«Normalizer» project

Technical report

«Normalizer» Development Team
Innopolis University
Innopolis, Tatarstan Republic, Russia

1 Introduction

The goal of the «Normalizer» project is to implement a rewrite-based approach for partial evaluation and optimization for EO and φ-calculus [1]. The «Normalizer» is a part of a larger project, aimed at optimizing the performance of Java code by focusing on the behavior and lifetime of dynamic objects and attempting to reduce their contribution to the overall performance, replacing, for instance, dynamic dispatch with static method invocations.

Our specific results so far are briefly described below:

1. We have implemented a rewrite-based software, called `normalizer`, able to work with φ-calculus expressions (as supported by the EO compiler) and utilize user-defined rewrite rules (specified in a YAML format) to apply certain rewrite-based transformations, as well as provide a step-by-step elaboration of the process. The project with its open-source code is available at github.com/objectionary/normalizer.
2. We have used a computer proof assistant Lean 4 [3] to completely formalize the minimal φ-calculus described in a previous project [2], including the confluence theorem. The formalization project is open source and available at github.com/objectionary/proof.
3. We have written down our formalization results and submitted a paper to ITP[1] 2024. The paper received 1 weak accept and 2 rejects. We are planning to submit a revised version of the paper to ITP 2025.
4. We have proposed a patent application for a compiler architecture with the support for two-phased intermediate representation optimizations. The application has been accepted by Huawei representatives.

2 Materials and Methods

Normalizer relies on φ-calculus extended with primitive data and operations (atoms). To process such expressions, a rewriting engine ("normalization") is coupled with partial evaluation mechanism (called "dataization"). This allows to continuously simplify and evaluate parts of a program, leading to its optimization.

We demonstrate the proposed optimization process on a slightly modified example provided by Möller [4]. In the φ-expression below, we have attribute `s` mapped to a stream of integers from 1 to `length` + 1. Stream `s` is then processed in the φ attribute, by conditionally applying a `map`, then applying `filter`, and then summing up the elements in the resulting stream.

The conditional `map` here prevents a direct application of a "stream fusion" optimization [5]. To unlock the optimization, we first evaluate the constant expressions.

```
⟦
  length ↦ 10,
  shouldSquare↦
    ξ.length.mod(α0 ↦ 2).equal(α0 ↦ 0),
  s ↦ Φ.org.eolang.stream.range(
    α0 ↦ 1,
    α1 ↦ ξ.length.add(α0 ↦ 1)),
  φ ↦ ξ.shouldSquare
    .if(α0 ↦ ξ.s.map(
          α0 ↦ ⟦x ↦ ∅,
                φ ↦ ξ.x.mul(α0 ↦ ξ.x)⟧),
          α1 ↦ ξ.s)
    .filter(α0 ↦ ⟦
      x ↦ ∅,
      φ ↦ ξ.x.mod(α0 ↦ 2)
            .equal(α0 ↦ 0)⟧)
    .sum
⟧
```

Now, that we understand that the condition is `true`, we can rewrite the entire conditional with its `true`-branch.

In this example we have been able to learn that the condition is always `true`, and we could eliminate the else-branch entirely. However, in a more general setting it is more likely that we will only partially evaluate the condition, and will have to distribute conditional over the `map` or `filter`, resulting either a stream with an embedded conditional or a conditional around two different stream pipelines (each of which will then be optimized separately).

In any case, it is critical for this step to be able to compute with values.

[1] The International Conference on Interactive Theorem Proving (a rank A conference according to the CORE ranking)

```
⟦
  length ↦ 10,
  shouldSquare ↦ Φ.org.eolang.true,
  s ↦ Φ.org.eolang
        .stream.range(α0 ↦ 1, α1 ↦ 11),
  φ ↦ Q.org.eolang.true
        .if(
          α0↦ξ.s.map(α0 ↦ ⟦
            x ↦ ∅,
            φ ↦ ξ.x.mul(α0 ↦ x)
          ⟧),
          α1 ↦ ξ.s)
        .filter(α0 ↦ ⟦
          x ↦ ∅,
          φ ↦ ξ.x.mod(α0 ↦ 2)
                .equal(α0 ↦ 0)
        ⟧)
        .sum
⟧
```

We now rewrite the `map`-`filter`-`sum` stream pipeline using a standard technique for stream fusion, replacing it with a single `for`-loop, where the body of the loop relies on the user code used in `map` and `filter`.

The idea is that instead of transforming the input stream twice (first via `map`, and then via `filter`) and then folding it into a single number (via `sum`), we could do all three operations at the same time when iterating (in a `for`-loop) over the elements in the input stream. This is a classical stream fusion transformation [5] that can be specified using rewrite rules.

```
⟦
  s ↦ Φ.org.eolang
        .stream.range(α0 ↦ 1, α1 ↦ 11),
  φ ↦ ξ.s.map(
        α0 ↦ ⟦
          x ↦ ∅,
          φ ↦ ξ.x.mul(α0 ↦ ξ.x)
        ⟧
      )
      .filter(
        α0 ↦ ⟦
          x ↦ ∅,
          φ ↦ ξ.x.mod(α0 ↦ 2)
                .equal(α0 ↦ 0)
        ⟧
      )
      .sum
⟧
```

The resulting program now consists of a stream generator and a tight loop that processes it, improving the overall processing of the data. The parts of the new code that are preserved from the original are highlighted in blue.

