Rewriting Dataflow Analyses with a Datalog Engine

Matthias Yee, Arjan Chakravarthy, Ignas Karvelis • Brown University • CSCI 1951Q

MOTIVATION

Dataflow analyses are critical for compiler optimizations. Datafrog provides greater correctness guarantees and speed at sclale for larger programs.

```
worklist = [...]
while worklist:
  loc = pop()
  live = compute()
  if changed:
    add_preds()
// 100+ lines
```

✗ Manual

→

```
live_at(L, x) :-
  use(L, x)

live_at(P, x) :-
  cfg(S, P),
  live_at(S, x),
  !def(P, x)
```

✓ Datafrog

APPROACH

Reimplemented 4 core analyses for Rice compiler using Datafrog:

- ✓ Constant Propagation
- ✓ Liveness Analysis
- ✓ Pointer Analysis
- ✓ Taint Analysis

Rice IR → Encode Facts → Datafrog Rules → Results

KEY TECHNIQUE: FACT ENCODING

Datafrog requires integer-indexed relations. Every domain value needs interning:

Rice IR

```
Location {
  block: bb0,
  instr: 2
}
let x = 42;
```

⇒

Datafrog Facts

```
LocationMap:
  5 ↔ (bb0, 2)

cfg(5, 6)
assign(5, x_id, 42_id)
```

CHALLENGE: FIELD-SENSITIVE POINTER ANALYSIS

Complex place expressions must be flattened into primitive operations:

Code

```
x.f.g = y.h
```

→

Flattened Ops

```
v1 = load x
v2 = load v1.f
v3 = load y
v4 = load v3.h
store v2.g = v4
```

Datalog Relations

```
// var points to alloc
var_points_to(v1, alloc_x)
// heap field points to target
heap_points_to(alloc_x, field_f, alloc_f)
var_points_to(v2, alloc_f)
```

RESULTS

✓ Correctness
All functionality tests match reference implementation

Performance Comparison

Test	Reference Time (s)	Datafrog Time (s)
Test 1	21.307	**0.121**
Test 2	36.447	**0.164**
Test 3	17.834	**0.156**
Test 4	18.641	**0.138**
Test 5	13.099	**0.137**

Key Findings: Datafrog achieved 100-200× speedup across all analyses tests, with execution times consistently under 0.2 seconds compared to the reference implementation's 13-36 second range. The declarative Datalog approach dramatically improved performance through efficient join operations.

Rewriting Dataflow Analyses with a Datalog Engine

Matthias Yee, Arjan Chakravarthy, Ignas Karvelis
Brown University

December 2025

1 Introduction

Dataflow analysis is a cornerstone of modern compilers, enabling optimizations from constant folding to register allocation. In this project, we built these analyses for the Rice compiler, a compiler for a simple imperative language with functions, closures, tuples, and arrays. Rice compiles to bytecode and includes a runtime.

Implementing dataflow analyses is notoriously challenging. Consider constant propagation: you must implement a worklist algorithm, carefully manage program points, compute meet operations at merge points, and handle the control flow graph correctly. The analysis logic ("which values are constant?") gets tangled with the fixpoint computation ("keep iterating until nothing changes"). This complexity multiplies as analyses grow more sophisticated, for example, pointer analysis with field sensitivity or taint analysis with control dependencies require even more intricate bookkeeping.

Datalog offers an alternative. Datalog is a declarative logic programming language where you express what you want to compute, not how to compute it. A Datalog program consists of facts (base truths) and rules (logical implications). The engine handles fixpoint computation automatically using semi-naive evaluation, which efficiently tracks new derivations to avoid redundant work.

