Serious Verification as the Unifier of Logicist Programming

RT • SB • GP • ED • J-BJ?

`rough non-detechnicalized version 1024191545NY`

Abstract

After distinguishing between two fundamentally different conceptions of logicist (computer) programs and programming (the declarative vs. the constructivist traditions), we consider the challenge of what we call 'serious' program verification, and find that it catalyzes the unification of these standardly competing conceptions. The unification can be viewed as offering to others an integrated logicist paradigm, cast at a philosophical level, for subsequently achieveing high-assurance computer programming.

Contents

1 Introduction

After distinguishing between two fundamentally different conceptions of logicist (computer) programs and programming (the **declarative** vs. the **constructivist** traditions), we consider the challenge of what we call 'serious' program verification, and find that it catalyzes the unification of these ordinarily competing conceptions. The unification can be viewed as offering to others an integrated logicist paradigm, cast at the philosophical level, for subsequently achieveing high-assurance computer programming.

The plan for the paper is as follows. We begin (§2) with a synopsis of basic declarative programming, including a few remarks about its historical roots and current real-world practice. Next (§3), we consider an approach to program verification on the declarative approach, and find that initial sanguinity regarding this approach melts away to reveal that what can be called 'serious' verification is outside the reach of this approach. The upshot is a turning to the constructivist approach for rescue; section 4 sets out in broad terms this approach, and then section 5 explains the rescue itself. A brief concluding section wraps up the paper, and includes some remarks regarding the future of the unified approach we have introduced.

2 Basic Declarative Logicist Programming

Let Φ be a set of wffs constructed in accordance with a formal grammar of an extensional logic $\mathscr{L}$ sufficiently less expressive than first-order logic $\mathscr{L}_1$ to avoid the *Entscheidungsproblem.*[1] We say that Φ is a **logic program**. Our logic $\mathscr{L}$ includes some collection $\mathcal{I}^*_{\mathscr{L}}$ of inference schemata for deduction that gives the proof theory of this logic, and when some formula ϕ is deducible from Φ in this theory we — following standard notation — write

$$\Phi \ \vdash^{Y}_{\mathcal{I}^*_{\mathscr{L}}} \phi.$$

If instead of 'Y' in this statement there is a 'N,' then what is being said is that deducibility fails to hold. When the provability relation symbol is superscripted as in

$$\Phi \ \vdash^{\pi}_{\mathcal{I}^*_{\mathscr{L}}} \phi,$$

[1] E.g. $\mathscr{L}$, formula-wise, can be restricted to those having at most two distinct variables, and allowing identity. Such a logic is decidable (as shown by Scott, who built on Gödel's completeness theory, and — for the allowance of identiy — Mortimer after Scott), and dodges Church's Theorem.

what is being said is that particular proof π goes from Φ to ϕ in the relevant proof theory. To say that it's a question and at present unknown as to whether there exists a proof from Φ to ϕ by the inference schemata in question, we write

$$\Phi \ \vdash^{?}_{\mathcal{I}^{*}_{\mathscr{L}}} \ \phi.$$

We have said that Φ is a program. Surely that will seem to be most to be rather odd. After all, this means that a mere collection of formulae is counted as a program, and such a collection seems rather static. How does something happen? After all, surely the point of a program is to set matters up so that something meaningful can happen under the governance of said program. In the present context, things begin to happen when, first, we issue a query $\mathfrak{q}$ against the program Φ. For example, set $\Phi' \ := \{p \rightarrow q, q \rightarrow r, \neg r\}$, and consider a query as to whether from Φ' is can be proved that $\neg p$. The query is given to a reasoner $\mathcal{R}$, and in this case, of course, the answer given back will be a Yes, accompanied by a simple proof that we shall label π'; this proof might for instance first use hypothetical syllogism to obtain $p \rightarrow r$, and from this and given $\neg r$ by *modus tollens* deduce $\neg p$. In the notation introduced above, we have the following query in the example now before us:

$$\mathfrak{q} := \Phi' \ \vdash^{?}_{\mathcal{I}^{*}_{\mathscr{L}}} \ \neg p$$

And we have have the following returned by automated reasoner $\mathcal{R}$:

$$\Phi' \ \vdash^{\pi'}_{\mathcal{I}^{*}_{\mathscr{L}}} \ \neg p.$$

3 The Verification Challenge to the Declarative Approach

Now, how do we carry out verification of the program and of the result of reasoner taking it and producing its output? The immediate and sanguine response to this question is that we have only to verify that the returned π is a valid proof, all the way through, that its starting givens match Φ, and that its ultimate conclusion matches ϕ. This can be accomplished by a dirt-simple automated proof checker $\mathcal{C}$.[2]

But this is absolutely *not* a meeting of the challenge of *serious* program verification. In serious program verification all elements of the overall

[2] Note that while in our example the proof π' can be checked by hand, nearly all proofs will be too long to be manually checked, and programs will be executed repeatedly through time for any real-world applications, which puts manual checking further out of the realm of feasibility.

process of computing must be scrutinized, top to bottom. From this perspective, what is left out? What gets a free pass? The answer is obvious: the checker has been ignored. Why should we trust $\mathcal{C}$? By definition, it is outside the paradigm that has been set out!

4 Basic Constructivist Logicist Programming

5 Constructivists to the Rescue

6 Conclusion

7 Acknowledgements

This paper was is an outcome of the ANR PROGRAMme project (ANR-17-CE38-0003-01), specifically of interaction at the Bertinoro 2018 meeting.

References