Functional Programming with Datalog

André Pacak
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

Datalog is a carefully restricted logic programming language. What makes Datalog attractive is its declarative fixpoint semantics: Datalog queries consist of simple Horn clauses, yet Datalog solvers efficiently compute all derivable tuples even for recursive queries. However, as we argue in this paper, Datalog is ill-suited as a programming language and Datalog programs are hard to write and maintain. We propose a "new" frontend for Datalog: functional programming with sets called *functional IncA*. While programmers write recursive functions over algebraic data types and sets, we transparently translate all code to Datalog relations. However, we retain Datalog's strengths: Functions that generate sets can encode arbitrary relations and mutually recursive functions have fixpoint semantics. We also ensure that the generated Datalog program terminates whenever the original functional program terminates, so that we can apply off-the-shelve bottom-up Datalog solvers. We demonstrate the versatility and ease of use of functional IncA by implementing a type checker, a program transformation, an interpreter of the untyped lambda calculus, two data-flow analyses, and clone detection of Java bytecode.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Datalog, functional programming, demand transformation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.7

1 Introduction

Datalog is a carefully restricted logic programming language that has seen a surge in popularity in recent years. Originally, Datalog was conceived as a database query language that operates on finite sets only [15], so that all queries are guaranteed to terminate. Nowadays, Datalog is being used in a wide array of applications [12], from program analysis [10, 13, 23] to network monitoring [1] and distributed computing [2, 3]. What makes Datalog so popular is that (i) there are highly efficient and scalable implementations available and (ii) Datalog programs are considered declarative. We argue that the latter is partly a misconception: Datalog's semantics is declarative, but Datalog's frontend is not.

Datalog is often primed as being declarative. This can be surprising given that a Datalog program consists of simple Horn clauses (a_0 :- $a_1, \ldots, a_n$), where a_0 holds if a_1 through a_n hold. In Datalog, a_0 is called the *head* of the rule and $a_1, \ldots, a_n$ form the *body* of the rule. Both head and body consist of *atoms* a, which are of the form $R(t_1, \ldots, t_n)$ for some relation R and terms t. A Datalog solver computes the least fixpoint of the Horn clauses such that the relations R contain all derivable ground tuples, called *facts* in Datalog. In the initial fixpoint iteration, the semantics collects all rule heads a_0 that have no precondition. In subsequent fixpoint iterations, the semantics collects all facts that can be derived by applying rules to previously derived facts. When terms range over finite sets, this fixpoint iteration terminates in finitely many steps. We concur that Datalog has a declarative semantics, because programmers do not need to think about *how* the derivable facts are computed.

The problem of Datalog is its frontend: It is ill-suited as a programming language and not declarative. Consider we want to construct control-flow graphs as a basis for program analysis. Figure 1 shows a functional program and a Datalog program that construct the control-flow graphs for the While language. The functional program uses pattern matching

```
// Functional programming
def flow(stm: Stm): Set[(Stm, Stm)] = stm match {
  case Assign(x, a) => {}
  case Sequence(s1, s2) => flow(s1) ++ flow(s2) ++ {(l1, init(s2)) | l1 in final(s1)}
  case If(c, s1, s2) => c match {
    case True() => flow(s1) ++ {(stm, init(s1))}
    case False() => flow(s2) ++ {(stm, init(s2))}
    case _ => flow(s1) ++ flow(s2) ++ {(stm, init(s1)), (stm, init(s2))} }
  case While(c, s) => flow(s) ++ {(stm, init(s))} ++ {(l,stm) | l in final(s)} }
```

```
// Datalog
flow(Stm, From, To) :- sequence(Stm, Stm1, _), flow(Stm1, From, To).
flow(Stm, From, To) :- sequence(Stm, _, Stm2), flow(Stm2, From, To).
flow(Stm, From, To) :- sequence(Stm, Stm1, Stm2), final(Stm1, From), init(Stm2, To).
flow(Stm, From, To) :- if(Stm, C, Stm1, _), true(C), flow(Stm1, From, To).
flow(Stm, Stm, To) :- if(Stm, C, Stm1, _), true(C), init(Stm1, To).
flow(Stm, From, To) :- if(Stm, C, _, Stm2), false(C), flow(Stm2, From, To).
flow(Stm, Stm, To) :- if(Stm, C, _, Stm2), false(C), init(Stm2, To).
flow(Stm, From, To) :- if(Stm, C, Stm1, _), not true(C), not false(C), flow(Stm1,From,To).
flow(Stm, Stm, To) :- if(Stm, C, Stm1, _), not true(C), not false(C), init(Stm1,To).
flow(Stm, From, To) :- if(Stm, C,_,Stm2), not true(C), not false(C), flow(Stm2,From,To).
flow(Stm, Stm, To) :- if(Stm, C,_,Stm2), not true(C), not false(C), init(Stm2,To).
flow(Stm, From, To) :- while(Stm, _, Stm1), flow(Stm1, From, To).
flow(Stm, Stm, To) :- while(Stm, _, Stm1), init(Stm1, To).
flow(Stm, From, Stm) :- while(Stm, _, Stm1), final(Stm1, From).
```

Figure 1 Constructing control-flow graphs using functional programming and Datalog.

and set-comprehensions to compute sets of edges similar to [16], whereas the Datalog program provides rules to constrain the logic variables `From` and `To`. The most prominent problem with Datalog in this example is the lack of structured programming and the duplication of atoms, especially for *if*-statements: we must query relation `if` 8 times, relation `true` 6 times, and relation `false` 6 times. Such Datalog code is hard to write and maintain. Another problem with programming Datalog is that rules must be *range-restricted*: Each variable in the head of a rule must be bound in the body of the rule. This restriction ensures relations can be computed using Datalog's least fixpoint semantics. For example, the increment relation `inc(X, Y) :- Y=X+1` would be correctly rejected by Datalog solvers such as Soufflé [20], because `X` is not bound in the rule's body. Datalog programmers need to work around this restriction.

So why would programmers want to use Datalog anyways instead of functional programming? Because of Datalog's declarative fixpoint semantics, which makes it easy to process cyclic data structures such as our control-flow graphs from above. For example, we can compute the transitive control-flow reachability with two simple rules:

```
flowTrans(Prog, From, To) :- flow(Prog, From, To).
flowTrans(Prog, From, To) :- flow(Prog, From, Inter), flowTrans(Prog, Inter, To).
```

What is remarkable is that we do not have to implement a termination condition and or detect when the relations are stable; Datalog takes care of that. This is why Datalog is a popular implementation language for data-flow analyses that propagate information along the control-flow graph until reaching a fixpoint [10], despite the shortcomings of its frontend.

In this paper, we design a functional programming language with fixpoint semantics and propose it as a "new" Datalog frontend: *functional IncA*. In particular, we show how functional programs with first-order functions and recursive algebraic data types can be faithfully translated to negation-free Datalog. A key idea of our approach is to systematically track the *demand* on functions: Which inputs must a function be run on to obtain the computation's final result. Since terminating functional programs only consider finitely many

```
def entry_var(stm: Stm, prog: Stm, x: String): Val =
  fold(BotVal(), joinVal, {exit_var(pred, prog, x) | (pred,stm) in flow(prog)})
def exit_var(stm: Stm, prog: Stm, x: String): Val = stm match {
  case Assign(y, exp) =>
    if (x == y)  aeval(exp, stm, prog)
    else         entry_var(stm, prog, x)
  case ... }
def aeval(exp: Exp, node: Stm, prog: Stm): Val = exp match {
  case Var(x) => entry_var(node, prog, x)
  case ... }
```

Figure 2 A data-flow analysis using functional programming with fixpoint semantics.

inputs, we can track these inputs in Datalog relations. For programs without algebraic data types, we can adopt a standard demand transformation [25]. However, for algebraic data types we need to carefully instrument the demand transformation to encode constructors and selectors through finite relations. Our translation preserves the semantics of the functional program and, in particular, the resulting Datalog program terminates whenever the functional program does. The translation targets Datalog with base types and operations on such types such as integers, float, strings, booleans as well as algebraic data types. The most common Datalog dialects such as Soufflé, IncA, Flix and Formulog all support these types. Whenever we reference Datalog we mean Datalog with the extensions listed above.

Functional IncA replaces Datalog's logic programming frontend, but we retain Datalog's key advantages: relations with fixpoint semantics. Specifically, we extend functional IncA with set types and set operations (comprehensions, union, and folds) such that programmers can describe and aggregate over relations. For example, Figure 2 shows a data-flow analysis implemented in functional IncA. The analysis queries the control-flow graph `flow` and propagates information about the value of variables (abstracted as intervals) along the control-flow graph. Note that `entry_var`, `exit_var`, and `aeval` are mutually recursive and that there is no termination condition (the program diverges under standard functional semantics). Despite using functional programming as a frontend, all code compiles to Datalog rules, which is a key advantage of our approach for two reasons. First, programmers can rely on the declarative Datalog semantics to find the least fixpoint. Second, we can use any existing Datalog solver to run the program, whereas prior Datalog dialects usually require a custom Datalog solver.

We have implemented functional IncA as part of the incremental Datalog framework IncA [23]. We translate functional IncA into a Datalog IR and provide two backends: one targeting IncA directly, the other targeting Soufflé [20]. While the IncA backend provides incremental re-evaluation after input changes, the Soufflé backend provides better non-incremental performance. The choice of the backend is transparent for the user of the frontend, except that Soufflé does not support user-defined aggregations. We have implemented three case studies using functional IncA. First, we implemented a type checker for the simply-typed lambda calculus, a type-erasure transformation for the same, and an interpreter for the untyped lambda calculus. While the encoding of type checkers in Datalog has recently been explored [17], we are the first to support program transformations and interpreters for Turing-complete languages in Datalog without relying on an embedded functional programming language. Second, we implemented textbook reaching definitions and interval analyses. Both analyses are flow-sensitive and compute a fixpoint over the control-flow graph. Last, we implement clone detection of Java bytecode which is represented as Soufflé facts. We generate abstract syntax trees by querying Soufflé relations. We then use the abstract syntax trees to determine if two methods are alpha-equivalent in respect to their identifiers and labels. Our case studies show that functional IncA is expressive and easy to use. Early performance measurements indicate that reusing established Datalog solvers yields more efficient execution times.

Practically speaking, we consider functional IncA to be a stepping stone for the compilation of other languages to Datalog. On one hand, our encoding paves the road for transferring years of research on functional programming languages to Datalog. For example, we show in this paper how standard defunctionalization [18] can be used to add first-class functions and first-class relations to functional IncA. Defunctionalization translates first-class functions to first-order functions and algebraic data types, which we can then compile to Datalog. On the other hand, we believe that our methodology for supporting user-defined functions and user-defined data types can be used to compile domain-specific languages to Datalog. We leave this avenue of research for future work.

