4]=    25
.                .
.                .
.                .
(bmcp@model 18) ./BlackboxB
Enter max:181
Enter bounda:749
Enter boundb:1275
Enter i: 1
c[1]=F
(bmcp@model 19) more arrayB.txt
a[0]=  -387   b[0]=   251
a[1]=   -48   b[1]=  1252
a[2]=  -141   b[2]=   480
a[3]=  -129   b[3]=   961
a[4]=  -470   b[4]=   464
.               .
.               .
.               .
(bmcp@model 20)
```

Table 5: Sample tests for ExampleB.c

References

[1] T. Abdel-Hamid and S.E. Madnick. *Software Project Dynamics. An Integrated Approach*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[2] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs* (2nd edition). Springer-Verlag, 1997.

[3] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[4] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Non-Classical Logics*, 5:199–217, 1995.

[5] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[6] J.A. Bergstra and J.V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *Journal of the ACM*, 42(6):1194–1230, 1996.

[7] B. Boehm. *Software Engineering Economics*, Prentice Hall, Upper Saddle River, NJ, 1981.

[8] H.M. Deitel and P.J. Deitel. *C: How to Program* (second edition), Prentice Hall, 1999.

[9] T. DeMarco. *Controlling Software Projects. Management, Measurement and Estimation*, Yourdon Press, Prentice Hall, Englewood Cliffs, NY, 1982.

[10] T. DeMarco and T. Lister. *Peopleware. Productive Projects and Teams*, Dorset House Publishing Co., NY, 1982.

[11] A. van Deursen, J. Heering, and P. Klint (editors). *Language Prototyping: An Algebraic Specification Approach*. AMAST Series in Computing (Volume 5), World Scientific Publishing Co., 1996.

[12] D.P. Freedman and G.M. Weinberg. *Handbook of Walkthroughs, Inspections*

and Technical Reviews (3rd edition). Dorset House, 1990. (Originally published by Little, Brown & Company, 1982.)

[13] R.L. Glass. *Building Quality Software*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[14] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[15] B.W. Kernighan and D.M. Ritchie. *The C Programming Language* (Second edition), Englewood Cliffs, NJ: Prentice Hall, 1988.

[16] C.A. Middelburg. VVSL: a language for structured VDM specifications. *Formal Aspects of Computing*, 1(1):115–135, 1989.

[17] G. Myers. *The Art of Software Testing.* Wiley, 1979.

[18] G.M. Weinberg. *Quality Software Management*, Dorset House Publishing, NY, 1992.

Platforms, Compilers, Interpreters and Portability

Jan Bergstra*

February 17, 2000

Abstract

Platform projection semantics (PPS) extends projection semantics by means of the encapsulation of a processor used for program execution. Platform projection semantics abstracts from platform specific details. For a given program its platform projection semantics may be platform independent. Two forms of platform projection semantics are distinguished: behavioral PPS and functional PPS. Combination of the two yields a formal definition of compilers, interpreters, assemblers and program portability.[1]

Contents

*J.A. Bergstra, University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl.

[1]This paper is a substantially revised and extended version of [?].

1 Introduction

Program algebra as developed in [2] explains the meaning of a program x written in program notation u by means of a projection function `u2pgla` and a behavioral semantics $|-|_{pgla}$ for PGLA programs. Here PGLA is a very simple program notation comparable to flow charts. The principal semantic equation is:

$$|x|_u = |\texttt{u2pgla}(x)|_{pgla}.$$

In this paper we will consider the case that additional information exists about the platform i on which the program is supposed to run. As it may be the case that the projection of the language u is platform specific due to the low level of PGLA the semantic equation can be specialized to platform i as follows:

$$|x|_u^i = |\texttt{u2ipgla}(x)|_{pgla}.$$

Here `u2ipgla` is a platform-dependent projection of u programs to PGLA programs. The platform i being fixed, however, a far more drastic abstraction makes sense. Let CMU_i be the part of the platform i that serves as a working memory for program execution. On the assumption that CMU_i returns to its initial state whenever a new program is run, all interaction between the projected program and the part CMU_i of the processor devoted to runtime data for the (projected) program can be encapsulated. This encapsulation hides differences between processors as long as these are not made visible by program execution. It should be noticed that projected programs will, in the majority of cases, have actions outside the service interface of CMU_i.[2]

1.1 Pure projection assumption

In the analysis of recursion and multi-threading by means of program algebra coprograms must be used in order to capture a datastructure needed to simulate the execution of a high-level program (say in u) by a low-level program (say in PGLA). We will simplify the discussion by assuming that all projections used (like `u2pgla`) are pure in the sense that no additional coprogram is needed. This simplification is quite helpful to reduce the notational overheads and does not affect the point we want to make.

1.2 MMM

MMM (minimal machine model) will be used as a name for the very rudimentary machine model used in this paper. MMM allows a flat file system where all files are named by natural numbers. In addition

[2] Such actions are listed under (i) in paragraph 1.3.

there is a processor memory described by a coprogram. The operating system OSM for MMM only has hardwired instructions. This means that no part of the OS functionality is implemented via software on the available processor. The interaction between processor and file system is entirely done by OSM commands.

MMM is needed in this paper because it allows a clear description of scripts in connection with the installation and porting of programs.

1.3 Modeling CMU_i by a coprogram

CMU_i may be modeled as a coprogram in the sense of [3]. The (pseudo) co-service interface of a machine program contains two kinds of actions: (i) actions in a target interface, aimed at effecting external side effects (like sending an email or storing data on a disk), (ii) actions in the reply co-service interface of the projected program. These actions are to be performed by the part CMU_i of the processor PR_i of platform i.

1.3.1 Coprogram generalities

We recall the concept of a coprogram from [3]. A coprogram consists of a pair $< \Sigma, F >$ with F a mapping from $\Sigma^\star$ to {`true`,`false`}. Σ is called the service interface of the coprogram, F is called its reply function.

States of a coprogram are coprograms as well. By giving a reply $F([a])$ to an action a, the coprogram $< \Sigma, F >$ develops into $< \Sigma, G >$ with $G(w) = F([a] \circ w)$.[3] A notation for G is: $F_{\partial a}$. The collection of all states of coprogram C is denoted with `S`(C).

If no confusion (regarding the service interface) can arrise we will usually neglect the distinction between a coprogramn and its reply function.

1.3.2 The service interface of CMU_i

CMU_i is a processor component admitting a number of instructions. These instructions are machine specific. Each instruction is carried out after a corresponding request made by the central processing unit of platform i (CPU_i). The collection of instructions available for working with CMU_i is called its service interface.[4] We denote the service interface of CMU_i with Σ^u_{cpcmui}.[5]

1.3.3 Data storage and retrieval on CMU_i

We asssume that CMU_i is formalized by means of a coprogram `cpcmui` $=< \Sigma^u_{cpcmui}, F_{cpcmui} >$. The collection of states of `cpcmui` is denoted with `S(cpcmui)`.

This coprogram allows an operator `store`. This operator takes two arguments: a string x (in some cases a program) and a coprogram H which coincides with one of the states of coprogram `cpcmui`. With `store`(x, H) we denote the modification of H obtained by storing data x in it. The storage operator can be applied repeatedly. We will write `store`$(y_1, y_2, ..., y_k,$ `cpcmui`$)$ for `store`$(y_1,$ `store`$(y_2, ..., y_k,$ `cpcmui`$))$.

Besides storing input data files CMU_i can store files resulting from running a program. There is at most one output file at a time. This file can be retrieved by means of the operation `getFile`. If no such file exists, the meaningless constant M is returned.

Futher there is a mapping `getTstatus()` which determines for each state of `cpcmui` its so-called termination status. This is a piece of data indicating whether the state originated from correct termination or from abortion. (If it is not the result of termination no value for `getTstatus()` is specified.) If termination leading to state H was correct `getTstatus`(H) equals 0 otherwise 1. The value of `getTstatus`(H) is set to 0 as a part of execution of ||!|| (the interpretation of correct termination), whereas it is set to 1 as a part of the execution of ||#|| (incorrect termination).

[3] We use [] for the empty list, $[a]$ for the list consisting of a single entry a and $\circ$ for list concatenation.

[4] Instead of service interface, another applicable term is API, for application programmer interface.

[5] The superscript indicates that this interface is an upper interface, which means that all its actions are to be called by other system components, in particular CPU_i. The subscript indicates that the interface is for a coprogram representing CMU_i.

1.4 Behavior versus functionality

When making a further abstraction to incorporate (encapsulate) part of the platform devoted to program execution we will distinguish two cases. In the behavioral view a semantic operator is found that assigns a behavior to a program and some parameter files. The actions of that behavior are exactly the actions outside the service interface of `cpcmui`. In the functional view the meaning of a program is a function from parameter files to output files.

1.4.1 Behavioral platform projection

The behavioral platform projection $|x|_u^{bpp/i}(y_1, ..., y_k)$ of a u program x on platform i given parameter files $(y_1, ..., y_k)$ is given by

$$|x|_u^{bpp/i}(y_1, ..., y_k) =$$

$$|x|_u^i/\texttt{store}(y_1, ..., y_k, \texttt{cpcmui}).$$

The operator / is the use-operator of [3] and a parameter file is just a sequence of bits. With

$$|x|_u^{bpp/i}$$

we denote the operator that takes a (possibly empty) series of files (bit sequences) and produces a behavior depending on these files.

1.4.2 Functional platform projection

The functional platform projection $|x|_u^{fpp/i}(y_1, ..., y_k)$ of a u program x on platform i given parameter files $(y_1, ..., y_k)$ is given by

$$|x|_u^{fpp/i}(y_1, ..., y_k) =$$

$$\texttt{getFile}(|x|_u^i \bullet \texttt{store}(y_1, ..., y_k, \texttt{cpcmui}))$$

provided the computation terminates correctly. In case of incorrect termination we write

$$|x|_u^{fpp/i}(y_1, ..., y_k) = M,$$

whereas in case of divergence we have

$$|x|_u^{fpp/i}(y_1, ..., y_k) = D.$$

The operator $\bullet$ is the apply-operator of [3]. The phrase functional is used in this case because $|x|_u^{fpp/i}$ is viewed as an ordinary mapping (function) from data (bit sequences) to data. With

$$|x|_u^{fpp/i}$$

we denote the operator that takes a (possibly empty) sequence of files (bit sequences) and produces a file (depending on the parameter files). The functional view merits a direct functional notation. We will write:

$$x \bullet \bullet_u^i y_1, ..., y_k = |x|_u^{fpp/i}(y_1, ..., y_k).$$

The operator $- \bullet \bullet_u^i -$ is called the computation operator for language u on platform i. It represents the functional meaning of a program on a platform in a very explicit way.

1.4.3 Incompatible views

It should be noticed that the behavioral view and the functional view are to some extent incompatible. They both constitute drastic abstractions. The two abstractions cannot be obtained from one another. Once one abstraction is made so much information has been dropped that the other abstraction cannot be reconstructed from the remaining abstraction. The two views have each been derived from projection semantics, which is far less abstract as it takes into account the details of the instructions of `cpcmui`.

