Aaron Turon — Research Statement

My research lies broadly in the areas of programming languages and verification, with the goal of building reliable software systems. I am drawn to software components that are usually characterized by "ugly" or "subtle" code—places where clean code and confidence of correctness take a back seat to performance. I aim to (1) discover new abstractions for cleanly expressing such code without compromising performance and (2) develop verification techniques to explain *why* subtle code works.

For the past several years I have targeted *concurrent* programming, where multiple computations (threads) overlap in time. To write reliable concurrent code, a programmer must consider all of the possible interactions between threads whose relative timing is unpredictable. The traditional way to tame concurrency is to disallow it: "critical sections" of code can be protected by locks, preventing interference from concurrent threads. But if *parallelism* is a goal—and it increasingly is—critical sections become critical bottlenecks that limit parallel speedup. Unfortunately, alternatives like lock-free synchronization are considered a "black art" even among concurrency experts.

My work focuses on ***identifying, harnessing, specifying and verifying abstractions in concurrent programming***, from low-level systems code to application programs. For example, I have systematically studied lock-free algorithms, showing how to (1) construct them from abstract building blocks; (2) apply them to parallel programming while guaranteeing determinism; (3) specify them with a clear contract for their clients; and (4) verify them with visual protocols embodying their key ideas.

Below I discuss my work in more detail, first from the perspective of language and library design (§1), and then from the perspective of verification (§2). I then discuss the most important problems I intend to study next (§3).

To keep focused, this statement does not discuss my earlier work on *metaprogramming* [9, 10], *parsing* [8, 11], *termination analysis* [7], or the *π-calculus* [17].

1 Identifying and harnessing concurrency abstractions

▸ **LVars: quasi-deterministic parallel programming.** Flexible parallelism requires tasks to be scheduled dynamically, in response to the vagaries of a workload. But if the resulting schedule nondeterminism is observable within a program, it becomes much more difficult to discover and correct bugs by testing: given the same input, the program may *usually* produce the correct answer, but *occasionally* produce the wrong one. ***Can we support flexible parallelism while guaranteeing determinism?***

Previous approaches guarantee determinism by severely limiting the lines of communication between threads. My work in POPL'14 [6], by contrast, develops a programming model based on "lattice variables" (LVars [5]), which allow threads to communicate through *any shared data structure*, so long as the contents of the data structure can be understood to be monotonically increasing within some lattice. The programming model guarantees a new property called *quasi*-determinism, which says that for a given input, a program can only produce a single answer; it may, however, halt with an *error* on some runs. The error case *always* designates a programming mistake, and the error pinpoints the offending code. Quasi-determinism provides most of the programming benefits of full determinism, but enables much greater flexibility in the programming model. LVars demonstrate good parallel speedup on a range of applications, including static analysis and computational biology.

▸ **Reagents: building blocks for fine-grained concurrency.** Parallelism mandates *fine-grained* concurrent data structures, in which coordination between threads is as local as possible, so that threads can work on independent parts of the data structure without any coordination at all. The lack of global coordination makes these data structures very difficult to develop, so even high-profile libraries like java.util.concurrent offer only a conservative selection of them. ***Can we build abstractions that ease the development of new concurrent data structures?***

My work in PLDI'12 [12] introduces a new abstraction, *reagents*, that provides several building blocks for fine-grained concurrency. Reagents interact either through small atomic updates to shared state or by passing messages, and they can be combined in several ways, including "conjunction" (forming larger atomic blocks) and "disjunction" (taking the choice between one of two atomic actions). Surprisingly, these simple building blocks can faithfully express some of the most sophisticated patterns of fine-grained concurrency—without loss of performance. Clients of a library can then extend and tailor it to their needs, without knowing the underlying algorithms, by using reagent combiners.