In summary, we present the following contributions:

- We identify 5 principles that are necessary for the semantics-preserving translation of first-order functions to Datalog. We define the translation formally and adapt a demand transformation. This constitutes the first version of functional IncA (Section 3).
- We show how to compile user-defined algebraic data types to Datalog and extend functional IncA accordingly (Section 4).
- We add sets and set operations to functional IncA, extend the translation, and show how standard defunctionalization can be used to add first-class functions and first-class relations (Section 6).
- We demonstrate the expressiveness and ease of use of functional IncA by implementing a type checker, program transformation, and interpreter for the lambda calculus (Section 5), data-flow analyses for the While language (Subsection 7.1), and clone detection of Java bytecode (Subsection 7.2).
- We provide two backends for functional IncA, one targeting the incremental Datalog solver used by IncA, the other targeting the non-incremental Datalog solver Soufflé (Section 8).

2 Datalog Frontends: State of the Art

We are by far not the first to recognize the shortcomings of Datalog's frontend. Two opposing approaches have been explored in prior work to improve the expressiveness and/or usability of Datalog. We call these approaches *backend-first* and *frontend-first* and discuss them below.

Backend-first approach. The backend-first approach uses existing Datalog solvers as a starting point and extends them with new language features. Usually, extensions considered in the backend-first approach aim to increase the expressivity of Datalog, but sometimes also focus on usability. The backend-first approach has a long tradition in Datalog solvers and some features have become standard nowadays. For example, Datalog solvers support stratified negation and arithmetic operations, even though neither is part of core Datalog [15].

Modern Datalog solvers provide a range of different extensions that their users can choose from. For example, Soufflé [19] provides records, algebraic data types, and user-defined functions; Viatra Query [29], the Datalog solver used by IncA, supports user-defined data types and recursive aggregation over user-defined functions [22, 23]. While all of these features improve the frontend and make Datalog programming easier, the core language design remains the same: Horn clauses.

Horn clauses (a_0 :- $a_1, \ldots, a_n$) encode implications ($a_1 \wedge \ldots \wedge a_n \rightarrow a_0$). We argue Horn clauses are inadequate as a programming language, since they inhibit structured programming and enforce a flat structure. For example, a nested function call `res = f(g(h(x)))` becomes:

```
R(x, res) :- h(x, y), g(y, z), f(z, res)
```

That is, we must flatten the call chain. Or consider an expression that contains nested conditionals `(if (b1) x1 else x2) + (if (b2) x3 else x4)`, which becomes 4 separate Horn clauses:

```
R(b1, b2, x1, x2, x3, x4, res) :-  b1,  b2, res = x1 + x3.
R(b1, b2, x1, x2, x3, x4, res) :-  b1, !b2, res = x1 + x4.
R(b1, b2, x1, x2, x3, x4, res) :- !b1,  b2, res = x2 + x3.
R(b1, b2, x1, x2, x3, x4, res) :- !b1, !b2, res = x2 + x4.
```

These encodings are cumbersome to work with; they make programming and maintenance unnecessarily difficult. We would much rather use functional programming as a frontend.

While the backend-first approach does not fundamentally improve Datalog's frontend, it has one decisive advantage: It leverages existing solvers. These solvers are often the result of years of research and engineering. They automatically optimize Datalog programs, employ highly optimized data structures and algorithms, support profiling and debugging, provide incremental execution, and more. When designing new Datalog frontends, we should aim to reuse these systems. However, the state of the art moves in another direction.

Frontend-first approach. Quite a few recent research projects try to improve the frontend of Datalog by designing new DSLs to be used in its stead. We call these approaches frontend-first because the newly designed frontend is their starting point. In particular, frontend-first approaches do not build on top of an existing solver but develop a new solver specific to the newly designed frontend. This allows for great flexibility in the frontend's design.

For example, Flix [14] provides a Datalog frontend extended with lattices and monotonic functions. Flix embeds its Datalog frontend into a functional programming language, where constraints are first-class and can be generated at run time [13]. Formulog [9] provides a Datalog frontend extended with a data type for constructing SMT formulas and a constraint for solving them. While Formulog constraints are not first-class, the Datalog frontend is also embedded into a functional programming language. In both Flix and Formulog, the Datalog constraints can invoke functional code to assert a property or to construct new terms. In Formulog, functional code can also recursively query Datalog relations. While both systems present interesting designs, they also both implement their own Datalog solvers and do not benefit from prior engineering efforts.

Datafun [6] proposes a more drastic redesign for Datalog, namely as a higher-order functional programming language with fixpoint semantics. Datafun functions can accept and produce relations and the language supports the aggregation over lattices. As such, we believe Datafun's frontend is a well-suited replacement for Datalog. However, there are two limiting factors, First, in contrast to other modern implementations of Datalog, Datafun programs are constructor-free and enforce termination. While this equips Datafun with a nicer theory, it is a practical limitation, although one that could be easily eliminated. Second, like Flix and Formulog above, Datafun provides its own Datalog solver and existing optimizations and advances in Datalog engines have to be retrofitted to Datafun. For example, semi-naïve evaluation had to be adapted for Datafun [5], even though it has been the standard bottom-up evaluation model for a long time [27].

Our approach: Frontend compilation. We would like to achieve the best of both prior approaches: Build on top of existing Datalog solvers as in the backend-first approach, but be free to design functional and domain-specific frontends as in the frontend-first approach. The solution to this problem is compilation: By compiling the frontend language to Datalog, we can use existing solvers to run programs. This way, Datalog really becomes the intermediate representation (IR) of a compiler framework, where different Datalog frontends all generate the same Datalog IR. This architecture is well-known from existing compiler frameworks such as LLVM; we propose to adopt it for Datalog.

Although frontend compilation may seem like the obvious solution, it is difficult to implement. The problem is that Datalog imposes severe restrictions on programs, so that bottom-up evaluation is well-defined and terminates. When generating Datalog code, we must adhere to these restrictions. In the remainder of this paper, we show how a first-order functional language (Section 3) with algebraic data types (Section 4), and sets (Section 6) can be compiled to Datalog. In doing so, we will solve key challenges regarding user-defined functions and user-defined data types that can be transferred to other frontends.

3 Compiling First-Order Functions to Datalog

We want to provide a functional-programming frontend for Datalog. In this section, we tackle the first step in this direction: Compiling user-defined first-order functions to Datalog. While we already outlined why this is challenging in the introduction, here we revisit the problem with a more involved example before presenting our solution.

3.1 Compilation by example

In this paper and in our implementation, we use a simple functional frontend language that features first-order function definitions, let bindings, conditionals, and arithmetic operations. We also support algebraic data types, set operations, and first-class functions, which we will explain later. Consider the following recursive factorial function in functional IncA:

```
def fact(n: Int): Int = if (n == 0) 1 else n * fact(n - 1)
```

We aim to write functions like this and compile them to Datalog, so that we can use them as part of larger Datalog programs. A simple strategy gets us close to the desired result:

Principle 1: Functions as relations. It is well-known that functions `f : (T1, ..., Tn) -> T` can be encoded as relations `f : (T1, ..., Tn, T)`. We use this encoding of functions.

Principle 2: Control-flow paths as rules. For each path from function entry to function exit, we generate a rule that describes how inputs translates to outputs. Since control-flow paths are mutually exclusive in deterministic languages, so are the rules we generate.

When we apply this strategy to our factorial function, we obtain a relation `fact: (Int, Int)`. Since the `fact` function has two exits, we derive two rules that collect all conditions and computations along the path from entry to exit. In doing so, we introduce auxiliary variables for intermediate results as needed.

```
fact(n, out) :- n = 0, out = 1.
fact(n, out) :- n != 0, fact(n-1, out'), out = n * out'.
```

Unfortunately, like in the introduction, the Datalog rules violate *range-restrictedness*. A rule is range-restricted if every variable that occurs in the head of the rule is bound in the body of the rule. Range-restrictedness is an important property for Datalog programs and a prerequisite for bottom-up evaluation. Datalog engines like Soufflé [20] apply bottom-up evaluation to exhaustively enumerate all derivable tuples. Usually, this is an efficient evaluation strategy, but it diverges for rules that are not range-restricted. In our example, the second rule is not range-restricted because `n` is not bound in the body, hence `n` could be any integer term. It follows that the `fact` relation contains infinitely many tuples. Therefore, Soufflé will reject the Datalog code we generated for the `fact` function.

It is hardly surprising that functions over (virtually) infinite domains describe (virtually) infinite relations. So is this approach doomed? To move forward, we make an important observation: Even though a function may be defined over an infinite domain, *any terminating application of that function will only see finitely many inputs.* If we can restrict a function's relation to these inputs, the entire relation turns finite and each rule becomes range-restricted.

To determine the relevant inputs of a function, we must consider how the function is used and what inputs it is applied to. For our factorial example, consider a main call `fact(5)`, which stipulates that `n = 5` is a relevant input of the `fact` relation. But since `fact` is recursive, we must also track which relevant inputs are induced by `n = 5`. If we collect all relevant inputs in `fact_input = {5,4,3,2,1,0}`, we can use this relation to guard the bodies of `fact`:

```
run_fact(out) :- fact(5,out).
fact(n, out) :- fact_input(n), n = 0, out = 1.
fact(n, out) :- fact_input(n), n != 0, fact(n-1, out'), out = n * out'.
```

Note how all rules are range-restricted now. Input variables are range-restricted by the query of the input relation; output variables are range-restricted because they are functionally dependent on the input variables. Thus, `fact` is finite when `fact_input` is finite.

Principle 3: Input relations as guards. For each function, collect all relevant inputs in an input relation and use the input relation as a guard for the function's relation.

Relevant inputs stem from external calls of the function or from recursive calls. Therefore, it is not easy to collect all relevant inputs in a relation. Fortunately, we can apply an existing algorithm that is well-known in the Datalog community: the magic-set transformation [8]. The magic-set transformation was developed to optimize the bottom-up evaluation of terminating Datalog programs. The key idea of the magic-set transformation is to only derive those tuples bottom-up that would also be derived by top-down evaluation, where the relevant inputs are known. To this end, the magic-set transformation generates Datalog rules for auxiliary relations that prescribe which inputs are relevant. Note that we say "inputs" here because the relations we care about correspond to functions; in general, the magic-set transformation collects terms that are known at the call-site during run time. Since function inputs are always known at the call-site during run time, the magic-set transformation will at least collect all relevant function inputs. Technically, we apply a more efficient variation of the magic-set transformation called the *demand transformation* [25] and we use that name in the remainder of the paper.

Principle 4: Demand transformation yields input relations. The demand transformation identifies all relevant inputs for each function in the program. Since all function call-sites must be known, our compilation strategy is not modular but requires the whole program.

For our example, the demand transformation will generate the following input relation:

```
fact_input(5).
fact_input(n-1) :- fact_input(n), n != 0.
```

We obtain one rule for each call of `fact`. The first rule collects the input of the main invocation `fact(5)`. The second rule collects the input of the recursive invocation and contains all constraints leading up to the call. Together, these two rules describe the required relation `fact_input = {5,4,3,2,1,0}`. Since `fact_input` is finite, `fact` is finite and contains the following tuples: `fact = {(5,120), (4,24), (3,6), (2,2), (1,1), (0,1)}`.

