A Systematic Approach to Deriving Incremental Type Checkers

ANONYMOUS AUTHOR(S)

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 paper, 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 solvers exist. The key contribution of this paper 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 and show that it supports simple types, some local type inference, operator overloading, and universal types.

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 paper presents a systematic approach for *deriving* incremental type checkers from textbook-style type system specifications. 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 solvers already exist [Ujhelyi et al. 2015]. 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.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Based on our transformations, we developed a domain-specific language (DSL) for type system descriptions that compiles to Datalog. We have used the DSL to express a wide range of type systems feature. In addition to PCF with product and sum types, we modeled bi-directional type checking, operator overloading, and universal types in the style of System F. 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).
- We introduce co-functional dependencies and define a Datalog transformation that moves co-functionally dependent data into a separate relation (Section 3).
- We show how to eliminate typing context propagation from type systems (Section 4).
- We show how to transform a type system to collect type errors on the side (Section 5).
- We implement all three transformations in the compiler of a type systems DSL (Section 6), demonstrate its applicability (Section 7), and benchmark its performance (Section 8).

2 WHY ARE TYPE SYSTEMS IN DATALOG CHALLENGING?

This paper 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 solvers 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:

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

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 solver 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.

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

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:

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

A Datalog program consists of a sequence of rules, each of the form $\mathsf{R}(t_1, \ldots, t_n) \text{ :- } \mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{m}$. The rule head $\mathsf{R}(t_1, \ldots, t_n)$ declares tuple $(t_1, \ldots, t_n) \in \mathsf{R}$ if all atoms $\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{m}$ 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 $\mathsf{R}(t_1, \ldots, t_n)$. This way, relations can depend on each other recursively. In this paper, 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 $!\mathsf{c}(y, x_1, \ldots, x_k)$ and $?\mathsf{c}(y, x_1, \ldots, x_k)$ to construct and deconstruct algebraic data x.

Given this Datalog background, it should be easy to see that $(e, T) \in \mathtt{typed}$ 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 solvers to the derived Datalog program. To this effect, it is important to know that incremental Datalog solvers evaluate Datalog rules *bottom-up*, inductively enumerating *all derivable tuples*. When the input changes, an incremental Datalog solver 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:

$$\mathtt{typed} = \{(\mathbf{unit}, \mathbf{Unit}), ((\mathbf{unit}\ \mathbf{unit}), \mathbf{Unit}), (((\mathbf{unit}\ \mathbf{unit})\ \mathbf{unit}), \mathbf{Unit}), \ldots\}$$

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 constant sets 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.

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:

$$\text{T-Unit}\ \frac{}{\mathbf{unit} : \mathbf{Unit}} \qquad \text{T-App}\ \frac{e_1 : \mathbf{Unit} \longrightarrow T \qquad e_2 : \mathbf{Unit}}{e_1\ e_2 : T}$$

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

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

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

Again, we must ask if these rules can be evaluated bottom-up by an incremental Datalog solver. 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 $T ::= \mathbf{Unit} \mid \mathbf{Unit} \longrightarrow T$ to expressions. Notably, this domain is infinite and relation $\mathtt{typed} \subseteq \mathtt{e}_{user} \times T$ could contain infinitely many tuples even if $\mathtt{e}_{user}$ 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 $(e, T1) \in$ `typed` and $(e, T2) \in$ `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 solvers 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. 2018a] 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:

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

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

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

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

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 solvers. To see why, let us inspect the `var` rule in more detail. This rule declares a tuple $(C, e, T) \in$ `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 potential typing derivations:

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

As we will show in Section 3, a different encoding of type systems in Datalog 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 during typing.

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. 2018a] 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:

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

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 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 programmer 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 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 paper is to translate type systems to Datalog to utilize state-of-the-art incremental Datalog solvers. 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 3–5.

3 TRANSFORMATION 1: CO-FUNCTIONAL DEPENDENCIES

Incremental Datalog solvers 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 solvers (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*.

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 (`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 also much 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 7).

In the remainder of this paper, 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 and dependencies:

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

A relation signature $(R : T_1 \times \ldots \times T_n \mid F, G)$ 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 $(\mathtt{typed} : C \times e \times T \mid \{\{1,2\} \rightsquigarrow 3\}, \{\{2\} \overset{co}{\rightsquigarrow} 1\})$. 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 paper. 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.

