Preprint of: Amnon H Eden, Raymond Turner. "Problems in the Ontology of Computer Programs." *Applied Ontology*, Vol. 2, No. 1 (2007), pp. 13–36. Amsterdam: IOS Press. Amsterdam, The Netherlands: IOS Press.

Also appeared as: Technical report CSM-461 (Sep. 2006), Department of Computer Science, University of Essex, ISSN 1744-8050.

PROBLEMS IN THE ONTOLOGY OF COMPUTER PROGRAMS

Amnon H. Eden ([1])

Department of Computer Science
University of Essex, UK
and
Center For Inquiry, Amherst, NY, USA

Raymond Turner

Department of Computer Science
University of Essex, UK

Abstract. As a first step in the larger project of charting the ontology of computer programs, we pose three central questions:

(1) *Can programs, hardware, and metaprograms be organized into a meaningful taxonomy?*
(2) *To what ontology are computer programs committed?*
(3) *What explains the proliferation of programming languages and how do they come about?*

Taking the complementary perspectives software engineering and mathematical logic, we take inventory of programs and related objects and conclude that the notions of *abstraction* and *concretization* take a central role in this investigation.

Key words: Philosophy of computer science, software ontology, science of software design.

Related terms: Mathematical logic, finite model theory, programming languages.

([1]) Corresponding author, address: Department of Computer Science, University of Essex Colchester, Essex CO4 3SQ, United Kingdom, phone: +44 (1206) 872677, fax: +44 (1206) 872788

1 Introduction

We take ontology to be that line of philosophical inquiry which seeks to answer the question *What exists?* We follow Quine's position according to which what exists is what science demands to exist. According to Smith [18],

> *Ontology seeks to provide a definitive and exhaustive classification of entities ... including the types of relations by which entities are tied together.*

As a first step in the larger project of charting the ontology of computer programs, we take inventory of the kinds of objects and relations that, prima facie, represent the ontology to which computer science is committed. At the focus of our investigation are computer programs, which can be broadly divided into programs qua static, linguistic types (scripts) and programs qua dynamic processes. We thus mark *Programs* as that category of entities which are referred to as computer programs by most computer scientists and laymen alike, divided as follows:

Program-Scripts A-temporal entities which consist of well-formed instructions to a given class of digital computing machines, commonly represented as inscriptions or text files.

Program-Processes Temporal entities that are created by a process of executing (running) a particular program-script in a particular physical setting, also known as operating system processes or 'threads'.

This description should only be taken as an intuitive classification of programs in the vernacular, serving as a starting point for our inquiry. Program-scripts and program-processes are examined in §2 and the respective categories are refined in the course of our investigation.

General ontology dates back to Aristotle's *Metaphysics* and to Kant's categories, but little has been written on the ontology of computer programs (although consider [18] and [1]). In line with general philosophy, we take the ontological investigation of computer programs to be an exercise in 'conceptual engineering' which seeks to provide answers to three central questions. We take the formulation of these questions and the examination of possible answers to be the central contribution of this paper.

The first question we pose is:

Q1 *Can programs, metaprograms, and hardware entities be organized into a meaningful taxonomy? Can well-defined differentiae between these categories be offered? In particular, can we distinguish between what is a program and what is not a program?*

We take a taxonomy to be a system of classification that can be represented as a mathematical graph (or possibly a tree), as demonstrated in Figure 1, each node is distinguished by using differentia (a distinction criterion) between two or more categories of entities. Some taxonomies begin with a top-level ontology, a list of upper-level categories offering an exhaustive partitioning of all entities under consideration, such that each top-level category is gradually refined into more specific subcategories.

As criteria of acceptance of an ontological taxonomy, we offer the following desiderata:

- *Unified*: The taxonomy should include a top-level ontology
- *Exhaustive*: The taxonomy should account for computer programs of any kind
- *Definitive*: Boundaries between categories should be demarcated by well-defined differentiae
- *Uniform*: The taxonomy should be committed to a minimal number of differentiae, all of which are expressed in one language and derived from a single notion

Our line of investigation shall begin with *abstraction* as a criterion of distinction.

The second question we pose can be expressed as follows:

Q2 *To what ontology are computer programs committed?*

Quine maintains that ontological commitments are simple to observe if a theory is formulated in the canonical representation of the predicate calculus:

> [A] *theory is committed to those and only those entities to which the bound variables of the theory must be capable of referring in order that the affirmations made in the theory be true. ... "To be is to be the value of a* [bound] *variable."* ([23] pp. 13–15)

This motivates our choice of the classical predicate calculus and finite model theory for making ontological commitments explicit.

We distinguish our line of investigation from ontological engineering [18], the process of encoding information for the purpose of knowledge extraction, reasoning, planning, and decision-support. In information technology jargon, ontology is synonymous with the attempt to harness software for the axiomatization and encoding of particular domains of human knowledge (such as common-sense [www.cyc.com] and anatomy [19]) as knowledge bases or data bases, thereby making it possible for example to discover inconsistencies in existing representations. Ontological engineering shall remain outside our scope.

The popularity of ontological engineering has led to the abundance of 'ontologies', some of which were not measured against any particular criterion of acceptance [7]. In contrast, Smith [18] demands that not any collection of 'objects and relations' is by itself an adequate answer to Q2 ("the ontologist's credo"). Quine offers the following acceptance criterion:

> *Our acceptance of an ontology is ... similar in principle to our acceptance of a scientific theory, say a system of physics: We adopt ... the simplest conceptual scheme into which disordered fragments of raw experience can be fitted and arranged.* ([23] p. 16)

The ontology to which computer programs are committed is thus expected to serve as a unifying conceptual scheme, taking the role that the periodic table of elements plays in chemistry and elementary particles play in physics. Such an ontology must offer elementary (primitive) objects which cannot be reduced within the discipline and which provide adequate building-blocks for the ontology of the program.

The investigation of the ontological commitments of programs led us to examine the role of programming languages. Observing that computer programs can be encoded in so many such languages, we pose a third question:

Q3 *Given that all turing-complete programming languages are computationally equivalent, what explains their proliferation? How do new languages come about?*

We shall examine the process of language synthesis (§4) and the process of *concretizing* a specific subset of metaprograms as an explanation to the formation and proliferation of programming languages.

Outline

The remainder of this paper is devoted to the investigation into possible answers to the there questions we posed. In Section 2 we explore the notion of abstraction and examine a taxonomy of programs and related entities, the program abstractions taxonomy. In Section 3 we examine the nature of the differentiae between the categories in the program abstractions taxonomy. In Section 4 we examine *concretization*, the process of synthesizing program-scripts, program-processes, hardware, and languages from more abstract entities. In Section 5 we suggest that programming paradigms furnish those ontologies to which computer programs are committed. In Section 6 we summarize and briefly discuss the answers offered to the questions we posed.

2 A top-level ontology

As a first step towards a taxonomy we seek to distinguish programs from entities that are not programs, such as digital computing machines (hardware) and descriptions of programs (metaprograms). Our top-level ontology thus consists of three categories which can be roughly characterized as follows:

Metaprograms contains statements describing programs, such as algorithms, abstract automata, and software design specifications.

Programs is divided into *Program-Scripts* and *Program-Processes*.

Hardware contains digital computing machines, such as computers belonging to the Intel 8086 microprocessor family.

We shall refer to this table of categories and their refinements as the program abstractions taxonomy, depicted in Figure 1.

