Datalog as a (non-logic) Programming Language

Dissertation submitted for the award of the title
"Doctor of Natural Sciences"
to the Faculty of Physics, Mathematics, and Computer Science
of Johannes Gutenberg University Mainz
in Mainz

André Pacak

Master of Science in Computer Science,
Technical University of Darmstadt, Germany,
born in Limburg an der Lahn, Germany.

Mainz, April 29, 2024

Referees:

[redacted]

Date of oral examination: November 08, 2024.
An electronic version of this dissertation is available at https://openscience.ub.uni-mainz.de.

D77

Contents

Summary

Datalog is a restricted query language for databases that was invented in the 1980s. In recent years, Datalog has seen a renewed interest in academia as well as industry. In particular, they use Datalog in application domains, including network monitoring, distributed computing, smart contract validation and decompilation, and non-incremental as well as incremental static program analysis.

Datalog is being used in these application domains for multiple reasons. First, Datalog has a *declarative least fixpoint semantics*. Programmers specify a Datalog program and provide the input in the form of relations. Then, a Datalog engine derives the minimal model satisfying the Datalog program in the form of relations, which is called the least fixpoint. Datalog's least fixpoint semantics is declarative because programmers do not need to consider the required steps to find the least fixpoint. Second, there exist *incremental algorithms* for executing Datalog programs. Instead of repetitively recomputing the least fixpoint from scratch when the input changes, an incremental Datalog algorithm processes the change of the input to update the output. That is, they only update the affected output relations. Incrementality can lead to order-of-magnitude speedups of running time.

Despite Datalog's strength, its low-level programming style is a major drawback. Datalog programs consist of horn clauses. A horn clause is an inference rule which derives tuples whenever a conjunction of predicates holds. Horn clauses do not always fit the computations a domain expert wants to express to tackle a problem. The data representation of Datalog, which are flat tuples, does not always fit the data that the domain operates on. These are representational gaps between the computations and data of the domain and horn clauses and flat tuples used by Datalog. These two gaps induce another gap when observing the behavior of Datalog programs, in particular, debugging Datalog programs. To debug Datalog programs, the state-of-the-art uses derivation trees as a an explanation vehicle, which fits well for inference rules. However, when experts express domain-specific computations operating on domain-specific data the visualization should fit the domain-specific computation and data instead. But how can we utilize Datalog's strengths while closing the representational gap between the domain and Datalog?

In this dissertation, we propose to close the representational gap by providing abstractions for Datalog. In particular, we propose to use linguistic abstractions for Datalog computations, data abstractions for structured data, and operational abstractions for debugging programs. First, we develop a domain-specific language for type system specifications, which compiles to Datalog and optimizes the generated program to enable efficient incremental updates. That is, we generate incremental type checker implementations by utilizing incremental Datalog engines. Second, we develop a functional frontend for Datalog that combines general-purpose functional programming with fixpoint computations supporting abstractions such as higher-order relations, parametric relations, and first-class relations. We develop a data abstraction that encodes structured input data such as abstract syntax trees relationally. Our relational encoding focuses on enabling efficient incremental updates. At last, we develop a new view on debugging Datalog programs by developing a top-down stepping debugger for Datalog, which acts as an operational abstraction. Based on the top-down stepping debugger we allow lifting the intermediate state of the Datalog program to its compilation source. Hence, we provide a debugger interface for debugging programs of the functional Datalog frontend.

Zusammenfassung

Datalog ist eine eingeschränkte Abfragesprache für Datenbanken, die in den 1980er Jahren entwickelt wurde. In den letzten Jahren hat Datalog sowohl in der Wissenschaft als auch in der Industrie neues Interesse geweckt. Insbesondere wird Datalog in Anwendungsbereichen wie Netzwerküberwachung, verteiltem Rechnen, Validierung und Dekompilierung von Smart Contracts sowie inkrementeller und nicht-inkrementeller statischer Programmanalyse verwendet.

Datalog wird in diesen Anwendungsbereichen aus mehreren Gründen eingesetzt. Erstens verfügt Datalog über eine deklarative Fixpunktsemantik. Programmierer implementieren ein Datalogprogramm und stellen die Eingabe in Form von Relationen bereit. Anschließend leitet eine Datalogengine das minimale Modell ab, das das Datalogprogramm in Form von Relationen erfüllt. Dieses Modell wird als der kleinste Fixpunkt bezeichnet. Die Fixpunktsemantik von Datalog ist deklarativ, da Programmierer die erforderlichen Schritte zur Ermittlung des kleinsten Fixpunkts nicht berücksichtigen müssen. Zweitens gibt es inkrementelle Algorithmen zur Ausführung von Datalogprogrammen. Anstatt den kleinsten Fixpunkt immer wieder von Grund auf neu zu berechnen, wenn sich die Eingabe ändert, verarbeitet ein inkrementeller Datalogalgorithmus die Änderung der Eingabe, um die Ausgabe zu aktualisieren. Das heißt, es werden nur die betroffenen Ausgaberelationen aktualisiert. Inkrementalität kann zu einer drastischen Beschleunigung der Laufzeit führen.

Trotz der Stärken von Datalog ist sein Low-Level-Programmierstil ein wesentlicher Nachteil. Datalogprogramme bestehen aus Hornklauseln. Eine Hornklausel ist eine Inferenzregel, die Tupel ableitet, wann immer eine Konjunktion von Prädikaten zutrifft. Hornklauseln eignen sich nicht immer für die Berechnungen, die ein Domänenexperte zur Lösung eines Problems ausdrücken möchte. Die Datenrepräsentation von Datalog, welche flache Tupel sind, passt nicht immer zu den Daten, mit denen die Problemdömane arbeitet. Es gibt also zwei Repräsentationsgefälle zwischen den Berechnungen und Daten der Problemdömane sowie den Hornklauseln und flachen Tupeln, die von Datalog verwendet werden. Diese beiden Gefälle erzeugen ein weiteres Gefälle beim Beobachten des Verhaltens von Datalogprogrammen, insbesondere beim Debuggen von Datalogprogrammen. Um Datalogprogramme zu debuggen, verwendet der Stand der Technik Ableitungsbäume als Erklärungsmittel, die gut zu Inferenzregeln passen. Wenn jedoch Experten domänenspezifische Berechnungen ausdrücken, die auf domänenspezifischen Daten basieren, sollte die Visualisierung zu den domänenspezifischen Berechnungen und Daten passen. Aber wie können wir die Stärken von Datalog nutzen und gleichzeitig die Repräsentationsgefälle zwischen der Domäne und Datalog schließen?

In dieser Dissertation schlagen wir vor, die Repräsentationsgefälle durch Bereitstellung von Abstraktionen für Datalog zu schließen. Insbesondere schlagen wir vor, linguistische Abstraktionen für Datalogberechnungen, Datenabstraktionen für strukturierte Daten und operationale Abstraktionen für das Debuggen von Programmen zu verwenden. Wir entwickeln eine domänenspezifische Sprache für Typsystemspezifikationen, die nach Datalog kompiliert und das generierte Programm optimiert, um effiziente inkrementelle Updates zu ermöglichen. Das bedeutet, wir generieren inkrementelle Type Checker, indem wir inkrementelle Datalogengines nutzen. Außerdem entwickeln wir ein funktionales Frontend für Datalog, die funktionale Programmierung mit Fixpunktberechnungen kombiniert und Abstraktionen wie Relationen höherer Ordnung, polymorphe Relationen und Relationen erster Klasse unterstützt. Wir entwickeln eine Datenabstraktion, die strukturierte

Eingabedaten wie abstrakte Syntaxbäume relational kodiert. Unsere relationale Kodierung konzentriert sich darauf, effiziente inkrementelle Updates zu ermöglichen. Schließlich entwickeln wir einen neuen Ansatz zum Debuggen von Datalogprogrammen, indem wir einen schrittweise operierenden Top-Down Debugger für Datalog entwickeln, der als operationale Abstraktion fungiert. Basierend auf dem Top-Down Debugger projizieren wir den Zwischenzustand des Datalogprogramms auf seine Kompilationsquelle. Somit bieten wir eine Debugger-Schnittstelle für das Debuggen von Programmen des funktionalen Datalogfrontends.

Acknowledgments

The last five years were the most intellectually challenging years of my life. I want to express my gratitude to everyone who helped me throughout the last few years while working towards this dissertation.

First and foremost, I am very thankful to my advisor [redacted] for asking me to join the Programming Languages group when he started his professorship in Mainz, even though I did not want to do a PhD at first. It was exciting to be part of the group at such an early stage. You taught me the scientific process and how to explore new ideas with rigor. Whenever I got stuck, you always pointed me in the right direction to push our ideas further. I am still in awe of how easy it is for you to communicate your thoughts, whether through writing or verbally, and that you are always equipped with a good example to explain your trail of thought.

I am really grateful to [redacted] for introducing me to IncA and helping me whenever I had problems with the implementation of IncA or when I had Datalog-related questions. Your PhD project was the foundation of the ideas that I explored in the last five years. It is impressive how you handled the workload of doing a PhD part-time while simultaneously being a software engineer in industry. I am very thankful for your detailed feedback on this dissertation.

I got really lucky to meet [redacted], who followed [redacted] to Mainz to finish his PhD. You helped me by sharing the experiences you collected in the previous years as a PhD student and preparing me for the ups and downs that are part of being a PhD student. I was always impressed by how excited you were about any research topic, even though it did not align with your research direction. You were always happy to help and introduced me to different research topics in the field of programming languages.

After [redacted] and [redacted] graduated, I was the only active member besides [redacted] for a while. Luckily, [redacted], [redacted], [redacted], and [redacted] joined one after another. I really enjoyed our insightful technical discussions and engaging conversations about life in general. I want to thank [redacted] and [redacted] for their detailed feedback on this dissertation. Maybe I will play squash with you someday.

I will forever be grateful to my parents, [redacted] and [redacted] and my brother, [redacted]. You always supported me and never pushed me in directions I did not want to take. You gave me the space whenever I needed it while still being there for me. At last, I'm extremely grateful to [redacted]. You never stopped believing in me and always provided your support when I questioned my abilities. Without you, I would not have had the energy and drive to finish this dissertation.

1 Introduction

Datalog is a logic programming language that was invented in the 1980s. It is a restricted query language for databases to express recursive queries. First, Datalog forbids constructing new data during evaluation and only allows to relate existing input data. Second, Datalog only allows limited non-monotonic queries such as negation. Datalog's restrictions ensure that a solution is always computable in finite time. There exist highly efficient and scalable engines for Datalog. Additionally, programmers consider Datalog to be a declarative language. These are some of the reasons why Datalog has gained the interest of academia and industry in the last decade. Hence, Datalog is used in various application domains. These domains include network monitoring [Abiteboul et al. 2005], distributed computing [Alvaro et al. 2010a, 2011], smart contract validation [Grech et al. 2018; Lagouvardos et al. 2020], smart contract decompilation [Grech et al. 2019, 2022], and program analysis [Hajiyev et al. 2006; Bravenboer and Smaragdakis 2009a,b; Avgustinov et al. 2016; Madsen et al. 2016; Szabó et al. 2021].

1.1 What has Logic Programming to Offer?

But why is Datalog is considered to be declarative? A Datalog program specifies the solution to a problem but does not state how to compute the solution. Hence, programmers are not required to think about how the program is executed, and thus the order of atoms within a rules body.

Figure 1.1 shows how programmers and users interact with Datalog. Programmers write Datalog programs, which consist of Datalog rules. Each Datalog rule forms a horn clause $a_1 \wedge \ldots a_n \rightarrow p(X_1, \ldots, X_n)$ where $a_1, \ldots, a_n$ form the body of the rule and $p(X_1, \ldots, X_n)$ forms the head of the rule. An atom can be a positive predicate call $q(t_1, \ldots, t_n)$ or a negative predicate call **not** $q(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms. Terms are either variables or constants such as numbers and strings. A Datalog program forms a disjunctive normal form. A disjunction connects rules belonging to the same predicate, while atoms within a rule are connected by a conjunction. The surface syntax of Datalog rules has the following structure:

$$p(X_1, \ldots, X_n) \text{ :- } a_1, \ldots, a_n.$$

1

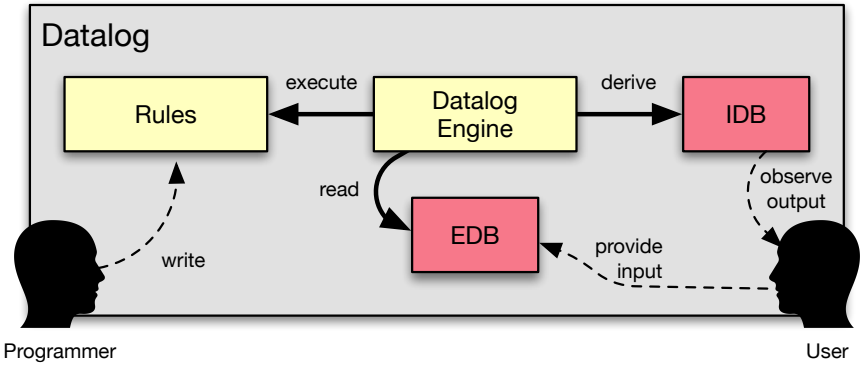


Figure 1.1: The basic architecture of a Datalog engine.

To execute a Datalog program, users provide input as a collection of relations called *extensional database*. The extensional database is also called *EDB*. Users can observe the output of a Datalog program by inspecting the *intensional database*, which is called IDB. Again, the IDB is a collection of relations. The IDB is derived based on the Datalog rules describing the executed program, which can query EDB and IDB relations.

Datalog's least fixpoint semantics. There are multiple ways of defining Datalog's semantics. Modern Datalog engines are based on fixpoint-theoretic semantics, hence we will focus on fixpoint-theoretic semantics in the remainder of this dissertation. The fixpoint-theoretic semantics applies the given Datalog rules over and over again until a fixpoint has been found. The fixpoint is always computable in a finite amount of time because standard Datalog forbids to construct new data during run time. The fixpoint-theoretic semantics not only finds an arbitrary fixpoint but actually finds the *least fixpoint* for a given Datalog program and EDB. Hence, Datalog employs a *least fixpoint semantics*, which is a key feature of Datalog. In particular, Datalog has a *declarative* least fixpoint semantics. Programmers only need to specify the solution using Datalog rules and do not describe how to actually compute the least fixpoint. Instead, a Datalog engine performs the required steps to compute the least fixpoint.

To highlight the strengths of Datalog's declarative least fixpoint semantics, we consider one common application of Datalog: Andersen-style points-to analysis [Andersen 1994]. The industry-strength Doop static analysis framework has its roots in the Andersen-style formulation of points-to. Doop provides sophisticated points-to analyses for JVM bytecode written in Datalog [Bravenboer and Smaragdakis 2009b]. We consider an idealized version of a flow-sensitive points-to analysis for a toy language inspired by the Doop framework.

To use Datalog to analyze a program, we need to provide the relevant information about the program as input, and encode it relationally as an EDB. Based on the relational encoding schema, it is possible to define a Datalog program that derives a points-to set for each variable. Consider the following program and its relational encoding:

Program:

```
1: a = new A #1
2: if (a.g + 2 > 5)
3:     b = new A #2
   else
4:     b = a
5: a = b
```

EDB Relations:

$\mathsf{new}(1, a, \#_1)$. $\mathsf{next}(1, 2)$.
$\mathsf{new}(3, b, \#_2)$. $\mathsf{next}(2, 5)$.
$\mathsf{assign}(4, b, a)$. $\mathsf{if}(2, 3, 4)$.
$\mathsf{assign}(5, a, b)$.

Note that we mark allocation sites, e.g., the allocation site in line 1 is marked with $\#_1$. The relational encoding uses relation new to encode object allocations. For example, at location 1 we allocate a new object $\#_1$ and assign it to variable a. Therefore, the EDB contains tuple $\mathsf{new}(1, a, \#_1)$. The relation assign encodes assigning a variable to another variable. We encode flow-sensitive information with next and if tuples. The predicate next enumerates all direct successors of statements. The tuple $\mathsf{if}(2, 3, 4)$ states that the if statement at location 2 continues for the then branch at location 3 and for the else branch at location 4. Note that next does not encode connections from the if statement to its *then* and *else* branches because this information is already encoded by if. Given these two EDB predicates, we define an IDB relation cflow describing the edges of the control-flow graph:

$$
\begin{array}{lll}
\mathsf{cflow}(l1, l2) & \text{:-} & \mathsf{next}(l1, l2). \\
\mathsf{cflow}(l1, l2) & \text{:-} & \mathsf{if}(l1, l2, \_). \\
\mathsf{cflow}(l3, l4) & \text{:-} & \mathsf{if}(l1, l2, \_),\ \mathsf{last}(l2, l3),\ \mathsf{next}(l1, l4). \\
\ldots &&
\end{array}
$$

The first rule transfers all tuples of EDB relation next to IDB relation cflow. The second rule derives a control-flow edge connecting the if statement and the first statement of the *then* branch. The third rule is a bit more involved. It derives an edge connecting the last statement of the *then* branch and the statement following the corresponding if statement. Note how the third rule utilizes last, which determines the last statement of a sequence of statements. For brevity, we do not discuss the omitted rules which handle other control-flow constructs and we do not show the definition of last.

Based on the predicate cflow, we can derive all paths of the control-flow graph captured by the cpath predicate:

$$
\begin{array}{lll}
\mathsf{cpath}(l1, l2) & \text{:-} & \mathsf{cflow}(l1, l2). \\
\mathsf{cpath}(l1, l3) & \text{:-} & \mathsf{cflow}(l1, l2),\ \mathsf{cpath}(l2, l3).
\end{array}
$$

The predicate cpath calculates the transitive closure over cflow by recursively querying itself. There is a control-flow path between $l1 \rightarrow l2$, if a control-flow edge $\mathsf{cflow}(l1, l2)$ connects the locations. Additionally, there is a control-flow path $l1 \rightarrow l3$, if we can prepend a control-flow path $l2 \rightarrow l3$ with an edge connecting $l1$ and $l2$.

Given that we can derive control-flow paths of a program, we can now define the flow-sensitive points-to predicates pt:

$$
\begin{array}{lll}
\mathsf{pt}(l, x, o) & \text{:-} & \mathsf{new}(l, x, o). \\
\mathsf{pt}(l2, to, o) & \text{:-} & \mathsf{assign}(l2, to, from),\ \mathsf{pt}(l1, from, o),\ \mathsf{cpath}(l1, l2).
\end{array}
$$

1

The predicate $\mathsf{pt}(l, x, o)$ states that variable x points to object o at location l. The first rule of pt states that whenever we assign an allocated object to a variable, it points to the object at the current location. The second rule transfers the points-to set of the right-hand side of the assignment to the left-hand side. Note how we utilize predicate cpath to ensure that we only transfer points-to information for locations that can reach the assignment statement. The formulation of the flow-sensitive points-to analysis is intentionally incomplete. For a complete formulation, we must stop propagating points-to information whenever a re-assignment occurs. We choose this formulation as an explanation vehicle and choose simplicity over completeness. We want to highlight that cpath and pt are both recursively defined. While the control-flow graph is acyclic, the points-to information is cyclic because of the assign statements at locations 4 and 5. This example shows the strength of Datalog: *declarative least fixpoint computations* for recursive programs over cyclic dependencies.

But how do Datalog engines actually compute the fixpoint? There a two classes of evaluation strategies: top-down and bottom-up. While top-down evaluation follows a proof-theoretic semantics, bottom-up evaluation is based on fixpoint-theoretic semantics. All modern Datalog systems use a bottom-up evaluation strategy because in general bottom-up is more efficient than top-down [Ullman 1989]. The easiest bottom-up evaluation strategy is called naïve evaluation. The naïve evaluation strategy applies all rules simultaneously one iteration at a time until a fixpoint has been reached [Green et al. 2013]. Consider the following iterations where we only look at tuples derived for pt:

1.	2.	3.	4.
$\mathsf{pt}(1, a, \#_1)$	$\mathsf{pt}(1, a, \#_1)$	$\mathsf{pt}(1, a, \#_1)$	$\mathsf{pt}(1, a, \#_1)$
$\mathsf{pt}(3, b, \#_2)$	$\mathsf{pt}(3, b, \#_2)$	$\mathsf{pt}(3, b, \#_2)$	$\mathsf{pt}(3, b, \#_2)$
	$\mathsf{pt}(4, b, \#_1)$	$\mathsf{pt}(4, b, \#_1)$	$\mathsf{pt}(4, b, \#_1)$
	$\mathsf{pt}(5, a, \#_2)$	$\mathsf{pt}(5, a, \#_2)$	$\mathsf{pt}(5, a, \#_2)$
		$\mathsf{pt}(5, a, \#_1)$	$\mathsf{pt}(5, a, \#_1)$

Each iteration re-derives tuples that were derived in all previous iterations, which leads to a lot of unnecessary rule applications. For example, naïve evaluation re-derives $\mathsf{pt}(1, a, \#_1)$ 4 times in total. To find the least fixpoint, naïve evaluation checks if the current iteration derived the same tuples as the previous iteration. In particular, the fourth iteration derives the same tuples as the third iteration. This strategy is sufficient to find a fixpoint. And indeed, it finds the least fixpoint: the smallest set of tuples satisfying the Datalog program. But this is a very inefficient evaluation strategy because we re-derive tuples over and over again. Hence, all state-of-the-art Datalog solvers use another strategy called *semi-naïve evaluation* [Green et al. 2013]. This strategy aims to avoid deriving tuples multiple times. That is, once a rule derived a tuple in a previous iteration, the rule will never re-derive this tuple again. Therefore, we only derive a tuple once for a given rule. Consider the iterations using semi-naïve evaluation:

1.	2.	3.	4.
$\mathsf{pt}(1, a, \#_1)$	$\mathsf{pt}(4, b, \#_1)$	$\mathsf{pt}(5, a, \#_1)$	
$\mathsf{pt}(3, b, \#_2)$	$\mathsf{pt}(5, a, \#_2)$		

Note how the number of tuples derived drops in every iteration. To detect that a fixpoint has been reached, we must continue until no new tuple is derived. In the example above,

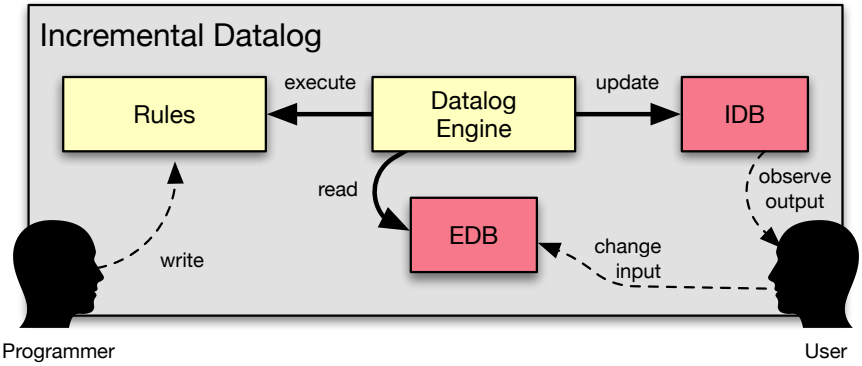


Figure 1.2: The architecture of an incremental Datalog engine.

the fourth iteration does not derive any tuples which implies a fixpoint has been found. Naïve and semi-naïve evaluation always derive the same answer, but semi-naïve evaluation improves the performance significantly. That is why all state-of-the-art Datalog solvers use semi-naïve evaluation.

Datalog's incremental execution. Not only does Datalog have an efficient evaluation strategy, but the declarative nature of Datalog allows for efficient incrementalization algorithms. But what does incrementality offer? A non-incremental computation f takes input x and produces output y. In contrast, an incremental computation Δf takes a change in the input Δx and produces a change in the output Δy. Incremental computations only process the change in the input to update the output instead of recomputing the output from scratch given the updated input. Incremental computations compute the same output as if we recompute the non-incremental computation for the changed input. That is, given input x_1 and an updated input x_2 where $x_2 = x_1 + \Delta x$ the following property holds:

$$f(x_2) = f(x_1) + \Delta f(\Delta x)$$

Thus, incrementality can achieve fast update times when the input changes under the condition that a small change in the input results in a small change in the output.

A key feature of Datalog is that we can incrementally execute a Datalog program while neither programmers nor users have to adapt the program for incrementality. Users only need to use an available incremental execution algorithm. Hence, *users can execute Datalog programs incrementally for free.*

Figure 1.2 highlights the model of an incremental Datalog engine. While programmers do not have to change the Datalog program, users interact with the engine differently. Instead of only providing the EDB, users can insert and delete tuples of the EDB after the initial run. The incremental Datalog engine reacts to tuple insertions and deletions. The engine then propagates the changes throughout the affected IDB relations. After the changes have been fully propagated, users can observe the updated IDB. Consider the following change to the program introduced above:

1

Old Program:

```
1: a = new A #1
2: if (a.g + 2 > 5)
3:     b = new A #2
   else
4:     b = a
5: a = b
```

Changed Program:

```
1: a = new A #1
2: if (a.g + 2 > 5)
3:     b = new A #2
   else
4:     b = new A #3
5: a = b
6: c = a
```

We change the assignment at location 4 to an object allocation and introduce a new assignment $c = a$ at the end of the program. These changes result in the following insertions and deletions:

$$+\mathsf{assign}(6,c,a) \qquad +\mathsf{next}(5,6) \qquad -\mathsf{assign}(4,b,a) \qquad +\mathsf{new}(4,b,\#_3)$$

Note how the change of the object allocation results in a deletion of the assign tuple and an accompanying new insertion.

Given insertions and deletions of EDB tuples, we introduce the standard incremental Datalog algorithm called *DRed*, which is short for *Delete and Re-derive* [Gupta et al. 1993]. It is based on semi-naïve evaluation and has two phases:

1. *Deletion Phase*: Apply semi-naïve evaluation to delete all tuples of the IDB that transitively depend on a deleted tuple.
2. *Re-derivation Phase*: Apply semi-naïve evaluation to re-derive tuples that have alternative derivations, and derive new tuples based on inserted tuples.

Note that the deletion phase is an over-approximation as it potentially deletes more tuples than necessary, especially considering tuples are derivable by multiple rules. Hence, the re-derivation phase not only derives new tuples based on EDB insertions but also corrects the over-deletion of the previous phase. We present an optimized version of DRed above. The original formulation operates in three phases, where the re-derivation phase only corrects the over-deletion. And only afterwards, derives new tuples based on insertions in the insertion phase.

Considering the optimized version of DRed and the change above, we first delete all tuples that depend on $\mathsf{assign}(4,b,a)$:

$$-\mathsf{pt}(4,b,\#_1) \qquad -\mathsf{pt}(5,a,\#_1)$$

Afterwards, the re-derivation phase derives all tuples based on the insertions and corrects the potential over-deletion of the previous phase:

$$+\mathsf{pt}(4,b,\#_3) \qquad +\mathsf{pt}(5,a,\#_3) \qquad +\mathsf{pt}(6,c,\#_2) \qquad +\mathsf{pt}(6,c,\#_3)$$

Hence, all IDB relations are updated correctly.

Incremental Datalog has been used for multiple domains. For example, the static analysis framework IncA executes static analyses incrementally by utilizing an incremental Datalog engine [Szabó 2021]. IncA has been used to incrementalize different static analyses

while enabling sub-second update times. These static analyses include FindBugs for Java, flow-sensitive points-to analysis and well-formedness checks for C, strong-update points-to analysis, and Interval analysis for Jimple [Szabó et al. 2016, 2018, 2021]. In particular, IncA introduces the incremental Datalog algorithm DRed_L [Szabó et al. 2018], which enables incremental processing of Datalog programs with user-defined recursive aggregation over lattices. Additionally, they introduce another incremental Datalog algorithm LADDDER, which scales to whole-program analyses [Szabó et al. 2021]. The algorithm is based on differential dataflow which is a framework for processing continuous stream data [McSherry et al. 2013]. Another Datalog solver called Soufflé introduces elastic incrementality [Zhao et al. 2021]. They incrementalize a subset of the DOOP framework, conflict-free replicated data types, and a medical ontology inference task called Galen [Rector and Rogers 2006]. Other applications tackled using the incremental Datalog engine DDlog include a controller for virtual networks and firewall management for virtual networks [Ryzhyk and Budiu 2019]. DDlog is also based on differential dataflow.

To summarize, Datalog's strength is its efficient declarative least fixpoint semantics, which enables programmers to solve fixpoint problems using recursive rules over cyclic dependencies declaratively. Additionally, Datalog provides efficient incremental algorithms that execute Datalog programs incrementally without changing the program itself.

1.2 Representational Gap of Datalog

Datalog has several strengths, including its least fixpoint semantics and efficient incremental engines. But where is Datalog lacking? One key disadvantage is that Datalog uses a low-level programming style: writing down horn clauses. This programming style entails a gap between the problem domain and Datalog, particularly a *representational gap*. Larman [2001] defines the term representational gap as follows:

> *"The gap between our mental model of the domain and its representation in software."*

We highlight the representational gap between Datalog and the mental model of the domain in Figure 1.3. The representational gap is two-fold. First, there is a representational gap between the problem that domain experts want to solve and the style in which Datalog programs are written. Secondly, there is a gap between the data of the problem domain and the representation of data that Datalog operates on. These two representational gaps induce another gap: observing the behavior of programs. Whenever a Datalog program is faulty, domain experts want to identify and fix the problem using a debugger. Domain experts have a specific mental model of a solution when solving a domain problem. For example, consider a function traversing structured data such as an interpreter. Experts have a mental model of the "control flow" of the function that visits expressions in a fixed order. However, the low-level programming style and Datalog's debugging style do not match the mental model of domain experts. That is, Datalog's debugging style does not highlight the control flow of the computation.

In the remainder of this section, we discuss the representational gaps based on the flow-sensitive points-to analysis introduced in the previous section. The driving example in the remainder of this section is the extension of the analysis with an abstract interpreter for expressions to increase the precision of the analysis.

1

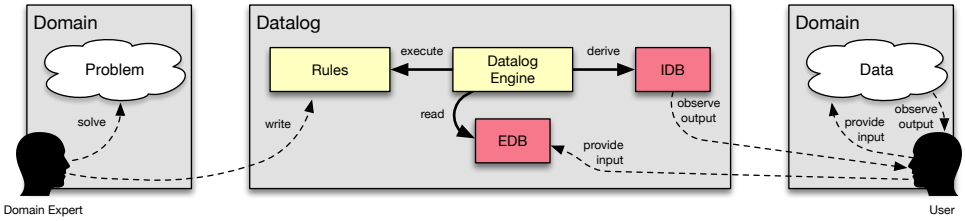


Figure 1.3: Representational gap between the domain's problem/data and Datalog's programming abstractions.

Representational Gap 1: No Abstraction of Data. Before we define and integrate the abstract interpreter using Datalog, we need to encode expressions relationally to provide them as part of the EDB. Datalog only accepts sets of tuples as input, while these tuples are not nested and only range over primitive data such as integers and strings. In the previous section, we already encoded statements relationally. However, the statements have a simple sequential structure. In contrast, expressions are inherently nested. But how can we encode nested data such as expressions relationally? To encode expression trees, we can define a relation for each expression kind that ranges over the expression itself and its children. Consider expression $a.g + 2 > 5$. We define a gt tuple linking the expression with its left-hand child $a.g + 2$ and right-hand child 5. Hence, we define tuple $\mathsf{gt}(a.g + 2 > 5, a.g + 2, 5)$. When we apply this strategy, we get the following relational encoding for the expressions of the original program:

$$\begin{array}{lll} \mathsf{gt}(a.g+2>5, a.g+2, 5) & \mathsf{add}(a.g+2, a.g, 2) & \mathsf{fieldread}(a.g, a, g) \\ \mathsf{var}(a, a) & \mathsf{lit}(5, 5) & \mathsf{lit}(2, 2) \end{array}$$

Based on this relational encoding, we can implement an abstract interpreter. However, one of Datalog's strengths is its incremental execution, but this encoding does not enable efficient incremental updates. Consider the following change:

$$a.g + 2 > 5 \qquad \rightsquigarrow \qquad a.g + 4 > 5$$

This change will result in the following changes in the relational encoding:

$$\begin{array}{lll} -\mathsf{gt}(a.g+2>5, a.g+2, 5) & -\mathsf{add}(a.g+2, a.g, 2) & -\mathsf{lit}(2, 2) \\ +\mathsf{gt}(a.g+4>5, a.g+4, 5) & +\mathsf{add}(a.g+4, a.g, 4) & +\mathsf{lit}(4, 4) \end{array}$$

Even though we only changed an integer literal, we delete and re-insert all tuples of the parent expressions to reflect the change in a child node. In particular, we need to delete and re-insert a gt, add, and lit tuple. A small change in the source program results in a big change in the relational encoding. The change in the relational encoding is propagated by the incremental Datalog algorithm DRed, resulting in potentially unnecessary deletions and re-derivations during an incremental update.

Even though the relational encoding does not enable efficient incremental updates, we can still use this encoding as a basis to implement an abstract interpreter for expressions.

Representational Gap 2: No Abstraction for Computations. We want to improve the precision of the flow-sensitive points-to analysis by introducing abstract interpretation of expressions in big-step style following the work of Darais et al. [2017] and Keidel et al. [2018]. Based on the abstract interpretation result, we can potentially decide whether only the *then* branch or the *else* branch of the if statement is executed. Hence, the control-flow graph is more precise, which can result in a more precise analysis result.

A big-step abstract interpreter traverses the expression tree recursively while computing abstract values for each sub-expression and then combines the results. We consider a constant lattice for boolean values and a sign lattice for integers. A boolean can either be true (True), false (False), or it is not decidable (BTop). An integer can either be positive (Pos), zero (Zero), negative (Neg), or it is not decidable (ITop). We define a new predicate $\mathsf{eval}(l,e,r)$ which relates a location l, the abstractly evaluated expression e, and the abstract interpretation result r. Note that location l contains expression e, and we use it to track the points-to information when abstractly interpreting an expression. We utilize the relational encoding of expression trees shown above to define Datalog rules that derive eval. Consider this excerpt of Datalog rules that define eval:

$$
\begin{aligned}
\mathsf{eval}(l,e,r) \quad &:\!\text{-} \quad \mathsf{input\_eval}(l,e),\ \mathsf{add}(e,e1,e2),\ \mathsf{eval}(l,e1,r1),\ \mathsf{eval}(l,e2,r2),\\
&\qquad r1 = \mathsf{Pos},\ r2 = \mathsf{ITop},\ r = \mathsf{ITop}.\\
\mathsf{eval}(l,e,r) \quad &:\!\text{-} \quad \mathsf{input\_eval}(l,e),\ \mathsf{add}(e,e1,e2),\ \mathsf{eval}(l,e1,r1),\ \mathsf{eval}(l,e2,r2),\\
&\qquad r1 = \mathsf{Pos},\ r2 = \mathsf{Pos},\ r = \mathsf{Pos}.\\
&\ldots
\end{aligned}
$$

Both rules first query a helper predicate input_eval, which enumerates all inputs required to derive eval. We will explain why querying input_eval is necessary shortly. Both rules query add to deconstruct the expression into its sub-expression $e1$ and $e2$. Then they query eval recursively on both sub-expressions to compute the abstract value for them. Both rules require that the result of sub-expression $e1$ evaluates to Pos, and only then do the rules differ. The first rule requires that the right sub-expression $e2$ evaluates to ITop. Hence, the abstract interpreter produces ITop for expression e, because the resulting number can either be positive, zero, or negative when adding a positive number with an arbitrary number. The second rule requires that the right sub-expressions evaluates to Pos, which leads to the fact that the addition expression also evaluates to be Pos.

Using the abstract interpreter predicate eval, we can refine the Datalog rules for cflow that derive control-flow edges when considering if statements:

$$
\begin{aligned}
&\mathsf{cflow}(l1,l2) \quad :\!\text{-} \quad \mathsf{if}(l1,e,l2,\_),\ \mathsf{eval}(l1,e,\mathsf{True}).\\
&\mathsf{cflow}(l1,l2) \quad :\!\text{-} \quad \mathsf{if}(l1,e,l2,\_),\ \mathsf{eval}(l1,e,\mathsf{BTop}).\\
&\mathsf{cflow}(l3,l4) \quad :\!\text{-} \quad \mathsf{if}(l1,e,l2,\_),\ \mathsf{eval}(l1,e,\mathsf{True}),\ \mathsf{last}(l2,l3),\ \mathsf{next}(l1,l4).\\
&\mathsf{cflow}(l3,l4) \quad :\!\text{-} \quad \mathsf{if}(l1,e,l2,\_),\ \mathsf{eval}(l1,e,\mathsf{BTop}),\ \mathsf{last}(l2,l3),\ \mathsf{next}(l1,l4).\\
&\ldots
\end{aligned}
$$

We modify the Datalog rules such that they are only applicable if the conditional of the if statement evaluates to an approximation of true, namely True or BTop.

Multiple problems arise in the encoding of eval. First, there is a lot of code duplication between both rules, sharing the first five atoms. Even worse, the two rules shown above are not sufficient to abstractly interpret addition expressions. We require 11 Datalog rules

in total to fully define eval for addition expressions. And all Datalog rules have the same prefix:

$$\mathsf{input\_eval}(l,e),\ \mathsf{add}(e,e1,e2),\ \mathsf{eval}(l,e1,r1),\ \mathsf{eval}(l,e2,r2)$$

Note that there are even more rules describing eval, which consider other expressions such as greater-than that we omit for brevity. The problem is that Datalog does not have abstractions for conditionals, which is a standard abstraction for programming paradigms such as structured programming. Whenever we want to encode structured programming ideas, it results in a lot of code duplication, which is hard to write and maintain.

Second, $\mathsf{eval}(l,e,r)$ encodes a function that generates new values during run time. This can result in a diverging Datalog program. However, there are only a finite number of locations and expressions in the actual program. And, the predicate encodes a function where the "inputs" l and e, which are finite, uniquely determine the "output" r. Hence, only a finite number of values are generated during run time, and thus, the Datalog program will terminate [Szabó et al. 2018]. This insight requires extensive knowledge of Datalog.

Lastly, how do we ensure that we only abstractly interpret the expressions necessary when refining the control-flow graph? Modern Datalog engines evaluate programs bottom-up: compute all derivable information. However, we have the domain knowledge that eval describes a recursive-descent function that traverses the expression tree top-down. To selectively force top-down evaluation in Datalog, we insert the predicate input_eval, which collects all inputs of the function dynamically. Another question arises: how do we determine the rules for input_eval? We must identify all call sites of eval and use the surrounding atoms to derive new rules. This requires a deep knowledge of Datalog. Luckily, we can achieve this by applying a transformation called magic-set transformation [Beeri and Ramakrishnan 1991] or its cousin the demand transformation [Tekle and Liu 2010]. But again, this requires extensive knowledge of Datalog.

Representational Gap 3: No Abstraction for Observable Behavior. But what happens when the Datalog rules describing the abstract interpreter are buggy? How do Datalog programmers detect and fix a bug? The state-of-the-art debugging technique for Datalog is provenance-based [Zhao et al. 2020]. That is, programmers inspect witnesses of data provenance to identify which tuples are required to derive a specific tuple. In particular, the state-of-the-art uses derivation trees as a visualization vehicle for highlighting data provenance. Programmers inspect derivation trees to identify which tuples are needed to derive a specific tuple, and which Datalog rules are applied to derive the required tuples. Derivation trees are read from bottom to top. The final derivation result tuple marks the root of the derivation tree, and the leaves mark EDB tuples. Every other node of the derivation tree represents an IDB tuple. For example, we want to explore why the expression $a.g+2>5$ evaluates to BTop. We inspect the derivation tree for the tuple $\mathsf{eval}(2,a.g+2>5,\mathsf{BTop})$:

$$\text{R6}\ \dfrac{\dfrac{\dots}{\mathsf{input\_eval}(2,a.g+2>5)}\quad \mathsf{gt}(a.g+2>5,a.g+2,5)\quad \dfrac{\dots}{\mathsf{eval}(2,a.g+2,\mathsf{ITop})}\quad \dfrac{\dots}{\mathsf{eval}(2,5,\mathsf{Pos})}}{\mathsf{eval}(2,a.g+2>5,\mathsf{BTop})}$$

The derivation tree above shows that tuples $\mathsf{input\_eval}(2,a.g+2>5)$, $\mathsf{gt}(a.g+2>5,a.g+2,5)$, $\mathsf{eval}(2,a.g+2,\mathsf{ITop})$, and $\mathsf{eval}(2,5,\mathsf{Pos})$ are required to derive the tuple in question. Additionally, the derivation tree shows that the following Datalog rule derives the tuple, and

the rule is applicable using the tuples listed above:

```
eval(l,e,r) :- input_eval(l,e), gt(e,e1,e2), eval(l,e1,ITop), eval(l,e2,Pos).      // R6
```

This Datalog rule handles abstractly interpreting greater-than expressions where the left-hand child evaluates to ITop and the right-hand child to Pos. Note that while the gt tuple marks a leaf node because gt is an EDB relation, the other tuples have sub-derivation trees because they are all IDB tuples. Provenance-based debugging techniques have a major drawback. Concretely, they only provide derivation trees for ground tuples. Ground tuples are tuples where each element is constant. For example, $(1,2,3)$ is a ground tuple, and $(1,2,X)$ is not. Hence, inspecting derivation trees for partial queries is not possible. Additionally, provenance-based debugging does not enable inspecting derivation trees for underivable tuples because there is no valid derivation tree for an underivable tuple.

But is this the correct way of debugging rules describing an abstract interpreter? Programmers have a specific mental model of the computations they express. Concretely, the abstract interpreter defines a recursive-descent function traversing expressions. In particular, the abstract interpreter visits sub-expressions in a fixed order following an expected control flow. Consider an abstract interpreter implemented using a procedural programming language. Programmers use stepping debuggers to identify bugs in procedural programs. Using a stepping debugger includes (i) identifying a buggy scenario, (ii) setting a breakpoint, (iii) stepping through the program, and (iv) inspecting intermediate execution states. For example, a stepping debugger highlights each intermediate state and highlights the control flow with the inspected execution trace for the abstract interpreter. A derivation tree fits for logic programs, but the derivation tree does not make this control flow explicit. This forms a representational gap between the observable program behavior domain experts expect and the observable program behavior Datalog provides.

To summarize, we presented three representational gaps: the representational gap of data, the representational gap of computations, and the representational gap of observable behavior. But, we change the order of the representational gaps because we believe that the representational gap of computations is more important than the gap of data. However, we needed to discuss the encoding of nested expressions first before discussing an encoding of the recursive-descent abstract interpreter acting on the relational encoding of nested expressions. We arrive at the following order for representational gaps, which we will reference in the remainder of this dissertation:

(RG1) Computation: The representational gap between the computations required by the domain and the computation style Datalog offers.

(RG2) Data: The representational gap between the data the domain operates on and the data Datalog operates on.

(RG3) Observable Behavior: The representational gap between the observable program behavior domain experts expect and Datalog provides.

This dissertation claims the following thesis:

> It is feasible and useful to provide programming abstractions for Datalog to close representational gaps by shielding programmers and users from logic programming.

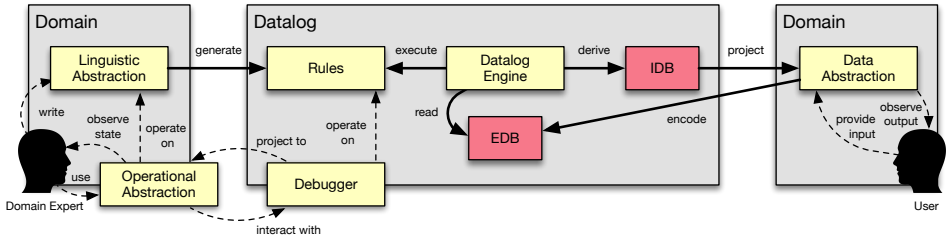


Figure 1.4: Solution to closing the representational gap between the problem domain and Datalog.

1.3 Closing the Representational Gap of Datalog

This dissertation validates the thesis by proposing abstractions to close all three representational gaps. Figure 1.4 highlights how to close the representational gaps between the problem domain and Datalog. A key insight is to view Datalog as an intermediate representation language instead of a user-facing programming language. We propose three types of abstractions to close the representation gaps: *linguistic abstractions*, *data abstractions*, and *operational abstractions*. We highlight how to use each abstraction to close the corresponding representational gap. Note that we talk about abstractions in general in this section and will highlight specific abstractions catered to problem domains, in the next section, which introduces the contributions of this dissertation.

Linguistic Abstractions. To close the gap for domain-specific computations (RG1) Computation, we propose to use *linguistic abstractions* for Datalog. Linguistic abstractions allow domain experts to work with language abstractions that cater to their domain. Hence, domain experts do not have to match their mental model of the solution to Datalog's style of solving problems, namely horn clauses. We still can utilize the strengths of Datalog, such as its declarative least fixpoint semantics, or its incremental engines whenever possible. We aim to develop compilers that transform linguistic abstractions to Datalog and perform optimizations, allowing domain experts to focus on expressing a domain solution instead of solving technical difficulties related to Datalog, such as its incremental execution.

Data Abstractions. Domain solutions operate on domain-specific data. To close the representational gap between domain-specific data and Datalog relations (RG2) Data, we propose *data abstractions*. Data abstractions shield users from interacting with Datalog's style of representing data, which is a collection of relations enumerating tuples. Specifically, providing inputs as relations (EDB) and observing outputs as relations (IDB). Additionally, data abstractions shield users from invariants that an encoding of domain-specific data requires. For example, when using incremental Datalog engines, small changes in domain-specific data should translate to small changes in the relational encoding. Otherwise, users cannot utilize the strengths of incremental Datalog engines, in particular, its efficient incremental updates.

Operational Abstractions. To close the representational gap between the domain-specific mental model and Datalog's observable behavior (RG3) Observable Behavior, we propose *operational abstractions*. Operational abstractions shield domain experts from debugging techniques used for Datalog, which are provenance-based, hence inspecting derivation trees. Using an abstraction enables domain experts to use domain-specific debugging models instead of the model catered to Datalog. Operational abstractions become even more important when using linguistic abstractions and data abstractions for Datalog. Domain experts interact with an operational abstraction, which translates the interactions of debugging a domain-specific solution to interactions of the Datalog debugger. Then an operational abstraction projects intermediate states of the Datalog debugger and links them to linguistic abstractions.

1.4 Dissertation Contributions

In this section, we highlight the contributions of the dissertation. Each contribution marks an extension to the Datalog framework IncA [Szabó et al. 2016, 2018, 2021]. Figure 1.5 shows the architecture of the IncA framework and highlights each extension providing an abstraction for Datalog. While extending the IncA framework, we also re-implemented the IncA framework and ported IncA from its original implementation in the projectional language workbench MPS [1] to Scala2. We use IncA_{MPS} when we reference the original IncA implementation using MPS, and $\text{IncA}_{\text{Scala}}$ when referencing the re-implementation in Scala2 in the remainder of this dissertation. In the remainder of this section, we highlight the contributions of this dissertation and group them according to the abstraction they provide for Datalog.

Linguistic Abstraction. We provide linguistic abstractions by developing different frontends for Datalog for specific domains. First, we develop a type system DSL that compiles to Datalog using an accompanying optimizing compiler (**Number 1**). The goal of the optimizing compiler is to enable efficient incremental updates. First, we transform the type system specification such that it is expressible as a valid Datalog program. Then, we apply two optimizations to enable efficient incremental execution. That is, the optimizations follow the invariant that small changes in the input program should result in small changes in the derived typing information if possible. This invariant avoids deleting and re-deriving typing tuples during an incremental run whenever the input program changes. Hence, we show how to systematically derive incremental type checkers by using an incremental Datalog engine. While the conceptual gap between type system inference rules and Datalog rules is small, the optimizations of the accompanying compiler enable type system experts to avoid thinking about incremental Datalog and how to keep the impact of program changes as low as possible.