So far, all function inputs were statically known. But we can easily extend our compilation strategy to support user-provided inputs. To this end, functional IncA allows the declaration of main functions:

```
@main def run_fact(n: Int): Int = fact(n)
```

(Functional programs)	p	$::=$	$\overline{F}$
(functions)	F	$::=$	$[@\texttt{main}]\ \texttt{def}\ f(\overline{x : T}) : T = e$
(expressions)	e	$::=$	$v \mid x \mid \mathsf{let}\, x = e\, \mathsf{in}\, e \mid \mathsf{if}\ (e)\, e\, \mathsf{else}\, e \mid f(\overline{e}) \mid \varphi(\overline{e})$
(values)	v	$::=$	base
(types)	T	$::=$	Base

Figure 3 Functional IncA with first-order functions, base values, and base functions φ.

(Datalog programs)	D	$::=$	$\overline{r}$
(rules)	r	$::=$	$\texttt{R}(\overline{t})\ \texttt{:-}\ \overline{a}.$
(atoms)	a	$::=$	$t = t \mid \texttt{R}(\overline{t})$
(terms)	t	$::=$	$v \mid x \mid \varphi(\overline{t})$
(values)	v	$::=$	base

Figure 4 An intermediate representation for Datalog with base values and base functions.

The demand transformation will correctly propagate the input of `run_fact` to `fact`:

```
fact_input(n) :- run_fact_input(n).
fact_input(n-1) :- fact_input(n), n != 0.
```

But what is the input of `run_fact`? The input of `run_fact` is dynamic and must be provided by the user of the program. In Datalog, such data lives in the so-called extensional database, which is filled by the user prior to Datalog execution. We modify the demand transformation to generate a query of the extensional database for main functions.

Principle 5: Main input in extensional database. For each main function, we add a rule to the input relation that retrieves dynamic inputs from the extensional database.

For our example, we obtain the following input relation for `run_fact`:

```
run_fact_input(n) :- ext_run_fact_input(n).
```

The user can provide any number of inputs to `run_fact` as part of the extensional database. The Datalog engine will propagate those inputs to `run_fact_input` and fill all relations.

Note that our encoding retains crucial Datalog behavior, such as memoization and reuse. For example, consider we want to run `fact` on multiple inputs `5`, `7`, and `9`, all of which we put into the extensional database. How many tuples will relation `fact` contain? Since queries of `fact` will retrieve existing tuples when possible, the three `fact` computations will share all intermediate results and `fact` will only contain 10 tuples (the largest input plus one). A similar effect can be observed for functions like Fibonacci, where recursive calls can share results. All of this is transparent to the user.

3.2 Translating functional programs to Datalog, technically

We now implement Principles 1 and 2 from the previous subsection, that is, we translate functional programs to Datalog. In the subsequent subsection, we will explain and apply the demand transformation to implement the remaining principles.

Figure 3 defines the syntax of functional IncA. The language consists of first-order functions, let bindings, conditionals, and function calls. We distinguish calls to user-defined functions f from calls to base functions φ. Our compilation target is an intermediate representation (IR) of Datalog extended with base values and base functions as shown in Figure 4. This Datalog IR is compatible with many existing Datalog solvers, which support different kind of base functions. Note that we excluded negation from the Datalog IR because our translation does not require it.

We first translate expressions to Datalog. While an expression is structured and eventually computes a value, Datalog only provides flat terms. Thus, a nested expression $f(g(x))$ must be

$$
\begin{aligned}
[\![.]\!] :&\;\; e \to \mathcal{P}(t \times \mathcal{P}(a))\\
[\![v]\!] &= \{(v,\ \emptyset)\}\\
[\![x]\!] &= \{(x,\ \emptyset)\}\\
[\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2]\!] &= \{(t_2, \{x = t_1\} \cup a_1 \cup a_2) \mid (t_1, a_1) \in [\![e_1]\!], (t_2, a_2) \in [\![e_2]\!]\}\\
[\![\mathsf{if}\ (e_1)\, e_2 \,\mathsf{else}\, e_3]\!] &= \{(t_2, \{t_1 = \mathsf{true}\} \cup a_1 \cup a_2) \mid (t_1, a_1) \in [\![e_1]\!], (t_2, a_2) \in [\![e_2]\!]\}\\
&\cup\ \{(t_3, \{t_1 = \mathsf{false}\} \cup a_1 \cup a_3) \mid (t_1, a_1) \in [\![e_1]\!], (t_3, a_3) \in [\![e_3]\!]\}\\
[\![f(e_1, \ldots, e_n)]\!] &= \{(y, \{f(t_1, \ldots, t_n, y)\} \cup a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in [\![e_1]\!], \ldots, (t_n, a_n) \in [\![e_n]\!]\}\\
&\quad \text{where } y \text{ is fresh}\\
[\![\varphi(e_1, \ldots, e_n)]\!] &= \{(\varphi(t_1, \ldots, t_n), a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in [\![e_1]\!], \ldots, (t_n, a_n) \in [\![e_n]\!]\}
\end{aligned}
$$

Figure 5 Compiling expressions yields a set of alternative terms, each guarded by constraints.

$$
[\![\mathtt{def}\ f(\overline{x : T}) : T' = e]\!]_{\mathit{fun}} = \{\mathtt{f}(\overline{x}, y)\ \mathtt{:\text{-}}\ a,\ y = t.\ \mid (t, a) \in [\![e]\!]\} \quad \text{where } y \text{ is fresh}
$$

$$
[\![\overline{F}]\!]_{\mathit{prog}} = \textstyle\bigcup_{f \in \overline{F}} [\![f]\!]_{\mathit{fun}}
$$

Figure 6 Compiling functions to Datalog rules.

compiled to a flat term that is guarded by constraints $(y_2, \{g(x, y_1), f(y_1, y_2)\})$. Since conditional expressions $(\mathsf{if}\ (b)\ f(x)\ \mathsf{else}\ g(x))$ yield alternative values depending on b, compilation in general yields a set of alternative terms $\{(y_1, \{b = \mathsf{true}, f(x, y_1)\}), (y_2, \{b = \mathsf{false}, g(x, y_2)\})\}$. This correponds to Principle 2 from the previous subsection.

Figure 5 defines the translation of expressions as a compositional function $[\![.]\!]$. Values v and variables x directly translate to Datalog values and variables. Let bindings yield the body's result under a constraint that binds the let-bound variable. Conditionals compile to two alternative sets of terms: If the condition is true, the resulting terms are taken from the *then*-branch, otherwise they are taken from the *else*-branch. Calls to user-defined functions f translate to queries of a relation of the same name f, which has the function's result as an additional column in accordance with Principle 1. In contrast, calls to base functions φ translate to a call of the same function, but passing Datalog terms as arguments.

We use the translation of expressions $[\![.]\!]$ to compile function definitions $[\![.]\!]_{\mathit{fun}}$ and programs $[\![.]\!]_{\mathit{prog}}$ as shown in Figure 6. For a function definition, we compile its body and generate a separate Datalog rule for each alternative term that the body can yield. The constraints a of the term become constraints in the generated rule. To compile a whole program, we simply compile each function and collect the resulting rules.

For a concrete example, consider the translation of the Fibonacci function to Datalog:

```
⟦def fib(n) = if (n<2) n else fib(n-1) + fib(n-2)⟧ = { fib(n, y3) :- n<2 = true, y3 = n.,
  fib(n, y4) :- n<2 = false, fib(n-1,y1), fib(n-2,y2), y4 = y1+y2. }
```

The Fibonacci function compiles to two Datalog rules, one for the base case and one for the recursive case. But is this translation correct?

Translation correctness. We claim that our translation preserves the semantics of the original functional program. More precisely, we claim that if a function call $f(\overline{v})$ evaluates to w, then the generated Datalog program will also provide w as the only result of the call *under top-down evaluation.* Here we must require top-down evaluation for Datalog, since the generated rules are not necessarily range-restricted yet, which we will fix in the next subsection. Top-down evaluation is possible nonetheless, because it only explores required results and uses known values in doing so. Since the values of function arguments are always known during evaluation, top-down evaluation of the generated Datalog closely corresponds to function evaluation. However, we did not formalize top-down evaluation and therefore formulate translation correctness as a conjecture:

▶ **Conjecture 1** (Translation correctness). *Given a functional program p with a main function f such that $f(\overline{v})$ evaluates to w. Then the top-down evaluation of the Datalog atom $f(\overline{v}, x)$ under $[\![p]\!]_{prog}$ yields a single substitution $\{x \mapsto w\}$.*

A key component for proving this conjecture is to ensure the Datalog constraints behave deterministically, just like the original expression did:

▶ **Lemma 2** (Deterministic atoms). *Given an expression e such that $[\![e]\!] = \{(t_1, \overline{a_1}), \ldots, (t_n, \overline{a_n})\}$ both of the following hold:*

i. $[\![e]\!]$ *yields at least one result:* $\overline{a_1} \vee \ldots \vee \overline{a_n}$
ii. $[\![e]\!]$ *yields at most one result:* $(\overline{a_i} \wedge \overline{a_j}) \rightarrow t_i = t_j$

Proof. By structural induction over e. The only interesting case are *if*-expressions, where $(t_1 = \mathsf{true})$ and $(t_1 = \mathsf{false})$ are mutually exclusive. ◀

▶ **Lemma 3** (Deterministic rules). *Given f such that $[\![f]\!]_{fun} = \{f(\overline{x}, y_1) \text{:-}\ \overline{a_1}., \ldots, f(\overline{x}, y_n) \text{:-}\ \overline{a_n}.\}$ both of the following hold:*

i. $[\![f]\!]_{fun}$ *yields at least one result:* $\overline{a_1} \vee \ldots \vee \overline{a_n}$
ii. $[\![f]\!]_{fun}$ *yields at most one result:* $(\overline{a_i} \wedge \overline{a_j}) \rightarrow y_i = y_j$

Proof. Follows from Lemma 2. ◀

Note that Conjecture 1 does not make any assertions about non-terminating function calls. Indeed, some diverging functions compile to terminating Datalog programs. For example, `def f(x) = f(x)` compiles to `f(x,y) :- f(x,y)`. While function call `f(1)` diverges, query `f(1, y)` terminates and yields the empty substitution. However, Conjecture 1 ensures terminating function calls translate to terminating Datalog programs under top-down evaluation.

3.3 Demand-driven bottom-up evaluation

We compile functional programs to Datalog rules that execute well in top-down fashion, but may diverge under bottom-up evaluation. In bottom-up evaluation, Datalog solvers exhaustively enumerate all derivable tuples, starting from known facts. For example, the bottom-up evaluation of the factorial function will start with `fact(0,1)`, from which it can derive `fact(1,1)`, `fact(2,2)`, `fact(3,6)`, `fact(4,24)`, and so on. This enumeration will not terminate, because bottom-up evaluation is unaware of the context in which relation `fact` is being used. Accordingly, we cannot apply any of the efficient Datalog solvers that use bottom-up evaluation, such as Soufflé.

The demand transformation by Tekle and Liu rewrites Datalog rules such that bottom-up evaluation becomes demand-driven and only computes tuples that are transitively demanded by the main query [25]. Indeed, bottom-up evaluation of the rewritten Datalog rules computes *exactly the same* tuples as top-down evaluation. Since we already asserted that top-down evaluation computes the correct result for terminating functional programs, the demand transformation allows us to apply bottom-up evaluation, also yielding the correct result.