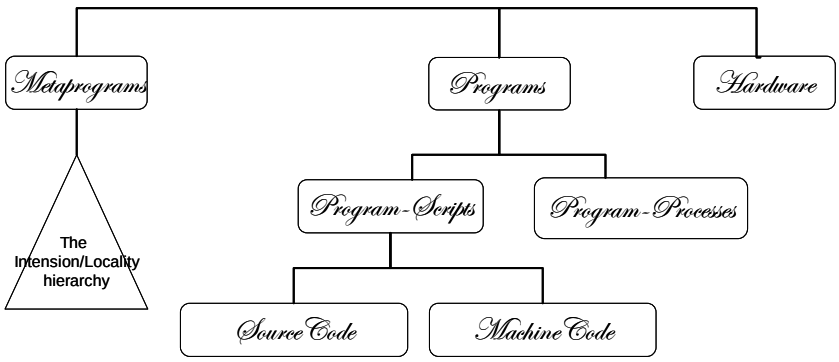

Figure 1. The program abstractions taxonomy

2.1 Abstraction

Among the arguments that computer programs pose unique philosophical questions is the observation that programs bridge between the abstract (e.g. turing automata) and the concrete (desktop computers, microwave ovens). Therefore, we are led to explore in detail the notion of *abstraction*.

***Ontology** (n). An account of being in the abstract.* (Oxford English Dictionary 1721)

We contend that the criteria of distinction between all categories in the program abstractions taxonomy can be formulated as interpretations of *abstraction*. We take **abstract** to mean any combination of the following interpretations:

A-I *Intangible*, namely not concrete, physical entities. Examples for intangible entities are a turing automaton and the number *1*.

A-II *Generalized*, as a category of entities. For example, *client-server software systems* and *vehicle* are more abstract than Microsoft Outlook and my bicycle.

A-III *Underspecified*, such as a logic statement that contains free variables. For example, the statement *men are mortal* is more abstract than the statement *Socrates is mortal*.

A-IV *Immanently meaningful to humans*, in the sense such as in the following sentence: "The Pascal script in Table 2 is more abstract than the equivalent Intel 8086 script in Table 1."

A-V *It from bit* [Floridi 2004], namely instances of *information*.

A-VI *A-temporal*, in the sense that a-temporal entities are 'timeless' whereas temporal entities extend in time.

The precise meaning of each one of these interpretations will become clear with the application of each as a differentia (§3). In the remainder of this section, we examine some of the categories that constitute the program abstractions taxonomy.

2.2 Programs

In this subsection we examine the kinds of programs and the notions of programming languages, program-scripts, and program-processes. Because program-scripts and program-processes are so intimately connected, they are both referred to as 'programs' in the vernacular. Our first task is therefore to distinguish between the two senses.

Program-scripts

Most programs are encoded as sequences of characters. Whether the sequence of characters s is a well-formed computer program depends on the grammar and semantics of the programming language in which it is encoded. This leads us to suggest the following definition of program-scripts:

$\mathscr{Program\text{-}Scripts}_{\text{DEF-1}}$ The category of entities s ("s is a program") for which there exists a programming language $\mathbb{L}$ such that s is a well-formed expression in $\mathbb{L}$.

This definition raises a difficulty since for every possible sequence of characters s, a programming language can in principle be tailored to allow s. This definition thus effectively admits any possible sequence of characters. This leads us to consider compilers as a criterion of acceptance:

$\mathscr{Program\text{-}Scripts}_{\text{DEF-2}}$ The category of entities s ("s is a program") for which there exists a compiler $C_{\mathbb{L}}$ such that $C_{\mathbb{L}}$ accepts s.

However this definition raises another difficulty. It implies that scripts depend on the existence of compilers, which are commercial and market-driven artefacts, the existence (or disappearance of which) is subject to market forces that are immaterial to ontological investigation. Moreover, scripts encoded in machine language do not require the notion of a compiler. We conclude that the very notion of a script is contingent upon a context of a programming language. An adequate criterion of acceptability requires us to explicitly formulate the programming language in which s is encoded. Therefore we may revise our definition as follows:

Program-Scripts$_{\text{DEF-3}}$ The category of entities $s_{\mathbb{L}}$ ("$s_{\mathbb{L}}$ is a program") such that $\mathbb{L}$ is a programming language and s is a well-formed expression in $\mathbb{L}$.

This definition raises a difficulty concerning the notion of a programming language. While it admits Pascal [26] and Lisp [17] as programming languages, we also wish to exclude formatting languages such as HTML and even more impoverished languages should not be accepted as programming languages in this context, such as the language of cooking recipes and similar 'programs'. We may therefore try and speak of sequences of instructions to a machine. But while the question *What is a programming language?* can be reduced to the question *What is a computer?*, the problem remains that any physical object can be described as a computer in some trivial sense ([2]). For example, a light switch can be taken to be a computer, the 'programming language' of which consists of any word (sequences of characters) in the alphabet consisting of the set $\{\mathrm{ON}, \mathrm{OFF}\}$. Naturally, such a definition empties the notion of a program-script from any content since any physical object is a 'computer' in some impoverished sense. What is necessary than is some formulation of the syntactic notion of a *programming language* as the set of well-formed sequences of instructions to a non-trivial machine.

Fortunately, we are provided with the notion of turing-completeness (e.g., [15]), which requires a programming language to support a non-trivial set of instructions. Turing-completeness is a property which ensures that the notion of a programming language is sophisticated enough to warrant independent ontological standing for computer programs. This observation leads us to a final revision of our definition:

Program-Scripts$_{\text{DEF-4}}$ The category of entities $s_{\mathbb{L}}$ ("$s_{\mathbb{L}}$ is a program") such that $\mathbb{L}$ is a turing-complete programming language and s is a well-formed expression in $\mathbb{L}$.

Program-processes

The notion of a program-process corresponds to the notion of a 'process' or a thread (also 'task', 'bot') as recognized by current operating systems. For example, both the Linux and the Windows XP operating systems refer to program-processes as 'processes', lists of which are demonstrated in Figure 2. Each program-process is a temporal entity generated by 'executing' or 'running' a particular program-script encoded in the language of the machine that was used to execute it. When generated, the instructions in the executed program-script are copied into a uniquely allocated segment(s) in the computer's memory (where the program-process holds instructions and data), called the process' 'image' (listed as 'image name' in Figure 2a). From that point on, the process proceeds with the execution of the instructions in its image.

Each particular program-script can be used to generate many simultaneous program-processes; for example, the list of Windows XP program-processes in Figure 2a includes two program-processes generated from the program-script recorded by the name `svchost.exe`.

([2]) This is the main thesis of pancomputation [16].

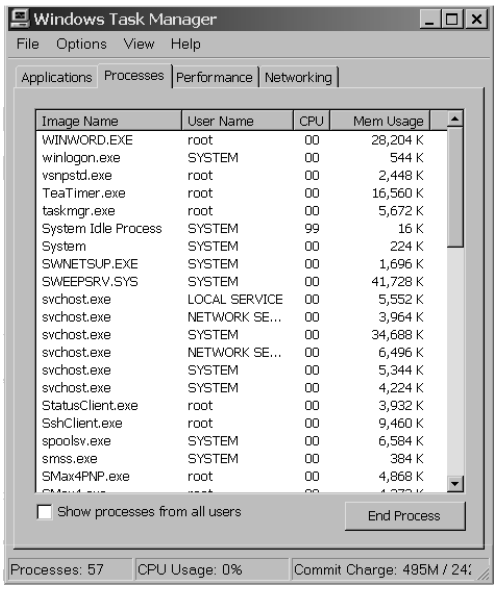

Figure 2a. Windows XP program-processes

```
% ps -ag
  PID TTY      TIME COMMAND
 3695 tty2     0:00 /sbin/mingetty tty2
 3696 tty3     0:00 /sbin/mingetty tty3
 3697 tty4     0:00 /sbin/mingetty tty4
 3700 tty5     0:00 /sbin/mingetty tty5
 3704 tty6     0:00 /sbin/mingetty tty6
12274 pts/2    0:00 -csh
12365 pts/3    0:00 -csh
 6435 pts/0    0:00 -csh
 8058 pts/1    0:00 -csh
 8145 pts/5    0:00 -csh
 8202 pts/5    0:00 ps –ag
```

Figure 2b. Linux 2.6.16 program-processes

The category *Program-Processes* includes simple processes, such as the process generated by pressing a button on a computer keyboard or in a microwave oven, the processes generated by executing the program-scripts in Table 1 and Table 2, as well as arbitrarily complex programs such as those involved in passing, processing, and returning a query for an Internet search engine ([3]).