1.4.4 A combined view

For pragmatic purposes the behavioral view and the functional view can be combined by simply pairing the operators emerging from both views:

$$|x|_u^{cpp/i} =< |x|_u^{bpp/i}, |x|_u^{fpp/i} > .$$

In terms of behaviors and files this means:

$$|x|_u^{cpp/i}(y_1, ..., y_k) =$$

$$< |x|_u^{bpp/i}(y_1, ..., y_k), |x|_u^{fpp/i}(y_1, ..., y_k) > .$$

1.5 Overview of the paper

The two forms of platform projection will be used to analyze a variety of platform-related concepts. In particular we will give precise definitions of compilers and interpreters. On the basis of these definitions a description of program portability is developed. The novelty lies in the use of black box projection semantics for platform analysis. In particular the definition of portability might be new. A minimal introduction to program algebra is provided so as to make the paper self-contained.

1.6 Platform analysis

The kind of concept analysis used in this paper will be referred to as platform analysis. This phrase seems to be novel. The result of platform analysis in this style is a mere listing of concept definitions, mostly formalized in terms of equations between different behavior expressions. Technically speaking platform analysis amounts to no more than a sequence of behavioral and functional identities.

1.6.1 Why platform analysis?

It is useful to have a very explicit statement concerning the usefulness of platform analysis. The following arguments can put forward.

1. Systematic formalization is a prerequisite for precise and correct software production. In its turn, concept analysis at many levels, including platform analysis aiming at the description of the lower levels of systems programming, is a prerequisite for systematic formalization. In this way platform analysis may be a part of the chain leading to the construction of flawless software systems

2. Program algebra provides conceptual definitions of programs and program notations and of the behavioral meaning of programs. Platform analysis extends these conceptual defintions to a family of surrounding concepts including: compilers, assemblers, virtual machines, byte code, assembly language, machine language, portability. Using the approach of platform analyisis a homogeneous and coherent grouping of technical definitions is found, complementary to the definitions based on program algebra.

3. The relation between software and hardware is interesting from a philosophical point of view. Platform analysis is helpful for the clarification of questions regarding this relation.

1.7 Related literature

The contents of the paper are new in form rather than in content. Classical use of T-diagrams [4] will provide similar insights. The novelty of our exposition lies in the use of the projection semantics strategy based on program algebra and, to the best of our knowledge, in the details of the definition of portability that we propose.

Many papers have been written on the abstract concept of a compiler, on virtual machines and the conceptual aspects of intepreters. Closest to our proposals are the elaborated developments using T-diagrams. The reader may consult [1] for an example of technical work on abstact compilers. That paper introduces far more detail than we will do, obtaining sharper results with more limited focus. T-diagrams were introduced in [4] and are treated in detail in [5].

2 Preliminaries on program algebra and notation

Program algebra [2] will only be used in a superficial (or rather abstract) way in this paper. The information needed for reading the paper is summarized here. The program notation PGLA is developed in program algebra as a very simple program notation based on a collection (Σ) of atomic instructions. Programs carry out instructions by offering these as tasks for execution to other components in an environment. If a component executes an action in response to a request made by a program, this component will in addition return a truth value. That value can be used by the PGLA program to decide which control path

to follow (including the option to terminate). An important class of components that may feature in the operating environment of a running PGLA program is formed by the so-called coprograms. Coprograms do no more than compute a reply when offered an action. The coprogram has a state which may be modified by the execution of an action, the intuition being that the coprogram performs (executes) actions following requests of a program. After having performed an action a truth value is returned (as a reply) to the program.

Semantically a coprogram is a pair of an interface (a collection of atomic instructions, also called a service interface) and a reply function. A reply function computes the boolean reply value after each non-empty sequence of atomic actions. An arbitrary program notation PGLY is given a meaning by introducing a transformation (projection) `pgly2pgla` which transforms each PGLY program to a PGLA program with the same meaning. For a PGLA program x the meaning is defined as a behavior written $|x|_{pgla}$. The precise details of the definition of a behavior given a program are entirely immaterial at the present level of abstraction. In [2] an example of a definition of behavior can be found. For PGLY the semantics of programs is then given by

$$|x|_{pgly} = |\texttt{pgly2pgla}(x)|_{pgla}.$$

For a behavior Q and a coprogram H two forms of cooperation have been established in [3]. Both will be used below. Q/H is the behavior of Q using H. The actions of Q contained in the service interface of H are to be processed by H, replies being used to determine the further development of Q. The actions featuring in Q/H are restricted to actions outside $\Sigma_u(H)$. Some actions in the action alphabet (pseudo interface) of Q may not appear as actions of Q/H because the path leading to these actions has been removed. The removal takes place if a boolean returned by H as a reply is used to evaluate a test in Q and produces a particular outcome. We will say that the coprogram H is encapsulated in Q/H.

Another form of cooperation is denoted with $Q \bullet H$. In this case it is assumed that all actions in the (pseudo-) interface of Q are contained in the service interface of the coprogram H. $Q \bullet H$ then denotes the state of the coprogram H which it is left in after termination of the behavior Q. Termination takes place when a particular termination action (denoted with $||!||$) is performed. Because all actions to be taken by Q are in $\Sigma_u(H)$ the entire development of the cooperation between Q and H is determined provided Q is a deterministic behavior. As we will only consider behaviors of the form $|x|_{pgla}$ this requirement of determinism is satisfied.

2.1 Notational conventions

Many notations are discussed below, in addition to the notations used and explained in the introduction. We begin with a survey of these notations, leaving more precise explanations to subsequent sections.

2.1.1 Program notations

With PGLA we denote the program notation introduced under that name in [2]. With PGLU, PGLV, PGLW we denote pairwise different but arbitrary program notations.[6]

With i, j, k different platforms are uniquely named (and hence characterized),[7] FFS_i denoting the flat file system for platform i. CPU_i+CMU_i consists of two parts, CPUi, the central processing unit of platform i, and CMU_i, the central memory unit of platform i. The collection of platforms (or rather platform names) is denoted with PLF. So $i, j, k \in$ PLF.

ASLi is the assembly language for CPU_i+CMU_i, (ASLj for Mj and so on). MCi denotes machine codes for CPU_i+CMU_i, executables for CPU_i+CMU_i being a subset of MCi. The assembler mapping `asli2mci` transforms an assembly program into a a machine code. The mapping `pglu2asli` is a so-called compiler mapping. It determines the effect of a possible (i.e. a correct) compiler from PGLU to ASLi.

Further for the program notation PGLU there is a virtual machine language BCNU (byte code notation

[6] We have in mind that, in comparison with PGLA, these languages play the role of high-level languages.

[7] Platform i is sometimes denoted with PF_i CPU_i+CMU_i being the processor/memory component of platform i.

for PGLU).[8] At this point some unconventional terminology is in order. With PGLUbcl we denote a program notation 'PGLU at byte code level'. PGLUbcl is a human-readable (though bit-sequence encoded) version of BCNU.[9]

2.1.2 Variables

The variables u, v and w will range over arbitrary program notations. These include the MCi and ASLi for processors CPU_i+CMU_i, the languages PGLA, PGLB, PGLC, PGLD from [2], the hypothetical program notations PGLU, PGLV, PGLW mentioned above, and the virtual machine program notation BCNU. Programs are denoted with variables x, y and z. In all cases these programs are bit sequences (or files) at the same time. Thus each program notation u has a domain P.u of programs, which is a set of finite bit sequences.

If a bit sequence is an MCi program it is also called a binary or a binary file.

2.1.3 Projections

The operator $u2v$ determines a canonical projection (or semantics-preserving transformation) from program notation u to notation v. In particular the following instances can be distinguished:

- `mci2pgla` is the black-box processor projection for MCi. It describes how a machine code is to be seen as a program. For a machine code to be seen as a program (according to the thesis[10] of program algebra) a projection into PGLA must be provided.

  In this case the pseudo lower interface of the resulting PGLA program is contained in the union Σ_i of the upper interface of the processor coprogram `cpcmui` and the target interface Σ_{tgi} for machine CPU_i+CMU_i. As a simplification it will be assumed that $\Sigma_{tgi} = \Sigma_{tgj}$ for all processors i and j. The projection `mci2pgla`, together with the coprogram `cpcmui` determine the semantics of MCi programs.

- The operator `asli2mci` is the assembler mapping for ASLi. It transforms ASLi programs into corresponding MCi programs. The additional `mci2pgla` being available, the assembler mapping provides a semantics for ASLi.

  In this paper availability will just mean: being an element of the family of available bit sequences. Typically if more structure is taken into account a file is (defined to be) available if it exists under some name in a file system which has been stored e.g. on hard disk.

- The operator `pglu2asli` is called a compiler mapping. It determines the semantics for PGLU by means of a transformation of each PGLU program into an ASLi program. The compiler mapping is not itself a compiler; rather, it may serve as the semantics of a compiler. Each compiler should produce ASLi programs 'equivalent' to the program given by the compiler mapping, however. There are compiler mappings for PGLV and PGLV as well (named `pglv2mci` and `pglw2mci` respectively.)

- The operator `pglu2bcnu` is the virtual compiler mapping for BCNU. Of course there is a semantic connection (equivalence) between the compiler mapping for PGLU and its virtual compiler mapping.

2.1.4 Projection functionalities

A projection functionality $u\underline{2}v/i$ describes the collection of all mappings that are considered correct representatives of the meaning of some compiler from u

[8] The virtual machine itself can be denoted with VMIU (virtual machine interpreter for PGLU).

[9] BCNU can be considered a virtual machine code (i.e. a machine code for a virtual machine) with PGLUbcl playing the role of a corresponding virtual assembly language. In the popular example of Java, the language Java itself can be represented as a language PGLU, the byte code as a corresponding notation BCNU. A corresponding JavaBcl (playing the role of PGLUbcl), a standardized readable ('disassembled') version of the bytecode, is unfortunately absent. However, several authors have made proposals for a JavaBcl together with a virtual assembler (i.e. a projection from JavaBcl to Java bytecode), and a disassembler for making transformations in the opposite direction.

[10] This thesis is maintained for all program notations, machine codes constituting no exception.

to v, relative to platform i. $u\underline{2}v$ represents the intersection of the $u\underline{2}v/i$ for all i.

With $u\underline{2}v/i.w/j$ we denote all w programs that represent on platform j a file transformation constituting a mapping within $u\underline{2}v/i$. $u\underline{2}v/i.w$ denotes the intersection of the $u\underline{2}v/i.w/j$ for all j in PLF. $u\underline{2}v.w$ denotes the intersection of $u\underline{2}v.w/j$ for all platforms j.

For instance $x \in$ `pglv2aslj.mci`$/i$ expresses the assumption that x is an MCi program that can serve as a compiler from PGLV to ASLj. (A cross-compiler for PGLV and platform j running on platform i.) It is important to notice that $u2v \in u\underline{2}v$.

2.1.5 Interpreters

Let u be some program notation, but not one of the form MCi. $u\underline{1}.v/i$ contains all v programs x that may serve as an interpreter for language u on platfrom i. If v is a platform independent notation $u\underline{1}.v$ denotes the intersection of the program collections $u\underline{1}.v/i$ for all platforms i.

A file $g \in$ `bcnu1.mci`$/i$ is a virtual BCNU machine. `pglu2bcnu` is a compiler mapping for PGLU producing BCNU programs (or rather 'bytecodes').