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 $(\mathsf{R}(\overline{x}, \overline{c}, c) \text{ :- } a)$ is a rule of R, then $(\mathsf{R}'(\overline{x}, \overline{c}) \text{ :- } \pi_{\mathsf{R}}(\overline{c}, c), a)$ 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. Since $\mathtt{typed} : C \times e \times T$ has $e \overset{co}{\rightsquigarrow} C$, we derive the auxiliary relation $\pi_{\mathtt{typed}} : e \times C$ and use it in typed:

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

Note how we dropped column C from all rule heads and usages of typed. Instead, we introduced the query $\pi_{\mathtt{typed}}(e, C)$ at the beginning of each typed rule to bind C.

For the derived relation $\pi_{\mathtt{typed}} : e \times C$, 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 $\mathtt{typed}(C, e, T)$ we copy the surrounding rule and replace the head with $\pi_{\mathtt{typed}}(e, C)$:

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

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 $\pi_{\mathtt{typed}}$ 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 $\pi_{\mathtt{typed}}$ 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, $\pi_{\mathtt{typed}}$) to preserve ill-typed terms. This yields the following minimal rule set with a clear division of labor: $\pi_{\mathtt{typed}}$ propagates and extends the context, whereas typed does the checking and only mentions the context in the var rule.

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

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

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

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 $\mathit{CoFunTransSigs}(\Sigma)$ defined as follows:

$$\mathit{CoFunTrans}\text{-UpdateSig}\ \frac{\begin{array}{ll}(R : T_1 \times \ldots \times T_n \,|\, F, G) \in \Sigma & \mathit{codepCols} = \{i \mid (I \overset{co}{\rightsquigarrow} i) \in G\}\\ J = \{1, \ldots, n\} \setminus \mathit{codepCols} & F' = \mathit{deleteShift}(F, \mathit{codepCols})\end{array}}{(R : T_J \,|\, F', \emptyset) \in \mathit{CoFunTransSigs}(\Sigma)}$$

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

Rule *CoFunTrans*-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, function $\mathit{deleteShift}(F, \mathit{codepCols})$ deletes *codepCols* from the functional dependencies in F and shifts the remaining indices to skip dropped columns. For example, $(\mathtt{typed}: C \times e \times T \,|\, \{\{1,2\} \rightsquigarrow 3\}, \{\{2\} \overset{co}{\rightsquigarrow} 1\})$ becomes $(\mathtt{typed}: e \times T \,|\, \{\{1\} \rightsquigarrow 2\}, \emptyset)$ after dropping column C.

Rule *CoFunTrans*-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 $\mathit{CoFunTransRules}(\Sigma, \mathrm{P})$ defined as follows.

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

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

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

Rule *CoFunTrans*-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 *CoFunTrans*-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 *CoFunTrans*-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 *CoFunTrans*-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, 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: $p \in \mathsf{ok}$ if and only if $p \in \mathit{CoFunTrans}(\mathsf{ok})$.

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 solvers 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 $\mathtt{Map}[A, \mathbf{Unit}]$ and lists as $\mathtt{Map}[\mathtt{Int}, A]$, 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".

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.

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

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

Our goal is to eliminate the typing context produced by $\pi_{\mathtt{typed}}$ 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:

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

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 $\pi_{\mathtt{typed}}$ and `lookup`. Specifically, since relation $\pi_{\mathtt{typed}}$ 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:

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

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, too. 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.

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. Then the rewritten Datalog program has signatures $\mathit{CtxFusionTransSigs}(\Sigma)$ defined as follows:

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

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

Rule *CtxFusionTrans*-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 *CtxFusionTrans*-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 $\mathit{CtxFusionTransRules}(\Sigma, \mathrm{P})$ defined as follows. In computing $\mathit{CtxFusionTransRules}(\Sigma, \mathrm{P})$, 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.

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

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

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

$$\mathit{CtxFusionTrans}\text{-Delegate}\;\frac{\begin{array}{cc}\multicolumn{2}{c}{(\varphi_{\mathsf{R},\mathtt{i}}(x_I, k, v) \mathrel{:\!-} a_1, \ldots, a_m, \mathtt{lookup}(x_i, k, v).) \in \mathit{Step}_z(\Sigma, \mathrm{P})} \\ \mathsf{Q}(\overline{y}) \in \{a_1, \ldots, a_m\} & a_1, \ldots, a_m \vdash x_i = y_j \\ (Q : T_1 \times \ldots \times T_n \,|\, F, G) \in \Sigma & (J \rightsquigarrow j) \in F\end{array}}{(\varphi_{\mathsf{R},\mathtt{i}}(x_I, k, v) \mathrel{:\!-} a_1, \ldots, a_m, \varphi_{\mathsf{Q},\mathtt{j}}(y_J, k, v).) \in \mathit{CtxFusionTransRules}(\Sigma, \mathrm{P})}$$