Further, we develop a general-purpose functional frontend supporting fixpoint computations that compiles to Datalog (**Number 3**). We compile to Datalog to utilize Datalog's least fixpoint semantics while using general-purpose features such as functional abstractions, algebraic data types, first-class functions, and parametric polymorphism. Hence, the

[1] `https://www.jetbrains.com/mps`

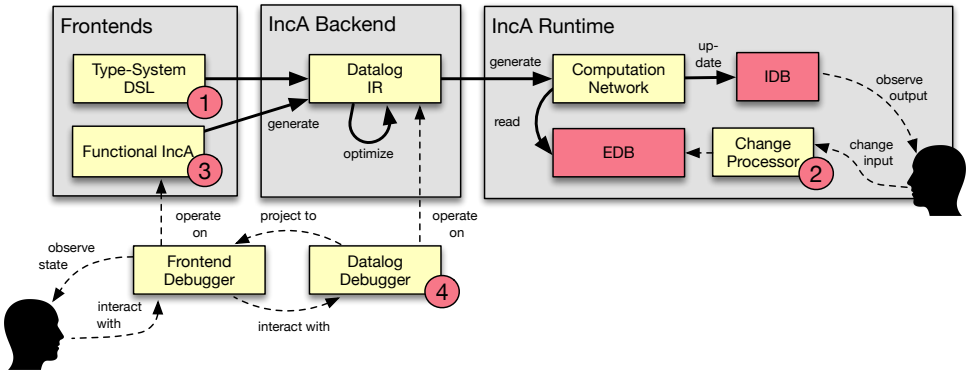


Figure 1.5: The architecture of the IncA framework highlighting the contributions of this dissertation.

functional Datalog frontend allows abstractions such as higher-order relations, parametric relations, and first-class relations, which is not possible with standard Datalog. The conceptual gap between the functional frontend and Datalog is larger than for the type system DSL, which affects the complexity of the compiler.

Data Abstraction. Next, we develop a data abstraction for structured data such as trees and lists. We encode structured data relationally to use them as input for Datalog programs (**Number 2**). While encoding such data is rather trivial, the focus of the encoding is to encode structured data such that it enables efficient incremental updates. To this end, we identify three design principles that guide the relational encoding of structured data. These design principles enable us to translate small changes in structured data into small changes in the relational encoding. Additionally, we develop efficient indexing structures for the relational encoding that utilizes invariants of the structured data layout. Thus, the relational encoding enables efficient incremental updates when using incremental Datalog engines.

Operational Abstraction. At last, we propose a new view of debugging Datalog programs (**Number 4**). That is, instead of inspecting derivation trees for derived tuples, we allow debugging Datalog queries by inspecting its top-down execution trace. This view enables programmers to debug partial queries and queries that derive an empty answer, which is not possible using the state-of-the-art debugging approach for Datalog. A top-down stepping Datalog debugger provides an operational abstraction for the functional Datalog frontend. The operational abstraction operates on the compiled Datalog program but projects all valid intermediate states of the execution trace back to the functional Datalog frontend program. Hence, programmers using the functional Datalog frontend interact with a debugger operating on the abstraction level of functional computations, even though the debugger operates on Datalog programs internally.

1.5 Dissertation Outline

This section outlines the remainder of this dissertation. All contributions of this dissertation have been published at peer-reviewed conferences and are the work of the author in collaboration with others. At the beginning of each chapter, we highlight the scientific work the chapter is based on and list all collaborators. This dissertation contains the following chapters:

- Chapter 2 presents a type system DSL and its accompanying optimizing compiler.
- Chapter 3 presents a relational encoding of structured data that enables efficient incremental updates while using incremental Datalog engines.
- Chapter 4 presents a functional Datalog frontend with implicit fixpoint computations and its translation to core Datalog.
- Chapter 5 presents a top-down Datalog debugger and its projection to the functional Datalog frontend.
- Chapter 6 compares the contributions of this dissertation with related work.
- Chapter 7 concludes the dissertation and discusses future research directions.

2

A Type System DSL for Incremental Datalog

This chapter is based on the peer-reviewed OOPSLA'20 paper "A Systematic Approach to Deriving Incremental Type Checkers" [Pacak et al. 2020] and is joint work with Sebastian Erdweg and Tamás Szabó.

Abstract — Static typing can guide programmers if feedback is immediate. Therefore, all major IDEs incrementalize type checking in some way. However, prior approaches to incremental type checking are often specialized and hard to transfer to new type systems. In this chapter, we propose a systematic approach for deriving incremental type checkers from textbook-style type system specifications. Our approach is based on compiling inference rules to Datalog, a carefully limited logic programming language for which incremental engines exist. The key contribution of this chapter is to discover an encoding of the infinite typing relation as a finite Datalog relation in a way that yields efficient incremental updates. We implemented the compiler as part of a type system DSL. This type system DSL acts as a linguistic abstraction for incremental Datalog. We show that the type system DSL and its accompanying compiler supports simple types, some local type inference, operator overloading, universal types, and iso-recursive types.

2.1 Introduction

Many programming languages employ a static type system to check user-defined invariants at compile time. Indeed, programmers of statically typed languages often rely on feedback from the type checker for guidance. Unfortunately, type checking can take significant time for larger programs and can interrupt the programmer's development flow. Therefore, it is hardly surprising that virtually all major IDEs incrementalize type checking in some way. Unfortunately, most of these solutions are highly specialized and generally hard to transfer to a new type system. We lack a principled solution for incrementalizing type checkers.

This chapter presents a systematic approach for *deriving* incremental type checkers from textbook-style type system specifications. While we hope to incrementalize advanced type systems eventually, in this chapter we focus on building a sound foundation for the most elemental type system features: name binding and type errors. These features constitute fundamental challenges for incremental type checking, and we believe it is essential to develop principled solutions to such fundamental challenges first. Nonetheless, our approach already supports a number of sophisticated type systems.

Our approach is based on the idea of compiling inference rules to the logic programming language Datalog. Targeting Datalog is promising because efficient incremental Datalog engines already exist such as ViatraQuery [Ujhelyi et al. 2015] and DDlog [Ryzhyk and Budiu 2019]. However, targeting Datalog is also challenging because Datalog's expressivity is carefully limited. Datalog programs can only compute finite relations, whereas the typing relation usually is an inductively defined infinite relation. Although this makes compiling type checkers to Datalog seemingly impossible, we have discovered a sequence of systematic transformations that make the resulting inference rules expressible in Datalog.

The first transformation utilizes a new property we call *co-functional dependencies*. While a functional dependency describes a uniquely determined output, a co-functional dependency describes a uniquely determined input. In particular, for algorithmic type systems, the typing context and other contextual information is co-functionally dependent on the syntax tree. Our transformation exploits this property to factor out the context from the typing relation, making the typing relation computable in Datalog. Unfortunately, the resulting type system won't admit efficient incrementalization, because even a small change to the typing context will affect large parts of the typing derivation. We discovered that we can eliminate this issue by complete deforestation [Wadler 1990] of all typing contexts. Thus, our second transformation is a specialized deforestation of Datalog programs. Our third and final transformation makes sure ill-typed terms do not unnecessarily prune typing derivations. Otherwise, any code change that fixes a type error would entail significant reanalysis. To this end, we developed a reformulation of type systems that separates error handling from computing a type. Our transformation rewrites any algorithmic type checker into one that collects type errors separately from the typing relation. This transformation may well be useful independent of the rest of our work.

Based on our transformations, we developed a domain-specific language (DSL) for type system descriptions that compiles to Datalog. This DSL acts as a linguistic abstraction for Datalog and closes the representational gap (RG1) Computation. Our compiler implements the three transformations by rewriting inference rules and only lowers those inference rules to Datalog as the last step. Thus, the DSL compiler can be used to simplify type

systems independent of Datalog. We have used the DSL to express a wide range of type system features. In addition to PCF with product and sum types, we modeled bi-directional type checking, operator overloading, universal types in the style of System F, and iso-recursive types. We can confirm that all these features can be compiled to Datalog by our transformations and that the resulting Datalog program is incrementally solvable. Effectively, our DSL derives incremental type checkers from textbook-like type system specifications. We also measured the incremental performance of compiled type systems for synthesized PCF programs. We designed a range of change scenarios to challenge the incremental performance. We found that even when large parts of the program are affected by a change, we still deliver updated typing information in at most several tens of milliseconds.

In summary, we make the following contributions:

- We analyze the challenges associated with compiling type systems to Datalog (Section 2.2).
- We introduce co-functional dependencies and define a Datalog transformation that moves co-functionally dependent data into a separate relation (Section 2.3).
- We show how to eliminate typing context propagation from type systems (Section 2.4).
- We show how to transform a type system to collect type errors on the side (Section 2.5).
- We implement all three transformations in the compiler of a type systems DSL (Section 2.6), demonstrate its applicability (Section 2.7), and benchmark its performance (Section 2.8).

2.2 Why are Type Systems in Datalog Challenging?

This chapter proposes to incrementalize type checkers by translation to Datalog. Our hypothesis is that such translation can be done systematically and is useful: Existing incremental Datalog engines provide efficient incremental running times. In the present section, we illustrate why encoding type checkers in Datalog is challenging in the first place. We highlight the challenges while translating a number of exemplary type systems, all using the following syntax:

$$
\begin{array}{lll}
\text{(program)} & p & ::= \text{main } e \\
\text{(expression)} & e & ::= \mathbf{unit} \mid x \mid \lambda x{:}T.e \mid e\ e \\
\text{(type)} & T & ::= \mathbf{Unit} \mid T \to T \\
\text{(context)} & C & ::= \varepsilon \mid C, x{:}T
\end{array}
$$

Our motivating examples will only differ in their typing relation but reuse the same syntax. For each typing relation, we show how the type rules can be translated to Datalog and discuss if and how a state-of-the-art incremental Datalog engine could handle them. As such, the current section also presents the required Datalog background.

Challenge 1: Expressions We start with the typing relation $(e : T)$ of a very simple type system that only permits unit constants and their application. This type system is not particularly useful but helps us illustrate how to translate a simple type system to Datalog.

$$\text{T-Unit}\frac{}{\textbf{unit} : \textbf{Unit}} \qquad \text{T-App}\frac{e_1 : \textbf{Unit} \quad e_2 : \textbf{Unit}}{e_1\ e_2 : \textbf{Unit}} \qquad \text{T-Main}\frac{e : T}{(\text{main}\ e)\ \textbf{ok}}$$

We can represent the typing relation $(e : T)$ as a binary Datalog relation $\texttt{typed}(e,T)$ and translate each inference rule to a Datalog rule as follows:

$$\begin{aligned}
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?unit}(e),\ \texttt{!Unit}(T).\\
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(e_1,T_1),\ \texttt{?Unit}(T_1),\ \texttt{typed}(e_2,T_2),\ \texttt{?Unit}(T_2),\\
&\qquad\qquad \texttt{!Unit}(T).\\
\texttt{ok}(p) &\texttt{ :- } \texttt{?main}(p,e),\ \texttt{typed}(e,T).
\end{aligned}$$

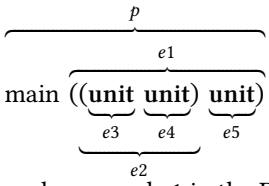
A Datalog program consists of a sequence of rules, each of the form $\texttt{R}(t_1,\ldots,t_n)\texttt{:-}\ \texttt{a}_1,\ldots,\texttt{a}_\texttt{m}$. The rule head $\texttt{R}(t_1,\ldots,t_n)$ declares tuple $(t_1,\ldots,t_n) \in \texttt{R}$ if all atoms $\texttt{a}_1,\ldots,\texttt{a}_\texttt{m}$ in the rule body hold. Terms t are usually logical variables that are shared between the head and body of a rule. Atoms a query relations $\texttt{R}(t_1,\ldots,t_n)$. This way, relations can depend on each other recursively. In this chapter, we use Datalog enriched with algebraic data types in order to model expressions, types, contexts, etc. For each constructor $c(x_1,\ldots,x_k)$ of an algebraic data type, we assume operations $\texttt{!c}(y,x_1,\ldots,x_k)$ and $\texttt{?c}(y,x_1,\ldots,x_k)$ to construct and deconstruct algebraic data x. All Datalog programs have to adhere to a syntactic safety check to be valid: All Datalog rules have to be range restricted. That is, all variables appearing in the head of a Datalog rule need to be positively bound in the body of the rule. For example, Datalog rule $\texttt{P}(X,Y)\texttt{:-}\ \texttt{Q}(Y)$. is not range restricted because X is not bound within the body.

Given this Datalog background, it should be easy to see that $(e,T) \in \texttt{typed}$ if and only if there is a derivation tree for $(e : T)$ according to the inference rules. We hope to incrementalize type checking (i.e., finding a derivation tree) by applying existing incremental Datalog engines to the derived Datalog program. To this effect, it is important to know that incremental Datalog engines evaluate Datalog rules *bottom-up*, inductively enumerating *all derivable tuples*. When the input changes, an incremental Datalog engine updates the relations by retracting those tuples no longer derivable and inserting the newly derivable tuples. Unfortunately, this strategy hinges on the Datalog relations being finite. However, our typing relation is infinite, because our example language contains infinitely many well-typed programs:

$$\texttt{typed} = \{(\textbf{unit},\textbf{Unit}),((\textbf{unit}\ \textbf{unit}),\textbf{Unit}),(((\textbf{unit}\ \textbf{unit})\ \textbf{unit}),\textbf{Unit}),\ldots\}$$

Dealing with an infinite language is a standard problem when using Datalog for static analysis. Even Datalog-based analysis systems without incrementalization such as Doop [Smaragdakis and Bravenboer 2010] require finite relations. Fortunately, there is a standard solution that we can employ here as well: Restrict the relations to only consider the user's current program. That is, rather than defining ?unit, ?app, and ?main inductively over all possible programs, we define them as sets of tuples that exactly reflect the

user program. Since the user program is finite by construction, we can now evaluate `typed` bottom-up, enumerating the well-typed subset of the nodes in `?unit`, `?app`, and `?main`. This strategy has been successfully employed in Datalog-based incremental analyzers before [Szabó et al. 2016], such that no further innovation is required for this first challenge. As an example we show how the program on the left is encoded in Datalog as you can see on the right:



$$\begin{aligned}\texttt{?main} &= \{(p, e1)\}\\ \texttt{?app} &= \{(e1, e2, e5), (e2, e3, e4)\}\\ \texttt{?unit} &= \{e3, e4, e5\}\end{aligned}$$

We use labels such as p and $e1$ in the Datalog encoding to reference nodes of the abstract syntax tree. The set of tuples `?app` contains tuples that represent application nodes. Take for example the tuple $(e1, e2, e5)$ where node $e1$ is an application node which has $e2$ and $e5$ as children. A Datalog engine will derive the following finite relations based on the Datalog rules above and the sets of tuples:

$$\begin{aligned}\texttt{typed} &= \{(e3, \textbf{Unit}), (e4, \textbf{Unit}), (e5, \textbf{Unit}), (e2, \textbf{Unit}), (e1, \textbf{Unit})\}\\ \texttt{ok} &= \{p\}\end{aligned}$$

Challenge 2: Types The previous type system is not very useful because it only inhabits the **Unit** type. We extend this type system by allowing thunks and their application:

$$\text{T-Unit}\ \frac{}{\textbf{unit} : \textbf{Unit}} \qquad \text{T-App}\ \frac{e_1 : \textbf{Unit} \to T \qquad e_2 : \textbf{Unit}}{e_1\ e_2 : T}$$

$$\text{T-Lam}\ \frac{e_1 : T_2}{\lambda x : \textbf{Unit}.\, e_1 : \textbf{Unit} \to T_2} \qquad \text{T-Main}\ \frac{e : T}{(\text{main}\ e)\ \textbf{ok}}$$

Again, we can translate these type rules to Datalog as explained above:

```
typed(e,T) :- ?unit(e), !Unit(T).
typed(e,T) :- ?app(e,e1,e2), typed(e1,Te), ?Fun(Te,T1,T), ?Unit(T1), typed(e2,T2),
              ?Unit(T2).
typed(e,T) :- ?lam(e,x,T1,e1), ?Unit(T1), typed(e1,T2), !Fun(T,T1,T2).
     ok(p) :- ?main(p,e), typed(e,T).
```

Again, we must ask if these rules can be evaluated bottom-up by an incremental Datalog engine. And again this hinges on the Datalog relations being finite. If we assume like above that `?app`, `?lam`, etc. are constants and only enumerate the user's current program, then only finitely many expressions can occur in `typed`. Indeed, expressions are not the problem but types are. While the previous type system only considered a single type **Unit**, this type system associates thunk types of the form $T ::= \textbf{Unit} \mid \textbf{Unit} \to T$ to expressions. Notably, this domain is infinite and relation $\texttt{typed} \subseteq \texttt{e}_{user} \times T$ could contain infinitely many tuples even if $\texttt{e}_{user}$ is finite.

In truth, `typed` will only ever contain finitely many tuples. This is because of the functional dependency $e \rightsquigarrow T$ in `typed`, which means that column T of `typed` is uniquely determined by column e of `typed`. That is, if $(e, T1) \in \texttt{typed}$ and $(e, T2) \in \texttt{typed}$, then

$T1 = T2$ [Watt 2018, Chap. 11]. Our type system satisfies the functional dependency $e \rightsquigarrow T$ because it is algorithmic. Consequently, if `typed` only contains finitely many entries in column e, then `typed` can also only contain finitely many tuples in $e \times T$. Therefore, `typed` is finite and incremental bottom-up evaluation succeeds [Ramakrishnan et al. 1987].

It is noteworthy that many (even non-incremental) Datalog engines would reject the derived Datalog program because it synthesizes data at run time. However, our type system must generate function types $!\texttt{Fun}(T, T_1, T_2)$ of arbitrary size to match the nesting level of lambdas in the user's program. The good news is that a few cutting-edge incremental Datalog solvers like IncA [Szabó et al. 2018] and DDlog [Ryzhyk and Budiu 2019] can handle the derived Datalog code. The bad news is that we must move beyond the cutting edge to support more interesting type systems.

Challenge 3: Contexts. The next challenge arises when introducing typing contexts. To this end, we consider the simply typed lambda calculus:

$$\text{T-Unit}\frac{}{C \vdash \textbf{unit} : \textbf{Unit}} \qquad \text{T-App}\frac{C \vdash e_1 : T_1 \to T \qquad C \vdash e_2 : T_1}{C \vdash e_1\ e_2 : T}$$

$$\text{T-Lam}\frac{C, x : T_1 \vdash b : T_2}{C \vdash \lambda x : T_1 . b : T_1 \to T_2} \qquad \text{T-Var}\frac{C(x) = T}{C \vdash x : T} \qquad \text{T-Main}\frac{\varepsilon \vdash e : T}{(\text{main}\ e)\ \textbf{ok}}$$

The typing relation now is ternary and the inference rules thread the typing context:

```
typed(C,e,T) :- ?unit(e), !Unit(T).
typed(C,e,T) :- ?app(e,e1,e2), typed(C,e1,Te), ?Fun(Te,T1,T), typed(C,e2,T1).
typed(C,e,T) :- ?lam(e,x,T1,b), !bind(C',C,x,T1), typed(C',b,T2), !Fun(T,T1,T2).
typed(C,e,T) :- ?var(e,x), lookup(C,x,T).
      ok(p) :- ?main(p,e), !empty(C), typed(C,e,T).
```

Note that we use `!bind` in the `lam` case to extend the context and `lookup` in the `var` case to extract a binding from the context. The main program is checked in the `!empty` context.

Unfortunately, our derived Datalog program is not computable in bottom-up style anymore and, thus, cannot be incrementalized by existing Datalog engines. To see why, let us inspect the `var` rule in more detail. This rule declares a tuple $(C, e, T) \in \texttt{typed}$ whenever $?\texttt{var}(e, x)$ and $\texttt{lookup}(C, x, T)$ hold. As argued above, we can restrict e to range over the user's program only, so that only finitely many variable symbols have to be considered here. But unlike before, e does no longer uniquely determine T because the type also depends on the context C. Therefore, even a single variable x has infinitely many typing derivations:

$$\begin{aligned}\texttt{typed} = \{\ &(x : \textbf{Unit}, x, \textbf{Unit}), \quad (x : \textbf{Unit} \to \textbf{Unit}, x, \textbf{Unit} \to \textbf{Unit}),\\ &(x : (\textbf{Unit} \to \textbf{Unit}) \to \textbf{Unit}, x, (\textbf{Unit} \to \textbf{Unit}) \to \textbf{Unit}),\ \ldots\}\end{aligned}$$

As we will show in Section 2.3, a different encoding of type systems can solve this problem. Our solution works for algorithmic type systems and is based on the following observations:

1. Algorithmic type systems do not guess substitutions of metavariables, but require metavariables to be positively bound. In particular, when a judgment $\texttt{typed}(C, e, T)$ occurs as a premise, the context C is uniquely determined.
2. Algorithmic type systems are syntax-directed and conduct a fold over the syntax tree. This means that each node in the syntax tree is visited at most once.

Together, these observations entail that each expression is checked under a single, uniquely determined context. We exploit this to factor out the context from relation `typed`, adding a new relation `context`(e,C) that associates contexts to expressions. Both relations are finite now. In particular, each variable x in the syntax tree occurs in a unique context `context`(x,C) and therefore has a unique type `typed`(x,T).

Challenge 4: Context propagation. By factoring out the context from relation `typed`, we obtained a Datalog program that is computable in bottom-up style. Thus, we can apply cutting-edge incremental Datalog solvers like IncA [Szabó et al. 2018] and DDlog [Ryzhyk and Budiu 2019] to it. Unfortunately, this will yield unsatisfactory incremental performance. In general, an incremental algorithm yields good incremental performance if the *size of a change* correlates with the *time it takes to process that change*. Conversely, the update time should be largely independent of the size of the overall input. However, for our derived Datalog code, many small changes in the user program can require a large amount of reanalysis. This problem is due to context propagation.

Consider the following example program, where we use **let** as syntactic sugar:

$$\begin{aligned}&\textbf{let } id \,:\, \textbf{Unit} \to \textbf{Unit} = \lambda x : \textbf{Unit}.\, x\\ &\textbf{in } \lambda y : \textbf{Unit}.\, \lambda z : \textbf{Unit}.\, e_0\end{aligned}$$

Expression e_0 will be checked in a typing context that binds id, y, and z. Now, if the type of id changes in any way, all previously propagated contexts have to be retracted and new contexts have to be propagated. Specifically, the tuples of relation `context`(e,C) become obsolete for all expressions e where id is in scope, even for expressions that do not actually refer to id. For declarations with a wide scope, such as top-level functions, this behavior will incur a significant incremental performance penalty.

The problem is that the derived Datalog code propagates entire contexts rather than individual context bindings, and that it ignores whether a binding is being used. As we will show in Section 2.4, we can systematically transform the Datalog code to solve this problem. To do so, we will generate a new relation `findBinding`(x,e,T) that finds the bound type of variable x occurring within expression e. This relation will walk the syntax tree in the opposite direction of context propagation until a binder for x is found. Since `findBinding` does not require a context, we will be able to drop relation `context` and rewrite the `var` rule as follows:

$$\texttt{typed}(e,T) \texttt{ :- } \texttt{?var}(e,x),\ \texttt{findBinding}(x,e,T).$$

That is, starting at the reference e, we find the bound type of x. With this change, no context propagation will be necessary anymore.

Challenge 5: Ill-typed terms. Static type systems restrict the syntactically well-formed terms and define a language of well-typed programs. A syntactically well-formed term is well-typed if there is a typing derivation for that term. This is a yes or no decision: in or out. For an algorithmic type system, as soon as any rule fails to satisfy a premise, the entire program is known to be ill-typed and typing can stop right there. However, aborting type checking early is unsatisfactory for Datalog-based incrementality and for user feedback.

For Datalog-based incrementality, aborting type checking is unsatisfactory since it prunes tuples from the typing relation unnecessarily. In particular, any typing that transitively depends on an ill-typed term will be dropped from the typing relation `typed`. For a simple example, consider a term using type ascription (e **as** T). If e is ill-typed, e and all its ancestors will be dropped from `typed` because the type rules require subterms to be well-typed. However, notice how the type of (e **as** T) really is independent of the well-typedness of e. We would like to retain $(e,T) \in$ `typed`, which also allows the ancestors to be checked as usual. As a developer makes changes in quick succession, alternating between a well-typed and an ill-typed program, a more stable `typed` relation means faster update times.

The second concern with aborting type checking at the first type error is that this is inconvenient in practice. Both compilers and programming editors usually try to report all type errors in the program. In Section 2.5, we show that the Datalog code can be systematically rewritten to collect all type errors and to avoid pruning the typing relation. To this end, we will generate another relation `errors`(e,err) that associates type errors to expressions. A program p then is only well-typed if $p \in$ `ok` and `errors` $= \emptyset$.

Problem Statement. The goal of this chapter is to translate algorithmic type systems to Datalog to utilize state-of-the-art incremental Datalog engines. The translation should be systematic, applicable to a wide range of type systems, and yield good incremental performance. In this section, we identified the following five challenges:

(C1) Expressions are drawn from an infinite domain.

(C2) Types are drawn from an infinite domain.

(C3) Contexts are drawn from an infinite domain.

(C4) Contexts are threaded through typing derivations.

(C5) Ill-typed subterms abort type checking.

While prior work on Datalog-based static analysis can be used to solve challenges **(C1)** and **(C2)**, the other challenges require novel solutions. We present Datalog transformations that solve challenges **(C3)**–**(C5)** in Sections 2.3–2.5.

Algorithmic Type Systems In this chapter, we assume type systems are given in algorithmic form. Specifically, challenges **(C2)** and **(C3)** require an algorithmic formulation. The word *algorithmic* means that the type system can be trivially translated into a recursive-descent algorithm, but the type system may still be defined using inference-style type rules. Most standard type checkers are naturally algorithmic: Type rules are syntax-directed and metavariables are positively bound. We therefore argue that requiring an algorithmic type system is a modest restriction.

To support our point, note that most textbooks introduce type systems in an algorithmic style, including Harper's *Practical Foundations for Programming Languages* [Harper 2016] and Pierce's *Types and Programming Languages* [Pierce 2002]. Hence, novice language designers learn about type systems by being introduced to their algorithmic formulation. Only complex type system features are sometimes given non-algorithmically and require a

rewriting. However, the same rewriting is necessary for non-incremental implementations of those type checkers [Grewe et al. 2015], meaning we do not actually impose an additional requirement on type system designers.

2.3 Transformation 1: Co-Functional Dependencies

Incremental Datalog engines evaluate Datalog programs bottom-up. In the previous section, we explained why a naive translation of a type system to Datalog does not permit the application of bottom-up Datalog engines (Challenge 3): Since contexts occur as a column in the typing relation `typed`, the typing relation has infinitely many tuples, as we illustrated for the `var` rule. Our solution to Challenge 3 is based on a property of algorithmic type systems that we discovered and named *co-functional dependencies*.

2.3.1 Co-Functional Dependencies

Co-functional dependencies express uniqueness relationships between columns of a relation, similar to functional dependencies. Intuitively, a functional dependency describes unique "outputs" of a relation, whereas a co-functional dependency describes unique "inputs" of a relation. For example, the typing relation of the simply typed lambda calculus ($\texttt{typed} \subseteq C \times e \times T$) has a functional dependency $(C \times e) \rightsquigarrow T$. That is, given C and e, type T is an "output" of typing that is uniquely determined by $C \times e$. Our new observation is that the typing relation also has a co-functional dependency $e \overset{co}{\rightsquigarrow} C$. That is, given e, context C is an "input" of typing that is uniquely determined by e and how `typed` is used. While the treatment of functional dependencies is standard in databases [Watt 2018, Chap. 11] and Datalog [Ramakrishnan et al. 1987], our notion of co-functional dependencies is novel to the best of our knowledge. Unfortunately, co-functional dependencies are harder to detect and utilize, since they depend on how a relation is being used.

The typing context C of the simply typed lambda calculus is an example of a co-functionally dependent column: Each expression is only checked under a single context. We do not know how to detect co-functional dependencies automatically, but instead rely on domain knowledge about algorithmic type systems. In general, all contextual information passed around in an algorithmic type system is uniquely determined for a syntax-tree node. This is because each syntax-tree node is visited at most once per relation (syntax-directedness), and the relevant context information is unique (no guessing of metavariables). We could in principle also allow multiple visits of the same syntax-tree node as long as relevant context information is identical in all visits. For example, this will allow us to support operator overloading with overlapping inference rules (Section 2.7).

In the remainder of this chapter, we assume functional and co-functional dependencies are declared as part of a relation's signature. To this end, we introduce the following notation for signatures:

$$
\begin{array}{lll}
\text{(relation signature)} & \sigma & ::= R : T_1 \times \ldots \times T_n \mid F, G \\
\text{(functional dependencies)} & F & ::= \{\mathcal{P}(\mathbb{N}) \rightsquigarrow \mathbb{N}, \ldots\} \\
\text{(co-functional dependencies)} & G & ::= \{\mathcal{P}(\mathbb{N}) \overset{co}{\rightsquigarrow} \mathbb{N}, \ldots\}
\end{array}
$$

A relation signature $(R : T_1 \times \ldots \times T_n | F, G)$ describes the columns of relation R, its functional dependencies F, and its co-functional dependencies G. Functional and co-functional dependencies are defined based on column indices. For example, we can represent the typing relation of the simply typed lambda calculus by signature $\texttt{typed} : C \times e \times T | \{\{1,2\} \rightsquigarrow 3\}, \{\{2\} \overset{co}{\rightsquigarrow} 1\}$. The functional dependency declares that columns 1 and 2 together uniquely determine column 3, that is, $C \times e \rightsquigarrow T$. The co-functional dependency declares that column 2 also uniquely determines column 1, that is, $e \overset{co}{\rightsquigarrow} C$. Datalog relations annotated this way enable us to utilize co-functional dependencies.

Notation. We frequently need to denote sequences and subsequences in this chapter. We write $\overline{x}$ or $x_1, \ldots, x_n$ for a sequence of x elements. Given a set of indices I, we write x_I for the subsequence of $\overline{x}$ consisting of $\{x_i \mid i \in I\}$ and ordered by their index. We leniently write $\overline{x}, y$ and x_I, y and x_I, x_J to concatenate sequences and sequence elements.

2.3.2 Utilizing Co-Functional Dependencies

A co-functional dependency $\overline{c} \overset{co}{\rightsquigarrow} c$ in relation R stipulates that column c of R is uniquely determined by some other columns $\overline{c}$ of R. This allows us to factor out c from R, since we can always use the other columns $\overline{c}$ to uniquely obtain c. However, the rules to obtain c from $\overline{c}$ are not obvious and depend on how R is being queried. This makes co-functional dependencies difficult to utilize.

We have developed a transformation of Datalog code that factors out co-functionally dependent columns c from their relation R. The key idea is to derive an auxiliary relation $\pi_R : \overline{c} \times c$ that has a (regular) functional dependency $\overline{c} \rightsquigarrow c$. Essentially, π_R witnesses the contextual uniqueness of c by mapping $\overline{c}$ to c locally. We then rewrite R to drop column c and to query π_R instead. Essentially, if $(\texttt{R}(\overline{x}, \overline{c}, c) \texttt{:-} a)$ is a rule of R, then $(\texttt{R}'(\overline{x}, \overline{c}) \texttt{:-} \pi_\texttt{R}(\overline{c}, c), a)$ will be a rule of the rewritten R'.

Before delving into the technical details of the transformation, let us consider its application to the simply typed lambda calculus whose Datalog rules we showed in Section 2.2. Since $\texttt{typed} : C \times e \times T$ has $e \overset{co}{\rightsquigarrow} C$, we derive the auxiliary relation $\pi_\texttt{typed} : e \times C$ and use it in `typed`:

$$
\begin{aligned}
\texttt{typed}(e,T) \texttt{ :- } & \pi_\texttt{typed}(e,C),\ \texttt{?unit}(e),\ \texttt{!Unit}(T).\\
\texttt{typed}(e,T) \texttt{ :- } & \pi_\texttt{typed}(e,C),\ \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(e_1,T_e),\ \texttt{?Fun}(T_e,T_1,T),\ \texttt{typed}(e_2,T_1).\\
\texttt{typed}(e,T) \texttt{ :- } & \pi_\texttt{typed}(e,C),\ \texttt{?lam}(e,x,T_1,b),\ \texttt{!bind}(C',C,x,T_1),\ \texttt{typed}(b,T_2),\\
& \texttt{!Fun}(T,T_1,T_2).\\
\texttt{typed}(e,T) \texttt{ :- } & \pi_\texttt{typed}(e,C),\ \texttt{?var}(e,x),\ \texttt{lookup}(C,x,T).\\
\texttt{ok}(p) \texttt{ :- } & \texttt{?main}(p,e),\ \texttt{!empty}(C),\ \texttt{typed}(e,T).
\end{aligned}
$$

Note how we dropped column C from all rule heads and usages of `typed`. Instead, we introduced the query $\pi_\texttt{typed}(e,C)$ at the beginning of each `typed` rule to bind C. All range-restricted rules remain range-restricted after the rewriting because $\pi_\texttt{typed}(e,C)$ positively binds e and C.

For the derived relation $\pi_\texttt{typed} : e \times C$, we generate one rule for each call of `typed`. Thus, there is no rule for `unit` because its rule does not call `typed`, but there are two rules for `app`. The derived rules reflect how the co-functionally dependent input C was constrained.

Essentially, for each call of $\texttt{typed}(C,e,T)$ we copy the surrounding rule and replace the head with $\pi_{\texttt{typed}}(e,C)$:

$$
\begin{aligned}
\pi_{\texttt{typed}}(e_1,C) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(e_1,T_e),\ \texttt{?Fun}(T_e,T_1,T),\ \texttt{typed}(e_2,T_1),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(e_2,C) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(e_1,T_e),\ \texttt{?Fun}(T_e,T_1,T),\ \texttt{typed}(e_2,T_1),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(b,C') &\texttt{ :- } \texttt{?lam}(e,x,T_1,b),\ \texttt{!bind}(C',C,x,T_1),\ \texttt{typed}(b,T_2),\ \texttt{!Fun}(T,T_1,T_2),\\
&\qquad \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(e,C) &\texttt{ :- } \texttt{?main}(p,e),\ \texttt{!empty}(C),\ \texttt{typed}(e,T).
\end{aligned}
$$

Note how the derived relation finds the co-functional column C of an expression e_1 by querying itself recursively for parent node of e_1. This way, the derived relation retraces the original context propagation. However, this initial version of $\pi_{\texttt{typed}}$ only yields a context for e if e is in `typed`, even though this does not influence which context is returned. To break this dependency, we drop all atoms from $\pi_{\texttt{typed}}$ that do not contribute to determining the co-functional column. We can also simplify `typed`, but we may only remove atoms that are infallible (`!bind`, `!empty`, $\pi_{\texttt{typed}}$) to preserve ill-typed terms. This yields the following minimal rule set with a clear division of labor: $\pi_{\texttt{typed}}$ propagates and extends the context, whereas `typed` does the checking and only mentions the context in the `var` rule.

$$
\begin{aligned}
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?unit}(e),\ \texttt{!Unit}(T).\\
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(e_1,T_e),\ \texttt{?Fun}(T_e,T_1,T),\ \texttt{typed}(e_2,T_1).\\
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?lam}(e,x,T_1,b),\ \texttt{typed}(b,T_2),\ \texttt{!Fun}(T,T_1,T_2).\\
\texttt{typed}(e,T) &\texttt{ :- } \pi_{\texttt{typed}}(e,C),\ \texttt{?var}(e,x),\ \texttt{lookup}(C,x,T).\\
\texttt{ok}(p) &\texttt{ :- } \texttt{?main}(p,e),\ \texttt{typed}(e,T).
\end{aligned}
$$

$$
\begin{aligned}
\pi_{\texttt{typed}}(e_1,C) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(e_2,C) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(b,C') &\texttt{ :- } \texttt{?lam}(e,x,T_1,b),\ \texttt{!bind}(C',C,x,T_1),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(e,C) &\texttt{ :- } \texttt{?main}(p,e),\ \texttt{!empty}(C).
\end{aligned}
$$

It is easy to show by induction that $\pi_{\texttt{typed}}$ satisfies the functional dependency $e \rightsquigarrow C$. As we explained for Challenge 2 in Section 2.2, this is sufficient to ensure the finiteness of $\pi_{\texttt{typed}}$. Hence, incremental Datalog engines can apply their bottom-up evaluation strategy to this Datalog program.

2.3.3 Formalizing Transformation *CoFunTrans*

We formalize the transformation *CoFunTrans* that we described informally above. The transformation takes a Datalog program as input and rewrites it to utilize co-functional dependencies. The transformation operates in two steps. First, we revise the signatures of existing relations and add the signatures of derived relations π_R. Second, we revise the rules of existing relations and add new rules for derived relations π_R.

***CoFunTrans* signatures.** Let Σ be the set of relational signatures of the input Datalog program. Then the rewritten Datalog program has signatures *CoFunSigs*(Σ) defined as follows:

$$\text{CoFun-UpdateSig}\ \frac{(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma \qquad codepCols = \{i \mid (I \overset{co}{\rightsquigarrow} i) \in G\} \\ J = \{1, \ldots, n\} \setminus codepCols \qquad F' = deleteShift(F, codepCols)}{(R : T_J | F', \emptyset) \in CoFunSigs(\Sigma)}$$

$$\text{CoFun-DeriveSig}\ \frac{(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma \qquad (I \overset{co}{\rightsquigarrow} i) \in G \\ f = \{1, \ldots, |I|\} \rightsquigarrow |I| + 1}{(\pi_{R,i} : T_I \times T_i | \{f\}, \emptyset) \in CoFunSigs(\Sigma)}$$

Rule CoFun-UpdateSig updates the signatures of existing relations R by dropping all columns *codepCols* that are co-functionally dependent. The updated signature of R only has columns J of types T_J left. The functional dependencies F are updated accordingly and the co-functional dependencies G are dropped entirely. In particular, the helper function $deleteShift(F, codepCols)$ deletes *codepCols* from the functional dependencies in F and shifts the remaining indices to skip dropped columns. For example, $(\texttt{typed} : C \times e \times T | \{\{1,2\} \rightsquigarrow 3\}, \{\{2\} \overset{co}{\rightsquigarrow} 1\})$ becomes $(\texttt{typed} : e \times T | \{\{1\} \rightsquigarrow 2\}, \emptyset)$ after dropping column C.

Rule CoFun-DeriveSig generates a separate signature $\pi_{R,i}$ for each co-functional dependency $I \overset{co}{\rightsquigarrow} i$ of a relation R. Relation $\pi_{R,i}$ maps columns I of types T_I to column i of type T_i, as expressed by its functional dependency f.

***CoFunTrans* rules.** Let Σ be the set of relational signatures of the input Datalog program and let P be the set of rules of the input Datalog program. Then the rewritten Datalog program has rules $CoFunRules(\Sigma, \mathrm{P})$ defined by the following inference rules:

$$\text{CoFun-DropCodepArgs}\ \frac{(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma \qquad codepCols = \{i \mid (I \overset{co}{\rightsquigarrow} i) \in G\} \\ J = \{1, \ldots, n\} \setminus codepCols}{\lfloor \mathtt{R}(\overline{x}) \rfloor = \mathtt{R}(x_J)}$$

$$\text{CoFun-UpdateRule}\ \frac{(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma \qquad (\mathtt{R}(\overline{x}) \texttt{:-} a_1, \ldots, a_m.) \in \mathrm{P} \\ \Pi = \{\pi_{\mathtt{R},\mathtt{i}}(x_I, x_i) \mid (I \overset{co}{\rightsquigarrow} i) \in G\}}{(\lfloor \mathtt{R}(\overline{x}) \rfloor \texttt{:-} \Pi, \lfloor a_1 \rfloor, \ldots, \lfloor a_m \rfloor.) \in CoFunRules(\Sigma, \mathrm{P})}$$

$$\text{CoFun-DeriveRule}\ \frac{(R : T_1 \times \ldots \times T_n | F_R, G_R) \in \Sigma \qquad (I \overset{co}{\rightsquigarrow} i) \in G_R \\ (\mathtt{Q}(\overline{y}) \texttt{:-} a_1, \ldots, a_k, \mathtt{R}(\overline{x}), a_{k+2}, \ldots, a_l.) \in \mathrm{P} \qquad (Q : U_1 \times \ldots \times U_m | F_Q, G_Q) \in \Sigma \\ A = \{\lfloor a_1 \rfloor, \ldots, \lfloor a_k \rfloor, \lfloor a_{k+2} \rfloor, \ldots, \lfloor a_l \rfloor\} \qquad \Pi = \{\pi_{\mathtt{Q},\mathtt{j}}(y_J, y_j) \mid (J \overset{co}{\rightsquigarrow} j) \in G_Q\}}{(\pi_{\mathtt{R},\mathtt{i}}(x_I, x_i) \texttt{:-} \mathbf{slice}_{x_i}(A \cup \Pi).) \in CoFunRules(\Sigma, \mathrm{P})}$$

Rule CoFun-DropCodepArgs defines an auxiliary function $\lfloor a \rfloor$ on atoms that removes co-functionally dependent arguments from calls of R. We use this function in the other two rules. Rule CoFun-UpdateRule updates the Datalog rules of existing relations R by making three changes. First, we remove co-functionally dependent columns from the rule head. Second, we insert queries against the newly derived relations $\pi_{R,i}$ into the body of the rule for each co-functionally dependent column i. Third, we remove co-functionally dependent arguments from calls to other relations in the rule body.

Rule CoFun-DeriveRule generates Datalog rules for the new relations $\pi_{R,i}$. Specifically, we generate one Datalog rule for each call of R and each co-functional dependency $I \overset{co}{\rightsquigarrow} i$ of relation R. Suppose the call of R occurs in a rule of Q. We derive the new rule by changing the rule of Q in three ways. First, we exchange the rule head since we are only interested in learning how x_I determines x_i. Second, we adapt the rule body just like CoFun-UpdateRule did: remove co-functionally dependent arguments and insert queries $\pi_{Q,j}$. This yields the body. Third, we slice the resulting $(A \cup \Pi)$ to only retain those that contribute to x_i.

The resulting Datalog program witnesses co-functional dependencies through the derived relations $\pi_{R,i}$. Note that the transformation only preserves the semantics of the main relation (e.g. `ok`), but not the semantics of individual Datalog relations. This is intended as we wanted to restrict `typed` to become finite. Importantly, we only remove unnecessary tuples from `typed` such that the main relation is preserved: $p \in \texttt{ok}$ if and only if $p \in \mathit{CoFunTrans}(\texttt{ok})$.

Correctness. We formulate precisely under which conditions the transformation is correct and preserves the semantics of the transformed type system. Note that we have not worked out a formal proof of correctness, but rather present the key invariants and properties that ensure correctness.

The formalization assumes a denotational semantics $[\![\mathrm{P}]\!]_{Base}$ for Datalog programs P [Alvarez-Picallo et al. 2019]. The denotational semantics takes a set of base facts *Base* as input (the extensional database in Datalog lingo) and computes the set of facts *Deriv* derived by P from *Base* (yielding the intensional database). Both *Base* and *Deriv* consist of ground tuples $R(C,\ldots,C)$, where R is a relation name and C are constants. In our work, the base facts describe the user program as we illustrated in Section 2.2.

Based on the denotational semantics we define the correctness theorem of *CoFunTrans*. Transformation *CoFunTrans* is correct because it leads to less derivable tuples, but all relevant tuples are still derivable. Hence, every program that is typeable with the original type system is typeable with the transformed type system. Without loss of generality, we assume there is a relation *Main* used as the entry point of type checking.

Theorem 1 (Correctness of *CoFunTrans*)**.** Let Σ be the set of relational signatures of the input Datalog program, let P be the set of rules of the input Datalog program, and let *Main* be the entry point of the type system. For all base relations *Base*, $\mathit{Main}(\bar{x}) \in [\![\mathrm{P}]\!]_{Base}$ if and only if $\mathit{Main}(\bar{x}) \in [\![\mathit{CoFunRules}(\Sigma,\mathrm{P})]\!]_{Base}$.

To prove this theorem, we need to ensure that the derived π relations correctly capture co-functional dependencies $I \overset{co}{\rightsquigarrow} i$: It uniquely maps x_I to x_i. That is, we need to ensure that if the original relation R contains a tuple, the derived π relation contains a tuple representing a projection of the tuple according to the co-functional dependency. In addition, we need to ensure that the derived π relations actually describe functional dependencies to guarantee that Datalog bottom-up evaluation succeeds for the transformed Datalog program.

Lemma 1 (Derived π is correct)**.** Let $(R : T_1 \times \ldots \times T_n \mid F, G) \in \Sigma$ with $(I \overset{co}{\rightsquigarrow} i) \in G$. For all base relations *Base*, if $R(\bar{x}) \in [\![\mathrm{P}]\!]_{Base}$, then the following propositions hold:

1. $\pi_{R,i}(x_I, x_i) \in [\![\mathit{CoFunRules}(\Sigma,\mathrm{P})]\!]_{Base}$.
2. For all $x' \neq x_i$, $\pi_{R,i}(x_I, x') \notin [\![\mathit{CoFunRules}(\Sigma,\mathrm{P})]\!]_{Base}$.

Lemma 1 is the key property of *CoFunTrans* required to prove correctness.

2.4 Transformation 2: Context Fusion

Transformation *CoFunTrans* from the previous section makes a Datalog-encoded type system amenable to bottom-up evaluation. It does so by eliminating co-functional dependencies in favor of functional dependencies. For a type system, this means that class tables, typing contexts, and other contextual information is uniquely associated with each expression. While this enabled bottom-up evaluation, it also introduced a new problem: Even a slight change to contextual information will affect all expressions. This is the problem of context propagation we introduced as Challenge 4.

Context propagation is problematic whenever the context consists of compound information (e.g., a typing context). When parts of the context are changed (e.g., the type of some variable), the entire context will regarded as changed. This is because incremental Datalog engines only trace dependencies between relations and propagate inserted and deleted tuples, but they cannot trace changes to individual components of those tuples. Therefore, when the type of a variable changes, all typing contexts that contain that binding change, and thus all tuples that associate these contexts to expressions need updating.

In this section, we present a Datalog transformation that eliminates intermediate compound data. Specifically, we eliminate context information represented as immutable maps, which is produced by the `!empty` and the `!bind` constructors and consumed with `lookup`. Our rewriting can be regarded as a special case of deforestation [Wadler 1990] for immutable maps but for Datalog programs and with support for recursively defined relations. Note also that immutable maps can encode sets as `Map`$[A, \textbf{Unit}]$ and lists as `Map`$[\texttt{Int}, A]$, such that our rewriting supports many type system specifications. Nonetheless, our primary motivation was the elimination of intermediate typing contexts, which is why we call the transformation "context fusion".