The first electronic computers were designed to generate and carry out the instructions of one program-process at a time. Contemporary operating systems allow a machine to give the appearance of many program-processes executing concurrently (although in fact at each clock cycle only one program-process is using the CPU.) A program-process is uniquely identified by the operating system by an entry in the list of current processes that the operating system recognizes at each clock cycle. For example, the list of Linux program-processes in Figure 2b enumerates the process id (`PID`), a unique identifier assigned to each program-process.

Increasingly sophisticated hardware, distributed processing, computer networks, and open systems in particular blur the boundaries between one program-process and another. For the purpose of this exploratory investigation, it suffices to identify each program-process with a single execution thread that occupies a CPU at each point in time.

2.3 Hardware

Our enquiry in the notion of programming languages ([4]) led us to examine the notion of digital computing machines. These can be assembled by a variety of technologies, the precise nature of which is immaterial to this discussion. What is relevant is that computing machines belong to a wide spectrum of simple and arbitrarily complex machines and that an ontological enquiry must establish whether we take *Hardware* to include light switches and abaci (definitely not), simple desk calculators (proba-

([3]) in §3.2 we examine compilers and in §4 the notion of *compilation*.

([4]) For a discussion in the ontology of programming languages see [22].

bly not), microwave ovens (unlikely), microprocessors embedded in microwave ovens (yes), and Babbage's difference engine (maybe). The increasing pervasiveness of microprocessors in modern life, including vehicles, television sets, and other gadgets, demands further examination of this category.

Turing's characterization of abstract automata offers a possible criterion of acceptance. It leads us to the following formulation to the notion of a Hardware entity:

Hardware$_{\text{DEF-1}}$ The category of computing machines whose behaviour can be modelled by a turing automaton.

Unfortunately, this definition accepts light switches and abaci since the behaviour of these entities can be captured by (trivial) turing automata. Another difficulty arises from the fact that a turing machine is an abstract mathematical construct that includes an infinite tape which cannot by realized in any finite physical system, whereas in reality every computer—no matter how extensible—has access at any point in time to a finite amount of memory. Furthermore, the modern notion of a digital computing machine is not restricted to those machines whose set of instructions constitute computer programs, a notion we have established by definition *Program-Scripts*$_{\text{DEF-4}}$. This requires us to revise our definition to include only those machines that are capable, at least in principle, of executing program-scripts encoded in turing-complete programming languages. This leads us to revise *Hardware*$_{\text{DEF-1}}$ as follows:

Hardware$_{\text{DEF-2}}$ The category of computing machines whose behaviour can carry the same class of computations as a universal turing automaton.

We also observe that a *Hardware* entity can be embedded in an entity that does not belong to that category. Thus, the microprocessor embedded in a microwave oven is taken to be an entity in *Hardware* but a microwave that does not offer a turing-complete set of instructions (a microwave that cannot be "programmed") is not. This should not pose us with a difficulty since the whole can be less computationally powerful than its parts.

2.4 Metaprograms

We mark *Metaprograms* as that category of statements describing programs. Such statements include informal descriptions, such as the statement "*The class of programs which take no input, terminate, and whose output is* `"hello world!"` (§3.4) , and formal specifications, such as the function $factorial$ (§4). The software engineering and formal methods literature generally refers to metaprograms as software **specifications**. For example, software design specifications consist of constraints imposed on the structure or behaviour of programs, such as software metrics, and in particular descriptions in the literature on software design, including architectural styles [10], design patterns [9], and abstract data types [2].

A definition capturing *Metaprograms* remains outside the scope of this paper. Instead, we restrict our discussion to descriptions formulated in the classical predicate calculus. In (§3.4) we examine two sample metaprograms and illustrate how each metaprogram can be taken to represent a category of programs.

3 Differentiae

We contend that the interpretations of *abstraction* listed in §2.1 are the differentiae between the categories and subcategories in the program abstractions taxonomy (Figure 1). In this section we examine these differentiae in detail and use them to distinguish between the categories in this taxonomy.

3.1 Program-scripts vs. program-processes

We distinguished program-scripts from program-processes as two subcategories of *Programs*. Since program-scripts are a-temporal and program-processes are temporal entities, the differentia between the categories *Program-Scripts* and *Program-Processes* is interpretation A-VI of *abstraction*. In addition, in §4 we examine how a program-process is generated from a program-script and demonstrate the process of program process-concretization which synthesizes a machine and time-specific program-process from a program-script (a process of *concretization* by interpretation A-III). Since each program-process is generated from some program-script, the differentia between *Program-Scripts* and *Program-Processes* is interpretation A-III of *abstraction* (underspecified vs. specific).

3.2 Source code vs. machine code

We observe two categories of program-scripts: scripts encoded in a machine language, the category of which is generally referred to as *Machine Code*, and scripts encoded in a high-level programming language, the category of which is generally referred to as *Source Code*. Using concrete examples from each category, we demonstrate below that source code entities are more abstract than machine code entities by interpretations A-II (generalized) and A-IV (immanently meaningful to humans).

Historically, *machine languages* (also *assembly languages* in the jargon) came earlier into being. A script encoded in a machine language consists of sequences of instructions that are directly interpreted by a class of digital computing machines. The class of machines for which a particular machine code script is meaningful can be abstracted using the notion of a machine language. Common machine-code instructions include assigning a particular number into a register (`write`), recalling the value assigned to a register (`read`), and adding the value assigned to one register to another (`add`). For example, Table 1 depicts a sequence of instructions which, when executed on a microprocessor from the Intel 8086 family, prints the string "`Hello, World!`" to the console.

Table 1. Program-script "Hello, World!" in machine code for Intel microprocessor 8086 ([5,6]).

```
C6 06 02 00 68
C6 06 04 00 65
C6 06 06 00 6C
C6 06 08 00 6C
C6 06 0A 00 6F
C6 06 0C 00 2C
C6 06 0E 00 77
C6 06 10 00 6F
C6 06 12 00 72
C6 06 14 00 6C
C6 06 16 00 64
C6 06 18 00 21
```

Rapid technological developments led to gradual improvements in the processing power of digital computers by several orders of magnitude, which allowed the introduction of increasingly more powerful machine languages. This improvement led during the 1950s to the development of compilers, ushering in the next phase in the history of programming languages. A **compiler** is a program-process which translates source code into machine code. We say that compilers *concretize* (§4) a source code into machine code. For example, a Pascal compiler for the Intel 8086 microprocessor family is likely to translate the source code depicted in Table 2 into the machine code depicted in Table 1.

Table 2. Program-script "Hello, World!" in Pascal.

```
program hello(output);
begin
  write('Hello, world!')
end.
```

Since Fortran and Lisp compilers were synthesized during the early 1950a, many different compilers were introduced for other high-level programming languages, including general-purpose languages such as Pascal and Java, Web scripting languages such as JavaScript, and protocols of computer networks a such as TCP/IP.

As a result from the variability of machine languages and from the ambiguity of the grammar and semantics of high-level programming language, the precise mapping from Pascal into the Intel 8086 microprocessor family varies with the target machine's language, the commercial compiler vendor, and with the release of the compiler. For example, the C programming language defines the type `int` as "implementation dependent". This allows vendors of C compilers for a 16-bit microprocessor to represent integers in 2 bytes and vendors offering compiler for 32-bit microprocessors to represent integers in 4 bytes. Such differences yield different interpretations of the same program with visibly different behaviour. In conclusion, the mapping from source code into machine code is a one-to-many relation. This demonstrates that program-scripts in each high-level (compiled) programming language

([5]) Not a complete 8086 program but the relevant extract thereof.

