Interview with Stephen Brookes and Peter W. O'Hearn Recipients of the 2016 Gödel Prize

Luca Aceto
ICE-TCS, School of Computer Science,
Reykjavik University
luca@ru.is

Stephen Brookes (Carnegie Mellon University, USA) and Peter W. O'Hearn (Facebook and University College London, UK) are the recipients of the 2016 Gödel Prize for their invention of Concurrent Separation Logic, as described in the following two papers:

- S. Brookes, A Semantics for Concurrent Separation Logic. Theoretical Computer Science 375(1–3):227–270 (2007) and
- P. W. O'Hearn, Resources, Concurrency, and Local Reasoning. Theoretical Computer Science 375(1–3):271–307 (2007).

Quoting from the citation for the prize:

> "Concurrent Separation Logic (CSL) is a revolutionary advance over previous proof systems for verifying properties of systems software, which commonly involve both pointer manipulation and shared-memory concurrency. For the last thirty years experts have regarded pointer manipulation as an unsolved challenge for program verification and shared-memory concurrency as an even greater challenge. Now, thanks to CSL, both of these problems have been elegantly and efficiently solved; and they have the same solution."

As highlighted in the citation by the 2016 Gödel Prize committee, the impact of CSL has been extraordinary, both from a theoretical and practical viewpoint. CSL has led to a wealth of follow-up research over the last decade and "its simplicity and structure also facilitates automation. As a result numerous tools and techniques in the research community are based on it and it is attracting attention in companies such as Microsoft, Facebook and Amazon."

In order to celebrate the award of the Gödel prize to this path-breaking work and to allow the research community in theoretical computer science at large to appreciate the origin of the ideas that led to their invention of Concurrent Separation Logic, I interviewed Stephen Brookes (abbreviated to SB in what follows) and Peter W. O'Hearn (referred to as PO in the text below) via email. I hope that the readers of the Bulletin of the EATCS will enjoy reading the text of the interview as much as I did.

1 The interview

LA: You are receiving the Gödel Prize 2016 for your invention of Concurrent Separation Logic (CSL), which, quoting from the prize citation, is "a revolutionary advance over previous proof systems for verifying properties of systems software, which commonly involve both pointer manipulation and shared-memory concurrency." Could you briefly describe the history of the ideas that led to the invention of CSL, what were the main inspirations and motivations for its invention and how CSL advanced the state of the art?

PO: After John Reynolds and I and others had done the initial work on separation logic for sequential programs it made sense to consider concurrency, just based on the idea of using the logic to keep track of the separation of resources used by different processes or threads. In the summer of 2001 I devised initial proof rules to do just that, by adapting an approach of Hoare to reasoning about concurrency from Hoare logic to separation logic.

This first step seemed straightforward enough, but then it hit me that we could use the logic to track dynamically changing partitions rather than the static partitioning in Hoare's approach. I used a little program, the pointer-transferring buffer, to explore this idea, and I made a program proof in which the fact that something was allocated seemed to move from one process to another... during the proof. The pointer itself was copied, but the fact that it was allocated (and hence the right to dereference it) was given up by the sending process. I began talking about "knowledge" or "ownership" transferring from one process to another. The striking thing was that there was no explicit concept of ownership in the logic or proof rules, but that this transfer seemed to be encoded in the way that the proofs of the processes worked. I circulated an unpublished note on proving the pointer-transferring buffer in August of 2001, and then a longer note in January 2002; these documents got a lot of attention from people working on separation logic.

It quickly became clear that quite a few concurrent programs would have much simpler proofs than before. Modular proofs were provided of semaphore programs, of a toy memory manager, and programs with interacting resources. I got

up a huge head of steam because it seemed as if the logic could explain the way that synchronisation had been used in the fundamental works on concurrent programming by Dijkstra, Hoare and Brinch Hansen. For example, in the paper that essentially founded concurrent programming, Dijkstra in 1965 (*Co-operating Sequential Processes*, still the most important paper in concurrency) had explained that the point of synchronisation was to enable programmers to avoid minute considerations of timing, to simplify reasoning. Brinch Hansen had hammered into me the importance of speed independence and resource separation for simplifying thinking about concurrent processes when I was his colleague at Syracuse in the 1990s, and what he said to me seemed to be mirrored in the proofs in this early concurrent separation logic. And although I had never written a paper on concurrency, I was opinionated: I thought that work in the theory of concurrency was often missing the mark because while it could describe semaphores and other synchronisation primitives, it did not explain their significance because it did not connect back to simplifying reasoning; which was their whole point. Hoare had made important steps in various points in his work, but it seemed as if we could go much further armed with the separating conjunction.