- **Join patterns: declarative, scalable synchronization.** Another fundamental tool in concurrent programming is *synchronization*, in which some threads wait until other threads have reached a particular state. In practice, it is paramount that synchronization minimize memory bus traffic and superfluous waiting—requirements that have led to a proliferation of specialized synchronization primitives. For example, java.util.concurrent contains locks, barriers, semaphores, count-down latches, condition variables, exchangers and futures, all carefully and separately engineered. Inevitably, though, programmers face new problems not directly addressed by existing primitives, forcing them to somehow combine those primitives into a solution. Doing so correctly and efficiently can be as difficult as designing a new primitive. ***Can we provide generic yet efficient synchronization directly to programmers?***

  My work in OOPSLA'11 [14] shows how to efficiently implement *join patterns* [1] as a basis for declarative and scalable synchronization. By *declarative*, I mean that the programmer needs only to write down the constraints of a synchronization problem, and the implementation will automatically derive a correct solution. By *scalable*, I mean that the derived solutions deliver robust performance with increasing processor count and problem complexity. I used join patterns to derive solutions to numerous common synchronization problems, in each case requiring only a few lines of code and no concurrency expertise. The derived solutions are competitive with—and often outperform—*specialized* algorithms from the literature, such as those implemented in Microsoft's .NET concurrency library.

2 Specifying and verifying concurrency abstractions

- **CaReSL: a logic for fine-grained concurrency.** Fine-grained concurrency abandons global locks in a quest for greater parallelism, with profound implications for algorithm design: it is no longer possible to acquire and manipulate a consistent global view of data. Instead, threads must work *locally*, observing and modifying small parts of a data structure while maintaining consistency at all times. While these techniques promote parallelism (and thus improve performance), they should in principle be invisible to the *clients* of a data structure. ***Can we find specifications for fine-grained data structures with simple client-side guarantees?***

  The traditional way to specify concurrent data structures is via *linearizability* [3], a property defined in terms of shuffling interaction histories which is not immediately useful for client code. My work in POPL'11 [18] proposes instead to directly prove a property called *contextual refinement*, and gives, for the first time, a logic for doing so. Contextual refinement says that the behavior of a fine-grained data structure is indistinguishable from its coarse-grained (global lock-based) counterpart. Clients can therefore benefit from fine-grained efficiency, while reasoning with coarse-grained simplicity.

  While this work opened the door to direct verification of refinement, the logic it gave handles only relatively simple data structures, leading to a deeper question: ***Can we verify refinement for sophisticated fine-grained concurrent algorithms?***

  My work in POPL'13 [15] and ICFP'13 [13] gives a systematic answer: rather than thinking of a single invariant governing an entire data structure, we should understand fine-grained concurrency on its own terms. That is, we should think of each *piece* of a data structure (*e.g.*, each node of a linked list) as being subject to a separate protocol governing thread interaction, thereby capturing the fine-grained observations and changes that threads make. These protocols also account for algorithms that employ "helping", where threads do work on each other's behalf to save on communication costs. I developed a program logic, *CaReSL*, for verifying refinement using protocols, and applied it to several challenging algorithms, including data structures with "higher-order features" like concurrent iterators. I also used CaReSL to give the first verification of the recent *flat combining* [2] construction.

- **GPS: navigating weak memory consistency.** Modern CPUs and compilers routinely execute code out of order for the sake of performance; for example, most CPUs employ write buffers to mask memory latency. Such critical optimizations are expected, of course, to respect the semantics—of *sequential* code. For concurrent code, they have a visible effect: they destroy the illusion of a single memory shared between all threads, leading to *weakly consistent memory models*. Weak memory complicates concurrency reasoning while rendering impotent our most effective weaponry: the sophisticated logics built to tame concurrency almost universally assume a strong memory model. ***Can we retain our advances in concurrency verification when our basic assumptions about memory must change?***

My recent work (under submission [16]) develops *GPS*, the first program logic to provide a full-fledged suite of modern verification techniques (including ghost state, protocols, and separation logic) in the context of weakly consistent memory. The key insight of GPS is the recognition that weak memory models provide strong guarantees for *individual* locations of memory. Thus, by taking the ideas of my work on CaReSL to their limit—constructing protocols that govern single memory locations—we can recover strong reasoning principles. I have used GPS to verify several challenging examples drawn from the Linux kernel, as well as classic lock-free data structures, all under the weak memory model provided by the recent C11 standard.