Recent systems suggest Datalog's potential for program analysis. Doop revolutionized Java pointer analysis by expressing complex analyses as succinct Datalog specifications. The Rust compiler's Polonius project uses Datafrog (a Rust implementation of Datalog) for borrow checking. These successes motivated our central question: *can Datafrog power all the core dataflow analyses in a compiler?*

We reimplemented four fundamental analyses in our Rice compiler using Datafrog [1]:

1. **Constant propagation**: tracks which variables hold constant values
2. **Liveness analysis**: determines which variables may be read in the future
3. **Pointer analysis**: computes what each pointer may point to (with field sensitivity)
4. **Taint analysis**: tracks information flow from sources to sinks

These analyses span the spectrum of dataflow complexity: forward vs. backward, intraprocedural vs. interprocedural, and whether or not the analysis is flow, field, or context sensitive. Our implementations demonstrate that Datafrog can express all of them clearly and correctly.

[1] Our code can be found here

2 Implementation

2.1 Architecture Overview

Each analysis follows a common pattern:

1. **Fact generation**: Extract relevant information from the Rice IR (intermediate representation) and encode it as integer-indexed relations
2. **Datalog rules**: Express analysis logic as relations and derivation rules
3. **Fixpoint computation**: Let Datafrog iterate to a fixpoint using its built-in semi-naive evaluation
4. **Result extraction**: Decode integer indices back to Rice IR types and use results for optimization

The trickiest part is step 1. Datafrog operates on relations of small integer tuples for performance, you can't directly store Rust enums or complex types. This means every analysis needs an *interner*: a bidirectional map between domain values (locations, variables, allocation sites) and small integers.

2.2 Challenge 1: Encoding the Control Flow Graph

Our first hurdle was representing program locations. Rice's IR is structured as basic blocks containing statements, each with a unique `Location` identifying a block and instruction index. For Datafrog, we needed integer IDs.

We considered addressing this challenge by using the instruction index, but instruction indices are only unique within a block, not globally. We initially tried combining block and instruction indices, but this created enormous sparse indices that wasted memory in Datafrog's dense bitvectors.

Our solution was a `LocationMap` that assigns a dense sequence of integers to locations:

```
pub struct LocationMap {
    to_id: HashMap<Location, u32>,
    to_loc: HashMap<u32, Location>,
}
```

This map is built once by traversing all locations and assigning sequential IDs. With this, we could efficiently encode CFG edges as `(u32, u32)` pairs.

For constant propagation and liveness, encoding the CFG was straightforward, we just emit a tuple for each edge. But for liveness (a backward analysis), we had to *reverse* all edges: `cfg_facts.push((v, u))` instead of `(u, v)`. This lets us write forward-style rules that naturally propagate backward.

2.3 Challenge 2: Field-Sensitive Pointer Analysis

Pointer analysis was the most complex implementation. We needed to track not just what variables point to, but also what fields of heap objects point to. For example:

```
tuple = (x, y);
p = tuple;
a = p.0;  // Load from field 0
p.1 = z;  // Store to field 1
```

The naive approach would be to treat each field access as a separate allocation. But that doesn't work as you need to track that `p.0` and `p.1` are projections from the same allocation.

Our solution uses three relations:

- `var_points_to(var, alloc)`: Variable points to allocation
- `heap_points_to(alloc, field, target)`: Field of allocation points to another allocation
- `assign/load/store`: Primitive operations derived from complex expressions

The key insight is to *flatten* complex place expressions. When we see `x.f.g = y.h`, we generate:

1. Load `x` to get allocation A_x
2. Load field `f` of A_x to get allocation A_f (via `heap_points_to`)
3. Load `y` to get allocation A_y
4. Load field `h` of A_y to get allocation A_h
5. Store A_h into field `g` of A_f

Each step is a Datalog rule. The load rule demonstrates the key pattern:

```
// load_base_resolved: ((base_alloc, field), dest)
// heap_keyed: ((alloc, field), target)
// Join to propagate: dest now points to target
var_points_to.from_join(
    &load_base_resolved,
    &heap_keyed,
    |&(_alloc, _field), &dest, &target| (dest, target)
);
```