So, years of thinking about these issues seemed poised to come together at once in this concurrent separation logic. But, for all my opinionated excitement, I ran into a blocker of a problem: I wasn't able to prove soundness of my proof rules. And it was the very feature which gave rise to the unexpected power, the ownership or knowledge transfer, that made soundness non-obvious. I worked hard on this problem for several months in the second half of 2001 and early in 2002, and got nowhere. Reynolds, Yang, and Calcagno, all semantics experts, also looked at the problem. Finally, I admitted to myself that it was technically beyond my expertise in concurrency theory and that I needed help. Luckily, I knew where to turn. Steve Brookes and I had never worked together before, but knew one another from way back. He was the external examiner of my 1991 PhD thesis, I was well aware of his fundamental work with Hoare and Roscoe on the foundations of CSP, and he had recently produced striking results on full abstraction for shared-memory parallelism, what I though of as the most impressive theoretical results in semantics of concurrency at the time. What is more, Steve had built up a powerful repertoire of techniques for proving properties of concurrent programming languages. So, sometime in 2002, I picked up the phone and gave him a call: "Steve, I have a problem!".

SB: As Peter said, he called me out of the blue. I knew Peter well, having served as the external examiner on his PhD thesis, and I had kept in contact with him after he took up his first academic positions at Syracuse and at Queen Mary (University of London). We were in fairly regular email contact, but he didn't normally phone me from England; I knew this must be important. I sat in my office at Carnegie

Mellon University in Pittsburgh, intrigued and fascinated to hear his ideas and excited by the challenge he was offering me. I think he also spoke by phone at a different time to John Reynolds, whose office was right next to mine. We all agreed that the best plan was for Peter to come to Pittsburgh and give a talk, after which we would do some brainstorming. That was how it started, from my viewpoint. Peter came to CMU in March of 2002 and gave a talk on his new logic. Peter was proposing a subtle and clever combination of key ideas from separation logic and a much earlier Owicki-Gries logic for shared-memory concurrency, itself based on ideas appearing in a classic paper of Tony Hoare (*Towards a theory of parallel programming*). Superficially the new logic was both very simple and also very perplexing. The Hoare-style logic has a simple rule for parallel composition that combines pre- and post-conditions using conventional conjunction, and a rule for conditional critical regions (essentially, binary semaphores) that makes use of "resource invariants". The inference rules ensure that a provable program obeys what has come to be known as a "rely-guarantee" discipline: each process assumes that whenever it acquires a semaphore the relevant invariant holds, and guarantees that the invariant holds again when releasing the semaphore. However, the earlier logic is not sound for programs using pointers, because of the possibility of aliasing. On the other hand separation logic was originally formulated for reasoning about sequential programs using pointers, but lacked rules for shared-memory concurrency and (until then) it was not clear how to generalize. Peter introduced a radically simple and elegant way to combine the two: an overly simplistic summary is that he allows separation logic formulas as pre- and post-conditions (and resource invariants) and strategically replaces certain occurrences of conjunction in the Hoare-Owicki-Gries rules with the separating conjunction operator from separation logic. Of course this naive characterization glosses over some profound issues: in Peter's approach the invariants and pre- and post-conditions are not really talking about the "global" state, but serve to specify the "footprint" of a program component, just the piece of state on which that component acts. I think this was the first time I saw use of terms such as "footprint" and "ownership transfer", notions which now seem very intuitive but as yet had no formal semantic counterpart. (Until this point, I think it is fair to say, semantic models for concurrent languages had typically dealt entirely with global state, and it was conventional wisdom that it was already hard enough to find a decent semantics for simple shared-memory programs, let alone try to incorporate mutable state, heap cells, allocation and deallocation.) My challenge was to develop a semantic model robust and flexible enough to handle the combination of concurrency and pointers, in which such notions as ownership transfer could be properly formalized: to build a foundation on which to establish soundness of Peter's proposal. Furthermore, prior semantic models for concurrent languages had pretty much ignored race conditions, typically by assuming that assignments

were executed atomically so that races never happen. While that worked well for "simple" shared-memory it was clearly an insufficiently sophisticated way to cope with mutable state and the more localized view of state that is so fundamental in Peter's approach.