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

$$\mathit{CtxFusionTrans}\text{-Retain}\;\frac{\begin{array}{c}(\mathsf{R}(\overline{x}) \mathrel{:\!-} a_1, \ldots, a_m.) \in \mathrm{P} \\ \mathit{CtxFusionTrans}\text{-Replace not applicable}\end{array}}{(\mathsf{R}(\overline{x}) \mathrel{:\!-} a_1, \ldots, a_m.) \in \mathit{CtxFusionTransRules}(\Sigma, \mathrm{P})}$$

Rule *CtxFusionTrans*-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 *CtxFusionTrans*-Unfold and *CtxFusionTrans*-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 *CtxFusionTrans*-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 $\mathit{Step}_{z+1}(\Sigma, \mathrm{P})$ to allow further transformation. Rule *CtxFusionTrans*-Bound captures the case where $k = k'$, so that we can yield $v = v'$. Since *CtxFusionTrans*-Bound fully eliminated the `lookup` call, we add the resulting rule to the output of the transformation.

Rule *CtxFusionTrans*-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 $\mathsf{Q}(\overline{y})$ 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 *CtxFusionTrans*-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 *CtxFusionTrans*-Replace and *CtxFusionTrans*-Retain propagate the original rules from P. Rule *CtxFusionTrans*-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 *CtxFusionTrans*-Retain copies over all other rules unchanged.

Note that it is possible for rules to starve rules within $Step_z$ that never make it to *CtxFusionTransRules*. 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 *CtxFusionTransRules* rule.

4.3 Example revisited

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

Input rules:

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

$Step_0(\Sigma, \mathrm{P})$:

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

$Step_1(\Sigma, \mathrm{P})$:

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

CtxFusionTransRules(Σ, P):

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

A subsequent optimization of the derived rules will remove all invocations of $\pi_{\mathtt{typed}}$ and `!bind`. This is supported by our implementation and will yield exactly those rules shown in Section 4.1.

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

Indeed it is sufficient to consider variables that are being referenced in an expression. Fortunately, we can derive an optimized version of φ_{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:

φ_{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**

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

$$
\begin{aligned}
\rho_{\mathsf{typed}}(e,x) &\coloneq ?\mathtt{var}(e,x).\\
\rho_{\mathsf{typed}}(e,k) &\coloneq ?\mathtt{app}(e,e_1,e_2),\ \ \rho_{\mathsf{typed}}(e_1,k).\\
\rho_{\mathsf{typed}}(e,k) &\coloneq ?\mathtt{app}(e,e_1,e_2),\ \ \rho_{\mathsf{typed}}(e_2,k).\\
\rho_{\mathsf{typed}}(e,k) &\coloneq ?\mathtt{lam}(e,x,T_1,b),\ \ k\neq x,\ \ \rho_{\mathsf{typed}}(b,k).
\end{aligned}
$$

For λx:**Unit**. (1+2)+(3+4), relation ρ_{typed} remains empty since no free variables occur. Consequently, φ_{typed} is empty as well. Our implementation supports this optimization.

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 3 and 4, but it does not require them and can be used independently.

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:

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

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**$_{\mathsf{typed}}$ = {?unit, ?app, ?lam, ?var} and **IgnoreStuck**$_{\mathsf{ok}}$ = {?main}.
- Some premises $\mathsf{R}(\overline{x})$ 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 $\varepsilon_{\mathsf{typed}} : C \times e \times \mathbf{Error}$ and $\varepsilon_{\mathsf{ok}} : p \times \mathbf{Error}$ that collect the errors that can occur during type checking:

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

There is no rule for unit because its first premise is in **IgnoreStuck**$_{\mathsf{typed}}$ 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 $\varepsilon_{\mathsf{typed}}(C, e_1, err)$. Second, if the type T_e of e_1 is not a function type (note the negation $\neg$ 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 $\neg$).

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 $(C, e, T) \in \mathsf{typed}$, we additionally require $(C, e, err) \notin \varepsilon_{\mathsf{typed}}$ 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:

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

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 **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 $\mathit{CollectErrorsTrans}(\Sigma)$ defined as follows:

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

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

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 $(t_1, \ldots, t_n, \mathit{err}) \in \varepsilon_R$ 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 $\mathit{CollectErrorsTransRules}(\Sigma, \mathrm{P})$ defined as follows.

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

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

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

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

Rule *CollectErrorsTrans*-Propagate generates a Datalog rule that propagates errors from sub-derivations 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 the following error rules amongst others:

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

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 *CollectErrorsTrans*-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 $\neg \texttt{Q}(\overline{y})$, 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 *CollectErrorsTrans*-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 *CollectErrorsTrans*-RetainNormal retains all other Datalog rules unchanged.

