Separation Logic and Trace Semantics

Tony Hoare and Peter O'Hearn
Microsoft Research, Cambridge, United Kingdom

Summary We construct a simple mathematical model of the relationship between a correct computer program and a complete trace of its successful execution. The theory is comprehensively illustrated by pictures. We treat many awkward features and aspects of modern programming. They include dynamic allocation, transmission and disposal of resources; multi-processing, multi-threading and multi-core; communication on unreliable, lossy, re-ordering, multiplexed, buffered or synchronised channels; out-of-order execution, instruction pipe-lining and relaxed memory models. We show how separation logic, used as an assertion language, controls the interference problems that arise. We fail to describe or to solve the problem of deadlock.

Introduction Our application of separation logic to trace semantics has several aims: (1) to deal with each programming feature independently, so that feature-rich languages can be understood simply in a modular fashion; (2) to exploit programmers$\tilde{\text{O}}$ intuitions about program execution, and strengthen it by use of diagrams; (3) to provide useful variations in the definition of our principles, so they can be adapted and selected to cover a wide range of program design patterns, expressed in different programming languages; (4) to facilitate modular proofs of programs expressed in these languages.

In pursuit of these aims, we have adopted a top-down analytic approach to the presentation of semantics. Given a particular program, together with an annotated trace of a particular execution of it, our task is to support or refute a claim that the trace is an accurate account of a successful execution of the given program. Annotation includes a record of causal dependencies, data flow, and timing. The most direct application of our theory might be in the specification of a program testing tool, which has to deal with the awkward features of modern concurrent programming. An important part of the specification is that the tool must ignore all the successful executions, and report on all the others.

Because the given trace is assumed to be complete, successful, and well annotated, our analytic approach is inferior to other essential and more familiar modes of presentation of program semantics in at least five respects. (1) Because the initial trace is complete, our approach does not inhibit definitions that violate compositionality; (2) the trace records all atomic events in program execution at the lowest possible level of abstraction; (3) our approach is not constructive, and gives little guidance on implementation, its efficiency, or even its computability; (4) we give no hint how to debug an incorrect program; (5) it is hard to know when (and in what sense) the description is complete. A denotational semantics addresses the first two problems, and an operational semantics addresses the next two, and both of them solve the last problem. The main justification of our approach is that it models individual programming language features separately, and independent the design of the language as a whole.

Among the possibly unfamiliar features of our presentation are: (1) there is no commitment to a uniform format or style of definition, and many notions and notations of ordinary mathematics are pressed into service; (2) we do not decide in advance what the entire set of features of our programming language will be; (3) our initial basic theorems are proved quite simply from the earliest definitions; (4) new

primitives (like values, resources and time) are phased into the model gradually, and in a coherent sequence; (5) earlier primitives are defined in terms of those introduced later, rather than the other way round; (6) early proofs and theorems remain valid and useful, even after introduction of the new primitives; (7) there is no problem of freshness, because all resources actually used in the execution are already present in the trace.

The next section represents a trace as a graph, with each node representing a single execution of an atomic command, and each arrow representing a dependency between them. We define four degrees of separation (absence of dependency) between the events in the graph. Any given complete trace (including the empty one!) can be analysed into two sub-traces; all the events of each one of them are separate in some degree from all the events the other. A program fragment is modelled as a predicate describing an arbitrary sub-trace of its complete and successful execution. Program combinations are defined as separating conjunctions, whose degree of separation indicates possibility of implementation by sequential, concurrent, parallel or alternative execution.

The following section uses the simple model to provide trivial proofs of the standard algebra of programming. We then define the Hoare triple, and prove the familiar laws of the calculus of assertional correctness of programs. Concurrent program correctness is treated by Jones quintuples, and the soundness of the rely/guarantee calculus is proved.

Separation logic is the logic of predicates which (implicitly or explicitly) declare their own alphabet of free variables. We use it to make assertions about the values communicated by the arrows which cross the internal interfaces between the modules of the same program.

Separation logic is used to define the atomic actions from which programs are constructed. Features covered include: assignment, subscripted assignment, input, output, dynamic allocation of memory and channels, and the sharing of these resources by semaphores or by explicit transmission of ownership. Among the features omitted from our treatment are higher level features such as classes, inheritance, types, exceptions, unstructured fork/join, etc.

Different kinds of communication channel are distinguished by their synchronisation properties, including unreliable, lossy, re-ordering, multiplexed, buffered and synchronised channels. The same properties also distinguish various kinds of relaxed memory model used in modern multi-core processors.

So far we have not needed to postulate that our original dependency relation should be acyclic. Violation of this principle in practice results in the phenomenon of deadlock deadlock, which is a symptom of serious failure of program execution. We suggest this problem as a challenge for future research. Meanwhile, our results can be used to reason only about conditional correctness of programs.

References

1. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall; 1985. `http://www.usingcsp.com/` – Chapters 1 to 3 (at most).
2. C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall; 1998. Chapters 1 to 3 (at most).