We also needed careful field interning. Initially, we interned fields with their full type information (`ProjectionElem::Field {index: 0, ty: Int}`). But this caused spurious mismatches, where a field 0 of type `Int` didn't match field 0 of type `Unknown` even though they're the same field structurally. We fixed this by normalizing all field types to `Type::unit()` during interning:

```
fn get_field_id(&mut self, field: &ProjectionElem) -> u32 {
    let key = match field {
        ProjectionElem::Field { index, .. } =>
            ProjectionElem::Field {
                index: *index,
                ty: Type::unit()
            },
        // Similar for Index and other variants
        other => other.clone(),
    };
    // ... intern the normalized key
}
```

2.4 Challenge 4: Implicit Flows in Taint Analysis

Taint analysis tracks information flow from sensitive sources (e.g., password input) to public sinks (e.g., print statements). The explicit flows as direct data dependencies are straightforward. The challenge is *implicit flows* through control dependencies.

Consider this code:

```
password = secure();  // Tainted source
if password == "secret" {
    x = 0;
} else {
    x = 1;
}
println(x);  // Implicit leak!
```

Even though `password` never directly flows to `x`, the value of `x` reveals information about `password`. This is an implicit flow via control dependency.

Computing control dependencies required building a post-dominator tree. We construct the reverse CFG, compute dominators, then determine: block B is control-dependent on block A if A has a successor that B post-dominates, but B doesn't post-dominate A itself.

The Datafrog encoding creates flow facts for both explicit and implicit flows:

```
// Explicit: src flows to dst
for s in &src_aliases {
    for d in &dst_aliases {
        flow_facts.push(((loc_id, s_id), (succ_id, d_id)));
    }
}

// Implicit: condition flows to control-dependent statements
if let Some(cdeps) = control_dependencies.get(&loc.block) {
    for dep_block in cdeps {
        for c_alias in &condition_aliases {
            for d in &dst_aliases {
                flow_facts.push(((loc_id, c_id), (succ_id, d_id)));
            }
        }
    }
}
```

The Datalog fixpoint then propagates both kinds of taint transitively. When checking sinks, we not only check if arguments are tainted, but also traverse transitive control dependencies to detect implicit flows.

3 Evaluation

To evaluate our Datafrog implementations of the different dataflow analyses, we directly compared the output (for correctness) and speed against a reference implementation that manually runs the worklist algorithm. We hypothesized that, on larger programs, our Datafrog implementations would generally be faster. In this section, we separately evaluate each of our different dataflow analyses (constant propagation, dead code elimination, pointer analysis, and taint analysis) on a wide array of programs.

3.1 Procedural Test Generation

In order to test whether our Datafrog implementation was more efficient than the reference implementations, we needed to create very large programs. To this end, we wrote a script to procedurally generate Rice programs with a controllable number of field accesses (for tuples and arrays). Specifically, we can control the number of functions generated and the nesting depth of each field access; we set the former to 5 and the latter to 8 for the tests below. This tooling would enable us to test our analyses on significantly larger programs where we suspect our Datafrog implementation should prove faster.

3.2 Results

For each of the four Dataflow analyses, we compare the output (bytecode produced) and the time required to run each test. In Table 1, we show the results of running both implementations on the procedurally generated pointer analysis tests. For all tests, the bytecode produced by the Datafrog implementation matched the bytecode produced by the reference dataflow implementation.

All Optimizations		
Test	Reference Time (s)	Datafrog Time (s)
Test 1	21.307	0.121
Test 2	36.447	0.164
Test 3	17.834	0.156
Test 4	18.641	0.138
Test 5	13.099	0.137

Table 1: Runtime on procedurally generated pointer analysis tests on both the reference compiler and Datafrog implementation.

As demonstrated in Table 1, our Datafrog implementation is several orders of magnitude faster than the implementation without Datafrog.