We adopt the demand transformation, which transforms a set of Datalog rules in three steps: compute demand patterns, introduce demand predicates, derive demand rules. In this section, we replace the first step of the demand transformation to take functional IncA into account, adopt the second step unchanged, and extend the third step to account for the inputs of main functions. Later sections will make further changes.

Step 1 We compute demand patterns $\langle g, s\rangle$, where g is the name of a relation and $s \in (b \mid f)^*$ is a pattern string that indicates how the relation is queried, namely if an argument occurs bound or free. For functions, demand patterns can be easily computed by finding all function calls reachable from the main functions. Formally, given a functional program p, the demand patterns $dp(p)$ of p is the smallest set such that:

- For each main function (`@main def` $g(x_1, \ldots, x_n) = \ldots$) in p, we have $\langle g, b^n f\rangle \in dp(p)$. That is, main functions have demand with n bound parameters and one free return value.
- If demand pattern $\langle g, s\rangle \in dp(p)$ and g is defined as (`def` $g(\ldots) = e$) in p, we have $\langle h, b^n f\rangle \in dp(p)$ for each call $h(e_1, \ldots, e_n)$ in e.

The second and third step of the demand transformation operate on and rewrite the generated Datalog rules $D = [\![p]\!]_{prog}$. In particular, we will make no assumptions about the format of pattern strings s, so that we can later introduce extensions of Step 1 easily.

Step 2 We introduce demand predicates as guards into existing rules to implement Principle 3 from Subsection 3.1. Formally, we obtain a rewritten Datalog program $guarded(D)$:

- For each $\langle g, s\rangle \in dp(p)$ and each $(g(t_1, \ldots, t_m)$ `:-` $a_1, \ldots, a_n.)$ in D, we obtain a rule

$$g(t_1, \ldots, t_m) \text{ :- } g\_{\scriptstyle input}\_s(t_1, \ldots, t_m|_s), a_1, \ldots, a_n.$$

  in $guarded(D)$, where $\bar{t}|_s$ selects those t_i that are bound according to pattern string s.

Note that the rules of unreachable functions are dropped and not propagated to $guarded(D)$.

Step 3 In the final step, we must derive those rules that define the input relations $g\_{\scriptstyle input}\_s$ to implement Principle 4 and Principle 5 from Subsection 3.1. Formally, we obtain a rewritten Datalog program $demanded(D)$ from $guarded(D)$ and the original program p as follows:

- We retain each rule from $guarded(D)$, such that $guarded(D) \subseteq demanded(D)$.
- For each main function (`@main def` $g(x_1, \ldots, x_n) = \ldots$) in p, we obtain a rule

$$g\_{\scriptstyle input}\_s(x_1, \ldots, x_n) \text{ :- } ext\_g\_{\scriptstyle input}\_s(x_1, \ldots, x_n).$$

  in $demanded(D)$, where $ext\_g\_{\scriptstyle input}\_s$ is an extensional relation to be filled by the user. This implements Principle 5.
- For each rule $(g(\ldots)$ `:-` $a_1, \ldots, a_n.)$ in $guarded(D)$ and each $a_i = h(t_1, \ldots, t_m)$, we obtain

$$h\_{\scriptstyle input}\_s(t_1, \ldots, t_m|_s) \text{ :- } a_1, \ldots, a_{i-1}$$

  to $demanded(D)$, where s is the pattern string of $h(t_1, \ldots, t_m)$, indicating which t_i are bound by the previous constraints $a_1, \ldots, a_{i-1}$ already.

The demand transformation implements Principles 3 - 5 and ensures that the resulting Datalog derives the same tuples in bottom-up evaluation as in top-down fashion.

Example To illustrate, consider again the Fibonacci function with a main call:

```
def fib(n) = if (n<2) n else fib(n-1) + fib(n-2)
@main def run(x: Int): Int = fib(x)
```

This program compiles to the following Datalog rules using the translation from Subsection 3.2:

```
fib(n, y3) :- n<2 = true, y3 = n.
fib(n, y4) :- n<2 = false, fib(n-1,y1), fib(n-2,y2), y4 = y1+y2.
run(x, y5) :- fib(x, y5).
```

We now apply our demand transformation. First, we derive demand patterns of the program, which are $\langle$`run`, $bf\rangle$ and $\langle$`fib`, $bf\rangle$. Note that all three calls of `fib` yield the same demand pattern. Second, we insert demand predicates into the rules according to the demand patterns:

```
fib(n, y3) :- fib_input_bf(n), n<2 = true, y3 = n.
fib(n, y4) :- fib_input_bf(n), n<2 = false, fib(n-1,y1), fib(n-2,y2), y4 = y1+y2.
run(x, y5) :- run_input_bf(x), fib(x, y5).
```

Third, to these rules we add the following rules to define the input relations:

```
run_input_bf(x) :- ext_run_input_bf(x).
fib_input_bf(x) :- run_input_bf(x).
fib_input_bf(n-1) :- fib_input_bf(n), n<2 = false.
fib_input_bf(n-2) :- fib_input_bf(n), n<2 = false, fib(n-1,y1).
```

The first and second rule are due to the main function `run`, which receives its input from the user and propagates it to `fib`. The third and fourth rule are due to the recursive calls of `fib`. Note how we retain all constraints prior to a call. In particular, we retain the first recursive call of `fib` as a constraint for the second recursive call of `fib`, although a smart compiler might eliminate this constraint subsequently. The resulting Datalog program is demand-driven and can be executed by standard bottom-up Datalog solvers.

Correctness The demand transformation yields a Datalog program that derives the exact same tuples as a top-down evaluation [25]. As of Conjecture 1, top-down evaluation yields the correct tuples. Hence, so does bottom-up evaluation of the demand-driven Datalog rules:

▶ **Corollary 4** (Bottom-up translation correctness). *Given a functional program p with a main function f such that $f(\overline{v})$ evaluates to w. Then the bottom-up evaluation of the Datalog program demanded($[\![p]\!]_{prog}$) yields a database in which the query $f(\overline{v}, x)$ has a single match $\{f(\overline{v}, w)\}$.*

4 Compiling Algebraic Data Types to Datalog

The functional IncA we presented in the previous section supports user-defined functions ranging over base types. In this section, we explore how to extend functional IncA to allow user-defined data types. In particular, we want to faithfully compile recursive functions over algebraic data types to Datalog rules that existing bottom-up Datalog solvers can execute.

4.1 Compiling user-defined data types by example

We extend functional IncA to allow recursive definitions of user-defined algebraic data types, constructor calls, and pattern matching. As a simple example, consider the Peano numbers:

```
data Nat = Zero() | Succ(Nat)
def plus(m: Nat, n: Nat): Nat = m match {
  case Zero() => n
  case Succ(pred) => Succ(plus(pred, n)) }
@main def twice(n: Nat): Nat = plus(n, n)
```

We generate three kind of relations for an algebraic data type:

- **Constructor relations** represent the constructor functions of algebraic data types. We translate constructor calls in the program to queries of constructor relations, similar to how we translated regular function calls. In doing so, it is crucial we ensure only finitely many values are constructed during bottom-up evaluation of the resulting Datalog code.
- **Selector relations** map a constructed value to its constituents. We use selector relations to implement pattern matching. Importantly, queries of selector relations may never lead to the construction of new values.
- **Instance relations** enumerate all constructed instances of a data type. They will become useful when we introduce relational programming in Section 6.

To construct user-defined data at run time, we extend the Datalog IR with a built-in constructor `#constr` for each constructor `constr`. For example, `#Succ(#Succ(#Zero()))` encodes two as a Peano number. In practice, there are different ways a Datalog solver can support such built-in constructors. For example, we can define a generic built-in function that creates a new value given the constructor's name and arguments. We have used this approach in our implementation using IncA, but this would work in any Datalog solver that supports user-defined built-in functions, including Soufflé, Flix, and Formulog. Alternatively, if a Datalog solver natively supports algebraic data types, we can use their constructors directly or encode them using a number representation. For example, Soufflé supports algebraic data types (but not recursive functions over them) and we can generate a Soufflé data type and use its constructors. This is to say that adding built-in constructors to the Datalog IR does not limit the applicability of our approach in practice. Flix, Formulog, IncA and Soufflé have support for algebraic data. However, they do not support enumerating all instances of a specific algebraic data type like functional IncA. We will see how to enumerate all instances of an algebraic data type by utilizing instance relations in Section 6.

For the Peano numbers, we derive the following Datalog rules initially:

```
// constructor relations
Zero(n) :- n = #Zero().
Succ(p, n) :- n = #Succ(p).
```

```
// selector relations
un_Zero(n) :- Zero(n).
un_Succ(n, p) :- Succ(p, n).
```

```
// instance relation
Nat(n) :- Zero(n).
Nat(n) :- Succ(_, n).
```

Note that the rule of the `Succ` constructor relation is not range-restricted and consequently cannot be computed bottom-up. However, the rules of the selector and instance relations merely query the constructor relations. Hence, if we can ensure the constructor relations remain finite, all three kind of relations will be finite.

Like in the previous section, we seek to apply the demand transformation in order to track the demand of constructor relations. However, we need to adapt the demand transformation to account for our encoding of algebraic data types. Specifically, the constructor queries within the selector and instance relations must be ignored, since they do not actually indicate additional demand. Moreover, selector and instance relations do not require any rewriting themselves, because they merely query constructor relations to enumerate constructor tuples.

Figure 7 shows the compilation result after demand transformation for the `plus` function on Peano numbers from above. Relation `Zero` has no demand relation because its demand pattern $\langle$`Zero`$, f\rangle$ does not specify bound inputs. Relation `Succ` has a demand relation `Succ_input_bf` that tracks the invocation of `Succ` in the recursive case of `plus`. Importantly, there is no demand on `Succ` from the selector or instance relations, as our adaption of the demand transformation will ensure. Relation `plus` shows how we compile pattern matching: Each case becomes an alternative rule that queries the selector. This is sufficient since we assume pattern matches are complete and overlap-free, so that their order does not matter.

Since `twice` is a main function, its demand relation queries an extensional input relation as described in the previous section. This way, users can for example request `twice(Succ(Zero()))`.

```
Zero(n) :- n = #Zero().   // no demand relation since there are no bound inputs
Succ(p, n) :- Succ_input_bf(p), n = #Succ(p).
Succ_input_bf(y4) :- plus_input_bbf(m, n), un_Succ(m, pred), plus(pred, n, y4).

// selector and instance relations un_Zero, un_Succ, and Nat as above
plus(m, n, out) :- plus_input_bbf(m, n), un_Zero(m), out = n.
plus(m, n, out) :- plus_input_bbf(m, n), un_Succ(m, pred),
                   plus(pred, n, y4), Succ(y4, y5), out=y5.
plus_input_bbf(n, n) :- twice_input_bf(n).
plus_input_bbf(pred, n) :- plus_input_bbf(m, n), un_Succ(m, pred).

twice(n, out) :- twice_input_bf(n), plus(n, n, out).
twice_input_bf(n) :- ext_twice_input_bf(n).

Zero(n) :- ext_Zero(n).
Succ(p, n) :- ext_Succ(p, n).
```

Figure 7 Compilation result for the `plus` and `twice` functions on Peano numbers.

