# Contents

# Logic and Programs: a Declaration of (In)dependence

## 1 Introduction

How does our Community understand the relation between Logic and Programs? In search for a narrative that could explain such relation in its many forms, history and politics could come handy. Is Logic a motherland to Programs? Are Programs rebels against the Empire?

Employing this figurative way of speech, the core problem can be expressed as follows: does Logic establish the rule of law accoding to which Programs have to behave? or do Programs enjoy a freedom of practice, generously allowed by Logic? How much control does a Program need? And how much of the freedom that an implemented Program enjoys from its Logic is given back, and in what terms? Is there a balanced, fruitful, "special relationship" between these two actors? And what does it mean for the design of logical systems and programming languages?

In this chapter we explore and formulate the problems and the opportunities of colonization, of independence and of regulated commerce between Logic and Programs. Our chart will start by establishing the possible forms of constitutional relationship within the Empire: Dominion, Federation and Independence. We move, historically and conceptually, from the Empire to a Representative Democracy, where the relationships between Model, Implementation and the Real World are constitutive of the relata: Formalisms, Programs and their Executions. It is here essential to define the relationships and to investigate the resulting ontology of Programs.

## 2 Constitution: Logic as Specification.

In a first sense, Logic is the Motherland to its colonies, the Programs. Logic intended as the Specification which fixes the behaviour of the Program. This relation answers to the question: what status does the Empire conceeds to the Colony? Or: what is a computer program from a logical point of view?

A possible answer to this question is that of formulating a general regulation, independently of the specifics of the colonies, the language they speak. In this sense, high-level programs can be expressed in many languages and, especially, they can be formulated through different programming language paradigms.

Also, the answer will be independent of the material regulations of the colonies, the means they use for production: low-level programs can be expressed in assembly language or in machine-code language, and also in this case the general regulation will have to be formulated independently of such specifics. But what should such a regulation look like, and what ontological constraints shuold it impose? How will the distinct linguistic dialects (paradigms and high-level langaugues) and the local production means (low-level langauges and architecture-dependent machine code) affect the generality of the regulation?

Ray (links down to Magna Carta)

Kowalski at a special meeting of the Royal Society in London emphasises the relationship between logic programs and specifications. He observe that the only difference between a complete specification and a program is one of efficiency.

– complexity and evaluation as criteria distinguish specification and implementation

– other level: physical execution

## 2.1 Dominion

this passage is inspired by Nicola's note

According to a first understanding of this problem, given distinct formulations of a program into different programming language paradigms, and distinct compilations into different machine-code languages, the operational nature of the Colony and its relation with the Motherland should remain invariant: the program *is* the same. Then one way to grasp the logical structure underneath those diverse program formulations (the unique relationship with the Motherland, as established by the Regulation) is to consider the common rule of law that the Emperor has imposed for any Colony: the abstract machine implemented by any program. A C program, a Java program, a Prolog program, are all distinct linguistic formulations, with distinct ontological commitments, of the "same program".

In the tradition, two formulations of this policy have been given:

- the Curry-Howard isomorphisms
- denotational semantics

this passage is inspired by Jean-Baptiste's note

The former [...]

The latter can be expressed as the set of operations specified by the underlying abstract machine. A program is logically equivalent to the set of operations it instructs. Imperative, procedural, and object oriented programs make explicit reference to the machine states and operations on those states; logical and functional programming languages leave this duty to the compiler or interpreter. According to this view, a program can be logically defined as the equivalent class of implementations, expressed using different programming languages, of

the same abstract machine. An abstract machine specifies the states of the system to be implemented , its executions, variables, data types, data structures, algorithms, independently of any chosen programming language. The present view on the nature of programs also accounts for low-level programs: assembly language and machine code programs belong to the equivalence class of programs implementing a given abstract machine. Indeed, both assembly language and machine code programs inherit the abstract machine implemented by some high-level language program.