We also evaluate the time taken for each Datafrog dataflow analysis separately. The tables below illustrate the times for each analysis. In each of these cases, we run a single analysis and omit the other analyses to evaluate each optimization separately.

Constant Propagation and Dead Code Elimination		
Test	Reference Time (s)	Datafrog Time (s)
Test 1	56.288	0.428
Test 2	22.077	0.205
Test 3	122.06	0.343

Table 2: Runtime on procedurally generated tests, running just the Constant Analysis and Dead Code Elimination analyses.

Table 2 illustrates the times for only running the Constant Propagation and Dead Code Elimination optimization passes. We procedurally generate 3 tests, following a similar paradigm explained aforementioned, just with a field access depth of 10. We note that the times for running the implementation without Datafrog is significantly slower than the times with Datafrog. We suspect this relates to the overhead of fixpoint convergence required in the reference implementation compared to the Datafrog implementation which is based on joins.

Pointer Analysis		
Test	Reference Time (s)	Datafrog Time (s)
Test 1	0.140	0.065
Test 2	0.086	0.048
Test 3	0.217	0.087

Table 3: Runtime on procedurally generated tests, running just the Pointer Analysis analysis.

Table 3 illustrates the times for exclusively running the pointer analysis. The runtimes for the pointer analysis pass in the reference implementation is significantly closer that of the Datafrog implementation,

though the Datafrog version still remains faster. On our entire test suite, the bytecode output and the final program output was consistent between our two implementations, guaranteeing correctness.

3.3 Discussion

The core benefit of Datafrog is *development velocity and correctness confidence*. Writing correct dataflow analyses is hard. Datafrog eliminates entire classes of bugs by handling fixpoint computation automatically.

The primary cost is flexibility. Datafrog's relational model is less flexible than custom data structures. When we needed intersection-based meet for constants or transitive control dependencies for taint, we had to drop out of pure Datalog. For each of our analyses, we had to create several intermediate variables to allow us to perform the joins due to limitations of Datafrog.

The ideal scenario might be a hybrid: use Datafrog for rapid prototyping and validation, then (if needed) hand-optimize analyses while keeping the Datafrog version. This matches how Rust's Polonius uses Datafrog: the primary borrow checker uses custom algorithms, but Polonius validates their correctness.

3.4 Reflection on Rice

Working with Rice revealed both strengths and limitations. The bytecode IR is clean and relatively simple, mostly consisting of basic blocks with statements and terminators, using `Place`s for lvalues and `Operand`s for rvalues. This structure maps naturally to dataflow facts.

However, Rice's simplicity sometimes created challenges. For instance, there's no explicit phi nodes at merge points, which meant we had to carefully handle the meet operation for constants outside the dataflow framework. The lack of SSA form also made liveness analysis more involved: without phi functions, we had to carefully track use/def through projection chains.

The pointer analysis had to handle Rice's closures specially. Closures capture their environment, and these captured variables can alias in complex ways. We modeled this by creating bidirectional aliasing between all closure-captured variables, which is conservative but sound.

4 Conclusion

We successfully reimplemented four fundamental dataflow analyses using Datafrog, demonstrating that modern Datalog engines can power compiler analyses. The implementations are more declarative and arguably more correct-by-construction than manual worklist algorithms, though they require careful encoding of domain values to integers and occasional escape hatches for non-monotonic operations.

Datafrog's main strength is separating the analysis specification from the fixpoint computation. The main weaknesses are the implementation complexity and the limited flexibility. However, for compilers where strict correctness is needed (such as research compilers, domain-specific languages, or exploratory analyses) Datafrog is an excellent choice.

Future work could explore incremental computation (reusing analysis results across edits), demand-driven analysis (computing only what's needed for a specific query), or hybrid approaches that use Datafrog for prototyping and custom algorithms for production. As Datalog engines continue to mature, they are likely to become a standard used tool.

5 References

https://github.com/rust-lang/datafrog
https://github.com/plast-lab/doop

https://github.com/rust-lang/polonius