(Functional programs)	*prog*	$::= \overline{F}, \overline{d}$
(data definitions)	d	$::= \texttt{data}\ N = \overline{c(T, \ldots, T)}$
(expressions)	e	$::= \ldots \mid c(\overline{e}) \mid e\ \texttt{match}\ \{\overline{\texttt{case}\ c(x, \ldots, x) => e}\}$
(types)	T	$::= \ldots \mid N$

Figure 8 Extending the frontend syntax with algebraic data types.

But how can our Datalog program deconstruct the user-provided data? Recall that selector relations simply query constructor relations. Thus, we must include the user-provided algebraic data in our constructor relations. To this end, we require users to insert algebraic data in extensional constructor relations. We then generate one additional rule for each constructor that queries the corresponding extensional constructor relation, as shown at the end of Figure 7. We need to provide the contents of extensional constructor relations in the form of tuples consistent with the format supported by the targeted Datalog dialect. In the case of Soufflé, we insert tuples containing algebraic data and literal values of the Soufflé language in the extensional constructor relations.

4.2 Extending functional IncA with algebraic data types

Based on the observations from the previous subsection, we add algebraic data types to functional IncA. We then extend the translation from functional code to Datalog code and the demand transformation accordingly.

Figure 8 extends the abstract syntax of functional IncA with algebraic data types. For pattern matching we assume that patterns are complete and overlap-free. We do not change the syntax of Datalog since we model constructors as built-in functions φ.

We extend the translation of Subsection 3.2 from functional code to Datalog code to handle algebraic data types as shown in Figure 9. We add a new translation function $[\![.]\!]_{data}$ for data types and use that when compiling programs in $[\![.]\!]_{prog}$. The translation of functions $[\![.]\!]_{fun}$ remains the same, but it uses an extended translation for expressions $[\![.]\!]$ that handles the new expressions: constructor calls and pattern matching. The translation of constructor calls is identical to the translation of regular function calls, except the generated code queries a constructor relation. Pattern matching yields alternative rules for each case, and each case queries the selector relation `un_`c to test if the term matches the pattern. The translation of data types $[\![.]\!]_{data}$ generates rules as described in the previous subsection: rules that invoke the built-in constructor functions, rules that query the extensional constructor relations, rules for the selector relations, and rules for the instance relations.

$$[\![\overline{F}, \overline{d}]\!]_{prog} = \bigcup_{f \in \overline{F}} [\![f]\!]_{fun} \quad \cup \quad \bigcup_{d \in \overline{d}} [\![d]\!]_{data}$$

$$\begin{aligned}[\![c(e_1, \ldots, e_n)]\!] &= \{(y, \{c(t_1, \ldots, t_n, y)\} \cup a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in [\![e_1]\!], \ldots, (t_n, a_n) \in [\![e_n]\!]\} \\ &\quad \text{where } y \text{ is fresh} \\ [\![e \; \texttt{match} \; \{\overline{cs}\}]\!] &= \bigcup_{(\texttt{case}\; c(\overline{x}) \; \texttt{=>} e') \in \overline{cs}} \{(t', \{\mathsf{un\_}c(t, \overline{x})\} \cup a \cup a') \mid (t, a) \in [\![e]\!], (t', a') \in [\![e']\!]\}\end{aligned}$$

$$\begin{aligned}[\![\texttt{data}\; N = \overline{C}]\!]_{data} &= \{c(x_1, \ldots, x_n, y) \;\texttt{:-}\; y = \#c(x_1, \ldots, x_n). \mid c(T_1, \ldots, T_n) \in \overline{C}\} \\ &\cup \{c(x_1, \ldots, x_n, y) \;\texttt{:-}\; y = \mathit{ext\_c}(x_1, \ldots, x_n, y). \mid c(T_1, \ldots, T_n) \in \overline{C}\} \\ &\cup \{\mathsf{un\_}c(y, x_1, \ldots, x_n) \;\texttt{:-}\; c(x_1, \ldots, x_n, y). \mid c(T_1, \ldots, T_n) \in \overline{C}\} \\ &\cup \{N(y) \;\texttt{:-}\; c(x_1, \ldots, x_n, y). \mid c(T_1, \ldots, T_n) \in \overline{C}\}\end{aligned}$$

Figure 9 Translating algebraic data types to Datalog.

Next, we extend the demand transformation from Subsection 3.3 to consider constructors:

- In Step 1, when considering reachable subexpressions $h(e_1, \ldots, e_n)$, we also generate a demand pattern $\langle h, b \ldots bf \rangle$ when g is a constructor.
- In Step 2, note that selector and instance relations are never demanded, since we ignored them in Step 1. Hence, we propagate their rules unchanged to $\mathit{guarded}(D)$.
- In the last case of Step 3, we ignore atoms $a_i = h(t_1, \ldots, t_m)$ that occur in the rules of selector or instance relations. These atoms always query a constructor relation and we do not want to treat these queries as demand.

With these modifications, the demand transformation will correctly track the demand of constructors while ignoring selectors and instance relations. Together, the extended translation and the demand transformation constitute a compiler for functional IncA with algebraic data types. Since all rules of the generated Datalog code are range-restricted, we can run the code with off-the-shelf bottom-up Datalog solvers.

5 Case study: Type Checking, Type Erasure, and Interpretation

Functional IncA supports user-defined functions and data types. In this section, we demonstrate that these features allow us to express interesting computations in Datalog. In particular, we implement a type checker, type erasure, and an interpreter for a lambda calculus with numbers as illustrated in Figure 10. These functions compile to complex Datalog code that could not practically be written by hand.

Figure 10 shows an excerpt of the relevant data types and functions, all of which are completely standard. In particular, we describe the expressions of the simply typed lambda calculus `Exp` and the untyped lambda calculus `UExp` as algebraic data types. We define a type checker `typeOf` as a function in functional IncA, but only show the `App` case here. Our implementation supports parametric polymorphism by applying monomorphization before translating to Datalog. Since the `App` case has five alternative control-flow paths, this case alone compiles into five Datalog rules for `typeOf`. For example, consider the rule generated for the path that yields `Just(ty2)`:

```
typeOf(ctx, exp, out0) :-
  typeOf_input_bbf(ctx,exp), un_App(exp,fun,arg), typeOf(ctx,fun,o1),
  un_JustType(o1,funty), un_TFun(funty,ty1,ty2), typeOf(ctx,arg,o2),
  un_JustType(o2,argty), eqType(argty,ty1,o3), o3 == true, JustType(ty2,out0).
```

```
data Exp = Num(Int) | Lam(String, Type, Exp) | App(Exp, Exp) | Var(String)
data Type = TInt() | TFun(Type, Type)
data UExp = UNum(Int) | ULam(String, Exp) | UApp(Exp, Exp) | UVar(String)
def typeOf(ctx: Ctx, exp: Exp): Maybe[Type] = exp match {
  case App(fun, arg) => typeOf(ctx, fun) match {
    case Just(TFun(ty1, ty2)) => typeOf(ctx, arg) match {
        case Just(argty) => if (eqType(argty, ty1)) Just(ty2) else Nothing()
  ... }
def erase(exp: Exp): UExp = exp match {
  case Num(v) => UNum(v)
  case Lam(n, ty, b) => ULam(n, erase(b))
  case App(fun, arg) => UApp(erase(fun), erase(arg))
  case Var(n) => UVar(n) }
def interp(env: Env, exp: UExp): Maybe[Val] = exp match {
  case UApp(fun, arg) => interp(env, fun) match {
    case Just(VClosure(param, prog, fenv)) => interp(env, arg) match {
      case Just(argv) => interp(BindEnv(param, argv, fenv), body)
  ... }
@main def run(exp: Exp): Maybe[Val] = typeOf(EmptyCtx(), exp) match {
  case Just(ty) => interp(EmptyEnv(), erase(exp))
  case Nothing() => Nothing() }
```

Figure 10 A type checker, type erasure, and interpreter for a lambda calculus with numbers.

This Datalog rule consists of 10 atoms, where the selector predicates ensure that the correct control-flow path has been chosen. Overall, the `typeOf` function consists of 24 lines of code, but compiles to 114 lines of complex Datalog code with mutually dependent relations `typeOf` and `typeOf_input`. These numbers represent the Datalog program after applying optimizations. In contrast to program optimizations of functional and imperative programs, our Datalog optimizations reduce the number of rules and atoms instead of increasing them.

Next, we define type erasure as a transformation from `Exp` to `UExp`. Although function `erase` is completely standard, this is the first program transformation implemented in Datalog to the best of our knowledge. While `erase` is guaranteed to terminate, we can also define functions whose termination is undecidable. Specifically, we implement a standard interpreter `interp` for the untyped lambda calculus, which is a Turing-complete language. Indeed, the Datalog program is only guaranteed to terminate when the original interpreter terminates.

Overall, the type checker, type erasure, and interpreter comprise 8 algebraic data types and 7 functions. We compile this code to 65 relations defined by 154 rules that contain 484 atoms in total. These numbers are measured after optimization, where we eliminate aliases and propagate constants.

Although implementing an interpreter in Datalog may seem to be of little use, this and similar challenges occur during program analysis regularly. For example, Pacak et al. recently have shown how to compile typing rules to Datalog to derive incremental type checkers systematically [17]. They also mention that it is necessary to translate the dynamic semantics of a language to Datalog in order to support the incremental type checking of a dependently typed programming language. Similarly, data-flow analyses often need to abstractly interpret programs, for example, to determine the bounds of numeric variables or the value of a Boolean condition. Functional IncA can also support such data-flow analyses, but we must be able to express control-flow graphs and other relations.

6 Mixing Functions and Relations

The previous sections showed how we can use functional programming as a frontend for Datalog. However, in doing so, we have also lost a key feature of Datalog: relations. Indeed,

```
data Exp = ...
data Stm = Assign(String, Exp) | Sequence(Stm, Stm) | If(Exp, Stm, Stm) | While(Exp, Stm)
def init(stm: Stm): Stm = ... // a regular function that finds the statement's entry
def final(stm: Stm): Set[Stm] = stm match { // finds all of the statement's exits
  case Assign(x, a) => {stm}
  case Sequence(s1, s2) => final(s2)
  case If(b, s1, s2) => final(s1) ++ final(s2)
  case While(b, s) => {stm} }
// flow as seen in Figure 1 (Introduction)
```

Figure 11 Computing the control-flow graph as a set of tuples in out extended Datalog frontend.

functional IncA makes it difficult to encode non-functional relations, such as the edges of a graph. In the present section, we show how we can elegantly extend functional IncA to re-introduce relations.

6.1 Computing a control-flow graph functionally

Consider we want to compute the control-flow graph (CFG) of a program as part of a Datalog-based program analysis. We want to represent the CFG such that it corresponds to a Datalog relation, so that we can easily compute its transitive closure later. While the functions of functional IncA compile to Datalog relations, our functions cannot be used to encode arbitrary relations. In particular, a function (`def flow(from: Stm): Stm = e`) cannot handle conditional statements that fork the control flow and connect to multiple successor statements. To support such relations, we must extend our frontend language.