2.1.6 The main event loop program

The PGLA program *pcpui* encodes the main event loop[11] of processor/memory combination CPU_i+CMU_i.[12] `cpcmui` is a program describing the behavior of CPU_i.

3 The coprogram for CMU_i

The platform i has a processor/memory combination, a file system, a printer, a modem and so on. The memory is denoted with CMU_i. For the purpose of formally modeling platform i, the processor is modeled by at least a coprogram named `cpcmui`. In addition there can be a program (or rather a behavior) *cpu* that determines the CPU of the processor. In section 5 below we will write *pcpui* (for main event loop) instead of *cpu*. When active the machine will execute a stored program. The actions of the program are included in a lower interface in the sence of [3]. The program only has a lower (pseudo) interface. This lower pseudo interface is included in a lower interface Σ, which has a practically low information content. This lower interface Σ is the union of two subinterfaces: the target interface (Σ_{tgi}) along which the program directs peripherals, screen mouse etc. and the reply co-service interface (Σ_{cpcmui}) complementary to the service interface of the coprogram `cpcmui` describing the memory part of machine CPU_i+CMU_i.

The coprogram `cpcmui` describes the memory which is only used by the machine as a working memory during program execution. All data that must persist from one program execution session to another one must be stored in a file. The actions with a modifying effect on the file system will not be spelled out in detail and will be assumed part of the target interface Σ_{tgi}. For the purpose of this paper the details of the processor are immaterial.

3.1 The flat file system

For this paper the existence and the nature of a file system on platform i are crucial and marginal at the same time. Essential is the fact that at any instance of time a collection of bit sequences is present in permanent memory on platform i. We will denote this collection of files with FFS_i (flat file system at i). In the sequel we will provide formalizations of what it means for u program x to be installed on platform i, to be preinstalled on platform i, and to be portable from platform i to platform j. In each case the definition depends on files being present in FFS_i (resp. FFS_j). We will assume that each file has a natural number as its name. At present we will assume that some external agent 'knows' for every bit sequence contained in FFS_i its content as well as its possible roles.

[11] The main event loop is the repetition of events executed by the hardware of a processor. It is the main loop of a system, all other loops are interpreted by actions of this main loop.

[12] `pcpui` stands for: the program for CPU_i.

3.2 The external operation assumption

Some philosophical remarks may be in order. In the simplest approach all programs obtain a meaning by means of a projection to a PGLA program using only actions from Σ. This projection semantics is adequate if one imagines the operating system activities being performed by a human operator. The role of the human operator is needed to avoid script processing and other processing tasks to be performed by the processor. With the introduction of an automatic operating system rudimentary forms of multi-threading (or even parallelism) are introduced making the overall picture significantly more complicated.

The absence of an automatic operating system is phrased as the 'external operation assumption'. This assumption explains in particular the absence of a naming scheme for programs and files, the absence of explicit scripts in the definition of portability below, and the possibility of not having to decide on the proper form of primitive actions needed for file interaction.

3.3 MMM/OSM

For MMM we have a small operating system called OSM. Unfortunately a conceptual definition of an operating system is not available to us. An OS is not necessarily a piece of software, even though most operating systems are. We will use following definitions, not claiming any generality.

(i) An operating system includes a command interface along which at least a collection of commands is accessible to load and and execute programs running on input data and to retrieve output data from a processor. Further, a minimum of primitives for the access of permanent memory is needed (in the case that there is permanent memory on a platform).

(ii) For a given platform there are at least four kinds of OS commands:

- HW/ES: hardwired external service actions: the working of the actions is not explained in terms of the execution of a program (but rather by a state transformation in the machine model e.g. MMM), the actions are called by an external agent. The actions are included in the (external) service interface of the platform.
- PR/ES: programmed external actions. These actions are explained in terms of the execution of a program on the platform. The actions are in fact program names and are viewed as calls to these programs. As in the case of HW/ES the PR/ES actions are called by an external user.
- HW/IS: hardwired internal service actions are explained without the concept of a program execution. In contrast with HW/ES these actions are called by programs running on the platform.
- PR/IS actions are macros for programs to be called by other programs.

In OSM there are only HW/ES actions. This implies the absence of any possibility of scripting for the OS. The existence of a scripting language presuposses the existense of some actions in HW/IS or in PR/IS.

3.3.1 MMM/OSM commands

The command set of OSM allows interaction between the processor CPUi and the file system FFS_i. The commands are:

(i) `runB` k: run file k and obtain a behavior. Here file k is supposed to contain an MCi program x which is run on processor CPUi (which may be storing some data that have just been loaded) in order to produce a behavior. After termination the processor is reset to its initial state.

(ii) `runF` k: run file k and apply its functionality. Here file k is supposed to contain an MCi program x which is run on processor CPUi (which may be storing some data that have just been loaded) in order to produce a functional output. We assume that the MCi program x (or, more exactly, `mci2pgla`(x)) is a program over co-service interface Σ^l_{cpcmui}, which contains the same actions as Σ^l_{cpcmui}, but then in the role of lower interface instructions. Stated differently: x is only allowed to call actions from the service interface of CMU_i. Whenever another instruction is called from x an incorrect termination occurs. (After termination the processor will only allow an action

`getFile` k or an action `reset`, all other actions leading to incorrect termination.)

(iii) `store` k: store file k. The processor state is transformed. In terms of coprograms this transformation amounts to: $z \rightarrow \texttt{store}(x, z)$, x being the contents of the file named k when the action is called. FFS_i remains unchanged.

(iv) `getFile2` k: get an output file (from CPU_i+CMU_i) and store it in file k in FFS_i. The effect of this action is (a) to introduce a file named k if that is not yet in existence, (b) to initialize that file with an empty sequence of bits, (c) to replace the contents of file k by the bit sequence `getFile`(z) (z denoting the processor coprogram state), (d) to initialize z (which is thus replaced by `cpcmui`.)

(v) `getTstatus2`k: take the termination status of the state of `cpcmui` and stores it in file k. If this staus equals 1 (wrong termination) the processor coprogram is reset to its original state immediately.

(vi) `reset`: reset the processor to its initial state. The new state of the processor coprogram is `cpcmui`.

(vii) `copy` k l: copy file k to file l.

(viii) `remove` k: remove file k from FFS.

(ix) `existenceTest` k: test whether or not a file named k exists in FFS_i. If so return `true`, if not return `false`.

3.3.2 Role of MMM/OSM

The role of MMM/OSM in this paper is to have the means available to use the mechanism of MMM in order to instal programs, to apply assemblers and compilers, to use interpreters. At the same time these means will not pressuppose the very concept of software (in particular OS systems programs) running on the platform. In this way a circularity is prevented that seems to become quite difficult to handle otherwise. Of course the OSM command set can be used as a basis for writing 'SHELL scripts'. Such scripts are to be executed by an operator rather than by the OSM, the OSM not offering any script interpreter. The restriction to operator execution of OSM scripts was termed the external operation assumption above. The assumption is meant to simplify the task of conceptual modeling.

Below the concept of portability is discussed in some detail. When a program x is ported from platform i to platform j some files will have to be transported from i to j. Transporting a file is not seen as an operating system primitive here. We call it an operator primitive. In fact three operator primitives are needed to model the transport of files:

(i) `copyFFSi2EM` k, l: copy file k (of FFS_i) to file l on the external medium. (This is a unique medium determined by its place in the physical world, e.g. the unique floppy in the unique floppy disk station.)

(ii) `transportEM` i j: transport the external medium from platform i to platform j.

(iii) `copyEM2FFSi` k,: copy the file k of the external medium to file l of the FFS_i of platform i.

4 Projection semantics

Before we can apply platform projections, be it behavioral or functional, we need a platform-independent projection semantics. In the case of machine codes this matter deserves ample attention. During execution a program is stored in the RAM of the machine. The stored program results from loading an executable file into the machine. The executable file is a binary: just a sequence of bits. Before loading this file sits in the file system under some name (say n). This is the form in which programs are installed on the machine: named executables as files in the file system. We assume that the loading procedure will not influence the algorithmic content of the binary being loaded, even if the loading process itself is quite sensitive to state aspects. This sensitivity regards form but not content. The installed binary has a meaning which is reflected by the meaning of the corresponding stored program. The latter is in fact the meaning of a program with basic instructions taken from Σ_i (see paragraph 2.1.3).

The semantics of an installed binary is a program over Σ. Therefore it makes sense to model the meaning of binaries by means of a projection `mci2pgla`. Here `mci` stands for machine code for machine CPU_i+CMU_i, and `pgla` refers to $PGLA_\Sigma$. This projection is completely dependent on the particular details of machine CPU_i+CMU_i. These details being

left unspecified, `mci2pgla` is a black box mechanism without further explanation. In fact `mci2pgla` can be used to specify CMU_i. Simply postulating the existence of `mci2pgla` leads us to a black-box (projection) semantics for binaries. In the presence of full details a derivation of `mci2pgla` from those details will provide a white box semantics of binaries. Black box semantics will support conceptual discussions at a high level of abstraction. White box semantics is essential if the algorithmic content of individual binaries is at stake.

4.1 Projection semantics of machine code

We assume that a behavior extraction operator

$$|-|_{pgla}$$

on PGLA programs is given.[13] As is the case in all other program notations the meaning of a program is ultimately based on a projection into PGLA. That projection may involve platform-specific decisions and ingredients. For this reason the behavior extraction operator for language u is (potentially) specific to platform i. The notation

$$|x|_u^i$$

will represent the (intended) behavior of u program x on platform i. This intended behavior is always defined irrespective of the ability to actually realize it on a given platform. In the case that u = PGLA we write $|-|_{pgla}^i = |-|_{pgla}$.

4.1.1 The meta disassembler equation

The semantics of a binary x is then determined by the identity

$$\text{PLA1} \quad |x|_{mci}^i = |\texttt{mci2pgla}(x)|_{pgla}.$$

A binary x is an executable if the first instruction of $\texttt{mci2pgla}(x)$ is not #. We call this equation the meta disassembler equation because `mci2pgla` is an operation comparable to a disassembler. First of all it moves the code to a higher level of abstraction and readability. Secondly PGLA has an expressive power comparable to many assembly codes in practice. Calling `mci2pgla` a disassembler, however is misleading because the inverse of the assembler mapping `asli2mci` deserves that name. Because `mci2pgla` will play a role in the meta theory we call it a meta disassembler.

[13] An example of a behavioral extraction operator is found in [2].

4.2 Projection semantics for non machine codes

For an arbitrary notation u the semantic projection uses a compiler mapping. This mapping produces (relative to the platform) an adequate translation of the program into another program notation. Compiler mappings are assumed to exist for all platforms, regardless of the actual presence of compilers.

$$\text{PLA2} \quad |x|_u^i = |u2\texttt{mci}(x)|_{mci}^i$$

4.2.1 Assembly programs

An assembly language is a textual format for programs very close to the binary format. Assembly languages are associated with processors rather than with program notations. The representation of assembly programs in binary files proceeds by means of a character encoding (e.g. ASCII). Such representations are not sufficiently concise and efficient to be used for machine loading. Further translation into machine code will be necessary for efficient program execution. We will only consider a single assembly language for processor CPU_i. It is denoted with ASLi. The assembler is a program which will translate assembly programs into machine codes. Of course the assembler is specific to a machine, because the machine architecture dictates the encodings that will be used. Rather than using an assembler program we will consider assembler mappings. An assembler mapping is a mathematical operator. Typically the assembler program will compute an assembler mapping. The mapping may be considered a functional specification of an assembler program.

