Science of Computer Programming 85 (2014) 102–114

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

The laws of programming unify process calculi

Tony Hoare [a], Stephan van Staden [b,*]

[a] Microsoft Research, Cambridge, United Kingdom
[b] ETH Zurich, Switzerland

A R T I C L E I N F O

Article history:
Received 20 December 2012
Received in revised form 15 August 2013
Accepted 22 August 2013
Available online 4 September 2013

Keywords:
Algebra of programming
Hoare logic
Process calculi
Operational semantics
Event model

A B S T R A C T

We survey the well-known algebraic laws of sequential programming, and propose some less familiar laws for concurrent programming. On the basis of these laws, we derive the rules of a number of classical programming and process calculi, for example, those due to Hoare, Milner, and Kahn. The algebraic laws are simpler than each of the calculi derived from it, and they are stronger than all the calculi put together. Conversely, most of the laws are derivable from one or more of the calculi. This suggests that the laws are useful as a presentation of program semantics, and correspond to a widely held common understanding of the meaning of programs. For further evidence, Appendix A describes a realistic and generic model of program behaviour, which has been proved to satisfy the laws.

1. Introduction

The basic ideas and content of the algebraic laws of sequential programming, formulated in [1], are familiar. That paper treated the main program structuring operators, including sequential composition, choice, and recursion. This paper (and the conference paper it extends [2]) introduces additional laws to deal with conjunction and concurrent execution of programs. Since none of the proofs use induction on the structure of the program, all theorems remain valid when new operators and new axioms are added to the language.

The main content of the paper is a unifying treatment of a varied collection of programming calculi, which have been proposed as formalisations of the meaning of sequential and concurrent programming languages. They have been successfully applied in human and mechanical reasoning about the properties of programs expressed in the given calculus. Examples of such calculi are due to Hoare [3], Milner [4], Kahn [5], Dijkstra [6], Back and von Wright [7], Morgan [8], Plotkin [9], and Jones [10]: in this paper, we shall concentrate on the first three.

The initial step in the unification of the calculi is to generalise the concept of a program to cover also program designs and program specifications. We regard them all as descriptions of the events that occur in and around a computer that is executing a program. The program itself is the most precise description of its own execution. The most abstract description is the user specification, which mentions only aspects of execution that are observable and controllable by the user. A design is a description of program behaviour expressed in a mixture of programming and specification notations.

The unification of specifications, designs and programs has the immediate advantage that the same laws apply to all these concepts. Furthermore, the concept of refinement, identified with logical implication, can be applied to all of them, and to meaningful relations between them. Finally, the specifications, being descriptions of machine behaviour rather than machine state, can describe many non-functional properties of a program, which have hitherto been the domain of different logical theories, such as temporal logic.

* Corresponding author.
E-mail address: Stephan.vanStaden@inf.ethz.ch (S. van Staden).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.08.012

For example: a specification may require only conditional correctness (if execution is finite, the final state will satisfy a certain predicate), or it may require termination (all executions of the program are finite). It may specify a relation between initial and final state (the final value of an array is a sorted permutation of the initial value). It may specify non-functional properties such as security (e.g. there is no leakage of data from high to low security threads), persistence (no executions terminate), fairness (every thread will eventually make progress), liveness (no infinite internal activity), timing (a response is always given to a request before a certain deadline), and even probability (as execution progresses, the ratio of a-events to b-events tends to unity).

To explore the relationship between the laws and these calculi, we define the basic judgement of each calculus in terms of one or more inequations between purely algebraic formulae. We then derive all the rules of each calculus as theorems of the algebra. Under basic and reasonable assumptions about descriptions, programming operators and judgements, it is also possible to show that every calculus is equivalent to an algebra. These algebraic characterisations describe the essence of each calculus. They make it easy to determine the strength of a calculus, and also serve to relate and compare the various calculi. These insights and results extend the conference version [2] of this paper.

We make no claims that our algebraic laws actually hold of the operators of any particular programming language. We rely on the good will of the readers to check the individual laws against their intuitive understanding and experience of the essential concepts of programming. The demonstration that these properties are valid for many historic programming calculi gives some independent evidence that the algebra is potentially useful, and that it corresponds to a widely held common understanding of the meaning of programs. We provide additional evidence in this extended version of the conference paper [2] by describing a generic and realistic model of program behaviour in Appendix A. All the laws are valid in the model. It is generic, in the sense that it offers a small set of fundamental parameters, which can be set differently for each of its many applications. The model is intended to be realistic, providing an abstraction of the way that real systems and computers behave.

The general typographical conventions of the paper are that postulates are bulleted, and that theorems (which follow from the postulates) are named. Bullets and names are omitted for material that has only local relevance.

All theorems of this paper have been formally checked with Isabelle/HOL. A proof script is available online [11].

2. Laws of programming

Programs, specifications and designs together form the set of descriptions that we consider. These descriptions are ordered according to refinement: $P \subseteq Q$ indicates that P refines Q. This refinement has several meanings. For example, it can say that the program P is more determinate than program Q (i.e. P has fewer behaviours or executions) or that the specification P is stronger than the specification Q (i.e. P implies Q). Generally, a description is more abstract or general compared to the descriptions that refine it, and the refined description is more deterministic. Refinement obeys three laws that make it a partial order:

- $P \subseteq P$
- $P \subseteq Q \quad \& \quad Q \subseteq R \quad \Rightarrow \quad P \subseteq R$
- $P \subseteq Q \quad \& \quad Q \subseteq P \quad \Rightarrow \quad P = Q$

Among the descriptions, there are three constants taken from programming languages and propositional logic: *skip*, $\bot$ and $\top$. The constant *skip* is a basic program that does nothing. Bottom $\bot$ represents the predicate False: it describes no execution. Bottom is the meaning of a program containing a fault like a syntax violation: the implementation is required to detect it, and to prevent the program from running. Top $\top$ is a program whose execution may have unbounded consequences. Think of a program that can behave in arbitrary ways, for example because it admits an attack by a virus. As a proposition, it can be identified with the predicate True. It is the programmer's responsibility to avoid submitting such a program for execution – the implementation is not required to detect it.

Apart from the constant descriptions, there are operators for forming descriptions in terms of others. The operators are likewise drawn from programming languages and propositional logic. For example, sequential composition (;) and concurrent composition (‖) are binary operators from programming: the formula $P ; Q$ describes the sequential composition of P and Q, while $P \parallel Q$ is a description of their concurrent behaviour.

Conjunction ($\wedge$) and disjunction ($\vee$) are operators familiar from propositional logic. If P and Q are programs, $P \vee Q$ is the nondeterministic choice between the components P and Q. The choice may be determined at some later stage in the design trajectory of the program, or by a specified condition tested at run time; failing this, it may be determined by an implementation of the language at compile time, or even nondeterministically at run time. It satisfies the following laws:

- $P \subseteq P \vee Q$ and $Q \subseteq P \vee Q$.
- Whenever $P \subseteq R$ and $Q \subseteq R$, then $P \vee Q \subseteq R$.

These laws say that ($\vee$) is the least upper bound with respect to the refinement order. Conjunction is its dual and corresponds to the greatest lower bound. Conjunction between programs is in general very inefficient to implement, because