([6]) Produced using EMU8086, an Intel microprocessor emulator. EMU8086 is a trade mark of EMU8086.com.

represents a category of program-scripts in machine code, and that *SourceCode* is more abstract than *MachineCode* by interpretation A-II of abstraction.

Each programming languages was designed to support specific set of abstraction mechanisms, such as procedures, recursion, records, classes, and modules. But these are conservative extensions of machine languages; in other words, despite the differences between them, machine and high-level programming languages are computationally equivalent (turing-complete). Instead, programming languages that support these abstraction mechanisms equip the human programmer with better tools for conceptualizing the program. Compare for example the machine-code script in Table 1 with the Pascal script in Table 2: understanding, writing, and debugging source code is easier than understanding, writing, and debugging machine code. Consider for example the task of changing the output of the script in Table 1 into "Goodbye, World!" ([7]). This demonstrates that source code is more immanently meaningful for humans than machine code and that *SourceCode* is more abstract than *MachineCode* also by interpretation A-IV of abstraction.

3.3 Hardware vs. programs

The differentia between the categories *Programs* and *Hardware* is interpretation A-I (intangible) of *abstraction* because Hardware entities are tangible whereas programs are not. Vacuum tubes, printed circuits, microprocessors, random-access memory, I/O ports, and all related circuitry are physical, material entities which occupy specific regions of time and space. Programs do not satisfy any of these criteria.

The second differentia between *Program-Scripts* and *Hardware* is interpretation A-V (it-from-bit) of abstraction. John Wheeler [24], the eminent evangelist of information theory ([8]), defines an entity x as an instance of information if every element of x derives its function, its meaning, its very existence entirely from the apparatus-elicited answers to yes-or-no questions (binary choices). In other words, x can be reduced to bits (hence 'it from bit'). This is evidently true for every program-script and every program-process: each program, no matter how complex, has a 'bottom', its complete representation derives from a finite, discrete set of 'binary choices'. ([9])

3.4 Metaprograms vs. Programs

We take *Metaprograms* to consist of formulas in the predicate calculus, each of which offers a description of a category of programs, such as the following:

The class of programs which take no input, terminate, and whose output is "`hello world!`" **HW**

The class of programs whose computational complexity is NP-complete **NPC**

A metaprogram can be specified as a requirement from a program. In software engineering jargon, we say that, for example, HW specifies the requirements from the program-scripts "Hello, World!" in Intel 808X (Table 1) and in Pascal (Table 2).

Metaprograms includes statements in functional specification languages [21], such as the function $factorial$ in §4, Z and VDM, which formulate (the behaviour of) programs as mappings from input

([7]) More generally, it is widely accepted that the ongoing 'software crisis' [11] is the result of lack of abstraction mechanisms in contemporary programming language.

([8]) who also coined the term 'black hole'.

([9]) Although Wheeler conjectured that every physical object is also an instance of information, this conjecture is yet to be corroborated.

into output. Software metrics and software design statements such as design principles, architectural styles [10], design patterns [9], and abstract data types are generally represented in a wide range of specification languages, including formal, semi-formal, and informal, textual and visual languages. Turner [21] furnishes a complete theory in logic which describes how functional specifications can be articulated as relations in the predicate calculus. The formulation of metaprograms in the classical predicate calculus is demonstrated below. Further examples can be found in [4].

We contend that the differentia between the categories *Metaprograms* and *Programs* is interpretation A-II of *abstraction* (category vs. elements thereof). To corroborate this claim, the remainder of this section is dedicated to demonstrate how each metaprogram can be taken to represent a category of program-scripts. This demonstration uses the first-order predicate calculus and finite model theory; it can be skipped without affecting the readability of the remainder of our discussion.

Information hiding

Information hiding [25], also referred to as *data abstraction* or *encapsulation*, is a software design principle that is supported (in one variation or another) by every object-based, object-oriented, and class-based programming language [3]. This principle is intended to make programs more maintainable by enforcing that all dependencies between modules of the program are made explicit, thereby minimizing the 'domino effect' ([10]).

We shall restrict our discussion to programming languages containing an explicit representation of modules (in Java: *classes*) and procedures (in Java: *methods*), such that each module consists of a collection of procedures (also: "procedure p is a *member* of module m"). Let $\mathbb{M} \subset$ *Program-Scripts* designate this set of program-scripts written in a range of modular programming languages that fall under this category. An example for such a program-script is the Java program `Stack-J` depicted in Table 3.

([10]) The domino effect is an undesirable property characteristic to "fragile" software systems, demonstrated when small changes in one module have unexpected effects on seemingly unrelated modules. The Y2K problem for example was a manifestation of the domino effect and its consequences.

Table 3. Program-script `Stack-J` in Java 1.4.2.

```
package java.util;

public class Stack extends Vector {
  public   Object push(Object item) {
    addElement(item);
    return item;
  }
  public synchronized Object pop() {
    Object obj;
    int  len = size();
    obj = peek();
    removeElementAt(len - 1);
    return obj;
  }
  public synchronized Object peek() {
    int  len = size();
    if (len == 0)
      throw new EmptyStackException();
    return elementAt(len - 1);
  }
  // ...
}
```

Information hiding mandates that each procedure is declared either as 'public' or as 'private', and that private procedures may only be invoked by procedures defined in the same module (namely the *members* of same module). This description is a metaprogram that can be formulated in the predicate calculus as follows:

$$\forall x,m,p \bullet Private(p) \wedge Member(p,m) \wedge \neg Member(x,m) \Rightarrow \neg Invoke(x,p) \qquad \textbf{IH}$$

The meaning of IH can be defined model-theoretically using Tarski's truth conditions ([11]). Tarski's truth condition furnishes us with a method of checking whether a program-script satisfies the principle of information hiding. Below, we demonstrate how IH may be taken to represent a category of program-scripts, and how to establish that `Stack-J` is in this category.

We may abstractly represent each program-script $p \in \mathbb{M}$ in mathematical logic as a finite structure, consisting of a finite collection of objects and relations amongst them. For example, taking *modules* to be Java classes and *procedures* to be Java methods, the finite structure representing `Stack-J`, designated $\mathfrak{m}_{\texttt{Stack}}$, may consist of the following objects:

$$Modules = \{\texttt{Stack}\}$$
$$Procedures = \{\texttt{Stack.push}, \texttt{Stack.pop}, \texttt{Stack.peek}\}$$

with relations such as

$$Member = \{(\texttt{Stack.push},\texttt{Stack}), \ldots (\texttt{Stack.peek},\texttt{Stack})\}$$
$$Private = \{\}$$
$$Invoke = \{\}$$

([11]) Note that the formulation we offer does not commit us to the axioms of Zermelo-Fraenkel's set theory. Rather, we follow the conventions of semantic formulation merely for finite structures.

