# Machine Function based Control Code Algebras

Jan A. Bergstra[1,2]

[1] University of Amsterdam, Programming Research Group
janb@science.uva.nl
[2] Utrecht University, Department of Philosophy, Applied Logic Group
janb@phil.uu.nl

**Abstract.** Machine functions have been introduced by Earley and Sturgis in [6] in order to provide a mathematical foundation of the use of the T-diagrams proposed by Bratman in [5]. Machine functions describe the operation of a machine at a very abstract level. A theory of hardware and software based on machine functions may be called a machine function theory, or alternatively when focusing on inputs and outputs for machine functions a control code algebra (CCA). In this paper we develop some control code algebras from first principles. Machine function types are designed specifically for various application such as program compilation, assembly, interpretation, managed interpretation and just-in-time compilation. Machine function dependent CCA's are used to formalize the well-known compiler fixed point, the managed execution of JIT compiled text and the concept of a verifying compiler.

## 1 Introduction

Machine models can be given at a very high level of abstraction by using so-called machine functions, a concept due to [6] as a basis. Machine functions are hypothetical functions, which may be classified by machine function types. Machine function types provide information about the expected inputs and outputs, or more general the behavior of a machine. Machine functions are named elements of machine function types. Machine functions are used as the primitive operators of a control code algebra. Machine functions may be viewed as black box containers of behavior. It is not expected that machine functions are actually either formally specified or algorithmically given in any detail. Important, however is the realization that different machine architectures may use different machine functions as realizations of the same machine function type. A number of issues is can be clarified using machine functions only: the so-called compiler fixed point, the distinction between compilation and interpretation, the role of intermediate code, managed execution and JIT compilation, and finally verifying compilation.

### 1.1 Motivation

The identification of machine function types and the description of machine models as composed from named but otherwise unspecified machine functions may be helpful for the following reasons:

- A multitude of different machine models and instantiations thereof can be described in some formal detail while at the same time ignoring the massive complexities of processor architecture and program execution.
- Machine function theory can be used to obtain logically complete machine models. A machine model is logically complete if all of its concepts are explained in internal technical terms in a bottom up fashion, permitting a reader to understand the model and stories about it without the use of subject related external knowledge and without understanding in advance the whole story of modern computer architecture.
- By giving names to machine functions simple calculations are made possible which may produce insights unavailable otherwise. In addition system specifications can be given in terms of combinations of requirements on a family of named machine functions. Machine function theory is a very elementary axiomatic theory of machine behavior, not claiming that the essential properties of machine functions are captured by axioms. The more limited claim, however, is that the role that machine functions play in a larger architectural framework can be analyzed in some useful detail.
- Pseudo-empirical semantics explains the meaning of codes and concepts regarding codes by means of hypothetical experiments with the relevant machine functions. The phrase 'the meaning of a code is given via its compiler' belongs to the dogma's of pseudo-empirical semantics. Pseudo-empirical semantics provides definitions close to what computer users without a background in formal semantics may have in mind when working with machines.

### 1.2 Control code algebras

Each machine function gives rise to an algebra with codes as its domain. Codes are represented as finite sequences of bits. The codes play the role of input data and output data as well as of control codes. As the main focus of the use of machine functions is in analyzing semantic generalities of control codes at a very abstract level these algebras will be referred to as control code algebras (CCA's).[3] Control code algebra (CCA) is the meta theory of the control code algebras. It might just as well be called machine function theory.

### 1.3 Scope of CCA

The simplest CCA may be viewed as another form of theory of T-diagrams from [5], and implicitly present in [7], which has been further developed in [6]. Similar work appeared in [1]. Clearly the limits of CCA are reached when modeling a phenomenon requires significant information concerning the details of machine code execution at the lowest level of abstraction. An extreme view might be that each insight in computer software processing which is independent of the details

[3] The acronym CCA is a mere abbreviation, having well over a million hits on Google and in no way specific. Its expansion 'Control Code Algebra' generates no single hit, however, at the time of this writing.

of code may lie within the scope of CCA. Here are some interesting issues for which that is unclear:

- Can the phenomenon of computer viruses can be modeled at the level of CCA. Many computer users are supposed to use their computers in such a way that infections are prevented, while no plausible argument can be given that these users must understand the details of how a machine works. The question may therefore be rephrased as follows: can an explanation via an appropriate CCA be given of what machine users must do in order to prevent virus infections of their systems. And, moreover can it be formulated what requirements systems must satisfy if these prescriptions for user behavior are to guarantee that systems remain clear of infections.
- Another question that may be answered at the level of an appropriate CCA is what an operating system is. Textbooks on operating systems never introduce the concept of an operating system as a concept that merits a definition in advance. Substantial questions concerning what minimum might be called an OS and whether or not an OS is a computer program or a control code are impossible to answer without proper definitions, however.
- A third question which may or may not lie within the scope of CCA is the question as to what constitutes a software component. By its nature the concept of a software component abstracts from the details of computation. But a first attempt to define software components in the framework of CCA reveals that some non-algorithmic knowledge about software component internals may be essential and outside the scope of CCA at the same time.

## 2 External functionality machine functions

A way to capture the behavior of a machine is to assume that it has a simple input output behavior, taking an number of bit sequences as its an put and producing similar outputs. Taking multiple inputs into account is a significant notational overhead but it will be indispensable for instance when discussing the notion of an interpreter (for the case of multiple inputs) whereas multiple outputs arise with a compiler that may produce a list of warnings in addition to its compilation result. A list (with length $k > 0$) of bit sequences (codes) is taken as an input and the result takes the form of one or more bit sequences. If $f$ is the name of a machine function and $n$ is a natural number then $f_n$ names the mapping that yields the $n$-th result. The result of $f$ on an input list $\boldsymbol{y}$ may be undefined ($M$, for meaningless) or divergent ($D$). In case of divergence none of the outputs gets computed, in the case of convergence from some index onwards all outputs are $M$, indicating that it has not been provided. The following axioms are imposed:

$$\forall n, m\,(f_n(\boldsymbol{y}) = D \rightarrow f_m(\boldsymbol{y}) = D)$$

$$\forall n, m\,(f_n(\boldsymbol{y}) = M \,\&\, m > n \rightarrow f_m(\boldsymbol{y}) = M)$$

$$\forall n\,(f_n(\boldsymbol{y}) \neq D \rightarrow \exists m > n\, f_m(\boldsymbol{y}) = M)$$

$$\forall n\,(f_n(\boldsymbol{y}^\frown\boldsymbol{z}) = M \rightarrow f_m(\boldsymbol{y}) = M)$$
$$\forall n\,(f_n(\boldsymbol{y}) = D \rightarrow f_m(\boldsymbol{y}^\frown\boldsymbol{z}) = D)$$

These rules express that

- The sequence of outputs is computed in a single computation. A divergence implies that no bit sequences can be produced, whereas an error ($M$) only indicates that a certain output is not produced. Only finitely many outputs bit sequences are produced, and beyond that number an error occurs.
- There is one algorithm doing the computation accessing more arguments when needed. When arguments are lacking an error ($M$) will occur, rather than $D$. Providing more inputs ($\boldsymbol{y}^\frown\boldsymbol{z}$ instead of $\boldsymbol{y}$) cannot cause an error ($M$). In other words if a computation on a list of inputs runs into an error than each computation on a shorter list must run into an error as well.
- A divergence taking place on a given sequence of inputs cannot be prevented by supplying more inputs. (This in contrast with an error which may result from a lack of inputs.)

### 2.1 External functionality.

A machine function produces external functionality if the transformations it achieves are directly contributing to what users intend to do. A typical example may be a formatting/typesetting functionality. Often user commands directly invoke the operation of an external functionality.

External functionality machine functions describe external functionalities. Their operation can (but need not) be exclusively under the control of manually entered user commands. Here is a somewhat artificial example involving a compiler and a cross-assembler producing executable code for another machine. Because the produced code must be moved elsewhere before use, its machine may be regarded external functionality.