2.4.1 Context Fusion by Example

Consider again the Datalog rules for the simply typed lambda calculus, as produced by transformation *CoFunTrans* from the previous section.

$$\begin{aligned}
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?unit}(e),\ \texttt{!Unit}(T).\\
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(e_1,T_e),\ \texttt{?Fun}(T_e,T_1,T),\ \texttt{typed}(e_2,T_1).\\
\texttt{typed}(e,T) &\texttt{ :- } \texttt{?lam}(e,x,T_1,b),\ \texttt{typed}(b,T_2),\ \texttt{!Fun}(T,T_1,T_2).\\
\texttt{typed}(e,T) &\texttt{ :- } \pi_{\texttt{typed}}(e,C),\ \texttt{?var}(e,x),\ \texttt{lookup}(C,x,T).\\
\texttt{ok}(p) &\texttt{ :- } \texttt{?main}(p,e),\ \texttt{typed}(e,T).
\end{aligned}$$

$$\begin{aligned}
\pi_{\texttt{typed}}(e_1,C) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(e_2,C) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(b,C') &\texttt{ :- } \texttt{?lam}(e,x,T_1,b),\ \texttt{!bind}(C',C,x,T_1),\ \pi_{\texttt{typed}}(e,C).\\
\pi_{\texttt{typed}}(e,C) &\texttt{ :- } \texttt{?main}(p,e),\ \texttt{!empty}(C).
\end{aligned}$$

Our goal is to eliminate the typing context produced by $\pi_{\texttt{typed}}$ and consumed by `lookup` in the `var` rule. Though we consider `lookup` to be a built-in operation, it can be defined in Datalog as follows:

$$\begin{aligned}
\texttt{lookup}(m,k,v) &\texttt{ :- } \texttt{?bind}(m,_,k,v).\\
\texttt{lookup}(m,k,v) &\texttt{ :- } \texttt{?bind}(m,m',k',v'),\ k \neq k',\ \texttt{lookup}(m',k,v).
\end{aligned}$$

The first rule yields value v if map m starts with a binding for key k. The second rule continues lookup in the rest of the map m' if k differs from k'.

To eliminate the context, we want to fuse $\pi_{\texttt{typed}}$ and `lookup`. Specifically, since relation $\pi_{\texttt{typed}}$ has a functional dependency $e \rightsquigarrow C$, it uniquely associates a context to an expression. Thus, instead of performing `lookup` on the context, can't we derive a specialized `lookup` relation that operates on the expression directly? Indeed, this is what our second transformation does.

We derive a specialized `lookup` relation $\varphi_{\texttt{typed}} : e \times v \times T$ that *finds* the binding of a variable v given an expression e. We find bindings by mimicking the rules of $\pi_{\texttt{typed}}$, When a `!bind` occurs in $\pi_{\texttt{typed}}$, we inline the definition of `lookup` to check if we have found the desired entry. For the simply typed lambda calculus we obtain the following rules:

$$\begin{array}{rcl} \ldots & & \ldots \\ \texttt{typed}(e,T) & \texttt{:-} & ?\texttt{var}(e,x),\ \varphi_{\texttt{typed}}(e,x,T). \end{array}$$

$$\begin{array}{rcl} \varphi_{\texttt{typed}}(e_1,k,v) & \texttt{:-} & ?\texttt{app}(e,e_1,e_2),\ \varphi_{\texttt{typed}}(e,k,v). \\ \varphi_{\texttt{typed}}(e_2,k,v) & \texttt{:-} & ?\texttt{app}(e,e_1,e_2),\ \varphi_{\texttt{typed}}(e,k,v). \\ \varphi_{\texttt{typed}}(b,k,v) & \texttt{:-} & ?\texttt{lam}(e,x,T_1,b),\ k = x,\ v = T_1. \\ \varphi_{\texttt{typed}}(b,k,v) & \texttt{:-} & ?\texttt{lam}(e,x,T_1,b),\ k \neq x,\ \varphi_{\texttt{typed}}(e,k,v). \end{array}$$

For applications, $\pi_{\texttt{typed}}$ propagated the context of the parent term e. Hence, $\varphi_{\texttt{typed}}$ continues its search for k in the parent term. For lambdas, $\pi_{\texttt{typed}}$ yielded an extended context $!\texttt{bind}(C', C, x, T_1)$. We inline the definition of `lookup` and hence obtain two $\varphi_{\texttt{typed}}$ rules. First, we yield T_1 if the bound variable x is the entry k we are looking for. Second, we continue searching in the parent term if x and k differ. For the main program, $\pi_{\texttt{typed}}$ yields the empty context $!\texttt{empty}(C)$. Since lookup fails on the empty context, we do not add a rule to $\varphi_{\texttt{typed}}$. Consequently, $\varphi_{\texttt{typed}}$ will fail (as it should) when we reached the root node and have not found a binding.

2.4.2 Formalizing Transformation *CtxFusionTrans*

We formalize the transformation *CtxFusionTrans* that we exemplified above. The transformation takes a Datalog program as input and rewrites it to derive and apply find relations φ_R. We first derive the new signatures and then update and add rules to the Datalog program.

***CtxFusionTrans* signatures.** Let Σ be the set of relational signatures of the input Datalog program. The rewritten Datalog program has signatures $\mathit{CtxFusionSigs}(\Sigma)$ defined as follows:

$$\text{CtxFusion-DeriveSig}\ \frac{\begin{array}{c}(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma \\ (I \rightsquigarrow i) \in F \qquad T_i = \texttt{Map}[K,V] \\ f = \{1,\ldots,|I|+1\} \rightsquigarrow |I|+2\end{array}}{(\varphi_{R,i} : T_I \times K \times V | \{f\}, \emptyset) \in \mathit{CtxFusionSigs}(\Sigma)}$$

$$\text{CtxFusion-RetainSig}\ \frac{(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma}{(R : T_1 \times \ldots \times T_n | F, G) \in \mathit{CtxFusionSigs}(\Sigma)}$$

Rule CtxFusion-DeriveSig generates a signature for the find relations φ_R. We generate a separate find relation for each functional dependency $I \rightsquigarrow i$ where column i has a `Map` type. That is, whenever it is possible to uniquely determine a map from other columns I, we want to find bindings based on I. The find relation uniquely maps values of types T_I together with a key of type K to a value of type V, as expressed by the functional dependency f.

Rule CtxFusion-RetainSig merely retains all existing signatures. Usually, it is possible to drop relations that only produce a map since we won't need them after the transformation. For example, we dropped relation $\pi_{\texttt{typed}}$ in our example from above, using $\varphi_{\texttt{typed}}$ instead. However, our transformation does not account for this simple post-processing.

***CtxFusionTrans* rules.** Let Σ be the set of relational signatures of the input Datalog program and let P be the set of rules of the input Datalog program. Then the rewritten Datalog program has rules $\mathit{CtxFusionRules}(\Sigma, \mathrm{P})$ defined by the following inference rules:

$$\text{CtxFusion-Init}\ \frac{\begin{array}{cc}(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma & (\mathtt{R}(\overline{x}) \texttt{:-} a_1, \ldots, a_m.) \in \mathrm{P} \\ (I \rightsquigarrow i) \in F & T_i = \mathtt{Map}[K, V]\end{array}}{(\varphi_{\mathtt{R},\mathtt{i}}(x_I, k, v) \texttt{:-} a_1, \ldots, a_m, \mathtt{lookup}(x_i, k, v).) \in \mathit{Step}_0(\Sigma, \mathrm{P})}$$

$$\text{CtxFusion-Unfold}\ \frac{\begin{array}{c}(\varphi_{\mathtt{R},\mathtt{i}}(x_I, k, v) \texttt{:-} a_1, \ldots, a_m, \mathtt{lookup}(x_i, k, v).) \in \mathit{Step}_z(\Sigma, \mathrm{P}) \\ \mathtt{!bind}(M, M', k', v') \in \{a_1, \ldots, a_m\} \qquad a_1, \ldots, a_m \vdash x_i = M\end{array}}{(\varphi_{\mathtt{R},\mathtt{i}}(x_I, k, v) \texttt{:-} a_1, \ldots, a_m, k \neq k', \mathtt{lookup}(M', k, v).) \in \mathit{Step}_{z+1}(\Sigma, \mathrm{P})}$$

$$\text{CtxFusion-Bound}\ \frac{\begin{array}{c}(\varphi_{\mathtt{R},\mathtt{i}}(x_I, k, v) \texttt{:-} a_1, \ldots, a_m, \mathtt{lookup}(x_i, k, v).) \in \mathit{Step}_z(\Sigma, \mathrm{P}) \\ \mathtt{!bind}(M, M', k', v') \in \{a_1, \ldots, a_m\} \qquad a_1, \ldots, a_m \vdash x_i = M\end{array}}{(\varphi_{\mathtt{R},\mathtt{i}}(x_I, k, v) \texttt{:-} a_1, \ldots, a_m, k = k', v = v'.) \in \mathit{CtxFusionRules}(\Sigma, \mathrm{P})}$$

$$\text{CtxFusion-Delegate}\ \frac{\begin{array}{cc}(\varphi_{\mathtt{R},\mathtt{i}}(x_I, k, v) \texttt{:-} a_1, \ldots, a_m, \mathtt{lookup}(x_i, k, v).) \in \mathit{Step}_z(\Sigma, \mathrm{P}) & \\ \mathtt{Q}(\overline{y}) \in \{a_1, \ldots, a_m\} & a_1, \ldots, a_m \vdash x_i = y_j \\ (Q : T_1 \times \ldots \times T_n | F, G) \in \Sigma & (J \rightsquigarrow j) \in F\end{array}}{(\varphi_{\mathtt{R},\mathtt{i}}(x_I, k, v) \texttt{:-} a_1, \ldots, a_m, \varphi_{\mathtt{Q},\mathtt{j}}(y_J, k, v).) \in \mathit{CtxFusionRules}(\Sigma, \mathrm{P})}$$

$$\text{CtxFusion-Replace}\ \frac{\begin{array}{cc}(\mathtt{R}(\overline{x}) \texttt{:-} a_1, \ldots, a_i, \mathtt{lookup}(M, k, v), a_{i+2}, \ldots, a_m.) \in \mathrm{P} & \\ \mathtt{Q}(\overline{y}) \in \{a_1, \ldots, a_m\} & a_1, \ldots, a_m \vdash M = y_j \\ (Q : T_1 \times \ldots \times T_n | F, G) \in \Sigma & (J \rightsquigarrow j) \in F\end{array}}{(\mathtt{R}(\overline{x}) \texttt{:-} a_1, \ldots, a_i, \varphi_{\mathtt{Q},\mathtt{j}}(y_J, k, v), a_{i+2}, \ldots, a_m.) \in \mathit{CtxFusionRules}(\Sigma, \mathrm{P})}$$

$$\text{CtxFusion-Retain}\ \frac{\begin{array}{c}(\mathtt{R}(\overline{x}) \texttt{:-} a_1, \ldots, a_m.) \in \mathrm{P} \\ \text{CtxFusion-Replace not applicable}\end{array}}{(\mathtt{R}(\overline{x}) \texttt{:-} a_1, \ldots, a_m.) \in \mathit{CtxFusionRules}(\Sigma, \mathrm{P})}$$

In computing $\mathit{CtxFusionRules}(\Sigma, \mathrm{P})$, we construct intermediate sets $\mathit{Step}_z(\Sigma, \mathrm{P})$ that contain rules after a z-fold unfolding of the `!bind` constructor. Note that the unfolding is bounded by the number of syntactic occurrences of `!bind` in the original rules.

Rule CtxFusion-Init derives the initial $\varphi_{R,i}$ rule for any R that has a functional dependency $I \rightsquigarrow i$ with column i being a `Map`. Given x_I and k, the initial rule uses $a_1, \ldots, a_m$ to uniquely obtain x_i and then perform a `lookup` on that. In the subsequent rules, we try to eliminate the invocation of `lookup` and with it the need for obtaining the map x_i explicitly.

Rules CtxFusion-Unfold and CtxFusion-Bound have the same premises. They check if `lookup` is invoked on an explicitly constructed map. To this end, we check if any of the atoms $a_1, \ldots, a_m$ is an invocation of `!bind` and if the `!bind`-constructed map M is used in `lookup`. We write $a_1, \ldots, a_m \vdash x_i = M$ to mean that x_i and M unify to the same logic variable under $a_1, \ldots, a_m$, which is decidable in Datalog. If so, we know that the `lookup` occurs on top of M. We can thus inline `lookup`. Rule CtxFusion-Unfold captures the case where $k \neq k'$ and `lookup` thus must continue on the rest of the map M'. Since the resulting rule still contains `lookup`, we add the rule to $Step_{z+1}(\Sigma, \mathrm{P})$ to allow further transformation. Rule CtxFusion-Bound captures the case where $k = k'$, so that we can yield $v = v'$. Since CtxFusion-Bound fully eliminated the `lookup` call, we add the resulting rule to the output of the transformation.

Rule CtxFusion-Delegate checks if `lookup` is invoked on a context obtained from another relation. That is the case if any of the atoms $a_1, \ldots, a_m$ is a query $\mathtt{Q}(\overline{y})$ and the map x_i corresponds to y_j for some j. Now, if Q has a functional dependency $J \rightsquigarrow j$ and uniquely determines the map y_j, then we can use the $\varphi_{Q,j}$ relation we created in CtxFusion-DeriveSig for Q. That is, we delegate the search in R and continue searching in Q, which may lead to a (mutually) recursively defined search relation.

Rules CtxFusion-Replace and CtxFusion-Retain propagate the original rules from P. Rule CtxFusion-Replace applies to rules that contain a `lookup` on a map that is uniquely obtained from relation Q. We replace these lookups by the corresponding search $\varphi_{Q,j}$. Rule CtxFusion-Retain copies over all other rules unchanged.

Note that rules can starve within $Step_z$ and never make it to *CtxFusionRules*. This is intended and accounts for the cases where the original `lookup` would have failed as well. In particular, a lookup on an empty map will not result in a *CtxFusionRules* rule.

Correctness. We formulate a correctness theorem for *CtxFusionTrans*. This transformation is an optimization and does not affect the derivable tuples in any way. Hence, every tuple that is derivable by the original type system is derivable by the transformed type system.

Theorem 2 (Correctness of *CtxFusionTrans*)**.** Let Σ be the relational signatures of the input Datalog program, let P be the rules of the input Datalog program, and let $(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma$. For all base relations *Base*, $R(\overline{x}) \in [\![\mathrm{P}]\!]_{Base}$ if and only if $R(\overline{x}) \in [\![\mathit{CtxFusionRules}(\Sigma, \mathrm{P})]\!]_{Base}$.

The only rewriting that changes existing relations is CtxFusion-Replace, which replaces invocations of `lookup` with invocations of φ. To justify this rewriting, we need to ensure the derived φ relations are correct, that is, they resolve key k to the same value v as `lookup`. We formulate the following lemma which ensures we can actually replace `lookup` with the derived φ relations. It states that `lookup` will resolve key k to the same value v within context x_j as the newly derived relation $\varphi_{Q,j}$. Note that Q functionally determines the context x_j based on x_J which in turn will be source arguments of $\varphi_{Q,j}$.

Lemma 2 (Derived φ is correct)**.** Let $(Q : T_1 \times \ldots \times T_n | F, G) \in \Sigma$ with $(J \rightsquigarrow j) \in F$. For all base relations *Base*, if $Q(x_J, x_j) \in [\![\mathrm{P}]\!]_{Base}$ and $\mathtt{lookup}(x_j, k, v) \in [\![\mathrm{P}]\!]_{Base}$, then $\varphi_{Q,\mathrm{j}}(x_J, k, v) \in [\![\mathit{CtxFusionRules}(\Sigma, \mathrm{P})]\!]_{Base}$.

This characterizes under which conditions the correctness of *CtxFusionTrans* is ensured.

2.4.3 Example Revisited

We illustrate the step-wise application of *CtxFusionRules* to the relevant rules of the simply typed lambda calculus.

Input rules:

$\pi_{\texttt{typed}}(e_1, C)$:- $\texttt{?app}(e, e_1, e_2)$, $\pi_{\texttt{typed}}(e, C)$.
$\pi_{\texttt{typed}}(e_2, C)$:- $\texttt{?app}(e, e_1, e_2)$, $\pi_{\texttt{typed}}(e, C)$.
$\pi_{\texttt{typed}}(b, C')$:- $\texttt{?lam}(e, x, T_1, b)$, $\texttt{!bind}(C', C, x, T_1)$, $\pi_{\texttt{typed}}(e, C)$.
$\pi_{\texttt{typed}}(e, C)$:- $\texttt{?main}(p, e)$, $\texttt{!empty}(C)$.
$\texttt{typed}(e, T)$:- $\pi_{\texttt{typed}}(e, C)$, $\texttt{?var}(e, x)$, $\texttt{lookup}(C, x, T)$.

Step$_0(\Sigma, \mathrm{P})$:

$\varphi_{\texttt{typed}}(e_1, k, v)$:- $\texttt{?app}(e, e_1, e_2)$, $\pi_{\texttt{typed}}(e, C)$, $\texttt{lookup}(C, k, v)$.
$\varphi_{\texttt{typed}}(e_2, k, v)$:- $\texttt{?app}(e, e_1, e_2)$, $\pi_{\texttt{typed}}(e, C)$, $\texttt{lookup}(C, k, v)$.
$\varphi_{\texttt{typed}}(b, k, v)$:- $\texttt{?lam}(e, x, T_1, b)$, $\texttt{!bind}(C', C, x, T_1)$, $\pi_{\texttt{typed}}(e, C)$, $\texttt{lookup}(C', k, v)$.
$\varphi_{\texttt{typed}}(e, k, v)$:- $\texttt{?main}(p, e)$, $\texttt{!empty}(C)$, $\texttt{lookup}(C, k, v)$.

Step$_1(\Sigma, \mathrm{P})$:

$\varphi_{\texttt{typed}}(b, k, v)$:- $\texttt{?lam}(e, x, T_1, b)$, $\texttt{!bind}(C', C, x, T_1)$, $\pi_{\texttt{typed}}(e, C)$, $k \neq x$, $\texttt{lookup}(C, k, v)$.

CtxFusionRules(Σ, P):

$\varphi_{\texttt{typed}}(e_1, k, v)$:- $\texttt{?app}(e, e_1, e_2)$, $\pi_{\texttt{typed}}(e, C)$, $\varphi_{\texttt{typed}}(e, k, v)$.
$\varphi_{\texttt{typed}}(e_2, k, v)$:- $\texttt{?app}(e, e_1, e_2)$, $\pi_{\texttt{typed}}(e, C)$, $\varphi_{\texttt{typed}}(e, k, v)$.
$\varphi_{\texttt{typed}}(b, k, v)$:- $\texttt{?lam}(e, x, T_1, b)$, $\texttt{!bind}(C', C, x, T_1)$, $\pi_{\texttt{typed}}(e, C)$, $k = x$, $v = T_1$.
$\varphi_{\texttt{typed}}(b, k, v)$:- $\texttt{?lam}(e, x, T_1, b)$, $\texttt{!bind}(C', C, x, T_1)$, $\pi_{\texttt{typed}}(e, C)$, $k \neq x$, $\varphi_{\texttt{typed}}(e, k, v)$.
$\texttt{typed}(e, T)$:- $\pi_{\texttt{typed}}(e, C)$, $\texttt{?var}(e, x)$, $\varphi_{\texttt{typed}}(e, x, T)$.

A subsequent optimization of the derived rules will remove all invocations of $\pi_{\texttt{typed}}$ and !bind. This is supported by our implementation and will yield exactly those rules shown in Section 2.4.1.

2.4.4 Optimizing Search Relations φ_R

Our transformation *CtxFusionTrans* successfully eliminated all intermediate contexts and introduced a bottom-up find function instead. As we will show in our empirical evaluation, the resulting Datalog code yields far superior incremental performance. However, there is one issue we need to take care of first: The derived find relations φ_R enumerate all referable bindings, not just those required by actual references.

Consider the example term $\lambda x : \textbf{Unit}.(1 + 2) + (3 + 4)$, where we used additions and numeric literals for convenience. Although this program contains no variable references, $\varphi_{\texttt{typed}}$ contains all of the entries shown in the table on the right. That is, $\varphi_{\texttt{typed}}$ contains one entry for each variable and each expression where that variable is in scope. This does not scale very well and it is unneeded.

$\varphi_{\texttt{typed}}$	e	x	T
	1	x	**Unit**
	2	x	**Unit**
	$1+2$	x	**Unit**
	3	x	**Unit**
	4	x	**Unit**
	$3+4$	x	**Unit**
	$(1+2)+(3+4)$	x	**Unit**

Indeed it is sufficient to consider variables that are being referenced in an expression. Fortunately, we can derive an optimized version of $\varphi_{\texttt{typed}}$ by restricting its entries. Specifically, we implemented a simple magic set transformation [Beeri and Ramakrishnan 1991] to derive a helper relation ρ_R that restricts φ_R to those tuples for which a lookup is needed. In particular, when φ_R corresponds to variable lookup, ρ_R corresponds to the free variables of an expression. We restrict φ_R to those tuples that ρ_R considers relevant:

$$\begin{aligned}
\varphi_{\texttt{typed}}(e_1,k,v) &\text{ :- } \rho_{\texttt{typed}}(e_1,k),\ ?\texttt{app}(e,e_1,e_2),\ \varphi_{\texttt{typed}}(e,k,v).\\
\varphi_{\texttt{typed}}(e_2,k,v) &\text{ :- } \rho_{\texttt{typed}}(e_2,k),\ ?\texttt{app}(e,e_1,e_2),\ \varphi_{\texttt{typed}}(e,k,v).\\
\varphi_{\texttt{typed}}(b,k,v) &\text{ :- } \rho_{\texttt{typed}}(b,k),\ ?\texttt{lam}(e,x,T_1,b),\ k=x,\ v=T_1.\\
\varphi_{\texttt{typed}}(b,k,v) &\text{ :- } \rho_{\texttt{typed}}(b,k),\ ?\texttt{lam}(e,x,T_1,b),\ k\neq x,\ \varphi_{\texttt{typed}}(e,k,v).
\end{aligned}$$

$$\begin{aligned}
\rho_{\texttt{typed}}(e,x) &\text{ :- } ?\texttt{var}(e,x).\\
\rho_{\texttt{typed}}(e,k) &\text{ :- } ?\texttt{app}(e,e_1,e_2),\ \rho_{\texttt{typed}}(e_1,k).\\
\rho_{\texttt{typed}}(e,k) &\text{ :- } ?\texttt{app}(e,e_1,e_2),\ \rho_{\texttt{typed}}(e_2,k).\\
\rho_{\texttt{typed}}(e,k) &\text{ :- } ?\texttt{lam}(e,x,T_1,b),\ k\neq x,\ \rho_{\texttt{typed}}(b,k).
\end{aligned}$$

For $\lambda x : \textbf{Unit}.\,(1+2)+(3+4)$, relation $\rho_{\texttt{typed}}$ remains empty since no free variables occur. Consequently, $\varphi_{\texttt{typed}}$ is empty as well. Our implementation supports this optimization.

2.5 Transformation 3: Collecting Errors

The traditional formulation of type systems is focused on deciding if a term is well-typed or ill-typed: There either exists a typing derivation or not. However, applications of type systems need more detailed information, namely the reason(s) a typing derivation could not be constructed. In this section, we propose an alternative formulation of type systems that separates finding a term's type from reporting type errors. This allows us (i) to sometimes find a term's type even though there are type errors and (ii) to report multiple type errors for the same term. We present a Datalog transformation that automatically transforms a traditional type system into one with separate error collection. Our transformation is compatible with the previous two transformations from Sections 2.3 and 2.4, but it does not require them and can be used independently.

2.5.1 Collecting Errors by Example

We illustrate how our transformation works by considering the simply typed lambda calculus again. However, to showcase that our transformation can be used independently from the other two transformations, we start with the original type rules from Section 2.2:

$$\begin{aligned}
\texttt{typed}(C,e,T) &\text{ :- } ?\texttt{unit}(e),\ !\texttt{Unit}(T).\\
\texttt{typed}(C,e,T) &\text{ :- } ?\texttt{app}(e,e_1,e_2),\ \texttt{typed}(C,e_1,T_e),\ ?\texttt{Fun}(T_e,T_1,T),\ \texttt{typed}(C,e_2,T_1).\\
\texttt{typed}(C,e,T) &\text{ :- } ?\texttt{lam}(e,x,T_1,b),\ !\texttt{bind}(C',C,x,T_1),\ \texttt{typed}(C',b,T_2),\ !\texttt{Fun}(T,T_1,T_2).\\
\texttt{typed}(C,e,T) &\text{ :- } ?\texttt{var}(e,x),\ \texttt{lookup}(C,x,T).\\
\texttt{ok}(p) &\text{ :- } ?\texttt{main}(p,e),\ !\texttt{empty}(C),\ \texttt{typed}(C,e,T).
\end{aligned}$$

The construction of a typing derivation fails when premises are unsatisfiable. However, different premises have different purposes and require different error handling. Therefore, we categorize premises as follows:

- **ReportStuck** is the set of relations whose stuckness should result in a type error. We only track the premises occurring in rules of **ReportStuck** relations. For our example, **ReportStuck** = {typed, ok}.
- For every $R \in$ **ReportStuck**, **IgnoreStuck**$_R$ is the set of relations that should be ignored when they occur as premises. We use **IgnoreStuck** for those constraints that merely help select the right type rule. For our example, **IgnoreStuck**$_{\texttt{typed}}$ = {?unit, ?app, ?lam, ?var} and **IgnoreStuck**$_{\texttt{ok}}$ = {?main}.
- Some premises $\texttt{R}(\overline{x})$ are known to be infallible and can be ignored during error handling. In our example, !Unity(T) amongst others will never fail and thus cannot produce a type error.

Based on this categorization, we can systematically derive relations $\varepsilon_{\texttt{typed}} : C \times e \times$ **Error** and $\varepsilon_{\texttt{ok}} : p \times$ **Error** that collect the errors that can occur during type checking:

$$
\begin{aligned}
\varepsilon_{\texttt{typed}}(C,e,err) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \varepsilon_{\texttt{typed}}(C,e_1,err).\\
\varepsilon_{\texttt{typed}}(C,e,err) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(C,e_1,T_e),\ \neg\texttt{?Fun}(T_e,T_1,T),\\
&\qquad err = \textit{“expected Fun type”}\\
\varepsilon_{\texttt{typed}}(C,e,err) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \varepsilon_{\texttt{typed}}(C,e_2,err).\\
\varepsilon_{\texttt{typed}}(C,e,err) &\texttt{ :- } \texttt{?app}(e,e_1,e_2),\ \texttt{typed}(C,e_1,T_e),\ \texttt{?Fun}(T_e,T_1,T),\ \texttt{typed}(C,e_2,T_2),\\
&\qquad T_1 \neq T_2,\ err = \textit{“type mismatch”}.\\
\varepsilon_{\texttt{typed}}(C,e,err) &\texttt{ :- } \texttt{?lam}(e,x,T_1,b),\ \texttt{!bind}(C',C,x,T_1),\ \varepsilon_{\texttt{typed}}(C',b,err).\\
\varepsilon_{\texttt{typed}}(C,e,err) &\texttt{ :- } \texttt{?var}(e,x),\ \neg\texttt{lookup}(C,x,T),\ err = \textit{“lookup failed”}.\\
\varepsilon_{\texttt{ok}}(p,err) &\texttt{ :- } \texttt{?main}(p,e),\ \texttt{!empty}(C),\ \varepsilon_{\texttt{typed}}(C,e,err).
\end{aligned}
$$

There is no rule for unit because its first premise is in **IgnoreStuck**$_{\texttt{typed}}$ and its second premise is infallible. For app we obtain three rules. First, if there are type errors in e_1, we propagate those. We stripped most other premises because they are irrelevant for the recursive call $\varepsilon_{\texttt{typed}}(C,e_1,err)$. Second, if the type T_e of e_1 is not a function type (note the negation ¬ in front of ?Fun), we generate a new error. Third, we propagate the type errors of e_2. A lam cannot introduce a new error and only propagate type errors from the lambda's body. For var we obtain a new type error when the lookup fails (again note the negation ¬).

Note that the collected type errors are not unique; an expression can have multiple errors. For example, both subterms of an app expression can propagate type errors. The derived ε_R relations collect *all* type errors that occur in the program, which was one of our declared goals.

Our other goal was to find a type despite type errors when possible. To this end, we refine what it means for a term to be well-typed in our encoding: A term is well-typed if we can find its type and there is no type error for it. That is, rather than only requiring $(C,e,T) \in$ typed, we additionally require $(C,e,err) \notin \varepsilon_{\texttt{typed}}$ for any err. This allows us to retain tuples in typed even when an expression contains type errors. Our transformation exploits this to relax the rules of typed: Premises that merely perform a check are discarded. For example, our transformation removes the check on an app's argument e_2 and on the body of main. The other rules are unaffected:

```
typed(C,e,T) :- ?unit(e), !Unit(T).
typed(C,e,T) :- ?app(e,e1,e2), typed(C,e1,Te), ?Fun(Te,T1,T).
typed(C,e,T) :- ?lam(e,x,T1,b), !bind(C',C,x,T1), typed(C',b,T2), !Fun(T,T1,T2).
typed(C,e,T) :- ?var(e,x), lookup(C,x,T).
      ok(p) :- ?main(p,e).
```


2.5.2 Formalizing Transformation *CollectErrorsTrans*

We formalize the transformation *CollectErrorsTrans* that we exemplified above. The transformation takes a Datalog program as input and rewrites it to generate relations ε_R and to relax existing relations. The transformation is parametric in the sets **ReportStuck** and $\textbf{IgnoreStuck}_R$ as described above. We first derive the new signatures and then update and add rules to the Datalog program.

***CollectErrorsTrans* signatures.** Let Σ be the set of relational signatures of the input Datalog program. The rewritten Datalog program has signatures *CollectErrorsTrans*(Σ) defined as follows:

$$\text{CollectErrors-DeriveSig}\ \frac{(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma \qquad R \in \textbf{ReportStuck} \qquad J = \{1,\ldots,n\} \setminus \{i \mid (I \rightsquigarrow i) \in F\}}{(\varepsilon_R : T_J \times \textbf{Error} | \emptyset, \emptyset) \in \mathit{CollectErrorsSigs}(\Sigma)}$$

$$\text{CollectErrors-RetainSig}\ \frac{(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma}{(R : T_1 \times \ldots \times T_n | F, G) \in \mathit{CollectErrorsSigs}(\Sigma)}$$

The first transformation rule adds new signatures ε_R for those relations R that are in **ReportStuck**. The new relation has all columns of R except for those that are functionally dependent. For an algorithmic type system this means that the error relation does not track the computed type. In addition to the columns of R, the error relation ε_R has a new column of type **Error**. A tuple $(t_1,\ldots,t_n,err) \in \varepsilon_R$ means that R is stuck for $(t_1,\ldots,t_n)$. The second transformation rule retains the signatures of all existing relations.

***CollectErrorsTrans* rules.** Let Σ be the set of relational signatures of the input Datalog program and let P be the set of rules of the input Datalog program. Then the rewritten Datalog program has rules *CollectErrorsRules*(Σ,P) defined by the inference rules seen in Figure 2.1.

Rule CollectErrors-Propagate generates a Datalog rule that propagates errors from subderivations upwards. Given the rule of a relation $R \in$ **ReportStuck**, if R calls another relation $Q \in$ **ReportStuck**, then we want to forward the errors of Q. Thus, we generate a rule for ε_R that forwards error err obtained from ε_Q. Since we dropped functionally dependent columns from error relations, we select the appropriate variables x_J and y_K to call ε_R and ε_Q respectively. Finally, we copy a slice of the other atoms A to the resulting rule, namely those that contribute to the call of ε_Q. This slicing is important for correctness. For example, consider a type rule for binary addition $e_1 + e_2$. Without slicing, we would get

$$\text{CollectErrors-Propagate}\ \frac{\begin{array}{cc}(\mathtt{R}(\overline{x})\text{:-}\,a_1,\ldots,a_k,\mathtt{Q}(\overline{y}),a_{k+2},\ldots,a_m.)\in \mathrm{P} & A=\{a_1,\ldots,a_k,a_{k+2},\ldots,a_m\}\\ R\in\textbf{ReportStuck} & Q\in\textbf{ReportStuck}\\ (R:T_1\times\ldots\times T_n|F_R,G_R)\in\Sigma & J=\{1,\ldots,n\}\setminus\{i\mid(I\rightsquigarrow i)\in F_R\}\\ (Q:T_1\times\ldots\times T_l|F_Q,G_Q)\in\Sigma & K=\{1,\ldots,l\}\setminus\{i\mid(I\rightsquigarrow i)\in F_Q\}\end{array}}{(\varepsilon_{\mathtt{R}}(x_J,err)\text{:-}\,\textbf{slice}_{y_K}(A),\varepsilon_{\mathtt{Q}}(y_K,err).)\in \mathit{CollectErrorsTrans}(\Sigma,\mathrm{P})}$$

$$\text{CollectErrors-NewError}\ \frac{\begin{array}{cc}(\mathtt{R}(\overline{x})\text{:-}\,a_1,\ldots,a_k,\mathtt{Q}(\overline{y}),a_{k+2},\ldots,a_m.)\in \mathrm{P} & A=\{a_1,\ldots,a_k,a_{k+2},\ldots,a_m\}\\ R\in\textbf{ReportStuck} & Q\notin\textbf{ReportStuck}\\ Q\notin\textbf{IgnoreStuck}_R & \mathtt{Q}(\overline{y})\text{ is fallible}\\ (R:T_1\times\ldots\times T_n|F_R,G_R)\in\Sigma & J=\{1,\ldots,n\}\setminus\{i\mid(I\rightsquigarrow i)\in F_R\}\\ \multicolumn{2}{c}{err=\text{new error describing the reason }\mathtt{Q}(\overline{y})\text{ got stuck}}\end{array}}{(\varepsilon_{\mathtt{R}}(x_J,err)\text{:-}\,\textbf{slice}_{\overline{y}}(A),\neg\mathtt{Q}(\overline{y}).)\in \mathit{CollectErrorsTrans}(\Sigma,\mathrm{P})}$$

$$\text{CollectErrors-RetainSliced}\ \frac{\begin{array}{cc}(\mathtt{R}(\overline{x})\text{:-}\,a_1,\ldots,a_m.)\in \mathrm{P} & R\in\textbf{ReportStuck}\\ \multicolumn{2}{c}{A=\{Q(\overline{y})\mid Q(\overline{y})\in\{a_1,\ldots,a_m\},Q\in\textbf{IgnoreStuck}_R\}}\end{array}}{(\mathtt{R}(\overline{x})\text{:-}\,A,\textbf{slice}_{\overline{x}}(\{a_1,\ldots,a_m\}\setminus A).)\in \mathit{CollectErrorsTrans}(\Sigma,\mathrm{P})}$$

$$\text{CollectErrors-RetainNormal}\ \frac{(\mathtt{R}(\overline{x})\text{:-}\,a_1,\ldots,a_m.)\in \mathrm{P}\qquad R\notin\textbf{ReportStuck}}{(\mathtt{R}(\overline{x})\text{:-}\,a_1,\ldots,a_m.)\in \mathit{CollectErrorsTrans}(\Sigma,\mathrm{P})}$$

Figure 2.1: Set of inference rules defining *CollectErrorsRules*(Σ, P).

the following error rules amongst others:

$$\varepsilon_{\mathtt{typed}}(C,e,err)\text{:-}\,?\mathtt{add}(e,e_1,e_2),\ \varepsilon_{\mathtt{typed}}(C,e_1,err),\ ?\mathtt{Nat}(T_1),\ \mathtt{typed}(C,e_2,T_2),\ ?\mathtt{Nat}(T_2).$$
$$\varepsilon_{\mathtt{typed}}(C,e,err)\text{:-}\,?\mathtt{add}(e,e_1,e_2),\ \mathtt{typed}(C,e_1,T_1),\ ?\mathtt{Nat}(T_1),\ \varepsilon_{\mathtt{typed}}(C,e_2,err),\ ?\mathtt{Nat}(T_2).$$

These rules work fine if one of the operands is ill-typed. But if both operands are ill-typed at the same time, neither rule can fire because of the remaining `typed` constraint on the other operand. Slicing eliminates this problem by discarding those premises that do not help to discover the propagated error.

The second transformation rule CollectErrors-NewError generates error rules for the origin of a stuck premise. If a relation $R \in$ **ReportStuck** calls another relation $Q \notin$ **ReportStuck** that is fallible and should not be ignored, then we derive a corresponding error rule. The derived error rule yields a new error description *err* if $\neg\mathtt{Q}(\overline{y})$, that is, the premise on Q fails. Like in the previous transformation rule, we use slicing to ensure the error rule can fire.

Transformation rule CollectErrors-RetainSliced carries out the relaxation of the original rules for $R \in$ **ReportStuck**. Once again we use slicing, this time to drop premises a_i that do not contribute to discovering the derivable tuples of R . However, the premises A that were ignored by the error rules may never be relaxed. Transformation rule CollectErrors-RetainNormal retains all other Datalog rules unchanged.

Correctness. We formulate a correctness theorem for transformation *CollectErrorsTrans*. This transformation relaxes existing relations and derives error relations to collect stuck derivations instead. That is, all previously derivable tuples are still derivable. In addition, if the original Datalog program will derive a tuple for a relation R marked to report type errors, then there will be no tuple in the corresponding error relation ε_R of the transformed Datalog program.

Theorem 3 (Correctness of *CollectErrorsTrans*)**.** Let Σ be the set of relational signatures of the input Datalog program and P be the set of rules of the input Datalog program. Let $(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma$ and $J = \{1, \ldots, n\} \setminus \{i \mid (I \rightsquigarrow i) \in F\}$. For all base relations *Base*, $R(\overline{x}) \in [\![\mathrm{P}]\!]_{Base}$ if and only if the two following properties hold:

1. $R(\overline{x}) \in [\![\mathit{CollectErrorsRules}(\Sigma, \mathrm{P})]\!]_{Base}$.
2. $R \in$ **ReportStuck** implies for all *err*, $\varepsilon_R(x_J, err) \notin [\![\mathit{CollectErrorsRules}(\Sigma, \mathrm{P})]\!]_{Base}$.

To prove this theorem, we need to ensure the error relation ε_R only contains errors for tuples that were previously underivable.

Lemma 3 (Derived ε is correct)**.** Let $(R : T_1 \times \ldots \times T_n | F, G) \in \Sigma$ and $J = \{1, \ldots, n\} \setminus \{i | (I \rightsquigarrow i) \in F\}$. For all base relations *Base*, if $\varepsilon_R(\overline{x}, err) \in [\![\mathit{CollectErrorsRules}(\Sigma, \mathrm{P})]\!]_{Base}$, then for all x', if $x'_J = \overline{x}$ then $R(x') \notin [\![\mathrm{P}]\!]_{Base}$.

2.5.3 Optimizing Error Propagation

The transformation described above generates rules that propagate errors. In general, this propagation is necessary to ensure we recognize a term as ill-typed when a type error occurs in a subterm. But the propagation of errors also induces a performance overhead: If an error occurs deeply nested in a subterm, that error will be associated with the subterm and all its ancestors. Thus, when the programmer introduces or fixes a type error, the corresponding error propagation takes time.

We found that for many type systems we can eliminate error propagation. If a type system visits all nodes of the syntax tree, an explicit propagation of errors toward the root is unnecessary. Instead, we can refine well-typedness once more and require all subterms to be free of type errors: p is well-typed if and only if $p \in \mathtt{ok}$ and $(e, err) \notin \varepsilon_{\mathtt{ok}}$ for any subterm e of p and any *err*. With this definition it is sufficient to find the sources of errors, but it is not necessary to propagate them. We can easily adapt our transformation by removing rule CollectErrors-Propagate, such that error relations ε_R are only filled according to CollectErrors-NewError. Moreover, the resulting error relations are perfectly suited for programming editors and compilers, which can extract type errors their origin.

2.6 A Type-System DSL compiled to Datalog

We have implemented a domain-specific language (DSL) for describing textbook-like type systems. The DSL poses as a linguistic abstraction for incremental Datalog and closes the representational gap (RG1) Computation. In our DSL, the programmer can declare arbitrary judgments with mixfix syntax and annotate the sorts to describe the co-functional and

functional dependencies of the judgments. The actual annotations in the DSL are simpler than in the chapter, but also slightly less expressive. These judgments can then be used to define type rules.

The screenshot on the right shows part of a type system specification as an example of our DSL. We implemented the DSL as a metalanguage in the projectional language workbench MPS.That is, our DSL can be used to define the type system of other languages defined with MPS. We developed a compiler for our DSL that generates Datalog code using the transformations described in this chapter. We can target any Datalog dialect that allows Datalog rules to synthesize algebraic data (e.g., to produce function types). However, we will only obtain an incremental checker for those Datalog dialects that have an incremental engine. Specifically, we generate code conforming to the Datalog dialect of IncA_{MPS} [Szabó et al. 2016, 2018], an incremental Datalog-based static analysis framework. We could also integrate the type system DSL into the re-implementation of IncA, namely $\text{IncA}_{\text{Scala}}$ or target other Datalog dialects such as DDlog [Ryzhyk and Budiu 2019]. There are a few differences between the transformations in our implementation and the transformations described in the chapter:

```
rule typeOf var
  lookup v in C => T
  C |- v:Var(_name) : T

rule typeOf lam
  Bind(_name, _T1, _C) |- t : T2
  C |- Lam(_name, _T1, _t) : Fun(_T1, _T2)

rule typeOf app
  C |- t1 : Fun(_T1, _T2)
  C |- t2 : T12
  T1 == T12
  C |- App(_t1, _t2) : T2
```

- In our DSL, the premises of type rules are ordered and the judgments declare input-like and output-like columns. We use this information to reason about metavariable bindings. For example, we require metavariable C to be bound before using it in a premise $C \vdash e : T$.
- In this chapter, we introduce the transformations acting on Datalog code, but our compiler actually transforms type system specifications described in our DSL. Only as a last step, the compiler will lower typing rules to the Datalog code. As such, the transformations can be understood to simplify data dependencies within typing rules, which is probably useful for any incrementalization attempt, but maybe even for parallelizing type checking. Additionally, we want to emphasize that the transformations presented in this chapter can be useful when implementing type checkers that do not utilize Datalog engines. The first transformation eliminates the propagation of typing contexts and turns it into a judgment that determines the typing context to consider on demand when a variable reference is encountered. This can enable more fine-grained dependency tracking that can be utilized in other approaches to implement type checkers, including parallelizing it. In combination with the second transformation we avoid the dependencies between intermediate typing context entirely.
- Since we can reason about metavariable bindings in the implementation, slicing becomes easier. Where we used $\textbf{slice}_X(A)$ in the chapter, our implementation can easily decide which atoms A are relevant.
- As usual in the type systems literature, but unlike our Datalog encodings, the conclusions of type rules in our DSL express a few syntactic constraints. Usually, these are used to dispatch the current term to the appropriate type rule. By default, our

implementation of *CollectErrorsTrans* uses the constraints found in the conclusion as **IgnoreStuck**, such that no explicit declaration of **IgnoreStuck** is required.

- In addition to the transformations described in the chapter, our implementation can also handle infinite relations with neither functional nor co-functional dependencies. For such relations, our implementation resolves to generating *non*-incremental Java code that can be invoked from within the Datalog rules. This is reasonable for embedding short yet intractable computations within a larger incremental computation. For example, this extension enabled us to support polymorphic types in our case studies.

The implementation is available open source at https://gitlab.rlp.net/plmz/itypes.

2.7 Case Studies

We conducted case studies to explore the expressivity of our DSL and of the underlying Datalog transformations. Using our DSL, we specified a range of type systems and compiled them to Datalog. In this section, we provide an overview of type system features we successfully encoded and discuss limitations.

Simple types. We encoded PCF, a simply typed lambda calculus with numeric literals, addition, if-zero, and fix. PCF extends our running example and the specification looks much the same. We also used PCF for benchmarking, which we discuss in Section 2.8.

Products and sums. To confirm that the DSL and Datalog transformations can handle types for compound data, we modeled product and sum types. The type rules in our DSL closely follow the rules described in *Types and Programming Languages* [Pierce 2002]. Our compiler translates the extended specification to incrementally executable Datalog code without difficulty. It is reassuring to see that our transformation rules are unchallenged by simple extensions.

Bi-directional type checking. Bi-directional type checking is a form of local type inference. The challenge of bi-directional type checking for our DSL is that there are two mutually recursive typing relations: one for checking and one for inferring types. Our transformations can handle this scenario since we never relied on the recursive structure of the typing relation, and since the underlying Datalog engine can compute mutually recursive Datalog relations. We can thus incrementalize bi-directional type systems.

```
rule infer App
  C |- t1 => Fun(_ty1, _ty2)
  C |- t2 <= ty1
  ------------------------
  C |- App(_t1, _t2) => ty2

rule check Lam
  ty match Fun(_ty1, _ty2)
  Bind(_name, _ty1, _C) |- t <= ty2
  ---------------------------------
  C |- Lam(_name, _t) <= ty
```

Overloading. When we introduced co-functional dependencies, we argued that in an algorithmic type system all contextual information is co-functionally dependent on the syntax-tree node. This is because each syntax-tree node is visited at most once per relation (syntax-directedness), and the relevant context information must not be guessed to avoid backtracking. To explore if the type system necessarily has to be algorithmic, we modeled simple operator overloading. Specifically, we added floating-point numbers to PCF such that there are two type rules for the + operator: one for integers and one for floating-point numbers. This type system is *not* algorithmic, since we have to try out multiple type rules when encountering a + operator. Hence, the question arises whether

the typing context is co-functionally dependent nonetheless, or if our transformations fail. As it turns out, overlapping type rules are not an issue for co-functional dependencies as long as all overlapping rules treat the contextual information uniformly. We believe that this usually is the case: The syntactic form governs the threading of contextual information, not the particular type rule applied.

Universal types. We extended PCF with universal types in the style of System F. The main challenge for incrementality and our transformations is the substitution function on types, that the type system uses to instantiate universal types. As a relation, type substitution takes the form $\texttt{tsubst} : T \times X \times T \times T$ for types T and type variables X. This relation is infinite and there are no co-functional dependencies. Therefore, our transformations cannot make this relation compatible with bottom-up evaluation and thus not incremental. In such cases, our implementation falls back to generating non-incremental Java code that is being invoked from within the incremental Datalog code. This is acceptable when only a small portion of the overall computation becomes non-incremental. For universal types, only type substitution is non-incremental, while tree traversal, variable lookups, type propagation, etc. are fully incremental. However, the size of types usually does not grow proportionally with the size of the AST, such that non-incremental type substitution does not endanger the incremental performance. Technically, our incremental Datalog engine selectively reruns non-incremental operations when any of their arguments changes.

Iso-recursive types. We can also support iso-recursive types as introduced in *Types and Programming Languages* [Pierce 2002]. The typing rules for the fold and unfold expressions depend on type substitution just like universal types. The extension of iso-recursive types does not violate the property of being an algorithmic type system. With this extension and the already supported sum, product and universal types, we can support an expressive functional programming languages type system.

Limitations. We are aware of a few limitations that we want to disclose. First, our DSL currently does not provide support for handling lists, which makes it difficult to encode type system features such as records, variants, or functions with multiple parameters. This is a DSL limitation, not a limitation of our approach of generating incremental Datalog programs. Second, since type substitution is difficult to incrementalize (see universal types), unification is difficult to incrementalize. Therefore, it is not clear if type systems with Hindley-Milner type inference can be supported by our approach. Third, as of right now our approach does not support dependent types. In order to support dependent types our approach needs to be extended such that it can translate operational semantics to Datalog. As a first experiment we were able to encode an interpreter for PCF in the Datalog dialect IncA_{MPS} which enables us to incrementalize operational semantics. Hence, it is possible to describe an type checker supporting dependent types in Datalog. Lastly, we investigated if our approach can support languages with nominal subtyping. Like type substitution, subtyping is an infinite relation without co-functional dependencies, and we must resolve to generating non-incremental Java code. However, nominal subtyping is not self-contained and requires access to the class table of the program in order to decide $C <: D$. This induces additional constraints on when to rerun a subtype check, which we cannot currently trace.