It is trivial to establish using Tarski's truth condition that the finite structure representing `Stack-J` *semantically entails* (also *satisfies*) IH, written

$$\mathfrak{m}_{\texttt{Stack}} \models \text{IH}$$

More generally, we may express the mapping from program-scripts in $\mathbb{M}$ into the respective finite structure in the class of all finite structures, designated $\mathfrak{M}$, using the interpretation function $\mathcal{I}$, namely

$$\mathcal{I} : \mathbb{M} \rightarrow \mathfrak{M} \tag{1}$$

This permits us to use Tarski's truth condition to conclude whether a program-script $p \in \mathbb{M}$ *semantically entails* a particular statement in the classical predicate calculus, φ, written

$$\mathcal{I}(p) \models \varphi$$

For instance, we say that $\mathcal{I}$ maps `Stack-J` to the finite structure $\mathfrak{m}_{\texttt{Stack}}$, which, given (1), proves that `Stack-J` *semantically entails* information hiding, written

$$\mathcal{I}(\texttt{Stack-J}) \models \text{IH}$$

The class of program-scripts that *satisfy* such a statement φ in the context of $\mathcal{I}$, written $[\![\varphi]\!]_{\mathcal{I}}$, can be defined as follows:

$$[\![\varphi]\!]_{\mathcal{I}} \triangleq \{ s \in \mathbb{M} \mid \mathcal{I}(s) \models \varphi \}$$

For example, the class of program-scripts that satisfy the principle of information hiding, written $[\![\text{IH}]\!]_{\mathcal{I}}$, is unpacked as follows:

$$[\![\text{IH}]\!]_{\mathcal{I}} \triangleq \{ s \in \mathbb{M} \mid \mathcal{I}(s) \models \text{IH} \}$$

Thus, the metaprogram information hiding (IH) can be taken to represent that category of program-scripts ($[\![\text{IH}]\!]_{\mathcal{I}}$) to which `Stack-J` (Table 3) belongs:

$$\texttt{Stack-J} \in [\![\text{IH}]\!]_{\mathcal{I}}$$

Stack

A data structure is a finite collection of objects that is generally defined by a set of operations on the collection. A *stack* [2] is a data structure that offers three operations as follows: $push(x,s)$ inserts object x into stack s, $pop(s)$ removes and returns the most recently 'pushed' object into s, and $top(s)$ returns the most recently-pushed object to s. A stack of integers can be formulated as a predicate on any user-defined type T (in Java: *class*) declared in a given program as follows:

$$\begin{aligned} &Stack(T) \triangleq \forall x \in \mathbb{N},\ t \in T \bullet \\ &\quad \exists push : T \times \mathbb{N} \rightarrow T,\ \exists pop : T \rightarrow T,\ \exists top : T \rightarrow \mathbb{N} \bullet \\ &\qquad pop(push(x,t)) = t \wedge top(push(x,t)) = x \end{aligned}$$

There are infinitely-many possible program-scripts satisfying the abstract notion of a stack. In other words, there can be any number of types T encoded in any programming language for which $Stack(T)$ holds.

The unary predicate $Stack$ enables the formulation of a program description as an existential statement in the predicate calculus:

$$\exists\, T \bullet Stack(\, T) \qquad \textbf{STK}$$

The semantics of STK can be established in manner similar to IH. Let $\mathbb{T} \subset \mathit{Program\text{-}Scripts}$ be the set of program-scripts which consist of explicit declarations of user-defined types, procedures, and the formal arguments and return types of each procedure. Each program-script in $\mathbb{T}$ is mapped by an interpretation function $\mathcal{J}$ into a finite structure consisting (among others) of a finite collection $Types = \{ t_1, \ldots t_n \}$. For example, the program-script `Stack-J` is likely to be mapped into a list of objects and relations as demonstrated in the previous subsection, except $\mathcal{J}$ shall also make explicit the types of objects that are the formal arguments and return values of each method, thereby establishing that `push`, `pop`, and `peek` are instances of the functions $push$, pop, and top in the predicate $Stack$, respectively. This confirms that $\mathcal{J}(\texttt{Stack-J})$, the finite structure representing program-script `Stack-J`, *semantically entails* STK, and that the program-script `Stack-J` is a member of the category represented by the statement STK:

$$\texttt{Stack-J} \in [\![\text{STK}]\!]_{\mathcal{J}}$$

This demonstrates that the metaprogram Stack (STK) can be taken to represent the category of program-scripts ($[\![\text{STK}]\!]_{\mathcal{J}}$).

The Intension/Locality hierarchy

Eden, Hirshfeld and Kazman [4] (also [5]) observe three categories of design statements (metaprograms) as follows:

1. **Strategic** statements ("architectural-design"), which determine global design concerns, include architectural styles [10] ("The architectural style of Microsoft Outlook is *Client-Server*"), programming principles ("information hiding"), component-based software engineering standards (CORBA, Enterprise JavaBeans), design principles ("all classes inherit possibly indirectly from class Object"), application frameworks, law-governed regularities, and assumptions that may lead to architectural mismatch;
2. **Tactical** statements ("detailed design"), which determine local concerns of limited scope, include as design patterns [9], refactorings ("extract common base class") and programming idioms ("counted pointer can be used to manage memory");
3. **Implementation** statements, which describe specific details of a particular program, such as UML class & collaboration diagrams.

Eden et. al formulate the differentiae between Strategic, Tactical, and Implementation statements using mathematical logic. The differentiae are formulated as the Locality criterion (distinguishing between local vs. non-local statements) and the Intension criterion (distinguishing between intensional vs. extensional statements), giving rise to the Intension/Locality hierarchy. We thus formulate the Intention/Locality hypothesis as follows:

> *Strategic statements are* non-local*, tactical statements are* local *and* intensional*, and implementation statements are* extensional*.*

Statements in all three categories are metaprograms. The Intension criterion can be trivially shown to be an application of interpretation A-III (underspecification) of abstraction. The relation between the locality criterion and interpretations of *abstraction* and the relation between the Intention/locality hierarchy and other metaprograms remain to be examined.

4 Concretization

Concretization is a process during which an entity or entities of one category are synthesized (come into being) from entities of a more abstract category. Below, we briefly examine five categories of concretization processes: synthesizing a hardware entity from a program-script (*hardware synthesis*), synthesizing a program-process from a program-script (*program process-synthesis*), synthesizing machine code from source code (*compilation*), synthesizing a program-script from a metaprogram (*programming*), and synthesizing a new class of program-scripts (the programming language) from a set of metaprograms (*language synthesis*). In §5 we also examine two examples of synthesizing a metaprogram from a more abstract metaprogram (*software design synthesis*). These examples suggest that the concretization process ties entities in one category in our taxonomy with entities of a more abstract category. This discussion shall demonstrate that *concretization* is a fundamental relation between entities in the ontology we examine.

Hardware synthesis is the process of constructing a microprocessor (electronic circuit) from a program-script. Although such technology is yet to mature, subsets of certain hardware description languages (such as VHDL and Verilog®) can be converted into a particular configuration of logic gates and micro-circuits ([12]), yielding a Hardware object. But hardware description languages (HDLs) are in effect programming languages for a number of reasons, in particular since encodings in an HDL must first be interpreted as instructions for a simulator of the target class of microprocessors ([13]). It follows that encodings in HDLs are program-scripts too. We conclude that hardware synthesis can be taken to be a process of *concretization* from a *Program-Scripts* entity into a *Hardware* entity by interpretation A-I (from intangible to tangible).

Program process-synthesis is the process of synthesizing a program-process from a program-script by executing or running the script. For example, when running the program-script in Table 1 on my computer, a new program-process is generated by adding a new entry is added to the list of active program-processes that the operating system holds and a copy of the instructions in Table 1 is placed in the memory of my computer, called the *image* of the program. In the first phase of creating an image, called *loading*, the operating system adapts the machine code (which was generated by my student on his computer) to the environment provided my computer, for example, by translating relative memory addresses (recorded with relation to the memory location of the first instruction in the program-script) into absolute memory addresses (recoded as actual memory locations in my computer). Execution commences by carrying out the (copy of) the first instruction in Table 1 in the program's image, and proceeds from there according to the flow of the program and the values it stored. Observe that in each execution of Table 1, the first command line may be located in a different location in the memory of my computer, the values stored in different locations in memory, and the input supplied thereto may be different. During execution, variables take different values depending on any number of these factors. For self-modifying programs, even the instructions themselves can change. In conclusion, the process of program-process synthesis is a process of *concretization* from a *Machine Code* entity into a *Program-Processes* entity by interpretation A-III (underspecified to specific).

Machine code-synthesis (compilation) is a process of synthesizing a machine-code entity from a source-code entity. In §3.2 we demonstrated that a compilation of the Pascal program-script in Table 2 produces the program-script in Table 1, which is encoded the language of the Intel 8086 machines. It follows that compilation is a process of *concretization* by interpretation A-IV (immanently meaningful to less meaningful) and interpretation A-IV (from a category to in instance thereof) of a *Machine Code* entity from a *Source Code* entity.