**Using bit sequence machine functions, an example.** One may imagine a machine $P$ powered by the, bit sequence machine function *Fc*, requiring one input and producing three outputs for each input, and the bit sequence machine function *Fd* also requiring a single input and producing 2 outputs in all cases. The inputs are taken from a stack, the first input from the top and so on, while the outputs are placed on the stack in reversed order, after removing the inputs.

For instance *Fc* takes a program in and compiles it in the context of a list of programs (encoded in the second argument), producing the output code, a listing with error messages and warnings and an assembled version of the compiled code. *Fd* applies a disassembler to its first argument producing the result as well as a listing of errors and/or warnings.

A manual operation of the system is assumed with the following instructions: `s.load:`$f$, pushing file $f$ on the stack and `s.store:`$g$, placing the top of the stack in file $g$ while subsequently popping the stack. Here `s` stands for the machine command interface able to accept and process manual user commands. `s.Fc` is

the command that invokes the operation of the first of the two machine functions and produces three output sequences that are placed on top of the stack. `s.Fd` is the command for the second one, which places two results on the top of the stack consecutively. How such commands are entered is left unspecified in the model at hand.

Consider the command sequence

$CS =$ `s.load:f;s.Fc;s.store:g;s.pop;`
`s.pop;s.load:g;s.Fd;s.store:h;s.pop`

If the content of `f` before operation is $x$ then after performing $CS$ the content of `h` is $Fd_1(Fc_1(x))$. This output is probably empty if either one of the two commands lead to error messages, whereas if the error message output is empty on may assume that the other outputs are 'correct'. It also follows from this model that the operation of the system is deterministic for command sequences of this kind. Several potentially relevant system properties may now be formulated:

**Code is produced unless errors are found** $Fc_2(x) = [\,] \rightarrow Fc_1(x) \neq [\,]$
**Disassembly succeeds unless errors are found** $Fd_2(x) = [\,] \rightarrow F_1(x) \neq [\,]$
**Disassembly inverts of assembly** $Fc_2(x) = [\,] \rightarrow Fd_1(Fc_3(x)) = Fc_1(x)$

The use of machine functions in the examples mentioned above is in making these requirement descriptions more readable than it would be possible without them. With these few primitives there appears to be no significant reasoning involving calculation, however.

**Non-programmable machines.** If a machine is given by means of a finite list of machine functions one may imagine its user to be able to enter inputs, select a function to be applied and to retrieve outputs thereafter. We will not provide syntax or notation for the description of these activities. As it stands a non-programmable machine may be given in terms of finite listing of machine functions. As the number of machine functions of a machine increases it becomes increasingly useful to represent machine functions as codes and to use a single universal machine function from which the other machine functions are derived. In this way a control code dependent machine emerges. This universal machine function is used to bring other codes into expression. Therefore the phrase code expression machine functions will be used rather than the term universal machine function, which is somewhat unclear regarding to the scope of its universality.

## 3 Code expression machine functions

It is now assumed that the first argument of a machine function consists of a bit sequence which is viewed as control code for a machine. Through the application of the machine function the control code finds its expression. Formally there is no difference with the case of an external functionality machine function. But the idea is that without a useful control code (i.e. a first argument for the machine function) no significant external functionality may be observed. It is a reasonable

assumption for a simplest model that a machine which can be controlled via exchangeable data must be controlled that way. At this stage the important question what makes a bit sequence a control code (rather than just its place in the argument listing of a machine function) is left for later discussion.

For code expression machine functions another notation is introduced which separates the first and other arguments. A notation for the $n$'th result of a code expression function taking $k$ arguments (except the code argument $x$) is as follows:

$$x \bullet \bullet^n y_1, .., y_k$$

By default the first argument is meant of no superscript is given:

$$x \bullet \bullet y_1, .., y_k = x \bullet \bullet^1 y_1, .., y_k$$

If the name $f$ is given this can be made explicit with $x \bullet \bullet_f^n y_1, .., y_k$. The semantics of a control code $x$ w.r.t. a machine function is the family of all machine functions taking $x$ fixed, denoted $|x|_{\bullet\bullet}$. Thus semantic equivalence for control codes reads

$$x \equiv_{beh} z \leftrightarrow |x|_{\bullet\bullet} = |z|_{\bullet\bullet} \leftrightarrow \forall n, k, y_1, .., y_k (x \bullet \bullet^n y_1, .., y_k = z \bullet \bullet^n y_1, .., y_k).$$

With an explicit name of the machine function this reads:

$$x \equiv_{beh} z \leftrightarrow |x|_{\bullet\bullet_f} = |z|_{\bullet\bullet_f} \leftrightarrow \forall n, k, y_1, .., y_k (x \bullet \bullet_f^n y_1, .., y_k = z \bullet \bullet_f^n y_1, .., y_k).$$

Bit sequence generating machine functions are less useful when it comes to the description of interactive systems. But code expression machine functions are very simple as a concept and they can be used until a lack of expressive power or flexibility forces one to move to a different model.[4]

### 3.1 Split control code machine models

A code expression machine function $- \bullet \bullet_f -$ determines all by itself a machine model. For an execution, which takes a single step, a triple of the code and a sequence of inputs and the machine function are needed. This may be formalized as $m_f(x, \boldsymbol{y})$. The code is not integrated in the machine in any way. Thus it is

[4] The discussion may be made more general by using a terminology consistent with the possibility that a machine function produces an interactive behavior (i.e. a process). A bit sequence generating machine function is just a very simple example of a behavior. If the behavior of a machine is described in terms of a polarized process its working may be determined through a function that produces a rational polarized behavior over a given collection $A$ of basic actions from the codes that have been placed in the machine as an input or as a control code. The reason to consider a mapping function, say $F_M : BS \times BS \times BS \to BPPA(A)$, as the description of a machine if a control code dependent machine is to be compared with a programmable machine. BPPA is the algebra of basic polarized process from [2]. As will become clear below polarized processes are well-suited for the specification of programs and programmed machine behavior. In the case such a machine needs to be viewed as a control code dependent machine a polarized process machine function results.

implausible to speak of a stored code. For that reason the model is classified as a split control code machine model. This is in contrast with a stored code machine model for which it is required that code storage is achieved by means of the same mechanisms as any storage during computation. As nothing is known about these mechanisms due to the abstract nature of machine functions, being no more than formal inhabitants of their types, the model cannot be classified as a stored control model.

Having available the notion of a split control code machine, it is possible to characterize when a code is an executable for it: $x$ is an *executable* for $\bullet\bullet_f$ if for some $\boldsymbol{y}$, $x \bullet \bullet_f \boldsymbol{y} \neq M$. Because this criterion depends on an application of the code in an execution it is a dynamic criterion. In practice that may be useless and entirely undecidable. For that reason a subset (or predicate) $E_c$ of the executables may be put forward as the collection of 'certified' executables. Here it is understood that certification can be checked efficiently. A pair $(\bullet\bullet_f, E_c)$ is a split control machine with certified executables.

It is always more or less taken for granted that modern computer programming is based on the so-called stored program computer model. In the opinion of the author a bottom up development of the concept of a program starts with a split program machine model, however, the register machine constituting a prime example of a split program machine model. Having available a split program machine model one may then carry on to develop stored program machine models. The usual objection against this argument is that a Turing machine ([9]) constitutes a stored program machine model already. That, however, is debatable because it is not clear which part of the tape content or state space of a Turing might be called a program on compelling grounds. Usually some intuition from an implicit split machine model is used to provide the suggestion that a part of the tape contains a program in encoded form. That, however, fails to be a compelling argument for it being a program.

### 3.2 Two conceptual issues on split code machine models

The split control code machine model aims at providing a very simple account of control code dependent machines while ignoring the aspect of the control code being either a program or being cast in software. There are two philosophical issues that immediately emerge from the introduction of the concept of a split control code machine.