During this visit we basically barricaded ourselves in a room with Reynolds and Calcagno, dropping all other distractions and tossing ideas around in an attempt to lay out a groundplan, exploring the apparent benefits of the new logic while probing for limitations and possibly even counterexamples. John's wife Mary recalls very intense discussions at the Reynolds's household, walls covered with sticky paper, only breaking for meals. Peter enunciated what became known as the Separation Principle: "at all stages, the global heap is partitioned into disjoint parts…", another way to articulate the idea of "ownership transfer". Again it is easy to express these notions informally, but it turned out to be surprisingly tricky to encapsulate them semantically, and Peter was right to be cautious.

I remember marveling at the ease with which Peter was able to articulate, with simple-looking little programs like a one-place buffer, the kinds of problem that arise when concurrent threads manipulate the heap. And the logic seemed elegantly suited for reasoning about the correctness of such programs, with the decided advantage that provable programs are guaranteed to be free of data races. (This was also true of the earlier Hoare-Owicki-Gries logic, but only in the much more limited setting of "simple" shared-memory without pointers.)

The use of separation in the logic neatly embodies a kind of disciplined use of resources in programs. Yet it was by no means clear how to formalize these ideas, and Peter was quite forthright about his unwillingness to publicize the ideas until it had been demonstrated that it all made sense. He had circulated an "unpublished manuscript" with the title *Notes on separation logic for shared-variable concurrency*, dated January 2002. Meanwhile I was excited to have a challenging problem to work on, and intense interactions continued between Peter, John and me as our understanding evolved.

I had plenty of experience in developing denotational semantic models for (simple) shared-memory programs and also for communication-based languages such as CSP. Just like the earliest denotational accounts of concurrency (dating back to David Park in the late 70's) the most widespread and generally most adaptable approach was to use traces (or sequences of actions) of some kind; I had introduced "transition traces", built from steps that represent (finite sequences of) atomic operations by a process, with gaps allowing for state changes made by the "environment". I think that transition trace semantics is what Peter was referring to, when he mentioned full abstraction, but I should emphasize that this model was tailored specifically to simple shared-memory and I could not see an obvious way to adapt it to incorporate pointers. I did try! My more recent focus had been on "action traces", in which steps represent atomic actions but the details con-

cerning state (when an action is enabled, and what state change it causes) are kept apart: a process denotes a set of action traces, and one can then plug in a model of state, give an interpretation of actions as state transformers, and be able to reason rigorously about program execution. It seemed to me that action traces offered the best basis for expansion to incorporate pointers: by augmenting the "alphabet" of actions to include heap look-up, update, allocation and deallocation. It should also be quite natural to treat semaphore operations (for acquiring and releasing) as actions. It took until some time in 2003 for me to iron out the technical details and be able to explain the key definitions and results clearly enough to be ready for publication.

In retrospect I regard this period of a couple of years as probably the most exciting and stimulating sustained research in which I have been involved. I am immensely grateful to have had the opportunity to work with Peter on this project. And throughout all of this I was strongly influenced by John Reynolds, whose good taste, deep originality of thought and sheer intellectual inquisitiveness served to keep me focussed. Echoing Peter's reluctance to publish without being sure, I always said to myself that I wouldn't be sure until I could convince John (and Peter!). John and I had lunch together almost every day, and I camped out in his office whenever I had an idea that needed a sounding board. He prompted me to always seek clearer explanations, isolate the key concepts and make the right definitions, and strive for generality. I remember in particular that at one point, after several weeks trying to figure out how to extend action traces, I told John that I might be able to give a semantics in which (only) provable programs would be definable. My naive idea was to modify the way that parallel composition gets modelled to keep track of the preservation of resource invariants explicitly and treat any violation as a "disaster". I recall John upbraiding me gently about this plan; in his view, one should develop a semantics that is "agnostic" about any intended logic, and instead built to express computational properties of programs in general terms. He did agree that we needed a semantics that reported the potential for race conditions, an idea that is echoed in his own work. Having found such a semantics, it ought then to be possible to show using the semantics that programs provable in the logic are indeed race-free. In essence, that's what happened. But it didn't happen overnight, and the path from start to end involved a few detours and dead-ends before finding a robust solution.

There was a crucial role in this played by the concept of "precise" assertion. I will hand back to Peter to make a few remarks on that.

