Polylang: Programming with polynomial functors and lenses (extended abstract)*

Dylan Braithwaite Jules Hedges Zanzi Mihejevs

The category of polynomial endofunctors and natural transformations, equivalently known as the category of containers and dependent lenses, is well known to have a wide variety of potential applications. We will demonstrate Polylang, a high level programming language and compiler designed to support all of these applications while being agnostic between them. From a theoretical perspective it is an implementation of the intrinsic simple type theory of **Poly**: its types denote polynomials and its functions denote lenses, and as such have a "forwards pass" and a "backwards pass".

The operator basis supported by Polylang includes the cartesisan product, tensor product, coproduct, composition product and fixpoints. The interaction of cartesian and tensor products requires a system of quantities inspired by graded and quantitative type theories. The composition product is the most difficult to include since it resists straightforward implementation in a natural deduction calculus, but we achieve it with an imperative style language via a sequent calculus exploiting the isomix linearly distributive structure [SS24]. Algebraic datatypes involve fixpoints of polynomial endofunctors on **Poly** itself, and significantly increase expressive power, including allowing the embedding of various categories of traversals as kleisli categories of definable monads.

The plan for the talk is to explain some of this theory in the syntax of the language itself while also demonstrating the compiler in action.

1 The composition product

The composition product of polynomial functors $\triangleright$ is extremely important (for being a primary topic of the second half of the Poly book [NS25]) but is very difficult to characterise syntactically. One solution uses dependent types,[1] however in Polylang we choose to avoid dependent types for pragmatic reasons. We use a design baed on a variant of the computational λ-calculus adapted to the graded monad $T_P(X) = X \triangleright P$ (which exists for any monoidal product). This is a noncommutative graded monad, which lends itself to an imperative-style programming language. Its corresponding algebraic signature is essentially that of a stack, with `push` and `pop` primitives. In particular a

*Part of this extended abstract is heavily condensed from two blog posts, [Hed26a] and [Hed26b].

[1]This is the topic of a parallel talk proposal by the same group of authors.

`pop2` primitive will take 2 values from the stack and package them as a sequence product. For example, the left-biased of the two morphisms $A \otimes B \to A \triangleright B$ in **Poly** is expressed using a syntax such as[2]

```
match x {
  (x1 * x2) => {push(x1); push(x2); pop2()}
}
```

Deconstructing values of a composition product type combines aspects of matching on a tensor product and a coproduct, revealing its similarity to the 'par' operator of linear logic. It is syntactically like matching on a coproduct, but operationally *both* branches are executed in sequence, and it has tensor-like scoping rules for linear variables.

```
match x {
  First x1 => ...
  Second x2 => ...
}
```

2 Cartesian products and quantities

The interaction of the cartesian product and tensor product in Polylang is described using a system of quantities inspired by quantitative type theory [Atk18]. Internally variables are linear-by-default, but 'erased' variables are semantically backwards-trivial and can be implicitly copied and deleted. Matching on a tensor product produces two linear variables, but when matching on a cartesian product one of the variables must be marked as erased. For example, the left-biased of the two morphisms $A \times B \to A \otimes B$ in **Poly** is expressed using a syntax such as

```
match x {
  (0x1, x2) => (x1 * x2)
}
```

This interaction is currently our only known source of incompleteness: our rules cannot derive the linear distributive law $A \times (B \otimes C) \to (A \times B) \otimes C$. This appears to require a calculus of bunched contexts, which we are currently not implementing.

3 Fixpoints

Algebraic datatypes in Polylang denote, loosely, fixpoints of polynomial endofunctors on **Poly**. This is theoretically subtle but highly expressive. Of particular interest are the three fixpoints $\mu X.1 + A \otimes X$, $\mu X.1 + A \times X$ and $\mu X.1 + A \triangleright X$, which define three different list monads on **Poly** that have important applications.

Our construction of fixpoints is taken from [GH14]; compare [PP26] which restricts to fixpoints of fibred endofunctors, which we are not using because the composition product $\triangleright$ is not fibred in its left variable.

[2]The concrete syntax given here is based on Rust, but this is relatively unimportant and may change.

References

[Atk18] Robert Atkey. The syntax and semantics of quantitative type theory. In *Proceedings of LiCS*, 2018.

[GH14] Neil Ghani and Peter Hancock. Containers, monads and induction-recursion. *Mathematical structures in computer science*, 2014.

[Hed26a] Jules Hedges. Sequents for sequence. Blog post available at `https://julesh.com/posts/2026-03-13-sequents-sequence.html`, 2026.

[Hed26b] Jules Hedges. Sequents for sequence II: Balancing the strangeness budget. Blog post available at `https://julesh.com/posts/2026-03-23-sequents-sequence-ii.html`, 2026.

[HM25] Jules Hedges and Zanzi Mihejevs. Canonical bidirectional typechecking. arXiv: 2512.07511, 2025.

[NS25] Nelson Niu and David Spivak. *Polynomial functors: A mathematical theory of interaction*. Cambridge University Press, 2025.

[PP26] Cécilia Pradic and Ian Price. Problems with fixpoints of polynomials of polynomials. arXiv: 2601.15420, 2026.

[SS24] David Spivak and Priyaa Varshinee Srinivasan. What kind of linearly distributive category do polynomial functors form? arXiv: 2407.01849, 2024.