**The control code identification problem.** An explanation is needed for why the first code argument is said to contain the control code. It seems to be a rather arbitrary choice to declare the first argument the control code argument rather than for instance the second argument. This is the control code identification problem for an external machine function. The question is about the machine function and irrespective of any of its actual arguments. So the problem is: determine, given a code expression machine function, which of its argument contains the control code if any.

Below in 5.2 we will see that the situation is not symmetric indeed and sound reasons may exist for taking one code argument to play the role of containing the control code rather than another argument.[5]

**The program recognition problem.** An obvious question raised by the split control code machine model is this: under which circumstances is it appropriate to view an executable control code as a program? This is the program recognition problem. It presumes that the control code identification has successfully led to the classification of an argument position as the (unique) position for control code. This question is important for an appreciation of the distinction between control code an programs. This question can be answered first by means of the hypothetical program explanation principle which has been proposed in [3].

## 4 Split program machine models

Complementary to split control code machine models there are split program machine models. For the development of CCA it is in fact immaterial how the concept of a program is defined. What matters is that for some code to be called a program some definition of what it is to be a program is needed. Let some theory of programming be given which provides one with a reliable definition of the concept of a program.

A split code machine model (i.e. a code expression machine function) qualifies as a split program machine model if there is for each code a mapping to an acknowledged program notation (which may be hypothetical) for a (hypothetical) machine model such that the machine function describes the behavior of the (hypothetical) machine as programmed by a code (viewed as a program). Thus each code may be read as a program for a machine with a well-understood operational meaning which produces results that happen to coincide with those of the given machine function. In other words the code expression machine function is an abstraction of a split program machine model.

As a notation for a split program machine model with name $f$ we take

$$x \bullet \bullet^{n}_{\mathrm{spm}-f} y_1, .., y_k$$

The semantics of a program $u$ in a split program machine model is derived in a completely similar style to the split control code case with notation $|u|_{\bullet\bullet_{\mathrm{spm}-f}}$.

A split program machine model will also called an analytical architecture because it focuses on the analysis and explanation of behavior, whereas a split control code machine model may be called a synthetic architecture because it focuses on the components from which the machine is physically synthesized, without any commitment to the explanation of behavior.

[5] When investigating stored control code machines or stored program machines this question reappears in the following form: which memory compartment contains control code or program code?

### 4.1 Hypothetical architectures and program recognition.

For this section it will be assumed that a split program machine is like a split control code machine with the additional information that the control code constitutes a program irrespective of the definition of a program that has been used. Further it is assumed that the behavior for a split program machine can be found in such a way that the program helps to understand the code expression machine function (or process machine) function at hand. That is to say, the split program machine is considered more easily understood or explained because its program and the working of that program is known and it is also known how the program is transformed into the produced behavior by means of interaction with other relevant parts of the machine.

A control code is just data without any indication that it has the logical structure of a program.

Having available the concept of a split control code machine model one may then investigate under which conditions a split control code may appropriately be called a control code and even a program. This is the program recognition that was problem mentioned above.

**The program recognition problem: an informal solution.** A code is a program if it can be viewed as the product of some computable transformation translating (machine) programs for some conceivable programmable architecture to code for a split code machine. This state of affairs indicates that the reasons for classifying code as a program lie in the possibility of reverse engineering, or rather disassembling, the code into a program for some programmable machine, which may serve as an explanation for the given code.

**The program recognition problem: a formalized solution.** Given the split control code machine $\bullet\bullet_f$ and a collection $R$ of relevant functionalities for $\bullet\bullet_f$, a control code $x$ is a program if there exists a split program machine model[6] $\bullet\bullet_{\mathrm{spm}-g}$ and a computable code generation mapping[7] $\psi_{\mathrm{g2f}}$ from programs for $\bullet\bullet_{\mathrm{spm}-g}$ to codes for $\bullet\bullet_f$ such that

- *Hypothetical program existence:* $x$ is in the range of $\psi$ (e.g. for some $z$, $x = \psi_{\mathrm{g2f}}(z)$),
- *Hypothetical assembler soundness:* for all programs $u$ (for $\bullet\bullet_{\mathrm{spm}-g}$) $|\psi_{\mathrm{g2f}}(u)|_{\bullet\bullet_{\mathrm{spm}-f}} = |u|_{\bullet\bullet_g}$, and:
- *Functional assembler completeness:* for all control codes $x$ for $\bullet\bullet_f$: if the behavior $|u|_{\bullet\bullet_f}$ belongs to the collection $R$, which covers the relevant functionalities for which the machine model has been designed, then either
  - $u$ has a disassembled version ($z$ with $\psi_{\mathrm{g2f}}(z) = x$)) (which qualifies as a program for the split program model architecture providing the same functionality), or otherwise,

[6] The hypothetical programmed machine architecture.

[7] Also called assembler mapping, the inverse being an assembly projection.

- there is a program $z$ such that $|z|_{\bullet\bullet_{\mathrm{spm}-g}} = |x|_{\bullet\bullet_f}$ and consequently $|\psi_{\mathrm{g2f}}(z)|_{\bullet\bullet_f} = |x|_{\bullet\bullet_f}$.

This last condition guarantees that the split program machine has not been concocted specifically to turn the particular control code $x$ into a program (according to our proposed definition) but instead that this hypothetical machine provides an explanation of the full operational range of $\bullet\bullet_f$ by providing a program for each relevant behavior.

### 4.2 Control code need not originate through programming

One may wonder whether split control code is in all cases a transformation product of programs. If that were true the conceptual distinction between programs and code is marginal, a matter of phase in the software life-cycle only, and as a consequence the concept of control code is only seemingly independent of that of a program. We give two examples of control code that fails to qualify as a program. The conclusion is drawn that there are (at least) two entirely different strategies for computer software engineering, only one of which involves programming. Machine functions provides an abstraction level that both strategies have in common.

**Two examples of non-programmed control code.** A counterexample to the hypothesis that all control code originates as the result of program production may be as follows: one imagines a neural network in hardware form, able to learn while working on a problem thereby defining parameter values for many firing thresholds for artificial neurons. Then the parameter values are downloaded to a file. Such a file can be independently loaded on a machine, depending on the particular problem it needs to address. These problem dependent parameter files can be considered control code by all means. In all likelihood they are not programs in any sense supported by our theory of programming, however. The particular property of neural networks is their ability to acquire control code along another path than human based computer programming.

Another example of control code which may rather not be understood as a program is the geographical information downloaded in a purely hardware made robot together with a target location it is supposed to find. The robot will apply its fixed processing method to these data, but the data determine the behavior of the robot. The loaded data constitute control code (this follows from the absence of other software in the robot). But programming (compliant with the assumed theory of computer programming) has played no role in the construction of this control code.

In both mentioned examples of control code which is not the result of program transformation artificial intelligence plays a role. In the case of the neural network the learning process itself is an example of artificial intelligence, whereas in the case of the robot the processing performed by the robot must involve significant AI applications. In the robot case the preparation of control code is

similar to the briefing a human being might get when confronted with the same task.

**Two software engineering strategies.** Machine learning may altogether be understood as an alternative form of control code production, contrasting with the currently more usual control code development starting with computer programming and followed by compilation and assembly phases.

Examples of non-programming based software engineering outside artificial intelligence seem to be hard to find. Both examples above embody different aspects of artificial intelligence based software engineering: control code construction by means of artificial intelligence and control code construction for an artificially intelligent system play a role in non-programming based software engineering. Therefore software engineering is in terms of software construction techniques covered strictly larger than computer programming, as it also covers these AI based techniques.

**A third option: genetic control code evolution.** A third option for software engineering that may avoid programming as well as neural network training lies in the application of genetic algorithms. This involves a number of operators for constructing new control codes from (pairs of) known ones. Then by randomly applying these operations on some initial codes and by filtering out codes that are bad solutions for the software engineering problem at hand an evolutionary process may produce adequate solutions.

