Incremental Computing by Differential Execution (Artifact)

Prashant Kumar
JGU Mainz, Germany

André Pacak
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

This artifact supports the paper titled "Incremental Computing by Differential Execution" accepted at ECOOP 2025. It provides a mechanized formalization in Rocq of the differential semantics and optimizations presented in the paper, along with a reference implementation of the differential interpreter in Scala. The artifact includes the Bellman-Ford benchmark used for the performance evaluation in Section 7. Both components are packaged as Docker images for ease of use and reproducibility, enabling verification of all claims made in the paper.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Formal language definitions; Software and its engineering → Formal methods; Software and its engineering → Incremental compilers

Keywords and phrases Incremental computing, differential semantics, programming language design, formal verification, big-step semantics

Digital Object Identifier 10.4230/DARTS.11.2.13

Acknowledgements This work is supported by ERC grant AutoInc (`https://doi.org/10.3030/101125325`).

Related Article Prashant Kumar, André Pacak, and Sebastian Erdweg, "Incremental Computing by Differential Execution", in 39th European Conference on Object-Oriented Programming (ECOOP 2025), LIPIcs, Vol. 333, pp. 20:1–20:24, 2025. `https://doi.org/10.4230/LIPIcs.ECOOP.2025.20`

Related Conference 39th European Conference on Object-Oriented Programming (ECOOP 2025), June 30–July 2, 2025, Bergen, Norway

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2025 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact enables verification of all experimental claims and theoretical results presented in the paper. Running the artifact components allows verification of:

- **Formal Correctness (Sections 3–5):** The differential semantics' completeness, validity, and consistency, along with the admissibility of all optimization rules.
- **Performance Improvements (Section 7):** The significant speedups achieved by differential execution compared to recomputation, and the initialization overhead tradeoff.

All verification steps are detailed in the Appendix, with a mapping between specific paper claims and artifact components provided in Section B.

2 Content

The artifact is distributed as Docker images for both AMD64 and ARM64 architectures: one set for the Rocq formalization and one set for the Scala implementation. Each image can execute its component end-to-end and then drops you into an interactive shell, so you can inspect the code, rerun proofs or benchmarks, and explore every step yourself.

2.1 Rocq Formalization

The Rocq formalization is contained in file `Imp.v`, which contains definitions and proofs that directly correspond to paper sections. Readers can jump directly to the artifact identifiers shown below and inspect them or rerun any proof.

- **Section 3 of the paper (Values and Changes):** Properties like `patch_diff`, `diff_elim`, `store_diff_patch`, `Δstore_diff_elim`, with typing definitions (`value_type`, `valid_change`) in the Rocq formalization.
- **Section 4 of the paper (Differential Semantics):** Definitions `exec` and `Δexec`, with theorems:
  - Theorem 4.1: `Δexec_completeness`
  - Theorem 4.2: `Δexec_validity`
  - Theorem 4.3: `exec_consistent`
- **Section 5 (Optimizations) of the paper:** Various optimization proofs as well as the formalization components of Rocq are as shown below.
  - No-Change: `NoChangeExpr`, `NoChangeStmt`, `exec_NoChange`
  - Idempotence: `exec_repeat_idempotence`, `Δexec_repeat_idempotence`
  - Loop Incr/Decr: `exec_NoChange_repeat_incr`, `exec_NoChange_repeat_decr`
  - Branch Switching: `exec_if_assign_TrueFalse`, `exec_if_assign_FalseTrue`

2.2 Scala Differential Interpreter:

Our differential interpreter implementation in Scala is organized into modules that implement and test the paper's concepts:

- `library/`: Core abstractions for changes, diffing, and operators
- `lang/`: IMP language definition (`Syntax.scala`), parser (`Parser.scala`), and static analyses (`TypeChecker.scala`, `WriteBeforeReadChecker.scala`)
- `interp/`: Interpreter implementations (`StandardInterpreter`, `BaseDifferentialInterpreter`) and testing infrastructure
- `benchmark/`: Bellman-Ford implementation with performance measurement infrastructure that reproduces Figures 6 and 7

Both components are packaged as Docker images for easy reproducibility and come with comprehensive documentation.

3 Reusability and Extensibility

Beyond verifying the paper's results, the artifact's design facilitates reuse and extension. The evaluation committee can consider the following scenarios:

- **Extending Language Features:** The modular architecture of the Scala interpreter (`lang/`, `interp/`) supports the addition of new language constructs. Researchers can define syntax, standard semantics, and differential rules for new features (e.g., while loops, records, arrays, objects) following the established patterns in the existing implementation.
- **Implementing Additional Optimizations:** The framework allows defining and verifying new differential execution optimizations beyond those in Section 5. This process follows our established methodology: implementing semantic rules in the interpreter and extending the Rocq formalization to prove their admissibility.
- **Applying to Other Algorithms:** The differential interpreter executes any valid program in the supported IMP dialect. Researchers can implement and analyze algorithms beyond Bellman-Ford to study their incremental behavior, leveraging the benchmarking infrastructure we provide. The interpreter's API (`BaseDifferentialInterpreter`) facilitates programmatic integration.
- **Extending Formal Verification:** The Rocq formalization in `Imp.v` provides a verified foundation that can be extended to prove properties of new language features, optimizations, or alternative differential semantics, following the pattern of our existing proofs.
- **Reusing Core Components:** The implementation of core abstractions like change representation (`Change`) and the diffing logic (`Diffing`) in the `library/` module can be reused in other incremental computation projects in Scala, independent of our specific interpreter.

