The Game Semantics of Reachability Types

Benedict Bunting *joint work with* Andrzej Murawski

May 2026

The presence of shared (or aliased) mutable state has long been recognised as the root cause of many difficulties in programming, particularly complicating understanding and reasoning about the behaviour of programs. Approaches to controlling aliasing are a rich area of study and have become increasingly adopted in mainstream programming languages, for example Rust [5] and Linear Haskell [3]. The policy many of these languages adopt (*aliasing XOR mutability* and linearity, in the examples, respectively) are incompatible with common, impure, higher-order patterns based on sharing of mutable state. For example, the following code generating a counter with variable increments cannot be easily handled by such approachs, and even then, information about which resulting closures share state is usually lost.

$$\mathit{new\_counter} \triangleq \lambda u.\mathbf{let}\, x = \mathbf{ref}\, \widehat{0}\, \mathbf{in}\, \lambda n.\, (\lambda v.x := !x + n; !x) : \mathsf{Unit} \to \mathsf{Int} \to \mathsf{Unit} \to \mathsf{Int}$$

Reachability Types [2] are a recent approach to provide control over sharing in a higher-order functional programming language, by tracking in the type system information about what resources are reachable from a term, which allows separation requirements to be enforced using types.

The basic premise of Reachability Types is to track what mutable references (or other resources) are reachable, by annotating types with type qualifiers. When applied to the type of a term, a qualifier can be seen as containing an (overapproximation of) the mutable variables and locations that are reachable from the resulting value of the term. Qualifiers are also employed to control the sharing between a function and its argument, and what can be captured by a returned closure. For example, the term $\mathbf{let}\, x = \mathbf{ref}\, \widehat{0}\, \mathbf{in}\, \lambda g.\lambda y.(x := g\,(!x + !\ell + y); !x)$, can be given type

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

Having ℓ in qualifiers indicates that the term and the returned function both reach the location ℓ. g in the returned qualifier reflects the fact that the returned function captures the argument of the outer function. h is a *self-reference*, which expresses that this returned function captures local state generated by this term, like the reference x. Finally, the argument qualifier Q can be varied to control the overlap between this term and its arguments. Using $Q = \emptyset$ means complete seperation, whereas $Q = \{h\}$ allows complete overlap. $Q = \{\ell\}$ allows the argument to reach ℓ, but not h itself (and so not x).

In this talk, I will discuss how to apply operational game semantics (OGS) to investigate the meaning of Reachability Types and the impact these have on the language. This work is detailed in my forthcoming DPhil thesis. An earlier version on this work was presented at GaLoP 2025, based upon the LICS 2025 paper *Reachability Types, Traces and Full Abstraction* [4]. This talked is intended to be a update on this project, with an improved pedagogical presentation, which I hope will be of interest to the GaLoP audience. In particular, the major differences will be:

- A simplified yet more expressive language (called λ^*_{sat}), leading to a cleaner model;
- An more incremental approach to building the model, highlighting they way it builds upon the standard OGS approach;
- New results using the trace model to express an important separation guarantee on the way variables can be instantiated;
- New results on simplifying the model to avoid, and connecting the result to a 'true' model of concurrency, a nominal version of Mazurkiewicz trace; and
- A clearer picture of planned future work.

I will now discuss the outline of the material in more detail.

A key way of understanding a language is through characterising the canonical notion of equivalence between terms, known as contextual equivalence (and the cor responding notion of approximation). In λ^*_{sat}, Reachability Types restrict the power of contexts to distinguish terms when compared to a simply typed language, leading to a more intricate notion that relates more terms. For example, the terms M_1 and M_2, below, are contextually equivalent when given the type $((\mathsf{Unit}^{\perp} \to \mathsf{Unit}^{\perp})^{\emptyset} \to \mathsf{Unit}^{\perp})^{\emptyset}$, whereas they are not in a simply typed language.

$$M_1 \triangleq \lambda g.g\,() \qquad M_2 \triangleq \mathbf{let}\, x = \mathbf{ref}\,\widehat{0}\,\mathbf{in} \\ (\lambda g.\mathbf{if}\,!x = \widehat{0}\,\mathbf{then}\,x := \widehat{1};\, g\,();x := \widehat{0}\,\mathbf{else}\,\Omega)$$

By discussing this and other examples, I will demonstrate the subtlety of contextual testing in λ^*_{sat}.

Following the Operational Game Semantics approach, we start with an initial labelled transition system, which generates traces that soundly capture the interaction between a term and all possible contexts. By examining the generation of these traces, we identify the property of Ψ-visibility, that is satisfied by the traces of λ^*_{sat} terms. Refining the classical notion of visibility, this captures the essences of Reachability Types, explaining when functions passed between the term and context can be used during the interaction. This principle is sufficient to handle the example above.

Moving beyond the standard OGS techniques, I will discuss how the notion of trace and the labelled transition system can be enriched to track the way in which mutable state is shared between different parts of a program. From this, properties can be identified which capture how Reachability Types constrain the sharing of mutable state. Combining these properties with Ψ-visibility enables a proof of a preservation-of-separation result that explains the key guarantee of Reachability Types.

A richer trace semantics can be given by incorporating the identified properties into the model, but it remains imprecise as λ^*_{sat} is not *observably sequential*: the constraints of Reachability Types mean contexts cannot observe the order of certain actions. For example, given $\lambda f.\lambda g.f();g()$ and $\lambda f.\lambda g.g();f()$ are equivalent at particular types. By exploiting the information about state sharing in traces, a novel concrete quotient of the model is developed based upon reordering actions in traces. This yields the most significant result of this talk: the first *fully abstract* model of a language equipped with Reachability Types, allowing precise reasoning about both contextual equivalence and approximation.

I will discuss how a simplification of the representation of sharing in traces can be done in a way that retains full abstract. The resulting model finds firm foundations in a nominal version of Mazurkiewicz traces.

At the high-level, this talk has two aims. The first is to promote the semantic study of Reachability Types. So far, the most closely related work in this direction is [1], which provides a sound logical relation for contextual equivalence. I will outline some directions for potential further study. The second is to present a blueprint for applying operational game semantics to languages which control the aliasing of mutable state. I aim to make the key elements of our approach clear, and hope they might be useful in other settings.

References

[1] Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, Effect Safety, and Equational Theory. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):1837–1864, 2025. OOPSLA '25.

[2] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–32, 2021. OOPSLA '21.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, December 2017. POPL '18.

[4] Benedict Bunting and Andrzej S. Murawski. Reachability types, traces and full abstraction. In *2025 40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 527–540, 2025.

[5] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104. ACM, 2014.