## 5 The code identification problem

The essential simplification of split control code in comparison to split programs lies in the decision to view code as binary data without any need to explain why these data play the role of programs or may be best understood as programs. This, however, leaves open the question as to why a certain code is classified as a control code.

### 5.1 Splitting control code from inputs

Given a split control code machine, one may take its first argument as just one of the arguments, thus obtaining a code expression machine function. Looking at the split control code machine from this level of abstraction, a question similar to the program recognition problem appears: why has the first of the arguments been split of to play the role of the control code. The notion of *overruling* is now proposed to deal with this issue.

**Symmetry prevents control code identification.** It is useful to experiment for a while with one simple design for the split control code machine, by assuming that there is just a single input. In particular consider the split control

code machine $\bullet\bullet_f$. By forgetting the control code role of the first argument a control code independent machine is found (say $S_f$) and its behavior is given by $S_f(x, \boldsymbol{y}) = x \bullet\bullet_f \boldsymbol{y}$ The only possible justification for making the first argument play the role of a control code and the second code argument the role of an input code must lie in properties of the code expression machine function $S_f$. Indeed suppose, hypothetically that for all $x$ and $y$, $S_f(x, y) = S_f(y, x)$. Then the symmetry is complete and a justification for putting the first argument of $S_f$ in the role of control code and the other argument in the role of input data cannot be found.

A justification for putting the first argument in the control code role can only be found if the code expression machine function is asymmetric and if, moreover, the first code argument ($x$) can be said to be 'more in control' than the second one ($y$) or any other argument. The control code identification problem will now be simplified to the case that the code expression machine function has exactly two inputs. The task is to select the control code input, if that can be done. Here are three informal criteria that may be applied to argue that indeed, the first argument has the role of a control code:

**Overruling argument positions.** For a two place function $F : BS \times BS \rightarrow BS \cup \{D, M\}$ the first argument overrules the second if there can be found different sequences e.g. $O_1 =$ "0" and $O_2 =$ "1" and codes $x_1$ and $x_2$ such that for all code arguments $y$, $F(x_1, y) = O_1$ and $F(x_2, y) = O_2$.

If the first argument overrules the second, the second one cannot overrule the first argument at the same time. Indeed suppose that $O_3 \neq O_4$ and that $y_3$ and $y_4$ are found such that for all $x$, $F(x, y_3) = O_3$ and $F(x, y_4) = O_4$. Then it follows that $O_3 = F(x_1, y_3) = O_1 = F(x_1, y_4) = O_4$, which contradicts the assumptions.

### 5.2 Control code overrules non-control code

Consider $S_f(x, Y)$, then the first argument is said to be at the control code position (for $S_f$) if the first argument overrules the second argument. If this condition is met that condition solves the control code identification problem and justifies the notation $x \bullet\bullet_f y = S_f(x, y)$.

The criterion of overruling becomes more interesting if there are more than two code arguments needed to have a successful (not yielding $M$) computation of the split control code function. Below the concept of a split interpreter control machine model will be outlined. In the case of split interpreter control two argument positions have great influence on machine operation: the control code position where the interpreter control code is located and the position where the code to be interpreted is located. In this setting the first position overrules the second position and the second position overrules the third position. But the dependence of behavior from the second argument is more flexible than for the first argument.

# 6 Control code assembly notations

In the sequel the assumption is made that for the class of split control code machines of interest all control codes are actually transformed programs in the sense of 4.1. This assumption justifies the jargon used.

## 6.1 Executables

Given a split control code machine the control codes may as well be called executables. This phrase is often used if control codes are transformed programs. It is common to make a distinction, however, between arbitrary control codes and executable control codes, or simply 'executables' where executables are those control codes really meant for execution on the model. Lacking any intrinsic criterion as to which codes are to be considered executable it is reasonable to assume that some collection $E_c$ of codes comprises the executables. This collection is specific to the code expression machine function of the machine model under consideration.

## 6.2 Assembly and executable

One may consider the notion of an assembly notation for a given split control code machine with certified executables $E_c$. An assembly code should be a useful tool for a human author in the production of control code. It allows the production of readable and meaningful texts which may be automatically transformed into control codes by means of an assembler which is given in the form of another control code. In this discussion the question who wrote the assembler and how this may have been done without the help of an appropriate design code is ignored. The simplest view is that producing the assembler in the absence of a suitable control code design notation has been an enormous amount of work that may be seen as a part of the investment of the development of a split control code machine, which is useless otherwise.

A control code assembly notation (say $A$) is simply viewed as a subset of the possible codes. An assembler for $A$ is an executable control code $n{:}a2e{:}e$. Here the colon is part of the name, which thereby carries the following information: code reference $n$ including a version number, functionality $a2e$ (from assembly to executable) and code class $e$ (for executable). The assembler must satisfy this criterion: for each code $x{:}a \in A$, $n{:}a2e{:}e \bullet\bullet x{:}a \in E_c$. Thus, a compiler transforms each control code design into an executable.[8]

## 6.3 Assembling an assembler in assembly

An interesting thought experiment now runs as follows. Let an assembler $x1{:}a2e{:}e$ for $A$ be given. Assume that a version of an assembler for $a$ is made available,

[8] It is a common additional requirement that for codes outside $A$ a non-executable output is produced together with a second output which contains a list of so-called errors allowing the control code designer to find out what is wrong.

written in the assembly notation $A$ represented by $a$ itself.[9] The following name will be used for this new version of the compiler: $u{:}a2e{:}a$. The name combines a local name ($u$), a functionality ($a2e$) and a notation that is used ($A$). The new assembler cannot be used without preparatory steps. It needs to be assembled itself by means of the existing assembler which is available in an executable form already. With that in mind first of all the correctness of this new assembler can be formalized as follows:

(1) It can be assembled successfully by means of the given assembler i.e., $x1 : a2e : e \bullet \bullet u : a2e : a \in E_c$, which permits one to write $x2 : a2e : e$ for $x1{:}a2e{:}e \bullet \bullet u{:}a2e{:}a$.
(2) $x2{:}a2e{:}e$ is a control code that transforms each $a$ code into an executable.
(3) For all control code designs $y{:}a$ the two executable assemblers mentioned in (2) produce equivalent control executables. That is, $|x1{:}a2e{:}e \bullet \bullet y{:}a|_{\bullet\bullet} = |x2{:}a2e{:}e \bullet \bullet y{:}a|_{\bullet\bullet}$.

**Assembling an assembler once more.** Now the following well-known question can be raised: is it useful to use $x2{:}a2e{:}e$ to assemble the code $u{:}a2e{:}a$ once more? Let $x3{:}a2e{:}e = x2{:}a2e{:}e \bullet \bullet u{:}a2e{:}a$. This must be an executable because $x2{:}a2e{:}e$ is an assembler for A. Due to the assumption that $x2{:}a2e{:}e$ is a correct compiler this second new assembler must also be a correct one because it determines a semantics preserving code transformation, i.e $x3{:}a2e{:}e =_{behavior} x2{:}a2e{:}e$, or equivalently: $|x3{:}a2e{:}e|_{\bullet\bullet} = |x2{:}a2e{:}e|_{\bullet\bullet}$.

Why may it be an advantage to make this second step? The second step takes the new executable assembler (obtained after assembling with the old one) to assemble the new non-executable. A useful criterion here is code compactness, measuring the length of control codes. Now suppose that the new assembler produces much shorter code than the old one, then the second step is useful to obtain a more code compact executable for the new assembler. The executable version of the new assembler $x2{:}a2e{:}e$ has already the virtue of producing short codes but its own code may still be 'long' because it was produced by the old assembler.

