Jeehoon Kang

Seoul National University, Korea

Research Statement

September 11, 2018

My research is motivated by the goal of making **non-blocking concurrent programming easier**. Non-blocking (think: lock-free) concurrency is an essential ingredient for exploiting parallelism—which is becoming more and more important since the slowdown of Moore's law—but it is notoriously error-prone and difficult to get right. I have personally suffered from this difficulty in the course of maintaining the Crossbeam project [1], which serves as the *de facto* standard concurrency library for the Rust programming language [2]. Crossbeam provides an epoch-based semi-automatic garbage collector and, on top of the garbage collector, various production-ready non-blocking concurrent data structures which are deployed in the recent versions of Firefox. Developing an efficient and yet correct concurrency library was difficult: bugs are constantly popping up and some of them are even found in the released versions of the library, even though we followed the best practice such as using the C/C++ concurrency features and testing with the state-of-the-art tools.

My general strategy for facilitating non-blocking concurrent programming is to develop **formal methods** for reasoning about programs. In my experience of Crossbeam, concurrent programming is difficult primarily due to the fact that concurrency bugs are usually manifested only in a subtle and nondeterministic way so that they are not easily uncovered by auditing or testing. Therefore I believe that formal methods, despite their usually high cost, are actually cost-effective for implementing efficient and correct concurrent algorithms. In particular, I aim to develop **reasoning principles** on which programmers can *manually* reason about concurrent program's safety and functional correctness, and based on the principles, to develop precise **analysis/verification tools** that are capable of *automatically* proving such properties. If successful, my research will advocate "correct-by-construction" approaches by helping programmers to fix subtle concurrency bugs without much labor even before deploying the code, thereby having a profound impact on real-world concurrent and parallel programming.

The main challenge I face is that most prior work on principles and tools of concurrent programming makes big simplifying assumptions on the underlying programming model—for example, that target programs are completely synchronized with locks and are completely free from any races (and thus "blocking"), or that they are based on sequential consistency (SC) semantics. Unfortunately, these assumptions do not hold for real-world concurrent programs, which intentionally cause *benign* race conditions and are based on *relaxed consistency* semantics for better performance. Thus my research aims to build, **without making such unrealistic assumptions**, new theoretical and practical foundations for formal reasoning about non-blocking concurrency in the wild.

In the rest of this research statement, I will describe some of my previous and ongoing projects on developing such reasoning principles and tools for concurrent programming.

Promising semantics for relaxed consistency Non-blocking concurrent programs in the wild are usually written in low-level programming languages like C/C++ or even assembly languages. The defining characteristic of the concurrency features in low-level programming language and hardware concurrency is their *relaxed* nature, or in other words, that instructions may be executed in an out-of-order fashion—for example, in certain circumstances, loads and stores can be hoisted above the previous instructions. Thus I aim to develop formal methods that account for load and store hoisting as well as other concurrency features such as thread interleaving, coherence, and synchronization.

The problem is that, despite many years of research, it has been proven very difficult to even define the semantics of store hoisting in C/C++ and other programming languages that adequately balances the conflicting desiderata of programmers, compilers, and hardware. Obviously, without an established semantics, it is simply impossible to formally reason about programs! C/C++ carelessly allows a too broad class of store hoisting and ends up allowing certain bad program behaviors (which we call "out-of-thin-air" behaviors) that break fundamental properties of concurrency semantics programmers expect to hold, such as the data-race freedom (DRF) guarantees and the soundness of simple invariant-based reasoning. On the other hand, Java reluctantly allows a too narrow class of store hoisting and fails to validate essential compiler optimizations that are actually performed by Java HotSpot VM. As far as we know, none of many proposals in the past decades succeeded in defining a programming language semantics of store hoisting that satisfies programmers, compilers, and hardware at the same time.