The assembler mapping `asli2mci` is a mapping transforming assembly programs into machine codes. The assembler mapping can be chosen to be a total mapping on binary files. In that case a file contains a (syntactically) correct ASLi program x exactly if `asli2mci`(x) is an executable (i.e. the first action of `mci2pgla(asli2mci`(x)) differs from #).

4.2.2 The assembler mapping equation

It will be clear that the semantics of the assembly programs is given in terms of a projection `asli2pgla`(x). This projection is the composition of `asli2mci` and `mci2pgla`. The semantics of assembly programs is thus determined by the following assembler mapping equation:

$$\text{PLA4} \quad |x|^i_{asli} = |\texttt{asli2mci}(x)|^i_{mci}$$

4.2.3 The compiler mapping equation

The concept of a program notation will be defined slightly more precise here than in [2]. The increased precision has to do with the presence of the intermediate assembly language. (That intermediate stage was ignored in [2].) A program notation PGLV is a collection of objects (represented by means of a character encoding as binary files) together with a projection function `pglv2asli` to ASLi. The operator `pglv2asli` can be taken as a total mapping. This mapping is called a compiler mapping for PGLV. A text x constitutes a legal (i.e. a syntactically correct) PGLV program exacly if the first action of `pglv2asli`(x) differs from #. The corresponding PLA equation reads:

$$\text{PLA5} \quad |x|^i_{pglv} = |\texttt{pglv2asli}(x)|^i_{asli}$$

Given the collection PLF of platforms (i.e. machines with installed software) a definition of platform independence (relative to PLF) of a program notation can be obtained.

4.3 Platform independence

At this point it is possible to provide a useful definition of platform independence for a program notation. The program notation PGLV is said to be behaviorally platform independent precisely if, for all PGLV programs x,

$$\forall i, j \in \text{PLF}\ (|x|^{bpp/i}_u = |x|^{bpp/j}_u).$$

The program notation PGLV is said to be functionally platform independent precisely if, for all PGLV programs x,

$$\forall i, j \in \text{PLF}\ (|x|^{fpp/i}_u = |x|^{fpp/j}_u).$$

The program notation PGLV is said to be platform independent if it is behaviorally platform independent and functionally platform independent at the same time.

5 The main event loop equation

The PGLA program *pcpui* will be used to denote the main event loop of the processor M_i. This main event loop can be described as a program. In practice it is rigidly encoded in the structure of the hardware.[14] The fact that machine code is interpreted by a main event loop is given by the main event loop equation PLA3, serving as an alternative definition of processor encapsulation at the same time:

$$\text{PLA3b} \quad |x|^{bpp/i}_{mci}(y_1, .., y_n) = |pcpui|^{bpp/i}_{pgla}(x, y_1, .., y_n)$$

and

$$\text{PLA3f} \quad |x|^{fpp/i}_{mci}(y_1, .., y_n) = |pcpui|^{fpp/i}_{pgla}(x, y_1, .., y_n).$$

PLA3 describes the realization of a machine code execution, whereas PLA1 describes an abstract semantics for it.

[14] Interestingly some hardware is reasonably classified as programs because it embodies a program controlling other hardware. The mere fact that the program has been silicon compiled need not turn the classification as a program into an unreasonable one. It follows that computer software is a proper subcollection of the class of 'programs'. This holds true in particular in the restricted context of artificial computing.

5.1 The disassembler soundness equations

Evidently for each platform the following constraints must be satisfied:

$$|pcpui|_{pgla}/\texttt{store}(x, y_1, .., y_n, \texttt{cpcmui}) =$$

$$|\texttt{mci2pgla}(x)|_{pgla}/\texttt{store}(y_1, .., y_n, \texttt{cpcmui})$$

and

$$\texttt{getFile}(|pcpui|_{pgla} \bullet \texttt{store}(x, y_1, .., y_n, \texttt{cpcmui})) =$$

$$\texttt{getFile}(|\texttt{mci2pgla}(x)|_{pgla} \bullet \texttt{store}(y_1, .., y_n, \texttt{cpcmui}))$$

These identities are called the disassembler soundness equations, in virtue of their validating the meta disassembler mci2pgla with respect to the reality of the technical implementation.

5.2 PGLD over Σ_i in the role of ASLi

If a technical realization of the concepts involved is to be designed the following viewpoint may be helpful. An assembly language such as ASLi will have a clear algorithmic content. Moreover ASLi is meant to write programs executing the basic instruction set Σ_i of platform i. This is certainly obtained by taking ASLi = PGLD_{Σ_i}. Here PGLD is the PGLA based language from [2] using forward and backward absolute jumps rather than relative forward jumps.

Let Σ_i have cardinality n. Then the actions in Σ_i can be encoded with bit sequences of length $l_1 = {}^2\log n$. In practice a finite upper length k may be imposed on programs as well. Let $l_2 = {}^2\log k$. Taking into account that PGLD uses 5 kinds of instructions (basic actions, positive tests, negative tests, jumps, termination instructions and crash instructions) no more than $l_0 = k(3 + \max(l_1, l_2))$ bits are needed to encode an ASLi program. Such bit sequences, encoding listings of (encoded) instructions from PGLD_{Σ_i} programs, constitute the corresponding machine code language MCi. The disassembler operator is definable as the inverse of the encoding from PGLD_{Σ_i} followed by the projection from PGLD to PGLA.

The main event loop program starts with the assumption that an encoding of a program is stored in memory. Then it will analyze the given bit patterns in the encoded program and evaluate which action from Σ_i is meant (or otherwise which jump or termination action). The main event loop will perform the action so determined and proceed until an encoded termination is detected or until the last instruction of the encoded program turns out to have been completed.

5.3 The interpreter equations

The main event loop interpretes machine codes. If programs different from a machine code must be interpreted this will require the execution of a special program called an interpreter.

The working of an interpreter $x \in u\underline{1}.v/i$ for program notation u written in v and effective on platform i is given by the following equation (for all u programs y). x is an interpreter of the mentioned kind if for all u programs y and for all sequences of files $(y_1, .., y_n)$ the following two conditions hold:

PLA6b $\quad |y|_u^{bpp/i}(y_1, .., y_n) = |x|_v^{bpp/i}(y, y_1, .., y_n)$

and

PLA6f $\quad |y|_u^{fpp/i}(y_1, .., y_n) = |x|_v^{fpp/i}(y, y_1, .., y_n).$

6 File transformer functionalities

A file transformer functionality is a collection of pairs of lists of bit sequences and individual bit sequences. The pairs are input-output pairs compatible with the functionality. A deterministic functionality contains for a given input list $y_1, .., y_n$ at most one output bit string z. Typically the meaning of an executable x viewed as a tuple transformer is a deterministic functionality given by the collection of all pairs $< (y_1, ..., y_n), z >$ such that $x \bullet \bullet^i_{mci} y_1, ..., y_n = z$.

6.1 Realization of a functionality

Given a functionality F and a program notation u (viewed as a subset of the bit strings) $F.u/i$ denotes

all programs in u which implement or realize the functionality on platform i.

We say that executable x implements functionality F (on i) if for all input bit string sequences $Y = (y_1, ..., y_n)$ the following two conditions hold:

(i) if there exists z with $< Y, z >$ in F then $x \bullet \bullet^i_{mci} Y = z'$ for some bit sequence z'.

(ii) Moreover, if there exists z with $< Y, z >$ in F and for some bit sequence z' the equation $x \bullet\bullet^i_{mci} Y = z'$ is satisfied, the pair $< Y, z' >$ must be contained in F.

A functionality can also be realized by an assembly program or a high-level program (in PGLV). In that case the definition uses operators $\bullet\bullet^i_{asli}$ and $\bullet\bullet^i_{pglv}$ respectively.

6.2 Assembler functionality and compiler functionality

The assembler functionality asli2mci/i contains all pairs $< z, u >$ where z is a correct ASLi program and

$$|z|^{cpp/i}_{asli} = |u|^{cpp/i}_{mci}.$$

Here the combined platform projection (see paragraph 1.4.4) is used to take the behavioral and functional view into account simultaneously .

The compiler functionality pglv2asli/i contains all pairs $< z, u >$ such that y is a correct PGLV program and

$$|z|^{cpp/i}_{pglv} = |u|^{cpp/i}_{asli}.$$

6.2.1 Assemblers and Compilers

An assembler is an element f of the assembler functionality asli2mci/$i.v/i$. The characteristic identity is:

$$|x|^{cpp/i}_{asli} = |f \bullet \bullet^i_v x|^{cpp/i}_{mci}.$$

On most platforms it will be useful to have assemblers and compilers available in machine code, in order to be ready for use. If only sources are available, it will be necessary to compile or interpret these sources first in order to achieve the required compilation or assemblage.

A compiler for u on platform i written in notation v is just a realization of the functionality u2asli/$i.v/i$. It follows that for $f \in$ u2asli/$i.v/i$ for all u programs x:

$$|x|^{cpp/i}_{u} = |f \bullet \bullet^i_v x|^{cpp/i}_{asli}.$$

6.2.2 Cross-assemblers and cross-compilers

A program in asli2mci/i.v/j is a cross-assembler for CPU$_i$ on CPU$_j$ written in notation v. A program in u2asli/i.v/j is a cross-compiler for u and and i on platform j written in notation v.

6.3 A compiler fixed point

In some case the following observation is used as a practical criterion for compiler correctness (or at least plausibility). Suppose that u program $x \in u$2mci.u. Then

$$(x \bullet \bullet^i_u x) \bullet \bullet^i_{mci} x = x \bullet \bullet^i_u x.$$

The fact simply follows from the assumption that $x \in u$2mci.u/i. The use takes place by evaluating both files and checking identity bitwise.

7 Program installation

A u program x is said to be installed on platform i if FFS$_i$ contains an executable y and files $y_1, ..., y_n$ such that for all bit sequence lists $z_1, ..., z_n$:

$$|x|^{cpp/i}_{u}(z_1, ..., z_n) = |y|^{cpp/i}_{mci}(y_1, ..., y_n, z_1, ..., z_n).$$

7.1 Software expansion

Expansion of the software in a platform takes place if new files z in the platform are made by means of program applications of the form

$$z = x \bullet \bullet^i_{mci} y_1, ..., y_n.$$

In this case it is assumed that FFS$_i$ already contains $x, y_1, ..., y_n$ and that z is now added. (In practice software expansion is often effected by means of so-called installation programs. The UNIX 'make' utility also generates a software expansion.)

7.1.1 Preinstallation

A program is preinstalled on a platform if the software contained in the platform can be expanded to a state in which the program is installed.

8 Program Portability

Portability is an interesting subject. Its relevance transpires from the impact that portability considerations have on the design of program notations and programming environments and methods. The purpose of this section is to provide a precise definition of portability. The proposed definition is only an approximation to the concept of portability but it covers some common cases. We take portabiliy to mean: cross-platform portability. Transportability is a property of all files. Indeed each bit sequence can be transported from any platform i to any platform j, e.g. by electronic mail or by means of the physical transport of a data storage device such as a book, a disk or a tape.

