Resources, Concurrency, and Local Reasoning (Abstract)

Peter W. O'Hearn

Queen Mary, University of London

In the 1960s Dijkstra suggested that, in order to limit the complexity of potential process interactions, concurrent programs should be designed so that different processes behave independently, except at rare moments of synchronization [3]. Then, in the 1970s Hoare and Brinch Hansen argued that debugging and reasoning about concurrent programs could be considerably simplified using compiler-enforceable syntactic constraints that preclude interference [4,1]; scope restrictions were described which had the effect that all process interaction was mediated by a critical region or monitor. Based on such restrictions Hoare described proof rules for shared-variable concurrency that were beautifully modular [4]: one could reason locally about a process, and simple syntactic checks ensured that no other process could tamper with its state in a way that invalidated the local reasoning.

The major problem with Hoare and Brinch Hansen's proposal is that its scope-based approach to resource separation is too restrictive for many realistic programs. It does not apply to flexible but unstructured constructs such as semaphores, and the syntactic checks it relies on are insufficient to rule out interference in the presence of pointers and aliasing. Proof methods were subsequently developed which allow reasoning in the presence of interference [9,10,5], and the reasoning that they support is much more powerful that that of [4], but also much less modular.

There is thus a mismatch, between the intuitive basis of concurrent programming with resources, where separation remains a vital design idea, and formal techniques for reasoning about such programs, where methods based on separation are severely limited. The purpose of this work is to revisit these issues, using the recent formalism of separation logic [11]. The general point is that by using a logic of resources [7] rather than syntactic constraints we can overcome the limitations of scope-based approaches, while retaining their modular character. We describe a variation on the proof rules of Hoare for contitional critical regions, using the "separating conjunction" connective to preclude pointer-based interference. With the modified rules we can at once handle many examples where a linked data structure rather than, say, an array is used within a process, or within a data abstraction that mediates interprocess interaction.

The rules have a further interesting effect when a data abstraction keeps track of pointers as part of its data, rather than just as part of its implementation. The separating conjunction allows us to partition memory in a dynamically reconfigurable way, extending the static partioning done by critical regions or monitors when there is no heap. This enables us to handle a number of subtler programs, where a pointer is transferred from one process to another, or between a process and a monitor, and the ownership of the storage pointed to transfers with it. Ownership transfer is common in systems programs. For example, interaction with a memory manager results in pieces of storage transferring between the manager and its clients as allocation and deallocation operations are performed [8]. Another typical example is efficient message passing, where a pointer is transferred from one process to another in order to avoid copying large pieces of data.

Dynamic partitioning turns out also to be the key to treating lower level, unstructured constructs which do not use explicit critical regions. In particular, our formalism supports a view of semaphores as ownership transformers, that (logically) release and seize portions of storage in addition to their (operational) behaviour as counting-based synchronizers. Local reasoning [6] is possible in such a situation because dynamic, non scope-based, uses of semaphores to protect resources are matched by the dynamic, non scope-based, approach to resource separation provided by separation logic.

A special role in this work is played by "precise" assertions, which are ones that unambiguously specify a portion of storage (an assertion is precise if, for any given heap, there is at most one subheap that satisfies it). Precision is essential for the soundness of the proof rules, which work by decomposing the state in a system into that owned by various processes and resources (or monitors). Precise assertions fulfill a similar role in recent work on information hiding [8], and are used by Brookes in his semantic analysis of our concurrency proof rules [2].

References

[1] P. Brinch Hansen. *Operating System Principles.* Prentice Hall, 1973.

[2] S. D. Brookes. A semantics for concurrent separation logic. Draft of 7/25/03, 2003.

[3] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[4] C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques.* Academic Press, 1972.

[5] C. B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.

[6] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, LNCS, pages 1–19. Springer-Verlag, 2001.

[7] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

[8] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268–280, Venice, January 2004.

[9] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, (19):319–340, 1976.

[10] A. Pnueli. The temporal semantics of concurrent programs. Theoretical Computer Science, 13(1), 45–60, 1981.

[11] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, LICS'02, 2002.