In our **POPL 2017** paper [3], we proposed the first relaxed consistency model for C/C++—which we call the *promising semantics*—that (1) accounts for a broad spectrum of features from the C++11 concurrency model, (2) is implementable, in the sense that it provably validates many standard compiler optimizations and reorderings, as well as standard compilation schemes to x86-TSO and Power, (3) justifies simple invariant-based reasoning, thus demonstrating the absence of bad out-of-thin-air behaviors, (4) supports DRF guarantees, ensuring that programmers who use sufficient synchronization need not understand the full complexities of relaxed-memory semantics, and (5) defines the semantics of racy programs without relying on undefined behaviors, which is a prerequisite for applicability to type-safe languages like Java.

The key novel idea behind our model is the notion of *promises* and *certification*: a thread may promise to execute a write in the future, thus enabling other threads to read from that write out of order. Crucially, to prevent out-of-thin-air behaviors, a promise step requires a thread-local certification that it will be possible to execute the promised write even in the absence of the promise. We demonstrated that the notion of promises and certification is not limited to modeling store hoisting in C/C++ but is actually universal for relaxed consistency semantics in general: in a submitted paper [4], we propose a promising semantics for hardware that models store hoisting in ARMv8 and RISC-V in a much simpler and efficiently explorable way than the previously proposed models.

To establish confidence in our promising semantics for C/C++ and ARMv8/RISC-V, we have formalized most of our key results in the Coq proof assistant [5], which rigorously checked that all our proofs are valid. In doing so, much to our surprise, we uncovered a severe flaw in the *official* C/C++ semantics: in contrary to published results [6, 7], the standard mapping from C/C++ SC atomics to Power processors used in mainstream compilers is unsound. We subsequently proposed a fix to the semantics of C/C++ SC atomics in our **PLDI 2017** paper [8], but we left its application to our C/C++ promising semantics as a future work.

We recently discovered that the standard mapping from C/C++ read-modify-write instructions to ARMv8 processors used in mainstream compilers is unsound for our C/C++ promising semantics. Roughly speaking, the specification of ARMv8 basically allows—while not observable on actual hardware—certain *cooperative* out-of-order execution among processor cores that happens to bring about certain *global* behaviors, which were beyond our imagination when designing the C/C++ promising semantics. These global behaviors are unique to ARMv8 since ARMv7, RISC-V, Power, and any other architectures we are aware of forbid cooperative out-of-order execution. As an ongoing work, we are slightly generalizing our C/C++ promising semantics to account for these global behaviors in order to validate the standard mapping from C/C++ to ARMv8.

Based on the promising semantics, I am currently working on developing practical verification/analysis tools for reasoning about concurrent programs, which I will explain in the following paragraphs.

Promise analysis Promises and certification—the key idea of the promising semantics—introduce considerable nondeterminism so that they are the main difficulty in reasoning about concurrent programs. Currently, we are designing *promise analysis* that statically checks whether promising to execute any store instruction introduces any new observable behaviors at the program level, and if so, inserts fences as few as possible in order to offset the effect of store hoisting. After a program is analyzed and fences are minimally inserted, it is safe to treat all the store instructions as if they are forbidden to be promised in the execution, thereby greatly simplifying the verification of concurrent programs. The key idea of our analysis is that a promise to write a store instruction introduces additional behaviors to a program *only if* the previous instructions of the store are affected by that promise via inter-thread interactions, which we can safely analyze by **conservatively tracking the dependency among instructions**.

We believe our analysis is precise enough that it will **insert fences into the real-world concurrent programs only rarely**. Actually, we observed that promises do not seem to introduce any new observable behaviors in most real-world concurrent programs, because they are already quite strongly synchronized and the effect of promises via store hoisting is confined to local regions of code: we informally examined dozens of real-world concurrent data structures—including queues, stacks, deques, hash tables, and B-trees—and so far we have found only one exception to the observation, namely the Chase-Lev deque [9]. Even for Chase-Lev deque, inserting a single fence in the code will nullify the effect of store hoisting.