3 Future directions and open problems

▸ **Impedance mismatches in concurrency.** In §1, I described three concurrency abstractions—LVars, reagents, and join patterns—each of which captures a particular style of concurrent interaction. These abstractions join a host of others, including "classics" like monitors, and more recent proposals such as software transactional memory and Concurrent ML's composable events. Of course, large concurrent systems rely on a variety of modes of interaction between threads, so if these abstractions are to scale, they must be able to work together. ***Can we safely combine different concurrency abstractions? Can we retain their benefits when doing so?***

Some particularly desirable combinations include (1) mixing reagents and monitors—in other words, lock-free and lock-based programming and (2) mixing LVars and DPJ-style deterministic parallelism, where the latter allows threads to *arbitrarily* mutate data structures but imposes restrictions on *when* this can be done through a static type system. In the first case, the combination will lose some of the liveness properties of lock-free code, but should still ensure atomicity; in the second, one would hope to use the full range of LVar and DPJ-style communication while guaranteeing quasi-determinism. On the other hand, some combinations are out of reach: my dissertation shows that "strong" software transactions are fundamentally incompatible with lock-free data structures that use helping (mentioned in §2).

I plan to thoroughly explore the space of interactions between important concurrency abstractions.

▸ **Beyond Hoare triples, beyond refinement: specifying concurrency abstractions.** Built-in language features have always had a privileged status in program logics, which provide specialized rules capturing their semantics. This special treatment is especially important in logics for concurrency, which make a fundamental distinction between the "primitive" atomic commands of a language (*e.g.* low-level synchronization operators) and everything else. All of this ignores, of course, the possibility of programmer-created abstractions, which include all of the abstractions discussed in the previous two sections. To take a concrete example, lock-free data structures provide operations that *appear* atomic, even though they are actually a composition of primitive atomic commands—a fact that prevents them from being treated as atomic in concurrency logics. ***Can we build logics with first-class support for abstractions built within a program?***

I have already done some work toward addressing this problem. For example, my work on refinement in CaReSL (§2) allows "observably" atomic abstractions to be used as if they were primitive atomic commands, but only in a somewhat roundabout way that uses *code* as a specification. In addition, my work in ICFP'12 [4] shows how the special treatment of the heap in separation logic can be applied to programmer-defined data structures like hashtables, which are intuitively heap-like.

But these steps are only the beginning. I hope to develop logics with an *extensible* notion of atomicity that can include programmer abstractions. Furthermore, many concurrent data structures support operations that are *not* atomic, but instead permit *e.g.* traversal of data even as it changes concurrently. At present, it is unclear how to give specifications for these operations; indeed, it seems unlikely that traditional Hoare logic, or even refinement, *can* adequately specify them. Another pressing problem is specifying concurrency abstractions in the context of weak memory: what are appropriate guarantees? Should the memory model itself be extensible with new programmer abstractions?

More generally, I plan to build logics that can encompass entire towers of abstraction, and thus to tie together the various strands of my work. For example, it should be possible to both verify an implementation of reagents and verify data structures built on top of reagents *in a single logic*, while allowing the latter verification to treat reagents as if they were a built-in abstraction.

References

[1] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL*, 1996.

[2] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[3] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[4] Neelakantan R. Krishnaswami, **Aaron Turon**, Derek Dreyer, and Deepak Garg. Superficially substrucural types. In *ICFP*, 2013.

[5] Lindsey Kuper and Ryan R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *FHPC*, 2013.

[6] Lindsey Kuper, **Aaron Turon**, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: quasi-deterministic parallel programming with LVars. In *POPL*, 2014.

[7] Panagiotis Manolios and **Aaron Turon**. All-Termination(T). In *TACAS*, 2009.

[8] Scott Owens, John Reppy, and **Aaron Turon**. Regular expression derivatives reexamined. *Journal of Functional Programming*, 19:173–190, March 2009.

[9] John Reppy and **Aaron Turon**. A foundation for trait-based metaprogramming. In *FOOL/WOOD*, 2006.

[10] John Reppy and **Aaron Turon**. Metaprogramming with traits. In *ECOOP*, 2007.

[11] Olin Shivers and **Aaron Turon**. Modular rollback through control logging. In *ICFP*, 2011.

[12] **Aaron Turon**. Reagents: expressing and composing fine-grained concurrency. In *PLDI*, 2012.

[13] **Aaron Turon**, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.

[14] **Aaron Turon** and Claudio V. Russo. Scalable Join Patterns. In *OOPSLA*, 2011.

[15] **Aaron Turon**, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.

[16] **Aaron Turon**, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation (under submission), 2014.

[17] **Aaron Turon** and Mitchell Wand. A resource analysis of the pi-calculus. In *MFPS*, 2011.

[18] **Aaron Turon** and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL*, 2011.