DCC071 Reproducibility of published results
SSA without Dominance for Higher-Order Programs

Augusto Guerra de Lima

Department of Computer Science, Universidade Federal de Minas Gerais
Belo Horizonte, MG, Brazil

Fall 2026

The assignment

The extra project consists in trying to reproduce some result published in the compiler literature.
To get the five extra points, the student must:

1. Choose a paper presented in a recent compiler related conference;
2. Download the material that the authors have made publicly available for the paper;
3. Run the implementation, trying to reproduce at least one of the experiments in the paper;
4. Write a short report about the entire procedure. This report must contain the url of the material that has been downloaded, plus a brief description of the student experience with it.

1 The paper

I have chosen the paper *SSA without Dominance for Higher-Order Programs* by Roland Leißa and Johannes Griebler, from University of Göttingen, which will be presented at PLDI 2026, June 17th. The preprint is available at: https://doi.org/10.48550/arXiv.2604.09961.

Brief. The paper is based on static analysis concepts, such as SSA form and dominance. The authors introduces λ_G, a graph-based typed λ-calculus designed as a foundation for intermetiate representations. Its motivation is a criticism of dominance, the underlining concept of SSA form in virtually every modern compiler (LLVM, MLIR). In SSA form, every use of a variable must be dominated by its definition, meaning every control-flow path to a use must pass through the definition. While this property is natural for first-order control-flow graphs, dominance becomes imprecise or even undefined for higher-order functions, which are ubiquitous in modern programming languages (closures, functional languages, MapReduce).
The paper proposes replacing dominance with a concept called nesting, derived from free variables, and proves that nesting implies dominance (Theorem 1), but nesting is strictly more precise: it ignores irrelevant control-flow paths and is well-defined even for higher-order programs where no control-flow graph exists.
The authors also claim a log-linearly complexity with program size in practice.

2 Data-Availability

The material is publicly available at Zenodo platform: doi:10.5281/zenodo.19069679.
The artifact accompanying the paper includes all benchmarks, measurement data, and Lean mechanization of all lemmas and theorems. The authors also refer to the MimIR framework, available at: mimir.github.io.

3 Warm up

To get started, the `README.md` file provides step-by-step instructions on how to evaluate the artifacts. I begin by setting up Docker with

```
docker load < mimir.tar.gz
docker run -it --name mimir-run mimir
```

After that, the artifact consists of two parts. The first is the research compiler IR MimIR, which is used to generate the figure and experimental results presented in the paper. The second is the Lean development *lambda-graph*, which verifies all lemmas and theorems in the paper.
The Lean component is straightfoward, took few minutes, and follows the instructions provided in `lambda-graph/README.md`. The MimIR component offers multiple benchmarking options. Running the full benchmark suite requires more than 15 hours on a modern computer. For this reason, the benchmark script provides options that reduce execution time.
For this evaluation, I selected the first configuration, which measurements are only done up to $2^{10}$and took one hour. According to the `README.md`, the remaining configurations eventually cause the Immer and `std::set` implementations to run out of memory in systems with less than 32GiB RAM.
While this configuration provides less precise measurements, `README.md` states that it is sufficient to reproduce the trends presented in the paper's figure , and is appropriate for artifact evaluation assignment.

Setup. The experiments reported in the paper were conducted on a CachyOS system equipped with an AMD Ryzen AI 9 HX PRO 370 and 32GiB RAM, but I reproduced the experiments on a Void Linux system with an Intel Core i5-1135G7 and 16GiB RAM, using Ubuntu-based Docker environment provided by the artifact.

4 Artifact evaluation

This section summarizes the entire procedure.

4.1 MimIR

The MimIR component of the artifact is the C++ compiler framework in which all algorithms from the paper are implemented. I ran:

```
cd mim
./build_all.sh                                          #1
./run_benchmarks.py                                     #2
docker cp mimir-run:/home/ubuntu/mim/tex/bench.pdf .    #3
```

The build step (1) compiles three separate sets of binaries – one for each set implmentation under evaluation: the indexed trie, the Immer persistent library, and the standart `std::set`. This takes 10 minutes and only needs to be done once.
Step (2) executes the reduced benchmark configuration. The script generates the benchmark figure automatically from the collected data and writes it, extracted from the container as in (3).