## 2.2 Federation

this passage is inspired by Dale's note

In many early programming systems, it was specific compilers (and interpreters) that determined the meaning of programs. Since computer processors were rapidly changing and since compilers map high-level languages to these evolving processors, compilers needed to evolve in order to exploit new processor architectures. Since the new compilers did not commit to preserving the same execution behavior of programs as earlier compilers, the meaning of programs would also change. For the many people writing high-level code, the fact that their code could break when moving it between computer systems or to higher version numbers eventually became a serious problem. This situation became untenable when programs also grew dependent on the services – such as memory management, file systems, and network access – offered by operating systems: now programs could also break whenever there were changes to operating systems.

Early efforts to formalize the meaning of programs employed the computation-as-model paradigm, i.e. where computing is understood as the process happening in machines, and logic is used to describe what happens. This can be understood as a federative approach, were each region has their own specific regulations, all falling under a certain general one. While this approach to reasoning about the meaning of programs has had some success and is used in several existing systems today, it has also had some significant failures. In fact, the topic of model checking, in which the search for counterexamples (bugs) replaced the search for formal proofs, arose from frustration that it was too difficult to use pre- and post-condition reasoning in many systems, particularly, those that had elements of distributed and concurrent execution.

## 2.3 Independence

this passage is inspired by Nicola's note

According to [**?**], contrasting ontological perspectives on the logical nature of programs come from distinct linguistic expressions thereof:

- high-level programs expressed by means of imperative languages can be defined as sets of operations upon machine states;

- programs should be rather considered mathematical functions if expressed in a functional programming language, or logical sentences when formulated through logic languages;
- finally, under the object-oriented paradigm, programs are sets of communicating objects.

Hence, the nature of the Colonies and how they are treated by the Regulation, differs according to their specificities, language and means of production.

# 3 Representative Democracy: Model, Implementation and the Real World.

this passage is inspired by Nick's note

Assuming that the form of the relation between Colonies and Motherland within the Empire is either that of the Federation, or rather that of the Dominion, the next question to be explored is: to what extent do Colonies depend on the Regulation, and thereby how much do they depend on the Empire? Or, out of the analogy, to what extent do computer programs depend on logic?

The question has a metaphysical motivation, and its answer necessarily aims for a better understanding of the ontological status of computer programs by analyzing the kind and degree of dependence. In metaphysics there are different forms of so called ontological dependence, often spelled out in modal terms. For instance:

> $x$ depends for its existence upon y $=_{df}$
> Necessarily, $x$ exists only if $y$ exists.

Illuminating the kind of ontological dependence of entities $x$ upon $y$ (in our case $x =$ "Colony" and $y =$ "Motherland"; or $x =$ "computer programs" and $y =$ "logic") may help one to better understand the ontological status of $x$. How does the original question about the dependence of computer programs to logic help us understand the the nature of the former?

Even though the question has a clearly philosophical connotation, it can not only be approached from a philosophical angle, but also from a historical and a practical perspective. For instance, has the relationship (especially "ontological dependence") between logic and computer programs historically always been the same? Can there be (historically) identified different kinds of dependence relations between computer programs and logic? Shedding light on these issues may help us to find an answer to the original question "To what extent do computer programs depend on logic?", which in turn would largely benefit our understanding of the metaphysics of programs in general.

## 3.1 The Dual Nature View

Many philosophical view support the so called dual-nature view; on the one hand, computer programs causally interact with the world and "do something",

they are physical (in a certain way); on the other hand they are perceived as these abstract-mathematical-object like entities, which don't bear any physical properties. Take for contrast, regular "material objects" like rocks or tigers – they seem to lack any ontological dependence from "logic". In other words, logic is not an integral ingredient of what constitutes those objects as being rocks or tigers.

Mathematical objects on the other hand, appear to be largely contingent on logic (in one form or another). But how is the situation then when it comes to computer programs? Is logic really an over and above necessary ingredient for constituting a computer program? Could there be computer programs without logic and what, subsequently, are the ramifications for the understanding of their ontological status?

### 3.2 The Layered Ontology View

this passage follows on Nicola's and Giuseppe's work for SEP

### 3.3 The Third Way: Pragmatism

this passage does not exist. A student of mine is doing some work in this direction, wonder whether we should explore.

