Dimi Racordon

Research Statement

My research advances methods and techniques that empower developers to write expressive, efficient, and reliable software. I gravitate toward model checking, formal verification, and programming language design, with a particular focus on advanced type systems.

My background as both researcher and software engineer has forged an interest in approaches bridging theory and practice. I enjoy crafting rigorously defined systems that are concise yet expressive and accessible, I value techniques that allow powerful software abstractions to interact with the constraints of hardware systems efficiently, and I strive to facilitate knowledge transfer between academia and industry through collaborations.

After studying type-based approaches for **memory safety** [11] and data structures for **symbolic model checking** [12], my research has focused on **value-oriented programming** for high-level systems programming [14]. The remainder of this statement gives a summary of these contributions, describes my work, and concludes with future research directions.

Value-oriented programming

A fundamental challenge for the adoption of formal methods is the semantic gap between the formalisms to reason about software and the techniques to implement it efficiently. At the center of this rift lie reference semantics, which impedes formal reasoning by introducing *aliasing*, a condition where two or more live variables refer to the same memory location. Specifically, pointers and references restrict the machine's ability to reason about code without considering the whole program, introducing combinatorial explosion. To mitigate this problem, verification techniques must either severely restrict mutation patterns, demand sophisticated annotations, lose accuracy, or suffer performance overhead to defer checks at run-time.

Mutable value semantics is a way to address this issue. In its strictest form, it is a value-oriented programming discipline that excludes pointers and references from the programming model, de facto preventing variables from sharing mutable state. However, unlike functional programming, mutable value semantics can express part-wise in-place mutations across function boundaries, thereby providing predictable performance guarantees on mutating algorithms.

My work on mutable value semantics started with a study of Swift [9], a language that supports this discipline. I formalized parts of Swift's type system to discover bugs in its implementation, which lead to a collaboration with partners from Google and Adobe on a core calculus to formally describe mutable value semantics [14]. We formalized a subset of Swift, proved its soundness with respect to type and memory safety, and described optimizing strategies to compile efficient programs. I implemented a compiler based on these techniques [13] to measure the performance of mutable value semantics against Scala's functional updates and C++ reference model, demonstrating the relevance of Swift's approach for safe and efficient programs.

44 Washington Street, Brookline, MA 02445, USA
dimitri.racordon@gmail.com

I currently collaborate with Adobe's Software Technology Lab on the design of a new programming language, called Val [1], which builds on my previous results. Val codifies best practices from performance-critical programming into its type system to enforce local reasoning, uphold memory safety, and reliably identify unnecessary memory allocations.

Val's main novelty is a feature called *projections*, which addresses the *view-update problem*, a situation where mutations applied to a transformed view of an object must be applied back to that object. Projections generalize a technique developed in Swift to implement lenses efficiently—without dynamic allocation—by transforming code into a continuation passing style to avoid the complexity costs of annotation systems meant to track object lifetimes.

Memory safety

Access to unallocated (e.g., use-after-free) or uninitialized memory can cause insidious bugs that are difficult to identify and may open vulnerability issues. **Memory safety** is defined as freedom from this kind of bug and can be upheld in different ways. The main challenge faced by approaches operating ahead-of-type—before the program runs—is to deal with aliasing.

My dissertation [5] reviewed type-based approaches to guarantee memory safety, formally described a core calculus to reason about program semantics with respect to memory safety, and devised a type system that soundly enforces memory safety. I designed two programming languages based on this work. The first was a dialect of Javascript that showcased the use of different assignment operators in conjunction with type capabilities to avoid mutation through unintended sharing [7]. The second was a general-purpose programming language whose goal was to study the interplay between assignment semantics, aliasing control mechanisms, and other language constructs, such as generic types and higher-order functions [8].

My research along this axis continued with the development of a low-level intermediate representation to introduce memory safety checks in a compiler pipeline with colleagues from the University of Geneva [11, 10]. The main novelty of this work was to leverage control flow information to refine static assumptions about aliasing, thereby reducing the need for overly conservative restrictions.

Symbolic model checking

Model checking consists of exploring the states of a system to verify if a given property holds or identify counter-examples. An important limitation of this approach is *state space explosion*, a phenomenon that occurs when the number of states of a model grows exponentially.

One approach to tackle this issue is **symbolic model checking**, which compresses state spaces into memory-efficient data structures, typically decision diagrams. These compression techniques can not only dramatically reduce the memory footprint of a state space but also improve the efficiency of set-based operations.

I developed a variant of decision diagrams, called a map family decision diagram (MFDD), which generalizes binary and data decision diagrams to encode and apply operations on large families of partial functions. I showed how MFDDs can be used to verify specifications expressed in terms of constraint-based systems [15, 12, 6], and developed a model checking tool based on this approach [4]. This work has resulted in a series of lectures that are now taught at the University of Geneva.

Earlier in my doctoral studies, I worked on cost linear temporal logic [3], which extends linear temporal logic with the ability to count events in systems with infinite behaviors. The main contribution of this work was a method to discover counter bounds as an extension of classical emptiness check methods for ω-automata [2], a class of automata that run on infinite inputs.