2.8 Performance Evaluation

We present a preliminary performance evaluation of our approach using a type checker for PCF and synthesized subject programs. Our goal is to assess the incremental performance of our approach, and to examine that impact of our transformation steps on the performance. We compare to a non-incremental recursive descent type checker written in Java.

We synthesize two PCF programs *Star* and *Chain* that have intricate dependencies and challenge our incremental approach. *Star* consists of n functions all calling f_0. *Chain* consists of n functions each calling f_{n-1}. These programs allow us to introduce changes with global effect on type checking.

Star:

$$\begin{array}{l}\textbf{let } f_0 = \lambda x : \mathsf{Nat}.\, 1 + x \textbf{ in} \\ \quad \textbf{let } f_1 = \lambda x : \mathsf{Nat}.\, 1 + f_0(x), \\ \qquad \ldots, \\ \qquad f_n = \lambda x : \mathsf{Nat}.\, 1 + f_0(x) \textbf{ in} \\ \qquad\quad 1 + f_0(1)\end{array}$$

Chain:

$$\begin{array}{l}\textbf{let } f_0 = \lambda x : \mathsf{Nat}.\, 1 + x \textbf{ in} \\ \quad \textbf{let } f_1 = \lambda x : \mathsf{Nat}.\, 1 + f_0(x) \textbf{ in} \\ \qquad \ldots \\ \qquad\quad \textbf{let } f_n = \lambda x : \mathsf{Nat}.\, 1 + f_{n-1}(x) \textbf{ in} \\ \qquad\qquad 1 + f_n(1)\end{array}$$

We generate small IDE-style program changes (as opposed to larger commit-style changes) for our evaluation. Our changes are local and only affect a single subterm. We change the program by deleting and inserting tuples into the sets of tuples that describe the program, but only those that are directly affected by the change. The incremental Datalog engine will make sure to propagate those deletions and insertions as to update all derived tuples. To stress-test our approach, we always apply changes to f_0, which all other functions (transitively) depend on. We consider the following 6 kind of changes and their inverse undo changes:

- *Num*: Increment the value of a numeric literal by 1.
- *Ref*: Change a variable reference to an unbound name.
- *Param*: Change the parameter name of a lambda abstraction.
- *Anno*: Change the type annotation of a lambda abstraction.
- *Lambda*: Insert a lambda abstraction in the body of an existing lambda abstraction.
- *AddApp*: Change an addition to an application while retaining the original operands.

Note that, except for *Num*, all of the above changes will result in an ill-typed program. We believe this realistically reflects programming sessions, where a developer changes one piece at a time.

We consider 4 type checker implementations:

- B: baseline type checker, non-incremental, written as a recursive Java function.
- T1: incremental checker, only using our first transformation *CoFunTrans*.
- T1+T2: incremental checker, additionally using our second transformation *CtxFusionTrans*.
- T1+T2+T3: incremental checker, additionally using our third transformation *CollectErrorsTrans*.

Checker	Initial	Num	Ref	Param	Anno	Lambda	AddApp
Star							
B	6.24						
T1	260.48	0.02±0.00	35.11±1.23	34.42±1.11	36.86±1.15	61.16±0.96	34.92±1.36
+T2	314.74	0.07±0.05	32.56±1.46	34.37±1.46	31.83±1.22	53.14±0.66	30.91±1.36
+T3	320.91	0.06±0.00	0.17±0.02	0.10±0.00	43.91±0.57	42.96±0.57	20.98±0.78
Chain							
B	49.31						
T1	176.00	0.06±0.02	85.61±8.58	127.26±4.25	126.99±4.74	44.44±2.00	47.32±2.32
+T2	1016.76	0.02±0.00	82.45±3.80	79.02±3.45	87.60±4.69	87.27±4.42	82.00±3.89
+T3	1040.44	0.02±0.00	0.08±0.00	0.096±0.00	1.46±0.07	1.71±0.08	1.16±0.07

Figure 2.2: Summary of the measurement results. All values are in milliseconds. For average update times, we also show the 95% confidence interval. B stands for the non-incremental baseline type checker. T1 is short for *CoFunTrans*, T2 is *CtxFusionTrans*, and T3 is *CollectErrorsTrans*. Our DSL yields the (T1+T2)+T3 running times.

For the measurements, we synthesize subject programs *Star* and *Chain* with $n = 200$. We apply each change and its undo 40 times after warmup. We measure the initial analysis time and the time it takes to process a change. 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 10.15.4, Java 1.8.0_222, and MPS version 2019.1.6.

Results. Figure 2.2 shows a summary of our measurement results. First, let us discuss the performance of the final transformation stage T1+T2+T3 that our DSL uses and compare it to the non-incremental baseline type checker. We observe that the incremental update times are really fast; they are at most several tens of milliseconds, which is exactly what we expect from a type checker running in an IDE. The initialization time is at most a second, which we consider acceptable, as this is a one-time cost. The run time of the baseline analysis is also fast. This is not surprising because our subject programs are small. However, incrementalization brings significant performance gains most of the time compared to the baseline version. For example, for *Num*, *Ref*, and *Param* changes we see multiple orders of magnitude speedups. We also see slowdowns in certain cases: For the *Star* program the *Anno*, *Lambda*, and *AddApp* changes induce an order of magnitude slowdown. This is due to the global effect these changes have on the program, which our incremental analysis has to retrace. But even in these cases, the incremental running times are still much faster than the initial run of our analysis. In future work, we will try to speed up the initial analysis run, which should improve the performance for changes with global effect.

Let us now examine the effect of our transformations on the performance. For *CtxFusionTrans* (T2), the *Chain* program is interesting because it requires a long threading of typing contexts. When we change the type of f_0 (changes *Param* and *Anno*), all threaded contexts become invalid. However, for changes *Lambda* and *AddApp* we can observe a negative effect of *CtxFusionTrans*. This is because those changes eliminate the binding of f_0 altogether, since its definition becomes ill-typed. Transformation *CollectErrorsTrans* (T3)

recovers these losses. Indeed, *CollectErrorsTrans* (T3) induces a significant speedup most of the time. This is because error collection makes the entire `typeOf` relation more resilient to changes. That is, the type checker can endure ill-typed terms and reuse the tuples in the relation more frequently. As it turns out, this separation of type inference and error collection is key to fast incremental type checking.

Given that incrementalization comes with extensive caching, we also benchmarked the memory overhead of our type checkers. We found that on average the memory overhead is around 10 MB, which is a negligible value compared to the 2 GB memory consumption of the IDE itself.

To summarize, we find that the incremental performance of our type checker is suitable for applications in IDEs. We often achieve order-of-magnitude speedups compared to the non-incremental baseline analysis. We pay the price for this with occasional slowdowns in update times and longer initialization time. The memory overhead of our approach is negligible for the synthesized programs.

2.9 Related Work

In this section, we discuss related work in the context of incremental type checking. We discuss the type system DSL and the accompanying compiler in the context of linguistic abstractions for Datalog in Chapter 6.

Typol [Despeyroux 1984] translates inference rules to Prolog. In contrast to Datalog, Prolog is a Turing-complete language and supports infinite relations that are explored on-demand through top-down evaluation. The same holds for other works such as [Farka et al. 2018; Franceschini et al. 2016] which compile to first-order Horn clauses where infinte relations are allowed. Thus, we face the more difficult challenges of translating inference rules into a style that permits the bottom-up evaluation of logic programs. Attali et al. [1992] implemented an incremental evaluator for Typol programs. However, for fast incremental update times, the context is not allowed to change. If the context changes in any way, type checking has to be started from scratch for the affected expressions. In contrast, we derive a Datalog program that is resilient to such changes.

Wachsmuth et al. [2013] propose a task engine for incremental name and type analysis. Tasks tend to be small and inter-dependent, encoding fine-grained dependencies. When a file changes, they (re)generate tasks for the entire file. Task evaluation relies on a cache of previous task results, only recomputing tasks that are new. If a change affects a task, its cache entry is invalidated and the task reevaluated. The task engine then triggers the reevaluation of all transitively dependent tasks. In contrast to this specialized approach, we rely on a generic incremental compilation target, namely Datalog. The transformations we presented in this chapter enable us to handle type systems based on standard typing rules, whereas the task engine requires language-specific rules for task generation.

Erdweg et al. [2015a] introduce a co-contextual formulation of type checking. Similar to our approach, co-contextual type checking eliminates context propagation. However, while we synthesize a find relation to lookup bindings as needed, co-contextual type checkers propagate lookup constraints when encountering a variable. This makes co-contextual type checkers compositional, allowing subderivations to be reused even when context information changes. The caveat of co-contextual type checking is that incremental

performance heavily relies on constraints being locally solved in the subderivations, which often is not the case [Kuci et al. 2017]. In our solution, we use Datalog's dependency tracking instead of trying to fit dependencies into a compositional structure.

The work on incremental type checking for the programming language B [Meertens 1983] decorates the syntax tree with the type requirements known for a specific node. When changing a node in the syntax tree, the decorated node is deleted which is followed by inserting the newly decorated node while reusing the decorated children of the deleted node. This technique only works because B does not support type declarations for variables but infers the type by discovering type requirements based on the usage of the variable. Hence, the type system of B does not require top-down context propagation, which we support by utilizing *co-functional dependencies*. As our case studies indicate, our approach is applicable to many type systems.

Busi et al. [2019] propose to incrementalize type checking by deriving type rules that utilize memoization. This allows the reuse of parts of the typing derivation when code changes occur. However, as soon as any part of the context or the expression is changed, the entire subderivation has to be reconstructed. Our approach uses much more fine-grained dependency tracking. In particular, our second transformation enables us to track individual bindings rather than entire contexts, which our evaluation confirmed to be essential for incremental type checking.

The transformation we presented in Section 2.3 has a strong resemblance with magic set transformations [Beeri and Ramakrishnan 1991], which are well-known in the Datalog community. Like our transformation, a magic set transformation rewrites a Datalog program to eliminate the derivation of irrelevant (unquerried) tuples. Traditionally, magic set transformations are used as an optimization that may filter some or all irrelevant tuples, and usually the original Datalog program is already computable (has finite relations). In contrast, we start with an incomputable Datalog program (infinite relations). Therefore, we developed a specialized transformation that exploits the new concept of co-functional dependencies. Our specialized transformation allows us to guarantee all irrelevant tuples are eliminated and that the typing relation becomes finite. Additionally, our transformation can exploit co-functional dependencies to avoid the generation of additional auxiliary relations that traditional magic set transformations would require.

Deforestation [Wadler 1990] is a technique to avoid intermediate immutable data structures that are produced and immediately consumed. The *context fusion* transformation of Section 2.4 follows the same idea. Instead of constructing intermediate maps (e.g. typing contexts) and letting the built-in `lookup` relation consume them, we directly perform lookup on the data that functionally determines the map that is passed to `lookup`. The consumer is fixed (`lookup`) in our approach, but every relation that functionally determines a map is viable as a producer. Our technique is applicable to recursive Datalog programs while deforestation is an optimization technique for functional programs.

2.10 Chapter Summary

We proposed a novel approach to systematically deriving incremental type checkers based on textbook-style type rules. We developed a type system DSL that translates to Datalog to utilize incremental Datalog engines. The type system DSL acts as a linguistic abstraction for

incremental Datalog to close the representational gap (RG1) Computation. Our solution is divided into three different transformations. The first transformation utilizes *co-functional dependencies* to translate type rules to Datalog such that bottom-up evaluation succeeds. The second transformation eliminate dependencies on compound data such as typing contexts to achieve more efficient incremental performance. And the third transformation separates the error collection from the type rules, which is interesting even outside of this work. While our transformations primarily tackle issues with incremental name bindings and type errors, we showcased that our transformations effectively support a wide range of type system features such as sum and product types, overloading, universal types, iso-recursive types, and bi-directional type checking. Further, we performed a preliminary performance evaluation to demonstrate that the derived type checkers indeed achieve fast incremental update times.

3

Incremental Processing of Structured Data in Datalog

This chapter is based on the peer-reviewed GPCE'22 paper "Incremental Processing of Structured Data in Datalog" [Pacak et al. 2022] and is joint work with Tamás Szabó and Sebastian Erdweg.

Abstract — Incremental computations react to input changes by updating their outputs. Compared to a non-incremental rerun, incremental computations can provide order-of-magnitude speedups, since often small input changes trigger small output changes. One popular means for implementing incremental computations is to encode the computation in Datalog, for which efficient incremental engines exist. However, Datalog is very restrictive in terms of the data types it can process: Atomic data organized in relations. While structured tree and graph-shaped data can be encoded in relations, a naive encoding inhibits incrementality. in this chapter, we develop a data abstraction for Datalog. In particular, we present an encoding of structured data in Datalog that supports efficient incrementality such that small input changes are expressible. We explain how to efficiently implement and integrate this encoding into an existing incremental Datalog engine, and we show how tree diffing algorithms can be used to change the encoded data.

3.1 Introduction

Incremental computations allow for dramatic performance speedups. Rather than rerunning a computation when its input changes, an incremental computation processes the change and updates its output by reusing intermediate results and only recomputes intermediate results that are effected by the change [Ramalingam and Reps 1993]. This is beneficial in many scenarios, namely when input changes are small (compared to the complete input) and they trigger output changes that are also small (ideally, proportional to the input change) which can lead to significant performance improvements over a complete re-computation. For example, incremental parsers react to the change of individual tokens to update the generated syntax tree [Wagner and Graham 1997], incremental program analyses react to changes of individual nodes of a syntax tree to update the analysis results [Szabó et al. 2016, 2018, 2021], and incremental build systems react to the change of individual files to update the generated build artifacts [Erdweg et al. 2015b; Konat et al. 2018]. All these scenarios exploit the principle of inertia: Computations continue to yield similar outputs when their inputs change over time [Gupta and Mumick 1999; Mitschke et al. 2014].

There are different ways to realize incremental computations; this chapter focuses on incremental computations realized in Datalog. Datalog is a logic programming language that was originally designed to formulate queries in deductive databases [Maier et al. 2018]. However, Datalog has become quite popular in recent years for solving a wide range of problems [Huang et al. 2011], from program analysis, to network monitoring and distributed computing. Unfortunately, most of these usages do not exploit one of Datalog's most unique features: incrementality. Datalog's incremental semantics is known since the 1990s [Gupta et al. 1993], yet only few modern Datalog implementations support incrementality today [Ryzhyk and Budiu 2019; Szabó et al. 2021; Zhao et al. 2021]. And even when incrementality is supported, it is not obvious how users can exploit it.

The problem is that Datalog can only process relations of atomic data, while many computations operate on structured data. To process structured data in Datalog, we must encode the data in flat relations, that is sets of tuples. Unfortunately, existing encodings inhibit efficient incrementality because small changes in the structured data require relatively large changes in the relations. For example, consider we want to implement a control-flow analysis in Datalog for the program in Figure 3.1. The analysis operates on the parsed syntax tree of the program, which we must encode into relations.

Figure 3.2 illustrates a possible encoding of the unchanged syntax tree as flat relations. For example, the relation Statement contains tuples for each statement of all methods in the program. This encoding is similar to the Java bytecode encoding used by Doop, a state-of-the-art analysis platform implemented in Datalog [Bravenboer and Smaragdakis 2009b]. We also show the derived relation `CfgEdge` that represents the control-flow graph (CFG) of the program as computed by our Datalog analysis. However, this encoding is ill-suited for incremental computing. Consider three possible changes of the program:

1. Rename of the class `App`. Since the class name occurs as part of unique IDs for methods and statements throughout the encoding, a renaming of `App` requires almost all tuples to be modified. An incremental Datalog engine must delete all information derived about `App` including those in CfgEdge and then re-derive all that information again.

```
class App { // change 1: rename App to Application
  // change 2: replace void by int
  public void run(int i) {
    int x = i;
    while (x >= 0) {
      // change 3: insert System.out.println(x);
      if (x % 2 == 0) {
        x = x - 2;
      } else {
        x = x - 1;
      }
    }
  }
}
```

Figure 3.1: Simple Java program and 3 incremental changes.

2. Change the return type of method `run`. Since relation Method contains all information about methods, we must delete the old tuple of run and insert a new one with the updated return type. Even though the return type does not influence the CFG, our Datalog analysis must process the replaced tuple carefully to discover that the CFG is unaffected. Generally, large compound tuples hamper incrementality because the entire tuple has to be replaced even if just one element changes.
3. Insert a `println` statement at the beginning of the while-loop (index 3). Since the encoding uses absolute indices for statements, the indices of all subsequent statements have to be incremented. For an incremental Datalog engine, this is equivalent to deleting those statements and reinserting them at a different position. In our example, we would have to recompute all but the very first CfgEdge tuples.

In this chapter, we develop a data abstraction for Datalog that abstracts over the encoding of structured data. The data abstraction closes the representational gap (RG2) Data. In particular, we present an encoding of structured data in Datalog that enables efficient incremental updates. Our encoding supports incremental changes like those described above by (i) avoiding context-sensitive unique IDs, (ii) avoiding large compound tuples, and (iii) avoiding absolute positioning. Specifically, we show how our encoding supports arbitrary tree and list data. We have implemented our encoding as part of the incremental Datalog engine IncA [Szabó et al. 2016, 2018, 2021]. It is present in the original version IncA_{MPS} and in its re-implementation $\text{IncA}_{\text{Scala}}$. In the remainder of this chapter, we discuss the implementation in $\text{IncA}_{\text{Scala}}$: Users provide structured input data, which we translate into flat relations for IncA to process. As part of this, we developed efficient index structures for relations that encode structured data and demonstrate how to use tree diffing algorithms to trigger incremental updates of the encoded data.

In summary, we present the following contributions:

- An encoding of structured data in Datalog that supports efficient incremental updates (Section 3.3).
- An implementation of efficient indices to answer Datalog queries against encoded structured data (Section 3.4).
- A processor that translates tree-diffing patches into updates of the encoded data (Section 3.5).

Method(*uid*, *name*, *params*, *class*, *type*) = [
  ("App.run(int)","run","(int)","App","void")
]

Statement(*method*, *index*, *uid*) = [
  ("App.run(int)",1,"App.run(int)/assign/1"),
  ("App.run(int)",2,"App.run(int)/while/1"),
  ("App.run(int)",3,"App.run(int)/if/1"),
  ("App.run(int)",4,"App.run(int)/assign/2"),
  ("App.run(int)",5,"App.run(int)/else/1"),
  ("App.run(int)",6,"App.run(int)/assign/3")
]

Derived Relation:
CfgEdge(*method*, *fromStmIndex*, *toStmIndex*) = [
  ("App.run(int)",1,2),
  ("App.run(int)",2,3),
  ("App.run(int)",3,4),
  ("App.run(int)",3,6),
  ("App.run(int)",4,2),
  ("App.run(int)",6,2)
]

Figure 3.2: Tabular encoding of the program from Figure 3.1.

Our encoding has been tested and refined for the last 6 years and various case studies showcase the efficient incrementalization our encoding enables. In particular, we use the encoding as a basis for the derived incremental type checkers we developed in Chapter 2. We report on these case studies in Section 3.6, but they have been previously published.

3.2 Background on IncA

IncA originated as an incremental static analysis framework using Datalog, which we call IncA_{MPS}. It is implemented using the projectional language workbench MPS. We re-implemented IncA in Scala2 and re-imagined it as an incremental Datalog platform. We call the re-implementation $\text{IncA}_{\text{Scala}}$. We extend $\text{IncA}_{\text{Scala}}$ to support the efficient incremental processing of structured data. Note that a preliminary relational encoding of abstract syntax graphs, which the projectional editor of MPS uses, are present in IncA_{MPS}. This section describes the architecture of $\text{IncA}_{\text{Scala}}$ and explains how users interact with an incremental Datalog engine. This background knowledge is required for understanding where and how to support structured data in Datalog.

Architecture. We show the architecture of $\text{IncA}_{\text{Scala}}$ in Figure 3.3, which consists of a compiler frontend, a compiler backend, and a runtime system. The frontend provides different user-facing languages to describe programs with Datalog-style least-fixpoint

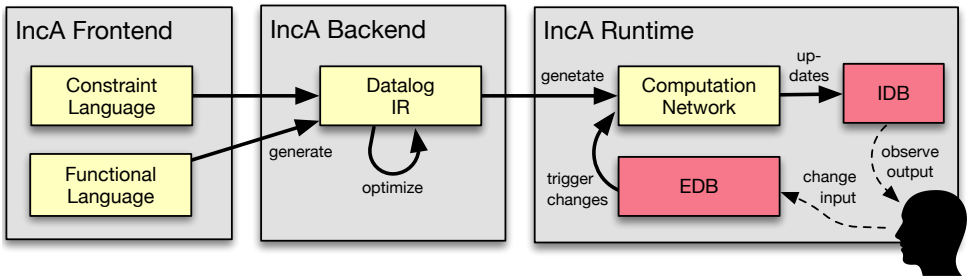


Figure 3.3: The architecture of IncA. Yellow boxes represent computations, red boxes represent input/output data.

semantics. In particular, the *constraint language* resembles traditional Datalog [Szabó et al. 2016; Szabó 2021], whereas the *functional language* promotes the definition of recursive functions. We will introduce the functional language in Chapter 4. All frontend languages must compile to IncA's intermediate representation (IR), which consists of *plain Datalog* with a few extensions: stratified negation, user-defined data types, and user-defined recursive aggregation [Szabó et al. 2018]. The architecture is open for extension: For example, there also is a frontend that compiles Soufflé-style Datalog [Scholz et al. 2016] to IncA's IR.

The compiler backend optimizes the Datalog IR generated by the frontends. Optimizations include constant folding, constant propagation, and inlining. After optimization, the backend generates code for a target platform to execute. The primary target platform of IncA is ViatraQuery, which implements incremental processing through a computation network [Varró et al. 2016]. We will focus on how to provide structured data for ViatraQuery in the remainder of this chapter. However, the architecture is open and our encoding is applicable to other target platforms, for example, $\text{IncA}_{\text{Scala}}$ can translate the IR to code that can be run by Soufflé.

Incremental user interaction. A Datalog program consists of logical rules that express implications. For example, consider the standard path example:

$$\begin{aligned} \mathsf{path}(X,Y) &\ \texttt{:-}\ \mathsf{edge}(X,Y). \\ \mathsf{path}(X,Y) &\ \texttt{:-}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y). \end{aligned}$$

This program computes the transitive closure path of a relation edge: (i) If there is an edge from X to Y, then there is a path from X to Y. (ii) If there is an edge from X to Z and if we already have found a path from Z to Y, then there also is path from X to Y. That is, this program computes tuples of a relation path given tuples of a relation edge as input.

In Datalog lingo, input relations define an extensional database (EDB), whereas the computed relations define an intensional database (IDB). When using a non-incremental Datalog engine, users must provide the EDB before execution and get back the IDB after execution. However, with an incremental Datalog engine, the user interactions become more interesting as shown on the right end of Figure 3.3.

With an incremental Datalog engine, users load a Datalog program with an initial EDB that is usually empty. For IncA, the Datalog engine instantiates a computation network that awaits user input. Users can then continuously provide EDB changes, which the computation network processes to update the IDB. The user can observe those changes

in form of deletions and insertions of tuples in the IDB. For example, consider the following incremental user interactions with the path program (+ is tuple insertion, − is tuple deletion):

$$
\begin{array}{lcl}
+\mathsf{edge}(1,2) & \Rightarrow & +\mathsf{path}(1,2) \\
+\mathsf{edge}(2,3) & \Rightarrow & +\mathsf{path}(2,3),\ +\mathsf{path}(1,3) \\
+\mathsf{edge}(3,1) & \Rightarrow & +\mathsf{path}(3,1),\ +\mathsf{path}(2,1),\ +\mathsf{path}(1,1),\ +\mathsf{path}(2,2),\ +\mathsf{path}(3,3)
\end{array}
$$

$$
\begin{array}{lcl}
-\mathsf{edge}(3,1) & & \\
+\mathsf{edge}(2,1) & \Rightarrow & -\mathsf{path}(3,1),\ -\mathsf{path}(3,3)
\end{array}
$$

Datalog interactions are inherently relational: tuple insertions and deletions. How can users provide and change non-relational data such as structured data?

Problem statement. We want to exploit Datalog's incrementality for all kinds of computations, including those that process structured data (e.g., program analyses). To this end, we must solve two challenges. First, we must find a relational encoding of structured data so that we can represent structured data in the EDB. Second, when the structured data changes, we must translate those changes into tuple insertions and deletions in the EDB. All the while, our encoding must not inhibit the incremental performance. That is, the encoding and the change translation may not increase the computational complexity of the incremental Datalog engine.

3.3 Encoding of Structured Data

When encoding structured data for an incremental Datalog engine, we must submit to one rule: Small changes in the structural data must translate to small changes of the EDB. In the remainder of this section, we show how to support three kinds of structured data: tree-shaped data, optional data, and lists. We follow three design principles in our encoding:

(i) Use context-insensitive unique identifiers.

(ii) Use unary and binary relations only.

(iii) Use relative positioning for list elements.

3.3.1 Encoding Trees

We use the following data type (in Scala3 syntax) as a running example to explain the relational encoding of trees:

```
1 enum Exp:
2   case Num(value: Int)
3   case Add(left: Exp, right: Exp)
```

For example, the tree `Add(Num(5), Num(3))` has an addition as root node with `Num(5)` as `left` subtree. We assign each node a unique ID, so that we can refer to and update it directly without any navigation through the tree. We annotate IDs as subscripts in our examples: $\texttt{Add}_{\#1}\texttt{(Num}_{\#2}\texttt{(5), Num}_{\#3}\texttt{(3))}$.

Our unique IDs are atomic and do not contain any context-sensitive information. This is important because it allows a tree to change independently from its context and vice versa. For the Java example from Section 3.1, we can rename a class without requiring changes to any of the contained trees that describe fields and methods. And conversely, we can move a tree to another context (e.g., another class) without updating its unique IDs. We show an example of such update below.

Unique IDs have an additional benefit: We can use them as keys in our relational encoding. Specifically, we encode trees using two families of relations: one that enumerates all nodes of a given type, and one that enumerates all links given a type and a field name:

$$\texttt{Node}_{type} \subseteq UID \qquad \texttt{Link}_{type.fieldname} \subseteq UID \times (UID \cup Value)$$

For each node type T, we collect the unique IDs of nodes in $\texttt{Node}_T$. For our example data type, these are three relations $\texttt{Node}_{Num}$, $\texttt{Node}_{Add}$, and $\texttt{Node}_{Exp}$, where the latter subsumes the former two due to subtyping. Datalog programs can query these node relations to emulate pattern matching.

To encode the shape of a tree, we use a binary link relation for each field. For each node type T and fld, we collect the links between a node and its field value in $\texttt{Link}_{T.fld}$. The value of a field is either a primitive $Value$ such as `Int`, or another node identified by its unique ID. Datalog programs can query these link relations to traverse structured data.

Using our node and link relations, we can encode the tree $\texttt{Add}_{\#1}\,\texttt{(Num}_{\#2}\,\texttt{(5), Num}_{\#3}\,\texttt{(3))}$ using the following Datalog tuples:

$$\begin{array}{lll}
\texttt{Node}_{Add}(\#1) & \texttt{Node}_{Num}(\#2) & \texttt{Node}_{Num}(\#3) \\
\texttt{Node}_{Exp}(\#1) & \texttt{Node}_{Exp}(\#2) & \texttt{Node}_{Exp}(\#3) \\
\texttt{Link}_{Add.left}(\#1, \#2) & \texttt{Link}_{Add.right}(\#1, \#3) & \\
\texttt{Link}_{Num.value}(\#2, 5) & \texttt{Link}_{Num.value}(\#3, 3) &
\end{array}$$

We can query these relations in Datalog code to collect all numeric literals found in an expression tree:

$$\begin{array}{lll}
\texttt{nums}(e,n) & \texttt{:-} & \texttt{Node}_{Num}(e),\ \texttt{Link}_{Num.value}(e,n). \\
\texttt{nums}(e,n) & \texttt{:-} & \texttt{Node}_{Add}(e),\ \texttt{Link}_{Add.left}(e,l),\ \texttt{nums}(l,n). \\
\texttt{nums}(e,n) & \texttt{:-} & \texttt{Node}_{Add}(e),\ \texttt{Link}_{Add.right}(e,r),\ \texttt{nums}(r,n).
\end{array}$$

This program computes a relation $\texttt{nums} \subseteq UID \times Value$. If e is a Num, then we extract its numeric literal stored in field $value$. Otherwise, e is an Add and we extract its left (resp. right) operand and recursively collect its numeric literals. For our example, this program will derive the following tuples:

$$\texttt{nums}(\#2,5) \quad \texttt{nums}(\#3,3) \quad \texttt{nums}(\#1,5) \quad \texttt{nums}(\#1,3)$$

Consider we negate the left-hand operand of our example tree, resulting in $\texttt{Add}_{\#1}\,\texttt{(Neg}_{\#4}\,\texttt{(Num}_{\#2}\,\texttt{(5)), Num}_{\#3}\,\texttt{(3))}$. We can represent this change compactly as tuple updates:

$$\begin{array}{llll}
+\texttt{Node}_{Neg}(\#4) & +\texttt{Node}_{Exp}(\#4) & & \\
-\texttt{Link}_{Add.left}(\#1, \#2) & +\texttt{Link}_{Add.left}(\#1, \#4) & +\texttt{Link}_{Neg.e}(\#4, \#2) &
\end{array}$$

The first, second, and last tuple change are responsible for loading the new *Neg* node. The amount of tuple changes needed for loading a new tree is proportional to its size. The other two changes manipulate $Add.left$ of $\texttt{Add}_{\#1}$ to replace the old operand by the new negation node. For updating existing nodes, the amount of tuple changes needed is proportional to the number of node changes in the structured data. Overall, this means that our encoding translates small changes in the structural data to small changes of the EDB.

3.3.2 Encoding Optional Data

Structured data regularly makes use of optional elements. For example, consider a lambda expression with an optional type annotation:

```
1  case Lam(param: String, ty: Option[Type], body: Exp)
```

We model optional data as fields whose value may be undefined. That is, if #1 is a *Lam* node, then $(\#1, v) \in \texttt{Link}_{Lam.ty}$ may be undefined for all v, meaning there is no type annotation present.

There are two concerns we must discuss: (i) what is the semantics of a field being undefined, and (ii) how can a Datalog program test the definedness of a field. To illustrate these concerns, consider the following Datalog rule, which type checks a lambda expression:

$$\mathsf{tcheck}(ctx, e) \quad :- \quad \texttt{Node}_{Lam}(e),\ \texttt{Link}_{Lam.ty}(e, t),\ \texttt{Link}_{Lam.param}(e, x),\ \texttt{Link}_{Lam.body}(e, b),\ \mathsf{bind}(ctx, x, t, ctx'),\ \mathsf{tcheck}(ctx', b).$$

The second call in the body of this rule queries the optional ty field, which may be undefined. Fortunately, the standard Datalog semantics dictates the correct semantics: When a subquery fails, then the entire rule is aborted and does not yield any outputs. That is, if ty is undefined, then the query $\texttt{Link}_{Lam.ty}(\mathrm{e}, \mathrm{t})$ fails, and thus the entire rule fails as desired. Hence, the query of $\texttt{Link}_{Lam.ty}$ tests for the definedness of that field. To allow users to test if a field is undefined, we add synthesized undefLink relations into the EDB:

$$\mathsf{tcheck}(ctx, e) \quad :- \quad \texttt{Node}_{Lam}(e),\ \texttt{undefLink}_{Lam.ty}(e),\ \mathsf{infer}(ctx, e).$$

We synthesize undefNode and undefLink for all node types and all fields. As we will explain in Section 3.4, these relations can be efficiently implemented as database views, that is, without any memory overhead.

3.3.3 Encoding Lists

At last, we encode lists as relations. Consider an additional `Exp` case for calls with multiple arguments **`case`** `Call(f: String, args: List[Exp])`. One option for handling lists is to represent them as standard structured data using `Cons` and `Nil` nodes. However, we would have to load and incrementally maintain a `Cons` node for each list element. Moreover, the *Cons* nodes would be visible to Datalog developers, for example, when querying the parent of a node. Alternatively, we could model a list node and assign indices to the list elements, similar to an array. However, as we have discussed in Section 3.1, this encoding inhibits incrementality, because deleting or inserting a list element affects the indices of all subsequent elements.

We propose an encoding that behaves like a linked list, but does not require explicit `Cons` nodes. To achieve that, we introduce two special relations:

$$\#first \subseteq UID \times UID \qquad \#next \subseteq UID \times UID$$

Relation $\#first$ associates a list node to its first list element; $\#first$ is undefined if a list is empty. And for each list element, relation $\#next$ associates it to the subsequent list element. For example, we encode the program $\texttt{Call}_{\#1}\texttt{("f", List}_{\#2}\texttt{(Num}_{\#3}\texttt{(5), Num}_{\#4}\texttt{(3), Num}_{\#5}\texttt{(7)))}$ as the following Datalog tuples:

$\texttt{Node}_{Call}(\#1)$ $\texttt{Node}_{Exp}(\#1)$ $\texttt{Link}_{Call.f}(\#1, \text{"f"})$
`// Node and Link tuples for Num nodes #3,#4,#5elided`
$\texttt{Node}_{List[Exp]}(\#2)$ $\texttt{Link}_{Call.args}(\#1, \#2)$ $\#first(\#2, \#3)$
$\#next(\#3, \#4)$ $\#next(\#4, \#5)$

The node with UID #2 is a list node of type $List[Exp]$. Its first element is #3 as defined by the $\#first$ fact, with subsequent elements #4 and #5. This encoding is incrementally efficient in the sense that we can replace, insert, or remove any list element in constant tuple changes, akin to the corresponding operations on linked lists. Finally, note that Datalog programs can query $\#next$ in the opposite direction to find a list element's predecessor. Thus, in fact, our encoding supports bidirectional traversals similar to doubly linked lists.

3.4 Querying Encoded Structured Data

In the previous section, we showed how to encode structured data in a relation format: To this end, we introduced the following relations:

- Node relations: $\texttt{Node}_{type} \subseteq UID$
- Link relations: $\texttt{Link}_{type.fieldname} \subseteq UID \times (UID \cup Value)$
- First link relation: $\#first \subseteq UID \times UID$
- Next link relation: $\#next \subseteq UID \times UID$
- Undefined nodes: $\mathsf{undefNode}_{type} \subseteq UID$
- Undefined links: $\mathsf{undefLink}_{type.fieldname} \subseteq UID$

The relations above serve as input (EDB) for the incremental Datalog engine and its computation network. Our encoding determines how structured data appears in the EDB. The computation network interacts with the EDB in three ways: (i) it enumerates tuples of an EDB relation, (ii) it tests the containment of a tuple in an EDB relation, (iii) it reacts to the insertion or deletion of a tuple from an EDB relation. In this section, we explain how these interactions can be supported efficiently by exploiting invariants of structured data. Specifically, we introduce data structures called *indices* for the EDB that can answer the queries of the computation network efficiently.

```
class UnarySetIndex[V]:
  private val set: mutable.Set[V]

  def entries: Iterable[V] = set
  def contains(v: V): Boolean = set.contains(v)

  def insert(v: V): Unit =
    set += v
    notify(v, true)

  def delete(v: V): Unit =
    set -= v
    notify(v, false)

  def notify(v: V, isInsert: Boolean): Unit = ...
  def addListener(l: Listener): Unit = ...
  def removeListener(l: Listener): Unit = ...
```

Figure 3.4: Unary set index implementation.

3.4.1 Node Relations

Nodes are fundamental entities of tree-shaped data. We encode nodes using node relations in the EDB as explained in Section 3.3. Hence, we need to enable efficient querying of node relations for the computation network. In our encoding, nodes exhibit an important invariant: each node carries a unique ID. For the node relations, this means that we do not need to consider duplicate node IDs and can use a simple set to store the UIDs of nodes per type.

Figure 3.4 shows the code of our unary set index. There is one instance of this class per type. In each type, we store the contents of the node relation in a mutable set. The index can answer queries such as enumerating all stored nodes (`entries`) and if a node is already indexed (`contains`).

The EDB will change over time by processing updates of the input data (details in Section 3.5). To this end, the index provides functions `insert` and `delete` to alter the underlying mutable set. The computation network installs listeners on EDB relations to be notified about changes. Our unary set index notifies these listeners when a node is inserted or deleted by calling `notify`. The first argument of `notify` states the value that the change is about and the second argument informs the listeners if the change was an insertion (`isInsert = true`) or a deletion (`isInsert = false`). Since nodes are uniquely identified, double insertions or double deletions cannot occur and do not have to be prevented.

3.4.2 Link Relations

We encode the shape of structured data using link relations in the EDB as explained in Section 3.3. Given a type and field name, the link relations associate nodes with other nodes or with primitive values. To efficiently implement indices for link relations, we need to identify invariants that we can exploit to detect and propagate changes.

As first invariant, we observe that the target of a link is always uniquely determined by the containing node. Formally, given a node n, a type T, and a field fld, then

$$|\{t \mid (n,t) \in \texttt{Link}_{T.fld}\}| \leq 1.$$

That is, a field's value is either undefined or definite; there cannot be multiple values assigned to a single field of a node. However, the computation network may also query the link relation in reverse order: given a link target, what are the nodes that point there? In this direction, the link relations do not provide a strong invariant. For example, the following tree `Add`$_{\#1}$`(Num`$_{\#2}$`(3), Num`$_{\#3}$`(3))` contains the integer value 3 twice, as target of #2 and #3. But this is only a problem for primitive values, which are not unique.

To implement link relations efficiently, we split them apart. We distinguish *node link relations* (targeting nodes) from *value link relations* (targeting primitive values). Node link relations range over $UID \times UID$ whereas value link relations range over $UID \times Value$. This separation is always possible for well-sorted trees, where a field targets a statically known type: either a node or a value. Now we can observe a second invariant for node link relations: the target of a link uniquely determines the containing node. This invariant does not hold for value link relations.

Due to these invariants, we can implement:

- node link relations as one-to-one indices and
- value link relations as many-to-one indices.

These indices share a common interface presented in Figure 3.5. The interface provides the same functionality as `UnarySetIndex` such as enumerating all elements as well as registering and notifying listeners but for binary tuples. Additionally, it provides two functions that answer directional queries. The function `index` gives all values that have `k` as key whereas `indexInverted` returns all keys where `v` is the value. The concrete implementations of these two functions will directly make use of the two invariants of binary relations.

We show both implementations at the bottom of Figure 3.5, utilizing the respective invariants. We implement `OneToOneIndex` that exploits the one-to-one mapping between keys and values. To exploit the invariant, we use a bidirectional map that maintains an inverse view of the data to store a one-to-one mapping efficiently. We implement `ManyToOneIndex` that exploits the many-to-one mapping between keys and values. That is, a key uniquely identifies its value, but a value may be contained by multiple keys. To exploit this invariant, we use a forward-directed map to answer `index` queries and a backwards-directed multimap for `indexInverted` queries.

3.4.3 Optional and List Relations

We encode optional data as links with undefined targets. We already took this into consideration when we defined the one-to-one and many-to-one indices above, where `index` and `indexInverted` may yield empty results. Thus, no additional handling is needed to support optional data.

```
trait BinaryIndex[K, V]:
  def entries: Iterable[(K, V)]

  def index(k: K): Iterable[V]
  def indexInverted(v: V): Iterable[K]
  def contains(k: K, v: V): Boolean = index(k).exists(_ == v)

  def insert(k: K, v: V): Unit
  def delete(k: K, v: V): Unit

  def notify(k: K, v: V, isInsert: Boolean): Unit = ...
  def addListener(l: Listener): Unit = ...
  def removeListener(l: Listener): Unit = ...

class OneToOneIndex[K, V] extends BinaryIndex[K, V]:
  private val biMap: mutable.BiMap[K, V]

  def entries: Iterable[(K, V)] = biMap.entries

  def index(k: K): Iterable[V] = biMap.get(k)
  def indexInverted(v: V): Iterable[K] = biMap.getInverse(v)

  def insert(k: K, v: V): Unit =
    biMap.put(k, v)
    notify(k, v, true)
  def delete(k: K, v: V): Unit =
    biMap.remove(k)
    notify(k, v, false)

class ManyToOneIndex[K, V] extends BinaryIndex[K, V]:
  private val forward: mutable.Map[K, V]
  private val backward: mutable.MultiMap[V, K]

  def entries: Iterable[(K, V)] = forward.entries

  def index(k: K): Iterable[V] = forward.get(k)
  def indexInverted(v: V): Iterable[K] = backward.get(v)

  def insert(k: K, v: V): Unit =
    forward.put(k, v)
    backward.put(v, k)
    notify(k, v, true)
  def delete(k: K, v: V): Unit =
    forward.remove(k)
    backward.remove(v, k)
    notify(k, v, false)
```

Figure 3.5: Generic binary index interface and binary one-to-one and many-to-one index implementations.

To encode lists, we use #*first* and #*next* relations. They also exhibit strong invariants: Each list has at most one first element and each node can only be the first element of a single list only. Hence, we can use our one-to-one index to implement the #*first* relation. Moreover, each node can only be the successor of a single list predecessor, so that we can use the one-to-one index for the #*next* relation, too.

```
class UndefNode(tyNodes: UnarySetIndex, allNodes: UnarySetIndex):
  def entries: Iterable[UID] =
    allNodes.entries.diff(tyNodes.entries)

  def contains(k: UID): Boolean =
    !tyNodes.contains(k)

  allNodes.addListener((v: URI, isInsertion: Boolean) =>
    if (!tyNodes.contains(v))
      notify(v, isInsertion)
  )
```

Figure 3.6: Index undefNode reacts to `allNodes` changes.

```
class UndefLink(tyNodes: UnarySetIndex, links: OneToOneIndex):
  def entries: Iterable[UID] =
    tyNodes.entries.filter(n => links.index(n).isEmpty)

  def contains(k: UID): Boolean =
    tyNodes.contains(k) && links.index(k).isEmpty

  tyNodes.addListener((v: URI, isInsertion: Boolean) =>
    // fields are undefined when a node is (un)loaded
    notify(v, isInsertion)
  )
  links.addListener((v: URI, t: URI, isIns: Boolean) =>
    notify(v, !isIns)
  )
```

Figure 3.7: undefLink reacts to `tyNodes` and `links` changes.

3.4.4 Virtual Indices

The goal of our indices is to enable efficient querying of structured data encoded in the EDB. However, sometimes it is not necessary to materialize the structured data explicitly. We found that we can answer queries for the relations undefNode and undefLink by relying on other EDB relations without storing extra data. In databases, this would be called a *view*. We implement views with what we call *virtual indices*.

A virtual index supports the same interactions as other indices: enumeration, containment, and incremental updates. However, it does so by delegation to other EDB indices. For example, consider the undefNode relation. For each type T, undefNode_T contains all nodes not of type T. Rather than storing all these nodes explicitly, we can answer queries for this relation indirectly as shown in Figure 3.6.

The virtual index `UndefNode` takes two other indices as input: one containing all nodes of type T and one containing all nodes. The nodes *not* of type T can then be computed as the difference of all nodes and the nodes of type T. For containment, we only need to test `tyNodes`, because the membership in `allNodes` is implicit. However, the most interesting and complicated part of virtual indices concerns the handling of incremental updates. To this end, we install two change listeners in other relations. When `allNodes` changes, we propagate the change if the changed node is not of type T. A new node will become part of the `UndefNode` relation if and only if it is not of type T. Note that we do not observe changes in `tyNodes`, because the type of a node can not change.

```
type Field = (Type,Link)

class EDB:
  val nodes: Map[Type, UnarySetIndex[UID]]
  val nodeLinks: Map[Field, OneToOneIndex[UID, UID]]
  val valLinks: Map[Field, ManyToOneIndex[UID,Value]]
  val first: OneToOneIndex[UID, UID]
  val next: OneToOneIndex[UID, UID]

  // UndefNode and UndefLink indices
  val virtualIndices: Map[VKey, VirtualIndex]
```

Figure 3.8: Relational encoding, backed by efficient indices.

Similarly, we can define a virtual index `UndefLink` that contains all nodes of type T with an undefined link fld. This virtual index listens to two other indices as shown in Figure 3.7. When a node is inserted into `tyNodes`, its links are undefined by default. Therefore, we notify the listeners of `UndefLink` whenever `tyNodes` changes. If later the value of $T.fld$ is changed by the user, we must update `UndefLink` accordingly. When a new tuple (v,t) is inserted into $\texttt{Link}_{T.fld}$ (the field is set to a new value), then we must remove v from $\texttt{undefLink}_{T.fld}$. Conversely, when a tuple (v,t) is deleted from $\texttt{Link}_{T.fld}$ (the field is undefined), then we must insert v into $\texttt{undefLink}_{T.fld}$. The `links` listener captures this behavior.

Virtual indices allows us to implement additional EDB relations without memory overhead. Note also that there are no insertion and deletion functions in virtual indices, because they do not store data themselves.

3.4.5 Summary

We have now presented all indices required to implement our encoding of structural data as an EDB. We summarize the EDB definition in Figure 3.8. Note how node relations are indexed by the node type and link relations are indexed by the node type and link. In our implementation, indices are instantiated on-demand when they are first needed. For node relations, we use the unary set index. For binary relations with a one-to-one invariant, we use the `OneToOneIndex`. For binary relations with a many-to-one invariant, we use the `ManyToOneIndex`. This indices allow the computation network to process changes of structural data efficiently.

3.5 Processing Changes of Structured Data

The EDB implementation describes the structured data in a relational encoding. To react to input changes, we need to translate changes of structured data to insertions and deletions of EDB tuples. To determine changes in structured data, we use the structural diffing algorithm *truediff* [Erdweg et al. 2021]. *truediff* generates edit scripts that precisely describe tree-diff patches, but it also describes how to construct an initial tree using edits. The runtime of $\text{IncA}_{\text{Scala}}$ will translate edit scripts to insertions and deletions of the EDB as we show below. But first, we introduce the edit script language used by *truediff*.

```
case class EditScript(edits: Seq[Edit])

enum Edit:
  case Load(n: Node, ks: Kids, vs: Vals)
  case Attach(n: Node, link: Link, p: Node)
  case Unload(n: Node, ks: Kids, vs: Vals)
  case Detach(n: Node, link: Link, p: Node)
  case Update(n: Node, ovs: Vals, nvs: Vals)

type Node = (Type, UID)
type Kids = Seq[(Link, UID)]
type Vals = Seq[(Link, Value)]
```

Figure 3.9: The edit script language *truechange* [Erdweg et al. 2021].

3.5.1 *truechange*: Structural Edit Scripts

Tree diffing algorithms like *truediff* generate edit scripts that describe how a tree was modified. For *truediff*, the corresponding edit script language is called *truechange*. *truechange* uses UIDs to uniquely identify nodes of abstract syntax trees. This way, *truediff* can directly reference changed nodes without requiring navigation through the tree. This design fits well with the extensional database encoding of structured data we introduced in Section 3.3, which also relies on UIDs.

We reiterate the edit script language *truechange* [Erdweg et al. 2021] in Figure 3.9. An edit script consists of a sequence of edits. Each edit is one of five atomic operations: load, attach, unload, detach or update. Edits reference nodes by the node's type together the node's UID, which we visualize as a subscript. For example, node $\texttt{Add}_{\#1}$ is a node of type `Add` with UID #1.

A `Load(n,ks,vs)` constructs a new node with a new UID and connects that node with all its child nodes `ks` and contained values `vs`. Note how child nodes and contained values also provide a link, with which they connect to the newly constructed node. An `Unload(n,ks,vs)` is the dual edit of `Load`: it deconstructs a node and disconnects all child nodes, which remain loaded. Disconnected nodes can be reattached through another `Load` or through `Attach`. An `Attach(n,link,p)` connects node `n` with the parent node `p` via link `link` if that link is currently unoccupied. Conversely, `Detach(n,link,p)` disconnects `n` from `p`. At last, an `Update(n,ovs,nvs)` edit which updates the old values `ovs` of the node with new values `nvs`.