## 4 The Parties: Declarativists and Constructivists.

Ray's

Logic programming versus constructive programming. They have different notions of computation and program. Philosophical differences in underlying semantics of the specification rather than the programs. In constructive approach the type is the specification that has constructive content that is unpacked in terms of the programs that provide witness to the truth of the proposition. In the logic paradigm the logical assertion is both the specification and the program but is given different interpretations – with all that is entailed by calling it a specification. Namely as assertion it provides the correctness criteria for its guise as program. This is reflected in the difference between the truth conditional and the operational semantics.

## 5 Regulations: Types in Logic and Programming.

Tomas?

## 6 Magna Carta: Linguistic Variants.

Ray's note

The different programming language paradigms impact upon the nature of programs. Roughly, different paradigms would seem to generate different notions of program. Functional, procedural, object oriented and logical paradigms present different notions of what a program is and different notions of computation. This provides a philosophical and historical perspective on the question what is a program. The parallel, non-deterministic dimension also contributes to the notion of program. On the other hand, are there aspects of programs that transcend the paradigms?

# 7 Habitus: Contexts and Practices.

## 7.1 Different Implementations

GP?

The ontological analysis of the nature of computer programs from a logical perspective is connected to the analysis of programs from the software systems viewpoint. Given the abstraction level hierarchy defining software systems, answering to the question "what is a (logic) program?" amounts to identifying the level of abstraction that better grasps the (logical) nature of programs. Choosing, as one might expect, the high-level language program level is reducing, insofar as assembly language and machine code programs would be ruled out. By contrast, by focusing on higher levels of abstraction, such as the level of formal specifications, one would include too many entities in the equivalence class defined by the programs implementing the same specifications. Indeed, the same specification may be realized by different algorithms and consequently different programs which do not necessarily share the same set of states, operations, data types and structure.

## 7.2 Verification

Note by Selmer on Different notions of Verification. Can he turn it into something that fits here?

# 8 Loopholes: Limits and Approximations.

## 8.1 Pathology vs. Correctness

Liesbeth?

## 8.2 The Kopetz Principle

Passage inspired by Edgar

> Many (predictive) properties that we assert about systems (determinism, timeliness, reliability, and correctness) are in fact not properties of an implemented system, but rather properties of a model

> of the system. We can make definitive statements about models, from which we can infer properties of system realizations. The validity of this inference depends on model fidelity , which is always approximate.

# 9 Dialogue: Another Role for Formal Methods.

Passage inspired by Liesbeth

In the introduction of the well-known and early collection of papers on the history of computing, one reads [**?**, p???]:

> The improbable symbolism of Peano, Russel, and Whitehead, the analysis of proofs by fowcharts spearheaded by Gentzen, the definition of computability by Church and Turing, all inventions motivated by the purest of mathematics, mark the beginning of the computer revolution. Once more, we find a confirmation of the sentence Leonardo jotted despondently on one of those rambling sheets where he confided his innermost thoughts: 'Theory is the captain, and application the soldier.'

The viewpoint of theory (logic) being the captain and the application (program) being the soldier requires that their role be strictly separated, both historically and epistemologically. This requirement appears difficult to satisfy. Arguably, a lot of the so-called theoretical foundations have been retro-fitted to the discourse; e.g, the idea that theoretical insights from the 1930s were the historical basis for the first computers. Hence, this approach seems the result of much academic strategy.

Much motivation for developing logical and formal approaches are driven by the idea of a kind of perfect program, where, perfect means, being in control of everything; having certainty that there are no errors and that there will be no errors. But being in control also means that one is limiting the possibilities of something else (the machine, the user, the language, etc). The history of computing, however, is full of examples where lack of control is exactly what is needed to make progress. One basic question might be in which contexts such complete control is desired and if that is achievable. Perhaps one well-known example is Turing's insistence on the machine's ability to make mistakes. This seems an important precondition in order to have genuine machine intelligence.

On the other hand, the view that application should be seen too much as the captain is the more popular view since the 1980s, not just in computing, but in science at large, see e.g. [**?**]. Considering different cases from the history of Computing, captain-like statements with respect to logic have had signifcant consequences, intensifying certain communication gaps. Alongside such gaps, physical computing systems seem to remain opaque, despite the (extensive or entirely missing) effort of formalization.