**The assembler fixed point.** The obvious next question is whether it is useful to repeat this step once more, i.e. to use $x3{:}a2e{:}e$ once more to assemble the new assembler. It is now easy to prove that this will bring no advantage, because it will produce exactly the code of $x3{:}a2e{:}e$. This is the so-called compiler fixed point phenomenon (but phrased in terms of an assembler). In more detail, let $x4{:}a2e{:}e = x3{:}a2e{:}e \bullet \bullet u{:}a2e{:}a$. Because $|x3{:}a2e{:}e|_{\bullet\bullet} = |x2{:}a2e{:}e|_{\bullet\bullet}$, which has been concluded above, $x3{:}a2e{:}e \bullet \bullet u{:}a2e{:}a = x2{:}a2e{:}e \bullet \bullet u{:}a2e{:}a = x3{:}a2e{:}e$.

This leads to the following conclusion: if a new assembler is brought in for a notation for which a assembler in an executable form is available and if the new code is written in the assembly notation itself then it is necessary to translate

[9] This is not implausible as the compiler is a complex piece of control code which may profit from advanced design technology.

the new assembler with the old executable assembler first. Thereafter it may be an advantage to do this once more with the executable assembler just obtained. Repeating this step a second time is never needed. This is a non-trivial folk-lore insight in systems administration/software configuration theory. It exists exactly at the level of abstraction of machine functions.

**The assembler application notation.** The machine function of an assembler code merits its own notation: $x{:}a \bullet \bullet_a \boldsymbol{y{:}d} = (x1{:}a2e{:}e \bullet \bullet x{:}a) \bullet \bullet \boldsymbol{y{:}d}$.
Using this notation the versions of the executable code that occur in the description of the fixed point are: $x3 : a2e : e = u : a2e : a \bullet \bullet_a u : a2e : a$, and $x4{:}a2e{:}e = (u{:}a2e{:}a \bullet \bullet_a u{:}a2e{:}a) \bullet \bullet u{:}a2e{:}a\,(= x3{:}a2e{:}e \bullet \bullet u{:}a2e{:}a)$. The fixed point fact itself reads: $(u{:}a2e{:}a \bullet \bullet_a u{:}a2e{:}a) \bullet \bullet u{:}a2e{:}a = u{:}a2e{:}a \bullet \bullet_a u{:}a2e{:}a$.

**Correct assembly code.** An assembly code is correct if its image under the assembler is an executable, thus $z \in A_c$ if $x1{:}a2e{:}e \bullet \bullet z \in E_c$.[10] The correct codes depend on the available assembler codes. Improved assemblers may accept (turn into an executable) more codes, however, thereby increasing the extension of 'correct assembly codes'.

## 7 More dedicated codes

It is reasonable to postulate the availability of an executable code which allows to test the certified executability of codes. Let $a1 : t4e : e$ be the application code $a$ testing for executability (functionality $t4e$) in executable form. Then $a1{:}t4e{:}e \bullet \bullet u{:}d$ produces an empty file on input $u{:}d$ (data file $u$ which may or may not admit the stronger typing $u{:}e$) if $u{:}d$ is a certified executable (i.e. $\in E_c$), and a non-empty result e.g. containing a listing of warnings or mistakes otherwise.

### 7.1 Source level code

A code notation for source level code $S$ may be modeled as a collection of bit sequences. Its codes are denoted e.g. with $x{:}s$. A special notation for the application of a source level code is useful. The action of source level code on a sequence of inputs $\boldsymbol{y}$ is given by $x{:}s \bullet \bullet_s \boldsymbol{y}$. Given a compiler $u{:}s2a{:}a$ for $S$ this can be defined by: $x{:}s \bullet \bullet_s \boldsymbol{y} = (u{:}s2a{:}a \bullet \bullet_a x{:}s) \bullet \bullet_a \boldsymbol{y}$.

It is assumed that $u{:}s2a{:}a \bullet \bullet_a x{:}s$ never produces $D$ and that the first output is a correct assembler code provided the second result $(u{:}s2a{:}a \bullet \bullet_a^2 x{:}s)$ is the empty sequence. The correct $S$ codes[11] are denoted with $S_c$.

[10] $A_c$ may be called the pseudo-empirical syntax of $A$.
[11] Compiler based pseudo-empirical syntax of $S$.

**Compiling a 'new' compiler.** Given a compiler $u1{:}s2a{:}a$ for $S$ a new compiler $u{:}s2a{:}s$ for $S$ written in $S$ may be provided. Then to make this new compiler effective it should itself be compiled first and subsequently be used to compile itself. Then the compiler fixed point is found entirely similar to the case of the assembler fixed point mentioned before. The new compiler may be compiled into assembly: $u2{:}s2a{:}a = u1{:}s2a{:}a \bullet\bullet_a u{:}a2e{:}s$. It is then required that for all $y{:}s \in S_c$, $|u1{:}s2a{:}a \bullet\bullet_a y{:}s|_{\bullet\bullet_a} = |u2{:}s2a{:}a \bullet\bullet_a y{:}s|_{\bullet\bullet_a}$.

**The compiler fixed point.** The next phases are given using source level application notation: $x{:}s \bullet\bullet_s \boldsymbol{y{:}d} = (u1{:}s2a{:}a \bullet\bullet_a x{:}s) \bullet\bullet_a \boldsymbol{y{:}d}$. Using this notation the versions of the assembler code that occur in the description of the fixed point are: $u3{:}s2a{:}a = u{:}s2a{:}s \bullet\bullet_s u{:}s2a{:}s$, and $x4{:}a2e{:}e = (u{:}a2e{:}a \bullet\bullet_a u{:}a2e{:}a) \bullet\bullet u{:}a2e{:}a$. The fixed point fact itself reads: $(u{:}s2a{:}s \bullet\bullet_s u{:}s2a{:}s) \bullet\bullet u{:}s2a{:}s = u{:}s2a{:}s \bullet\bullet_s u{:}s2a{:}s$.

### 7.2 Interpreted intermediate code

Taking three or more inputs and two outputs it becomes possible to view the first input as the control code for an interpreter, the second input as a control code to be interpreted and the third input (and further inputs) as an ordinary argument, while the first output serves as the standard output and the second output serves as error message listing when needed. In actual computer technology compilation and interpretation are not alternatives for the same codes. A plausible setting uses an interpreted intermediate code notation $I$ Correct intermediate codes must be defined via some form of external criterion. Thus $I_c$ may be the collection of certified intermediate codes. The functionality of interpreters for $I$ is denoted $int4i$, an interpreter for $I$ is an executable $u{:}int4i{:}e$. The application of $I$ code is then defined as $x{:}i \bullet\bullet_{int4i} \boldsymbol{y} = u{:}int4i{:}e \bullet\bullet x{:}i \frown \boldsymbol{y}$.

**Compiling to interpreted code.** A compiler from source code notation $S$ to $I$ is an executable $u{:}s2i{:}e$. Having available a compiler to $I$ an alternative definition of the correct codes for $S$ is given by: $x \in S_{c:int}$ if and only if $u{:}s2i{:}e \bullet\bullet x \in I_c$.[12]

In the presence of a compiler for $S$ to $I$ as well as to $A$ (which has a definitional status for $S$ by assumption) the following correctness criterion emerges: (i) for all $x{:}s \in S_c \cap S_{c:int}$, and for all data files $\boldsymbol{y}$: $x{:}s \bullet\bullet_s \boldsymbol{y} = (u{:}s2i{:}e \bullet\bullet x{:}s) \bullet\bullet_{int4i} \boldsymbol{y}$ and (ii) $S_c \subseteq S_{c:int}$.

It is also possible that $S$ is primarily defined in terms of its projection to $I$. Then the second condition becomes $S_c \supseteq S_{c:int}$, thus expressing that the compiler which now lacks a definitional status can support all required processing.

### 7.3 Managed execution of compiled intermediate code

The intermediate code is managed (executed in a managed way) if it is both assembled and compiled. Thus an interpreter, $vm$ called a run-time (system)

[12] Interpreter based pseudo-empirical syntax for $S$.