We want to extend functional IncA in a way that integrates functions and relations elegantly. This is a language-design challenge and therefore naturally somewhat subjective. But it is the reason why we rejected the first idea that came to mind: to introduce relations next to functions. For example, a top-level relation (`rel flow(from: Stm, to: Stm) :- constraints`) could capture the CFG of a program. The problem is that we are now back at constraint programming, which is exactly what we wanted to avoid with functional IncA.

We propose a different extension of functional IncA that not only avoids this problem but that is simpler too: We introduce sets and tuples. Immutable sets and tuples are staple ingredients of functional programming and programmers already know how to use them. Moreover, any relation can be encoded as a set containing tuples of related values. Thus, the only question is if and how we can map functional programs over sets and tuples to Datalog. But first, let us illustrate how the extended functional IncA can be used.

In their classic textbook, Nielson et al. [16] compute the control flow of a While-statement through three functions. We can represent these functions in the extended functional IncA almost verbatim as shown in Figure 11. Here, `init` is a regular function whereas `final` and `flow` compute sets. A set literal `{e1,...,en}` constructs a set and set union `++` composes two sets. For example, `final` uses these features to compute the final statement of each conditional branch. Sets can be processed through set comprehensions as shown in the definition of `flow` which can be seen in Figure 1. In particular, `(x1,...,xn) in set` retrieves the elements of `set`, binds those `x` that are free, and tests for membership of those `x` that are bound.

Our encoding of relations makes it easy to implement computations that exercise Datalog's declarative fixpoint semantics, such as transitive closure, cycle detection, and recursive aggregation. We have already demonstrated such computations in the introduction of this paper and refrain from repeating them here. Instead, we show how to translate functional programs with sets and tuples to Datalog.

(functions)	F	$::= \ldots \mid [\texttt{@main}]\ \texttt{def}\ f(\overline{x : T}) : \mathsf{Set}[T] = s$
(set expressions)	s	$::= \{\overline{e}\} \mid s \mathbin{++} s \mid \{e \vert \overline{pred}\} \mid \mathsf{let}\, x = e\, \mathsf{in}\, s \mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s \mid f(\overline{e})$
(predicates)	$pred$	$::= e \mid e\ \mathsf{in}\ s \mid e\ \mathsf{in}\ N$
(expressions)	e	$::= \ldots \mid \mathsf{fold}(f, f, f)$

Figure 12 Extended abstract syntax with set and set operations.

$$\llbracket \{\overline{e}\} \rrbracket = \textstyle\bigcup_{e \in \overline{e}} \llbracket e \rrbracket$$

$$\llbracket s_1 \mathbin{++} s_2 \rrbracket = \llbracket s_1 \rrbracket \cup \llbracket s_2 \rrbracket$$

$$\llbracket \{e \mid p_1, \ldots, p_n\} \rrbracket = \{(t, \{t_1 = \mathsf{true}, \ldots, t_n = \mathsf{true}\} \cup a \cup a_1 \cup \ldots \cup a_n) \mid (t, a) \in \llbracket e \rrbracket, (t_1, a_1) \in \llbracket p_1 \rrbracket_{pred}, \ldots, (t_n, a_n) \in \llbracket p_n \rrbracket_{pred}\}$$

$$\llbracket \mathsf{fold}(f_{init}, f_{op}, f_{set}) \rrbracket = \{(\mathsf{aggregate}(f_{set}, \mathsf{toPrimitiveFun}(f_{init}), \mathsf{toPrimitiveFun}(f_{op})), \emptyset)\}$$

$$\llbracket e \rrbracket_{pred} = \llbracket e \rrbracket$$

$$\llbracket e\ \mathsf{in}\ s \rrbracket_{pred} = \{(\mathsf{true}, \{t_1 = t_2\} \cup a_1 \cup a_2 \mid (t_1, a_1) \in \llbracket e \rrbracket, (t_2, a_2) \in \llbracket s \rrbracket\}$$

$$\llbracket e\ \mathsf{in}\ N \rrbracket_{pred} = \{(\mathsf{true}, \{N(t)\} \cup a \mid (t, a) \in \llbracket e \rrbracket\}$$

Figure 13 Compiling sets and set operations to Datalog.

6.2 Translating tuples and first-order sets to Datalog

The translation of sets and tuples to Datalog is mostly straightforward except for one thing: neither sets nor tuples are first-class in Datalog. For tuples this is hardly an issue since we can simply flatten tuples when translating them to Datalog. For example, a function `foo(t: (T1,...,Tn)): (U1,...,Um)` becomes a flat relation `foo(T1,...,Tn,U1,...,Um)`, and a function call `foo(e)` becomes `foo(t1,...,tn,u1,...,um)`, where `e` translates to `n` terms `(t1,...,tn)` and the function call yields `m` result terms `(u1,...,um)`. Although our implementation supports tuples, we omit tuples from our translation semantics and focus on sets instead.

We want to translate sets to Datalog relations, but relations are first-order in Datalog and can only appear as top-level definitions. Thus, if we want to support first-class sets in functional IncA, we need to lift those sets first. For example, to translate a call `transitive({(1,2),(2,3),(3,4)})` to Datalog, we have to translate `{(1,2),(2,3),(3,4)}` to a top-level relation that can be queried from within `transitive`. To achieve this, we propose a clean solution in two steps:

1. We extend functional IncA first-order sets, which may only appear as function results. First-order sets translate to first-order relations as shown in the present subsection.
2. The subsequent subsection shows that a standard defunctionalization transformation simultaneously adds support for first-class functions and first-class sets to functional IncA.

Figure 12 defines the extended functional IncA, where we introduce first-order sets syntactically through a new non-terminal `s`. This syntactic differentiation does not replace type checking of the functional code, but serves to explain which expressions may yield sets without presenting functional IncA's type system, which is completely standard and uninteresting. First-order sets may only occur as the body of a function that yields a set and within other set expressions. A set comprehension can use predicates *pred* to check a boolean condition e, to query another set (e in s), or to query all instances of an algebraic data type (e in N). Here we finally see why we introduced instance relations for algebraic data types in Section 4. At last, we can convert a set to an atomic value through $\mathsf{fold}(f_{init}, f_{op}, f_{set})$, where f_{set} must be the name of a top-level definition.

$$\begin{array}{lll} \text{(expressions)} & e & ::= \ldots \mid f \mid (\overline{x : T}) \Rightarrow e \mid (\overline{x : T}) \Rightarrow s \mid e(\overline{e}) \\ \text{(types)} & T & ::= \ldots \mid \overline{T} \Rightarrow T \end{array}$$

Figure 14 Adding first-class functions to our Datalog frontend.

We extend $[\![.]\!]$ to also handle set expressions s, and we add a translation function $[\![.]\!]_{pred}$ for predicates. Figure 13 shows both translation functions. A set literal translates to a set of alternative terms and set union computes the union of alternative terms. A set comprehension builds all terms t generated by e for which all predicates are true.

We can only translate folds if the targeted Datalog engine supports aggregation over user-defined functions. In our experience, such user-defined functions must be implemented in the same language as the Datalog engine (e.g., C++ for Soufflé, a JVM language for Formulog and IncA). Thus, fold operations are considered built-in functions φ by Datalog engines. We extend the Datalog IR with aggregation accordingly:

$$\text{(Datalog terms)} \quad t \ ::= \ \ldots \mid \mathsf{aggregate}(R, \varphi, \varphi)$$

All we have left to do is to translate frontend functions f to built-in functions φ, which we assume function toPrimitiveFun accomplishes. In our implementation, we target IncA and compile user-defined frontend functions to Scala, which was straightforward. Soufflé does not support aggregation over user-defined functions, hence we cannot target Soufflé if the functional IncA program contains fold operations.

6.3 First-class functions and first-class sets

Functional IncA paves the road for transferring insights from functional programming languages to Datalog. Here, we exemplify this potential by studying defunctionalization in the context of functional IncA. Defunctionalization [18] is a well-known compilation technique that compiles higher-order functions into first-order functions and first-class function values into algebraic data. In particular, defunctionalization generates auxiliary *apply* functions that dispatch on the algebraic data to execute the corresponding function body. Since functional IncA supports first-order functions and algebraic data types, we can apply defunctionalization to extend functional IncA with first-class functions.

Figure 14 shows how we extend functional IncA's syntax with first-class functions. A function value is either a reference to a top-level function f or a lambda. Note that we permit lambdas to yield sets, since they will translate to first-order functions, which we translate to first-order relations. Finally, we adapt function application to allow any expressions in function position.

For example, consider an excerpt from our data-flow analyses of the While language:

```
def findExps(exp: Exp, f: Exp => Boolean): Set[Exp] = (exp match {
  case Var(s) => {}
  case Num(i) => {}
  case Add(e1, e2) => findExps(e1, f) ++ findExps(e2, f)
}) ++ (if (f(exp)) {exp} else {})
def freevars(exp: Exp): Set[String] = {varName(e) | e in findExps(exp, isVar)}
def availableExps(exp: Exp): Set[Exp] = findExps(exp, (e: Exp) => e match {
  case Var(s) => false
  case Num(i) => false
  case Add(e1, e2) => true })
```

We define a higher-order function `findExps` that selects all subexpressions satisfying predicate `f`. We use `findExps` twice, once to find all free variables of an expression and once to find all non-trivial subexpressions. We implement a standard defunctionalization transformation that translates this program into a first-order functional program:

```
data Defun0 = Funref0() | Lambda0()
def applyDefun0(fun: Defun0, e: Exp): Boolean = fun match {
  case Funref0() => isVar(e)
  case Lambda0() => e match {
    case Var(s) => false
    case Num(i) => false
    case Add(e1, e2) => true } }
def findExps(exp: Exp, f: Defun0): Set[Exp] = (...) ++ (if (applyDefun0(f, exp)) {exp}
     else {})
def freevars(exp: Exp): Set[String] = {varName(e) | e in findExps(exp, Funref0())}
def availableExps(exp: Exp): Set[Exp] = findExps(exp, Lambda0())
```

We can then translate the defunctionalized program to Datalog as described before. Thus, we have successfully extended functional IncA with first-class functions.

But how does this enable first-class sets? We already added support for first-order sets, which may only occur as function results. But since first-class functions translate to first-order functions, first-class functions may also yield sets. Thus, we can encode a first-class set `s` as a thunk `() => s`. For example, we can define a higher-order relation `transitive` as follows:

```
def transitive(cfg: () => Set[(Stm, Stm)]): Set[(Stm, Stm)] =
  cfg() ++ {(s1,s3) | (s1,s2) in cfg(), (s2,s3) in transitive(cfg)}
@main def transitiveFlow(prog: Stm): Set[(Stm, Stm)] = let cfg = () => flow(prog) in
  transitive(cfg)
```

Since function values become algebraic data, thunk-encoded sets are truly first-class: They can be assigned to variables and they can be passed as arguments. This shows how functional IncA permits insights from functional programming languages to carry over to Datalog, where they can unleash additional benefits.