PO: Yes, a memorable moment came when John poked his head into my office one day: "want some bad news?". He showed me an example where the proof rule for critical regions together with the usual Hoare proof rule for using conjunction in the postcondition lead to an inconsistency. This problem had nothing to do

with concurrency per se, but rather was about the potential indeterminacy of the "angel" affecting ownership transfer. (We had been attempting to get a handle on ownership transfer by couching it in adversarial terms, involving an "angel" making program choices and a "demon" trying to invalidate invariants or induce race conditions.) The problem also arose in a sequential setting, in proof rules for modules that I was working on with John and Yang (which we eventually published in *Separation and Information Hiding*, in POPL'04). In any case, I quickly proposed a concept of "precise" assertion, one that unambiguously picks out an area of storage, as a way to get around the problems cause by the indeterminacy of the angel, and this concept is used in the resource invariants in concurrent separation logic.

This indicates what a subtle problem we were up against. Some experienced semantics researchers had been working on soundness of CSL and of information hiding and there lurked a problem that none of us had spotted. We were lucky that John Reynolds was not only keeping us honest by tracking our progress, but thinking deeply about these problems himself. Concurrent separation logic owes a lot to John's brilliance.

That being said, this problem with the ownership angel was not the biggest hurdle Steve had to get over. Setting up the concurrency semantics and identifying the right properties to prove to get inductions to go through was just hard. Once he had done it, others were able to generalize and sometimes (arguably) simplify his proof method; e.g., in work of Calcagno, Yang, Gotsman, Parkinson, Vafeiadis, Nanevsky, Sergey and others, and some of it drops the precision requirement (but also the conjunction rule). But, from my perspective, Steve solving this problem for the first time was difficult, and important. I like to say: he saved my logic!

LA: I noticed that you each published short versions of your papers in the same conference, CONCUR'04, and then again the long versions appeared in the same journal issue in TCS'07. How did that come about?

SB: My email to Peter and John, announcing that I had found the "right" semantics and that the concurrency rules can be shown to be sound when resource invariants are chosen to be precise, dates from early June 2003. (I also showed that the rules remain sound when invariants are chosen to be "supported", so that in any state with a sub-heap satisfying an invariant there is a unique minimal sub-heap with that property.) This was obviously very exciting and I headed over to England to spend some time in London with Peter. Peter organized informal discussion meetings (called "yak sessions" to distinguish from full-blown seminars) and Peter and I both gave presentations there on CSL. Philippa Gardner witnessed these presentations, and she was so enthusiastic that she decided (as organizer for the CONCUR 2004 conference in the following year) to invite us to give back-to-back hour-long tutorials at that meeting. That's how the original short versions of

our papers ended up at the same conference. We felt honoured to be asked, effectively, to give tutorial invited lectures on as-yet unpublished work! And thanks to Philippa for her part in encouraging this scenario.

We published the full, journal-length versions together in TCS as tributes to John on his 70th birthday. The Festschrift volume appeared as a book, under the TCS imprimatur, edited by Peter along with Olivier Danvy and Phil Wadler, in 2007. There's an amusing story that Peter reminded me of. One day Philippa was walking with me, John and Peter, some time before the CONCUR short versions were finalized. John was berating Peter for what he perceived perhaps as dilatoriness in not submitting long versions to POPL or some other conference instead of preparing the journal-length versions; he couldn't understand why it was taking us so long to get around to it. What he did not realize was that the plans were already afoot for his birthday festschrift, and we had already agreed to publish there. After all, what better way to acknowledge the profound influence John had on the work. Philippa leaned over to Peter and whispered that we couldn't tell him because those (long) papers are for his "bloody festschrift".

LA: In your opinion and experience, how important are modularity and compositionality in proving properties of programs and systems?

SB: I think the benefits of modularity — in particular, allowing more "localized" reasoning by considering each component largely in isolation from the others — have been touted right from the early days. For example Dijkstra was in favor of "loose coupling", and argued that the art of parallel program design was to ensure that processes can be regarded almost as independent, except for the (ideally small number of) places where they synchronize. The claim was, and continues to be, that this would improve our ability to manage the sheer complexity caused by interactions between concurrent threads. For similar reasons, and dating back to Hoare logic (1969!), we seek compositional proof systems for proving program properties. In a compositional proof system one can derive correctness properties of a program by reasoning about program components individually, and the inference rules show how the properties of components determine the behaviour of the whole program. In short, compositional means "syntax-directed", and again a major desire is to exploit syntax-directed analysis to tame the combinatorial explosion. But to design a compositional logic for a specific programming language, for use in establishing a specific class of program behaviour, you need to start with a suitably chosen assertion language — one for which compositional reasoning like this is even possible. This is particularly difficult for concurrent programs, since it has long been known that Hoare-style partial correctness assertions about a multithreaded program cannot be deduced simply on the basis of partial correctness properties of individual threads. A partial correctness assertion of form $\{p\}\ c\ \{q\}$ says that every terminating execution of c from an initial state satisfying p ends in