For example, consider the change we have already seen in Section 3.3 on the left-hand side where *truediff* computes the *truechange* edit script on the right-hand side:

$\texttt{Add}_{\#1}(\texttt{Num}_{\#2}(5), \texttt{Num}_{\#3}(3))$
$\texttt{Add}_{\#1}(\texttt{Neg}_{\#4}(\texttt{Num}_{\#2}(5)), \texttt{Num}_{\#3}(3))$

Detach(Num$_{\#2}$, "left", Add$_{\#1}$)
Load(Neg$_{\#4}$, Seq("e" -> #2), Seq())
Attach(Neg$_{\#4}$, "left", Add$_{\#1}$)

First, we detach node $\texttt{Num}_{\#2}$ from link `left` of node $\texttt{Add}_{\#1}$. Then we load a new node of type $\texttt{Neg}_{\#4}$ with node $\texttt{Num}_{\#2}$ attached at link `e`. Lastly, we attach the newly loaded node $\texttt{Neg}_{\#4}$ at the previously freed link `left` of node $\texttt{Add}_{\#1}$. We process edit scripts like this in $\text{IncA}_{\text{Scala}}$ to update the structured data encoded in the EDB.

3.5.2 Translating Edit Scripts

The runtime of IncA$_{\text{Scala}}$ takes an edit script as input and translates it to changes of the extensional database. We assume that the type of the structured data is consistent with the node and link relations in the EDB: For each node type, there are corresponding node and link relations in the EDB. The translation then follows the code in Figure 3.10.

The EDB provides the function `processEditscript`, which processes each edit individually by calling `processEdit`. The EDB only notifies the computation network after the whole edit script has been processed and the EDB has been altered. The function `processEdit` handles each of the five edits differently. When processing a load edit, we insert the UID of the node into all supertype node indices. We also insert into the supertype indices to enable the computation network to query nodes of all types without requiring extra computation. Additionally, we need to insert all kids in the corresponding link index. The same holds for values of the node. Note that loads of list nodes do not have direct kids and values.

When attaching a node we distinguish between the provided links. When considering link `First`, we insert it into the `first` index. The same holds for the when encountering `Next`. We need to make this distinction because first and next links are not indexed by the nodes type. If the link is neither a first or next link, we insert the parent-child pair into the link relation indexed by the parent's type and the link itself.

Since `Load` and `Unload` are dual edits, we delete instead of insert into the same indices when encountering an unload edit. The same reasoning holds for `Attach` and `Detach`.

At last, we process the Update edit by deleting all old values `ovs` for the respective value link relations and inserting all new values `nvs` afterwards. We assume that `ovs` and `nvs` mention the same links.

Consider the edit script we have seen in the previous subsection. This edit script will be processed into the following insertions and deletions:

$$
\begin{array}{l}
-\texttt{Link}_{Add.left}(\#1, \#2) \\
\hline
+\texttt{Node}_{Neg}(\#4) \qquad +\texttt{Node}_{Exp}(\#4) \quad +\texttt{Node}_{Any}(\#4) \\
+\texttt{Link}_{Neg.e}(\#4, \#2) \\
\hline
+\texttt{Link}_{Add.left}(\#1, \#4)
\end{array}
$$

We separate the insertions and deletions into three packages where each package is emitted by one edit. The detach edit results into a single link relation deletion of $Add.left$. The load edit results in three node relation insertions: Neg, Exp, and Any, one for each supertype of Neg. Our implementation assumes that every node type has `Any` as its supertype. Additionally, we insert into the link relation of $Neg.e$. The attach edit results in another link relation insertion of $Add.left$.

```
def processEditScript(es: EditScript): Unit =
  es.edits.foreach(processEdit)

def processEdit(edit: Edit): Unit = edit match
  case Load((ty, uid), kids, vals) =>
    supertypes(ty).foreach { supty =>
      nodes(supty).insert(uid)
    }
    kids.foreach { case (link, kuid) =>
      links(ty -> link).insert(uid, kuid)
    }
    vals.foreach { case (link, v) =>
      valLinks(ty -> link).insert(uid, v)
    }
  case Attach((nty, nuid), First, (pty, puid)) =>
    first.insert(puid, nuid)

  case Attach((nty, nuid), Next, (pty, puid)) =>
    first.next(puid, nuid)

  case Attach((nty, nuid), link, (pty, puid)) =>
    links(pty -> link).insert(puid, nuid)

  case Unload((ty, uid), kids, lits) =>
    supertypes(ty).foreach { supty =>
      nodes(supty).delete(uid)
    }
    kids.foreach { case (link, kuid) =>
      links(ty -> link).delete(uid, kuid)
    }
    vals.foreach { case (link, v) =>
      valLinks(ty -> link).delete(uid, v)
    }

  case Detach((nty, nuid), First, (pty, puid)) =>
    first.delete(puid, nuid)

  case Detach((nty, nuid), Next, (pty, puid)) =>
    first.delete(puid, nuid)

  case Detach((nty, nuid), link, (pty, puid)) =>
    links(pty -> link).insert(puid, nuid)

  case Update((nty, nuid), ovs, nvs) =>
    ovs.foreach { case (link, v) =>
      valLinks(nty -> link).delete(v)
    }
    nvs.foreach { case (link, v) =>
      valLinks(nty -> link).insert(v)
    }
```

Figure 3.10: Translating edits to EDB insertions and deletions.

Language	Analysis	Programs	Non-inc. time	Inc. time	Speedup
Java	FindBugs	mbeddr importer (10k LoC)	n/a	7 ms	n/a
C	flow-sensitive points-to	Toyota ITC code[2](15k LoC)	5800 ms	23.3 ms	249x
C	well-formedness checks	Smart Meter (44k LoC)	209 ms	12.8 ms	16.3x
Jimple	Strong-update points-to	GTruth (9k), Gson (14k)	6500–64300 ms	1–10 ms	650–6430 x
Jimple	String analysis	PGSQL, JDBC (45k) BerkleyDB (70k)	13500–20400 ms	2 ms	6500–10000x
Jimple	Constant propagation	antlr (22k), emma (26k)	5000–23000 ms	1–3 ms	1600–7600x
Jimple	Interval analysis	pmd (61k), ant (105k)	3000–23000 ms	1–6 ms	500–3800x
Lambda	type checking	synthesized code	6/50 ms	44/2 ms	0.14/25x

Figure 3.11: IncA performance experiments based on our encoding of structured data.

3.6 Evaluation

We implemented our encoding for structural data as part of the incremental Datalog engine IncA. IncA is focused on static program analysis that processes the abstract syntax tree of programs. Our encoding was used to represent those trees and to make incremental changes to them since 2016. Our encoding has been extensively experimented with and we have refined it multiple times, as we report below.

3.6.1 Evolution of IncA

IncA was introduced in 2016 [Szabó et al. 2016] as a constraint language for incremental static analysis based on Datalog. In 2018, IncA was extended with user-defined recursive aggregation by introducing a new incremental algorithm for Datalog [Szabó et al. 2018]. In 2021, this algorithm was refined to improve the scalability of IncA for whole-program analyses [Szabó et al. 2021]. During this time, IncA was implemented in the JetBrains Meta Programming System (MPS) [1], which uses projectional editing. Projectional editing means that the user edits a view of the program's abstract syntax. To change the tree, the user triggers change requests that are applied to the abstract syntax, which then is re-projected for the user. Within IncA, these change requests were translated to trigger corresponding changes in the EDB, where the encoded structured data resides.

IncA was re-implemented in Scala2 as an open-source project in 2020. Conceptually, the main change is that IncA does not rely on projectional editing anymore, but uses efficient tree diffing to trigger concise changes [Erdweg et al. 2021]. In Section 3.5, we showed how to process tree-diff patches to update the relational encoding of structured data. The re-implementation also realizes a number of performance improvements: Compiler optimizations like constant folding, constant propagation, and inlining, and virtual indices in the EDB.

3.6.2 Performance Evaluation

Our encoding has been used in various IncA performance experiments. We provide an overview of these experiments in Figure 3.11.

[1] `https://www.jetbrains.com/mps`

The initial IncA paper evaluated the incremental performance of IncA on three program analyses for Java and C [Szabó et al. 2016]. These analyses process the structured source-level code directly, whereas later analyses processed some intermediate format such as Java bytecode in form of Jimple [Lam et al. 2011]. In all cases, the input language was modeled in MPS as abstract syntax that can be modified programmatically or through projectional user interactions. The abstract syntax was then transformed into an EDB schema using the encoding presented in Section 3.3.

Except the last one, all analyses were written against the encoded structured data in Datalog. That is, the analyses query our Node and Link relations and make use of our option and list encoding. In contrast, the last analysis was written in the type system DSL that compiles to Datalog, hiding the details of our encoding from the developer, which we discussed in Chapter 2. In general, we show that generating Datalog code is feasible by developing a functional programming language, which is a completely unrelated programming paradigm in Chapter 4.

Unfortunately, there is no standard benchmark for incremental source-code changes. Therefore, the IncA performance experiments synthesized changes programmatically, trying to impact the analysis result in order to challenge the incremental engine. The evaluation results demonstrate that our encoding of structured data does not impede incremental performance. Indeed, small source-code changes were translated into small EDB changes, thus triggering a minimal incremental update.

3.7 Related Work

We propose to encode structured data following three principles: (i) use context-insensitive unique identifiers to reference structured data, (ii) only use unary and binary relations to describe relations between data, and (iii) use relative positioning to encode list data. In the remainder of this section, we compare our design with related work, without discussing data abstractions for Datalog. In Chapter 6, we will discuss related Datalog systems and compare if and how they provide data abstractions. Note that except where we say so explicitly, the encodings used by related work were not designed with incremental updates in mind.

DIMPLE [Benton and Fischer 2007] and Eichberg et al. [2007] use tabled Prolog to describe static analyses, where the latter also supports incrementality. They use nested constructors to describe the abstract syntax. While this is possible when using Prolog, we focus on using Datalog to incrementalize computations, which requires flat relations containing primitive data such as strings, integers, and booleans. Eichberg et al. [2007] also uses absolute position indices to encode the instruction ordering, while our approach uses first and next links.

The work on query shredding [Cheney et al. 2014; Koch et al. 2016; Smith et al. 2020] tackles a problem of nested queries over nested data in the form of bags. These queries return nested data itself. In our work we only consider nested data in the form of tree-shaped data that is the input of Datalog programs. Our work does not focus on generating tree-shaped data during the evaluation of Datalog programs. The derived tuples can

[2] https://github.com/regehr/itc-benchmarks

reference tree-shaped data in the form of unique identifiers, but these unique identifiers only ever reference tree-shaped data that is part of the extensional database. Hence, we do not encounter the problem of deep updates that query shredding tries to solve.

TreeToaster [Balakrishnan et al. 2021] proposes to implement compiler optimizations by applying incremental view maintenance on abstract syntax trees (ASTs) for pattern rewritings. They encode ASTs using relations as well. They also identify AST nodes with unique identifiers. However, they use n-ary tuples to describe ASTs nodes. We propose to only use at most binary relations to encode tree-shaped data to enable small insertions and deletions when editing a single child node. TreeToaster does not allow recursive pattern matches while we integrated our work with state-of-the-art Datalog engines which allow processing highly recursive Datalog programs and terminate even in the presence of cyclic data.

Adapton [Hammer et al. 2015] is a general-purpose programming language to describe incremental computations. The language is ML-like and supports algebraic data types to encode structured data. While it is natural for Adapton to describe structured data, the challenge they encounter is to incrementalize computations. Our work uses Datalog which has an incremental semantics since the 1990's [Gupta et al. 1993]. In contrast, describing structured data in Datalog is not natural and, in particular, it is not obvious how to describe structured data such that it enables efficient incremental performance. Hence, our work tackles incremental computing from a different angle.

There are other incrementalization techniques such as memoization [Pugh and Teitelbaum 1989; Liu and Teitelbaum 1995; Abadi et al. 1996]. Memoization reuses a result when the input of a function does not change, even if the changed part of the input does not contribute to the result. Hence, if one part of the structured data changes, memoization requires to redo the computation. In contrast, Datalog is more fine-grained in the incremental update propagation and our encoding of structured data retains this granularity. That is, we only update a computation's result if relevant parts of the input data changes.

3.8 Chapter Summary

In this chapter, we showed how to process structured data in Datalog incrementally. Specifically, we developed a data abstraction for Datalog abstracting over the encoding of structured data that closes the representational gap (RG2) Data. We presented an encoding of structured data in flat relations that is amenable to incremental updates such that small changes to the structured data translate to small changes of the relations. In doing so, we discovered and followed three design principles: (i) use context-insensitive unique identifiers, (ii) use unary and binary relations only, and (iii) use relative positioning for list elements. We also showed how to implement the relations required by our encoding efficiently, exploiting invariants about the structural data to optimize time and memory. Finally, we explained how users can provide and update the encoded structural data using the tree-diffing algorithm *truediff*, which finds structural differences between trees efficiently and describes them concisely. Our encoding has been part of the IncA framework since 2016, but this chapter presents its first systematic description.

4

A Functional Datalog Frontend

This chapter is based on the peer-reviewed ECOOP'22 paper "Functional Programming with Datalog" [Pacak and Erdweg 2022] and is joint work with Sebastian Erdweg.

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 engines efficiently compute all derivable tuples even for recursive queries. However, as we argue in this chapter, Datalog is ill-suited as a programming language and Datalog programs are hard to write and maintain. We propose a linguistic abstraction for Datalog. That is, 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 engines. 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.

4.1 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 [Maier et al. 2018], so that all queries are guaranteed to terminate. Nowadays, Datalog is being used in a wide array of applications [Huang et al. 2011], from program analysis [Bravenboer and Smaragdakis 2009b; Madsen and Lhoták 2020; Szabó et al. 2021] to network monitoring [Abiteboul et al. 2005] and distributed computing [Alvaro et al. 2010a, 2011]. 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.

4

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 engine 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* 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 4.1 shows a functional program and a Datalog program that construct the control-flow graphs for the While language. The functional program uses pattern matching and set-comprehensions to compute sets of edges similar to [Nielson et al. 1999], 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 engines such as Soufflé [Scholz et al. 2015], 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:

$$\begin{aligned} \mathsf{flowTrans}(Prog, From, To) &\text{ :- } \mathsf{flow}(Prog, From, To). \\ \mathsf{flowTrans}(Prog, From, To) &\text{ :- } \mathsf{flow}(Prog, From, Inter), \mathsf{flowTrans}(Prog, Inter, To). \end{aligned}$$

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:

$$
\begin{array}{lll}
\mathsf{flow}(Stm,From,To) & :- & \mathsf{sequence}(Stm,Stm1,\_),\ \mathsf{flow}(Stm1,From,To).\\
\mathsf{flow}(Stm,From,To) & :- & \mathsf{sequence}(Stm,\_,Stm2),\ \mathsf{flow}(Stm2,From,To).\\
\mathsf{flow}(Stm,From,To) & :- & \mathsf{sequence}(Stm,Stm1,Stm2),\ \mathsf{final}(Stm1,From),\ \mathsf{init}(Stm2,To).\\
\mathsf{flow}(Stm,From,To) & :- & \mathsf{if}(Stm,C,Stm1,\_),\ \mathsf{true}(C),\ \mathsf{flow}(Stm1,From,To).\\
\mathsf{flow}(Stm,Stm,To) & :- & \mathsf{if}(Stm,C,Stm1,\_),\ \mathsf{true}(C),\ \mathsf{init}(Stm1,To).\\
\mathsf{flow}(Stm,From,To) & :- & \mathsf{if}(Stm,C,\_,Stm2),\ \mathsf{false}(C),\ \mathsf{flow}(Stm2,From,To).\\
\mathsf{flow}(Stm,Stm,To) & :- & \mathsf{if}(Stm,C,\_,Stm2),\ \mathsf{false}(C),\ \mathsf{init}(Stm2,To).\\
\mathsf{flow}(Stm,From,To) & :- & \mathsf{if}(Stm,C,Stm1,\_),\ !\mathsf{true}(\mathsf{C}(,)\ not\mathsf{false}(C),\mathsf{flow}(Stm1,From,To).\\
\mathsf{flow}(Stm,Stm,To) & :- & \mathsf{if}(Stm,C,Stm1,\_),\ !\mathsf{true}(\mathsf{C}(,)\ not\mathsf{false}(C),\mathsf{init}(Stm1,To).\\
\mathsf{flow}(Stm,From,To) & :- & \mathsf{if}(Stm,C,\_,Stm2),\ !\mathsf{true}(\mathsf{C}(,)\ not\mathsf{false}(C),\mathsf{flow}(Stm2,From,To).\\
\mathsf{flow}(Stm,Stm,To) & :- & \mathsf{if}(Stm,C,\_,Stm2),\ !\mathsf{true}(\mathsf{C}(,)\ not\mathsf{false}(C),\mathsf{init}(Stm2,To).\\
\mathsf{flow}(Stm,From,To) & :- & \mathsf{while}(Stm,\_,Stm1),\ \mathsf{flow}(Stm1,From,To).\\
\mathsf{flow}(Stm,Stm,To) & :- & \mathsf{while}(Stm,\_,Stm1),\ \mathsf{init}(Stm1,To).\\
\mathsf{flow}(Stm,From,Stm) & :- & \mathsf{while}(Stm,\_,Stm1),\ \mathsf{final}(Stm1,From).
\end{array}
$$

Figure 4.1: Constructing control-flow graphs using functional programming and Datalog.

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 [Bravenboer and Smaragdakis 2009b], despite the shortcomings of its frontend.

In this chapter, we design a functional programming language with fixpoint semantics and propose it as a "new" Datalog frontend: *functional IncA*. The new Datalog frontend poses as a linguistic abstraction for Datalog which closes the representational gap (RG1) Computation. 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 inputs, we can track these inputs in Datalog

```
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 4.2: A data-flow analysis using functional programming with fixpoint semantics.

relations. For programs without algebraic data types, we can adopt a standard demand transformation [Tekle and Liu 2010]. 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 4.2 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 engine to run the program, whereas prior Datalog dialects usually require a custom Datalog engine.

We have implemented functional IncA as part of the incremental Datalog framework IncA [Szabó et al. 2021]. Specifically, we implement it as part of the Scala2 reimplementation $\text{IncA}_{\text{Scala}}$. We translate functional IncA into a Datalog IR and provide two backends: one targeting IncA directly, the other targeting Soufflé [Scholz et al. 2015]. While the IncA backend provides incremental re-evaluation after input changes, the Souf-

flé 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 recursive 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 we already explored encoding type checkers in Datalog in Chapter 2, 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 engines 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 chapter how standard defunctionalization [Reynolds 1998] 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 4.3).
- We show how to compile user-defined algebraic data types to Datalog and extend functional IncA accordingly (Section 4.4).
- 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 4.6).
- 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 4.5), data-flow analyses for the While language (Section 4.7.1), and clone detection of Java bytecode (Section 4.7.2).
- We provide two backends for functional IncA, one targeting the incremental Datalog engine used by IncA, the other targeting the non-incremental Datalog engine Soufflé (Section 4.8).

4.2 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 engines 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 engines and some features have become standard nowadays. For example, Datalog engines support stratified negation and arithmetic operations, even though neither is part of core Datalog [Maier et al. 2018].

Modern Datalog engines provide a range of different extensions that their users can choose from. For example, Soufflé [Scholz et al. 2016] provides records, algebraic data types, and user-defined functions; ViatraQuery [Varró et al. 2016], the Datalog engine used by IncA, supports user-defined data types and recursive aggregation over user-defined functions [Szabó et al. 2018, 2021]. 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:

$$\mathsf{R}(x, res) \quad :- \quad \mathsf{h}(x, y), \mathsf{g}(y, z), \mathsf{f}(z, res).$$

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:

$$\begin{array}{lllll} \mathsf{R}(b1,b2,x1,x2,x3,x4,res) & :- & b1, & b2, & res = x1 + x3. \\ \mathsf{R}(b1,b2,x1,x2,x3,x4,res) & :- & b1, & !b2, & res = x1 + x4. \\ \mathsf{R}(b1,b2,x1,x2,x3,x4,res) & :- & !b1, & b2, & res = x2 + x3. \\ \mathsf{R}(b1,b2,x1,x2,x3,x4,res) & :- & !b1, & !b2, & res = x2 + x4. \end{array}$$

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 engines. These engines 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 engine but develop a new engine specific to the newly designed frontend. This allows for great flexibility in the frontend's design.

For example, Flix [Madsen et al. 2016] 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 [Madsen and Lhoták 2020]. Formulog [Bembenek et al. 2020] 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 engines and do not benefit from prior engineering efforts.

Datafun [Arntzenius and Krishnaswami 2016] 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 engine 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 [Arntzenius and Krishnaswami 2020], even though it has been the standard bottom-up evaluation model for a long time [Ullman 1989].

Our approach: Frontend compilation. We would like to achieve the best of both prior approaches: Build on top of existing Datalog engines 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 engines 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 chapter, we show how a first-order functional language (Section 4.3) with algebraic data types (Section 4.4), and sets (Section 4.6) 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.

4.3 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 of this chapter, here we revisit the problem with a more involved example before presenting our solution.

4.3.1 Compilation by Example

In this chapter 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:


```
1 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 $f : (T_1, \ldots, T_n) \to T$ can be encoded as relations $f : (T_1, \ldots, T_n, T)$. 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 $\mathsf{fact} : (Int, Int)$. 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.

$$\begin{array}{lll} \mathsf{fact}(n, out) & :- & n = 0,\ out = 1. \\ \mathsf{fact}(n, out) & :- & n\ != 0,\ \mathsf{fact}(n-1, out'),\ out = n * out'. \end{array}$$

Unfortunately, like in the introduction of this chapter, 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 [Scholz et al. 2015] 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:

$$\begin{array}{lll} \mathsf{run\_fact}(out) & \text{:-} & \mathsf{fact}(5, out). \\ \mathsf{fact}(n, out) & \text{:-} & \mathsf{fact\_input}(n),\ n = 0,\ out = 1. \\ \mathsf{fact}(n, out) & \text{:-} & \mathsf{fact\_input}(n),\ n\ !\!= 0,\ \mathsf{fact}(n-1, out'),\ out = n * out'. \end{array}$$

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 [Beeri and Ramakrishnan 1991]. 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* [Tekle and Liu 2010] and we use that name in the remainder of the chapter.

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:

$$\begin{array}{lll} \mathsf{fact\_input}(5). & & \\ \mathsf{fact\_input}(n-1) & \text{:-} & \mathsf{fact\_input}(n),\ n\ !\!= 0. \end{array}$$

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

(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	$::= \mathsf{base}$
(types)	T	$::= \mathsf{Base}$

Figure 4.3: Functional IncA with first-order functions, base values, and base functions φ.

constraints leading up to the call. Together, these two rules describe the required relation $\mathsf{fact\_input} = \{5, 4, 3, 2, 1, 0\}$. Since fact_input is finite, fact is finite and contains the following tuples: $\mathsf{fact} = \{(5, 120), (4, 24), (3, 6), (2, 2), (1, 1), (0, 1)\}$.

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:

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

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

$$\begin{array}{lll} \mathsf{fact\_input}(n) & \texttt{:-} & \mathsf{run\_fact\_input}(n). \\ \mathsf{fact\_input}(n-1) & \texttt{:-} & \mathsf{fact\_input}(n),\ n \mathrel{!}= 0. \end{array}$$

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:

$$\mathsf{run\_fact\_input}(n) \quad \texttt{:-} \quad \mathsf{ext\_run\_fact\_input}(n).$$

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.

4.3.2 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.

$$
\begin{array}{lll}
\text{(Datalog programs)} & D & ::= \overline{r} \\
\text{(rules)} & r & ::= \mathrm{R}(\overline{t}) \text{ :- } \overline{a}. \\
\text{(atoms)} & a & ::= t = t \mid \mathrm{R}(\overline{t}) \\
\text{(terms)} & t & ::= v \mid x \mid \varphi(\overline{t}) \\
\text{(values)} & v & ::= \mathsf{base}
\end{array}
$$

Figure 4.4: An intermediate representation for Datalog with base values and base functions.

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

Figure 4.5: Compiling expressions yields a set of alternative terms, each guarded by constraints.

Figure 4.3 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 4.4. This Datalog IR is compatible with many existing Datalog engines, 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 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 4.5 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.

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

Figure 4.6: Compiling functions to Datalog rules.

We use the translation of expressions $[\![.]\!]$ to compile function definitions $[\![.]\!]_{fun}$ and programs $[\![.]\!]_{prog}$ as shown in Figure 4.6. 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.

4

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 4 (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 5 (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 4. □

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 $\mathsf{f}(x, y)$:- $\mathsf{f}(x, y)$. While function call `f(1)` diverges, query $\mathsf{f}(1, y)$ terminates and yields the empty substitution. However, Conjecture 1 ensures terminating function calls translate to terminating Datalog programs under top-down evaluation.

4

4.3.3 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 engines exhaustively enumerate all derivable tuples, starting from known facts. For example, the bottom-up evaluation of the factorial function will start with $\mathsf{fact}(0, 1)$, from which it can derive $\mathsf{fact}(1, 1)$, $\mathsf{fact}(2, 2)$, $\mathsf{fact}(3, 6)$, $\mathsf{fact}(4, 24)$, 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 engines 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 [Tekle and Liu 2010]. 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 ($@\mathsf{main}\ \mathsf{def}\ g(x_1, \ldots, x_n) = \ldots$) 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 ($\mathsf{def}\ g(\ldots) = e$) 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 Section 4.3.1. Formally, we obtain a rewritten Datalog program $guarded(D)$:

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

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

  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_input_s to implement Principle 4 and Principle 5 from Section 4.3.1. 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 ($@\mathsf{main}\ \mathsf{def}\ g(x_1,\ldots,x_n) = \ldots$) in p, we obtain a rule

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

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

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

  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:

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

This program compiles to the following Datalog rules using the translation from Section 4.3.2:

$$\begin{aligned}
\mathsf{fib}(n, y3) &\ \text{:-}\ n < 2 = \mathsf{true},\ y3 = n.\\
\mathsf{fib}(n, y4) &\ \text{:-}\ n < 2 = \mathsf{false},\ \mathsf{fib}(n-1, y1),\ \mathsf{fib}(n-2, y2),\ y4 = y1 + y2.\\
\mathsf{run}(x, y5) &\ \text{:-}\ \mathsf{fib}(x, y5).
\end{aligned}$$

We now apply our demand transformation. First, we derive demand patterns of the program, which are $\langle \mathsf{run}, bf \rangle$ and $\langle \mathsf{fib}, bf \rangle$. 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:

$$\begin{array}{lll} \mathsf{fib}(n, y3) & :- & \mathsf{fib\_input\_bf}(n),\ n < 2 = \mathsf{true},\ y3 = n. \\ \mathsf{fib}(n, y4) & :- & \mathsf{fib\_input\_bf}(n),\ n < 2 = \mathsf{false},\ \mathsf{fib}(n-1, y1),\ \mathsf{fib}(n-2, y2),\ y4 = y1 + y2. \\ \mathsf{run}(x, y5) & :- & \mathsf{run\_input\_bf}(x),\ \mathsf{fib}(x, y5). \end{array}$$

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

$$\begin{array}{lll} \mathsf{run\_input\_bf}(x) & :- & \mathsf{ext\_run\_input\_bf}(x). \\ \mathsf{fib\_input\_bf}(x) & :- & \mathsf{run\_input\_bf}(x). \\ \mathsf{fib\_input\_bf}(n-1) & :- & \mathsf{fib\_input\_bf}(n),\ n < 2 = \mathsf{false}. \\ \mathsf{fib\_input\_bf}(n-2) & :- & \mathsf{fib\_input\_bf}(n),\ n < 2 = \mathsf{false},\ \mathsf{fib}(n-1, y1). \end{array}$$

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 engines.

Correctness The demand transformation yields a Datalog program that derives the exact same tuples as a top-down evaluation [Tekle and Liu 2010]. As of Conjecture 1, top-down evaluation yields the correct tuples. Hence, so does bottom-up evaluation of the demand-driven Datalog rules:

Corollary 1 (Bottom-up translation correctness)**.** Given a functional program p with a main function f such that $f(\bar{v})$ evaluates to w. Then the bottom-up evaluation of the Datalog program $\mathit{demanded}(\llbracket p \rrbracket_{\mathit{prog}})$ yields a database in which the query $f(\bar{v}, x)$ has a single match $\{f(\bar{v}, w)\}$.

4.4 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 that existing bottom-up Datalog engines can execute.

4.4.1 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 an example, consider the Peano numbers:

```
1 data Nat = Zero() | Succ(Nat)
2
3 def plus(m: Nat, n: Nat): Nat = m match {
4   case Zero() => n
5   case Succ(pred) => Succ(plus(pred, n))
6 }
7
8 @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 4.6.

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 engine 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 engine that supports user-defined built-in functions, including Soufflé, Flix, and Formulog. Alternatively, if a Datalog engine 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 4.6.

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

Constructor Relations:			**Selector Relations:**			**Instance Relation:**		
Zero(n)	:-	n = #Zero().	un_Zero(n)	:-	Zero(n).	Nat(n)	:-	Zero(n).
Succ(p,n)	:-	n = #Succ(p).	un_Succ(n,p)	:-	Succ(p,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.

```
Zero(n)                 :- n = #Zero().
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).

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 4.7: Compilation result for the `plus` and `twice` functions on Peano numbers.

Figure 4.7 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 $\langle \text{Zero}, f \rangle$ 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. Note that we omit the selector relations un_Zero/un_Succ, and the instance relation Nat, which we are already defined previously.

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())). 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 4.7. 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 $\text{IncA}_{\text{Scala}}$, we use the diffing algorithm *truediff* [Erdweg et al. 2021] to derive edit scripts for the provided input and populate the extensional constructor relations as shown in Section 3.5. In the case of Soufflé, we insert tuples containing algebraic data and literal values of the Soufflé language in the extensional constructor relations.

4.4.2 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.

$$
\begin{array}{lll}
\text{(Functional programs)} & \mathit{prog} & ::= \overline{F}, \overline{d} \\
\text{(data definitions)} & d & ::= \texttt{data}\ N = \overline{c(T, \ldots, T)} \\
\text{(expressions)} & e & ::= \ldots \mid c(\overline{e}) \mid e\ \texttt{match}\ \{\overline{\texttt{case}\, c(x, \ldots, x) \texttt{=>} e}\} \\
\text{(types)} & T & ::= \ldots \mid N
\end{array}
$$

Figure 4.8: Extending the frontend syntax with algebraic data types.

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

$$
[\![c(e_1, \ldots, e_n)]\!] = \left\{ (y, \{c(t_1, \ldots, t_n, y)\} \cup a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in [\![e_1]\!], \ldots, (t_n, a_n) \in [\![e_n]\!] \right\}
$$
where y is fresh

$$
[\![e\ \texttt{match}\ \{\overline{cs}\}]\!] = \bigcup_{(\texttt{case}\, c(\overline{x}) \texttt{=>} e') \in \overline{cs}} \{(t', \{\texttt{un\_}c(t, \overline{x})\} \cup a \cup a') \mid (t, a) \in [\![e]\!], (t', a') \in [\![e']\!]\}
$$

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

Figure 4.9: Translating algebraic data types to Datalog.

Figure 4.8 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 Section 4.3.2 from functional code to Datalog code to handle algebraic data types as shown in Figure 4.9. 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.

Next, we extend the demand transformation from Section 4.3.3 to consider constructors:

- In Step 1, when considering reachable subexpressions $h(e_1, \ldots, e_n)$, we also generate a demand pattern $\langle h, b \ldots b f \rangle$ 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 $guarded(D)$.
- 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 engines.

4.5 Case Study: Typing, 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 4.10. These functions compile to complex Datalog code that could not practically be written by hand.

Figure 4.10 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)`:

$$\begin{aligned}\mathsf{typeOf}(ctx, exp, out0) \;\; \text{:-} \;\; & \mathsf{typeOf\_input\_bbf}(ctx, exp),\ \mathsf{un\_App}(exp, fun, arg),\\ & \mathsf{typeOf}(ctx, fun, o1),\ \mathsf{un\_JustType}(o1, funty),\\ & \mathsf{un\_TFun}(funty, ty1, ty2),\ \mathsf{typeOf}(ctx, arg, o2),\\ & \mathsf{un\_JustType}(o2, argty),\ \mathsf{eqType}(argty, ty1, o3),\\ & o3 = \mathsf{true},\ \mathsf{JustType}(ty2, out0).\end{aligned}$$

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_bbf. 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

```
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 4.10: A type checker, type erasure, and interpreter for a lambda calculus with numbers.

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, we have shown in Chapter 2 how to compile typing rules to Datalog to derive incremental type checkers systematically. Note 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.

4.6 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, 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.

```
data Exp = ...
data Stm = Assign(String, Exp) | Sequence(Stm, Stm) | If(Exp, Stm, Stm) |
  While(Exp, Stm)

// a regular function that finds the statement's entry
def init(stm: Stm): Stm = ...

// finds all of the statement's exits
def final(stm: Stm): Set[Stm] = stm match {
  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 4.1 (Introduction)
```

Figure 4.11: Computing the control-flow graph as a set of tuples in out extended Datalog frontend.

4.6.1 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 `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. [Nielson et al. 1999] 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 4.11. 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 4.1. 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.

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

Figure 4.12: Extended abstract syntax with set and set operations.

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 chapter and refrain from repeating them here. Instead, we show how to translate functional programs with sets and tuples to Datalog.

4.6.2 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 $\mathsf{foo}(T_1,\ldots,T_n,U_1,\ldots,U_m)$, and a function call `foo(e)` becomes $\mathsf{foo}(t_1,\ldots,t_n,u_1,\ldots,u_m)$, where `e` translates to n terms $t_1,\ldots,t_n$ and the function call yields m result terms $u_1,\ldots,u_m$. 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 4.12 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 4.4. 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.

$$
\begin{aligned}
[\![\{\overline{e}\}]\!] &= \textstyle\bigcup_{e \in \overline{e}} [\![e]\!] \\
[\![s_1 \mathbin{+\!\!+} s_2]\!] &= [\![s_1]\!] \cup [\![s_2]\!] \\
[\![\{e \mid p_1, \ldots, p_n\}]\!] &= \{(t, \{t_1 = \mathsf{true}, \ldots, t_n = \mathsf{true}\} \cup a \cup a_1 \cup \ldots \cup a_n) \\
&\quad \mid (t,a) \in [\![e]\!], (t_1, a_1) \in [\![p_1]\!]_{pred}, \ldots, (t_n, a_n) \in [\![p_n]\!]_{pred}\} \\
[\![\mathsf{fold}(f_{init}, f_{op}, f_{set})]\!] &= \{(\mathsf{aggregate}(f_{set}, \mathsf{toPrimitiveFun}(f_{init}), \mathsf{toPrimitiveFun}(f_{op})), \emptyset)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![e]\!]_{pred} &= [\![e]\!] \\
[\![e \text{ in } s]\!]_{pred} &= \{(\mathsf{true}, \{t_1 = t_2\} \cup a_1 \cup a_2 \mid (t_1, a_1) \in [\![e]\!], (t_2, a_2) \in [\![s]\!]\} \\
[\![e \text{ in } N]\!]_{pred} &= \{(\mathsf{true}, \{N(t)\} \cup a \mid (t,a) \in [\![e]\!]\}
\end{aligned}
$$

Figure 4.13: Compiling sets and set operations to Datalog.

We extend $[\![.]\!]$ to also handle set expressions s, and we add a translation function $[\![.]\!]_{pred}$ for predicates. Figure 4.13 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.

4.6.3 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 [Reynolds 1998] 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. We show how we extend functional IncA's syntax with first-class functions below:

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

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.

4.7 Case Studies: Data-Flow Analysis & 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.

4.7.1 Data-Flow Analyses

Figure 4.14 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.

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 4.15 and Figure 4.2. 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

```
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 4.14: 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 4.2 (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 4.15: Interval analysis of the While language using abstract interpretation.

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

```
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 4.16: Clone detection of Shimple Code.

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.

4.7.2 Clone Detection

Figure 4.16 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 [Vallee-Rai and Hendren 1998] in SSA form. To access the Shimple representation, we extend functional IncA to read Soufflé relations, because the Doop framework [Bravenboer and Smaragdakis 2009b] 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 [Bravenboer and Smaragdakis 2009b], 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 [Erdweg et al. 2021].

4.8 Implementation and Performance Evaluation

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

4.8.1 Implementation

We implemented functional IncA by compiling it to a Datalog IR provided by the $\text{IncA}_{\text{Scala}}$ framework. The Datalog IR can target two different backends namely IncA and Soufflé without any change to the underlying Datalog engines ViatraQuery [Varró et al. 2016] and Soufflé [Scholz et al. 2015] respectively. Our compiler generates Datalog code as shown in this chapter, 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 engines. In particular, the ViatraQuery Datalog engine 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 Section 4.7.1 incrementally by diffing the input programs and feeding the resulting patch to IncA [Erdweg et al. 2021]. Targeting Soufflé allows us to generate efficient and scalable C++ programs that run on multi-core machines. However, Soufflé does not support user-defined recursive 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`.

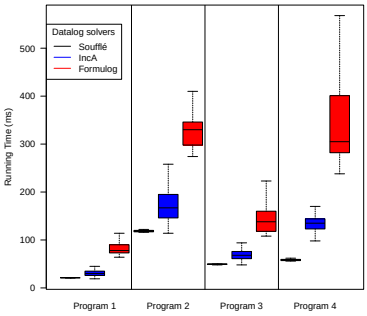
4.8.2 Performance Evaluation

We evaluate the performance of functional IncA and show that it is advantageous to use established engines instead of implementing custom Datalog engines 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. For IncA, we use the Scala2 re-implementation, namely $\text{IncA}_{\text{Scala}}$. 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 engine. Even though IncA uses an incremental Datalog engine ViatraQuery [Varró et al. 2016], 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 engines 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 engine 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 engine for Formulog is slower than the established engines such as Soufflé and IncA for all input programs. The Formulog engine is $\sim 3.7x$ slower than the Soufflé engine and $\sim 2.2x$ slower than the IncA engine. Note that the compiled executable of Soufflé has the fastest running time of all three engines. This shows that is desirable to compile Datalog with functional constructs to already established Datalog dialects instead of implementing custom engines for new Datalog frontends if possible.

4.9 Related Work

We propose functional programming with sets as a frontend for Datalog, which combines functional programming with logic programming. In the remainder of this section, we discuss other approaches combining functional and logic programming without discussing Datalog systems. We compare our approach with other linguistic abstractions for Datalog in Chapter 6.

Mercury [Somogyi et al. 1996] 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 engine 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.

Functional logic programming combines the paradigms of functional programming and logic programming [Hanus 2013]. Curry [Hanus 1997] is one representative of functional logic programming. The language supports features such as higher-order functions originating from functional languages. They also support existentially quantified variables (*logic variables*) and equational constraints from logic programming. Additionally, constraints are first-class, hence higher-order functions can on logic variables and constraints. To support higher-order functions, they rely on defunctionalization. For the functional Datalog frontend, we also rely on defunctionalization to support higher-order functions and relations. To define the semantics of Curry, they follow different approaches, such as defining semantics based on a technique called needed narrowing and translating programs to Prolog. We compile to Datalog, which is a cousin of the logic programming language Prolog. Non-deterministic is a key property of Curry, which differs from our functional Datalog frontend because we compile to Datalog to derive all answers exhaustively.

The Verse calculus [Augustsson et al. 2023] is a core calculus for functional logic programming, which they define as a rewrite system. The calculus contains terms to including logical variables, choice, selecting one choice, selecting all choices, and equality acting as a constraint and not only as an equality test. Additionally, it includes the standard lambda calculus. Therefore, the calculus supports higher-order functions natively. In contrast, our functional Datalog frontend relies on defunctionalization to support higher-order functions because standard Datalog does not support higher-order constructs. One key feature of the calculus is that it is deterministic, which is different from other functional logic languages. That is, selecting a choice will always return the same choice across different runs. This problem does not arise with the functional Datalog frontend, as Datalog derives all choices exhaustively for a given input.

4.10 Chapter Summary

Datalog is supposedly declarative, but many programs are hard to express as constraints. We present a linguistic abstraction for Datalog and closes the representational gap (RG1) Computation. That is, 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 engines rather than building our own. Specifically, we implemented our approach as a frontend for IncA [Szabó et al. 2021] as well as Soufflé [Scholz et al. 2015] 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 engines than implement custom engines 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 [Tekle and Liu 2019] can be used.

5

Interactive Debugging of Datalog Programs

This chapter is based on the peer-reviewed OOPSLA'23 paper "Interactive Debugging of Datalog Programms" [Pacak and Erdweg 2023] and is joint work with Sebastian Erdweg.

Abstract — Datalog is used for complex programming tasks nowadays, consisting of numerous inter-dependent predicates. But Datalog lacks interactive debugging techniques that support the stepwise execution and inspection of the execution state.

In this chapter, we propose an operational abstraction for observing Datalog executions. In particular, we propose interactive debugging of Datalog programs following a top-down evaluation strategy called recursive query/subquery. While the recursive query/subquery approach is well-known in the literature, we are the first to provide a complete programming-language semantics based on it. Specifically, we develop the first small-step operational semantics for top-down Datalog, where subqueries occur as nested intermediate terms. The small-step semantics forms the basis of step-into interactions in the debugger. Moreover, we show how step-over interactions can be realized efficiently based on a hybrid Datalog semantics that adds a bottom-up database to our top-down operational semantics. We implemented a debugger for core Datalog following these semantics and explain how to adopt it for debugging the frontend languages of Soufflé and the functional Datalog frontend of IncA. Our evaluation shows that our hybrid Datalog semantics can be used to debug real-world Datalog programs with realistic workloads.

5.1 Introduction

Datalog is a logic programming language that was invented in the 1980s as a recursive query language for databases [Maier et al. 2018]. However, this is not how Datalog is being used nowadays. In the last 20 years or so, it has become increasingly popular to use Datalog as a programming language to solve all kinds of problems, from program analysis [Bravenboer and Smaragdakis 2009b; Madsen et al. 2016; Szabó et al. 2021] to distributed computing [Abiteboul et al. 2005] and network monitoring [Alvaro et al. 2010a, 2011]. Usages of Datalog usually have two things in common. First, they process graph-structured data of considerable size, which is well supported by Datalog's fixpoint semantics. Second, they involve complex Datalog programs, consisting of many inter-dependent rules. For example, even the simplest analysis from the Doop framework for Java bytecode consists of 560 lines of Datalog code that query 32 relations from the database and compute 78 inter-dependent derived relations [Bravenboer and Smaragdakis 2009b]. Sophisticated Doop analyses are an order of magnitude larger still.

Unfortunately, development tools for Datalog have not been able to keep up with the growing size and complexity of Datalog programs. Specifically, there is no interactive debugging support available for Datalog to date. Interactive debuggers assist developers in program understanding and bug finding by executing a program stepwise and exposing the program's execution state to the developer after each step. Interactive debugging support for Datalog is necessary to support modern Datalog programming. Specifically, there is an increasing number of Datalog programs that involve intricate "control flow" to capture the user's mental model of what happens when. A debugger should enable users to follow the intended control flow of such Datalog programs to trace the program's behavior, as in the following scenarios:

- Type checkers written in Datalog, including derived type checkers presented in Chapter 2 and type checkers implemented using the Datalog dialect Formulog [Bembenek et al. 2020]. Type checkers traverse the syntax tree and the debugger should retain that traversal order. Users need to be able to see the typing context during execution, as well as the set of candidate types in case of local ambiguity.

- Symbolic evaluators [Bembenek et al. 2020] and abstract interpreters implemented in the functional Datalog frontend presented in Chapter 4. Evaluators and interpreters execute (sub-)programs according to the evaluation order of the interpreted language. The debugger must retain that order. The execution state contains the current bindings of variables to values, which are relevant when debugging such interpreters.

- Flow-sensitive data-flow analysis as found in IncA [Szabó et al. 2018, 2021] and Doop [Bravenboer and Smaragdakis 2009b]. A data-flow analysis propagates data-flow facts along the CFG of the analyzed program, either in forward or backward direction. A forward analysis traverses the CFG from function/main entry to function/main exit, a backward analysis is the other way around. The debugger must present the execution in that same order. What is more, data-flow analyses compute a fixpoint and the debugger must enable inspection of the execution state during each fixpoint iteration, not just the final result.

There are existing debugging tools for Datalog, but they treat Datalog as a query language for databases. In particular, existing debugging techniques for Datalog use post-mortem debugging and are based on provenance: They explain why a tuple was derived by computing the tuple's derivation tree. For example, consider the standard Datalog program that finds the paths of a directed graph:

```
edge(1,2).   edge(2,3).   edge(3,1).
path(X,Y) :- edge(X,Y).                  // rule R1
path(X,Y) :- edge(X,Z), path(Z,Y).       // rule R2
```

Given tuple path(3,3), provenance-based approaches will give the following derivation tree as explanation of the derivation of the tuple:

$$\dfrac{\text{edge}(3,1) \quad \dfrac{\text{edge}(1,2) \quad \dfrac{\text{edge}(2,3)}{\text{path}(2,3)}\,\text{R1}}{\text{path}(1,3)}\,\text{R2}}{\text{path}(3,3)}\,\text{R2}$$

Such visualizations can be useful in understanding a Datalog program, in particular for debugging data-driven Datalog programs without control flow. But why does provenance-based debugging not enable adequate debugging of the examples above? Provenance-based debugging does not follow the program's execution trace, but instead follows data-dependencies by using derivation trees. This induces the following shortcomings:

1. The derivation tree only shows why a tuple was derived and does not show the execution state when a specific tuple was derived.
2. Provenance-based debugging can only answer questions about ground tuples. For example, we cannot ask path(3,Y): what paths start at node 3 and why?
3. The derivation tree shows a single derivation, but does not allow exploration of alternative (sub-)derivations that succeeded, nor of alternative (sub-)derivations that failed. That is, the derivation tree does not represent the execution trace of a Datalog program.

This chapter develops interactive debugging for Datalog (with negation and constructors) that allows users to step through a Datalog execution and explore the intermediate execution state. To do so, we must first define what an execution trace for Datalog is. We argue that the standard bottom-up semantics of Datalog is ill-suited for debugging, since it results in a trace that follows the data rather than the structure of the program. Instead, we base our debugger on Datalog's top-down semantics, which starts with a query and steps in and out of rules until all query results have been found. In the literature, this semantics is known as the query/subquery approach, for which iterative and recursive algorithms exist [Vieille 1986]. To the best of our knowledge, we present the first small-step formulation of Datalog's top-down semantics, which forms the foundation of stepwise execution in our debugger. The interactive debugger acts as an operational abstraction for Datalog and closes the representation gap (RG3) Observable Behavior.

One disadvantage of using the top-down semantics for debugging is its inefficiency in practice [Ullman 1989]. This is not much of an issue for individual step-into interactions, but

when a user triggers a step-over or resume interaction, the debugger must run many steps in sequence and the inefficiency becomes a show-stopper. For example, consider a taint analysis implemented in Datalog that constructs a data-flow graph and then propagates taint along its edges. When debugging the taint analysis, a user may want to step over the data-flow graph construction and inspect the taint propagation only. Indeed, one of our experiments confirmed that the top-down semantics is too inefficient to evaluate even a simple Doop points-to analysis on a medium-sized Java codebase (timeout after a 10 minutes). This is why almost all Datalog engines in practice rely on the bottom-up semantics, which can run the same analysis in less than 30 seconds. Technically, the bottom-up semantics is more efficient because it can compute n-ary joins for subqueries, whereas the top-down semantics must execute them in order and perform many binary joins instead. How can we make our debugger scale to real-world Datalog programs nonetheless?

We propose a novel hybrid debugging semantics for Datalog that mixes top-down and bottom-up evaluation. While stepping through the program trace, the user follows the top-down reduction steps. But, when stepping over a predicate call, we rely on the bottom-up semantics to compute the result of the skipped code. In fact, we can run the bottom-up semantics *once* prior to debugging and use the resulting database to provide the results for any number of stepped-over predicates. We further show how this approach can be extended to support stepping over recursive predicate calls, which may only produce a partial result in accordance with the current recursion depth. To this end, we exploit an incremental bottom-up semantics to temporarily "forget" tuples that will only be derived in later iterations.

We implemented two interactive Datalog debuggers, respectively based on the top-down and hybrid semantics. We evaluate the step-into and step-over performance of the debuggers on the path program and on an inter-procedural Java points-to analysis from Doop for realistic debugging scenarios. Our measurements show that the step-into performance is good, but the step-over interaction is a bottleneck in the top-down-only semantics. But using our hybrid semantics, the Datalog debugger scales to realistic workloads. We make the following contributions:

- We present the first formulation of a small-step operational semantics for Datalog. Our semantics corresponds to the well-known recursive query/subquery algorithm (Section 5.3).
- We present a novel hybrid semantics for Datalog that mixes top-down and bottom-up evaluation. A debugger can use top-down reductions for step-into and bottom-up results for step-over interactions (Section 5.4).
- We extend the hybrid semantics to allow step-over of recursive predicate calls through incremental maintenance of a logical relation between the two semantics (Section 5.5).
- We have implemented a debugger for core Datalog and show it can be used to debug languages that compile to core Datalog (Section 5.6).
- We evaluate the performance of our debugging approach on an inter-procedural points-to analysis on realistic workloads (Section 5.7).

5.2 Why we need Interactive Debugging for Datalog

This chapter proposes an interactive debugging approach for Datalog programs, where users can explore and guide the execution of Datalog code. Our goal is to provide a debugger interface for Datalog that mirrors debuggers from imperative programming languages, featuring step into, step over, step out, resume, and breakpoints. In contrast, today's state-of-the-art Datalog debuggers support post-mortem debugging, where users can inspect derivation trees after tuples have been derived. This technique is known as provenance-based debugging. In this section, we discuss why provenance-based debugging is not sufficient for Datalog and why we need interactive debugging support for Datalog instead.

Why not to use derivation trees. Consider an idealized version of the encoding of type checking in Datalog as described in Chapter 2. We encoded the typing relation with two Datalog relations:

$$\mathsf{typeOf} \subseteq \mathrm{Exp} \times \mathrm{Type}$$
$$\mathsf{lookup} \subseteq \mathrm{Exp} \times \mathrm{Name} \times \mathrm{Type}$$

Relation typeOf assigns types to expressions, but does not carry a typing context. Instead, typeOf relies on lookup to resolve variable references, and lookup proceeds in reverse environment-passing style: it starts at the reference and walks up the tree until it finds a corresponding declaration using the auxiliary relation parent between AST nodes. This avoids the construction of typing contexts at Datalog run time, which has performance benefits as discussed in Section 2.4.

We consider a buggy Datalog program consisting of the rules for typeOf shown below:

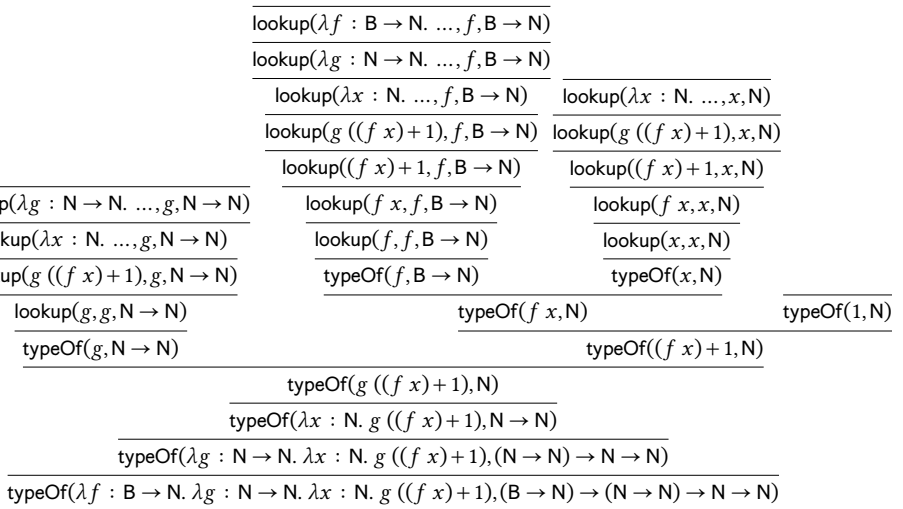
$$\begin{array}{lll}
\mathsf{typeOf}(e, T) & \text{:-} & \mathsf{var}(e, x), \mathsf{lookup}(e, x, T). \\
\mathsf{typeOf}(e, \mathsf{Bool}) & \text{:-} & \mathsf{bool}(e, _). \\
\mathsf{typeOf}(e, \mathsf{Nat}) & \text{:-} & \mathsf{num}(e, _). \\
\mathsf{typeOf}(e, \mathsf{Nat}) & \text{:-} & \mathsf{add}(e, e1, e2), \mathsf{typeOf}(e1, \mathsf{Nat}), \mathsf{typeOf}(e2, \mathsf{Nat}). \\
\mathsf{typeOf}(e, T \to T') & \text{:-} & \mathsf{lam}(e, x, T, b), \mathsf{typeOf}(b, T'). \\
\mathsf{typeOf}(e, T') & \text{:-} & \mathsf{app}(e, e1, e2), \mathsf{typeOf}(e1, T \to T'), \mathsf{typeOf}(e2, \mathsf{Nat}).
\end{array}$$

The relations var, bool, num, add, lam, and app are extensional relations encoding the input program. Note that this Datalog program constructs and destructs data in the form of types such as Nat, Bool and function types $T \to T$. This is unproblematic as long as the constructed data is bound by extensional relations. We omit the encoding of the lookup relation for brevity.

We injected a bug into the typing rules above and will reveal it shortly. But first, consider the following term, which is accepted by the above rules even though it is not well-typed:

$$\lambda f : \mathsf{Bool} \to \mathsf{Nat}.\ \lambda g : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda x : \mathsf{Nat}.\ g\ ((f\ x) + 1)$$

We can try to debug our Datalog program using provenance-based debugging, which allows us to inspect the derivation tree. We need to inspect each node until we find an invalid derivation step, which indicates a buggy Datalog rule. We encourage the reader to try to find the buggy Datalog rule using the derivation tree before reading further.



5

We abbreviate the derivation tree. For example, we do not show the queries of extensional relations such as lam and app. We also abbreviate Nat with N and Bool with B. We highlight some statistics about the complete tree:

typeOf derivations	=	10
lookup derivations	=	16
extensional derivations	=	29

Even for this small expression, the derivation tree is quite large and unwieldy containing 55 nodes. This makes debugging Datalog programs using provenance-based approaches unwieldy. That is, navigating and inspecting derivation trees is clunky.

The buggy Datalog rule is the one that handles typing function applications. The rule requires the argument to have type Nat, independent of the function's parameter type. Using provenance-based debugging, it can be very difficult to find such invalid rules. In particular, considering that Datalog programs are usually applied to large amounts of data (e.g., analyzing the entire JDK), yielding derivation trees that are many orders of magnitude larger. But this is not the only reason we need interactive Datalog debugging.

Our buggy Datalog rule for applications not only permits ill-typed terms, it also prohibits well-typed ones. Consider the following well-typed program p, for which typeOf fails to provide a derivation tree:

$$p = (\lambda f : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\ f\ y)(\lambda x : \mathsf{Nat}.x + 1)$$

Again, the problem is that the rule for applications requires function arguments of type Nat, which is not the case here. Consequentially, there is no derivation tree that provenance-based debugging could provide to the user.

Why we need interactive debugging. Interactive debugging provides a well-known debugging interface to Datalog developers: breakpoints, resume, and stepping. Interactive debugging also allows developers to inspect the internal state of the Datalog program, such

as the bindings of logical variables. The starting point of an interactive debugging session is a query that the developer would like to be answered, much like a unit test. And it does not matter whether the query is derivable or not: The user observes the progress of the Datalog engine interactively until it completes or fails.

Let's use interactive debugging on our Datalog typing rules using the same input program p from above. To find the bug, we set a breakpoint at the beginning of the application rule because we suspect an issue here. We start the session given the query typeOf(p, T). The first breakpoint we reach is for the outermost application with the following bindings:

$$e1 \mapsto \lambda f : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\ f\ y \qquad e2 \mapsto \lambda x : \mathsf{Nat}.\ x + 1$$

We resume the computation and we reach the breakpoint again for the innermost application $f\ y$. We can step over the first typeOf atom or inspect its derivation using step into. After this atom, we obtain the bindings:

$$e1 \mapsto f \qquad e2 \mapsto y \qquad T \mapsto \mathsf{Nat} \qquad T' \mapsto \mathsf{Nat}$$

We can step into the second typeOf call to observe how lookup validates that variable y indeed has type Nat. Since all atoms of the application succeeded, we obtain a typeOf tuple for $f\ y$. We step out until we reach the rule processing the outermost application again, yielding the bindings:

$$\begin{array}{lcl} e1 & \mapsto & \lambda f : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\ f\ y \\ e2 & \mapsto & \lambda x : \mathsf{Nat}.\ x + 1 \\ T & \mapsto & \mathsf{Nat} \\ T' & \mapsto & \mathsf{Nat} \end{array}$$

We step into the second typeOf call, which yields a query typeOf$(\lambda x : \mathsf{Nat}.\ x + 1, \mathsf{Nat})$. By stepping further, we can observe that none of the typeOf rules can derive this query. In particular, the rule for lambda abstractions fails because the requested type Nat is not a function type. With this information, the user can discover the bug in the application rule, which should not have requested type Nat for arguments unconditionally.

Interactive debugging enables inspecting each step while determining if a tuple is derivable. In contrast, derivation trees only show the final result. This shows why interactive debugging is necessary: To manage the complexity of derivations using breakpoints, resume, step over, and step into. And to trace the logical reasoning, whether it succeeded or failed to derive a tuple.

5.3 Small-Step Semantics for Top-Down Datalog

In this chapter, we propose to use top-down evaluation as a debugging semantics for Datalog programs. The top-down semantics is well-suited for debugging because it starts with a Datalog query issued by the user. Given a query, the top-down semantics recursively explores the rules of the Datalog program to satisfy the query and to derive the matching tuples. That is, the top-down semantics is goal-directed, while the bottom-up semantics

(Datalog programs)	D	$::= \overline{r}, \overline{f}$
(rules)	r	$::= p(\overline{X})$:- $\overline{a}$.
(atoms)	a	$::= p^s(\overline{t}) \mid \mathsf{edb}\ p^s(\overline{t}) \mid t = t \mid t \neq t$
(signs)	s	$::= + \mid -$
(terms)	t	$::= c \mid X$
(facts)	f	$::= p(\overline{c})$.
(constants)	c	
(variables)	X	

Figure 5.1: Abstract syntax of Datalog's surface language.

(value tables)	v	$::= \text{table}(\overline{X}, \overline{T})$
(tuples)	T	$::= \overline{c}$
(rules)	r	$::= \ldots \mid v$
(atoms)	a	$::= \ldots \mid Q$
(queries)	Q	$::= \mathbf{sq}(p^s(\overline{t}), v, v, v, r \vee \ldots \vee r) \mid v^s$

Figure 5.2: Abstract syntax of Datalog's intermediate terms.

is data-driven and populates tables eagerly. Since both semantics yield the exact same result [Green et al. 2013], the top-down semantics can safely be used for debugging even for systems that implement the bottom-up semantics.

The basis of our small-step semantics is the standard recursive query/subquery approach (QSQR) developed by Vieille [1986]. While this approach is well-documented in the literature [Abiteboul et al. 1995], its original formulation was soon found to be incomplete [Vieille 1987; Nejdl 1987]. The first complete QSQR semantics that provably finds all derivable tuples was proposed much later by Madalinska-Bugaj and Nguyen [2008]. Our semantics follows Madalinska-Bugaj and Nguyen and, in particular, their Remark 3.2 explaining that active queries can be maintained in a call stack.

In this section, we present the first small-step top-down Datalog semantics. Prior top-down Datalog semantics are ill-suited for debugging for three reasons. First, existing QSQR algorithms apply rules in a single big step, rather than stepping through their bodies. Second, they apply many rules simultaneously to compute a complete predicate, rather than considering one rule at a time to trace a predicate's growth. Third, they are presented as pseudo code that leaves many technical details implicit, such as how to modify tables using relational algebra. In contrast, we reformulate QSQR as a small-step operational semantics that makes all details explicit. This semantics will not only serve as the basis for our debugger, but represents an important semantic artifact that will help substantiate programming-language research on Datalog.

5.3.1 Datalog Abstract Syntax

Before we discuss the small-step operational semantics, we introduce the abstract syntax of Datalog's surface language formally in Figure 5.1. A Datalog program consists of a

collection of Datalog rules $\overline{r}$ describing the intensional database (IDB) and collection of facts $\overline{f}$ such as edge(1,2). describing the extensional database (EDB). We assume that the EDB is finite. A rule $p(\overline{X})$:- $\overline{a}$. consists of a rule head $p(\overline{X})$ and a rule body $\overline{a}$. The rule head $p(\overline{X})$ names the predicate the rule belongs to and declares columns, whose bindings will determine the derived tuples. The rule body is a sequence of atoms. An atom is either an (intensional) predicate call $p^s(\overline{t})$, extensional predicate call edb $p^s(\overline{t})$, an equality constraint $t = t'$, or an inequality constraint $t \neq t'$. We consider Datalog with stratified negation in this chapter, which is why predicate calls have sign annotations s: A positive sign (+) indicates a regular predicate call, whereas a negative sign (−) indicates a negated call. Terms that occur inside atoms are either a constant value or a variable.

Note that a rule head only ranges over columns and not constants. Additionally, we only allow linear patterns in rule heads (no duplicate variable names) and assume rules belonging to the same predicate use the same column names. We make these assumptions without loss of generality as we can easily normalize arbitrary rules as the following example illustrates:

$$\begin{array}{ll} p(X,1) \text{ :- } \overline{a_1}. & \\ p(X,A) \text{ :- } \overline{a_2}. & \rightarrow \quad p(X,Y) \text{ :- } \overline{a_2}, Y = A. \\ p(X,X) \text{ :- } \overline{a_3}. & \end{array} \qquad \begin{array}{l} \\ \\ p(X,Y) \text{ :- } \overline{a_1}, Y = 1. \\ \\ p(X,Y) \text{ :- } \overline{a_3}, Y = X. \end{array}$$

Additionally, we only consider range-restricted Datalog programs. Range-restricted programs are Datalog programs where every Datalog rule adheres to the following rule: Every variable appearing in the head of a Datalog rule is positively bound within the body of the rule. This is not a prohibiting restriction because all state-of-the-art Datalog engines require range-restriction.

The abstract syntax describes the surface language of Datalog. To formalize the small-step operational semantics of Datalog, we extend the abstract syntax with intermediate terms that only occur during evaluation and are not available to Datalog programmers. We summarize the required intermediate terms in Figure 5.2. First, rules evaluate to value tables v, which we add as an alternative to r. A value table table$(\overline{X}, \overline{T})$ consists of a list of column names $\overline{X}$ and a sequence of tuples $\overline{T}$, each of which is a sequence of constants. For readability, we usually denote the content of a table as a set of actual tuples, like $\{(1,2),(2,3)\}$. A table is only well-formed if the column names are mutually different and the number of columns matches the arity of all contained tuples. Note that table$(\overline{X}, \emptyset)$ represents the empty table for any columns $\overline{X}$, whereas table$(\epsilon, \{()\})$ represents the unit table: a table without columns but with a single row, namely the empty tuple. The empty table is an absorbing element for natural joins and represents a failing computation. In contrast, the unit table is the neutral element for natural joins, which we use to represent computations that do not add bindings to the current environment:

$$\begin{array}{rcccl} \text{table}(\overline{X}, \emptyset) \bowtie v & = & \text{table}(\overline{X}, \emptyset) & = & v \bowtie \text{table}(\overline{X}, \emptyset) \\ \text{table}(\epsilon, \{()\}) \bowtie v & = & v & = & v \bowtie \text{table}(\epsilon, \{()\}) \end{array}$$

The most interesting intermediate term is the one for subqueries. Subqueries Q occur when a predicate call $p^s(\overline{t})$ is reduced, which spawns a new subquery. Following the recursive query/subquery approach QSQR, a subquery runs until it reaches a fixpoint. This may require multiple fixpoint iterations, which is why the subquery must manage quite a

Reduction Relations:		**Global Information:**	
(rule reduction)	$v \vdash r \ \rightarrow^{R} \ r \dashv v$	(extensional database)	$EDB \ \in p \rightarrow v$
(atom reduction)	$v \vdash a \ \rightarrow^{A} \ Q$	(intensional database)	$IDB \ \in p \rightarrow v$
(query reduction)	$Q \ \rightarrow^{Q} \ Q$	(active queries)	$\Gamma \quad \in p_\alpha \rightarrow \bar{v}$

Figure 5.3: Reduction relations for top-down Datalog and the global state they interact with.

bit of auxiliary state. Specifically, a subquery $\mathbf{sq}(p^s(\bar{t}), v_a, v_r, v_{sup}, r_1 \vee \ldots \vee r_n)$ consists of five components:

1. The original predicate call $p^s(\bar{t})$ to compute the bindings of free variables in $\bar{t}$ once the subquery reaches a fixpoint and terminates,
2. the value table v_a that captures the arguments of the subquery,
3. the subquery result v_r for the rules already evaluated,
4. the supplementary table v_{sup} for the intermediate result of currently evaluating rule, and
5. the current and remaining rules $r_1 \vee \ldots \vee r_n$.

The role of each component will become clearer when we discuss the semantics of subqueries. Eventually, a subquery evaluates to a value table v^s that carries the sign of the original predicate call. Note that all value tables are implicitly positive whenever we omit the sign annotation.

5.3.2 Reduction Relations and Global State

We model the small-step operational semantics of top-down Datalog through three mutually recursive reduction relations. Figure 5.3 shows the signatures of the three relations. Rule reduction $v_{sup} \vdash r \ \rightarrow^{R} \ r' \dashv v'_{sup}$ of a rule r reduces and eventually eliminates atoms from the rule body. That is, we rewrite rules until all atoms are satisfied and consumed. Rule reduction happens under a supplementary table v_{sup} and produces a new supplementary table v'_{sup}. Supplementary tables are a standard element of top-down Datalog evaluation and roughly correspond to environments in other programming-language semantics. For example, a rule $\mathsf{edge}(X,Y)$:- $X = 1, Y = 2$. may start with a unit supplementary table, then reduce to $\mathsf{edge}(X,Y)$:- $Y = 2$. with a supplementary table$(X,\{(1)\})$, then to $\mathsf{edge}(X,Y)$:- . with table$(X,Y,\{(1,2)\})$. In contrast, to environments, the supplementary table may capture many possible values for a column and evaluation may filter entries that do not satisfy the rule body. For example, the evaluation of rule $\mathsf{edge}(X,Y)$:- $X = 1, Y = 2$. with initial supplementary table$(Y,\{(2),(3),(4)\})$ lists three alternative values for Y. The rule then reduces to $\mathsf{edge}(X,Y)$:- $Y = 2$. with table$(X,Y,\{(1,2),(1,3),(1,4)\})$, and then to $\mathsf{edge}(X,Y)$:- . with table$(X,Y,\{(1,2)\})$ by filtering rows that do not satisfy $Y = 2$.

Rule reduction depends on atom reduction $v_{sup} \vdash a \ \rightarrow^{A} \ Q$ of a under the supplementary table v_{sup}. An atom either reduces to an updated supplementary table or produces a new subquery. For subqueries, query reduction $Q \ \rightarrow^{Q} \ Q'$ is responsible for their execution and fixpoint iteration.

Our semantics uses three pieces of global state, which is shared between all reduction rules and also shown in Figure 5.3. We decided to model this state outside the intermediate terms to obtain a more concise semantics. Alternatively, the global state could be threaded in standard state-passing style. Function *EDB* collects all pre-defined facts $\overline{f}$ of the current Datalog program and represents the extensional database. For each extensional predicate p, *EDB* maps p to a value table v. Function *IDB* collects all tuples derived during the execution of a Datalog program and represents the intensional database. Initially, *IDB* maps all predicates to the empty table with columns matching the predicate's signature. During execution, *IDB* grows monotonically until we reach a fixpoint.

While *EDB* and *IDB* are common to all Datalog semantics, our top-down semantics also must track the active queries Γ. Essentially, Γ maps predicates to value tables that represent active argument values. The idea is to skip active arguments in recursive calls to prevent infinite recursion, while the fixpoint loop still ensures all derivable tuples are found. Technically, Γ must distinguish between different calls based on which parameters are bound at the call site, since the corresponding value tables have different columns. We employ adornments to distinguish bound columns b from free columns f as is standard for top-down Datalog, but also known from the magic set transformation [Beeri and Ramakrishnan 1991]. For example, the call $\mathsf{path}(1,3)$ binds two parameters through adornment bb (bound and bound), but the adornment of call $\mathsf{path}(A,B)$ depends on the boundedness of A and B at the call site. Given a predicate p and an adornment α for each parameter of p, function Γ maps p_α to a stack of currently active query arguments.

5.3.3 Reduction Rules

We now present the reduction rules of the three reduction relations shown in Figure 5.4.

Rule reduction. There are three reduction rules defining the rule reduction relation: R-Step, R-Merge, and R-Result. R-Step is a simple congruence rule that reduces the next atom a of the current rule under the current supplementary table v_{sup}. This way, R-Step iteratively normalizes the frontmost atom, but it does not change the supplementary table.

Rule R-Merge applies when the first atom of a Datalog rule has been reduced to a value table v^s. We remove the value table v^s from the Datalog rule and merge it into the supplementary table v_{sup} using a helper function *merge* defined as follows:

$$merge(v_1, v_2^s) = \begin{cases} v_1 \bowtie v_2, & \text{if } s = + \\ v_1 \rhd v_2, & \text{if } s = - \end{cases}$$

This function combines two value tables based on the sign of the second operand. If the right-hand value table carries a positive sign (i.e., it stems from a positive predicate call), we merge the tables using a natural join. This effectively extends the current supplementary table with bindings for those variables that were free in the call of the predicate. In contrast, if the right-hand value table stems from a negative predicate call and carries a negative sign, we merge the tables using an anti-join. An anti-join $v_1 \rhd v_2$ only retains those rows of v_1 that do *not* have a match in v_2. In particular, an anti-join does not add any bindings to the supplementary table, since all Datalog variables must be bound by positive calls or positive equations.

$$\text{R-Step}\ \frac{v_{sup} \vdash a\ \rightarrow^{A}\ Q}{v_{sup} \vdash p(\overline{X})\ \text{:-}\ a, \overline{as}.\ \rightarrow^{R}\ p(\overline{X})\ \text{:-}\ Q, \overline{as}. \dashv v_{sup}}$$

$$\text{R-Merge}\ \frac{}{v_{sup} \vdash p(\overline{X})\ \text{:-}\ v^{s}, \overline{as}.\ \rightarrow^{R}\ p(\overline{X})\ \text{:-}\ \overline{as}. \dashv merge(v_{sup}, v^{s})}$$

$$\text{R-Result}\ \frac{}{v_{sup} \vdash p(\overline{X})\ \text{:-}\ \epsilon.\ \rightarrow^{R}\ \Pi_{\overline{X}}(v_{sup}) \dashv v_{sup}}$$

$$\text{A-Eq}\ \frac{unfree(t, v_{sup}) \quad unfree(t', v_{sup})}{v_{sup} \vdash t = t'\ \rightarrow^{A}\ \sigma_{t=t'}(v_{sup})} \qquad \text{A-New}\ \frac{unfree(t, v_{sup}) \quad unfree(t', v_{sup})}{v_{sup} \vdash t \neq t'\ \rightarrow^{A}\ \sigma_{t\neq t'}(v_{sup})}$$

$$\text{A-Eq-L}\ \frac{X \notin cols(v_{sup}) \quad unfree(t, v_{sup})}{v_{sup} \vdash X = t\ \rightarrow^{A}\ bind(X, t, v)} \qquad \text{A-Eq-R}\ \frac{X \notin cols(v_{sup}) \quad unfree(t, v_{sup})}{v_{sup} \vdash t = X\ \rightarrow^{A}\ bind(X, t, v)}$$

$$\text{A-EDB}\ \frac{v_a = eval(cols(p), \bar{t}, v_{sup})}{v_{sup} \vdash \mathsf{edb}\ p^{s}(\bar{t})\ \rightarrow^{A}\ \rho_{\bar{t}/cols(p)}(EDB(p) \bowtie v_a)^{s}}$$

$$\text{A-Into}\ \frac{v_a = eval(cols(p), \bar{t}, v_{sup}) \quad (\Gamma', v'_a) = pushQuery(\Gamma, p, v_a) \quad v'_a\ \mathsf{not\ empty}}{v_{sup} \vdash p^{s}(\bar{t})\ \rightarrow^{A}\ \mathbf{sq}(p^{s}(\bar{t}), v'_a, empty(p), v'_a, rules(p))}\ \Gamma := \Gamma'$$

$$\text{A-Skip}\ \frac{v_a = eval(cols(p), \bar{t}, v_{sup}) \quad (\Gamma', v'_a) = pushQuery(\Gamma, p, v_a) \quad v'_a\ \mathsf{empty}}{v_{sup} \vdash p^{s}(\bar{t})\ \rightarrow^{A}\ \rho_{\bar{t}/cols(p)}(IDB(p) \bowtie v_a)^{s}}$$

$$\text{A-Step}\ \frac{Q\ \rightarrow^{Q}\ Q'}{v_{sup} \vdash Q\ \rightarrow^{A}\ Q'}$$

$$\text{Q-Step}\ \frac{v_{sup} \vdash r\ \rightarrow^{R}\ r' \dashv v'_{sup}}{\mathbf{sq}(p^{s}(\bar{t}), v_q, v_r, v_{sup}, r \vee \overline{rs})\ \rightarrow^{Q}\ \mathbf{sq}(p^{s}(\bar{t}), v_q, v_r, v'_{sup}, r' \vee \overline{rs})}$$

$$\text{Q-Union}\ \frac{}{\mathbf{sq}(p^{s}(\bar{t}), v_q, v_r, v_{sup}, v \vee \overline{rs})\ \rightarrow^{Q}\ \mathbf{sq}(p^{s}(\bar{t}), v_q, v_r \cup v, v_q, \overline{rs})}$$

$$\text{Q-Stable}\ \frac{v_r \subseteq IDB(p) \quad (\Gamma', v_a) = popQuery(\Gamma, p, v_q)}{\mathbf{sq}(p^{s}(\bar{t}), v_q, v_r, v_{sup}, \epsilon)\ \rightarrow^{Q}\ \rho_{\bar{t}/cols(p)}(IDB(p) \bowtie v_a)^{s}}\ \Gamma := \Gamma'$$

$$\text{Q-Iterate}\ \frac{v_r \nsubseteq IDB(p) \quad IDB' = IDB \uplus p \mapsto (IDB(p) \cup v_r)}{\mathbf{sq}(p^{s}(\bar{t}), v_q, v_r, v_{sup}, \epsilon)\ \rightarrow^{Q}\ \mathbf{sq}(p^{s}(\bar{t}), v_q, empty(p), v_q, rules(p))}\ IDB := IDB'$$

Figure 5.4: Reduction rules for rule reduction, atom reduction, and query reduction.

At last, when the rule body is empty, R-Result replaces the rule by the value table it produces. We obtain a rule's value table by projecting the final supplementary table according to the parameters in the rule's head. Projecting the head variables of the rule for the supplementary table will always succeed because all Datalog rules adhere to range-restriction. Hence, after fully evaluating a Datalog rule the supplementary table will range over the variables appearing in the head of the Datalog rule.

Atom reduction. The atom reduction relation handles equalities and predicate calls. The rules A-Eq and A-New handle equality and inequality atoms whenever both operands are unfree. A term is unfree with respect to a table v if the term is a constant c or it is a variable X and $X \in cols(v)$. For unfree terms, A-Eq and A-New filter the current supplementary table v_{sup} according to the their constraint. This filtered supplementary table later replaces v_{sup} in the next R-Merge step since $v_{sup} \bowtie \sigma_f(v_{sup}) = \sigma_f(v_{sup})$ for any f. An actual implementation can of course avoid the extra join.

In addition, we define two rules A-Eq-L and A-Eq-R for the equality atom when one term is unfree but the other term is a free variable. These rules bind the free variable in the current supplementary table using the helper function *bind*:

$$bind(X,t,v) = \begin{cases} v \bowtie \text{table}(X,Y,\{(c,c) \mid c \in \Pi_Y(v)\}), & \text{if } X \notin cols(v), t = Y, Y \in cols(v) \\ v \bowtie \text{table}(X,\{(c)\}), & \text{if } X \notin cols(v), t = c \end{cases}$$

The first case is only triggered if the term is a variable Y bound by table v. Effectively, we extend table v with a new column X that is a copy of column Y. The second case is only triggered if the term is a constant. In this case, we extend the table v with a new column X that contains c in each row. Note that A-Eq-R is the same as A-Eq-L but where the right operand is an unbound variable.

The next three rules handle different kinds of predicate calls. Reduction rule A-EDB handles queries against the extensional database *EDB*. To this end, we first compute the argument table v_a of the call using a helper function *eval* to bind the columns of p to the argument values in $\bar{t}$ under v_{sup}:

$$eval(\overline{X},\bar{t},v) = \rho_{\{X_i/t_i \mid i \in I_{var}\}}(\Pi_{\bar{t}_{I_{var}}}(v)) \times \text{table}(\overline{X}_{I_{const}}, \{\bar{t}_{I_{const}}\})$$

Function *eval* takes a sequence of columns, a sequence of terms, and a value table. It utilizes two helper index sets. That is, I_{var} collects all indices of terms in sequence $\bar{t}$ that are variables and are already bound by value table v. The second index set I_{const} collects all indices of terms $\bar{t}$ that are constants. We can apply index sets to sequences to select a subsequence. For example, we apply index set $I = \{1,3,4\}$ to sequence $\overline{X} = A,B,C,D$, which selects $\overline{X}_I = A,C,D$. Using index set I_{var}, we project out all bound columns of v occurring in $\bar{t}$ and afterward rename them such that they are compatible with the head variables of the called predicate p. We produce the final result by building the Cartesian product with a singleton value table containing a tuple enumerating all constants in $\bar{t}$. The singleton value table uses the corresponding head variables of predicate p.

Based on the resulting argument table, rule A-EDB can find the matching tuples in *EDB* using a natural join, yielding a subset of $EDB(p)$. We then need to rename the columns to match the argument terms $\bar{t}$ of the predicate call, dropping columns that are constant in $\bar{t}$.

We also superimpose the sign s of the call, which the subsequent R-Merge will take into account to update the supplementary table accordingly.

Reduction rule A-Into is only applicable if the query produced by $p^s(\bar{t})$ explores new argument tuples. It is important to track the previously visited argument tuples to ensure termination in the presence of recursive programs. For example, the Datalog program introduced in the introduction features a recursively defined predicate path, which could lead to the infinite call chain $\mathsf{path}(1, X)$, $\mathsf{path}(1, X)$, etc. when there is $\mathsf{edge}(1, 1)$. Note that the idea of distinguishing new from previous argument tuples is part of the query/subquery algorithm [Vieille 1986]; the contribution of this section is to reformulate this algorithm as a small-step operational semantics. We add a new query of p with argument table v_a to the global state Γ using helper function *pushQuery*:

$$
\begin{aligned}
&\mathit{pushQuery}(\Gamma, p, v) = (\Gamma', v') \\
&\text{where} \qquad \alpha = \mathit{adorn}(p, v) \\
&\qquad\qquad v_1, \ldots, v_n = \Gamma(p_\alpha) \\
&\qquad\qquad v' = v \setminus (v_1 \cup \cdots \cup v_n) \\
&\qquad\qquad \Gamma' = \Gamma \uplus p_\alpha \mapsto v, v_1, \ldots, v_n
\end{aligned}
$$

This function yields a filtered argument table v' that only contains new arguments (not already present on the stack) and an updated Γ' where we push v onto the stack associated with adorned predicate p_α. If the filtered argument table is non-empty, we create a new subquery with v'_a as argument table, an empty result table which we create by calling the helper function *empty*, and an initial supplementary table v'_a for the first rule of p.

The next reduction rule A-Skip complements A-Into and is only applicable if all argument tuples are already active . In this case, we do not recurse but read the tuples found so far from $\mathit{IDB}(p)$. We join this table with the argument table v_a to obtain only those tuples that match the current arguments. Finally, we have the congruence rule A-Step, which applies query reduction.

Query reduction. The query reduction relation takes care of subqueries that arise from predicate calls in A-Into. Rule Q-Step is a congruence rule that reduces the frontmost rule of the subquery using the rule reduction relation, updating the supplementary table of the subquery accordingly. Once a rule is normalized to a value table v, rule Q-Union adds the rule's result table v to the subquery result v_r. It then resets the supplementary table to the original argument table v_q so that the next rule in $\overline{rs}$ can continue.

The main responsibility of the query reduction relation is to iterate subqueries until all derivable tuples have been computed. To this end, rules Q-Stable and Q-Iterate handle complete subqueries that have no more rules to evaluate. If all tuples in the subquery result v_r have been derived before $v_r \subseteq \mathit{IDB}(p)$, we have found a fixpoint for the query of p with argument table v_q. We then remove the subquery's argument table v_q from Γ using the helper function *popQuery*:

$$
\begin{aligned}
&popQuery(\Gamma, p, v) = (\Gamma', v_1) \\
&\quad \text{where} \qquad \alpha = adorn(p, v) \\
&\qquad\qquad v_1, \ldots, v_n = \Gamma(p_\alpha) \\
&\qquad\qquad \Gamma' = \Gamma \uplus p_\alpha \mapsto v_2, \ldots, v_n
\end{aligned}
$$

Removing v_q from Γ is necessary to allow exploring the subquery again later, when the intensional database IDB may have grown. Resetting Γ is the crucial fix to the QSQR algorithm proposed by Madalinska-Bugaj and Nguyen to guarantee completeness of the semantics. Since v_r did not contain new tuples, Q-Stable simply returns the tuples found in IDB for p and the original argument table v_a from the predicate call. If instead v_r contains new tuples $v_r \not\subseteq IDB(p)$, we must iterate the rules of p since we have not yet found a fixpoint. To this end, Q-Iterate adds the newly found tuples to IDB and re-initializes the subquery: discard the subquery result and restart with all rules of p. Since Datalog can only explore finitely many tuples, the iteration will terminate eventually when no new tuples are being derived.

5

5.3.4 Reduction Trace by Example

We have defined a complete small-step operational semantics for top-down Datalog. Here, we illustrate how the reduction semantics can be used to evaluate the example Datalog program from the introduction, but with additional edges:

$$
\begin{array}{ll}
\mathsf{edge}(1,2). & \mathsf{edge}(2,3). \\
\mathsf{edge}(3,1). & \mathsf{edge}(3,4). \\
\mathsf{path}(X,Y) \text{ :- } & \mathsf{edb}\ \mathsf{edge}(X,Y). \\
\mathsf{path}(X,Y) \text{ :- } & \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).
\end{array}
$$

Top-down evaluation always starts with a query. For example, consider the query $\mathsf{path}(1,Y)$ that finds all nodes Y reachable from 1. We initialize the intensional database map IDB as empty, and initialize active queries Γ with mapping $\mathsf{path}_{bf} \mapsto v_q$ where $v_q = \mathsf{table}(X, \{(1)\})$. Here and in subsequent examples of the chapter, we omit positive signs from predicate calls and table values: these are implicitly positive. The extensional database EDB only contains the mapping

$$\mathsf{edge} \mapsto \mathsf{table}(X, Y, \{(1,2),(2,3),(3,1),(3,4)\})$$

We start with the following subquery:

$$
\mathbf{sq}\left(\mathsf{path}(1,Y), v_q, \emptyset, v_q, \begin{array}{l} \mathsf{path}(X,Y) \text{ :- } \mathsf{edb}\ \mathsf{edge}(X,Y). \\ \mathsf{path}(X,Y) \text{ :- } \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y). \end{array}\right)
$$

After a sequence of Q-Step applications for the first Datalog rule we arrive at the Datalog program where $v = \mathsf{table}(X, Y, \{(1,2)\})$. We reach the following intermediate term:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, \emptyset, v, v \vee \mathsf{path}(X,Y) \text{ :- } \mathsf{edb}\ \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).)$$

Now, Q-Union is applicable hence, we extend the subquery result to $v_r = \emptyset \cup v$, reset the supplementary table to v_q, and discard the first rule:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_q, \mathsf{path}(X,Y) \text{ :- edb } \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).)$$

Again, after a sequence of Q-Step applications we get to $v_{sup} = \mathsf{table}(X,Z,\{(1,2)\})$:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_{sup}, \mathsf{path}(X,Y) \text{ :- } \mathsf{path}(Z,Y).)$$

A-Into extends Γ to $\mathsf{path}_{bf} \mapsto v'_q, v_q$ where $v'_q = \mathsf{table}(X,\{(2)\})$ and replaces the path call with a nested subquery Q:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_{sup}, \mathsf{path}(X,Y) \text{ :- } Q.)$$

$$Q = \mathbf{sq}\left(\mathsf{path}(Z,Y), v'_q, \emptyset, v'_q, \begin{array}{l}\mathsf{path}(X,Y) \text{ :- edb } \mathsf{edge}(X,Y).\\ \mathsf{path}(X,Y) \text{ :- edb } \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).\end{array}\right)$$

While reducing the inner subquery Q, we will eventually encounter a predicate call with the same argument tuples again. At this point A-Skip is applicable and will force termination of the subquery. Hence, the inner subquery is replaced with table $v'' = \mathsf{table}(X,Y,\{(2,3),(2,4),(2,1),(2,2)\})$ to yield the intermediate term:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v_{sup}, \mathsf{path}(X,Y) \text{ :- } v''.)$$

Now we apply R-Merge to yield $v'_{sup} = \mathsf{table}(X,Z,Y,\{(1,2,3),(1,2,4),(1,2,1),(1,2,2)\})$ followed by R-Result to produce the rule result:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v_r, v'_{sup}, \mathsf{table}(X,Y,\{(1,3),(1,4),(1,1),(1,2)\}))$$

Next, Q-Union extends the subquery result and discards the rule result:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, \mathsf{table}(X,Y,\{(1,2),(1,3),(1,4),(1,1)\}), v_q, \epsilon)$$

We derived new tuples hence, Q-Iterate is applicable and will extend the intensional database map IDB. The rule replaces the processed subquery with its initial version:

$$\mathbf{sq}\left(\mathsf{path}(1,Y), v_q, \emptyset, v_q, \begin{array}{l}\mathsf{path}(X,Y) \text{ :- edb } \mathsf{edge}(X,Y).\\ \mathsf{path}(X,Y) \text{ :- edb } \mathsf{edge}(X,Z), \mathsf{path}(Z,Y).\end{array}\right)$$

At this point, the only active query is the initial query $\mathsf{path}(1,Y)$ as all other subqueries stabilized and therefore have been removed from the active queries map Γ by Q-Stable, hence Γ is $\mathsf{path}_{bf} \mapsto v_q$. Therefore, we will take the same steps as in the last iteration, producing the same subquery:

$$\mathbf{sq}(\mathsf{path}(1,Y), v_q, v'_r, v'_{sup}, \epsilon)$$

Hence, Q-Stable triggers as all tuples have already been discovered. The query has been successfully evaluated, there are no active subqueries as Γ is empty and a value table remains:

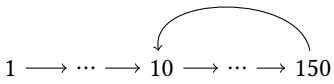
$$\mathsf{table}(X,Y,\{(1,2),(1,3),(1,4),(1,1)\})$$

Now we have shown a reduction trace for a concrete program given a starting query.

5.4 A Hybrid Datalog Semantics

The small-step operational semantics presented in the previous section can be used as a debugging semantics to follow the reduction trace of Datalog programs. The reduction rules describe the steps taken with the *step-into* interaction. This is not sufficient for debugging programs since developers also want to skip sub-computations using the *step-over* interaction. One way of implementing step-over of a predicate call is to step through the predicate using *step-into* until the predicate computation terminates, and show the predicate's result to the user. Unfortunately, this strategy has scalability issues for Datalog: The top-down debugging semantics is not fast enough to simulate predicate computations.

Consider we want to debug the path predicate over the following graph with 150 nodes:



We start the debugging session with query $\mathsf{path}(1,Y)$ and interactively use step-into until we reach query $\mathsf{path}(11,Y)$, which is first inner node of the cycle. To determine the result of $\mathsf{path}(11,Y)$ we want to use a step-over interaction. When we simulate the step-over interaction by repeatedly using step-into until the subquery terminates, it takes roughly 5 minutes to complete debugging of the program. Such long delays are unacceptable for a debugging interaction. Note that the simulated debugging trace consists of 255849 step-into interactions, which take 1-2 ms on average. We will discuss the scalability issue of step-into interactions in more detail in Section 5.7. The problem is not the step-into performance, but the simulation of step-over on top of step-into. Is there a better way to support step-over interactions for Datalog?

We propose to construct a hybrid Datalog semantics that uses top-down stepping for step-into but bottom-up results for step-over. As mentioned before, Datalog systems usually implement a bottom-up semantics, which is more efficient but ill-suited for debugging. The bottom-up semantics computes a database of all derivable tuples. The key idea is to read the predicate result from the bottom-up database when a step over occurs, rather than computing it on demand. Since the bottom-up database contains the tuples for all predicates in the program, we can effectively step over any predicate call in the program. Indeed, when using the bottom-up database to step over $\mathsf{path}(11,Y)$ in our example, the program completes debugging in only 180 ms. In this section, we formalize and exemplify step-over for non-recursive predicate calls, while Section 5.5 extends this idea to support recursive predicate calls.

5.4.1 Efficient Step-Over by Reading the Bottom-Up Database

We add the bottom-up database as global state *BU* and assume it has been computed prior to reduction. That is, when starting the debugger, a bottom-up Datalog engine must compute *BU*.

Global Information:

(bottom-up derived database) $\quad BU \in p \to v$

For each predicate in the debugged program, *BU* provides the final value table of derivable tuples for that predicate. For example, for the path predicate, *BU* contains all pairs (X,Y) of nodes for which X can reach Y. When stepping over a call path$(1,Y)$, we can select all matching pairs from *BU* with a single join operation.

We formalize this behavior as an additional reduction rule A-Over for predicate calls:

$$\text{A-Over}\quad \frac{nonRecursiveCall(p^s(\bar{t}))\qquad v_a = eval(cols(p),\bar{t},v_{sup})}{v_{sup} \vdash p^s(\bar{t}) \;\rightarrow^A\; \rho_{\bar{t}/cols(p)}(BU(p) \bowtie v_a)^s}$$

Rule A-Over only applies to non-recursive predicate calls p (also not indirectly recursive) for reasons that we explain below. Technically, A-Over is applicable when the called predicate p is not in the currently active strongly connected component of the predicate call graph. We then determine the argument table v_a just like before in A-Into and A-Skip, but then look up the predicate result in *BU*. The resulting table carries the sign of the predicate call to mark if it was produced by a positive or negative call. We then merge the argument table v_a with the predicate result to obtain the tuples that match v_a according to this particular predicate call. Finally, we have to rename the columns of the table according to the argument terms like before.

Our new rule makes the semantics non-deterministic: predicate calls can be reduced with either one of the mutually exclusive A-Into or A-Skip, or alternatively with A-Over. This is ambiguity is by design, since users of the debugger should be able to choose how to continue evaluation. Thus, an important property of our extended semantics is confluence: All non-stuck reduction traces of a term normalize to the same value table. We provide a proof sketch for atom reduction.

Theorem 4 (Local Confluence for Atom Reduction)**.** Let $a \rightarrow^A b$ and $a \rightarrow^A c$, then there exists d such that $b \rightarrow^{A*} d$ and $c \rightarrow^{A*} d$.

Proof sketch. There are only two interesting cases to consider, namely a is reduced by A-Skip and A-Over, or when a is reduced by A-Into and A-Over. In both cases $a = p^s(\bar{t})$

Case 1: Let $a \rightarrow^A b$ by A-Skip and $a \rightarrow^A c$ by A-Over. Then

$$b = \rho_{\bar{t}/cols(p)}(IDB(p) \bowtie v_a)^s \quad \text{and} \quad c = \rho_{\bar{t}/cols(p)}(BU(p) \bowtie v_a)^s.$$

Since the query/subquery approach and the bottom-up semantics compute the same result for the same query, $IDB(p) \bowtie v_a = BU(p) \bowtie v_a$ for stable p under argument table v_a. In A-Skip, p is stable and hence $b = c$.

Case 2: Let $a \rightarrow b$ by A-Into and $a \rightarrow c$ by A-Over. Then

$$b = \mathbf{sq}(p^s(\bar{t}), v'_a, \text{table}(cols(p), \emptyset), v'_a, rules(p)) \quad \text{and} \quad c = \rho_{\bar{t}/cols(p)}(BU(p) \bowtie v_a)^s.$$

Subquery b normalizes to $d = \rho_{\bar{t}/cols(p)}(IDB(p) \bowtie v_d)^s$ under $\rightarrow^Q$ using A-Step, where the last step is computed by Q-Stable. Since $v_d = v_a$ is the argument table pushed by A-Into and returned by *popQuery* in Q-Stable, we have $IDB(p) \bowtie v_d = BU(p) \bowtie v_a$ because the top-down and the bottom-up semantics coincide. Thus, $b \rightarrow^{A*} d = c$, which proofs confluence.

□

This proof sketch oversimplifies some concerns, in particular, the induction for the mutually recursive reduction relations and the base assumption that our small-step semantics for top-down Datalog is equivalent to a standard bottom-up semantics of Datalog. Resolving these concerns formally is far from trivial, given that the top-down and bottom-up semantics start evaluation from opposite sides (no shared control flow) and use contrary fixpoint strategies (depth-first in top-down, breadth-first in bottom-up). Mitigating these concerns will be a focus of our future work.

We want to highlight that even though the confluence property holds, repeat applications of A-Into do not necessarily simulate A-Over. Or conversely, a step over does not necessarily correspond to a sequence of step intos. Simply put, the difference is that A-Into modifies the intensional database IDB whereas A-Over does not. This can affect subsequent step-intos in a subtle way, as illustrated by the following scenario. If we step into a query $p(\overline{t})$, we need to determine the fixpoints for all predicate calls $q_1(\overline{t_1}), \ldots, q_n(\overline{t_n})$ required to answer $p(\overline{t})$. Therefore, Q-Iterate is executed for each required subquery at least once, hence extending the intensional database IDB. In contrast, if we step over $p(\overline{t})$, we do not reach the calls $q_1(\overline{t_1}), \ldots, q_n(\overline{t_n})$ and do not store their results in IDB. Therefore, when we later encounter another call of $q_1(\overline{t_1}), \ldots, q_n(\overline{t_n})$, we may need extra fixpoint iterations to reach a stable result. Nonetheless, the fixpoint result of each subquery is the same, which is why confluence holds.

5.4.2 Reduction Trace by Example

To showcase A-Over, let us have a look at a reduction trace for the following Datalog program that derives parents and grandparents:

$$\begin{array}{lll} \mathsf{father}(Bob, Charlie). & \mathsf{father}(Bob, Dave). & \mathsf{father}(Dave, Mallory). \\ \mathsf{mother}(Alice, Charlie). & \mathsf{mother}(Eve, Dave). & \end{array}$$

$$\begin{array}{rl} \mathsf{parent}(X,Y) & \texttt{:-}\ \ \mathsf{edb}\ \mathsf{father}(X,Y). \\ \mathsf{parent}(X,Y) & \texttt{:-}\ \ \mathsf{edb}\ \mathsf{mother}(X,Y). \\ \mathsf{grandparent}(X,Y) & \texttt{:-}\ \ \mathsf{parent}(X,Z), \mathsf{parent}(Z,Y). \end{array}$$

We start the evaluation of the program with query $\mathsf{grandparent}(Bob, Y)$. Hence, we initialize IDB as empty and the active queries Γ with mapping $\mathsf{grandparent}_{bf} \mapsto v_q$ where $v_q = \mathsf{table}(X, \{(Bob)\})$. The initialized EDB contains the following mappings:

$$\begin{array}{lll} \mathsf{father} & \mapsto & \mathsf{table}(X, Y, \{(Bob, Charlie), (Bob, Dave), (Dave, Mallory)\}) \\ \mathsf{mother} & \mapsto & \mathsf{table}(X, Y, \{(Alice, Charlie), (Eve, Dave)\}) \end{array}$$

For the hybrid semantics, we need to define the set of bottom-up derived tables BU as well. A bottom-up Datalog engine will derive the following set of tables:

$$\begin{array}{lll} \mathsf{father} & \mapsto & \mathsf{table}(X, Y, \{(Bob, Charlie), (Bob, Dave), (Dave, Mallory)\}) \\ \mathsf{mother} & \mapsto & \mathsf{table}(X, Y, \{(Alice, Charlie), (Eve, Dave)\}) \\ \mathsf{parent} & \mapsto & \mathsf{table}(X, Y, \{(Bob, Charlie), (Bob, Dave), (Dave, Mallory), \\ & & \qquad (Alice, Charlie), (Eve, Dave)\}) \\ \mathsf{grandparent} & \mapsto & \mathsf{table}((X, Y, \{(Bob, Mallory), (Eve, Mallory)\}) \end{array}$$

For query grandprarent(Bob, Y) we start evaluation with following subquery:

$$\textbf{sq}(\mathsf{grandparent}(Bob, Y), v_q, \emptyset, v_q, \mathsf{grandparent}(X, Y) \text{ :- } \mathsf{parent}(X, Z), \mathsf{parent}(Z, Y).)$$

Q-Step is applicable which applies R-Step which will reduce the first atom of the Datalog rule. The first atom is a predicate call of parent. We can either apply A-Into or A-Over because the call of parent is not a recursive one. We choose A-Over that utilizes *BU* to derive the table of the call. To produce the correct result for the predicate call, we join the bottom-up table $EDB(\mathsf{parent})$ with $v_a = \mathsf{table}(X, \{(Bob)\})$. At last, we need to rename the columns to match the arguments of the predicate call: Hence, A-Over replaces the atom with table

$$v = \mathsf{table}(X, Z, \{(Bob, Charlie), (Bob, Dave)\})$$

such that we obtain

$$\textbf{sq}(\mathsf{grandparent}(Bob, Y), v_q, \emptyset, v_q, \mathsf{grandparent}(X, Y) \text{ :- } v, \mathsf{parent}(Z, Y).)$$

This shows how we can utilize the bottom-up derived database to step-over a non-recursive predicate call instead of producing a new subquery that has to determine a fixpoint. Recursive predicate calls are more complicated as the next section explains.

5

5.5 A Hybrid Semantics for Recursive Datalog

The previous section introduces the idea of a hybrid semantics that steps into according to top-down but steps over according to bottom-up. However, we limited this feature to non-recursive predicate calls. This conflicts with reality, where Datalog programs are highly recursive. Indeed, a Datalog debugger should allow stepping through the fixpoint computation of (mutually) recursive predicates, while also allowing to skip individual recursive sub-computations with step-over. However, supporting step-over for recursive predicate calls requires a more sophisticated semantics.