7 Case Studies: Data-Flow Analyses and Clone Detection

We have presented functional Datalog frontend with relations that compiles to Datalog. In these final case studies, we want to demonstrate why this design is useful and how it enables a new way of implementing Datalog-based static analyses. To this end, we implemented flow-sensitive reaching definitions and interval analyses for the WHILE language in functional IncA. Additionally, we show how to describe clone detection of Java bytecode.

7.1 Data-Flow Analyses

Figure 15 shows an excerpt of the reaching definitions analysis, which determines where a variable was last defined. Our analysis implementation is completely standard except that we use a `retain` filter in place of the usual `kill` set. This is because functional IncA does not support negation yet, which is needed for set difference. We hope to extend functional IncA with negation in future work, but note that negation in Datalog is far from trivial and deserves a separate study.

The reaching definitions case study shows how we benefit from using functions and relations. The main benefit of functions is the ease of implementation in a well-known programming paradigm, as illustrated by `gen` in our example. The main benefit of relations is the implicit fixpoint semantics provided by Datalog. Specifically, note that `entry` and `exit` call each other unconditionally and diverges under functional-programming semantics. However, Datalog implicitly computes the least fixpoint of relations, which is computable because the relations are finite: There are only finitely many variables and assignments in any program. Here, functional IncA reaps the rewards of compiling to Datalog.

```
def gen(stm: Stm): Set[(String,Maybe[Stm])] = stm match {
  case Assign(x, a) => {(x, Just(stm))}
  case Sequence(s1, s2) => {}
  case If(c, s1, s2) => {}
  case While(c, s) => {} }
def retain(stm: Stm, x: String): Boolean = ...
def entry(stm: Stm, prog: Stm): Set[(String, Maybe[Stm])] =
  if (stm == init(prog))   {(x, Nothing()) | x in freevarsStm(prog)}
  else                     {(x,d) | (pred, stm) in flow(prog), (x,d) in exit(pred, prog)}
def exit(stm: Stm, prog: Stm): Set[(String,Maybe[Stm])] =
  gen(stm) ++ {(x,d) | (x,d) in entry(stm, prog), retain(stm, x)}
@main def allExits(prog: Stm): Set[(Stm, String, Maybe[Stm])] =
  {(s,x,d) | s in Stm, (x,d) in exit(s, prog)}
```

Figure 15 A reaching definitions analysis for the While language that we compile to Datalog.

```
data Val = BotVal() | IntervalVal(Interval) | BoolVal(Bool) | TopVal()
...
// entry_var, exit_var, and aeval as shown in Figure 2 (Introduction)
def add(v1: Val, v2: Val): Val = ...
def addInterval(iv1: Interval, iv2: Interval): Interval = iv1 match {
  case TopInterval() => TopInterval()
  case IV(l1, h1) => iv2 match {
    case TopInterval() => TopInterval()
    case IV(l2, h2) => IV(l1 + l2, h1 + h2) } }
def joinVal(v1: Val, v2: Val): Val = ...
def joinInterval(iv1: Interval, iv2: Interval): Interval = iv1 match {
  case TopInterval() => TopInterval()
  case IV(l1, h1) => iv2 match {
    case TopInterval() => TopInterval()
    case IV(l2, h2) => widenInterval(IV(Math.min(l1, l2), Math.max(h1, h2))) } }
```

Figure 16 Interval analysis of the While language using abstract interpretation.

In the reaching definitions analysis, the fixpoint computation within `entry` and `exit` only invokes simple functions `gen` and `retain`. Therefore, it is reasonable to implement the reaching definitions in Datalog directly, although we believe functional IncA is easier to use. In contrast, our second data-flow analysis implements an interval analysis that requires complex functions to abstractly interpret expressions. We show an excerpt of the interval analysis in Figure 16 and Figure 2. We use data type `Val` to represent abstract values and use relations `entry_var` and `exit_var` to map variables to their abstract value. For an `Assign` statement, `exit_var` invokes an abstract interpreter `aeval` that computes the abstract value of the assigned expression. Even for this simple WHILE language, the abstract interpreter already consists of 90 lines of functional code that compile to 342 lines of Datalog code. Moreover, `aeval` is part of the fixpoint loop, because it invokes `entry_var` for variable references, which invokes `exit_var`, which invokes `aeval`. Therefore, `aeval` really must translate to Datalog rules and cannot be represented as a built-in function, because then it could not invoke `entry_var`. Finally, note that we use a user-defined function `joinVal` to aggregate abstract values in `entry_var`. In particular, `joinVal` implements widening on intervals to ensure the analysis always terminates. All of these concerns are easy to address in functional IncA, because we can use functional programming while relying on Datalog's fixpoint semantics.

7.2 Clone Detection

Figure 17 shows an excerpt of how to construct an abstract syntax tree of Shimple code and apply clone-detection techniques such as testing for alpha-equivalence. Shimple is a variant of the Java bytecode representation Jimple [28] in SSA form. To access the

```
def getStm(inst: Instruction): Set[Stm] = {
  InvokeStm(recvExp, meth, args) |
    (inst, v) not in _AssignReturnValue,
    (inst, _, meth, recv, _) in _VirtualMethodInvocation,
    recvExp in getExp(recv),
    args in getArgs(inst, 0) } ++ ...
def getExp(v: String): Set[Exp] = ...
def getArgs(inst: Instruction, currentIdx: Int): Set[List[Exp]] = ...
def isStmClone(s1:Stm, s2:Stm, iPairs:NPairs, lPairs:NPairs): Boolean = (s1,s2) match {
    case (InvokeStm(r1, m1, a1), InvokeStm(r2, m2, a2)) =>
      m1 == m2 && isExpClone(r1, r2, iPairs) && isArgListClone(a1, a2, iPairs)
    ... }
```

Figure 17 Clone detection of Shimple Code.

Shimple representation, we extend functional IncA to read Soufflé relations, because the Doop framework [10] generates Soufflé facts. Using Soufflé facts enables us to detect clones of real-world Java programs. The Soufflé relations are prefixed by an underscore. Technically, we compile the Soufflé program and the functional program to a single Datalog program. However, we do not derive demand patterns for relations of the Soufflé program.

The function `getStm` constructs an abstract syntax tree representation of Shimple code. We highlight the case for constructing an invocation statement. We only generate an invocation statement for an instruction `inst` if it is a virtual method invocation and the instruction does not assign a return value for the given instruction. Note that functional IncA does not allow negation in general. However, it is possible to query Soufflé relations negatively as we do not apply the demand transformation to Soufflé relations. Hence, the demand transformation does not introduce negated dependency cycles. We generate the receiver of the method call by using function `getExp` which constructs an expression tree given a variable name. At last, we construct the argument of the given invocation statement by calling `getArgs`. Note while `getStm`, `getExp` and `getArgs` have `Set` as return type, the functions yield singleton sets. Returning a set is necessary due to the fact that we query Soufflé relations.

Next, we use the constructed abstract syntax trees as a basis to detect clones. We show a clone-detection function `isStmClone` which checks if the statements are alpha-equivalent. We traverse the statements `s1` and `s2` simultaneously while checking that the statements and inner expressions are equal. Because we rely on Soufflé relations generated by the Doop framework [10], we could integrate static analysis information such as points-to information into clone detection. The case study shows that describing alpha-equivalence of Java bytecode in a functional style is straightforward. It is possible to realize more sophisticated clone-detection techniques using functional IncA such as structural diffing [11].

8 Implementation and Performance Evaluation

In this section we will discuss our implementation and do an early performance evaluation.

8.1 Implementation

We implemented functional IncA by compiling it to a Datalog IR provided by the IncA framework. The Datalog IR can target two different backends namely IncA and Soufflé without any change to the underlying Datalog solvers VIATRA [29] and Soufflé [20] respectively. Our compiler generates Datalog code as shown in this paper, including the demand transformation. This implementation not only demonstrates the feasibility of our design, but also shows how advantageous it is to reuse existing Datalog solvers. In particular, the VIATRA Datalog solver supports incrementality: Changes in extensional relations trigger incremental updates in derived relations. We inherit this incrementality for free. For example, we can run the interval analysis of Subsection 7.1 incrementally by diffing the input programs and feeding the resulting patch to IncA [11]. Targeting Soufflé allows us to generate efficient and scalable C++ programs that run on multi-core machines. However, Soufflé does not support user-defined aggregation, hence we do not support translating functional IncA programs containing fold operations. The implementation is available at `https://gitlab.rlp.net/plmz/inca-scala`.

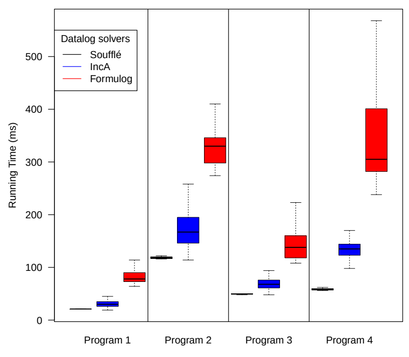
8.2 Performance Evaluation

We evaluate the performance of functional IncA and show that it is advantageous to use established solvers instead of implementing custom Datalog solvers for new frontends. We compare the running times of executing a data-flow analysis for the While language run with Souffé, IncA, and Formulog. We choose Soufflé and IncA as they are already established Datalog frameworks. We choose Formulog because it is one representative of the *frontend-first* approach which combines first-order ML functions with Datalog by implementing a custom Datalog solver. Even though IncA uses an incremental Datalog solver VIATRA [29], we do not measure the incremental performance of IncA which we leave as future work.

The data-flow analysis that we run is an adapted interval analysis. The analysis collects all integers $-100 \leq i \leq 100$, a variable can be assigned to. Whenever we encounter an integer $i < -100$ we return the default value -1000 and when we encounter an integer $i > 100$ we return 1000. We implement this cut-off to ensure that the data-flow analysis terminates in the presence of loops. We have chosen this type of analysis instead of an interval analysis, because an interval analysis requires user-defined aggregation which Formulog and Soufflé currently do not support. We implement four different programs as input of the data-flow analysis. The programs consist of nested *while* and *if* statements and are designed in such a way that a lot of information has to propagated along the edges of the control-flow graph.

For Formulog and IncA, both of which are Datalog solvers that run on the JVM, we first do 10 warmup runs and then measure 90 runs. We do not measure the time it takes to initialize the extensional database but only measure the running times of deriving the intensional database. For Soufflé, we compile an executable and measure the running time of the compiled Soufflé solver to derive the intensional database 9 times. Note that we do no warmup for Soufflé programs as they are compiled to C++ and then to executable machine code. We store the extensional database within input files and do not measure the I/O actions needed to read those input files. We load the contents of the input files into RAM by executing the compiled Soufflé program once. Hence, the following measured runs access the extensional database stored in RAM. We performed our benchmarks on a machine with an Intel Core i7 at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 11.4, Java 1.14.0_1.

We show the running times of deriving the intensional database in milliseconds for each program on the right-hand sight. We see that the custom Datalog solver for Formulog is slower than the established solvers such as Soufflé and IncA for all input programs. The Formulog solver is $\sim 3.7x$ slower than the Soufflé solver and $\sim 2.2x$ slower than the IncA solver. Note that the compiled executable of Soufflé has the fastest running time of all three solvers. This shows that is desirable to compile Datalog with functional constructs to already established Datalog dialects instead of implementing custom solvers for new Datalog frontends if possible.