a state satisfying q. In the early Hoare-style logics for shared-memory programs (Hoare-Owicki-Gries, as mentioned earlier) assertions look just like conventional partial correctness but are interpreted semantically as expressing a much more sophisticated property, allowing for the program to be running in an "environment" of other processes. It has become common to describe this interpretation in terms of "rely/guarantee". I think one of the key ingredients in my semantic model is that it allows formalization of the kind of ownership transfer discipline inherent in Peter's inference rules. Turning this on its head, you could say that the semantic model supports compositional reasoning about programs whose correctness is justified by appeal to Peter's separation principle and the ownership discipline. I'll let Peter step in here too, as I'm sure he feels strongly about compositionality as a virtue. Maybe he can say something about monitors as well, which I know were a strong motivating factor in how he came up with the inference rules.

PO: Generally speaking, when proving a program it is possible in principle to construct a global proof, one that talks about the global state of the system. But global proofs tend to be much harder to maintain when the program is changed. And programs are constantly being changed in the real world: the world won't accept prove-it-and-forget-it proof efforts, verification needs to be active and move with the programmers. This, even more than efficiency considerations in constructing proofs at the outset, is the strongest reason for wanting modularity.

Separation logic has just provided a theory that often matches the intuitive modularity that comes up in data structure designs. The degree of modularity in proofs that have been done has been surprising. For instance, when I was thinking about proof rules for CSL my first idea was to axiomatize monitors (class-like abstractions for concurrency due to Brinch Hansen and Hoare), because I thought that low-level primitives like semaphores were too unstructured and that modular proofs would be impossible. Of course I knew that monitors could simulate semaphores, but I didn't expect to find nice proofs of semaphore programs. It therefore came as a bit of a shock when I was able to provide very local independent reasoning about semaphores in some nontrivial examples.

But, while modularity is important, you should be careful not to try to take it too far. For instance, sometimes multiple resources participate together in the implementation of a data abstraction, like the use of several locks in hand-over-hand locking on linked lists. Vafeiadis and Parkinson have some lovely work on RGSep, a descendant of the original concurrent separation logic, in which they show how you can describe the effects of operations in a very local way, but where the description sometimes involves two locks and a bit of a linked list, rather than only one lock; you would just be causing yourself trouble if you tried to formulate everything in terms of independent reasoning about the individual locks in this case. So I like to think that you should make your specifications and

proofs as modular as is natural, but not more so; logic should not block making specifications and proofs modular, but neither should it force you to shoehorn your descriptions into a fixed granularity of modularity.

Some of the work that follows on from mine and Steve's work is very flexible in the degree of abstraction that can be given to the way that state is composed and decomposed. For instance, work of Nanevski, Sergey, Dreyer, Birkedal and others is all based on proof methods which allow the granularity of modularity or separation to be chosen, but specifying a partial commutative monoid of composition (in place of the standard separation logic monoid of heaplets and disjoint union).

LA: What are some of the main developments since your papers appeared.

PO: First let me mention developments in theory. To me, the most surprising has been the demonstration that the most basic principles of concurrent separation logic, particularly independent reasoning about threads using the separating conjunction, cover a much broader range of situations than we ever expected. There have been proofs of fine-grained locking and non-blocking concurrency and cases that involve interference and general graph structures, what might have been though of as bad cases originally for separation logic.

SB: We should also mention generalizations based on permissions (for example, Boyland's account of fractional permissions). In particular this path led to versions of CSL capable of dealing naturally with concurrent reads, and to some very elegant program proofs (due to Peter with Calcagno and Bornat) in which the correctness of the program depends on a permissive form of ownership transfer.

PO: Interestingly, the unexpected power of this is based on what you might call "non-standard models" of separation logic; I mean this by analogy with the usual situation in logic, where a theory (e.g. reals, or integers) has an intended model, but then additional non-standard models of the same axioms. The proof theory can then accomplish unexpected things when applied to the non-standard models. The standard model of separation logic is the original model based on splitting portions of the heap, or heaplets. There are lots of other models stemming from the "resource semantics" of bunched logic invented by David Pym (based on having a partial commutative monoid of possible worlds). The surprise is that some of these nonstandard models involve composing highly intertwined structures and interfering processes. Gardner coined the phrase "fiction of separation" to describe this phenomenon in the nonstandard models.