8.1 Platforms with installed software

A platform is characterized not only by its processor/memory unit (represented by the coprogram `cpcmui`) but also by the the software that has been installed. Without the notion of installed software the concept of cross-platform portability of software seems to make little sense. In this section a platform with installed software is simply characterized by a processor coprogram together with a collection of programs (= bit sequences) together with a type. The only types used for the definition of portability are the projection types (i.e compiler types), and the interpreter types. Questions as to how to obtain or call a program on the platform and issues of program naming are ignored at this level of abstraction.

Denoting the collection of typed programs with V, the platform with installed software base V centered around processor i can be written as $\mathrm{ISBP}_i(\mathrm{V})$.

8.2 Portability formalized

Portability of u program x from platform i to platform j requires the combined view identity $|x|_u^{cpp/i} = |x|_u^{cpp/j}$. If a program has no platform independent meaning it cannot possibly be portable (platform independence referring merely to the two platforms in question.)

The portability of software will informally amount to this: u program x is portable from $\mathrm{ISBP}_i(\mathrm{V})$ to $\mathrm{ISBP}_j(\mathrm{W})$ if the following steps are possible:

1. Generate a number of files $y_1, ..., y_n$ on platform $\mathrm{ISBP}_i(\mathrm{V})$ thus obtaining $\mathrm{ISBP}_i(\mathrm{V+Y})$, with Y representing the generated files together with their types. (This step involves an expansion of the software in platform i.)

2. Transport Y to $\mathrm{ISBP}_j(\mathrm{W})$ thereby obtaining $\mathrm{ISBP}_j(\mathrm{W+Y})$, and thereafter remove Y from $\mathrm{ISBP}_i(\mathrm{V+Y})$. (Achieving a state in which x is preinstalled on j.)

3. Generate some files $z_1, ..., z_m$ from the software present in V+Y in
$\mathrm{ISBP}_j(\mathrm{W+Y})$ thus obtaining $\mathrm{ISBP}_j(\mathrm{W+Y+Z})$
(Expansion of the software in platform j until the stage that x is installed on platform j.)

It should be noticed that x need not be preinstalled in $\mathrm{ISBP}_i(\mathrm{V})$. Further it should be noticed that $|x|_u^i$ is defined always, irrespective of the availability of compilers and interpreters needed to find a program execution expression over $|x|_u^i$.

8.3 Knowledge management

An obvious difficulty of the portability definition presented above is that it still gives little clue as to how it works in practice. If an FFS_i contains a number of files, how can one assert that some of those are compilers of a useful kind whereas others are not. The definitions can be applied only in theory. Quantification over all possible lists of argument files is prohibitive. At the level of abstraction of this paper we take for granted the existence of an external authority (the user) who knows the functionality of each file. This makes it quite hard to prove negative facts about

portability. Can it be the case that 'by accident' a file obtained after some calculation steps is a useful compiler? If not the user is equipped with some sort of default logic. If so, s/he may embark on experimentation and use a logic of plausibility if a reasonable number of tests establish that a file implements some functionality correctly. From a logical point of view one may prefer to have a clear type system containing all sure and positive information about the files in possesion of FFS_i. Only by means of the type system can it be concluded that two programs behave equivalently. In practice most users have a type system in mind which is reflected by the names of the files. They use a typed logic until unexpected results occur. In that case a shift to an experimental mode with plausibility reasoning may take place. If that brings no useful results a pessimistic default logic may be used allowing only information that has been proven somehow and denying every unprovable hypothesis rightaway.

8.4 The example of PGLU interpretation

In the notational scheme of paragraph 2.1 PGLU is a program notation to be compiled to a 'bytecode' BCNU (byte code notation from PGLU). The following assumptions are made: *cu* compiles from PGLU to BCNU. It is written in PGLV. For PGLV a compiler *cv* is given. Finally *iu* is a BCNU interpreter also written in PGLV. It is assumed that the compiler *cu* includes the assembler. So *cu* takes a PGLV program all the way down to MCi.

Having these files available we can discuss the processing for PGLU programs and their portability by means of transportation of bytecodes (BCNU programs). Assume that $|x|^{cpp/i}_{pglu} = |x|^{cpp/j}_{pglu}$.

8.4.1 Running PGLU programs

A program execution expression for $|x|^{i/pce}_{pglu}$ is derived as follows (with $Y = y_1, ..., y_k$):

$|x|^{cpp/i}_{pglu}(Y) =$
$|\texttt{pglu2bcnu}(x)|^{cpp/i}_{bcnu}(Y) =$
$|cu \bullet \bullet_{pglv} x|^{cpp/i}_{bcnu}(Y) =$
$|(cv \bullet \bullet^i_{mci} cu) \bullet \bullet^i_{mci} x|^{cpp/i}_{bcnu}(Y) =$
$|iu|^{cpp/i}_{pglv}((cv \bullet \bullet^i_{mci} cu) \bullet \bullet^i_{mci} x, Y) =$
$|cv \bullet \bullet^i_{mci} iu|^{cpp/i}_{mci}((cv \bullet \bullet^i_{mci} cu) \bullet \bullet^i_{mci} x, Y)$

8.4.2 An MMM/OSM command script for running PGLU programs

Using the OSM commands this pattern can be turned into a script of shell MMM shell commands which result in the expression of the behavior of $|x|^{cpp/i}_{pglu}(Y)$.

We write $< i : n >$ for the contents of file n in FFS_i. We assume that $x =< i : 5 >$, $Y = y_1, y_2, y_3$, and $y_1 =< i : 7 >, y_2 =< i : 8 >, y_3 =< i : 9 >$. Further we need a compiler for PGLU (cu $=< i : 10 >$), an interpreter for PGLU (iu $=< i : 11 >$) and a compiler for PGLV (cv $=< i : 12 >$).

The MMM/OSM command sequence (in fact a PGLA program over the OSM command set, ! representing termination in PGLA) corresponding to the above derivation is as follows:

```
store 10; runF 12; getFile2 1;
store 5; runF 1; getFile2 2;
store 11; runF 12; getFile2 3;
reset; store 9; store 8;
store 7; store 2; runB 3;!.
```

8.4.3 Transporting the bytecode

We assume that we actually work on platform P27 (availing of FFS27 etc). The bytecode corresponding to the program x in the above example is the file

$$(cv \bullet \bullet^i_{mci} cu) \bullet \bullet^i_{mci} x$$

This file is placed in file 2 of FFS_i. At any position in the script after the instruction `getFile2 2` has been carried out an action `copyFFSi2EM 2 9` will place the byte code on file 9 of a external medium. (e.g. a writeable CD) with identity 43. The medium is uniquely determined by its position in the disk writer at the time of writing. Then an action `transportEM43 27 458` takes this EM (which after removal from the writer needs to be identified

by its name), transporting it to site 458. A subsequent action `copyEM2FFS458 9 5` willl copy it to file 5 of FFS458. There it can be a starting-point for software installation.

8.4.4 Heterogeneous scripts

We then assume that the files $Y = y_1, y_2, y_3$ are present in the file 7,8 and 9 of FFS458 (rather than in files of FFS27). It is possible to find a program execution expression for $|x|_{pglu}^{cpp/27}$ on P458. The following heterogeneous script serves this purpose. A heterogeneous script has all non-transport atomic instructions tagged by the location (here identified with the platform identity) as follows:

```
store 10/27; runF 12/27; getFile2 1/27;
store 5/27; runF 1/27; getFile2 2/27;
copy FFSi2EM 2 9/27;!/27;
transport EM43 27 458;
copyEM2FFS458 9 5/458;
store 11/458; runF 12/458; getFile2 3/458;
reset/458; store 9/458; store 8/458;
store 7/458; store 5/458; runB
3/458;!/458;!.
```

This heterogeneous script allows one to port x from P27 to P458 and to apply x on P458 with three files from FFS458 as arguments.

8.4.5 Data carrying programs

The notation used above is slightly redundant because of the repetition of the argument list Y. This redundancy can be removed by means of 'Currying'. This step allows one to have more concise notation at a higher type (i.e. not mentioning specific arguments or argument variables). For that purpose it is useful to allow so-called data carrying programs. A data-carrying program has the form $x[Y]$ with x a bit sequence and $Y = y_1, \ldots, y_k$ a list of bit sequences. Y contains the data. If $k = 0$ the program $x[]$ results which for most purposes can be identified with x. If x is a u program then $x[Y]$ is a (data-carrying) u program. Two rules of calculation are needed:

$$|x[Y]|_u^{cpp/i}(Z) = |x|_{pglu}^{cpp/i}(Y, Z)$$

and

$$x \bullet\!\bullet_u^i (y[Z]) = (x \bullet\!\bullet_{mci}^i y)[Z].$$

The main fact about data-carrying programs is in their relationship with interpreters: if $y \in u\underline{1}.v/i$, then

$$|x|_u^{cpp/i} = |y[x]|_v^{cpp/i}.$$

These notations allow a simplification of the notation used for PGLU above:

$$|x|_{pglu}^{cpp/i} = |cv \bullet\!\bullet_{mci}^i iu[(cv \bullet\!\bullet_{mci}^i cu) \bullet\!\bullet_{mci}^i x]|_{mci}^{cpp/i}.$$

8.4.6 Universal program notations

The concept of a universal program notation is an idealized one, unfortunately unavailable in practice. Consider program notation PGLV. PGLV will be called universal if the following two conditions are satisfied: (i) for all platforms i and j and for all PGLV programs x $|x|_{pglv}^{i/pre} = |x|_{pglv}^{j/pre}$, (ii) at each platform (i) a PGLV compiler (i.e. an element of $pglv\underline{2}asli.mci.i$) is available as well as an assembler program (i.e. an element of `asli2mci/i.mci/`i).

It follows from this definition that all PGLV programs are portable from any platform to any other platform. Moreover the porting can be done in a uniform way.

8.4.7 Bytecode portability

Assuming that the PGLU program x is platform independent (when comparing platforms i and j), x can be ported to j whenever the following ingredients are available at j: cv and iu. cv can be assumed available if PGLV is a universal program notation. iu is entirely platform independent. This file can be distributed physically or electronically from a central site. In order to port x from platform i to platform j, the file $cu \bullet\!\bullet_{pglv}^i x$ is generated at platform i and subsequently transported to platform j.

8.4.8 Performance aspects

There may be several reasons to organize the execution architecture of a program notation PGLU along the lines mentioned above. First of all the existence of efficient PGLV compilers on many platforms can be

exploited, thus removing the need to write equally efficient PGLU compilers for all platforms. The portability of PGLV programs induces portability of PGLU programs. Secondly, if many instances of a PGLU program x must be executed at many different sites it is more efficient to do PGLU-to-BCNU compilation only once. Thirdly, if on a certain platform no PGLU programs are developed there is no need to keep a PGLU to BCNU compiler available (with its unavoidable version management burden). Therefore such compilers are only needed at platforms hosting PGLU production processes. All of these arguments hold in case we take Java for PGLU, Java Bytecode for BCNU and C for PGLV.

9 Conclusions and acknowledgements

Platform projection semantics has been introduced as a combination of behavioral platform projection semantics (bpp) and functional platform projection semantics (fpp).