([12]) manually or even mechanistically, by a *synthesizer* and appropriate robots, at least in potential.

([13]) for the purpose of verification, for example.

Program script-synthesis (programming) is a process of encoding a program-script. In software engineering terms, programming is generally conceived as a process whose product is a program-script that satisfies a given set of *Metaprogram* entities (specifications). Consider for example a student learning Pascal who is required to encode a program-script that satisfies HW ("*The class of programs which take no input, terminate, and whose output is* "`hello world!`"). In §3.4 we demonstrated that HW can be taken to mean the representation the category of program-scripts that satisfy it, written $[\![\text{HW}]\!]_{\mathcal{I}}$ (for some interpretation function $\mathcal{I}$). More generally, programming is a process of concretization by interpretation A-II of a *Program-Scripts* entity from a set of *Metaprograms* entities.

In §5 we demonstrate two examples of *software design synthesis*, a concretization process that yields a metaprogram that is a concretization of a more general metaprogram.

Language synthesis

Historically, high-level programming languages were motivated by supporting increasingly abstract metaprograms, such as instruction blocks, arithmetic operations (such as the introduction of floating-point expressions), procedural decomposition, recursion, dynamic binding, and inheritance. We postulate that high-level programming languages are formed in a process of synthesis of a subcategory of *Program-Scripts* from of a subcategory *Metaprograms*. Taking the process of language synthesis to be the process of defining and constructing a compiler for the programming language, we examine below the concretization of two programming languages, Java and Lisp. It is possible to demonstrate in a similar manner that the PROLOG programming language was synthesized from of metaprograms that consist of Horn clauses (with finite structures and the closed-world assumption in mind.)

The Java programming language [13] enforces a variation on the principle of information hiding (§3.4). This variation, which we designate IHJ, can be axiomatized as follows:

$$\text{IH} \wedge \text{IHC} \qquad \textbf{IHJ}$$

where IH is defined in §3.4 and IHC represents the additional requirement imposed by class-based programming languages (such as Java, C++, and C#), defined as follows:

$$\forall m \bullet Method(m) \Rightarrow Access(m) \wedge \exists c \bullet Class(c) \wedge Member(m,c) \qquad \textbf{IHC}$$

where $Access(m)$ is the predicate requiring that method m satisfies exactly one of the following: $Public(m)$, $Protected(m)$, or $Private(m)$.

Thus, the synthesis of the Java compiler is a process of program script-synthesis, during which IHJ is concretized, followed by a process of program-process synthesis, during which the compiler's script is executed for the purpose of translating source code into machine code. More generally, the formation of the Java programming language is the result of concretizing by interpretation A-II a *Program-Scripts* entity (the compiler's script) from a subcategory of *Metaprograms* entities (e.g. IHJ), which in turn is concretized into a *Program-Processes* entity.

The Lisp programming language was inspired by Post's notion of recursive functions which McCarthy [17] has taken to mean those mathematical functions that are defined in terms of *recursion* and *functional composition*. For example, the recursive function $factorial$ is defined mathematically as follows:

$$factorial(n) = \begin{cases} n = 0 & 1 \\ n > 0 & n \cdot factorial(n-1) \end{cases}$$

More generally, many recursive functions can be characterized as follows:

$$f(n) = \begin{cases} n = a & c \\ n > a & g \circ f(n-1) \end{cases}$$ R

where a and n are in $\mathbb{N}$ (the set of non-negative integers), f and g are unary functions from $\mathbb{N}$ to $\mathbb{N}$.

Scheme (a dialect of Lisp) was designed to concretize almost a literal representation of recursive functions. Consider for example the Scheme concretization (Table 4) of the $factorial$ function defined above.

Table 4. Program-script `factorial` in Scheme.

```
(define (factorial n)
  (if (= n 0) 1
    (* n (- n 1))))
```

The Scheme syntax is specifically designed to emphasize the definition of a Scheme procedure as a mathematical function ([14,15]). This example demonstrates that the synthesis of a compiler to the Lisp programming language is the process in which a *Program-Scripts* entity (the compiler's script) is synthesized from a subcategory of *Metaprograms* (including entities such as R), namely a process of concretization by interpretation A-II.

We also observe that the mathematical function $factorial$ is *not* equivalent to the program-script `factorial`. In its set-theoretic account, the mathematical function $factorial$ represents an infinite set of pairs

$$factorial = \{(1,1!),\ (2,2!),\ (3,3!),\ \ldots\}$$

whereas the program `factorial` is a *Program-Scripts* entity which, if compiled, executed, and fed with a certain valid non-negative integer n, may calculate the factorial of n, if the hardware, operating system, and all related services function as expected. In this sense, program-script `factorial` (Table 4) is a *concretization* of $factorial$.

5 Ontological commitments of a program-script

Smith [18] describes the ontology to which a representation is committed as those objects and relations whose existence is asserted by the representation. Considering Quine's interpretation (§1), such ontology should provide a small number of elementary building-blocks (objects and relations) into which any account of the ontology of a program can be reduced. Such a conceptual scheme furnishes us with the ontological commitments of certain programs (Q3).

A possible line of inquiry in the ontological commitments of program-script s is offered by examining the ontological commitments made by the programming language in which s is encoded [22]. This suggests that the semantics and underlying axioms of a programming language may furnish us with the ontology to which all program-scripts encoded in this language are committed. Since ex-

([14]) for which reason Lisp procedures are called *functions*.

([15]) or, more accurately, a lambda expression.

plicit axiomatization of actual programming languages is not on offer, we turn to semantic theories. While several flavours of semantics were suggested, *denotational semantics* [20] provides an attractive picture with semantics and ontology as close bedfellows. However, denotational semantics suggests that all programs are ultimately committed to set theoretic *domains* [22]. Unfortunately, these do not offer any explanation to the variety of programming languages (Q3). Moreover, suggesting that set-theoretic constructs are the elementary building-blocks of every program is perverse [22], and at most should be taken as a *reductio ad absurdum*.

Alternatively, consider the ontology offered by the notion of *programming paradigms*. In §4 we demonstrated that the Java compiler concretizes a variation of the principle of information hiding formulated as IHJ. Variables in the formula IHJ range over *classes* and *methods*, the ontology of which is recognized as the *class-based programming* ([16]) *paradigm* [3]. Since the Java compiler enforces IHJ on any Java program-script, we may suggest that every Java program-script is committed to the ontology offered by the class-based programming paradigm. More generally, we may argue that the answer to Q3 is that every program-script $s_{\mathbb{L}}$ encoded in language $\mathbb{L}$ is committed to the ontology furnished by the programming paradigm to which $\mathbb{L}$ is committed.

This assertion however is likely to be false. Not every program in Java is indeed committed to the ontology of classes and methods. The reason is because, while the Java language's syntax requires that all Java program-scripts have at least one class defined therein, it does not necessarily commit all Java program-scripts to contain a *meaningful* collection of classes and members. For example, a 20,000-line Java program-script which consists of a single class is *not* committed to the class-based programming paradigm; such a program-script is only superficially structured that way, but its classes and the methods do not provide us with the simplest conceptual scheme to which the program-script is committed. For this reason, object-oriented programmers jest about how certain other programmers "write Fortran in every programming language", suggesting that a program-script can be committed to a programming paradigm which is not associated with the programming language in which it was encoded.

It is nonetheless evident that programming paradigms play some role in the ontological commitments. To explore this role, we examine below two specific examples. We begin our investigation with a (simplified) description of the category of programs of *email clients*, which can be summarily characterized by the following metaprogram:

EM *An effective representation of text and rich-text (HTML) email messages and their respective operations, encoded in a form which can be communicated between email clients and servers.*

The process of synthesizing an email client which satisfies EM is a process of *program script-synthesis*—namely a process of *concretization* (§4). Let us examine two ways it can be carried out, each of which results in a program committed to a different ontology.

