Science of Computer Programming 85 (2014) 204–223

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Concurrency and local reasoning under reverse exchange [☆]

H.-H. Dang [*], B. Möller

Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany

H I G H L I G H T S

- A relation-algebraic semantic model for true concurrency and separating conjunction.
- Proof of an exchange law between sequential and concurrent composition.
- Further semantic background, in particular on so-called precise program commands.
- Proofs of several variants of Hoare-style inference rules for concurrency and locality.
- Application of these rules in some examples.

A R T I C L E I N F O

Article history:
Received 20 December 2012
Received in revised form 29 May 2013
Accepted 17 July 2013
Available online 1 August 2013

Keywords:
True concurrency
Relational semantics
Hoare logic
Concurrent separation logic
Locality

A B S T R A C T

Quite a number of aspects of concurrency are reflected by the inequational exchange law $(P * Q) ; (R * S) \leqslant (P ; R) * (Q ; S)$ between sequential composition ; and concurrent composition $*$. In particular, recent research has shown that, under a certain semantic definition, validity of this law is equivalent to that of the familiar concurrency rule for Hoare triples. Unfortunately, while the law holds in the standard model of concurrent Kleene algebra, its is not true in the relationally based setting of algebraic separation logic. However, we show that under mild conditions the reverse inequation $(P ; R) * (Q ; S) \leqslant (P * Q) ; (R * S)$ still holds there. From this reverse exchange law we derive slightly restricted but still reasonably useful variants of the concurrency rule. Moreover, using a corresponding definition of locality, we obtain also a variant of the frame rule, where $*$ now is interpreted as separating conjunction. These results allow using the relational setting also for modular and concurrency reasoning. Finally, we interpret the results further by discussing several variations of the approach.

1. Introduction

Over the recent years, logical techniques in program semantics have been supplemented by algebraic approaches which frequently allow more concise and perspicuous reasoning. The present paper extends one particular approach in that area, viz. *Algebraic Separation Logic* [2]. That framework was developed to reflect *separation logic* (*SL*) [3]. Although SL originally was developed to facilitate reasoning about shared mutable data structures, it has proved to be also very effective for modular reasoning about concurrency [4,5]. For this logic there are already several abstract approaches that capture corresponding calculi, e.g., [6]. A more comprehensive general algebraic structure is provided by *Concurrent Kleene Algebra* (*CKA*) [7]. A central concept of that algebra is that it allows easy soundness proofs of important rules like the concurrency and frame rules used in logics for concurrency and modular reasoning.

[☆] This paper is a significantly extended and revised version of Dang and Möller (2012) [1].
[*] Corresponding author.
E-mail addresses: h.dang@informatik.uni-augsburg.de (H.-H. Dang), moeller@informatik.uni-augsburg.de (B. Möller).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.07.006

The *concurrency* and *frame* rules have the form

$$\frac{\{P_1\}\,Q_1\,\{R_1\} \qquad \{P_2\}\,Q_2\,\{R_2\}}{\{P_1 * P_2\}\,Q_1 * Q_2\,\{R_1 * R_2\}}\ (\mathit{conc}) \qquad \frac{\{P\}\,Q\,\{R\}}{\{P * S\}\,Q\,\{R * S\}}\ (\mathit{frame}).$$

Here Q and Q_i denote programs while all other letters denote assertions. The essential feature of these rules is the *separating conjunction* $*$ which stands for non-interfering concurrency or disjointness of resources. Hence these rules express that one may reason in a modular way about program parts when the context does not interfere with them.

Interestingly, the recent paper [8] shows that validity of the concurrency rule using the triple interpretation $\{P\}\,Q\,\{R\} \Leftrightarrow_{df} P\,;Q \subseteq R$ is equivalent to validity of the *exchange law*

$$(P_1 * P_2)\,;(Q_1 * Q_2) \leqslant (P_1\,;Q_1) * (P_2\,;Q_2),$$

for programs P_i and Q_i. Likewise, validity of the frame rule is equivalent to validity of the *small exchange law*

$$(P_1 * P_2)\,;Q_1 \leqslant (P_1\,;Q_1) * P_2.$$

In these laws, semicolon denotes sequential composition, while $\leqslant$ denotes a partial ordering expressing refinement. The exchange laws abstractly characterise the interplay between sequential and concurrent composition. Each of them expresses that the program on the right-hand side has fewer sequential dependences than the one on the left-hand side.