A definition of portability has been developed on the basis of platform projection semantics of program notations. In addition the concepts of a compiler and an interpreter have been formalized using platform projection semantics. These developments are satisfactory given the high level of abstraction at hand. Further details can be added when needed.

Marijke Loots has made many useful comments on previous versions of this text. Sietse van der Meulen is acknowledged for pointing out to the author the paramount importance of portability.[15]

10 References

[1] A.W. Appel. Axiomatic bootstrapping: a guide for compiler hackers. *ACM TOPLAS*, 16(6):1699–1719, 1994.

[2] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998. To appear in Formal Aspects of Computing.

[3] J.A. Bergstra and M.E. Loots. Abstract data type coprograms. Technical report, Department of Philosophy, Utrecht University, 1999. In: Bergstra/Loots, Programs, Interfaces and Components, Artificial Intelligence Preprint Series 006.

[4] H.Bratman. An alternate form of the UNCOL diagram. *CACM*, 4(3):142, 1961.

[5] J.Earley and H.Sturgis. A formalism for translator interactions. *CACM*, 13(10):607–617, 1994.

A Exercises

1. Give two examples[16] of programs (and their purpose or role) for which an analysis using functional platform projections may well be adequate.

2. Give two examples of programs (and their purpose or role) for which an analysis using functional platform projections is probably inadequate. Explain in these cases why behavioral platform projections may be better suited to formalize the semantics of the programs.

3. For the following program notations there is a preferred way of implementation with respect to the choice between compilation into assembly language, compilation into a byte code (and subsequent interpretation of that byte code), and interpretation. What is this most common choice? Consider: C, C^{++}, FORTRAN, LISP, COBOL85, PERL, Java, PASCAL, Ada, Tcl/Tk, PROLOG.

4. Write a very simple program x in C. Find the compiler option needed to compile it in to an

[15] This dates back to 1979. Since then many developments have proven that the pursuit of portability of programs and data is a major driving factor for the development of software technology.

[16] The examples should be outside the context of compilers and interpreters.

assembly program y (e.g. using `gcc`). Try to read this assembly program and to see if you can distinguish instructions from the co-service interface that y assumes (i.e. uses). Use the assembler of your OS (`as` in case of Solaris) to assemble y. 'See that it works'.

5. Make a design of a set of instructions useful as a service interface for an FFS. (Assume that instructions can return only a boolean value in spite of the fact that practice is more advanced regarding the transport of data between hard disk and processor memory. JavaScript can be taken as a source of inspiration for people lacking a clue. The remarkably complicated (and ugly?) Java file IO can be taken as a point of departure for the more ambitious designer.)

6. In this exercise the option to use a Turing tape as the CMU_i is discussed.

   (i) Give a description of the Turing machine tape as a coprogram.[17]

   (ii) Add definitions for `store()`, `getFile()` and `getTstatus()`.

   (iii)What are the implications for the realization of ||!|| and ||#|| if this coprogram is used as a `cpcmui`.

   (iv) Is it possible to write programs for OSM instructions such as `store` k, and `getFile2` k (assuming that FFS is accessible via the service interface that was designed in the previous exercise)?

7. Describe an example of a case where a program x is pre-installed on say platform P_3, and where it is necessary to compile an assembler in order to run a compiler on platform P_3 in order to achieve the installation of the program x on platform P_3. Give a script for the installation.

8. Describe an example of a case where a program x is portable from say platform P_7 to platform P_5, and where it is necessary to compile a compiler on platform P_5 in order to achieve the porting of the program.

9. Program algebra characterizes programs as: texts that can be transformed (projected) to program objects (via PGLA). Such texts are program texts. You may consider this definition too strict, too formal, too limited or unattractive for any other reason. If so: is it possible to use the terminology of this paper to provide a more liberal definition of what it means for a bit sequence x that has been stored as file k in FFS_i to be a program.

[17] Consider a two-way infinite tape, having squares containing 0,1 or B (for blank), and having all non-blank symbols in a row without intermediate blanks at all times. Start with the description of an appropriate service interface.

The Turing Platform

Jan Bergstra*

February 17, 2000

Abstract

Using the tape of the Turing Machine as the main data structure for processor memory, a description is given of several highlights of a Turing-Machine-based theory of computation. The discussion takes place in the setting of program algebra, the famous tape being modeled by means of a coprogram.

Contents

*J.A. University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl

1 Introduction

This paper presumes acquaintance with program algebra and projection semantics as outlined in []. Moreover acquintance is presumed with the exposition on platforms, compilers and interpreters in []. We will follow the line of the latter paper, instantiating the so-called processor coprogram at site i (`cppi`) by means of a coprogram `cptmt` embodying the Turing Machine tape using a tape alphabet with 2 symbols (0,1,3) excluding the usual blank symbol B. Of course the whole story can easily be done for

three tape symbols or for any higher number. We have chosen 2 because that easily fits our expository demands. We will only consider a platform at a single site. Therefore no mention is made of site identifiers.

The following concepts and mechanisms will be defined in a precise manner below:

1. The Turing tape TMT, its intuitive form and role, as well as a precise rendering in the form of a coprogram with a well-defined service interface `sitmt` and a reply function.
2. The semantics of PGLA programs over `sitmt` is complemented with a count of the number of computation steps needed to reach termination.
3. The concept of a computable transformation of files.
4. The concept of a polynomial time computable decision problem (the class of P-problems). The concept of a non-deterministically polynomial time decision problem (NP-problems). The concept of an NP-complete decision problem.
5. The satisfiability problem for propositional logic, as an example of an NP-complete problem.

In this text the notations of program algebra (PGLA, PGLC), coprograms and platform analysis will be used without further explanation or justification.

2 A coprogram for TMT

The Turing Machine tape with two symbols 0 and 1 can be imagined as a two-way infinite sequence of squares, each square containing one of the symbols in {0,1}, and initially containing 0 on all squares, except two consecutive symbols 1. The square containing the left-most of these 1s is called the initial square zero (0). During computation there is always a square named square 0. This square is either zero the left-most square containing a 1, or it is the square visited by the head, if that happens to be more to the left. The head is the read and write unit fot the tape.

Each square can carry 1 bit of information, because it always contains either 0 or 1. During operation the contents of various squares are modified and inspected. This happens by means of the head of the tape unit.

2.1 cptmt

The tape will be modeled by means of a coprogram `cptmt`. A state of `cptmt` is determined by an integer denoting the current position of the head (relative to the current square zero) and an assignment of symbols to all squares. The assignments are also called valuations. Of course only finitely many squares have a content diferent from 0. Indeed, all computations having finite histories, no computation will be capable of writing an infinite number of 1s in finite time.

An assignment takes the form of a function `contentOfSquare()` from NN (the set of natural numbers) to {0,1}. In all cases from some number n upwards all values of `contentsOfSquare(-)` are equal to 0. Initially we have `contentOfSquare(0) = 1`, `contentOfSquare(1) = 1`, and `contentOfSquare(n+2) = 0`. A state is a pair $< n, c >$ of a natural number n and a tape valuation, n indicating the current position of the head (relative to the square zero).

The collection of states of `cptmt` is denoted with S_{cptmt}. It will turn out that each assignment can result after a finite number of instructions, and that all different assignments correspond to states having different reply functions. Therefore the exact collection of states of `cptmt` can be represented by the collection of assignments of bits to squares (with the restriction that only finitely many squares will carry a 1).

2.1.1 Service interface and reply function

The next step is to develop a useful service interface (Σ_{sitmt}) for `cptmt` and a reply function F_{tmt} for `cptmt`. We will identify `cptmt` with the coprogram $< \Sigma_{sitmt}, F_{tmt} >$. The actions of Σ_{sitmt} are as follows:

1. for $s \in$ 0,1 the action `test`(s) tests whether s equals the contents of the current square (i.e. the square at which the head is currently positioned). If so it returns `true`, otherwise `false`.

These 2 actions will not effect any change of state on cptmt.

2. for $s \in$ 0,1,B the action set(s) writes s on the current square, obliterating its previous content. By default the boolean true is returned in this case. The head position is not changed. All other squares have their contents unmodified by these actions.

3. for $s \in$ {left,right} the action mv(s) makes the head move one square to the left or right respectively. In the case the head is at a position > 0, the corresponding transition is from$< n, c >$ to $< n-1, c >$ or from $< n, c >$ to $< n+1, c >$ respectively.

   In the case the head is at position 0 it will again be at position zero after the execution of a mv(left) instruction, all tape contents having been shifted one step upwards. In the case of the execution of a mv(right) instruction two conditions must be distinguished. If square 0 contains a 0 before the move, the tape will be at square 0 after the move and all tape contents will have been shifted backwards one square. If square 0 contains a 1 the head moves to position 0. A default reply true is returned.

The reply function for cptmt can be defined from these ingredients in a standard fashion. A remarkable amount of the theory of computing takes place in the simple world of programs using cptmt. It is the objective of this paper to provide a survey of some salient parts of this theory. With cptmt we denote cptmt together with its underlying state space and transitions.

2.2 Information storage and retrieval

In order to use the computing power of the Turing tape a number of operators are used for storing data into and retrieving data from states of cptmt.

2.2.1 Computation exit status: getTstatus()

A computation may diverge and may terminate correctly (ending in !) or incorrectly (aborting via #). After termination an exit status is written into the termination status bit and the head stays at that point.

If after a terminating computation the current square contains 0 the computation has successfully terminated, if it contains 1 the computation has terminated due to an error casued by an execution of #. The termination status is retrieved from a state by the operator getTstatus().

2.2.2 getFile()

The operator getFile produces a bit string s from a state $< n, c >$ of cptmt as follows:

If $c(n+1) = c(n+2)$ then getFile$(n, c >= [])$. Otherwise getFile$(n, c >= [c(n)] \circ$ getFile $< n+2, c >$. In other words: the retrieved file starts to the right of the head, encoding 0 as 01 and 1 as 10 while using both 00 and 11 as 'end of file' indicators.

2.2.3 store(-)

The operator store(-) takes a bit sequence s as its argument. This bit sequence is stored on the tape.

Let s have length k. Obtain u from s by consecutively replacing 0 by 01 and 1 by 10. Clearly the length of u is $2k$. Denote the i-th element of u with u_i. Now store$(s)(< n, c >) =< n, c' >$ with c' as follows: (i) for $m < n$ $c'(m) = c(m)$, (ii) $c'(n) = c'(n+1) = 1$, (iii) for $n+2 < M < n+2+2k+1$, $c'(m) = u_{m-n-2}$, for $m > n+2+2k$, $c'(m) = c(m-2-2k)$.

In other words: s is encoded by coding 0 as 01 and 1 as 10, then 11 is prefixed and the result is inserted on the tape at the position of the head. Thereafter the head is placed at position n again.

