Computation Calculus
Bridging a Formalization Gap

Rutger M. Dijkstra
Rijksuniversiteit Groningen

Abstract. We present an algebra that seeks to bridge the gap between programming formalisms that have a high level of abstraction and the operational interpretations these formalisms have been designed to be sound for.

In order to prove a high level formalism sound for its intended operational interpretation, one needs a mathematical handle on the latter. To this end we design the computation calculus. As an expression mechanism, it is sufficiently transparent to avoid begging the question. As an algebra, it is quite powerful and relatively simple.

0 Introduction

For reasoning about (imperative style) programs, lots of extensions of predicate calculus or logic are in circulation: Hoare-logic, wp-calculus, temporal logics of various kinds, UNITY-logic, etc. All these extensions enrich the language of predicate calculus with ‘primitives’ intended to capture some operational aspects of programs. E.g. in wp-calculus

\[ wp.s.q : \text{ holds in those initial states for which every execution of program } s \text{ terminates in a state satisfying the predicate } q, \]

in linear time temporal logic

\[ G s : \text{ holds for those computations (i.e. infinite sequences of states) for which every suffix satisfies } s, \]

or for a UNITY program

\[ wlt.q : \text{ holds in those (initial) states for which every computation of the program contains at least one state satisfying the predicate } q. \]

The verbal right-hand sides above are ‘intended operational interpretation’ without any formal status: they serve only to translate operational intentions into the formalism. In order to actually prove properties of programs, the formalisms supply rules in the form of postulates, definitions, axioms, inference rules, and so forth. E.g.

\[ wp.(S;T).q = wp.S.(wp.T.q) \]
\[ \vdash G(s \Rightarrow Xs) \Rightarrow (s \Rightarrow Gs) \]
\[ wlt.q = \langle \mu x :: (\exists S :: (\forall y :: q \lor (\forall T :: wp.T.y) \land wp.S.x)) \rangle \]

This is where the problems start. Although for many of the rules it is quite clear that they are indeed adequate for the intended operational interpretation
—see (a)— this is by no means always the case —see (c). This distance between intended interpretation and mathematical formalization is what we call “the formalization gap”. If large, the formalization gap can become a serious problem, since it means that the things we write down do not necessarily mean what we think they do.

In this paper we develop an algebra that we call “the computation calculus” and that is intended to bridge this formalization gap. The idea is that intended operational interpretations can be expressed succinctly in the computation calculus and that characterizations on a higher level of abstraction can subsequently be derived from that. E.g. for the predicate transformer $wlt$, we would ideally like to be able to derive the fixpoint expression in (c) from some ‘direct’ formalization of the intended operational interpretation.

We abstract the computation calculus from a simple model of computations. Thus, we start with a model wherein we define some operators satisfying certain properties. Subsequently we forget the definitions and postulate the relevant properties. This ensures that all results in our algebra are valid for the model and it provides a clear view on the properties we use thereof, i.e. the assumptions we rely on.

The procedure we follow is incremental for both the model and the algebra. We begin with a model that is far too general and we extract from that some very elementary insights that we capture in our postulates. As we become more ambitious in the things we want to prove, we may find that our algebra is not yet strong enough. We then add what is missing to the algebra and impose, if necessary, the appropriate restrictions on the model.

After the initial exploration of the basics, the first slightly more ambitious target we aim for is the pre-/postcondition semantics of simple sequential programs. The biggest hurdle to overcome here is the treatment of (tail-) recursion. Subsequently, we prepare the ground for higher targets by enriching our algebra with some insights concerning atomic computations. This leaves us with an algebra that is very powerful indeed: subsuming various temporal logics. We end with a short discussion on UNITY.

We build on top of the predicate calculus of E.W. Dijkstra and C.S. Scholten [3] augmented with whatever facts from lattice theory we have use for, in particular the fixpoint calculus as explored by R.C. Backhouse, J.C.S.P. van der Woude, and colleagues [13]. Some highlights of this calculus are listed in the appendix.

Readers familiar with relation algebra in any form, the sequential calculus of B. von Karger and C.A.R. Hoare [9], or any of various temporal logics may recognise where we got our inspiration from. However, unfamiliarity with these fields should not stand in the way of understanding our algebra.

1 Basics of the Model

Our model is in essence taken from Johan Lukkien’s PhD-thesis [11]. A “program (fragment)” or “statement” operates on a set of states. Starting a statement in