$vm{:}rt4ae{:}e$ acts as an interpreter for a compiled version of the intermediate code. $rt4ae$ is the role of run-time for an almost executable code. This code is also called a virtual machine, which has been reflected in its mnemonic name. The intermediate code is compiled by $c{:}i2ae{:}e$ to a stage between the assembly level and the executable level denoted $AE$. This leads to the definition $x{:}i \bullet \bullet_{i:m} \boldsymbol{y} = vm{:}rt4ae{:}e \bullet \bullet (c{:}i2ae{:}e \bullet \bullet x{:}i) \frown \boldsymbol{y}$.

Managed code execution allows the virtual machine to perform activities like garbage collection in a pragmatic way, not too often and not too late, depending on statistics regarding the actual machine. Other activities may involve various forms of load balancing in a multi-processor system. Also various security checks may be incorporated in managed execution.

**JIT compiled intermediate code.** At this stage it is possible to indicate in informal terms what is actually supposed to happen in a modern computer system. Source codes are compiled to an intermediate code format. That code format is assumed to be widely known thus permitting execution on a large range of machines. Its execution involves managed execution after compilation. However, compilation is performed in a modular and fashion and the run-time system calls the compiler which then compiles only parts of the intermediate code when needed, throwing the resulting codes away when not immediately needed anymore, and recompiling again when needed again. This is called 'just in time compilation' (or JITting for short). The use of JIT compiler based managed execution has the following advantages:

(i) The accumulation of a machine specific (or rather a machine architecture specific) code base is avoided, in favor of a code base consisting of intermediate code which can be processed on many different machine architectures. Especially if the code is very big and during an execution only a small fraction of it is actually executed integral compilation before execution becomes unfeasible. Therefore mere code size implies that a decision to design an intermediate code for compilation forces one to opt for JIT compilation.

(ii) The advantages of managed execution are as mentioned above. Only managed execution can provide a high quality garbage collection service, because a naive approach where all garbage is removed immediately after its production (i.e. becoming garbage) is impractical and inefficient.

(iii) The advantages of (JIT) compilation in comparison to intermediate code interpretation are classical ones. Compiled code execution may be faster than code interpretation, and, perhaps more importantly, because the intermediate code will be (JIT) compiled, it may be more abstract and therefore shorter, more comprehensible and more amenable to maintenance than it would be were it to be interpreted.

## 8 Machine functions for JIT compilation

The formalization of JIT compilation requires code to consist of parts that can be independently compiled. The syntax of multi-part code below will admit

this aspect of formalization. For the various parts of multi-part code to work together intermediate results of a computation need to be maintained. That can be modeled by using machine functions that produce a state from a state rather than a code from one or more codes. State machine functions (also termed state transition functions) will be introduced to enable the formalization of the sequential operation of different program parts. Finally program parts may be executed several times during a computation in such a way that different entry points in the code are used. Natural numbers will be used as code entry points.

### 8.1 Multi-part code

Code can be extended to code families in various ways. Here we will consider so-called flat multi-part code. A flat multi-part code is a sequence of codes separated by colons such as for instance 10011:1110:0:0001. The collection of these codes is denoted with MPC (multi-part codes). It will now be assumed that the input of a machine function is a code family. The case dealt with up to now is the special case where multi-part codes have just one part (single part codes: SPC). Access to a part can be given via its rank number in the flat multipart code listing, starting with 1 for the first part. MPCN, (multi-part code notation) admits bit sequences and the associative operator ':'. Brackets are admitted in MPCN but do not constitute elements of the code families, for instance: (10:011):00 = 10:(011:00) = 10:011:00. When multi-part codes are used it is practical to modify the type of machine functions in order to admit multi-part inputs.

The $n$'th part of a multi-part code $x$ is denote $\mathtt{part}_n(x)$. This notation produces empty codes if $n$ is too large.

**Non-separate assembly and compilation.** An assembler or a compiler for multipart code that produces a single executable will be called a non-separate assembler or a non-separate compiler. To use a positive jargon a non-separate assembler or compiler may be called a gluing assembler or compiler. It is assumed that a gluing assembler/compiler detects itself which parts of a code must be used, for instance by looking at a code that contains some marker and using some form of cross referencing between the parts. Let MPA be a multipart assembly code notation. $c{:}mpa2e{:}e$ is an executable which transforms multi-part control codes to single-part executables. A gluing compiler can be used to define the meaning of multi-part control codes as follows:

$$x{:}mpa \bullet \bullet_{mpa} \boldsymbol{y} = (c{:}mpa2e{:}e \bullet \bullet x{:}mpa) \bullet \bullet \boldsymbol{y}.$$

Similarly for a notation $s$, $mps$ may be its multi-part extension. Then one may define its meaning given a non-separate compiler by

$$x{:}mps \bullet \bullet_{mps} \boldsymbol{y} = (c{:}mps2a{:}e \bullet \bullet x{:}mps) \bullet \bullet_a \boldsymbol{y}.$$

**Separate compilation.** Separate compilation holds for code families of some code notation if a compiler $\psi$ distributes over the code part separator : $\psi(x{:}y) = \psi(x){:}\psi(y)$). A compiler with this property is called separation modular. So it

may now be assumed that $mS$ (modular $S$) consists of sequences of $S$ separated by ':' and that $u{:}ms2e{:}e$ is a separation modular compiler for $mS$ i.e., using MPCN, $u{:}ms2e{:}e \bullet \bullet y{:}z = (u{:}ms2e{:}e \bullet \bullet y){:}(u{:}ms2e{:}e \bullet \bullet z)$ for all $y$ and $z$ which may themselves be code families recursively. For the second output component there is a similar equation: $u{:}ms2e{:}e \bullet \bullet^2 y{:}z = (u{:}ms2e{:}e \bullet \bullet^2 y){:}(u{:}ms2e{:}e \bullet \bullet^2 z)$; corresponding equations are assumed for the other output components.

### 8.2 State machine functions

Machine functions, as used above, transform sequences of (possibly multi-part) input files into output files. This is adequate as long as intermediate stages of a computation need not be taken into account. Modeling separate compilation typically calls for taking into account intermediate stages of a computation. In order to do so we will use state machine functions rather than machine functions. State machine functions provide a new state given a control code and a state. At the same time it becomes necessary to deal with code entry points. A code entry point is a natural number used to indicate where in (the bit sequence of) a code execution is supposed to start. Code entry points will arise if code has been produced via imperative programming using a program notation with subroutine calls and subsequent compilation and/or assembly. In addition a state machine function must produce information concerning which part of the multi-part code is to be executed next. It is now assumed that $ST$ denotes the collection of states of a machine model. A multipart part code state machine function has the following type:

$$(SPC \times CEP \times ST) \to ((ST \cup \{D, M\}) \times CPN \times CEP)$$

Here CPN is the collection of code part numbers and CEP is the the collection of code entry points. The CEP argument indicates where in the control code execution is supposed to start. These two numbers at the output side indicate the code part that must be executed in the next phase of the execution and the entry point within that code part which will be used to start the computation for the next phase. If the code part number is outside the range of parts of the MPC under execution (in particular if it equals 0), the computation terminates, and if the exit point is outside the range of exit points of a part termination will also result.

**Relating machine functions with state machine functions.** The connection with machine functions can be given if input files can be explicitly incorporated in the state (using a function 'in') and output functions can be extracted (by means of 'out'). State machine functions will be denoted with a single bullet. A CEP argument will be represented as superscript, a CPN argument may be given as another superscript, (preceding the CPN superscript), and other information may be given in subscripts. For single part code $x$ one may write:

$$x{:}e \bullet \bullet \boldsymbol{y} = \mathbf{out}(\mathrm{state}(x{:}e \bullet^1 \mathrm{in}(\boldsymbol{y}, s_0))).$$

Here **out** produces a sequence of outputs different from $M$, unless the computation diverges. This equation provides a definition that works under the assumptions that the computation starts in state $s_0$ and that at the end of the single part computation the next part result equals 0: i.e. for some $s'$ and $p'$, $x{:}e \bullet^1 (\mathrm{in}(\boldsymbol{y}, s_0))) = (s', 0, p')$. Moreover it must be noticed that in the notation used the separator in the code argument separates name parts for a single part code rather than code parts of a multi-part code.