Future work

The tension between algebraic semantic reasoning and unconstrained mutation has been well known at least since Backus' 1977 Turing Award paper introducing functional programming. On one hand, functional programming gave rise to languages and tools whose power and expressiveness go beyond what could have only been dreamed a few decades ago. On the other, performance-critical applications often cannot afford the run-time costs of functional abstractions.

In response to this issue, tremendous efforts have been poured into methods and techniques preventing the unintended consequences of shared mutation through aliasing. Perhaps the most successful product of this work is Rust, a language that supports common low-level programming idioms yet guarantees safety through its type system. However, Rust is a complex language, to the point that "fighting its compiler" has become a meme among programmers.

I aim to tackle this problem and settle the apparent struggle between correctness and efficiency. A key insight from my ongoing collaborations with practitioners is that neither sophisticated aliasing restrictions nor pure functional programming are solutions in practice. Instead, developers rely on strict coding discipline to uphold local reasoning without loss of efficiency, leveraging familiar concepts such as type encapsulation and parameter passing conventions.

My goal is to devise sound mathematical foundations to capture and enforce these practices under the umbrella of mutable value semantics, thereby allowing idealized formal models of computation *and* efficient mutating algorithms to coexist. These foundations will lead the way to further advance the state of the art along these axes:

Software verification Mutable value semantics offers an opportunity to revisit approaches developed for pure functional systems that failed to scale to systems with shared mutation. I intend to build on this observation to explore automated verification techniques, in particular using typestates and refinement types.

Structured concurrency Concurrency is an old yet still timely challenge. Most applications rely on unstructured and error-prone synchronization mechanisms that do not compose well. Upholding value independence uncovers leads to implementing systems that do not surface low-level synchronization primitives and can guarantee thread safety by construction.

Composition of safe and unsafe languages The experience of interoperating between safe and unsafe languages is currently undermined by a lack of mechanisms to mediate memory access, therefore compelling developers to enforce all guarantees manually. I believe this burden can be lifted by developing new techniques to define richer foreign function interfaces.

A cross-cutting constraint of my research will be to develop approaches that are not only rooted in solid theoretical foundations but also practical, scalable, amenable to gradual adoption, and suitable to performance-critical domains. By maintaining strong collaborations with the industry in parallel to my research, I intend to provide a clear path toward safer and more efficient software.

Publications (full list in CV)

[1] Dave Abrahams, Sean Parent, Dimitri Racordon, and David Sankel. The val object model. *C++ Standards Committee Papers*, 2022(10), 2022.

[2] Maximilien Colange, Dimitri Racordon, and Didier Buchs. A cegar-like approach for cost LTL bounds. *CoRR*, abs/1506.05728, 2015.

[3] Maximilien Colange, Dimitri Racordon, and Didier Buchs. Computing bounds for counter automata. *Electronic Communication of the European Association*, 72, 2015.

[4] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, Y. Thierry-Mieg, K. Wolf, J. van de Pol, C. Rohr, M. Heiner, G. Tran, and J. Srba. Complete results for the 2016 edition of the model checking contest. `http://mcc.lip6.fr/2016/results.php`, 2016.

[5] Dimitri Racordon. *Revisiting Memory Assignment Semantics in Imperative Programming Languages*. PhD thesis, University of Geneva, Switzerland, 2019.

[6] Dimitri Racordon and Didier Buchs. Verifying multi-core schedulability with data decision diagrams. In Ivica Crnkovic and Elena Troubitsyna, editors, *International Workshop on Software Engineering for Resilient Systems (SERENE)*, volume 9823 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2016.

[7] Dimitri Racordon and Didier Buchs. A practical type system for safe aliasing. In David Pearce, Tanja Mayerhofer, and Friedrich Steimann, editors, *International Conference on Software Language Engineering (SLE)*, pages 133–146. ACM, 2018.

[8] Dimitri Racordon and Didier Buchs. Implementing a language with explicit assignment semantics. In Daniele Bonetta and Yu David Liu, editors, *International Workshop on Virtual Machines and Intermediate Languages (VMIL)*, pages 12–21. ACM, 2019.

[9] Dimitri Racordon and Didier Buchs. Featherweight swift: a core calculus for swift's type system. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *International Conference on Software Language Engineering (SLE)*, pages 140–154. ACM, 2020.

[10] Dimitri Racordon, Aurélien Coet, and Didier Buchs. Fuel: A compiler framework for safe memory management. In *International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS)*, 2021.

[11] Dimitri Racordon, Aurélien Coet, and Didier Buchs. Toward a lingua franca for memory safety. *Journal of Object Technology*, 21(2):2:1–11, 2022.

[12] Dimitri Racordon, Aurélien Coet, Emmanouela Stachtiari, and Didier Buchs. Solving schedulability as a search space problem with decision diagrams. In Aldeida Aleti and Annibale Panichella, editors, *International Symposium on Search-Based Software*

Engineering (SSBSE), volume 12420 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2020.

[13] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Native implementation of mutable value semantics. In *International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS)*, 2021.

[14] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Implementation strategies for mutable value semantics. *Journal of Object Technology*, 21(2):2:1–11, 2022.

[15] Dimitri Racordon Silvio Fossati, Aurélien Coet. Belief programming with map family decision diagrams. In *International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS)*, 2022.