The comprehensive documentation, and modular structure facilitate these reuse scenarios.

4 Getting the artifact

The artifact archive is permanently available on Zenodo: `https://doi.org/10.5281/zenodo.15363328`. This includes platform-specific tar file versions of our docker images for both AMD64 and ARM64 architectures, which can be used to reproduce the results of our experiments.

Alternatively, pre-built Docker images are available on Docker Hub:

- Rocq formalization (AMD64): `prashantkumar10011989/coq-formalization:amd64`
- Rocq formalization (ARM64): `prashantkumar10011989/coq-formalization:arm64`
- Scala implementation (AMD64): `prashantkumar10011989/autoinc-scala-image:amd64`
- Scala implementation (ARM64): `prashantkumar10011989/autoinc-scala-image:arm64`

Source code repositories are also available with extensive README files and examples:

- Rocq formalization:
  `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-formalization-ecoop25`
- Scala implementation:
  `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-implementation-ecoop25`

5 Tested platforms

Specifics of the hardware and software on which the testing was carried out is provided in the table below.

System Requirements

Hardware	
RAM	At least 24GB to run the full benchmarks
Disk Space	About 8 GB for both Docker images plus 20MB for results
Software	
Docker	Any recent version (20.10+) with sufficient memory allocation
Tested On	
Operating System	macOS 15.1.1 (64-bit) with Apple M4 Pro chip, 24GB RAM

Docker Image Contents

Docker images are provided for both AMD64 and ARM64 architectures and encapsulate all required dependencies:

Rocq Images	Rocq Proof Assistant v8.18.0 on Debian base image (available for both AMD64 and ARM64)
Scala Images	Scala 3.4.1, SBT 1.9.7, OpenJDK 23.0.1, Python 3.10 with matplotlib and pandas libraries on Ubuntu base image (available for both AMD64 and ARM64)

Note: Performance results may vary depending on system hardware. The key verification goal is to reproduce the relative performance trends shown in the paper, not necessarily the exact timing values.

6 License

This artifact is available under the MIT License.

7 MD5 sum of the artifact

8700e93eb87a4a3a80a6d3cb9024c7dc

8 All MD5 Checksums

- coq-formalization-amd64.tar: c9aec99bf3cf7c8d2dab23d6d82ff1a7
- coq-formalization-arm64.tar: 373ab922b09c0735b3ad41470f8bd4b0
- autoinc-scala-image-amd64.tar: e48cfeef7c22dd13e5d456b01c9fc186
- autoinc-scala-image-arm64.tar: a2d08b1bb84c59c81dc6bd8f5e6509bb

9 Size of the artifact

Approximately 3.7 GB total (across all platform-specific images).

A Artifact Evaluation Instructions

This section provides instructions for evaluating the artifact components.

A.1 Preparing the Environment

1. Ensure Docker is installed and running with at least 24GB memory allocation.
2. Create a local directory for storing benchmark results as shown below:

```
mkdir -p results
```

3. Load the appropriate Docker images from the provided platform-specific `.tar` files:
For AMD64 Architecture:

```
docker load -i coq-formalization-amd64.tar
docker load -i autoinc-scala-image-amd64.tar
```

For ARM64 Architecture:

```
docker load -i coq-formalization-arm64.tar
docker load -i autoinc-scala-image-arm64.tar
```

Alternatively, you can also pull the pre-built Docker images from Docker Hub:
For AMD64 Architecture:

```
docker pull prashantkumar10011989/coq-formalization:amd64
docker pull prashantkumar10011989/autoinc-scala-image:amd64
```

For ARM64 Architecture:

```
docker pull prashantkumar10011989/coq-formalization:arm64
docker pull prashantkumar10011989/autoinc-scala-image:arm64
```

A.2 Rocq Formalization Verification

This step verifies the mechanized proofs corresponding to Sections 3-5 of the paper.
For AMD64 Architecture:

```
docker run --platform linux/amd64 -it coq-formalization:amd64
```

For ARM64 Architecture:

```
docker run -it coq-formalization:arm64
```

Expected outcome: The container automatically compiles and verifies the Rocq proof script (Imp.v). You should see compilation messages ending successfully without errors. The script will then drop you into an interactive shell within the container.

Estimated time: 30-60 seconds for compilation.

Sanctity Check and Optional Exploration: At the shell prompt, you can check the following. This allows the reviewers to be assured of the sanctity of the artifact.