3 PGLA program semantics

The functional (rather than behavioral) meaning of PGLA(Σ_{sitmt}) programs over cptmt is given by the application operator $\bullet$ linking program behaviors and coprograms. Application views a program (or rather its extracted behavior) as a transformation of coprograms:

$$|p|_{pgla} \bullet \mathbf{s} = \mathbf{s}'$$

expresses the fact that p executed over coprogram c leads to a terminating computation ending in s'. If the state s' is correctly terminated a file can be extracted. Specializing this to the Turing platform this leads to the following notation:

$$|x|_{pgla}^{fpp/tmt}(y_1, .., y_k) =$$

$$\texttt{getFile}(|x|_{pgla} \bullet \texttt{store}(y_1, .., y_k, \texttt{cptmt}))$$

provided correct termination has taken place. The superscript fpp indicates functional platform projection: a projection taking into account the knowledge about the processor memory coprogram aiming at obtaining a functional operator (i.e. a partial transformation of files). A shorthand for the above expression is

$$x \bullet \bullet_{pgla}^{tmt} y_1, .., y_k.$$

3.0.4 Decidability

A set of bit sequences V is decidable if there exists a PGLA(Σ_{cptmt}) program x such that

$$x \bullet \bullet_{pgla}^{tmt} y_1, .., y_k = [0] \text{ if } (y_1, .., y_k) \in \text{V}$$

and

$$x \bullet \bullet_{pgla}^{tmt} y_1, .., y_k = [1] \text{ otherwise.}$$

3.1 Non-deterministic choice

PGLAndc is PGLA equipped with an instruction ndc. That instruction produces no external state changes; instead it chooses between returning a true and a false. For the present purposes it is reasonable to have in mind that both options are selected with an equal probability. Behavior extraction for PGLAndc introduces a choice (we give two samples of behavior extraction equations):

$$|\texttt{ndc}; X|_{pglandc} = |X|_{pglandc}$$

$$|\texttt{+ndc}; u; X|_{pglandc} = |u; X|_{pglandc} + |X|_{pglandc}.$$

4 PGLCtli, an assembly notation for the Turing platform

The program notation PGLC with instructions taken from Sicptmtb is useful as an assembly language for TMT. It is useful to consider programs satisfying the additional constraint that termination will take place only by means of control exactly arriving at the point following the last instruction (either by executing the last instruction or by jumping over it). This subset of PGLC is denoted with PGLCtli. Sequential composition for two or more PGLCtli programs can be obtained by means of concatenation. So we will use PGLCtli in the role of ASLtmt.

The following grammar generates PGLC programs over Σ_{sitmt}:

```
PGLCPROGRAM =
  INSTRUCTION
  |INSTRUCTION;PGLCPROGRAM
INSTRUCTION =
  JUMP
  |ATOM
  |SIGN ATOM
  |#
  |JUMP
SIGN =
  +
  |-
ATOM =
  set(0)
  |set(1)
  |test(0)
  |test(1)
  |mv(left)
  |mv(right)
  }
JUMP =
  # NN
  | \# NN
NN =
  0
  |POSDIGIT
  |POSDIGIT DIGITS
DIGIT =
  0
```

```
  |POSDIGIT
POSDIGIT =
  1
  |2
  |3
  |4
  |5
  |6
  |7
  |8
  |9
DIGITS =
  DIGIT
  | DIGIT DIGITS
```

PGLCtli is a subset of PGLC. The following constraints must be satisfied. (i) The last instruction is not signed (no + no −), (ii) backward jumps do not jump outside the program (if the k-th instruction is a backward jump $\backslash\#n$ it is required that $n < k$, (iii) forward jumps jump to the last instruction plus one at most (if the k-th instruction is a forward jump $\#n$ and the program text has m instructions it is required that $n + k < m + 2$).

4.1 Projection semantics for PGLCtli

The language PGLCtli can be viewed as a collection of bit sequences by interpreting each character as two bytes according to the UNICODE encoding standard. The readable character sequence is a representation of the underlying bit sequence rather than the other way around. The result of this encoding can be used in the role of ASLtmtb. Leaving out the machine code as an intermediate, a projection of PGLCtli is obtained as follows: $\mathtt{pglctli2pgla}(x) = \mathtt{pglc2pgla}(x)$ if $x \in \mathrm{PGLCtli}(\Sigma_{sicptmtb})$ and $\#$ otherwise. The corresponding semantic equation reads:

$$|x|_{pglctli} = |\mathtt{pglctli2pgla}(x)|_{pgla}$$

The operation of PGLCtli programs on the platform `cptmt` is as follows (for a sequence of bit sequences $x, y_1, .., y_k$):

$$|x|^{fpp/tmt}_{pglctli}(y_1, .., y_k) =$$

$$|\mathtt{pglctli2pgla}(x)|_{pgla} \bullet (\mathtt{store}(y_1, .., y_k, \mathtt{cptmt}))$$

4.1.1 Abbreviations

Below the expression $|x|^{tmt}_{pglctli}(y_1, .., y_k)$ will be abbreviated to $x \bullet\bullet y_1, .., y_k$. Moreover $|x|\bullet \mathtt{s}$ will abbreviate $|x|^{fpp/tmt}_{pglctli} \bullet \mathtt{s}$.

4.1.2 Compositionality

The reason to prefer working with PGLCtli in the present setting is the following identity enabling easy reasoning about computations and intermediate stages of computations.

$$|x; y| \bullet \mathtt{s} = |y| \bullet (|x| \bullet \mathtt{s}).$$

This identity expressis that in the case of PGLDtli the effect of program application on a coprogram is compositional with respect to concatenation. It is important that p and q are complete PGLDtli programs.

4.2 The Halting problem for `tmtb`

In 1936 Alan Turing posed and solved the following problem (see [2]). Is there an effective method for deciding whether or not a computation $x \bullet \bullet y_1, .., y_k$ will achieve proper termination. This question has been called the Halting problem. Of course Turing used a different technical setup, having no need for assets like PGLCtli. The tape he used was reasonably similar to ours, however. Turing concluded that there cannot exist an effective method (an algorithm) for solving this problem, at least not a method that can be programmed in PGLCtli using the memory structure `tmt`.

Now one may think that the Turing tape is a very weak memory structure and that real computers can do much more. That is not the case, however. Ample evidence has been accumulated that all effective methods can be programmed over the Turing tape by means of a program notation like PGLCtli (or PGLA of course). The only problem that the tape confronts its users with is an efficiency problem. That is a nasty problem indeed but it does not reflect on very principal matters about computability and decidability.

It stands as an unchallenged 'thesis' (Church's thesis) that every computable transformation of bit sequences can be programmed by means of the Turing

tape. If this is ever refuted, the refutation will come from ideas like quantum computing, EPR based computing or even more futuristic discoveries.

The proof of the undecidability of the Halting problem is still a slightly complicated matter as it requires some programming in PGLCtli over `tmt`. It will be assumed that this programming is possible. In particular the existence of two programs will be presumed.

4.3 The Halting Problem

The set HP is defined as HP $= \{(x,y)|\exists\ z.\ x \bullet\bullet y = z\}$. HP contains complete information about program termination. HP has in fact two parameters: the program notation (in this case instantiated with PGLCtli($\Sigma pv1$)) and the coprogram (here `cptmt`).

The Halting Problem consists of algoritmically deciding membership of HP.

It is assumed that there is a program X able to decide the Halting problem. X will always lead to correct termination (making the termination status bit 0) and it will place a 0 or a 1 in the next bit (the first bit to be retrieved by `get()` depending on whether the two input sequences are a pair of program and data with the program terminating on the data.)

4.3.1 Statement of Turing's theorem

Turing has demonstrated that HP is not decidable by a PGLCtli program running over the Turing tape.

4.4 Proof of Turing's theorem

Suppose program $q \in$ PGLCtli decides HP. So for all bit sequences x and y:

$$q \bullet\bullet x, y = [0] \text{ if } \exists z\ x \bullet\bullet y = z \text{ and}$$

$$q \bullet\bullet x, y = [1] \text{ if } \nexists z\ x \bullet\bullet y = z$$

4.4.1 Two auxiliary programs

With some effort the existence of two PGLCtli programs p and r can be established providing the following functionalities:

$$|p| \bullet \mathtt{store}(z, \mathtt{s}) = \mathtt{store}(z, z, \mathtt{s})$$

where $\mathtt{s} \in S_{cptmt}$ is such that the head is on position zero. (From the definition of `store` it follows that this also holds for $\mathtt{store}(z, \mathtt{s})$ and $\mathtt{store}(z, z, \mathtt{s})(= \mathtt{store}(z, \mathtt{store}(z, \mathtt{s}))$.)

$$|r| \bullet \mathtt{s} = \mathtt{s}'$$

where $\mathtt{s}'$ is obtained from $\mathtt{s}$ by changing the bits on the two squares to the right of the head. Further, if the bit pattern has become has become 01 (i.e. an encoding of the negation of output 0 which in its turn represents `false`) concerning the question of halting) the program diverges, otherwise it terminates. Thus $v; r$ modifies the output generated by v from 1 to 0 if it was 1 and introduces a divergence if it was 0.

This implies that if $\mathtt{getFile(s)} = [1]$ $\mathtt{getFile(s')} = [0]$ with termination status 0 as well. But if $\mathtt{getFile(s)} = [0]$ this implies divergence of r.

4.4.2 A contradiction

Let $u \in$ PGLCtli $= p; q; r$. Suppose $(u, u) \in$ HP. Then for some state $\mathtt{s}$ of `cptmt`:

$$|q| \bullet \mathtt{store}(u, u, \mathtt{cptmt}) = \mathtt{s}$$

while the termination status of $\mathtt{s}$ is 0 and $\mathtt{getFile(s)} = [0]$.

It follows that $|r| \bullet \mathtt{s}$ diverges. Therefore $|u| \bullet \mathtt{store}(u, \mathtt{cptmt}) = |p; q; r| \bullet \mathtt{store}(u, \mathtt{cptmt}) = |q; r| \bullet \mathtt{store}(u, u, \mathtt{cptmt}) = |r| \bullet s$ diverges. Hence $(u, u) \notin$ HP.

Now assume $(u, u) \notin$ HP. Then $q \bullet\bullet u, u = [1]$. Therefore $q; r \bullet\bullet u, u = [0]$ and also $p; q; r \bullet\bullet u = [0]$. This, however, implies $u \bullet\bullet u = [0]$ which implies $u \in$ HP.

Both assumptions about the membersip of (u, u) of HP having been refuted, it may be concluded that the existence of the program q solving the halting problem on `cptmt` is a contradictory one: the halting problem cannot be algorithmically settled on the Turing platform.

5 Decision problems

5.1 BS separation problems

The collection of finite bit sequences BS is defined by the following BNF grammar: BS = 0 | 1 | BS BS. A bit sequence separation problem is a pair of non-empty subsets Vpos and Vneg of BS, such that Vpos $\cap$ Vneg = $\emptyset$.

A TMT-based solution to a BS separation problem consists of a PGLA program x (using instructions of Σ_{sitmt}) satisfying the following conditions:

(i) for each $w \in$ Vpos $\cup$ Vneg the computation $|x|^{fpp/tpl}(w)$ terminates correctly.

(ii) for each $w \in$ Vpos, $\texttt{get}(|x|^{fpp/tpl}(w)) = 0$

(iii) for each $w \in$ Vneg, $\texttt{get}(|x|^{fpp/tpl}(w)) = 1$.

A non-deterministic TMT-based solution to a BS separation problem consists of a PGLAndc (using instructions of Σ_{sitmt}) program x satisfying the following conditions:[1]

(i) for each $w \in$ Vpos $\cup$ Vneg the computation $|x|^{fpp/tpl}(w)$ terminates correctly irrespective of the successive boolean values returned by the action `ndc`. (Stated differently: all possible computations of x starting on the TMT state $\texttt{store}(w, \texttt{cptpl})$ will terminate correctly.)