Procedural programming

If taken during the 1970s, programming a concretization to EM was likely to begin with the formulation of the following two software design statements:

Define one record (data structure) representing text email and a second data structure representing HTML emails; **EMP**$_1$

Define the primary instruction structures (procedures), each representing a particular operation modifying a particular email record. **EMP**$_2$

([16]) commonly confused with object-oriented programming.

We shall refer to the conjunction of statements EMP1 and EMP2 (henceforth EMP) as a *procedural* concretization of EM. The formulation of EMP takes form as an intermediate stage in the process of synthesizing a program-script from EM. It describes a subcategory of the programs that are described by EM, that is,

$$[\![\text{EMP}]\!]_{\mathcal{I}} \subset [\![\text{EM}]\!]_{\mathcal{I}}$$

for some interpretation function $\mathcal{I}$ (§3.4). Therefore, although both are metaprograms, EMP is a subcategory of EM.

EMP can be formulated in the classical predicate calculus as existential statements with variables ranging over *records* and *procedures* and relations such as *procedure* p *modifies record* r and *procedure* p_1 *calls procedure* p_2. By Quine's dictum (§1), EMP is committed to the procedural programming paradigm.

EMP can naturally be programmed as a program-script in any procedural programming language from the Algol family. Table 5 depicts the concretization of EMP in Pascal ([17]).

Table 5. Program-script `Email-Pascal` in Pascal

```
{* Various bookkeeping here *}

{******************************************** Text Email: ***}
record TextEmail is
  subject, destinationAddress, replyToAddress: char[MAX];
  contents: char[MAX]
end;
procedure send_TextEmail...
procedure receive_TextEmail...
procedure edit_TextEmail...
procedure display_TextEmail...

{******************************************** HTML Email: ***}
record HTMLEmail is
  subject, destinationAddress, replyToAddress: char[MAX];
  contents: HTML;
end;
procedure send_HTMLEmail...
procedure receive_HTMLEmail...
procedure edit_HTMLEmail...
procedure display_HTMLEmail...
```

We further observe that the syntactic structure and the typesetting conventions of Pascal program-scripts ([18]) make explicit their commitment to specific objects (*procedures* and *records*) and relations (*call* and *modify*). The ontology of `Email-Pascal` can further be made evident by a finite structure that explicitly captures the objects

$$Procedures = \{\texttt{send\_TextEmail}, \ldots \texttt{display\_HTMLEmail}\}$$
$$Records = \{\texttt{TextEmail}, \texttt{HTMLEmail}\}$$

([17]) In a more realistic setting, the programming concretization process will consist of several iterations including the modification and/or fine-tuning of `Email-P`.

([18]) Ignoring variations on these notions that are idiosyncratic to Pascal.

and the binary relations

$$Call = \{(\texttt{edit\_TextEmail}, \texttt{display\_TextEmail}), \ldots\}$$
$$Modify = \{(\texttt{edit\_TextEmail}, \texttt{TextEmail}), \ldots\}$$

In conclusion, when a *Metaprograms* entity (EM) is concretized into a software design statement that is committed to the procedural programming paradigm (EMP), it lends itself naturally to a concretization into a program-script (`Email-Pascal`) encoded in a programming language (Pascal) that is committed to same programming paradigm (procedural programming).

Class-based programming

If taken during the 1980s, encoding a program-script concretizing EM was likely to take an altogether different direction, possibly leading to the formulation of the following software design statements:

Encapsulate the data and instruction structures associated with the polymorphic class `Email`. **EMC**$_1$

Encapsulate the data and instruction structures associated with `TextEmail` *in a monomorphic subclass of* `Email`. **EMC**$_2$

Encapsulate the data and instruction structures associated with `HTMLEmail` *in a monomorphic subclass of* `Email`. **EMC**$_3$

We shall refer to the conjunction of EMC1, EMC2, and EMC3 (henceforth EMC) as a *class-based design* ([16]) concretization of EM. The formulation of EMC takes form as an intermediate stage in the process of synthesizing a program-script from EM. It describes a subcategory of the programs that are described by EM, that is,

$$[\![\text{EMC}]\!]_{\mathcal{I}} \subset [\![\text{EM}]\!]_{\mathcal{I}}$$

As expressed in the terminology established in §3.4, for some interpretation function $\mathcal{I}$. Therefore, although both are metaprograms, EMC is a subcategory of EM.

EMC can be formulated in the classical predicate calculus as existential statements ranging over *classes* and *methods*, and relations such as *class* c_2 *is a subclass of class* c_1 and *method* m *is a member of class* c. By Quine's dictum (§1), EMC is committed to the ontology furnished by the class-based programming paradigm.

EMC is naturally programmed as a program-script in any class-based language. In Smalltalk-80, for example, a *class* encapsulate data and instruction structures (specifically, polymorphic classes are *abstract classes*, monomorphic classes are *concrete classes*) and the *subclass* relation is directly supported. Table 6 depicts the concretization of EMC in Smalltalk-80.

Table 6. Program-script `Email-Smalltalk` in Smalltalk-80 ([19]).

```
Object subclass: #Email
  instanceVariableNames: subject, destinationAddress, replyToAddress
  "methods:"
    send ...
    receive ...

Email subclass: #TextEmail
  instanceVariableNames: textContents
  "methods:"
    edit ...
    display ...

Email subclass: #HTMLEmail
  instanceVariableNames: htmlContents
  "methods:"
    edit ...
    Display ...
```

We further observe that the syntactic structure and typesetting conventions of Smalltalk-80 program-scripts make explicit their commitment to same objects (*classes* and *methods*) and relations (*is-subclass-of* and *is-member-of*). The ontology of `Email-Smalltalk` can further be made evident by a finite structure that explicitly captures the objects

$$Classes = \{\texttt{Email}, \texttt{TextEmail}, \texttt{HTMLEmail}\}$$
$$Methods = \{\texttt{Email>>send}, \ldots \texttt{HTMLEmail>>display}\}$$

the binary relations

$$Subclass = \{(\texttt{TextEmail},\texttt{Email}), (\texttt{HTMLEmail},\texttt{Email})\}$$
$$MemberOf = \{(\texttt{Email}, \texttt{subject}), \ldots\}$$

In conclusion, if a *Metaprograms* entity (EM) is concretized into as a software design statement that is committed to the class-based programming paradigm (EMC), it lends itself naturally to a concretization into a program-script (`Email-Smalltalk`) encoded in a programming language (Smalltalk) that is committed to same programming paradigm (class-based programming).

Analysis and conclusions

We conclude that a **software design concretization** is a process of concretizing an abstract metaprogram (EM) into a less abstract metaprogram (EMP or EMC) that is committed to a particular programming paradigm (procedural or class-based). We showed that software design concretization restricts our interpretation of the original metaprogram ($[\![\text{EM}]\!]_{\mathcal{I}}$) into a specific subcategory of programs that are committed to the same ontology ($[\![\text{EMP}]\!]_{\mathcal{I}}$ or $[\![\text{EMC}]\!]_{\mathcal{I}}$). It the software design metaprogram is further concretized into a program encoded in a programming language (Pascal or Smalltalk) that is also committed to same ontology, the result is a program-script (`Email-Pascal` or `Email-Smalltalk`) that is evidently committed to the respective programming paradigm.

([19]) Some Smalltalk typesetting conventions have been sacrificed for the sake of simplicity.

Finally, we conclude that a program-script $s_{\mathbb{L}}$ encoded in language $\mathbb{L}$ is evidently committed to the ontology of the programming paradigm P if

1. $\mathbb{L}$ is committed to P; and
2. $s_{\mathbb{L}}$ is the product of concretizing a metaprogram that is evidently committed to P (the software design specification).