- View the proof file: `less Imp.v`
- Re-check proofs: `coqc -q Imp.v`

- Validate the compiled object: `coqchk Imp.vo`
- Start an interactive session: `coqtop`
- Exit the container: `exit`

A.3 Scala Interpreter & Benchmark Verification

This container runs the interpreter tests and the Bellman-Ford benchmark (Section 7).

A.3.1 Short Run

This runs core tests and a minimal version of the benchmark suite to quickly verify functionality. We do this by setting the `SHORT_BENCHMARK` environment variable to `true`. To make this faster, rather than performing the experiments on graphs of size from 10,20,. . ., 150 we do so only till 90, that is, 10,20,. . ., 90.

For AMD64 Architecture:

```
docker run -it --platform linux/amd64 \
  -e SHORT_BENCHMARK=true \
  -v "$PWD/results:/results" autoinc-scala-image:amd64
```

For ARM64 Architecture:

```
docker run -it \
  -e SHORT_BENCHMARK=true \
  -v "$PWD/results:/results" autoinc-scala-image:arm64
```

Expected outcome: There are two outcomes:

1. A test suite associated with the differential interpreter which includes property based testing for various examples is executed.
2. The container executes tests and a small benchmark subset. It generates simplified versions of the plots from the Experiment Evaluation section of our paper (Section 7) and copies them to your local `./results` directory. These plots are named appropriately, that is, `Figure_6_a_paper.pdf` is the Figure 6(a) from the paper. However, note that due to simplification, we would not be able to look at the whole trend. To do so, we will have to do a full run of the benchmarks. In the `./results` directory you will also find the .csv file resulting from the experiment which are then used to draw the plots seen in Section 7.
   - `Figure_6_a_paper.pdf` (Figure 6a: Differential vs Baseline - Change i)
   - `Figure_6_b_paper.pdf` (Figure 6b: Differential vs Baseline - Change ii)
   - `Figure_7_a_paper.pdf` (Figure 7a: Initialization vs Baseline - Change i)
   - `Figure_7_b_paper.pdf` (Figure 7b: Initialization vs Baseline - Change ii)

Estimated time: Approximately 8 minutes.

Sanctity Check and Optional Exploration: After the benchmarks complete, the container will drop you into an interactive shell where you can inspect the various files present there. In particular, we have provided a simple example file (SampleTest.scala) to explain the working of our differential interpreter. This file is also shown in README file associated with the git lab repo of the artifact of our differential interpeter. You can inspect this file in the shell as well as run it. Additonally, you can execute the test suite in the shell by executing the shell script `runTests.sh`. A few concrete examples of things you can do in the shell to help with evaluating the artifact:

- View sample test: `cat interp/src/test/scala/autoinc/interp/SampleTest.scala`
- Run sample test: `sbt "interp/testOnly autoinc.interp.SampleTest"`
- Explore implementation:
  `cat interp/src/main/scala/autoinc/interp/BaseDifferentialInterpreter.scala`
- Run additional tests: `sbt interp/test or ./runTests.sh`
- Exit the container: `exit`

A.3.2 Full Run (Reproducing Paper Results)

This executes the complete benchmark suite necessary to reproduce the performance results presented in the paper.

For AMD64 Architecture:

```
docker run -it --platform linux/amd64 \
  -e SHORT_BENCHMARK=false \
  -v "$PWD/results:/results" autoinc-scala-image:amd64
```

For ARM64 Architecture:

```
docker run -it \
  -e SHORT_BENCHMARK=false \
  -v "$PWD/results:/results" autoinc-scala-image:arm64
```

Estimated time: About 1 hour on a modern machine with the aforementioned configuration in Section 4.

B Mapping to Paper Claims

The artifact components support the paper's claims as shown in Table 1. Additionally, Note that running the full Scala benchmark (A.3.2) automatically executes all test cases, including those for the optimizations described in Section 5, verifying their correct implementation.

C Troubleshooting

- **Memory Issues:** Ensure Docker has adequate memory allocated (at least 24 GB for Scala full run).
- **Slow Benchmark Performance:** The full benchmark duration depends heavily on CPU. The short run should be reasonably fast.

Table 1 Mapping between paper claims and corresponding artifact components.

Paper Section/Figure/Theorem	**Artifact Component with Location**	**Verification Method**
Sec 3, 4, Thm 4.1-4.3	Rocq formalization: theorems for completeness, validity, and consistency in `Imp.v`	Run Rocq Verification (A.2)
Sec 5 (Optimizations)	Rocq formalization: admissibility proofs for No-Change, Loop, and Branch Switching optimizations in `Imp.v`	Run Rocq Verification (A.2)
Sec 7, Fig 6, Fig 7	Scala benchmark suite: Bellman-Ford implementation in `autoinc-scala/benchmark`	Run Scala Full Benchmark (A.3.2); check generated PDF files in `./results/`

- **Platform Compatibility:** Use the appropriate Docker image for your system architecture (AMD64 or ARM64). If unsure about your architecture, you can check with `uname -m` (x86_64 indicates AMD64, arm64 or aarch64 indicates ARM64).
- **Exiting Container Early:** Use Ctrl+C to stop the running process.