The problem is this: The bottom-up database contains the final result of a predicate. But when stepping over a recursive predicate call, we need to see a result that is consistent with the current progress within the fixpoint computation, not the final result. Consider the following example, where the directed graph contains two non-overlapping cycles: $1 \to 2 \to 3 \to 1$ and $4 \to 5 \to 4$

$$\begin{array}{lll} \mathsf{edge}(1,2). & \mathsf{edge}(2,3). & \mathsf{edge}(3,1). \\ \mathsf{edge}(1,4). & \mathsf{edge}(4,5). & \mathsf{edge}(5,4). \\ \mathsf{path}(X,Y) \text{ :- edb } \mathsf{edge}(X,Y). & & \\ \mathsf{path}(X,Y) \text{ :- edb } \mathsf{edge}(X,Z), \mathsf{path}(Z,Y). & & \end{array}$$

Assume we want to debug the computation of cycle $1 \to 2 \to 3 \to 1$, starting with query $\mathsf{path}(1, Y)$. Let's say we stepped to the second atom of the second rule. That is, the current program point is a subquery of $\mathsf{path}(2, Y)$ with

$$v_q = \mathsf{table}(X, \{(1)\}), \qquad v_r = \mathsf{table}(X, Y, \{(1,2)\}), \qquad v_{sup} = \mathsf{table}(X, Z, \{(1,2)\})$$

such that

$$\textbf{sq}(\mathsf{path}(1, Y), v_q, v_r, v_{sup}, \mathsf{path}(X, Y) \text{ :- } \mathsf{path}(Z, Y).).$$

If we now apply A-Over, we obtain all tuples of table path from the bottom-up database where column X is 2. The problem is that *BU* contains the final fixpoint of the derivation, which is inconsistent with our current debugging trace. Specifically, the bottom-up derived database *BU* is:

$$\begin{array}{lcl} \mathsf{edge} & \mapsto & \mathsf{table}(X,Y,\{(1,2),(1,4),(2,3),(3,1),(4,5),(5,4)\}) \\ \mathsf{path} & \mapsto & \mathsf{table}(X,Y,\{(1,2),(1,4),(2,3),(3,1),(4,5),(5,4),(1,3),(1,1),(1,5),(2,1), \\ & & \quad (2,2),(2,4),(2,5),(3,2),(3,3),(3,4),(3,5),(4,4),(5,5)\}) \end{array}$$

Hence, A-Over obtains $\mathsf{table}(X,Y,\{(2,1),(2,2),(2,3),(2,4),(2,5)\})$. However, paths $1 \to 2$, $1 \to 4$, and $1 \to 5$ are only just being discovered and would not normally affect a recursive predicate call. Thus, A-Over should only yield $\mathsf{table}(X,Y,\{(2,1),(2,3)\})$: the tuples derivable for $\mathsf{path}(2,Y)$ in the current fixpoint iteration.

Fundamentally, the problem is that the bottom-up and top-down semantics employ conflicting fixpoint strategies. A bottom-up semantics computes the tuples of recursive predicates in iterations, where each iteration uses at least one newly derived tuple from the previous iteration (known as semi-naïve evaluation). In contrast, the recursive query/-subquery approach of the top-down semantics follows the recursive predicate calls until a query is revisited. Thus, even if we were to track the iteration count of tuples in the bottom-up database, we would not know which tuples to produce when in the top-down semantics.

One correct solution for this problem would be go back to simulating step-over using step-into. We propose an alternative that exploits the bottom-up database, but ignores all tuples not derivable in the current fixpoint iteration.

5.5.1 Blacklisting Tuples That Are Ahead of Their Time

blacklisting tuples that are ahead of their time To support step-over for recursive predicate calls efficiently, we want to read the call result from the bottom-up database like before. But, as explained above, the bottom-up database contains tuples we need to ignore to produce a result that is consistent with the fixpoint computation of the top-down semantics. To this end, we must answer the following two questions:

(i) Which tuples do we need to ignore to obtain a correct result from the bottom-up database?

(ii) How can we realize this strategy efficiently?

To answer the first question, consider the following intermediate Datalog term:

$$\mathbf{sq}(\mathsf{path}(X,Y),v_q,v_r,v_{sup},\mathsf{path}(X,Y)\ \texttt{:-}\ \mathbf{sq}(\mathsf{path}(Z,Y),v'_q,v'_r,v'_{sup},\mathsf{path}(X,Y)\ \texttt{:-}\ \mathsf{path}(Z,Y).).)$$

When stepping over the inner predicate call $\mathsf{path}(Z,Y)$, which tuples do we need to ignore from the bottom-up database? Careful investigation of the query/subquery approach reveals the answer: We must ignore tuples that are currently being computed by active subqueries and all tuples that depend on them. For an active subquery of p, only tuples already written to $IDB(p)$ by Q-Iterate may be considered, but no other intermediate answers. For our example, we need to ignore tuples that depend on the tuples $\mathsf{path}(X,Y)$

under table v_q and $\mathsf{path}(Z,Y)$ under table v'_q. More concretely, when we start with query $\mathsf{path}(1,Y)$, we ignore all paths that begin at node 1 for the outer subquery. The inner subquery is of the form $\mathsf{path}(Z,Y)$ under supplementary table $\mathsf{table}(X,Z,\{(1,2)\})$, hence we also ignore all paths that begin at node 2. Hence, a step over the innermost $\mathsf{path}(Z,Y)$ under $\mathsf{table}(Z,\{(3)\})$ will produce $\mathsf{table}(Z,Y,\{(3,1)\})$ only, because paths starting at 1 are only visible in later fixpoint iterations.

Now the question arises how we can exploit the bottom-up database while ignoring specific tuples and their dependents. Clearly, we cannot compute a new bottom-up database whenever the set of active subqueries changes. But, what if we use an incremental bottom-up semantics for Datalog that can react to external changes, such as IncA [Szabó et al. 2021] or DDlog [Ryzhyk and Budiu 2019]? Is it then possible to mark ignored tuples and have the incremental semantics update the bottom-up database appropriately without recomputing it from scratch?

We found a way to engineer the incremental bottom-up database to do exactly that. Our solution consists of two steps: Generate extensional *blacklist* predicates into the original Datalog program, and insert/remove ignored tuples into the *blacklist* predicates during debugging. For example, we rewrite the rules of our path predicate as follows:

$$\mathsf{path}(X,Y) \text{ :- } \mathsf{edb\ bl\_path\_bf}^{-}(X), \mathsf{edb\ bl\_path\_fb}^{-}(Y), \mathsf{edb\ bl\_path\_bb}^{-}(X,Y), \mathsf{edb\ edge}(X,Y).$$
$$\mathsf{path}(X,Y) \text{ :- } \mathsf{edb\ bl\_path\_bf}^{-}(X), \mathsf{edb\ bl\_path\_fb}^{-}(Y), \mathsf{edb\ bl\_path\_bb}^{-}(X,Y), \mathsf{edb\ edge}(X,Z), \mathsf{path}(Z,Y).$$

For each adornment of path that occurs in the program, we add a blacklist guard to the path rules. Blacklist guards are negative calls that ensure a tuple is not blacklisted and thus may be derived.

The blacklist transformation is relatively simple and syntactically rewrites a Datalog program:

$$\mathit{Blacklisted}(p(\bar{t}) \text{ :- } \bar{a}.) = \begin{cases} p(\bar{t}) \text{ :- } \bar{a}. & \text{if } \mathit{nonRecursive}(p) \\ p(\bar{t}) \text{ :- } \bar{a}, \bar{b}. & \text{if } \mathit{recursive}(p),\ \bar{\alpha} = \mathit{adornments}(p), \\ & \quad b_i = \mathit{bl\_p\_}\alpha_i^{-}(\pi_{\alpha_i}(\bar{t})) \end{cases}$$

For a non-recursive predicate (also not indirectly recursive), no changes are necessary since the predicate becomes stable in a single fixpoint iteration. However, for recursive predicates (also indirectly recursive), we introduce additional atoms $\bar{b}$, one for each adornment in $\bar{\alpha}$. Each new atom b_i is a negative call to a new extensional predicate named $\mathit{bl\_p\_}\alpha_i$, where p is the name of the current predicate and α_i is the adornment currently considered. The blacklist predicate is applied to those terms in $\bar{t}$ that are marked bound in the adornment α_i. The blacklist transformation only needs to be applied to the Datalog program evaluated bottom-up. The top-down evaluation can operate on the original Datalog program.

5.5.2 Formalizing Step-Over for Recursive Predicates

The previous subsection introduced blacklist predicates that prevent certain derivations in the bottom-up database. The blacklist predicates are all part of the extensional database, meaning their contents are specified manually. Initially, all blacklists are empty. When using

an incremental Datalog engine, we can insert and remove tuples from extensional predicates dynamically, and the incremental engine updates all derived predicates accordingly. We make use of this feature to blacklist tuples that match active subqueries as part of the debugging semantics, as we explain here.

We extend our hybrid semantics from Section 5.4 to maintain and use blacklist predicates. To this end, we require global state to keep track of the blacklisted queries and the updated bottom-up database. Luckily, we already maintain global state to track active subqueries, namely Γ. Therefore, we only introduce new global state that represents the updated bottom-up database.

Global Information:

(blacklist-aware bottom-up database) $BU \in (p, \Gamma) \to v$

Since the bottom-up database operates on the blacklist-transformed program, it only yields derived tuples not depending on active queries. We make this explicit by adding Γ as a parameter to BU.

We can now extend and adapt the semantics to support efficient step-over recursive predicate calls. We introduce a new reduction rule A-OverRecursive to step over recursive predicate calls:

$$\text{A-OverRecursive}\ \frac{recursiveCall(p^s(\bar{t})) \qquad v_a = eval(cols(p), \bar{t}, v_{sup})}{v_{sup} \vdash p(\bar{t})^s \ \to^A\ \rho_{\bar{t}/cols(p)}(BU(p,\Gamma) \bowtie v_a \ \cup\ IDB(p) \bowtie v_a)^s}$$

We determine the argument table v_a as in A-Over, but in A-OverRecursive we read from the blacklist-aware bottom-up database $BU(p,\Gamma)$. This yields the derivable tuples for the predicate call, except for tuples that depend on active subqueries. For the first fixpoint iteration of a subquery, this behavior is correct and sufficient. Only in later fixpoint iterations, it is important to also consider the tuples derived by the top-down debugger itself in $IDB(p)$. Therefore, A-OverRecursive yields the relevant bottom-up tuples and the tuples derived top-down.

With the help of the blacklist and an incremental bottom-up Datalog engine, we can now step over arbitrary predicate calls efficiently. Specifically, we do not need to simulate stepped-over predicates with step-into. We now have all the tools to implement a working debugger for Datalog.

5.6 Debugger Implementation and Frontend Debugging

The small-step operational semantics presented in the previous sections are sufficient for the definition of an interactive Datalog debugger. We validate this claim by implementing a fully functional debugger that we make available open source[1] as part of the Scala2 re-implementation IncA$_{\text{Scala}}$ of the IncA Datalog framework [Szabó et al. 2016]. In this section, we describe the debugger implementation and optimizations we applied on top of the formal semantics. Since Datalog is often used as an intermediate representation (IR)

[1] https://gitlab.rlp.net/plmz/inca-scala

rather than as a frontend language, we also show how the debugger can support debugging of languages that compile to Datalog.

5.6.1 From Small-Step Semantics to Debugger

We implemented a debugger for Datalog following the formal semantics presented in this chapter. In particular, the debugger supports stepping through query execution with the following interactions:

- Step into: A single step of the query reduction relation $\rightarrow^{Q}$, using any rule but A-Over.
- Step over: Conversely, a single step of the query reduction relation, using any rule but A-Into.
- Step out: Abort and discard the current subquery and perform a step-over interaction of its call instead. Note how this avoids the repeated application of step-over until reaching the end of the current subquery.

The debugger uses IncA's incremental bottom-up Datalog engine [Szabó et al. 2021].

Our debugger implementation deviates from the formal semantics in multiple aspects. First of all, the debugger uses a flattened representation for the intermediate terms of Datalog: Rather than nesting active subqueries in the syntax, we maintain a call stack where each active subquery has its own call frame. This is possible because the semantics deterministically executes rules and atoms from left to right, so that there can be only one active subquery at each recursion level (otherwise would need to maintain a call tree). Rule A-Into pushes a new call frame to the stack (instead of creating a subquery) and rule Q-Stable replaces the current call frame with the subquery result. To pop from the call stack, we introduce a new rule Q-Result that pops the subquery result from the stack and replaces the predicate call of the current active subquery with the resulting atom table. This way, the call stack provides constant-time access to the most recently created subquery, to which all other reduction rules apply. In addition, the implementation groups the atom reduction rules that handle equalities within a single rule A-Eq. At last, the IR supports executing arbitrary Scala code. Hence, we extend the debugger with rule A-Prim that executes the Scala code with each supplementary table entry as input.

The debugger incorporates other optimizations that are more local, but also have a noticeable performance impact. First, rules with overlapping preconditions (like A-Into and A-Skip) are implemented such that the preconditions are run at most once. Second, we implement the termination condition in rules Q-Stable and Q-Iterate by comparing the size of the current result to the result size in the bottom-up database, which avoids a costly subset check in each iteration. We also only store derived tuples for recursive predicates in *IDB* as non-recursive predicates do not need fixpoint iteration. Third, we represent value tables as immutable b-trees with efficient table operations [Jordan et al. 2019b]. In particular, this data type provides an efficient join implementation, which is crucial for executing Datalog on considerable amounts of data.

Finally, our debugger also supports breakpoints. A breakpoint identifies a predicate, a rule inside a predicate, or an atom inside a rule at which execution should stop. Breakpoints can be registered and de-registered with the debugger. Breakpoints affect the semantics of

the step-over reduction: A-Over may not be applied to predicate calls that can transitively reach an active breakpoint. We add this as an additional precondition. Similarly, a step-out interaction is not applicable if the remainder of the current subquery may reach a breakpoint. In both situations, we fall back to a new *resume* interaction instead:

- Resume: The resume interaction runs the program until it terminates or a breakpoint is hit. To run the program, resume iteratively applies prioritized interactions: 1. step out if possible, 2. step over if possible, 3. step into otherwise.

5.6.2 Debugging Datalog Frontends

Our debugger supports core Datalog with negation. However, Datalog dialects usually add numerous language features on top of core Datalog or provide entirely different frontend languages. For example, the Souffle language provides a component system, typed relations, and multi-head rules [Scholz et al. 2015]. And IncA not only provides a Datalog-flavored constraint language [Szabó et al. 2016], but also functional programming frontend called functional IncA that compiles to core Datalog, which we discussed in Chapter 4. In principle, we can use the core Datalog debugger to evaluate core Datalog code generated by Souffle and IncA. However, the user would see debugging steps in terms of the core Datalog code: core Datalog rules with auxiliary atoms and variables. This breaks the abstraction of the frontend language and hinders the applicability of the debugger severely. This subsection explains how we adopted our core Datalog debugger to debug Souffle and functional IncA code at their abstract level.

The basic idea for debugging Datalog frontends is simple: We define a partial function $lift$ from core Datalog intermediate terms to the intermediate terms of the frontend language that should be displayed as steps to the user. We also define the inverse of $lift$ called *lower*, which translates the frontend term back to its core Datalog original. Given a reduction relation $\rightarrow^Q$ of core Datalog, we can then define a reduction relation $\rightsquigarrow$ for the frontend language as follows:

$$\frac{lower(b) \rightarrow^Q Q_1 \rightarrow^Q \ldots \rightarrow^Q Q_{n+1} \qquad lift(Q_1) = \bot \quad \ldots \quad lift(Q_n) = \bot \qquad lift(Q_{n+1}) = b'}{b \rightsquigarrow b'}$$

That is, the frontend debugger steps in core Datalog using query reduction until it reaches an intermediate term that can be lifted into the frontend. That is, lifting a query Q is successful ($lift(Q) \neq \bot$). While this approach is not particularly novel, this chapter provides the necessary formal infrastructure to explain and specify this technique precisely. For breakpoints, we define another lowering function that translates frontend breakpoints into the core Datalog breakpoints. The breakpoint-lowering function should be defined such that the lowered breakpoints stop at intermediate terms that can be lifted back into the frontend. In practice, we found that implementing the lifting and the two lowering functions was easy for Souffle and functional IncA.

The frontend reduction relation $\rightsquigarrow$ specified above works well when a Datalog frontend merely elaborates into core Datalog. That is, a single construct compiles to a single Datalog rule, as is the case with most Souffle features. Frontend languages that apply more complex compilation rules require extra care. For example, functional IncA features *if-then-else*

expressions that compile into two rules: one rule that asserts the condition evaluates to true and executes the *then* branch, and one rule that asserts the condition evaluates to false and executes the *else* branch. When using the lifting approach of $\rightsquigarrow$ from above, we obtain an inadequate reduction relation for functional IncA:

- If the condition evaluates to true, we correctly step through the corresponding rule first. However, we will subsequently step through the *else* rule, re-evaluate the condition, and only then discard this rule as unsatisfiable (always failing).
- Conversely, if the condition fails, we step through the *then* rule until the condition failed, then step through the same prefix in the *else* rule, and only then step through the *else* branch.

Either way, we ran the condition and all atoms leading up to the condition twice, which is not what we expect from a debugger of a functional language. Therefore, we augment the frontend debugger with extra logic to skip unsatisfiable rules and satisfied prefixes. Unsatisfiable rules result from conditionals (and pattern matching) when execution has already committed to another branch (or match case). Satisfied prefixes also result from conditionals (and pattern matching) when execution reached a condition that failed and needs to try another branch (or match case). With these two mechanisms in place, we can adopt the core Datalog debugger to provide debugger for functional IncA that meets user expectations.

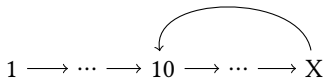
5.7 Performance Evaluation

In this section, we evaluate if the proposed top-down debugging approach can be used for realistic Datalog programs, and if the hybrid semantics is necessary. To this end, we measure the step-into and step-over running times for both semantics on synthetic and real-world Datalog programs. We performed all measurements on a machine with an Intel Core i7 at 2.7 GHz with 16 GB of RAM, running 64-bit OSX 13.2.1, Java 11.0.18.

5.7.1 Is a Hybrid Semantics Necessary?

Without our hybrid semantics, top-down Datalog debuggers must simulate step-over interactions through a sequence of step-into interactions. To evaluate if this is feasible, we measure their respective running times and compare them to the time required by a bottom-up Datalog engine.

Setup. We use the educational Datalog example that derives transitive paths between nodes of a graph, which we have seen throughout this chapter. We consider the following synthetic graph, where we can configure the length of the cycle by setting node $X \in \{10, 20, \ldots, 990, 1000\}$:



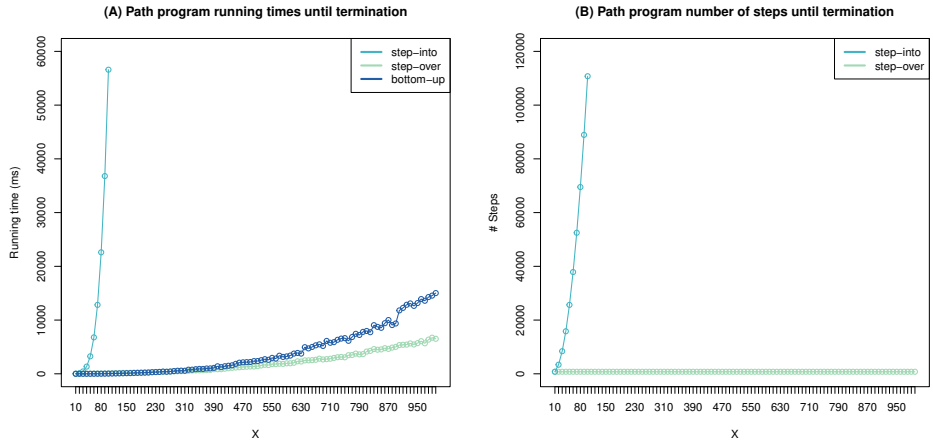


Figure 5.5: Running times of step-into vs step-over vs bottom-up for path program

We measure and compare three scenarios for the initial query $\mathsf{path}(1,Y)$:

(1) Exclusively use step-into interactions until the query terminates.

(2) Use step-into interactions until reaching query $\mathsf{path}(11,Y)$, then use step over. Since $\mathsf{path}(11,Y)$ is within the graph's cycle, this exercises our A-OverRecursive from the hybrid semantics.

(3) Use a bottom-up semantics without stepping, as required to initialize the hybrid semantics.

Note that we chose to step over the path call on first inner node of the cycle as the blacklist-propagation will delete all paths that cross node 11. This scenario is the worst-case scenario which will stress the blacklist-propagation of the incremental bottom-up semantics when using A-OverRecursive. We execute these scenarios and measure their running times for $X \in \{10, 20, \ldots, 990, 1000\}$. We report the average times of 20 runs after discarding 5 warmup runs.

Results. Figure 5.5 shows the running times on the left and number of executed steps on the right. For the step-into scenario (1), we observe an exponential running time, whereas the step-over scenario (2) and bottom-up scenario (3) exercise quadratic running times. Interestingly, the number of steps remains constant when using a step over at $\mathsf{path}(11,Y)$, since we skip the computation for remainder of the graph as X grows. Only the running time grows with X, because the incremental blacklist propagation has to delete a quadratic number of path tuples with increasing X.

The results show that using repeated step-into interactions to simulate a step-over interactions is infeasible as it requires too many steps, taking too much time. Our hybrid semantics and its step-over interaction based on an incremental bottom-up Datalog is necessary and effective.

5.7.2 Real-World Workloads

The previously measured path example is an educational example but not a real-world program. Even for the educational example it is infeasible to only use step-into when debugging because it takes an excessive amount of steps. Hence, interactive debugging of real-world programs stands or falls with the responsiveness of the step-over interaction. To show that our hybrid top-down debugging approach is applicable to real-world programs with realistic workloads, we measure the step-over performance for an inter-procedural points-to analysis of JVM bytecode provided by the Doop framework [Bravenboer and Smaragdakis 2009b].

Setup. We debug Doop's context and flow insensitive points-to analysis on the MiniJava compiler.[2] This program only consists of 6.5k lines of Java code, but the inter-procedural analysis also transitively analyzes all reachable parts of the JDK. We use the Doop fact extractor to translate the JVM bytecode to an EDB that describes the program and the transitively reachable parts of the JDK, amounting to 380 MB of data.

We consider three debugging scenarios for measuring the step-into performance:

(1) Derive all classes implementing method `accept` and show how they are computed.

(2) Derive all supertypes of two given JDK types and show how they are computed.

(3) Derive if a given variable points to a given heap location and follow the computation.

Additionally, we define three variations of scenario (3) for measuring the step-over performance:

(3a) Step into VarPointsTo, but step over all other predicate calls.

(3b) Step into VarPointsTo and StaticFieldsPointsTo, but step over all other predicate calls.

(3c) Step into VarPointsTo and InstanceFieldPointsTo, but step over all other predicate calls.

(3d) Step into VarPointsTo and Reachable, but step over all other predicate calls.

Note that VarPointsTo, StaticFieldPointsTo, InstanceFieldPointsTo, and Reachable are mutually recursive. We run each of these debugging scenarios only once, where we consider a timeout of 10 minutes after doing warmup with a timeout of 2 minutes 5 times. For scenarios (1–3), we measure the running time of each individual reduction rule, whereas for scenarios (3a–3d) we only measure the running time of step-over interactions (A-Over or A-OverRecursive).

Results. We present the results of the benchmark in Figure 5.6. The boxplots on the left show the running times (log scale) of all reduction rules executed during scenarios (1–3), including an additional boxplot combining all 38029 reduction steps. Most reduction rules only require at most 1 ms to execute, except A-EDB (up to 1000 ms) and A-Prim (up to 100 ms). The boxplots also exclude outliers, but we want to mention that out of the 2202 steps with A-Into, there was one extreme outlier that took 40s and a few of outliers

[2] `https://github.com/mtache/minijavac`

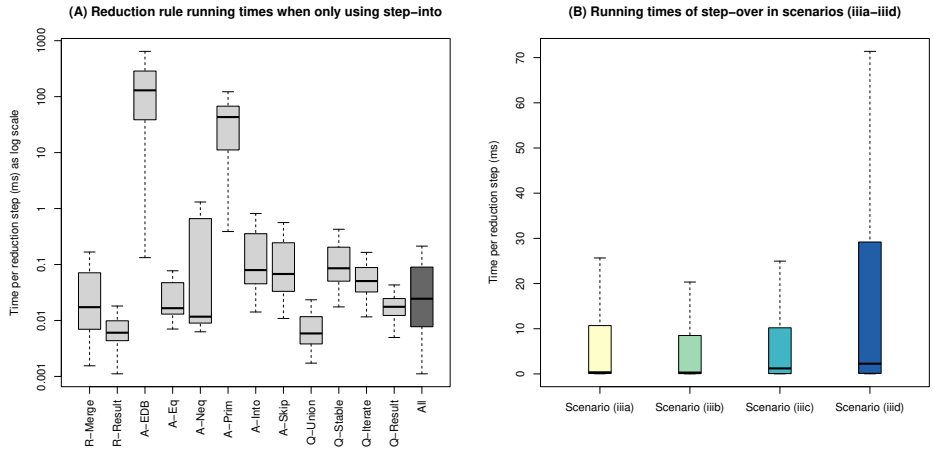


Figure 5.6: Running times (ms) of reduction rules of Doop points-to analysis.

5

that took around 10s to execute. The running times of A-EDB and the outliers of A-Into are slow because they construct an index on-the-fly for a complete *EDB* or argument relation prior to joining, which can take a significant amount of time. With enough engineering effort, it should be possible to optimize the index construction and operations on tables to reduce these times considerably. A-Prim executes Scala code using reflection on each entry of a value table, which has a severe penalty in our current implementation.

We show the step-over running times (linear scale) for the interactive debugging scenarios (3a–3d) on the right-hand side of Figure 5.6. The mean step-over running time is below 10 ms and indeed almost all step-over steps require less than 100 ms. Again, the boxplots exclude outliers: Scenario (3c) has one outlier in 1278 step-over interactions that took 168s and scenario (3d) has two outliers in 888 step overs taking 62s and 40s. These outliers are due to the incremental deletion of tuples that have a high impact on the bottom-up database. Szabó et al. [2021] showed that such high-impact changes exist but are rare for static-analysis applications of Datalog; most changes have low impact and can be executed efficiently.

We conclude that interactive debugging of real-world Datalog programs is feasible with the hybrid top-down debugger. Both step-into and step-over interactions are executed fast enough for interactive debugging. Even though there are some extreme outliers, they are rare and do not dominate the debugging session. Thus, a hybrid debugging semantics is necessary (previous subsection) and sufficient (this subsection).

5.8 Related Work

We propose to enable an interactive debugging experience for Datalog programs by exploring the top-down reduction trace of a Datalog program using a top-down evaluation strategy. To efficiently allow for step-over interactions, we provide a hybrid semantics for

Datalog that mainly uses top-down but relies on a bottom-up derived database when stepping over predicate calls. We will compare our debugging approach of Datalog programs with related work in this section. However, we leave out some related Datalog systems, which we will discuss in Chapter 6.

The explanation system of DeDEx utilizes derivation trees to show why specific tuples have been derived [Wieland 1990]. This system can only show derivation trees for tuples that have been derived. Hence, it cannot explain why a tuple has not been derived. Additionally, the system requires a complete tuple. In contrast, our approach allows to start with arbitrary queries of predicates. That is, we are not forced to only select tuples that have been derived and we can provide partial queries to explain how a specific query has been answered. This allows to show the reduction trace for multiple tuples. Their system allows for a restricted query facility mode. This mode allows to ignoring and adding rules and tuples when constructing derivation trees for predicates. Our approach does not allow for ignoring or adding specific rules. However, our hybrid approach allows for ignoring tuples to allow for stepping over cyclic predicate calls. Their approach will only ignore tuples and it will not effect tuples that depend on the existence of the ignored tuple. We propose to use an incremental Datalog engine to always keep a consistent deductive database as a basis.

Russo and Sancassani [1991] propose to explore an evaluation post-mortem, meaning after executing a Datalog program. They use a derivation tree to show why a tuples has been derived as well. They instruct their Datalog engine to derive derivation tree fragments during bottom-up evaluation. This allows for efficient exploration of derivation trees. We also rely on the tuples derived by a Datalog engine. However, our approach allows to use an off-the-shelf incremental Datalog engine instead of extending the Datalog engine to generate additional information.

Explain is capable of explaining how specific tuples were derived and which tuples are derived based on a specific tuple [Arora et al. 1993]. They use derivation trees to visualize how a specific tuple has been derived as well. Again, this system does not allow partial tuples which our approach is capable of. They also adapt the Datalog engine to derive fragments of the derivation tree. Again, our approach does not require extending the Datalog engine. Explain is capable of supporting aggregation. Our formalization does not consider aggregation, but we support it because the Datalog engine IncA we use supports recursive lattice-based aggregation [Szabó et al. 2018].

Caballero et al. [2008a,b] propose algorithmic debugging for Datalog programs. They derive a computation graph representing the evaluation of a Datalog query where the nodes represent predicate tables for a specific query and the edges determine which queries are needed to derive the predicate table. Their approach traverses the computation graph and ask the user if stored predicate table is valid to identify buggy vertices or buggy circuits. A node of the computation graph is buggy if the predicate table is non-valid while all predicate tables of the immediate descendants are valid. A buggy circuit is a cycle in the computation graph where all vertices are invalid. Köhler et al. [2012] propose a similar approach, that is declarative debugging, since they also derive a graph connecting tuples with rule firings. They use Statelog [Lausen et al. 1998] to record in which iteration and how often a tuple has been derived. Based on the graph it is possible to extract and explore subgraphs with pre-defined and ad hoc defined queries such as counting rule firings. In

contrast, our approach does not follow algorithmic or declarative debugging. We propose to follow the computation model of a Datalog program instead. That is, we follow the execution trace of a top-down evaluation while utilizing a bottom-up derived database to answer step-overs efficiently.

5.9 Chapter Summary

Current Datalog debugging techniques are based on the view of Datalog as a query language for databases where they use provenance information and follow the data instead of the program execution. In recent years, Datalog has been used as a programming language instead. In this chapter, we propose an interactive debugging technique for Datalog programs that follows a top-down evaluation strategy while exposing the execution state. Interactive debugging acts as an operational abstraction and closes the representational gap (RG3) Observable Behavior. We define the top-down evaluation strategy as a small-step operational semantics where each application corresponds to a step-into interaction. We base the small-step operational semantics on the evaluation strategy recursive query/-subquery. While recursive query/subquery is not novel [Vieille 1986], we are the first to define a small-step operational semantics formulation of it. This semantic artifact will help substantiate programming-language research on Datalog. Top-down evaluation of Datalog is less efficient than bottom-up semantics, hence all state-of-the-art Datalog engines use semi-naïve evaluation. Thus, simulating step-over using only step-into interactions is highly inefficient. We propose to define a hybrid semantics that combines top-down and bottom-up evaluation. To this end, we can access the bottom-up derived database when stepping over a predicate call. In particular, we use an incremental Datalog engine to allow for debugging fixpoint iteration efficiently. We implement the interactive debugger based on the small-step operational semantics within the IncA framework [Szabó et al. 2021]. Our implementation allows for setting breakpoints as well. We enable debugging frontends that compile to core Datalog such as Soufflé and functional IncA by lifting and lowering intermediate terms back and forth. In future work we want to explore how we can extend the interactive debugger with condition breakpoints to explore specific evaluation points more efficiently. The performance evaluation shows that top-down debugging of Datalog programs is feasible when using a hybrid top-down semantics to efficiently answer step-over interactions.

6

Related Work

In this chapter, we consider if and how related Datalog systems close the following representational gaps we introduced in Section 1.2:

(RG1) Computation: The representational gap between the computations required by the domain and the computation style Datalog offers.

(RG2) Data: The representational gap between the data the domain operates on and the data Datalog operates on.

(RG3) Observable Behavior: The representational gap between the observable program behavior domain experts expect and Datalog provides.

We organize this chapter in the style of a survey. That is, we introduce each related Datalog system successively while comparing each system with the contributions of this dissertation. Figure 6.1 provides an overview of related Datalog systems. We compare them with the contributions of this dissertation in four aspects. First, we consider if the Datalog system uses Datalog as an intermediate representation or a user-facing programming language. Then, we compare if they provide linguistic abstractions, data abstractions, or operational abstractions for Datalog. During the discussion, we only discuss differences between the abstractions we propose and the related systems provide. Whenever a related system does not provide an abstraction, we leave it out of the discussion.

In the first line of the table, we highlight $\text{IncA}_{\text{Scala}}$, which is the Datalog system we extend in this dissertation. We split related Datalog systems into two categories based on their origin: (i) database systems, and (ii) programming languages (PL) systems. We introduce these categories because each system views Datalog differently based on its origin. Thus, each system tackles different research questions. Systems having their origin in database systems consider simplistic Datalog programs operating on large-scale data. In contrast, systems having their origin in programming languages focus on extensibility and expressing complex Datalog programs. First, we discuss systems with a database point of view in Section 6.1. In Section 6.2, we discuss Datalog systems originating from programming language research.

Approaches	Datalog as an IR	Linguistic Abstraction	Data Abstraction	Operational Abstraction
$\text{IncA}_{\text{Scala}}$ (*This dissertation*)	●	●	●	●
Database Origin				
BigDatalog [Shkapsky et al. 2016]	○	○	○	○
RecStep [Fan et al. 2019]	○	○	○	○
DcDatalog [Wu et al. 2022]	○	○	○	○
SociaLite [Seo et al. 2013a,b]	○	○	○	○
LogicBlox [Green 2015]	○	○	○	○
Logica [log 2024]	○	○	◐	○
DDlog [Ryzhyk and Budiu 2019]	○	◐	◐	○
Rel Language [rel 2024]	●	●	○	◐
Viatra [Ujhelyi et al. 2015; Varró et al. 2016]	○	◐	◐	◐
Programming Languages Origin				
egglog [Ryzhyk and Budiu 2019]	○	○	◐	○
bddbddb [Whaley and Lam 2004; Lam et al. 2005]	○	○	●	○
Ascent [Sahebolamri et al. 2022, 2023]	○	◐	◐	○
Flan [Abeysinghe et al. 2024]	○	◐	◐	○
Flix [Madsen et al. 2016; Madsen and Lhoták 2020]	○	◐	◐	○
Formulog [Bembenek et al. 2020]	○	◐	◐	○
CodeQL [Avgustinov et al. 2016]	●	◐	●	○
Bloom [Alvaro et al. 2011; Conway et al. 2012]	○	●	◐	○
Soufflé [Scholz et al. 2016]	○	◐	◐	◐
IncA_{MPS} [Szabó et al. 2016, 2018, 2021; Szabó 2021]	○	◐	●	○
Datafun [Arntzenius and Krishnaswami 2016, 2020]	○	●	●	○

Figure 6.1: This table lists related Datalog systems categorized by their respective community. The first column indicates if a Datalog system uses Datalog as an intermediate representation (full circle ●) or a user-facing programming language (empty circle ○). The other columns show if a Datalog system provides a specific abstraction. For the columns of linguistic abstraction, data abstraction and operational abstraction we use an empty circle ○ if the system does not provide any abstraction, a half circle ◐ if it provides partial abstraction, or a full circle ● if it provides full abstraction.

6.1 Datalog Systems with a Database Origin

In this section, we discuss Datalog systems having their origin in database systems. They primarily consider simple Datalog without extending Datalog's expressivity or improving its usability. These systems aim to improve the performance of evaluating simple Datalog programs on large datasets using single-node multi-core machines or multi-node systems by distributing processing. All Datalog systems we discuss in this section support Datalog with aggregation.

BigDatalog. BigDatalog [Shkapsky et al. 2016] is a distributed Datalog system supporting standard Datalog, including aggregation. The system compiles Datalog programs to Apache Spark [apa 2024]. Apache Spark is an engine performing big data analytics by allowing execution on multi-node systems. BigDatalog only supports standard Datalog features such as monotonic recursive aggregation using built-in aggregation operators and non-monotonic non-recursive aggregation. The system exposes users to Datalog's logic programming style. Users write Datalog rules because it does not provide abstractions for computations or data representation. Additionally, the system does not provide an operational abstraction for observing the behavior of BigDatalog programs. The focus of BigDatalog is to compile Apache Spark and apply optimizations for efficient large-scale execution of the Apache Spark engine and not ease the use of Datalog.

RecStep. RecStep [Fan et al. 2019] is a single-node parallel Datalog system supporting standard Datalog, including aggregation. The authors implement Datalog's semi-naïve evaluation strategy based on a parallel in-memory relational database system called QuickStep [Patel et al. 2018]. The system supports non-recursive as well as recursive aggregation. Users need to use appropriate aggregation operators when using recursive aggregation to enforce termination. The system applies system-level and Datalog-to-SQL-level optimizations to achieve efficient parallel execution for general-purpose workloads. Again, RecStep exposes the standard Datalog interface in the form of horn clauses. Hence, it does not provide any abstractions for Datalog.

DcDatalog. DcDatalog [Wu et al. 2022] is a Datalog system operating on shared-memory multi-core systems. It provides a standard Datalog language, including recursive aggregation. The authors aim to utilize multi-core systems to enable parallel execution and provide a dynamic scheduling algorithm that generates a parallel execution plan on the fly. Additionally, they aim for a lightweight coordination scheme during parallel execution to reduce concurrent access to shared memory. The authors do not focus on easing the use of Datalog. Thus, DcDatalog exposes the standard Datalog surface syntax to users. The authors do not aim to provide any abstractions for Datalog.

SociaLite. SociaLite [Seo et al. 2013a] is a Datalog system extending Datalog with features to enable efficient and expressive social network analysis. These features include data layout representation for graphs by allowing tail-nested tables where the last column contains another table. This enables them to efficiently perform graph traversals during

Datalog evaluation while keeping memory usage low. Additionally, they allow users to provide hints, which specify the execution order. At last, SociaLite's Datalog dialect supports recursive aggregate functions, which are crucial in expressing social network analyses. In follow-up work, Seo et al. [2013b] introduces a distributed version operating on a cluster of multi-core machines. Users declare how data is shared across the cluster while the system determines how to distribute computations and the required communication between each node. The system operates in a master-slave mode where the master machine executes the Datalog program and distributes the workload to a group of slave machines.

SociaLite exposes users to Datalog's logic programming style, because the authors state Datalog is already the right abstraction for their domain. In contrast, we provide a type system DSL (Chapter 2) and a general-purpose functional language (Chapter 4), which both act as linguistic abstractions for Datalog. SociaLite has a data abstraction to enable efficient graph traversal, which differs from ours for incrementally processing structured data, which we introduce in Chapter 3.

LogicBlox. LogicBlox [Aref et al. 2015; Green 2015] is a commercial platform for solving business analytics problems using Datalog. LogicBlox provides an incremental engine based on DRed. The system provides Datalog as a surface language with primitives and entities, which are a core data construct. Entities are defined by a unary predicate and they can be arranged in a subtyping hierarchy. Based on entities, users can define integrity constraints on entities as well, and every subtype entity will inherit the integrity constraints of its supertype entity. LogicBlox's Datalog is part of an interactive UI environment. In particular, their Datalog dialect can express how to populate UI views of the LogicBlox system. Even more interesting, UI events can trigger a re-evaluation of the Datalog program, and Datalog rules only fire if a UI event gets triggered. Special rules can also populate EDB relations when a UI event gets triggered. At last, they can construct data during evaluation using constructors. Using constructors can lead to non-terminating Datalog programs. However, they use a safety check warning that termination is not guaranteed. The authors use a connection between Datalog and the chase procedure [Meier et al. 2009].

LogicBlox does not provide linguistic abstractions for Datalog but exposes Datalog as a surface language that integrates tightly into a reactive environment. In contrast, we provide two linguistic abstractions for Datalog. One for type system specifications and one for functional programming with fixpoints. LogicBlox supports constructing algebraic data during run time which functional IncA does as well, but we do not provide a safety check for termination. However, it would be interesting if the same idea applies to functional IncA programs. LogicBlox uses provenance-based debugging for Datalog programs as an operational abstraction. We develop a stepping debugger that follows a top-down execution order. While provenance-based debugging only allows the exploration of derivation trees of derivable tuples, our approach explores the execution of any query. Hence, we can explore how partial queries and unsatisfiable queries are derived.

Logica. Logica [log 2024] is a general-purpose Datalog system that compiles Datalog programs to SQL queries. In particular, Logica can target the systems BigQuery [Fernandes and Bernardino 2015], SQLite, and PostgreSQL. The rationale is to utilize the strengths of industrial-grade relational database systems. The Datalog dialect supports standard

Datalog, including aggregation. Additionally, they provide support for constructing JSON-style records. They allow record construction during evaluation, but each constructed record is stored in a database table cell. In contrast, the data abstraction we propose allows for storing structured data like records relationally while enabling efficient incremental updates. Logica does not aim to enable incrementality because its system does not provide an incremental execution model.

DDlog. DDlog [Ryzhyk and Budiu 2019] is a Datalog system that utilizes differential dataflow (DDF) [McSherry et al. 2013] as an execution model. DDF is an execution model for computations operating on large data sets and incrementally maintaining the computation result by propagation changes through a network of operators. DDlog utilizes DDF to implement an incremental Datalog engine. The Datalog dialect provided by DDlog combines Datalog with functions defined in a custom functional language. The functional language supports algebraic data, first-class functions, and higher-order functions. It is possible to mutate data, but only within Datalog rules. Hence, the mutation is only locally visible. The main computations in DDlog are Datalog rules and functions, while functions cannot access Datalog relations.

This is different from the tight integration of relations and functions that functional IncA (Chapter 4) offers. We enable the tight integration by compiling both functions and relations to Datalog. Another shortcoming of DDlog's design is that first-order functions cannot promote code reuse of relations, thus, organizing large DDlog programs becomes difficult. DDlog supports algebraic data as input by storing algebraic data values as a whole. In contrast, we provide a data abstraction for structured data such as ASTs or algebraic data, which encodes structured data relationally across multiple relations (Chapter 3). This encoding follows design principles to enable efficient incremental updates when running Datalog incrementally. For DDlog, users need to relationally encode structured data manually if they aim for efficient incremental updates. DDlog allows constructing algebraic data during evaluation. Functional IncA allows constructing algebraic data during evaluation by encoding algebraic data values relationally. The relational encoding allows functional IncA programs to enumerate all constructed values during execution and implement functions operating on the constructed values. This is not possible with DDlog. DDlog provides capabilities to inspect the execution trace of a Datalog program, which dumps all invocations of a Datalog rule into a logging file. This is a low-level approach to inspecting and debugging Datalog programs. In contrast, we propose an operational abstraction in the form of a top-down stepping debugger similar to debugging other paradigms such as imperative programming (Chapter 5). Our approach allows us to observe the execution trace and inspect the execution state as it happens.

Rel Language. RelationalAI [rel 2024] is a company developing a relational knowledge graph system. They aim to combine business rules and graph analytics to build one cloud-native system. RelationalAI develops a relational language on top of their system, called Rel. Rel is an expression-based language to construct and query relational databases. The language provides expressions such as if expressions and set comprehension to derive relations. Users can define higher-order relations and modules to organize and compose large programs. Rel can use negation as well as recursive aggregation using built-in

aggregation operators. Users can encode knowledge graphs relationally and define integrity constraints for the relational encoding. The system has its foundation in Datalog. They use Datalog as an intermediate representation language and translate Rel programs to Datalog. They utilize techniques to define an incremental and demand-driven execution model for Rel programs.

In this dissertation, we propose to use Datalog as an intermediate presentation as well, and provide linguistic abstractions for it. In particular, a type system DSL and functional IncA, which is a general-purpose functional language. Functional IncA supports conditional expressions such as if-then-else and provides set-comprehension expressions to derive sets like Rel. We also support higher-order functions and parametric polymorphism to promote code reuse. Our functional language lacks a module system, but introducing a module system should be possible. Our language does not allow querying relations negatively, because the demand transformation can turn a stratifiable program into an unstratisfiable one. To overcome this shortcoming, we need to adapt the targeted Datalog engines like Tekle and Liu [2019] propose. Rel does not support structured data. Therefore, users need to relationally encode structured data such as knowledge graphs themselves. In contrast, we provide a data abstraction for structured input data, which encodes tree-shaped data. The data abstraction follows principles that enable efficient incremental updates for incremental Datalog engines. When using Rel, users need to follow these principles themselves.

Viatra. Viatra [Ujhelyi et al. 2015; Varró et al. 2016] is a reactive and incremental model transformation platform. Viatra contains the incremental query engine ViatraQuery, which IncA_{MPS} and $\text{IncA}_{\text{Scala}}$ are built on. The system provides a query specification language, which is semantically equivalent to Datalog. Based on a query specification, they generate a computation network that propagates changes between nodes when the input changes. The work of Szabó [2021] extends ViatraQuery with the incremental algorithms DRed_L and LADDDER. Their engine supports calling arbitrary JVM code, which they use as a basis to support recursive user-defined aggregation. They propose a debugging technique that derives a sub-network to identify faulty nodes [Ujhelyi et al. 2016]. In addition to the sub-network, they identify inputs that derive or do not derive a specific tuple. Programmers can use this information to identify errors in the query specification.

In contrast, the stepping debugger we propose for Datalog only explores Datalog predicates that contribute to answering the initial query. The stepping debugger can be used as a basis for a debugger of Datalog frontends such as functional IncA. Viatra identifies a sub-network and it is not obvious how to map the sub-network to Datalog frontends and back. Hence, it is an open question if it is possible to provide operational abstractions for Datalog frontends.

6.2 Datalog System with a PL Origin

In this section, we discuss Datalog systems with origins in programming language research. These systems aim to extend the expressivity of Datalog or improve its usability as a programming language and are closely related to the work presented in this dissertation.

egglog. egglog [Zhang et al. 2023] is a Datalog system combining Datalog with equality saturation. Equality saturation is an approach for term rewriting using equalities. The approach finds all possible equal terms by exhaustively applying equalities until the resulting e-graph is stable. To this end, applying an equality extends a specialized graph representing all rewritten terms by grouping equal sub-terms. Equality saturation is another way of finding a fixpoint using rules in the form of equalities. It is well-suited for term rewriting, finding congruence closures, and term extractions. In contrast, equality saturation is not well-suited for lattice-based reasoning and incremental execution, which are Datalog's strengths. egglog aims to combine logic programming and equality saturation to utilize the strengths of both approaches. Both approaches share a similar style of solving problems: provide a program in the form of rules. However, while Datalog rules are uni-directional, equalities are bi-directional. egglog extends standard Datalog with equalities and uninterpreted functions. Because terms are a core concept of equality saturation, egglog allows structured data as input and produces structured data during evaluation.

egglog programs use extended Datalog and do not provide any linguistic abstractions for computations. We provide two linguistic abstractions, one for type system specification and one for general-purpose functional programming. However, egglog extends Datalog to support equality saturation capabilities efficiently. While we could use functional IncA to write a term rewriting system, we believe we are not able to achieve the efficiency of egglog's equality saturation support, because they utilize efficient equality saturation techniques in their engine. Equality saturation operates on terms, which is just algebraic data. While relations can range over terms in egglog, we provide a data abstraction for structured data by encoding them relationally over multiple relations. This encoding enables efficient incremental updates when using incremental Datalog engines.

bddbddb. bddbddb [Whaley and Lam 2004; Lam et al. 2005] is a Datalog system designed to develop static program analyses. bddbddb provides various context-sensitive pointer analyses for Java programs in the form of Datalog programs. The system compiles Datalog programs to programs using binary decision diagrams (BDDs), which represent relations. BDDs enable concise representation of relations and enable efficient querying. The system provides a standard Datalog language with support for specifying binary decision diagram orderings. To write analyses against Java programs, they encode Java programs relationally.

bddbddb does not provide any linguistic abstractions but provides Datalog as a surface language. The authors provide a data abstraction for Java programs by encoding them relationally. However, their encoding does not follow the design principles we introduce in Chapter 3. In addition to using unary and binary relations to encode structured data like our data abstraction, the authors also use larger n-ary relations. The authors also use absolute positions to encode lists. Both design decisions hurt incremental performance.

Ascent. Ascent [Sahebolamri et al. 2022, 2023] is a Datalog system written in Rust. Ascent provides a macro-based Datalog DSL that tightly integrates with Rust, which allows Datalog rules to call arbitrary Rust code. By using Rust, they support user-defined algebraic data types, user-defined functions, and user-defined recursive lattice-based aggregation. The tight integration of Datalog and Rust allows users to utilize it as a meta programming language. Users utilize Rust to construct and compose large Datalog programs by