**State machine functions for multi-part control code.** Assuming that multi-part code execution starts by default with the first code part at its first entry point the following pair of equations (start equation and progress equation) defines the machine function for multi-part executable code (tagged with *mpe*). Start equation:

$$x{:}mpe \bullet \bullet \boldsymbol{y} = \mathbf{out}(\mathrm{state}(x{:}mpe \bullet^{1,1} (\mathrm{in}(\boldsymbol{y}, s_0))))$$

expressing that execution starts with the first code part at entry point 1, and with the following progress equation:

$$\mathrm{part}_p(x{:}mpe) \bullet^q s = (p', q', s') \rightarrow x{:}mpe \bullet^{p,q} s = (s' \lhd p' = 0 \rhd (x{:}mpe \bullet^{q',p'} s'))$$

where the progress equation is valid under the assumption that the only returned code part number outside the code part range of $x{:}mpe$ equals 0.

### 8.3 Unmanaged JIT

The JIT equation for (almost) unmanaged[13] multi-part intermediate code execution connects the various ingredients to a semantic definition of JIT execution for a multi-part intermediate code $x{:}i$. The start equation reads

$$x{:}mpi \bullet \bullet_{jit} \boldsymbol{y} = \mathbf{out}(\mathrm{state}(x{:}mpi \bullet_{jit}^{1,1} (\mathrm{in}(\boldsymbol{y}, s_0))))$$

with the progress equation:

$$(c{:}i2e{:}e \bullet \bullet \mathrm{part}_p(x{:}mpi)) \bullet^q s = (p', q', s') \rightarrow$$
$$x{:}mpi \bullet_{jit}^{p,q} s = (s' \lhd p' = 0 \rhd (x{:}mpi \bullet_{jit}^{q',p'} s'))$$

For the progress equation it is assumed that a compiler/assembler $c{:}i2e{:}e$ for the intermediate code is available that translates it into executable code.

**Limited buffering of compiled parts.** A buffer with the most recent compilations (i.e. files of the form $c{:}i2e{:}e \bullet \bullet \mathrm{part}_p(x{:}i)$) of code parts can be maintained during the execution of the multi-part intermediate code. Ass this buffer has a bounded size, some code fragments will probably have to be deleted during a run[14] and may subsequently need to be recompiled. The idea is that a compilation is done only when absolutely needed, which justifies the phrase JIT.

[13] The only aspect of execution management is taking care of JIT compiling an intermediate code part when needed and starting its execution subsequently.

[14] Code garbage collection.

### 8.4 Managed (and interpreted) multi-part code execution

Managed code execution involves the interpretation of (JIT) compiled intermediate code. The start equation reads

$$x{:}mpi \bullet \bullet_{manjit} \boldsymbol{y} = \mathbf{out}(\mathrm{state}(x{:}mpi \bullet^{1,1}_{manjit} (\mathrm{in}(\boldsymbol{y}, s_0))))$$

The corresponding progress equation reads

$vm{:}rt4ae{:}e \bullet^q \mathrm{in}(c{:}i2ae{:}e \bullet \bullet \mathrm{part}_p(x{:}mpi), s) = (p', q', s') \rightarrow$

$$x{:}mpi \bullet^{p,q}_{manjit} s = (s' \lhd p' = 0 \rhd (x{:}mpi \bullet^{q',p'}_{manjit} s'))$$

The virtual machine $vm{:}rt4ae{:}e$ computes the intermediate state reached after execution has exited from code part $p$ after incorporating the code to be executed in the state. The interpreter takes into account that the code to be interpreted has to be started at entry state $q$.

**A requirement specification for managed JITting.** Provided a compiler $c{:}mpi2a{:}e$ is known the following identity must hold:

$$x{:}mpi \bullet \bullet_{\mathrm{manjit}} \boldsymbol{y} = (c{:}mpi2a{:}a \bullet \bullet x{:}mpi) \bullet \bullet_a \boldsymbol{y}.$$

This equation may be used alternatively as a correctness criterion for the definitions using managed JIT compiled execution. The compiler based definition will not capture any of the features of execution management but it may be very useful as a semantic characterization of the execution of multi-part intermediate code.

## 9 Verifying compilers

Recent work of Hoare (e.g. [4]) has triggered a renewed interest in the idea that program verification may be included in the tasks of a compiler. The notion of a verifying compiler has been advocated before by Floyd in 1967, [8], but it has not yet been made practical, and according to Hoare it might well be considered a unifying objective which may enforce systematic cooperation from many parts of the field. Getting verifying compilers used in practice at a large scale is certainly a hard task and Hoare refers to this objective as a 'grand challenge' in computing. Turning quantum computing into practice, resolving P versus NP, proofchecking major parts of mathematics and the theory of computing in the tradition of de Bruijn's type theory, removing worms and viruses from the internet, and (more pragmatically) the .NET strategy for using the same intermediate program notation for all purposes may be viewed as other grand challenges. In computational science the adequate simulation of protein unfolding stands out as a grand challenge; in system engineering the realization of the computing grid and the design of open source environments of industrial strength for all major application areas, constitute grand challenges as well.

Program verification has been advocated by Dijkstra (e.g. in EWD 303) because testing and debugging cannot provide adequate certainty. Indeed, if human certainty is to be obtained proofs may be unavoidable. For the design of complex systems, however, the need for proofs is harder to establish. The biological evolution of the human mind has produced a complex system, at least from the perspective of current computer technology, through a design process using natural selection (which seems to be a form of testing rather than a form of verification) as its major methodology for the avoidance of design errors. This might suggest a way around program verification as well: rather than verifying program $X$, one may design a far more complex system $C$ using $X$ and many variations of it which is then tested. A test of $X$ may involve millions of runs of $X$ and its variations. Usability of $C$, as demonstrated using tests, will provide confidence in components like $X$, as well as the possibility to use these components in a complex setting. Whether confidence or certainty is what people expect from computing systems is not an obvious issue, however. As a consequence this particular grand challenge has a subjective aspect regarding the value of its objective that the other examples mentioned above seem not to have.

In this section the phenomenon of a verifying compiler will be discussed at the level of (state) machine functions and control code algebra.

### 9.1 Requirement codes and satisfaction predicates

REQ will represent a collection of predicates on behaviors that will viewed as descriptions of properties of codes under execution. $r \in$ REC may serve a s a requirement on a behavior. A given predicate $\underline{sat}_{req}$ ($B\,\underline{sat}_{req}r$, for behavior $B$ and requirement $r \in$ REQ) determines the meaning of requirements. Having available the satisfaction predicate at the level of behaviors, it may be gradually lifted to source codes.

If $B$ is the behavior of executable code $x$ on machine $m$ ($B = |x|_{\bullet\bullet^m}$), then $x\,\underline{sat}_{m:e}r$ if $B\,\underline{sat}_{req}r$. Further for an assembly code $x{:}a$ one may write $x{:}a\,\underline{sat}_{m:a}r$ if $(c{:}a2e{:}e \bullet \bullet^m x{:}a)\,\underline{sat}_{m:e}r$, given an assembler $c{:}a2e{:}e$ for $A$.
For a high-level and machine independent program notation $s$ we define $x{:}s\,\underline{sat}_s r$ if $(v{:}s2a{:}e \bullet \bullet^m x{:}s)\,\underline{sat}_{m:a}r$ for a machine $m$ and a compiler $v{:}s2a{:}e$ on the same machine $m$. In the sequel the machine superscripts will be omitted, because machine dependence is not taken into account.