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 $p \in \texttt{ok}$ and $(e, err) \notin \varepsilon_{\texttt{ok}}$ 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 *CollectErrorsTrans*-Propagate, such that error relations ε_R are only filled according to *CollectErrorsTrans*-NewError. Moreover, the resulting error relations are perfectly suited for programming editors and compilers, which can extract type errors their origin.

6 IMPLEMENTATION: A TYPE-SYSTEM DSL COMPILED TO DATALOG

We have implemented a domain-specific language (DSL) for describing textbook-like type systems. In our DSL, the programmer can declare arbitrary judgments with mixfix syntax. 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.[1] That is, our DSL can be used to define the type system of other languages defined with MPS.

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

rule typeOf lam
  Bind(name: name, type: T1, rest: C) |- t : T2
  ------------------------------------------------
  C |- Lam(name: name, ty: T1, t: t) : Fun(ty1: T1, ty2: T2)

rule typeOf app
  C |- t1 : Fun(ty1: T1, ty2: T2)
  C |- t2 : T12
  T1 == T12
  ------------------------------------------------
  C |- App(t1: t1, t2: t2) : T2
```

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

We developed a compiler for our DSL that generates Datalog code using the transformations described in this paper. Specifically, we generate code conforming to the Datalog dialect of IncA [Szabó et al. 2018a, 2016], an incremental Datalog-based static analysis framework. There are few differences between the transformations in our implementation and the transformations described in the paper:

- 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 $C \vdash e : T$.
- Since we can reason about metavariable bindings in the implementation, slicing becomes easier. Where we used $\textbf{slice}_X(A)$ in the paper, 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 paper, 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 link to be added after double-blind reviewing.

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 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.

```
rule infer App
  C |- t1 => Fun(ty1: ty1, ty2: ty2)
  C |- t2 <= ty1
  ------------------------------------
  C |- App(t1: t1, t2: t2) => ty2

rule check Lam
  ty match Fun(ty1: ty1, ty2: ty2)
  Bind(name: name, type: ty1, rest: C) |- t <= ty2
  ------------------------------------
  C |- Lam(name: name, t: t) <= ty
```

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 solver can compute mutually recursive Datalog relations. We can thus incrementalize bi-directional type systems.

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 `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.

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, 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.

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:

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

Program	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

Fig. 1. 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.

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. 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*.

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 1 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 our synthesized programs.

9 RELATED WORK

IncA [Szabó et al. 2016] is an incremental static analysis framework based on Datalog. We use IncA in our work as the incremental evaluation engine for the Datalog code we generate. Specifically, we show how to systematically construct type checkers that can then be automatically incrementalized by IncA. IncA has been shown previously to deliver fast incremental updates for a range of program analyses: FindBugs-style linting, control-flow analysis [Szabó et al. 2016], data-flow analyses [Szabó et al. 2018a], and overload resolution [Szabó et al. 2018b]. This paper is the first to generate IncA code from high-level specifications.

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. 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 paper 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. [2015] 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.

Datafun [Arntzenius and Krishnaswami 2020] is a higher-order functional language that incorporates Datalog's semi-naive bottom-up evaluation. While Datafun does not aim for incrementality, it would be interesting to see if our transformations can expand the expressivity of Datafun. Specifically, it would be interesting to demonstrate that the bottom-up computable Datalog code we generate indeed is admitted by Datafun's type system. Their type system enforces monotonicity constraints that our Datalog solver relies on, too.

The transformation we presented in Section 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 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.

10 CONCLUSION

We proposed a novel approach to systematically deriving incremental type checkers based on textbook-style type rules. 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. We showcased that our transformations can handle different type system features such as sum and product types, overloading, universal 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.

REFERENCES

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

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)*, Maurice Bruynooghe and Martin Wirsing (Eds.), Vol. 631. Springer, 87–99. https://doi.org/10.1007/3-540-55844-6_129

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

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)*, Julia M. Badger and Kristin Yvonne Rozier (Eds.), Vol. 11460. Springer, 106–122. https://doi.org/10.1007/978-3-030-20652-9_7

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)*, Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin (Eds.), Vol. 173. Springer, 215–233. https://doi.org/10.1007/3-540-13346-1_11

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. 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

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)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:26. https://doi.org/10.4230/LIPIcs.ECOOP.2017.18

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

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

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

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)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.), Vol. 6702. Springer, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

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

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, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. 2018b. Incremental overload resolution in object-oriented programming languages. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, Julian Dolby, William G. J. Halfond, and Ashish Mishra (Eds.). ACM, 27–33. https://doi.org/10.1145/3236454.3236485

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

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)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. 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

Adrienne Watt. 2018. *Database design.*