implementing higher-order and polymorphic relations. Additionally, Ascent introduces the "Bring Your Own Data Structures" approach for Datalog [Sahebolamri et al. 2023]. This approach allows plugging in new data structures as long as they form a lattice. Using this approach users do not rely on the framework developer to provide specialized data structures such as a union-find data structure. Using specialized data structures can improve performance drastically. Additionally, this approach allows them to support recursive lattice-based aggregation.

Ascent supports a uni-directional integration between Datalog rules and Rust because Datalog rules can call arbitrary Rust functions, but Rust cannot query Datalog relations. In contrast, the linguistic abstraction functional IncA (Chapter 4) supports a bi-directional integration, which we utilize in the interval analysis. To this end, we lower functions and relations both to Datalog. Ascent and functional IncA both support higher-order relations and polymorphic relations to compose large Datalog programs. Ascent uses Rust to define and create user-defined algebraic data where each ADT value is stored within a tuple element. In contrast, we propose a data abstraction for structured data. We encode structured input data relationally to enable efficient incremental updates when using an incremental Datalog engine (Chapter 3). Functional IncA also encodes algebraic data constructed during evaluation relationally. Encoding algebraic data relationally allows functional IncA to enumerate all constructed values during run time, which is not possible with the approach of Ascent.

Flan. Flan [Abeysinghe et al. 2024] is a Datalog system using lightweight modular staging to implement a Datalog engine. Lightweight modular staging [Rompf and Odersky 2010] is a multi-stage programming technique to generate specialized programs based on a given input. In particular, Flan uses the Datalog program as input to generate a specialized Scala program performing semi-naïve evaluation. The authors aim to provide a flexible and extensible Datalog system where users can introduce new join algorithms, such as multi-way joins or index selection strategies. Flan tightly integrates Datalog with Scala code by providing a DSL to construct Datalog programs, which can call arbitrary Scala code. Using the DSL, Scala acts as a meta programming language to construct and compose Flan programs, including polymorphic relations, higher-order relations, or user-defined lattice types. Users can utilize these features to construct large Datalog programs while promoting code reuse.

In contrast, the frontend language functional IncA presented in Chapter 4 provides features that tightly integrate Datalog relations and functions by compiling both to Datalog relations. Flan does not provide such integration as Datalog rules can call Scala functions, but Scala functions cannot access Datalog relations. Flan stores Scala algebraic data type values directly within tuple elements. We provide a data abstraction for structured data by relationally encoding input data and data constructed during evaluation. This has two advantages. First, encoding structured input data relationally enables efficient incremental update times when utilizing Datalog's incremental engines. Second, relationally encoding algebraic data constructed during evaluation enables functional IncA to query all constructed values.

Flix. Flix [Madsen et al. 2016] started out as a Datalog system combining Datalog programs with functions and algebraic data to allow lattice-based recursive aggregation. The authors define their own functional language with algebraic data types. Users can define lattices based on algebraic data types and functions acting on algebraic data. The integration of the functional language and Datalog is uni-directional. Datalog rules can contain expressions of the functional language and construct values of algebraic data types, but functions cannot query Datalog relations. Nowadays, Flix [Madsen and Lhoták 2020] evolved into a meta programming language for Datalog. The authors extend the lambda calculus with Datalog programs as first-class values. Flix can organize and compose large Datalog programs using functional language features such as parametric polymorphism and higher-order functions at run time.

Functional IncA also supports composing large programs using higher-order functions and parametric polymorphism. While Flix extends Datalog with a functional language, Flix only allows to call functions from Datalog rules during Datalog evaluation. Functional IncA allows for an intertwined operation of Datalog rules and functions by compiling both to the same level, namely Datalog. This is one of the strengths of Datalog as an intermediate representation. Another strength of this approach is that we can use existing Datalog engines to execute functional IncA programs. In contrast, Flix provides its own Datalog engine to allow Datalog programs to evaluate expressions of the functional language. Flix allows algebraic data as input as well as constructing it during execution. Flix stores structured data values as whole values within relations. We follow a different approach for two reasons: incrementality and expressivity. We encode structured data relationally by encoding it across multiple relations. First, this approach enables efficient incremental updates when using an incremental Datalog engine. Relationally encoding structured data allows us to change the input in a very detailed manner following the mantra that small changes in structured data should translate to small changes in the relational encoding. Second, this approach allows us to enumerate all constructed structured data values, which is not possible with Flix.

Formulog. Formulog [Bembenek et al. 2020] combines first-order ML functions with Datalog and SMT formulas. A Formulog program can construct SMT formulas during evaluation and invoke an SMT solver to solve the constructed formulas. The system enables a bi-directional integration of Datalog relations and ML expressions. It is possible to evaluate ML expressions within Datalog rules, and ML expressions can query Datalog relations. To achieve this tight integration, Formulog provides a specialized Datalog engine. Additionally, they provide a type system to ensure that SMT formulas and ML expressions cannot get stuck during evaluation.

We propose functional IncA in Chapter 4. Functional IncA achieves the integration of functions and relations differently as compared to Formulog. That is, we compile both to Datalog, which entails that relations and functions all live on the same language level. Hence, we can utilize different Datalog engines and still enable a bi-directional integration of relations and functions. Formulog only provides first-order functions. In contrast, functional IncA supports parametric polymorphism and higher-order functions. This combination allows functional IncA programs to organize and compose large Datalog programs, which is hard only using first-order functions. Formulog can construct algebraic

data during evaluation by storing them as values in relations. We follow a different approach by providing a data abstraction for structured data, which encodes structured data relationally. This enables efficient incremental updates and allows users to enumerate all structured data values during run time.

CodeQL. CodeQL [Avgustinov et al. 2016] is a static analysis tool developed by Semmle, and acquired by GitHub in 2019. CodeQL uses Datalog to statically analyze programs. CodeQL supports programming languages such as Java, C/C++, Go, Ruby, and Python. Each programming language is accompanied by a relational schema, which encodes programs. The relational schema can be queried by Datalog programs. The majority of schemas are generated automatically using the tree-sitter grammar, but in some languages, the relational schema must be defined by hand. CodeQL provides the language QL, which is an object-oriented Datalog language for classes with predicates and inheritance to structure large QL programs [Avgustinov et al. 2016]. In particular, they use Datalog as an intermediate representation and compile QL programs to standard Datalog programs. CodeQL lowers object-oriented features such as classes and inheritance to standard Datalog.

We follow the same approach of using Datalog as an intermediate representation by providing linguistic abstractions for Datalog. In particular, the type system DSL (Chapter 2) and functional IncA (Chapter 4) use Datalog as an intermediate representation. While CodeQL extends standard Datalog with object-oriented features, we provide new frontends for Datalog for specific domains such as type system specifications and general-purpose functional programming. It would be interesting to see how to combine functional IncA with object-oriented features and investigate how they interact when compiling to Datalog. CodeQL provides data abstraction for programs by encoding them relationally, which allows for expressing static analyses against the relational encoding. We propose a similar approach in Chapter 3. However, the design principles we follow focus on enabling efficient incremental updates when using an incremental Datalog engine. These design principles aim to keep changes local. We avoid n-ary relations and only use unary and binary relations. Additionally, our encoding avoids using context information such as names occurring in a program or the position of a list element in the relational encoding. In contrast, CodeQL's encoding uses n-ary relations and context information such as names and positional information in its relational encoding. This encoding hurts incremental performance.

Bloom. Bloom [Alvaro et al. 2011; Conway et al. 2012] is a language for distributed systems that has Datalog as its foundation, in particular, the Datalog dialect Dedalus [Alvaro et al. 2010b]. Dedalus extends Datalog with state, which they encode using timestamps. Bloom provides built-in functions, such as map and reduce, that operate over collections. Bloom is embedded in Ruby, and users can define Ruby functions and data types that are accessible from Bloom.

Bloom follows the same idea of linguistic abstraction for Datalog but considers a different domain. While we propose abstractions for the domains of type system specifications and general-purpose functional programming, Bloom considers distributed computing. Functional IncA can describe functions and relations that are intertwined. That means func-

tions can query relations, and relations can call functions. This bi-directional interaction is not possible with Bloom as mentioned above.

Soufflé. Soufflé [Scholz et al. 2015, 2016] is a Datalog system that can execute Datalog programs directly or compile them to C++. Soufflé focuses on executing Datalog programs in parallel. The authors introduce multiple indexing structures such as b-trees [Jordan et al. 2019b], bries [Jordan et al. 2019a], or indexing structures for equality relations [Nappa et al. 2019] focusing on parallel execution. The Datalog dialect that Soufflé provides supports user-defined functions implemented as external C++ functions, non-recursive aggregation, and algebraic data types. The language provides a component system including subtyping and parametric polymorphism to promote code reuse, enabling organizing and constructing large Soufflé programs. Soufflé provides provenance-based debugging, which uses derivation trees to highlight how a given tuple has been derived [Zhao et al. 2020]. The debugging technique focuses on providing minimal-height derivation trees. This approach avoids exploring the derivation of infinite height for recursive rules on cyclic data. Soufflé introduces a new provenance lattice that stores derivation annotations including the rule deriving the tuple and the minimal height of the derivation tree. Additionally, they describe a new bottom-up evaluation strategy that derives tuples of the provenance lattice. Soufflé supports elastic incremental execution [Zhao et al. 2021], which re-computes from scratch if the incoming change has a high impact on the output. Otherwise, they process the change incrementally.

Soufflé exposes the user to standard Datalog with extensions. In this dissertation, we propose to provide linguistic abstractions for Datalog. In particular, we define a type system DSL and a general-purpose functional language called functional IncA, which compile to Datalog. Functional IncA promotes code reuse by providing features such as higher-order functions, higher-order relations, and parametric polymorphism. Soufflé, on the other hand, builds a component system on top of standard Datalog, which functional IncA could benefit from as well. Soufflé allows to construct algebraic data during evaluation by storing such values as a whole within a tuple. In contrast, we propose a data abstraction for structured data in this dissertation. That is, for structured input data, we encode it relationally and follow design principles that enable efficient incremental processing. When constructing algebraic data during the evaluation of functional IncA programs, we relationally encode algebraic data values. This allows functional IncA to enumerate and query all constructed values during evaluation and operate on these values. Soufflé provides an operational abstraction for Datalog, which is provenance-based. Their approach requires a complete tuple when inspecting a derivation tree. In contrast, we propose a new view on debugging Datalog programs. That is, we develop a top-down stepping debugger, which allows exploring the execution of partial queries. Soufflé supports exploring why a tuple is not derivable in a semi-automated way. Users provide the applicable rule and a substitution for all free variables. We follow a different approach by letting users follow each step of the top-down evaluation of an unjustifiable query, which is fully automatic.

IncA$_{\text{MPS}}$. IncA$_{\text{MPS}}$ [Szabó et al. 2016] is an incremental static analysis framework using Datalog as a surface language. Internally, it uses the Datalog engine ViatraQuery [Ujhelyi et al. 2015]. The authors extend ViatraQuery with the incremental Datalog algorithm

DRed_L to support lattice-based recursive aggregation [Szabó et al. 2018]. Additionally, they develop an incremental Datalog algorithm based on differential dataflow to scale lattice-based analyses to whole-program analyses [Szabó et al. 2021]. IncA_{MPS} is built on top of the projectional language framework MPS [1] developed by JetBrains. Projectional editing allows users to change the abstract syntax tree directly instead of editing text, which provides precise changes of the syntax tree. IncA_{MPS} encodes the abstract syntax trees relationally. Users can write static program analyses using a language strongly resembling Datalog while supporting user-defined data types, user-defined functions, conditionals, and pattern matching. User-defined data types and functions must be implemented in a JVM language, in particular, an MPS-enhanced version of Java. Based on the relational encoding, users can query the relational encoding of structured data, hence writing an analysis against a syntactic language definition.

We implement the type system DSL presented in Section 2.6 on top of IncA_{MPS}, which poses as a linguistic abstraction for incremental Datalog. In particular, the type system DSL builds on top of MPS language definitions. The compiler is a sequence of MPS transformations targeting the surface language of IncA_{MPS}. We re-implement IncA in Scala2 and integrate the diffing algorithm truediff [Erdweg et al. 2021] to avoid the need for projectional editing. We use the diffing algorithm to produce edit scripts, which we translate to changes in the relational encoding of the AST. Additionally, the re-implementation of IncA aims to be a general-purpose Datalog framework instead of a static analysis framework. We develop functional IncA as a new frontend. In contrast, IncA_{MPS} provides a shallow abstraction for Datalog-like pattern functions [Szabó et al. 2016] separating the notion of "input" and "output" of relations. Users are required to ensure that all parameters, including input and output, are positively bound within each body. That is, each body is range restricted. Functional IncA utilizes the demand transformation [Tekle and Liu 2010] to automatically derive range-restricted rules. In addition, they provide extensions to Datalog, including if constraints and pattern matching. Functional IncA provides these features as well, but it also supports more features, including higher-order functions, higher-order relations, and first-class relations. Functional IncA allows to define functions and relations in the same language. The frontend of IncA_{MPS} separates the language of user-defined data types and functions from the language to define relations. The system does not allow querying relations within functions, which is different from the bi-directional integration functional IncA provides. IncA_{MPS} provides a data abstraction for encoding ASTs provided by MPS relationally, which follows a very similar idea of how $\text{IncA}_{\text{Scala}}$ encodes structured data shown in Chapter 3. However, IncA_{MPS} does not relationally encode algebraic data constructed during evaluation, which functional IncA supports.

Datafun. Datafun is a higher-order functional programming language with sets and least fixpoint semantics [Arntzenius and Krishnaswami 2016]. It is not a Datalog system but a programming language with its own least fixpoint semantics inspired by Datalog. It is closely related to the work we develop in this dissertation, specifically to functional IncA. In particular, they aim to combine Datalog's least fixpoint semantics and higher-order functional programming. Datafun allows computing fixpoints over lattices and relations, which users can define by developing functions that produce sets. Higher-order

[1] `https://www.jetbrains.com/mps/`

programming allows Datafun to support abstractions for Datalog-like computations such as first-class and higher-order relations. These features promote code reuse to organize large programs. Datafun programs can potentially diverge. To ensure the termination of Datafun programs, it provides a type system that tracks the monotonicity of functions. The original semantics is inspired by the naïve evaluation strategy of Datalog [Arntzenius and Krishnaswami 2016]. To speed up the evaluation of Datafun programs, they develop a new semantics inspired by semi-naïve evaluation of Datalog [Arntzenius and Krishnaswami 2020].

Datafun is closely related to functional IncA. Datafun is not a linguistic abstraction for Datalog but also provides functional abstractions for fixpoint computations. Both systems allow a tight integration of relations and functions where functions can query relations and relations can call functions. Additionally, both support first-class relations, higher-order relations, and polymorphic relations to promote code reuse when developing large programs. One key difference is that we compile functional IncA to Datalog instead of defining a new runtime system. The advantage of this approach is that we can utilize techniques of Datalog evaluation such as semi-naïve evaluation and incremental algorithms such as DRed [Gupta et al. 1993] or LADDDER [Szabó et al. 2021]. However, Datafun defines semi-naïve evaluation of Datafun programs themselves, and an incremental Datafun semantics does not exist at all. Additionally, we can use different Datalog engines to execute the generated program. To this end, we need to compile the intermediate representation program to the Datalog dialect of the targeted Datalog engine. Datafun allows users to define algebraic data types, however, they do not encode them relationally. In contrast, we provide a data abstraction for structured input data that relationally encodes structured data such as abstract syntax trees. Additionally, functional IncA allows constructing algebraic data during run time which we encode relationally as well. Based on the relational encoding, we can enumerate all algebraic data during run time, which is not possible with Datafun. Datafun provides a type system that ensures that well-typed Datafun programs terminate. In contrast, not all well-typed functional IncA programs terminate. Instead, users have to ensure the termination of functional IncA programs manually. Datafun is more restrictive than functional IncA but provides stronger termination guarantees.

Doop. Doop is an industry-strength points-to analysis framework for Java programs that provides highly configurable points-to analyses written in Datalog [Bravenboer and Smaragdakis 2009b; Smaragdakis and Bravenboer 2010]. In particular, Doop uses Soufflé Datalog to describe static analyses. Doop provides a data abstraction for JVM bytecode by encoding programs relationally. The Doop analyses query the relational encoding of JVM bytecode. However, the encoding does not consider incrementality. Doop's encoding uses context-sensitive information as identifiers and positional information of list elements. Additionally, Doop uses large relations instead of only using unary and binary relations to encode bytecode. All these design decisions hurt incremental update times because small changes in the input program could translate to large changes in the relational encoding. In contrast, we propose an approach to encode structured data enabling efficient incremental update times. Our approach follows three design principles: (i) only use context-insensitive unique identifiers, (ii) only use unary and binary relations, and (iii) use relative position for list elements.

Gigahorse and MadMax. Gigahorse is a decompiler toolchain for smart contracts operating on the Ethereum virtual machine (EVM) [Grech et al. 2019]. The decompiler compiles EVM bytecode to a high-level 3-address code representation. In particular, the decompiler itself is defined in Datalog. The decompilation target is a relational encoding. Based on this decompilation, the analysis framework MadMax provides static analyses for EVM bytecode [Grech et al. 2018].

In Chapter 3, we propose a data abstraction for structured data. That is, we encode structured input data such as abstract syntax trees relationally. The relation encoding is the basis of writing a Datalog program against abstract syntax trees. In particular, incremental type system specifications and static analyses written in Datalog. Our encoding aims to enable efficient incremental updates for propagating changes using incremental Datalog algorithms such as DRed [Gupta et al. 1993] or LADDDER [Szabó et al. 2021]. Since Gigahorse does not focus on incrementality, they do not follow the design principles we re-iterated when discussing the Doop framework. In particular, they use context-sensitive information, relations ranging over more than two arguments, and positional information to encode lists such as argument positions.

7

Conclusion and Future Work

In this chapter, we consider the thesis of this dissertation again and evaluate if the thesis holds. We conclude this dissertation by discussing interesting future research directions.

7.1 Evaluating this Dissertation's Thesis

Datalog's strengths such as its declarative least fixpoint semantics, and the fact that Datalog offers efficient non-incremental and incremental engines caught the interest of academia and industry in the last decade. Despite Datalog's strengths, we identified shortcomings of Datalog. That is, we identified a representational gap between Datalog's low-level programming style and the mental model of domain experts when solving problems within a problem domain. In particular, there are three representational gaps. First, we identified a representational gap between computations domain experts want to express and Datalog's low-level description, which is rule-based (RG1) Computation. Second, we identified a representational gap between data that computations of the domain operate on and Datalog's way of using relations to describe inputs and outputs (RG2) Data. Based on these two representational gaps, we identified a third one. In particular, because of the representational gap between computations/data of the problem domain and Datalog's relational style, observing the behavior of computations and identifying bugs is not aligned with the mental model of domain experts (RG3) Observable Behavior. Based on these representational gaps, we claimed the following thesis:

> It is feasible and useful to provide programming abstractions for Datalog to close representational gaps by shielding programmers and users from logic programming.

To close the representational gaps, we propose to view Datalog as an intermediate representation language instead of a user-facing language. Based on this idea, we developed linguistic abstractions, a data abstraction, and an operational abstraction to close the representational gaps of computations, data, and observable behavior respectively.

We developed two linguistic abstractions for different domains. First, we developed a

type system DSL that compiles to Datalog and an accompanying compiler that employs typing-specific optimizations to enable efficient incremental updates. The compiler enables deriving incremental type checkers systematically. We showed that it is possible to derive incremental type checkers for PCF, bi-directional type systems, System F, and support overloading. In particular, we evaluated the performance of an incremental type checker for PCF and showed that it achieves speedups up to 25x. Note that typing rules and Datalog rules are conceptually very similar. But to enable efficient incremental updates, users require deep knowledge of incremental Datalog algorithms. Our compiler allows users to derive incremental type checkers without having this knowledge. Second, we developed a functional Datalog frontend combining functional programming with fixpoint computations. The frontend allows programmers to use functional abstractions when describing fixpoint computations by supporting higher-order relations, first-class relations, and polymorphic relations, which is not possible using plain Datalog. These abstractions promote code reuse when developing large Datalog programs. We showed that the functional frontend is usable to implement functional programs such as type checkers, type erasure, and interpreters. Additionally, we showed that the new frontend allows describing concise flow-sensitive data-flow analyses such as reaching definitions and interval analysis or code clone analysis. Both linguistic abstractions differ in their purpose. The type system DSL abstracts away Datalog-specific transformations required to enable efficient incrementality and the functional Datalog frontend abstracts away from Datalog's surface language.

We developed a data abstraction for structured data which enables efficient incremental updates when using an incremental Datalog engine. To this end, we identified design principles that allow translating small changes in the structured input data to small changes in the relational encoding. We applied the abstraction to encode abstract syntax trees that the incremental type checkers operate on. The relational encoding enabled efficient incremental updates for the incremental type checkers. Additionally, it has been used to enable other incremental Datalog programs such as strong-update points-to and interval analysis for Java.

At last, we developed a stepping debugger for Datalog that enables observing Datalog execution in a top-down fashion, which is different from the standard bottom-up evaluation strategy for Datalog. In particular, we designed a small-step operational semantics for top-down evaluated Datalog as a base for the debugger. The operational semantics defines each observable intermediate state when evaluating Datalog top-down. Based on the top-down debugger implementation, we lift Datalog's intermediate states to intermediate states of functional IncA programs. Hence, we enabled debugging programs that compile to Datalog by connecting intermediate states between the source and target language.

To conclude, we argue that the thesis of this dissertation holds. That is, it is feasible to provide abstractions for Datalog that close the representational gap between the problem domain and Datalog's logic programming. But we can still utilize Datalog's strengths such as its declarative least fixpoint semantics and its efficient incremental engines.

7.2 Future Work

In this section, we discuss future research directions that are based on the provided solution to the thesis of this dissertation.

Generalizing Optimizations of Type System Compiler. We developed an optimizing compiler for the type system DSL, which aims for efficient incremental updates. The optimizations include context fusion to avoid deleting and re-deriving typing information because of an updated context, even though the context did not affect the resulting type. Additionally, we implemented an error collection transformation that separates deriving typing information and error collection. Both optimizations aim to keep the typing relation as stable as possible across incremental runs. We want to investigate if it is possible to generalize these optimizations and apply them to the functional Datalog frontend as well. Additionally, we want to explore if other optimizations apply to functional IncA programs to enable efficient incremental execution. Applying such optimizations could lead to efficient incremental execution of functional IncA programs.

Other Linguistic Abstractions. In this dissertation, we developed two linguistic abstractions. First, we developed a type system DSL for Datalog to derive incremental type checker implementations. Second, we developed a functional frontend for Datalog, which combines functional programming abstractions and fixpoint computations. We want to explore other linguistic abstractions for Datalog. For example, it would be interesting to investigate how to combine other programming paradigms with fixpoint computations. In particular, exploring how to combine object-oriented features such as class hierarchies, constructing objects at run time, and dynamic dispatch with fixpoint computations. There already exist Datalog dialects that introduce object-oriented features such as CodeQL [Avgustinov et al. 2016], but they do not allow constructing objects at run time. For CodeQL, all class objects are predefined as part of the input. Additionally, it is interesting to introduce a Datalog frontend with local and non-local mutation, because Datalog describes monotonic computations and mutation describes non-monotonic destructive updates.

Selective Compilation to Datalog. For the functional Datalog frontend, we compile all functions to Datalog. This encoding strategy can impact the performance of the generated Datalog programs, especially for functions not contributing to a fixpoint computation. Consider a definitional interpreter that is a recursive-descent function traversing an expression tree. We believe that executing a definitional interpreter directly will be faster than executing the generated Datalog program. Based on this observation, we want to explore how to compile functions to Datalog while compiling other functions to primitive functions. To tackle this problem, we need to answer the following research question: which functions are required to be compiled to Datalog, and how to detect them automatically? We hope selective compilation to Datalog leads to better performance when using non-incremental Datalog engines. However, it could hurt incremental execution because only Datalog relations are maintained incrementally, and we re-compute primitive functions from scratch during an incremental run.

Datalog as an IR. In this dissertation, we propose to view Datalog as an intermediate representation instead of using it as a surface language. We want to explore if we can push this idea further by providing a Datalog IR framework similar to LLVM [Lattner and Adve 2004]. In particular, we want to follow the idea of a multi-level IR that LLVM introduced [Lattner et al. 2021]. We want to develop a multi-level IR for Datalog, which provides expressive extensions to the base IR. Such extensions could include algebraic data types, sets, and tuples. Each extension comes with an isolated transformation that lowers to Datalog, which can include other extensions. Thus, the compiler progressively lowers Datalog until only core Datalog remains. At the end, we can compile core Datalog programs to specific target Datalog dialects such as ViatraQuery [Varró et al. 2016], Soufflé [Scholz et al. 2016], DDlog [Ryzhyk and Budiu 2019], and Ascent [Sahebolamri et al. 2022, 2023]. By providing expressive Datalog extensions, implementing Datalog frontends becomes easier because developers can reuse provided extensions. But more importantly, developers do not need to implement a monolithic compiler that considers encoding all features simultaneously because the transformations for each extension act in isolation. Additionally, this design enables us to explore different encoding strategies for extensions such that we can explore how encoding strategies affect incremental performance when using algorithms such as DRed [Gupta et al. 1993] and LADDDER [Szabó et al. 2021].

Bibliography

References

2024. Apache Spark. https://spark.apache.org/ [Accessed: 26.02.2024].

2024. Logica. https://logica.dev/ [Accessed: 26.02.2024].

2024. Rel Language. https://docs.relational.ai/getting-started/walkthrough/rel-language [Accessed: 26.02.2024].

Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. 1996. Analysis and Caching of Dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 83–91. https://doi.org/10.1145/232627.232638

Supun Abeysinghe, Anxhelo Xhebraj, and Tiark Rompf. 2024. Flan: An Expressive and Efficient Datalog Compiler for Program Analysis. *Proc. ACM Program. Lang.* 8, POPL, Article 86 (jan 2024), 33 pages. https://doi.org/10.1145/3632928

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

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley. http://webdam.inria.fr/Alice/

Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. 2019. Fixing Incremental Computation - Derivatives of Fixpoints, and the Recursive Semantics of Datalog. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 525–552. https://doi.org/10.1007/978-3-030-17184-1_19

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

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. 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*. www.cidrdb.org, 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2010b. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6702)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.). Springer, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16

Larse Ole Andersen. 1994. Program Analysis and Specialization for the C Programming Language. *PhD Thesis, University of Copenhagen* (1994).

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

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

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

Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. 1993. Explaining Program Execution in Deductive Systems. In *Deductive and Object-Oriented Databases, Third International Conference, DOOD'93, Phoenix, Arizona, USA, December 6-8, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 760)*, Stefano Ceri, Katsumi Tanaka, and Shalom Tsur (Eds.). Springer, 101–119. https://doi.org/10.1007/3-540-57530-8_7

Isabelle Attali, Jacques Chazarain, and Serge Gilette. 1992. Incremental Evaluation of Natural Semantics Specification. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 631)*, Maurice Bruynooghe and Martin Wirsing (Eds.). Springer, 87–99. https://doi.org/10.1007/3-540-55844-6_129

Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele Jr., and Tim Sweeney. 2023. The Verse Calculus: A Core

Calculus for Deterministic Functional Logic Programming. *Proc. ACM Program. Lang.* 7, ICFP (2023), 417–447. https://doi.org/10.1145/3607845

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

Darshana Balakrishnan, Carl Nuessle, Oliver Kennedy, and Lukasz Ziarek. 2021. TreeToaster: Towards an IVM-Optimized Compiler. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 155–167. https://doi.org/10.1145/3448016.3459244

Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *J. Log. Program.* 10, 3&4 (1991), 255–299. https://doi.org/10.1016/0743-1066(91)90038-Q

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

William C. Benton and Charles N. Fischer. 2007. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wroclaw, Poland*, Michael Leuschel and Andreas Podelski (Eds.). ACM, 13–24. https://doi.org/10.1145/1273920.1273923

Martin Bravenboer and Yannis Smaragdakis. 2009a. Exception analysis and points-to analysis: better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, Gregg Rothermel and Laura K. Dillon (Eds.). ACM, 1–12. https://doi.org/10.1145/1572272.1572274

Martin Bravenboer and Yannis Smaragdakis. 2009b. Strictly declarative specification of sophisticated points-to analyses. In *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*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2019. Using Standard Typing Algorithms Incrementally. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11460)*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer, 106–122. https://doi.org/10.1007/978-3-030-20652-9_7

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2008a. A New Proposal for Debugging Datalog Programs. *Electron. Notes Theor. Comput. Sci.* 216 (2008), 79–92. https://doi.org/10.1016/j.entcs.2008.06.035

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2008b. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *Semantics in Data and Knowledge Bases, Third International Workshop, SDKB 2008, Nantes, France, March 29, 2008, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4925)*, Klaus-Dieter Schewe and Bernhard Thalheim (Eds.). Springer, 143–159. https://doi.org/10.1007/978-3-540-88594-8_8

James Cheney, Sam Lindley, and Philip Wadler. 2014. Query shredding: efficient relational evaluation of queries over nested multisets. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1027–1038. https://doi.org/10.1145/2588555.2612186

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, Michael J. Carey and Steven Hand (Eds.). ACM, 1. https://doi.org/10.1145/2391229.2391230

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

Thierry Despeyroux. 1984. Executable Specification of Static Semantics. In *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings (Lecture Notes in Computer Science, Vol. 173)*, Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin (Eds.). Springer, 215–233. https://doi.org/10.1007/3-540-13346-1_11

Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. 2007. Automatic Incrementalization of Prolog Based Static Analyses. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007 (Lecture Notes in Computer Science, Vol. 4354)*, Michael Hanus (Ed.). Springer, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015a. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. https://doi.org/10.1145/2814270.2814277

Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015b. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 89–106.

Sebastian Erdweg, Tamás Szabó, and André Pacak. 2021. Concise, type-safe, and efficient structural diffing. In *PLDI '21: 42nd ACM SIGPLAN International Conference*

on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, Stephen N. Freund and Eran Yahav (Eds.). ACM, 406–419. https://doi.org/10.1145/3453483.3454052

Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-up in-memory datalog processing: observations and techniques. *Proc. VLDB Endow.* 12, 6 (feb 2019), 695–708. https://doi.org/10.14778/3311880.3311886

Frantisek Farka, Ekaterina Komendantskaya, and Kevin Hammond. 2018. Proof-relevant Horn Clauses for Dependent Type Inference and Term Synthesis. *Theory Pract. Log. Program.* 18, 3-4 (2018), 484–501. https://doi.org/10.1017/S1471068418000212

Sérgio Fernandes and Jorge Bernardino. 2015. What is bigquery?. In *Proceedings of the 19th International Database Engineering & Applications Symposium*. 202–203.

Luca Franceschini, Davide Ancona, and Ekaterina Komendantskaya. 2016. Structural Resolution for Abstract Compilation of Object-Oriented Languages. In *Proceedings of the First Workshop on Coalgebra, Horn Clause Logic Programming and Types, CoALP-Ty 2016, Edinburgh, UK, 28-29 November 2016 (EPTCS, Vol. 258)*, Ekaterina Komendantskaya and John Power (Eds.). 19–35. https://doi.org/10.4204/EPTCS.258.2

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 116:1–116:27. https://doi.org/10.1145/3276486

Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: advanced decompilation of Ethereum smart contracts. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527321

Todd J. Green. 2015. LogiQL: A Declarative Language for Enterprise Applications. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (<conf-loc>, <city>Melbourne</city>, <state>Victoria</state>, <country>Australia</country>, </conf-loc>) *(PODS '15)*. Association for Computing Machinery, New York, NY, USA, 59–64. https://doi.org/10.1145/2745754.2745780

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Foundations and Trends in Databases* 5, 2 (2013), 105–195. https://doi.org/10.1561/1900000017

Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. 2015. Type systems for the masses: deriving soundness proofs and efficient checkers. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Gail C. Murphy and Guy L. Steele Jr. (Eds.). ACM, 137–150. https://doi.org/10.1145/2814228.2814239

Ashish Gupta and Inderpal Singh. Mumick. 1999. *Materialized views : techniques, implementations, and applications.* MIT Press. 589 pages. https://mitpress.mit.edu/books/materialized-views

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 157–166. https://doi.org/10.1145/170035.170066

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. *codeQuest:* Scalable Source Code Queries with Datalog. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4067)*, Dave Thomas (Ed.). Springer, 2–27. https://doi.org/10.1007/11785477_2

Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 748–766. https://doi.org/10.1145/2814270.2814305

Michael Hanus. 1997. Curry: A Multi-Paradigm Declarative Language (system description). In *Twelfth Workshop Logic Programming, WLP 1997, 17-19 September 1997, München, Germany, Technical Report PMS-FB-1997-10*, François Bry, Burkhard Freitag, and Dietmar Seipel (Eds.). Ludwig Maximilians Universität München.

M. Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 123–168.

Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press. https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

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

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019a. Brie: A Specialized Trie for Concurrent Datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2019,*

Washington, DC, USA, February 17, 2019, Quan Chen, Zhiyi Huang, and Min Si (Eds.). ACM, 31–40. https://doi.org/10.1145/3303084.3309490

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019b. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 327–339. https://doi.org/10.1145/3293883.3295719

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 75–90. https://doi.org/10.1145/2902251.2902286

Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. 2012. Declarative Datalog Debugging for Mere Mortals. In *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7494)*, Pablo Barceló and Reinhard Pichler (Eds.). Springer, 111–122. https://doi.org/10.1007/978-3-642-32925-8_12

Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. ACM Press, New York, New York, USA, 76–86. https://doi.org/10.1145/3238147.3238196

Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:26. https://doi.org/10.4230/LIPIcs.ECOOP.2017.18

Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise static modeling of Ethereum "memory". *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 190:1–190:26. https://doi.org/10.1145/3428258

Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, Chen Li (Ed.). ACM, 1–12. https://doi.org/10.1145/1065167.1065169

Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

Craig Larman. 2001. Applying UML and pattern: an introduction to object oriented analysis and design and the unified process. (2001).

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Georg Lausen, Bertram Ludäscher, and Wolfgang May. 1998. On Active Deductive Databases: The Statelog Approach. In *Transactions and Change in Logic Databases, International Seminar on Logic Databases and the Meaning of Change, Schloss Dagstuhl, Germany, September 23-27, 1996 and ILPS '97 Post-Conference Workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases, (DYNAMICS'97) Port Jefferson, NY, USA, October 17, 1997, Invited Surveys and Selected Papers (Lecture Notes in Computer Science, Vol. 1472)*, Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov (Eds.). Springer, 69–106. https://doi.org/10.1007/BFb0055496

Yanhong A. Liu and Tim Teitelbaum. 1995. Systematic Derivation of Incremental Programs. *Sci. Comput. Program.* 24, 1 (1995), 1–39. https://doi.org/10.1016/0167-6423(94)00031-9

Ewa Madalinska-Bugaj and Linh Anh Nguyen. 2008. Generalizing the QSQR Evaluation Method for Horn Knowledge Bases. In *New Challenges in Applied Intelligence Technologies*, Ngoc Thanh Nguyen and Radoslaw P. Katarzyniak (Eds.). Studies in Computational Intelligence, Vol. 134. Springer, 145–154. https://doi.org/10.1007/978-3-540-79355-7_14

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

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

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

Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013,*

Asilomar, CA, USA, January 6-9, 2013, Online Proceedings. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

Lambert G. L. T. Meertens. 1983. Incremental Polymorphic Type Checking in B. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum (Eds.). ACM Press, 265–275. https://doi.org/10.1145/567067.567092

Michael Meier, Michael Schmidt, and Georg Lausen. 2009. On chase termination beyond stratification. *Proc. VLDB Endow.* 2, 1 (aug 2009), 970–981. https://doi.org/10.14778/1687627.1687737

Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. i3QL: Language-Integrated Live Data Views. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 417–432.

Patrick Nappa, David Zhao, Pavle Subotic, and Bernhard Scholz. 2019. Fast Parallel Equivalence Relations in a Datalog Compiler. In *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, 82–96. https://doi.org/10.1109/PACT.2019.00015

Wolfgang Nejdl. 1987. Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, Peter M. Stocker, William Kent, and Peter Hammersley (Eds.). Morgan Kaufmann, 43–50. http://www.vldb.org/conf/1987/P043.PDF

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

André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.7

André Pacak and Sebastian Erdweg. 2023. Interactive Debugging of Datalog Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 745–772. https://doi.org/10.1145/3622824

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

André Pacak, Tamás Szabó, and Sebastian Erdweg. 2022. Incremental Processing of Structured Data in Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, Bernhard Scholz and Yukiyoshi Kameyama (Eds.). ACM, 20–32. https://doi.org/10.1145/3564719.3568686

Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: a data platform based on the scaling-up approach. *Proc. VLDB Endow.* 11, 6 (feb 2018), 663–676.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

William W. Pugh and Tim Teitelbaum. 1989. Incremental Computation via Function Caching. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989.* ACM Press, 315–328. https://doi.org/10.1145/75277.75305

Raghu Ramakrishnan, François Bancilhon, and Abraham Silberschatz. 1987. Safety of Recursive Horn Clauses With Infinite Relations. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, USA*, Moshe Y. Vardi (Ed.). ACM, 328–339. https://doi.org/10.1145/28659.28694

G Ramalingam and Thomas Reps. 1993. A categorized bibliography on incremental computation. In *Proceedings of Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, New York, USA, 502–510. https://doi.org/10.1145/158511.158710

Alan Rector and Jeremy Rogers. 2006. Ontological and Practical Issues in Using a Description Logic to Represent Medical Concept Systems: Experience from GALEN. 197–231. https://doi.org/10.1007/11837787_9

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10).* Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

Francesco Russo and Mirko Sancassani. 1991. A Declarative Debugging Environment for DATALOG. In *Logic Programming, First Russian Conference on Logic Programming, Irkutsk, Russia, September 14-18, 1990 - Second Russian Conference on Logic Programming, St. Petersburg, Russia, September 11-16, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 592)*, Andrei Voronkov (Ed.). Springer, 433–441. https://doi.org/10.1007/3-540-55460-2_32

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019 (CEUR Workshop Proceedings, Vol. 2368)*, Mario Alviano and Andreas Pieris (Eds.). CEUR-WS.org, 56–67. http://ceur-ws.org/Vol-2368/paper6.pdf

Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 264 (oct 2023), 26 pages. https://doi.org/10.1145/3622840

Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. 2022. Seamless deductive inference via macros. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 77–88. https://doi.org/10.1145/3497776.3517779

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

Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. 2015. 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*. IEEE Computer Society, 28–37. https://doi.org/10.1109/ASWEC.2015.15

Jiwon Seo, Stephen Guo, and Monica S. Lam. 2013a. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 278–289. https://doi.org/10.1109/ICDE.2013.6544832

Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013b. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1906–1917. https://doi.org/10.14778/2556549.2556572

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6702)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.). Springer, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2020. Scalable Querying of Nested Data. *Proc. VLDB Endow.* 14, 3 (2020), 445–457. https://doi.org/10.5555/3430915.3442441

Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *J. Log. Program.* 29, 1-3 (1996), 17–64. https://doi.org/10.1016/S0743-1066(96)00068-4

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

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1–15. https://doi.org/10.1145/3453483.3454026

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

Tamás Szabó. 2021. *Incrementalizing Static Analyses in Datalog*. Ph. D. Dissertation. Mainz. https://doi.org/10.25358/openscience-5613

K. Tuncay Tekle and Yanhong A. Liu. 2010. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández (Eds.). ACM, 35–44. https://doi.org/10.1145/1836089.1836094

K. Tuncay Tekle and Yanhong A. Liu. 2019. Extended Magic for Negation: Efficient Demand-Driven Evaluation of Stratified Datalog with Precise Complexity Guarantees. In *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019 (EPTCS, Vol. 306)*, Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang (Eds.). 241–254. https://doi.org/10.4204/EPTCS.306.28

Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98 (2015), 80–99. https://doi.org/10.1016/j.scico.2014.01.004

Zoltán Ujhelyi, Gábor Bergmann, and Dániel Varró. 2016. Rete Network Slicing for Model Queries. In *Graph Transformation - 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9761)*, Rachid Echahed and Mark Minas (Eds.). Springer, 137–152. https://doi.org/10.1007/978-3-319-40530-8_9

Jeffrey D. Ullman. 1989. Bottom-Up Beats Top-Down for Datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, Avi Silberschatz (Ed.). ACM Press, 140–149. https://doi.org/10.1145/73721.73736

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

Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling* 15, 3 (01 Jul 2016), 609–629. https://doi.org/10.1007/s10270-016-0530-4

Laurent Vieille. 1986. Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In *Expert Database Systems, Proceedings From the First International Conference*. Benjamin/Cummings, 253–267.

Laurent Vieille. 1987. A Database-Complete Proof Procedure Based on SLD-Resolution. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, Victoria, Australia, May 25-29, 1987 (2 Volumes)*, Jean-Louis Lassez (Ed.). MIT Press, 74–103.

Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. 2013. A Language Independent Task Engine for Incremental Name and Type Analysis. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, 260–280. https://doi.org/10.1007/978-3-319-02654-1_15

Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Tim A. Wagner and Susan L. Graham. 1997. Incremental Analysis of real Programming Languages. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, USA, June 15-18, 1997*, Marina C. Chen, Ron K. Cytron, and A. Michael Berman (Eds.). ACM, 31–43. https://doi.org/10.1145/258915.258920

Adrienne Watt. 2018. *Database design.*

John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William W. Pugh and Craig Chambers (Eds.). ACM, 131–144. https://doi.org/10.1145/996841.996859

Christian A. Wieland. 1990. Two Explanation Facilities for the Deductive Database Management System DeDEx. In *Proceedings of the 9th International Conference on Entity-Relationship Approach (ER'90), 8-10 October, 1990, Lausanne, Switzerland*, Hannu Kangassalo (Ed.). ER Institute, 189–203.

Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines. In *Proceedings of the 2022 International Conference*

on Management of Data (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1433–1446. https://doi.org/10.1145/3514221.3517853

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (jun 2023), 25 pages. https://doi.org/10.1145/3591239

David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, Niccolò Veltri, Nick Benton, and Silvia Ghilezan (Eds.). ACM, 20:1–20:16. https://doi.org/10.1145/3479394.3479415

David Zhao, Pavle Subotic, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 7:1–7:35. https://doi.org/10.1145/3379446