(ii) for each $w \in$ Vpos, and for each computation R and state G $\in$ `states(cptpl)` with G equal to the final state of R $\texttt{get}(G) \in \{0, 1\}$, and for at least one computation R and final state G of R $\texttt{get}(G) = 0$

(iii) for each $w \in$ Vneg, and for each computation R and state G $\in$ `states(cptpl)` with G equal to the final state of R $\texttt{get}(g) = 1$

5.1.1 Specialized definitions I

A separation problem on BS is computable or decidable if there exists a TMT-based solution for it.

A separation problem on BS is non-deterministically computable (or non-deterministically decidable) if there exists a non-deterministic TMT-based solution for it.

A separation problem on BS is in P (polynomial time decidable) if there exists a TMT-based solution x for it such that there exists a polynomial function f (say $f(n) = c_1 \cdot n^{c_2} + c_3$) for some natural number constants c_1, c_2 and c_3 (for all arguments $n \in$ NN) such that for all $w \in$ Vpos $\cup$ Vneg the computation $|x|^{fpp/tpl}(w)$ takes no more than $f(\texttt{lengthOf}(w))$ steps.

A separation problem on BS is in NP (non-deterministically polynomial time decidable) if there exists a TMT-based non-deterministic solution x for it for which there exists a polynomial function f (say $f(n) = c_1 \cdot n^{c_2} + c_3$) for some natural number constants c_1, c_2 and c_3 (for all arguments $n \in$ NN) such that for all $w \in$ Vpos $\cup$ Vneg the computation $|x|^{fpp/tpl}(w)$ can take no more than $f(\texttt{lengthOf}(w))$ steps (irrespective of the sequence of non-deterministic outcomes of the test action `ndc` that occur within the computation).

5.2 Decision problems

A decision problem is a collection of objects C together with a subset Vc of C and a mapping $\lfloor - \rfloor$ from C into BS. This mapping is called the encoding of the decision problem. Often this mapping is not explicitly given. Instead a more elaborate definition of C is given and 'well-known methods' are to be used to encode C into BS in a 'natural way'.[2] Still we will require for a decision problem the existence of an encoding mapping. Each decision problem <C,V,$\lfloor - \rfloor$ > naturally is translated into a separation problem on BS. Indeed take Vpos = $\{\lfloor x \rfloor | x \in \text{Vc}\}$ and Vneg = $\{\lfloor x \rfloor | x \notin \text{Vc}\}$. The decision $x \in$ Vc is translated into deciding whether $\lfloor x \rfloor \in$ Vpos or $\lfloor x \rfloor \in$ Vneg

[1] Under the convention that 0 $\in$ NN represents `true` and that 1 $\in$ NN represents `false` a non-deterministic computation computes the disjunction of all its possible outcomes.

[2] If a problem is phrased as a subset Vc of C without explicit meantion of a mapping from C to BS, it is essential that an encoding of the problem as a BS decision problem by means of a mapping is done in a natural way. An unnatural mapping for instance is as follows: $\lfloor p \rfloor = 0$ if $p \in$Vc and $\lfloor p \rfloor = 1$ if $p \notin$Vc This encoding is unnatural because it requires a solution to the decision problem beforehand. Clearly it also defeats the entire purpose of encoding problems in a form appropriate for TMT computation.

5.2.1 Specialized definitions II

A decision problem is computable (also termed decidable) if its associated BS separation problem is computable.

A decision problem is non-deterministically computable (also termed non-deterministically decidable) if its associated BS separation problem is non-deterministically computable.

A decision problem is computable in polynomial time (denoted: is in P) if its associated BS separation problem is in P.

A decision problem is computable in non-deterministic polynomial time (denoted: is in NP) if its associated BS separation problem is in NP.

6 The satisfaction problem is in P

The propositional satisfaction problem concerns the following question: given a propositional formula together with a valuation of its logical atoms, decide whether the valuation satisfies the formula. There are many encodings possible. First of all a syntax is needed for logical atoms, valuations and propositions.

A BNF grammar for all of these ingredients is as follows (for instance):

```
BOOLEAN =
true
|false
DIGIT =
0
| 1
| 2
| 3
| 4
| 5
| 6
| 7
| 8
| 9
DIGITS =
DGIT
| DIGIT DIGITS
LOGATOM =
lat DIGITS
LOGATLIST =
LOGATOM
| LOGATOM,LOGATOMLIST
VAL =
{ }
|{LOGATOMLIST}
PROP =
BOOLEAN
| non(PROP)
| and(PROP,PROP)
| or(PROP,PROP)
PROPANDVAL = (PROP,VAL)
```

A valuation contains a collection of variables which get value `true`, all other values being given `false` by default. Strings of symbols are mapped onto sequences of bits via the ASCII encoding. Everyone of the characters mentioned in the above BNF is mapped onto a unique byte. A sequence of characters is mapped onto the concatentation of the corresponding bytes.

Having these definitions available the satisfaction problem can now be stated in more precise terms: given a bit sequence x, decide whether (i) it represents a string generated from the non-terminal PROPANDVAL, (ii) decide whether the first component, as a proposition, is satisfied on the second component viewed as a valuation (negative truth values being obtained for all logical atoms not mentioned in the set).

It requires a lot of Turing machine programming to establish an algorithm for this problem, working over TMT in polynomial time. That can be done, however. The intuition for the existence of such a program is as follows: first the proposition is copied in such a way that for each logical atom and for each operator additional space is reserved for a truth value. Then for each logical atom mentioned in the valuation `true` is recorded in the new space for each of

its occurrences in the proposition. Subsequently using a depth-first recursion over the logical structure of the proposition for all operator symbols a truth value is computed. The outermost truth value corrsponds with the required answer. This computation requires not more than $c \cdot n^2$ steps for an NN appropriate constant c.

6.1 The satisfiability problem is in NP

The satisfiability problem (for logical propositions) is as follows: given a bit sequence encoding a proposition (i.e. generated from the non-terminal PROP): decide whether there exists a valuation satisfying the proposition.

This problem is in NP. The argument is as follows: first use the non-deterministic feature to generate a valuation. The procedure should be performed in such a way that each of the logical atoms occurring in the given formula may be selected as `true`. Then apply the satisfiability algoritm. Generating a single valuation can be done in linear time, the subsequent satisfaction problem can be solved in polynomial time.

6.2 NP-complete problems

A problem in NP is said to be NP-complete if all other NP problems can be reduced to it in polynomial time. This definition depends on two further definitions: a computable transformation of decision problems, and a reduction of one decision problem to another.

6.2.1 P time problem transformations

A mapping g from BS into BS is called P time computable if there exists a PGLA program for TMT computing the mapping in plynomial time. The mapping is said to be computed by the program x if for all input sequences s the following holds:
(i) $\texttt{getTstatus}(|x|_{pgla} \bullet (\texttt{store}(s, \texttt{cptmt}))) = \texttt{true}$
(ii) $g(s) = \texttt{get}(|x|_{pgla} \bullet (\texttt{store}(s, \texttt{cptmt})))$, and
(iii) the computation of x on s is bounded in time by a polynomial in the length of s.

6.2.2 Reductions between problems

Given two decision problems U and V. U is said to be P time reducible to V if there exists a P time computable transformation f such that for all s in BS: $s \in$ U if and only if $f(s) \in$ V.

7 Cook's theorem: satisfiability is NP-complete

Cook's theorem ([1]) asserts that the satifiability problem for propositions of boolean logic is an NP-comlete problem. We will now consider the argument for that observation.

Let U be a decision problem in NP. We assume that x is a PGLAndc program over the service interface of `cptmt` solving U in non-deterministic polynomial time, say all computations are bounded by a polynomial $g(l)$ in the length l of problem instance s.

In order to demonstrate Cook's theorem it is now needed to transform a bit sequence s into a bit encoding of a proposition such that this proposition is satisfiable if and only if x has a computation on s resulting in output 1. The transformation must be computable in polynomial time in the length of s. We will provide an informal argument, the details being quite arduous indeed.

In $g(l)$ steps the head can visit no more than $g(l)$ squares. Certainly all squares lie within the range $[0, g(l)]$ Further each square has been visited at most $g(l)$ times. A complete tracing of the computation is possible by having information about $g(l)$ squares at $g(l)$ instants of time (one time step corresponding to one step of the compuation). It is also necessary to keep track of the program. Clearly not more than $g(l)$ instructions have been executed. Thus the computations are not changed if we unfold the given PGLA program and take the first $g(l)$ steps only. Let this finite program be y. With $\texttt{Del}(k, y)$ the program is denoted resulting from deleting the first k instructions from y (if k exceeds the length of y $\texttt{Del}(k, y) = \#$.

At this point a number of meaningful propositional constants can be introduced.

`atStepPosSquareIs(i,j,v)` expresses that at step `i` of the computation square `j` takes value `v`

$\in \{0,1\}$,

`atStepNegSquareIs(i,j,v)` expresses that at step `i` of the computation square `-j` takes value `v` $\in \{0,1\}$,

`atStepProg(i,j)` expresses that at step i of the computation the part of the program still to be executed equals `Del(i,`y`)`.

`atStepNdc(i,j)` expresses that at step i of the computation f a non-deterministic choice happens to be made the outcome is `true` if `j = 0` and `false` if `j = 1`. `Del(i,`y`)`.

A computation can be fully documented by means of giving the truth value of all these propositions for all time instances and all coordinates outlined above. This amounts to a limited number of propositions: not more then $4g(l)^2$. Further it is not difficult to see (though tedious to describe in full detail) that there must be a propositional formula containing the mentioned propositional constants, which is satifiable if and only if there is a computation of the machine to a correct terminating state ending with output [0]. This formula consists of the conjunction of

(i) a big conjunction where each conjunct threats a case: if program control is i, and if the head is at square j and if the value of square j is b_1, the value of square $j-1$ is b_2 and the value of square $j+1$ equals b_3 then 'the next state is ..'; and,

(ii) a simple proposition stating that if the next instruction is !, the output equals '0'.

The size of this propositional formula is polynomial in terms of the lenth of the originals computation. Satisfiability of this proposition coincides with the decision taken by the non-deterministic Turing machine. This concludes the argument.

7.1 Remark

It is a major open research problem wether or not the NP decision problems are in fact in P. It is widely believed that P $\neq$ NP, however. The importance of this matter is hard to overestimate. If P = NP is shown to be the case a major innovation in computing may emerge. If not, or if P $\neq$ NP is shown, things remain completely as they are. So in other words, the most clearly outstanding question in logical complexity is this: can one really improve the decision algorith for satisfiability consisting of a complete construction and inspection of the truth table of the proposition to be assessed. It should be noticed that decision problems for propositions emerge all over the place: optimization, scheduling design, software correctness, software testing. There is a significant industry for the design and implementation of algorithms capable of efficiently deciding satisfiability for restricted classes of propositions.

References

[1] S.A. Cook. The complexity of theorem proving procedures. In *Proceedings 3th Ann.Symp. on Theory of Computing*, pages 151–178. ACM, New York, 1971.

[2] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. Ser 2*, 42,43:230–265,544–564, 1936.