Benchmark. The benchmark evaluates three synthetic programs: Loop Cascade, Accumulating Loop Cascade , and Loop Nest – across three operations: free variable computation, β-reduction, and nesting tree construction. For each operation and program, the three set implementation are compared, and the first two operations are also compared against LLVM's dominance analysis and inliner.

The figure I obtained (Figure 1 in this report) closely matches Figure 11 in the paper. The key trends are cleary visible, relative orders and asymtotic trends are consistent.

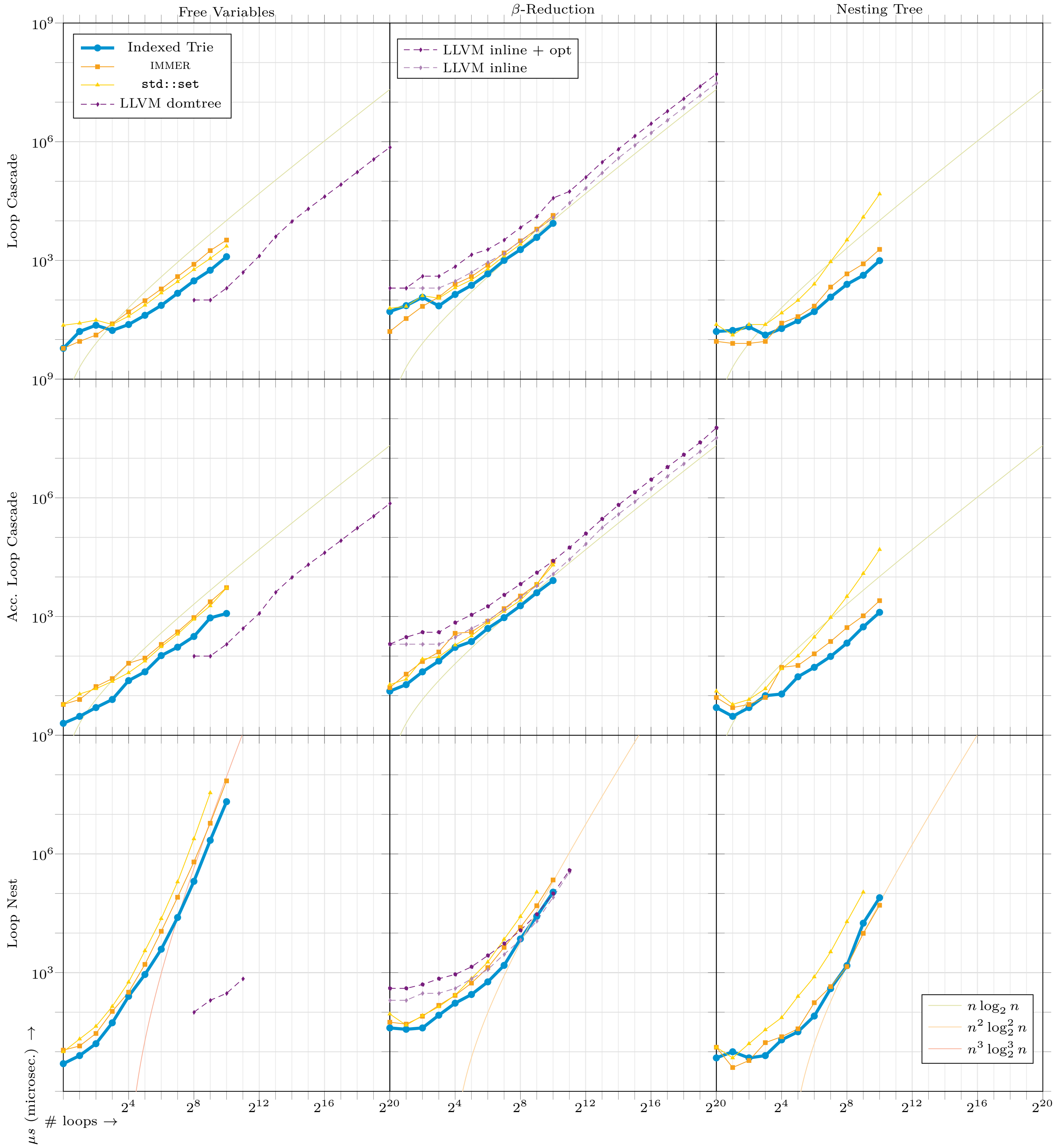

Figure 1: Performance (in μs, lower is better) of three operations–free variable computation, β-reduction, and nesting tree computation–on three synthetic programs with increasing number of loops. The plots compare three different set implementations: the indexed trie, IMMER, and `std::set`. Additionally, free variable computation is compared against LLVM's dominance analysis; β-reduction is compared against LLVM's inliner, both with and without optimizations that simulate MimIR's on-the-fly optimizations. For reference, $10^3\mu\text{s} = 1\text{ms}$, $10^6\mu\text{s} = 1\text{s}$.

Results. For Loop Cascade and Accumulating Loop Cascade, all three MimIR implementations exhibit $\mathcal{O}(n \log n)$ scaling. Indexed trie is fastest. LLVM's dominance analysis is several times faster than MimIR's free-variable computation (as expected, since LLVM uses a near-linear algorithm), but MimIR's β-reduction is faster than LLVM's inliner in most configurations.
For Loop Nest, the behavior is more expensive, consistent with $\mathcal{O}(n^3 \log^3 n)$ scaling for free-variable computation, because loop connectedness grows with program size. But, as the paper argues, deeply nested loops beyond tens of levels are essentially absent in real-world programs.

4.2 λ_G Lean mechanization

The Lean component verifies the metatheory of λ_G. I ran:

```
cd ../lambda-graph
lake build
```

Completed in a few minutes, producing 7 compilation jobs with no errors.

```
Build completed successfully (7 jobs).
```

I audit the Lean mechanization, to confirm the absence of admitted proofs:

```
grep -rn "sorry" .
```

Which returned no results, meaning every proof is complete.

I was interested in getting an overview of the theorems verified, I ran `grep -n "theorem"` across all Lean files in `LambdaGraph/` directory:

```
grep -n "theorem" LambdaGraph/*.lean
```

Which returned (in short) :

```
...
LambdaGraph/Basic.lean:547:theorem Program.validRefs_setBody {p : Program} {i : Nat}
{b : Expr}
...
LambdaGraph/Nest.lean:474:theorem Program.dominates_of_nestsEq {p : Program} {m n k :
Nat}
...
LambdaGraph/Step.lean:192:theorem Computation.soundness {c c' : Computation} {t : Ty}
(ht : ⊢ c : t)
...
LambdaGraph/Subst.lean:2060:theorem Computation.subst_types_and_wf {c : Computation}
{n : Nat} {v : Expr}
```

For example:

Theorem 1 (Nesting-Dominance) : it is verified in `Nest.lean`, if a program is well-formed and l_0 nests l_1 which nests l_2, then l_1 dominates l_2 in the induced control-flow graph.

Theorem 2 (Progress) : it is verified in `Step.lean`, a closed, well-typed expression is either a value or can take a reduction step. This means λ_G does not get stuck on well-typed programs.

Theorem 3 (Preservation) : it is also verified in `Step.lean`, reduction preserves typing and well-formedness. Together with (Theorem 2), this establishes type soundness of λ_G.

Formalization is nice. As described in `lambda-graph/README.md`, the Lean formalization differs from the paper in a few deliberate simplifications: let-expressions are omitted, they are treated as syntatic sugar, the branch function is replaced by a conditional expression, and some theorems have fewer assumptions than originally stated, as those assumptions turned out to be unnecessary. These are not gaps, they represent refinements made during fomalization.

5 Considerations

This extra assignment consisted of evaluating the artifact accompanying the chosen paper. The experience was great.

On the technical side, the artifact was well-organized and the provided Docker environment made the setup easy. On the academic side, this assignment gave me the opportunity to engage with a research paper again, that revisits topics covered in the DCC071 course. Concepts such as SSA form, the dominator tree, type systems, formal semantics, and the progress and preservation properties, saw in the second half of the course. I'm glad because DCC071 gave me the background to read and understand this paper, which I consider one of the valuable outcomes of studying static analysis.

I intend to pursue research in compilers and programming languages and it was nice to engage with this work as part of my undergraduate formation.