### 9.2 Proofs and proof checking

Given $r$ it may be viewed a design task to find a control code $x{:}s$ such that $x{:}s\,\underline{sat}_s r$. A proof for this fact is a code $p$ in a collection PROOFS of codes such that $p \vdash x{:}s\,\underline{sat}_s r$, where $\vdash$ is a relation such that $p \vdash x{:}s\,\underline{sat}_s r$ implies that $x{:}s\,\underline{sat}_s r$. This implication represents the so-called soundness of the proof method. Validating $\vdash$ requires a proof checking tool $w{:}ch4p4s{:}e$ (check for being a proof for an $s$ code) such that $w{:}ch4p4s{:}e \bullet \bullet p, x{:}s, r$ always terminates, never

produces an error, and produces the code "0" exactly if the code $p$ constitutes a proof of $x{:}s\ \underline{sat}_s r$.

**Non-automatic proof generation.** Proving the correctness of control codes by hand, as a control code production strategy, amounts to the production of a pair $(x, p)$ such that $w{:}ch4p4s{:}e\bullet\bullet p, x, r =$ "0". Here it is taken for granted that if there is to be any chance of obtaining a proof for the control code that code may have to be specifically designed with the objective to deliver the required proof in mind. It seems to be a folk-lore fact in system verification practice that hand made proofs can only be given if the code to be verified is somehow prepared for that purpose, though in theory that need not be the case. A complete proof method guarantees the existence of a proof for each required fact (concerning a code that satisfies a requirement) . But actually finding a proof which theory predicts to exist is by no means an easy matter.

**Infeasibility of proof search.** The task to prove code correctness manually (and have it machine checked thereafter) may be considered an unrealistic burden for the programmer. Therefore control code production methods are needed which can take this task out of the hands of human designers. The most obvious strategy is automated proof search.

Having available a proof method and a proof checker, one may design an automated proof generator as a code $u{:}pg4s{:}e$, such that $u{:}pg4s{:}e\bullet\bullet x{:}s, r$ produces a proof $p$ with $p \vdash x{:}s\ \underline{sat}_s r$ if such a proof exists with the computation diverging otherwise (or preferably producing a negative answer, i.e. a code outside REQ). Even if a proof checker exists in theory its realization as an executable code with useful performance seems to be elusive. Therefore this strategy to avoid the need for people to design proofs has been considered infeasible and a different strategy is needed.

### 9.3 Verifying compilers

Given the high-level control code notation $s$ a version $as$ of it that admits annotations is designed. A code in $as$ is considered an annotated code for $s$. Using a tool $strip : as2s : e$ the annotations can be removed and a code in $s$ is found which determines the meaning of an annotated $s$ code. In other words: $x{:}as \bullet\bullet_{as} \boldsymbol{y} = (strip{:}as2s{:}e \bullet\bullet x{:}as) \bullet\bullet_s \boldsymbol{y}$.

A requirement $r$ may will be included as a part of the annotation. The requirement can be obtained (extracted) from the annotated code by means of the application of a tool $u{:}as2r{:}e$. The computation $u{:}as2r{:}e\bullet\bullet x{:}as$ either produces a requirement $r \in$ REQ or it produces an error (M for meaningless). If a requirement is produced that implies that as a part of the extraction the computation has succeeded in generating a proof for the asserted program and checking that proof. Thus in this case it is guaranteed that $strip{:}as2s{:}e \bullet\bullet x{:}as\ \underline{sat}_s (u{:}as2r{:}e \bullet\bullet x{:}as)$.

The production of adequately annotated control code may be simpler than the production of proofs as it allows the designer to distribute requirements over

the code thus simplifying the proofs that the extraction tool needs to find by itself. In addition it is held by some that for each construction of a source code a 'natural' assertion may be found, which the conscious control code author should have in mind. This allows to provide a canonical annotation for any source code, which can be proved without much trouble. Then follows a stage of logic that helps to derive the 'required' requirement from a requirement that has been automatically synthesized from the canonical annotation. This piece of logic needs careful attention, because by itself it is as unfeasible as P$\neq$ NP. The asserted code needs a good deal of guidance cast in the form of annotations for this matter.

**Verifying compilers and multi-part code.** Large amounts of code will be organized as multi-part codes, admitting JIT compilation strategies. A computation in this setting uses the JIT compiler to transform part after part in an (almost) executable form. It may be assumed that verifying compilers are used to obtain the various code parts in the code base at hand. It is reasonable to assume that these requirements have been stored in a database annex to the code base. In that context it is preferable to refer to the stored requirements as specifications. Indeed, because the JIT compiler using the various parts must rely on the use of valid code it cannot accept the possibility that a code part defeats the extraction of a requirement.

Now complimentary to JIT compilation there is JIT requirements integration, a process that allows the computation to dynamically synthesize a requirement from the requirements known to be satisfied by the various parts it uses. The JIT compiler should combine requirements synthesis and verification. This is a difficult task that by itself leads to different solution architectures.

**Trivial requirement synthesis JIT compilation.** A straightforward way to obtain the requirements integration functionality is to assume that all specifications for parts used during a computation are identical (which is plausible if these are understood as global system invariants), thereby shifting the engineering burden to the production of the code base. This solution architecture will be called the trivial requirements synthesis JIT architecture (TRS-JIT).

For TRS-JIT it becomes helpful to admit storing a collection of requirements for the various code parts in the annex database. It should be noticed, however, that the previous discussion of verifying compilers provides no technique for finding a multitude of validated requirements for the same (compiled) code. In addition, as the code base contains code parts with different requirements (stored in the annex data base), the JIT compiler now faces the possibility of running into a code part that is not equipped with a suitable requirement. It may be the case that the requirement for the part is logically stronger than needed, but that is impossible to check dynamically. Thus it must be accepted that executions may always stop in an error condition indicating that a part had to be JIT compiled for which the needed requirement was absent. This is a severe drawback because it is useless for real time control applications. If this drawback is unacceptable

the JIT compiler must be shown always to ask for a code that exists in the code base and that is equipped with the required specification.

**A verifying JIT compiler.** Guaranteeing that a computation will not halt at an ill-specified of even absent code part is the task of the verifying JIT compiler in the case of the trivial requirements synthesis architecture. This leads to a two-phase strategy that first checks this latter property using a dataflow analysis on the initial code, and thereafter a check that all needed parts are equipped with the required specification. In this stage a limited amount of proof generation may be used to allow parts that have logically stronger specifications to be used if the required specifications can be derived by these limited means.

## 10 Conclusion

Machine functions have been used to formalize several software processing mechanisms at a high level of abstraction, by means of the formation of code algebras. This abstract formalization has been proposed in particular for compilation, assemblage, interpretation, and for managed and unmanaged, interpreted and just in time compiled multi-part code execution, and for verifying compilers. While the notion of a verifying compiler can be captured to some extent at the abstraction level of CCA, this seems not to be the case for the unavidable concept of a verifying JIT compiler, however.

## 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 sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
3. J.A. Bergstra and S.F.M. van Vlijmen. *Theoretische Software-Engineering*. ZENO-Institute, Leiden, Utrecht, The Netherlands, 1998. In Dutch.
4. C.A.Hoare. The verifying compiler, a grand challenge for computer research. *JACM*, 50(1):63–69, 2003.
5. H.Bratman. An alternate form of the UNCOL diagram. *CACM*, 4(3):142, 1961.
6. J.Earley and H.Sturgis. A formalism for translator interactions. *CACM*, 13(10):607–617, 1970.
7. M.I.Halpern. Machine independence: Its technology and economics. *CACM*, 8(12):782–785, 1965.
8. R.W.Floyd. Assigning meanings to programs. *Proc. Amer. Soc. Symp. Appl. Math.*, 19:19–31, 1967.
9. A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. Ser 2*, 42,43:230–265,544–564, 1936.