In this example, we could further fuse the generator and the `for`-loop, or even fold the entire expression into a constant, but in general the generator would be provided externally at runtime.

```
⟦
  s ↦ Φ.org.eolang
        .stream.range(α0 ↦ 1, α1 ↦ 11),
  Φ ↦ Φ.org.eolang.map-for-each(
    α0 ↦ ξ.s,
    α1 ↦ ⟦
      x ↦ ∅,
      sum ↦ ∅,
      t2 ↦ ξ.x.mul(α0 ↦ x),
      φ ↦ ξ.x.mod(α0 ↦ 2)
            .equal(α0 ↦ 0)
            .if(
              α0 ↦ ξ.sum.add(α0 ↦ t2),
              α1 ↦ ξ.sum
            )
    ⟧,
    α2 ↦ 0
  )
⟧
```

3 Results

We have implemented a prototype rewrite-based tool for φ-calculus that can be used to optimize and partially evaluate φ-expressions. We have formalized φ-calculus and shown that it satisfies confluence, meaning that we can reasonably use rewriting for program transformations. More specifically:

1. The Normalizer software is implemented and documented. It is open source and available at https://github.com/objectionary/eo-phi-normalizer.
2. The Normalizer improves the key metrics on EO tests:
   a. *Dataless formations* decrease by at least 20% on 125 out of 127 tests (98.43%).
   b. *Applications* decrease by at least 20% on 119 out of 127 tests (93.70%).
   c. *Formations* decrease by at least 10% on 125 out of 127 tests (98.43%).
   d. *Dispatches* decrease by at least 20% on 120 out of 127 tests (94.49%).
3. A translator from XMIR to φ-calculus is contributed to EO and is available at https://github.com/objectionary/eo/blob/master/eo-maven-plugin/src/main/java/org/eolang/maven/PhiMojo.java.
4. A formalization of φ-calculus, including confluence proof, is done in Lean 4. The formalization, all proofs, and examples are available at https://github.com/objectionary/proof.

We have produced one paper that has been approved by Huawei representatives and submitted to an appropriate confer- ence:

1. Anatoliy Baskakov, Nikolai Kudasov, Violetta Sim. Formal confluence for φ-calculus in Lean 4. *Rejected from ITP 2024. Plans to resubmit a revised version to ITP 2025.*

We have also produced one patent application that has been accepted by Huawei representatives:

1. Nikolai Kudasov, Danila Danko, Yegor Bugayenko. Two-phased Intermediate Representation Optimizer.

4 Discussion and Summary

At the moment, Normalizer is a prototype rewrite-engine for φ-calculus, that can be used to optimize programs. Rewrite-based program transformations are usually easier to implement and maintain compared to transformations directly built into a compiler, but the ergonomics of such engine must be complemented by a good integration into a compiler. Normalizer works with φ-calculus, making it universally applicable to any language that has integration with EO. Unfortunately, such integrations are not mature yet, and we are not able to verify the full potential of the Normalizer on, say, Java programs. In particular, our examples for Stream API optimizations are not yet verifiable on real programs. Although this was not a goal of the project, we wish that EO ecosystem becomes more stable and feature-rich to support such applications.

There are also several directions of improvement for the Normalizer tool:

1. Introduction of a type system into the rules and patterns would allow for more feature-rich transformations and optimizations. However, at the moment there is no consensus on what type system would suit EO and φ-calculus for best results.
2. The core of Normalizer is a general-purpose rewrite engine, and it should be possible to factor out that core to make it useful for other representations, including LLVM or JVM directly. This might make Normalizer more applicable and useful to other teams.
3. Normalizer has not only formal proof of confluence for selected rules, but also confluence tests for user-defined rules. It should be possible to reveal such test via the command-line interface to the end user, so that they could check if their rules have any unwanted behavior.

References

[1] Yegor Bugayenko. 2024. EOLANG and φ-calculus. arXiv:2111.13384 [cs.PL]

[2] Nikolai Kudasov and Violetta Sim. 2023. Formalizing φ-Calculus: A Purely Object-Oriented Calculus of Decorated Objects. In *Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs* (Berlin, Germany) *(FTfJP '22)*. Association for Computing Machinery, New York, NY, USA, 29–36. https://doi.org/10.1145/3611096.3611103

[3] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635.

[4] Anders Møller and Oskar Haarklou Veileborg. [n. d.]. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. 4 ([n. d.]), 1–29. Issue OOPSLA. https://doi.org/10.1145/3428236

[5] Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A