Program verification in the absence of promises Even if we can safely ignore the effect of store hoisting in reasoning thanks to promise analysis, concurrent programs are still difficult to reason about due to the nondeterministic interleaving of multiple threads, load hoisting, coherence, synchronization, and other concurrency features. Fortunately, recent advances in *concurrent separation logic* (CSL) allow programmers to verify more and more realistic concurrent programs in terms of ownership [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. In particular, iGPS [19], which is one of the state-of-the-art CSL's, supports a significant subset of the C/C++ concurrency features—such as release, acquire, and nonatomic accesses, but *not* store hoisting—and successfully verifies various real-world concurrent algorithms like spinlock, message passing, circular buffer, bounded ticket lock, Michael-Scott queue, and the read-copy-update (RCU) technique employed in the Linux kernel. What is particularly interesting is that by combining the result of promise analysis and verification in any CSL, one can formally guarantee that a program's implementation—even considering the effect of store hoisting due to promises—satisfies its specification.

The problem is that most of the concurrent algorithms verified in an existing CSL are actually simplified from the original algorithms: most notably, C/C++ relaxed accesses, which are beyond the reach of the current generation of CSL's, are strengthened to acquire/release accesses. I aim to solve this problem by **generalizing the iGPS program logic to account for relaxed accesses** (except for store hoisting, as it is the responsibility of promise analysis) and verifying the state-of-the-art concurrent algorithms without any simplifications in the generalized iGPS.

As a first step, we informally verified various real-world concurrent algorithms to get the insight of why they are correct. In particular, we successfully verified the Chase-Lev deque [9], which employs one of the most complex synchronization patterns we are aware of [20]. Furthermore, in the course of doing so, we observed that a few synchronization patterns are frequently used for multiple algorithms [21]. We believe this experience and observation will help in generalizing iGPS for verifying real-world concurrent algorithms.

Model checking An alternative approach to verifying concurrent programs is *model checking*, which is an automated method that basically enumerate all possible executions of a given program and checks if all the executions satisfy desirable properties—for example, safety or functional correctness. The biggest advantage of model checkers in verifying concurrent programs is that they are push-button solutions and are much easier to use than CSL's. However, it is difficult to develop a scalable model checker for concurrent programs: the existing model checkers for concurrent programs can verify only simple properties of tiny examples [22, 23, 24].

We observe that we can improve the scalability of model checkers for concurrent programs by exploiting the precise characterization of store hoisting in the promising semantics. The existing model checkers either ignore store hoisting at all, thereby resorting to a simplified programming model, or naively enumerate the executions with store hoisting, suffering from unbounded nondeterminism. The promising semantics can tame this nondeterminism by precisely classifying store hoistings into valid and invalid ones, thereby exponentially reducing the number of states to consider compared to the existing model checkers. This optimization is particularly effective for ARMv8/RISC-V thanks to their inherently operational nature: we already successfully implemented a prototype model checker for ARMv8/RISC-V and verified spinlock, and we are working on verifying the concurrent algorithms in the Linux kernel such as RCU and hash tables. We also have a plan to develop an efficient model checker for C/C++.

End-to-end verification of concurrent programs So far I introduced my research projects on verifying the implementation of concurrent programs, but it is not enough to guarantee end-to-end correctness concurrent programs from specification to *bare-metal*. In order to fill the gap, I also aim to verify that the concurrent program's semantics is *preserved by compilers* and then *observed by hardware*.

Compiler verification is a holy grail in the study of programming languages, but it is known to be an extremely difficult problem even for sequential programs without concurrency. Yet a decade ago Xavier Leroy and his collaborators demonstrated in the CompCert project [25] that formal verification of a C compiler within a theorem prover is actually feasible. Since then CompCert has been extended to support almost the entirety of ISO C99, major optimizations of GCC and LLVM, and commodity architectures such as x86, Power, ARM, and RISC-V. After many years of effort, CompCert is now mature enough to be used for real-world safety-critical systems [26].