There are several algebraic models satisfying those laws:

– The standard model is based on sets of traces. This model is defined very abstractly so that it needs to be refined further to model concurrency with concrete programs adequately enough. However, it enables elegant and simple proofs.
– Another model employs predicate transformers to abstractly capture program behaviour of separation logic. It validates a certain part of the CKA laws, in particular the exchange law. But it fails to satisfy other important laws needed for program proofs as, e.g., laws in connection with non-deterministic choice.

More details may be found in [7,8].

The purpose of the present paper is to investigate relationally based Algebraic Separation Logic mentioned above with respect to exchange laws, extending [1]. As a relational structure it copes well with non-determinacy; moreover, it allows the re-use of a large and well studied body of algebraic laws in connection with assertion logic. Surprisingly, although the model satisfies neither of the mentioned exchange laws, it validates an exchange law with the reversed refinement order. This entails variants of the concurrency and frame rules with similarly simple soundness proofs as in the original Concurrent Kleene Algebra approach. Additionally, we establish an equivalence between the concurrency rule and the reverse exchange law analogous to the one in [8]. This shows that the relational model can be applied in reasoning about programs that involve true concurrency and modularity. To underpin this further, we also study a number of variations of our main relational model and discuss their adequacy and usefulness.

2. Basic definitions and properties

We start by repeating some basic definitions from [2] and some direct consequences. Summarised, the central concept of this paper is a relational structure enriched by an operator that ensures disjointness of program states or executions. Notationally, we follow [2,8].

Definition 2.1. A *separation algebra* is a partial commutative monoid $(\Sigma, \bullet, u)$; the elements of Σ are called *states* and denoted by $\sigma, \tau, \ldots$. The operator $\bullet$ denotes state combination and the *empty state* u is its unit. A partial commutative monoid is given by a partial binary operation satisfying the unity, commutativity and associativity laws w.r.t. the equality that holds for two terms iff both are defined and equal or both are undefined. The induced *combinability* or *disjointness* relation # is defined by

$$\sigma_0 \# \sigma_1 \quad \Leftrightarrow_{df} \quad \sigma_0 \bullet \sigma_1 \quad \text{is defined.}$$

As a concrete example one can instantiate the states to heaps. For this one has $\Sigma = \mathbb{N} \rightsquigarrow \mathbb{N}$, i.e., the set of partial functions from naturals to naturals. Moreover $\bullet$ is the union of domain-disjoint heaps and $u = \emptyset$, the empty heap. The corresponding combinability relation is $h_0 \# h_1 \Leftrightarrow \mathit{dom}(h_0) \cap \mathit{dom}(h_1) = \emptyset$ for heaps h_0, h_1. More concrete examples can be found in [6].

Definition 2.2. We assume a separation algebra $(\Sigma, \bullet, u)$. A *command* is a relation $P \subseteq \Sigma \times \Sigma$ between states. Relational composition is denoted by ;. The command skip is the identity relation between states. A *test* is a sub-identity, i.e., a command P with $P \subseteq \mathsf{skip}$. In the remainder we will denote tests by lower case letters $p, q, \ldots$. A particular test that characterises the empty state u is provided by $\mathsf{emp} =_{df} \{(u, u)\}$. Moreover, the domain of a command P, represented as a test, will be denoted by ΔP. It is characterised by the universal property

$$\Delta P \subseteq q \quad \Leftrightarrow \quad P \subseteq q\,;P.$$

In particular, $P \subseteq \Delta P\,;P$ and hence $P = \Delta P\,;P$.