For example, in their POPL'13 work on Views, Dinsdale-Young, Parkinson and colleagues show that a simple abstract version of concurrent separation logic can embed many other techniques for reasoning about concurrency including type systems and even the classic rely-guarantee method, which was invented for the

purpose of reasoning about interference. Work of Nanevski and Sergey on their Fine-Grained Concurrent Separation Logic shows how one of the sources of non-modularity in the classic Owicki-Gries approach to concurrency, the treatment of auxiliary variables, can be addressed by a suitable non-standard model of separation. They also obtain great mileage out of a model that interprets the separating conjunction in terms of a composition of histories, thus combining temporal and spatial aspects of reasoning. Finally, Hoare and others have been pursuing a very general theory of "Concurrent Kleene Algebra", which encompasses message passing as well as shared variable concurrency, and its associated program logic is again a very general form of CSL.

Although they technically use simple abstract version of CSL, these works conceptually go well beyond the original because the nonstandard models have meanings so far removed from the standard models. And they represent technically significant advances as well. For instance, the Views work has a new and very flexible proof of soundness, which is needed I think to cover the concurrency in the nonstandard models. There is a lot happening in this space, and I have left out other very good work by Birkedal, Dreyer, Raad, Feng, Shao and others; a number of competing logics are being advanced, and there is just too much good work to mention it all here. It will take some time to see these developments shake out, but already it is clear that the principles of CSL apply much more broadly one could have guessed at the time of mine and Steve's papers.

Second, I would like to mention that a surprising practical development has been the degree of progress in tools for mechanized verification. The first implementation of CSL actually preceded the publication of mine and Steve's papers. Cristiano Calcagno included CSL in his earliest prototypes of Smallfoot, the first separation logic verification tool, around 2002. But, since then, many tools have appeared for verification with CSL and relatives. Examples of the state of the art include the Verifast tool of Jacobs et al. and the implementation in Coq of the aforementioned Fine-grained CSL: in both tools there are examples of mechanized proofs of nontrivial concurrent programs including hand-over-hand locking on linked lists, lock-free queues, and a concurrent spanning tree construction. It is also possible to verify custom synchronisation primitives, such as for locks implemented using compare-and-swap, and then to use the specifications of the primitives in verifications of client code without having to look at the lock internals.

LA: What are some of the current directions of interest, or future problems, for research.

PO: One important direction in pure theory is unification, to bring to a form of local conclusion all of the developments on non-standard models. I sense that there is good and possibly deep theoretical work to be done there.

SB: I agree. And I think the time is ripe for a systematic attempt to develop a denotational framework capable of supporting such a unification. My own recent research into the foundations of weak memory concurrency is an attempt to start in this direction, and seems like a natural generalization from the action trace semantics that we used to formalize CSL. The main idea here is to go from traces (essentially, linearly ordered sets of actions) to a more general partial-order setting. Similar themes are also appearing in work of others, and we are starting to see papers proposing the use of pomsets (my own work, building on early ideas of Vaughan Pratt) and event structures (originally introduced by Winskel) in semantic accounts of weak memory. I think this is exciting, and I believe it would be a valuable service to cast these developments into a uniform framework, and to use such a framework to establish soundness of new CSL-style logics for weak memory and explore the relationships between such logics.

PO: As I said above, there has been tremendous progress in mechanized verification of concurrent programs; but there has been less in automatic program analysis. With program analysis we would like to give programmers feedback without requiring annotations, say by trying to prove specific integrity properties (such as memory safety or race freedom); annotations can help the analysis along, but are not needed to get started, and this greatly eases broad deployment. A number of prototype concurrency analyses based on CSL have been developed, but there has been much more work applying sequential separation logic to program analysis. For example, the Infer program analyser, which is in production at Facebook, uses sequential but not concurrent separation logic. To make advanced program analysis for concurrency which brings value to programmers in the real world is in the main an open problem. And not an easy one.

I would finally like to mention language design. There have been experimental type systems which incorporate ideas from CSL into a programming language, such as the Mezzo language and Asynchronous Liquid Separation Types. Related ideas can be found earlier in Cyclone, and more recently in the ownership typing that happens in the Rust language. It seems as if there is a lot of room for experimentation and innovation in this space.

LA: Peter, Steve, thank you very much for sharing your recollections and knowledge with the theoretical-computer-science community, and congratulations for the 2016 Gödel Prize!