9 Related Work

We propose functional programming with sets as a frontend for Datalog to replace Datalog's traditional constraint programming. This design differs from most prior works, which retain constraint programming as a basis and add functional aspects on top of it. Our approach has three advantages: (i) functional programming is easy to use, (ii) we can compute fixpoints across functions and relations, and (iii) we can reuse existing Datalog solvers. In the remainder of this section, we discuss related work.

IncA is an incremental Datalog framework that supports recursive aggregation over user-defined functions and data types [22, 23]. These user-defined functions and data types must be implemented in a JVM language and cannot query Datalog relations. The original frontend of IncA provides a shallow abstraction over Datalog called *pattern functions* [24]. These pattern functions consist of sequences of constraints and really are not comparable to the functional programming we support in functional IncA.

Flix [14] exposes constraint programming to the user, but extends it with functional programming. The runtime system of Flix executes functional code but also contains a custom Datalog solver. While functional and Datalog aspects are intertwined in Flix, they cannot interact as tightly as the functions and relations in our approach. Specifically, user-defined functions cannot recursively query derived relations, as required by our interval analysis. However, it is also not obvious how to extend our approach to compile Flix to Datalog, because Flix supports the generation of additional Datalog constraints at run time [13].

Formulog [9] combines first-order ML functions with Datalog and SMT solvers. In particular, Datalog rules can contain ML expressions and ML code can recursively query Datalog relations. Formulog's runtime understands both languages, which is why a custom Datalog solver was needed that can evaluate ML expressions and Datalog constraints interleaved. Our approach should naturally extend to Formulog. Indeed, we could add SMT solving as a built-in function `(def solveSMT(spec: String): String)` and rely on user-defined data types for SMT formulae and models, both of which are built-in types in Formulog.

Datafun [6] defines a higher-order functional programming language with sets and fixpoint semantics. From a language-design perspective, Datafun is the most closely related work. Both languages support commonly known functional expressions. One difference is how fixpoint computations are expressed in the surface syntax. Datafun provides a fixpoint expression which explicitly states over which function a fixpoint will be computed. However, in functional IncA the fixpoint computation is not explicitly given but implicitly given by the dependencies between functions. Datafun and functional IncA follow different design philosophies. Datafun provides a termination guarantee: If a Datafun program is well-typed, then a unique least fixed point exists and the program will terminate. Our language does not provide such a guarantee since a well-typed program can still diverge. Consequently, Datafun is more restrictive to guarantee termination while functional IncA gives developers more freedom (and responsibility). Datafun requires that the lattice type over which a fixpoint is computed does not contain an infinite ascending chain. One disadvantage of Datafun's design is that some programs that terminate in our system are not accepted by Datafun. For example, the interval analysis we presented is not well-typed in Datafun as the interval lattice has an infinite ascending chains. To ensure termination in functional IncA, we had to introduce widening to break the infinite ascending chains of the interval lattice. Many interesting static analyses use infinite lattices with infinite ascending chains. Hence, Datafun cannot be used to express such analyses. While it is an interesting question how to guarantee termination for as many programs as possible, our system is more viable to implement real-world programs. Another difference is that Datafun has its own bottom-up semantics which was recently extended to support semi-naïve evaluation [5]. In contrast, we translate programs to Datalog and utilize off-the-shelve Datalog solvers, which readily implement semi-naïve evaluation and other optimizations.

QL [7] is a logic programming language with object-oriented features such as classes and methods to structure logic programs. QL compiles to Datalog to encode inheritance and virtual dispatch of member predicates. We also propose to compile to Datalog, but focus on functional programming with algebraic data types. It would be interesting to see how we can extend functional IncA with object-oriented features and how these interact.

Mercury [21] is a logic programming language that consists of relations and rules deriving those relations. Like any Datalog, Mercury also supports the encoding of functions as relations, but in Mercury users can additionally annotate parameters as inputs and deterministic outputs. Mercury implements a custom Datalog solver that exploits such functional relations by executing them like a deterministic program. It would be interesting to explore *generic* Datalog optimizations that exploit functional relations, since we can easily generate the necessary annotations in functional IncA.

Bloom [2, 3] is a domain-specific language for distributed systems that uses the Datalog variant Dedalus [4] under the hood. Bloom provides built-in higher-order functions such as `map` and `reduce` that operate over collections. Bloom is embedded in Ruby and user-defined functions and data types can be written in Ruby, but these user-defined functions cannot access the contents of relations. Therefore, we cannot describe an interval analysis in the same style we have shown in the previous section in Bloom.

Soufflé [20] an efficient Datalog solver that can interpret Datalog rules directly or translate them to C++. It is possible to define user-defined functions as C++ functions, but again these functions cannot access the contents of relations. Soufflé has support for algebraic data types, but developers have to ensure that only finitely many values are constructed. For our use cases, this amounts to encoding the input relations by hand.

10 Conclusion

Datalog is supposedly declarative, but many programs are hard to express as constraints. We propose functional programming with sets as a new frontend for Datalog that solves this problem: *functional IncA*. Specifically, we translate functional IncA programs to Datalog and employ a demand transformation to ensure the Datalog program terminates whenever the original program terminates. While users of functional IncA only need to learn a single functional programming language, they enjoy Datalog's fixpoint semantics across functions and relations. Moreover, since all generated code is pure Datalog, we can use off-the-shelve Datalog solvers rather than building our own. Specifically, we implemented our approach as a frontend for IncA [23] as well as Soufflé [20] and demonstrated how easy it is to express complex Datalog programs with it. Our case studies include clone detection of real-world Java programs, program analyses, a program transformation, and an interpreter, all of which are easy to express functionally but translate to highly complex Datalog code. We have shown through early performance measurements that it is indeed desirable to use established Datalog solvers than implement custom solvers that embed a functional programming language as Formulog did. In future work, we want to investigate the performance of the generated Datalog code and study how compiler optimization can help. We also want to support negation in functional IncA, but the demand transformation potentially breaks the stratifiability of programs. We want to explore if the solution by Tekle and Liu [26] can be used. At last, we want to investigate how to properly debug functional IncA programs.

References

1 Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In Chen Li, editor, *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, pages 358–367. ACM, 2005. URL: `https://doi.org/10.1145/1065167.1065214`, `doi:10.1145/1065167.1065214`.

2 Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In Christine Morin and Gilles Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 223–236. ACM, 2010. URL: `https://doi.org/10.1145/1755913.1755937`, `doi:10.1145/1755913.1755937`.

3 Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260. www.cidrdb.org, 2011. URL: `http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf`.

4 Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2010. URL: `https://doi.org/10.1007/978-3-642-24206-9_16`, `doi:10.1007/978-3-642-24206-9\_16`.

5 Michael Arntzenius and Neel Krishnaswami. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.*, 4(POPL):22:1–22:28, 2020. URL: `https://doi.org/10.1145/3371090`, `doi:10.1145/3371090`.

6 Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional Datalog. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 214–227. ACM, 2016. URL: `https://doi.org/10.1145/2951913.2951948`, `doi:10.1145/2951913.2951948`.

7 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2016.2`, `doi:10.4230/LIPIcs.ECOOP.2016.2`.

8 Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10(3):255–299, 1991. Special Issue: Database Logic Progamming. URL: `https://www.sciencedirect.com/science/article/pii/074310669190038Q`, `doi:https://doi.org/10.1016/0743-1066(91)90038-Q`.

9 Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.*, 4(OOPSLA):141:1–141:31, 2020. URL: `https://doi.org/10.1145/3428209`, `doi:10.1145/3428209`.

10 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 243–262. ACM, 2009. URL: `https://doi.org/10.1145/1640089.1640108`, `doi:10.1145/1640089.1640108`.

11 Sebastian Erdweg, Tamás Szabó, and André Pacak and. Concise, type-safe, and efficient structural diffing. In *Programming Language Design and Implementation (PLDI)*. ACM, 2021.

12 Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1213–1216. ACM, 2011. URL: `https://doi.org/10.1145/1989323.1989456`, `doi:10.1145/1989323.1989456`.

13 Magnus Madsen and Ondrej Lhoták. Fixpoints for the masses: programming with first-class Datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA):125:1–125:28, 2020. URL: `https://doi.org/10.1145/3428193`, `doi:10.1145/3428193`.

14 Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 194–208. ACM, 2016. URL: `https://doi.org/10.1145/2908080.2908096`, `doi:10.1145/2908080.2908096`.

15 David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. ACM / Morgan & Claypool, 2018. URL: `https://doi.org/10.1145/3191315.3191317`, `doi:10.1145/3191315.3191317`.

16 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999.

17 André Pacak, Sebastian Erdweg, and Tamás Szabó. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.*, 4(OOPSLA):127:1–127:28, 2020. URL: `https://doi.org/10.1145/3428195`, `doi:10.1145/3428195`.

18 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. URL: `https://doi.org/10.1023/A:1010027404223`, `doi:10.1023/A:1010027404223`.

19 Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. On fast large-scale program analysis in Datalog. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 196–206. ACM, 2016. URL: `https://doi.org/10.1145/2892208.2892226`, `doi:10.1145/2892208.2892226`.

20 Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. A Datalog source-to-source translator for static program analysis: An experience report. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*, pages 28–37. IEEE Computer Society, 2015. URL: `https://doi.org/10.1109/ASWEC.2015.15`, `doi:10.1109/ASWEC.2015.15`.

21 Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *J. Log. Program.*, 29(1-3):17–64, 1996. URL: `https://doi.org/10.1016/S0743-1066(96)00068-4`, `doi:10.1016/S0743-1066(96)00068-4`.

22 Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA):139:1–139:29, 2018. URL: `https://doi.org/10.1145/3276509`, `doi:10.1145/3276509`.

23 Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental whole-program analysis in Datalog with lattices. In *Programming Language Design and Implementation (PLDI)*. ACM, 2021.

24 Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: a DSL for the definition of incremental program analyses. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 320–331. ACM, 2016. URL: `https://doi.org/10.1145/2970276.2970298`, `doi:10.1145/2970276.2970298`.

25 K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient datalog queries. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the*

12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria, pages 35–44. ACM, 2010. URL: `https://doi.org/10.1145/1836089.1836094`, `doi:10.1145/1836089.1836094`.

26 K. Tuncay Tekle and Yanhong A. Liu. Extended magic for negation: Efficient demand-driven evaluation of stratified Datalog with precise complexity guarantees. In Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang, editors, *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019*, volume 306 of *EPTCS*, pages 241–254, 2019. URL: `https://doi.org/10.4204/EPTCS.306.28`, `doi:10.4204/EPTCS.306.28`.

27 Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In Avi Silberschatz, editor, *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, pages 140–149. ACM Press, 1989. URL: `https://doi.org/10.1145/73721.73736`, `doi:10.1145/73721.73736`.

28 Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

29 Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling*, 15(3):609–629, Jul 2016. URL: `https://doi.org/10.1007/s10270-016-0530-4`, `doi:10.1007/s10270-016-0530-4`.