I also have contributed on applying compiler verification techniques to real-world C programs and mainstream compilers in the following projects:

- First, we observed that CompCert does not properly support casts between integers and pointers, which is one of the defining characteristic of C. Here again, the problem is that it is difficult to define the programming language semantics of integer-pointer casts that adequately balances the conflicting desiderata of programmers and compilers. In our **PLDI 2015** paper [27], we proposed the first formal semantics of integer-pointer casts that supports both reasoning principles and compiler optimizations at the same time.

- Furthermore, we observed that CompCert simplified the verification problem by restricting attention to the correctness of whole-program compilation, leaving open the question of how to verify the correctness of separate compilation and linking. In our **POPL 2016** paper [28], we developed several lightweight techniques that recast the verification of separate compilation in terms of that of whole-program compilation, thereby enabling us to verify separate compilation for CompCert with only a 3% increase of LOC. In the course of doing so, we found a miscompilation bug in CompCert due to an analysis that is invalidated in the presence of linking. This bug was subsequently fixed in CompCert 2.5, and our verification techniques were adopted in CompCert 2.7.

- Last but not least, compiler verification techniques had not yet been applied to formally verifying mainstream compilers such as GCC and LLVM due to the huge cost of applying formal methods to millions of lines of code in compilers. In our **PLDI 2018** paper [29], we proposed CRELLVM: a verified credible compilation framework for LLVM, which can be used as a systematic way of providing a high level of reliability for major optimizations in LLVM. Specifically, we augment an LLVM optimizer to generate translation results together with their correctness proofs, which can then be checked by a formally verified proof checker. As case studies, we applied our approach to two major optimizations of LLVM, namely register promotion and global value numbering, having found four new miscompilation bugs (two in each).

To establish confidence in our results, we formalized most of the results of the above research projects in Coq.

Based on these experiences, now I aim to verify compilers for concurrent programs. In our C/C++ promising semantics paper [3], we already verified peephole optimizations for concurrent programs—such as reorderings and merges—that serve as the building block of major optimizations including register promotion, store forwarding, or loop-invariant code motion. As a future work, I will introduce C/C++ concurrency features to CompCert and re-verify CompCert's existing optimizations, and based on the experience, validate LLVM optimizations in the presence of C/C++ concurrency features on CRELLVM.

Now down to the bare-metal, **hardware verification** aims to verify the design of electronic circuits before taping it out. It is a crucial process in the semiconductor industry because circuits, once fabricated, cannot be modified. The problem is that the current practice of hardware verification requires manual inspection of experts for each component and thus incurs tremendous cost: automated tools like model checkers, which would greatly reduce verification cost if exist, are usually limited to verifying only small components of hardware due to the huge space of states to consider. Even worse, verification tasks are severely time-constrained due to the importance of time-to-market. Furthermore, concurrency—which I aim to verify in the beginning—adds significant additional complexity to hardware design and verification problems because concurrent components are tightly coupled with each other. I personally suffered from this difficulty in architecting Furiosa AI's highly-concurrent, massively-parallel MadRun deep learning accelerator (currently in preparation for RTL freeze).

As a future work, I would like to approach this problem by applying software verification techniques to hardware verification problem. Recent advances in formal verification of compilers, operating systems [30, 31], and database management systems [32] show that formal verification techniques scale up for real-world systems. I believe it can be successfully applied to hardware verification as well, and besides, formal verification of hardware is actually cost-effective in the long run because once a component is formally verified, it can be reused or adapted to other designs with low cost.

I will use Furiosa AI's MadRun accelerator as the testbed because its high concurrency makes it an excellent playground for concurrent hardware verification problems, and Furiosa AI, for which I am currently working as Co-founder & Chief Scientist, will give me full cooperation. As a preliminary step, we wrote MadRun in the Chisel hardware description language [33], which provides programming language features—such as data abstraction and module system—that enable a precise and efficient description of specification. In the future, we will write a precise specification for each component of MadRun, and then formally verify it in Coq or using other verification tools such as SMT solvers, model checkers, and program analyzers.