6 Summary and discussion

In §1 we posed three questions, the answers to which can be summarized as follows:

Q1' Programs can be distinguished from metaprograms and from hardware by the systematic application of interpretations of *abstraction* as differentiae, the refinement of which leads to the program abstractions taxonomy (Figure 1). This taxonomy is

- *exhaustive* in the sense that it accounts for all programs we characterized, including the distinction between program-processes and program-scripts and the distinction between source-code and machine code;
- *unified* in the sense that the categories *Metaprograms*, *Programs*, and *Hardware* form the top-level ontology;
- *uniform* to the extent that all differentiae are interpretations of *abstraction*;
- *definitive* to the extent that all differentiae are unambiguous, some of which were expressed in mathematical logic.

We examined the ontological commitments of two sample program-scripts (§5), from which we concluded the following:

Q2' A program-script $s_{\mathbb{L}}$ encoded in programming language $\mathbb{L}$ is evidently committed to the ontology furnished by the programming paradigm P if the following conditions are satisfied:

- $\mathbb{L}$ is committed to P; and
- $s_{\mathbb{L}}$ is a concretization of a metaprogram committed to P.

Our investigation in *language synthesis* (§4), in particular the process of synthesizing a compiler from a set of descriptions, led us to suggest the following answer to Q3:

Q3' Programming languages are formed by concretizing a specific set of metaprograms. The proliferation of programming languages is explained by the proliferation of possible subsets of *Metaprograms* entities to choose from, the concretization of every consistent combination of which ultimately yields a different language.

The questions we posed and the answers we examined are only the first step in the larger project of investigating the ontology of computer programs. Details of some of the arguments we examined were left out, the formulation of which has been merely sketched. Particular examples were analyzed to corroborate the answers suggested, albeit further examination must demonstrate that the evidence shown is not anecdotal.

We further observe the following open questions:

Q4 While program-processes are intangible, they are nonetheless causal objects [6][1]: They move produce pictures on computer screens and operate machinery. How can intangible objects be

causal objects? Since minds are at once intangible and causal, is this the same problem as the mind-body problem?

Q5 Are any of the interpretations of *abstraction* (A-I to A-V) equivalent? For example, can we identify generalization (A-II) with underspecification (A-III)?

Q6 What is the differentia between software design specifications and other metaprograms? What is the relation to the distinction between functional and non-functional specifications? Does this distinction offer two subcategories of *Metaprograms*?

Q7 Does the distinction between Strategic, Tactical, and Implementation statements (§3.4) offer a exhaustive partitioning of *Metaprograms*?

Q8 Can the Locality criterion [4] be articulated as one of the interpretations of *abstraction*?

Q9 What is the precise relation between languages of metaprograms ("specification languages") and programming languages? Between programming languages and Hardware specification languages?

Q10 Can programming paradigms be fully axiomatized to make their ontologies explicit?

Q11 What is the identity criterion for programs? How exactly are program-scripts and program-processes distinguished from each other? Is this a syntactic, semantic, or another criterion?

Acknowledgements

The authors wish to thank Barry Smith for his detailed comments and for referring us to Roman Ingarden's work [14]; Jack Copeland for his insight; and Bill Andersen for his comments. The authors also wish to thank Naomi Draaijer for her support and Mary J. Anna for her inspiration. This research was supported by the Royal Academy of Engineering and by The Engineering and Physical Sciences Research Council (EPSRC), United Kingdom.

References

[1] Timothy R. Colburn. *Philosophy and Computer Science*. Armonk: M.E. Sharpe, 2003.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. Cambridge: MIT Press, 1990.

[3] Iain Craig. *The Interpretation of Object-Oriented Programming Languages*. New York: Springer-Verlag, 2000.

[4] Amnon H. Eden, Yoram Hirshfeld, Rick Kazman. "Abstraction Classes in Software Design." *IEE Software*, Vol. 153, No. 4 (Aug. 2006), pp. 163–182. London, UK: The Institution of Engineering and Technology.

[5] Amnon H. Eden, Raymond Turner. "Towards an ontology of software design: The Intension/Locality Hypothesis." *3rd European conf. Computing And Philosophy—ECAP* (2-4 Jun. 2005), Västerås, Sweden.

[6] James H. Fetzer. "Program verification: the very idea." *Communications of the ACM*, Vol. 31, No. 9 (Sep. 1988), pp. 1048–1063.

[7] James M. Fielding, Jonathan Simon, Werner Ceusters, Barry Smith. "Ontological Theory for Ontological Engineering." *Proc. 9th Int'l Conf. Principles of Knowledge Representation and Reasoning—KR2004* (2–5 Jun. 2004), Whistler, BC.

[8] Luciano Floridi. “Information”. Ch. in: L. Floridi (ed.) *The Blackwell Guide to Philosophy of Computing and Information*. Malden: Blackwell, 2004.

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading: Addison-Wesley 1995.

[10] David Garlan, Mary Shaw. An Introduction to Software Architecture. In: V. Ambriola and G. Tortora (ed.), A*dvances in Software Engineering and Knowledge Engineering*, Series on Software Engineering and Knowledge Engineering, Vol. 2, World Scientific Publishing Company, Singapore, pp. 1–39, 1993.

[11] W. Wayt Gibbs. “Software's Chronic Crisis.” *Scientific American* (Sep. 1994), p. 86–95.

[12] Adel Goldberg, David Robson. *Smalltalk-80: the language and its implementation*. Reading: Addison-Wesley, 1983.

[13] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification,* 1st edition. Reading: Addison Wesley, 1996.

[14] Roman Ingarden. *The Ontology of the Work of Art*. Translated by R. Meyer with John T. Goldthwait. Athens: Ohio University Press, 1989.

[15] John C. Martin. *Introduction to Languages and the Theory of Computation*. Boston: McGraw Hill, 1996.

[16] Brian McLaughlin. “Computationalism, Connectionism, and the Philosophy of Mind”. Ch. in: Luciano Floridi (ed.) *The Blackwell Guide to Philosophy of Computing and Information*. Malden: Blackwell, 2004.

[17] John McCarthy. “Recursive functions of symbolic expressions and their computation by machine, Part I.” *Communications of the ACM*, Vol. 3, No. 4 (1960), pp. 184–195.

[18] Barry Smith. “Ontology.” In: Luciano Floridi (ed.) *The Blackwell Guide to Philosophy of Computing and Information*. Malden: Blackwell, 2004.

[19] Barry Smith, Jose L.V. Mejino Jr., Stefan Schulz, Anand Kumar and Cornelius Rosse. “Anatomical Information Science”. In: A. G. Cohn and D. M. Mark (eds.), *Spatial Information Theory*. Proc. COSIT 2005 (Lecture Notes in Computer Science), Berlin: Springer, pp. 149–164.

[20] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge: MIT Press, 1977.

[21] Raymond Turner. “The foundations of specification”. *Journal of Logic and Computation*, Vol. 15, No. 5 (Oct. 2005), pp. 623–663. (Also: Technical Reports CSM-397 and CSM-398, Department of Computer Science, University of Essex.)

[22] Raymond Turner, Amnon H. Eden. “Towards a programming language ontology.” Ch. in: Gordana Dodig-Crnkovic, Susan Stuart (ed.) *Computing, Philosophy, and Cognitive Science*. Cambridge: Cambridge Scholars Press (2007).

[23] Willard van Orman Quine. “On what there is.” Ch. in: *From a Logical Point of View*. Cambridge: Harvard University Press, 1961.

[24] John A. Wheeler. “Information, Physics, Quantum: The Search for links.” In: Wojciech H. Zurek (ed.) *Complexity, Entropy, and the Physics of Information*. Redwood: Addison-Wesley, 1991.

[25] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiene. *Designing Object Oriented Software.* Upper Saddle River: Prentice Hall PTR, 1990.

[26] Nicholas Wirth. “Recollections about the Development of Pascal”. In: Thomas J. Bergin II, Richard G. Gibson, II (eds.) *History of Programming Languages-II*. New York: ACM Press, 1996.