Reachability Types, Traces and Full Abstraction

Benedict Bunting Andrzej Murawski

April 2025

This talk is based upon the paper of the same name appearing at LICS 2025 [6]

Type systems that offer control over sharing are seen as a promising technique for improving program safety and performance. This has been demonstrated by the recent success of Rust, whose core is based on the *shared XOR mutable* principle, i.e. sharing is restricted to immutable variables. This policy turns out to be quite restrictive when it comes to expressing common programming patterns involving higher-order functions and state, like those expressible in languages such as ML or Scala. Reachability types [3, 12] are a recent proposal to address this gap and provide a type system that is capable of collecting information about sharing as well as lack thereof, i.e. separation.

The key idea of reachability types is to track reachable variables/locations by annotating types with *type qualifiers*, which contain functions or locations that may be reachable from a given term. For example, the term $h \triangleq \textsf{let}\, r = \textsf{ref}\ (0)\ \textsf{in}\ \lambda f.\lambda x.(!\ell + f(!r) + g(x))$, where ℓ is a memory location, ! stands for dereferencing and $g : \tau$ is free, would normally be typed as $\tau \to \tau$ with $\tau \triangleq \mathsf{Int} \to \mathsf{Int}$. With reachability types, it can be given the more accurate type below.

$$(\mu h.(f : (\mathsf{Int} \to \mathsf{Int})^{Q}) \to (\mathsf{Int} \to \mathsf{Int})^{\{f,g,h,\ell\}})^{\{g,\ell\}}$$

The presence of g, ℓ in qualifiers indicates that both the whole term and its functional result may directly reach the unknown function g and location ℓ. In contrast, f corresponds to a dependency on resources contributed by the input. h in turn is a *self-reference*, which allows one to express the fact that the functional result of h may reach (private) locations (such as r) created by h itself. In addition, the argument type τ^{Q} can be used to specify the degree of overlap between h and its arguments. Setting Q to $\emptyset$ corresponds to demanding that what the argument can reach must be disjoint from what h can reach, while $\{\ell\}$ would allow for scenarios in which the argument may also reach ℓ, but not g. Similarly, $Q = \{\ell, g\}$ would permit h to share ℓ, g with its arguments. Overall, the idea of tracking reachability at the type level turns out very powerful and can be used to express many common programming scenarios such as non-interference, non-escaping, non-accessibility and scoped borrowing [3].

The use of qualifiers in reachability types complicates arguments about contextual equivalence, such as may be needed justify reachability-type-based program transformations and optimisations [5]. For example, whether the term $\textsf{let}\ f = h()\ \textsf{in}\ (\textsf{let}\, g = h()\, \textsf{in}\, f(); g())$ is equivalent to $\textsf{let}\ f = h()\ \textsf{in}\ (\textsf{let}\, g = h()\, \textsf{in}\, g(); f())$ depends on how $h : \mathsf{Unit} \to (\mathsf{Unit} \to \mathsf{Unit})$ is typed. If $h : (\mu h.\mathsf{Unit} \to (\mathsf{Unit} \to \mathsf{Unit})^{\{h\}})^{\emptyset}$ then f, g may share the private state of h and the terms will not be equivalent, because the order of calls can be detected through the state. In contrast, for $h : (\mu h.\mathsf{Unit} \to (\mathsf{Unit} \to \mathsf{Unit})^{\emptyset})^{\emptyset}$ the terms will behave in the same way.

In this talk, we explain how to employ operational game semantics [7,8] to provide a formal and precise account of the underpinning theory of contextual equivalence (and the associated notion of approximation). Game semantics is already known for providing a wide range of fully abstract models for various programming paradigms [2,10]. They rely on representing interactions between programs and contexts as a dialogue between two players: O (context) and P (program). A characteristic feature of operational game semantics is that these dialogues are generated as traces of a carefully crafted labelled transition system (LTS), which uses names to represent unknown functions in the spirit of open/normal-form bisimulation [9, 11]. In addition, to capture the sharing of private state, the traces of our LTS will be decorated with sets of abstract states revealed by the environment.

In order to demonstrate our approach, we introduce a minimalistic language with reachability types, modelled after λ^* [3]. As our methods exploit the ability to η-expand terms, the calculus is based on a notion of *well-behaved* types, for which η-expansion is guaranteed to be type-preserving.

We develop an LTS $\mathcal{L}_P$, which provides a sound model. In particular, the LTS non-trivially adapts the classic notion of visibility (originally used to characterise functions that are reachable/visible to players in a language with first-order references) [1] to the setting of reachability types. Intuitively, this is done by identifying a family of subtle technical conditions that further restrict visibility to make it compatible with types.

Building on a definability result, we refine the trace model to a fully abstract one by introducing rearrangement relations. We show that one can capture contextual equivalence via complete trace equivalence up to allowable trace permutations. Program approximation in turn can be characterised through an ordering that allows one to omit certain actions. For example, it turns out that the term $\mathsf{let}\ f = h()\ \mathsf{in}\ f(1); f(2)$ approximates $\mathsf{let}\ f = h()\ \mathsf{in}\ f(1)$ when $h : (\mu h.\mathsf{Unit} \to (\mathsf{Int} \to \mathsf{Unit})^{\emptyset})^{\emptyset}$, because $f(1)$ and $f(2)$ cannot reach any state other than f's private one and, consequently, $f(2)$ does not interfere with the subsequent computation.

At a high level, we see our work as the beginning of a semantic study of reachability types. So far, the most closely related work in this direction is [4], which provides a sound logical relation for contextual equivalence.

References

[1] S. Abramsky and G. McCusker. Call-by-value games. In *Proceedings of CSL*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1997.

[2] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Logic and Computation*. Springer-Verlag, 1998. Proceedings of the NATO Advanced Study Institute, Marktoberdorf.

[3] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021.

[4] Yuyan Bao, Guannan Wei, Oliver Bracevac, and Tiark Rompf. Modeling Reachability Types with Logical Relations. *CoRR*, abs/2309.05885, 2023.

[5] Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM on Program. Lang.*, 7(OOPSLA2):400–430, 2023.

[6] Benedict Bunting and Andrzej S. Murawski. Reachablity types, traces and full abstraction. In *LICS*, 2025.

[7] Guilhem Jaber and Andrzej S. Murawski. Complete trace models of state and control. In *Proceedings of ESOP*, volume 12648 of *Lecture Notes in Computer Science*, pages 348–374. Springer, 2021.

[8] J. Laird. A fully abstract trace semantics for general references. In *Proceedings of ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.

[9] S. B. Lassen and P. B. Levy. Typed normal form bisimulation. In *Proceedings of CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2007.

[10] Andrzej S. Murawski and Nikos Tzevelekos. An invitation to game semantics. *ACM SIGLOG News*, 3(2):56–67, 2016.

[11] D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Inf.*, 33(1):69–97, 1996.

[12] Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.*, 8(POPL):393–424, 2024.