Funding For the first few years, I would like to set a budget of \$170K/year. First, I intend to advise up to four graduate students, whose labor cost amounts to approximately \$100K/year. Second, I want myself and my students to attend conferences and other academic events, totaling eight times a year, which will cost approximately \$30K/year. Furthermore, I want to buy new computer systems worth \$30K/year. Finally, I expect there will be miscellaneous expenditure worth \$10K/year.

In order to get funded, I will make several proposals for research programs and industry collaboration. For research funds, I will apply for Basic Science Research Program of National Research Foundation (한국연구재단 이공분야 기초연구사업 신진/중견연구자지원사업) and ICT Creative Research Projects of Samsung Future Technology Foundation (삼성미래기술육성센터 ICT 창의과제). For industry collaboration, I will set up R&D projects with Furiosa AI and other companies working on concurrent and parallel systems such as SAP and Microsoft.

Acknowledgments The work I have described in this research statement——which is supported in part by generous funding from National Research Foundation, Samsung Future Technology Foundation, Kwanjeong Educational Foundation, and Korea Foundation for Advanced Studies——would not have been possible without the efforts of my past and present collaborators at Seoul National University and elsewhere. I would like to thank

Sungkeun Cho, Joonwon Choi, Mark Dongyeon Shin, Derek Dreyer, Dmitri Garbuzov, Chung-Kil Hur, Yoonseung Kim, Yonghyun Kim, Ori Lahav, Juneyoung Lee, Sung-hwan Lee, William Mansky, Sanghoon Park, Jean Pichon-Pharabod, Christopher Pulte, Youngju Song, Viktor Vafeiadis, Kwangkeun Yi, and Steve Zdancewic for their essential contributions.

References

[1] The Crossbeam developers. Crossbeam. `https://github.com/crossbeam-rs`, 2015.

[2] The Rust developers. The Rust programming language. `https://www.rust-lang.org`, 2015.

[3] **Jeehoon Kang**, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017.

[4] Christopher Pulte, Jean Pichon-Pharabod, **Jeehoon Kang**, Sung-Hwan Lee, and Chung-Kil Hur. Promising-arm/risc-v: a simpler and faster operational concurrency model. `http://sf.snu.ac.kr/promising-arm-riscv/`, 2018.

[5] The Coq developers. The Coq proof assistant.

[6] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling c/c++ concurrency: From c++11 to power. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12.

[7] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising c/c++ and power. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12.

[8] Ori Lahav, Viktor Vafeiadis, **Jeehoon Kang**, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017.

[9] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05.

[10] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3).

[11] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3).

[12] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *Proceedings of the 18th International Conference on Concurrency Theory*, CONCUR'07.

[13] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09.

[14] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10.

[15] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*.

[16] Pedro Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*.

[17] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13.

[18] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*.

[19] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In *The European Conference on Object-Oriented Programming*, ECOOP 2017.

[20] **Jeehoon Kang**. Prove the correctness of a work-stealing deque. `https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md`, 2017.

[21] **Jeehoon Kang**. Relaxed-memory concurrency synchronization patterns. `https://jeehoonkang.github.io/2017/08/23/synchronization-patterns.html`, 2017.

[22] Brian Norris and Brian Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13.

[23] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for c/c++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL).

[24] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2).

[25] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7).

[26] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE, January 2018.

[27] **Jeehoon Kang**, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal c memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15.

[28] **Jeehoon Kang**, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16.

[29] **Jeehoon Kang**, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: Verified credible compilation for llvm. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018.

[30] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6).

[31] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16.

[32] Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler. Certifying a file system using crash hoare logic: Correctness in the presence of crashes. *Commun. ACM*, 60(4), March.

[33] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14.