Fixpoints for the Masses: Programming with First-Class Datalog Constraints

David R. Cheriton School of Computer Science Technical Report CS-2020-05

MAGNUS MADSEN, Aarhus University, Denmark
ONDŘEJ LHOTÁK, University of Waterloo, Canada

Datalog is a declarative logic programming language that has been used in a variety of applications, including big-data analytics, language processing, networking and distributed systems, and program analysis.

In this paper, we propose first-class Datalog constraints as a mechanism to construct, compose, and solve Datalog programs at run time. The benefits are twofold: We gain the full power of a functional programming language to operate on Datalog constraints-as-values, while simultaneously we can use Datalog where it really shines: to declaratively express and solve fixpoint problems.

We present an extension of the lambda calculus with first-class Datalog constraints, including its semantics and a type system with row polymorphism based on Hindley-Milner. We prove soundness of the type system and implement it as an extension of the Flix programming language.

CCS Concepts: • **Software and its engineering → Functional languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: functional programming, logic programming, first-class datalog

1 INTRODUCTION

Datalog is a simple, yet surprisingly powerful, declarative logic programming language. A Datalog program is a collection of constraints. Each constraint is a fact or a rule. Together, the facts and rules imply a minimal model, a unique solution to any Datalog program [Ceri et al. 1989].

Datalog has roots in the database community [Ullman 1984] and has been used in a wide variety of applications [de Moor et al. 2011; Huang et al. 2011], including in bioinformatics [King 2004], big-data analytics [Halperin et al. 2014; Seo et al. 2013; Shkapsky et al. 2016], natural language processing [Mooney 1996], networking and distributed systems [Alvaro et al. 2010; Conway et al. 2012; Loo et al. 2009], program understanding [Hajiyev et al. 2006], and program analysis [Bravenboer and Smaragdakis 2009; Lam et al. 2005; Smaragdakis and Bravenboer 2011].

Datalog has several properties of theoretical and practical interest: (i) every Datalog program eventually terminates, (ii) every Datalog program has a unique solution, (iii) efficient and parallel evaluation strategies exist, and (iv) any polynomial time algorithm can be expressed in Datalog [Papadimitriou 1985]. For practical purposes, (i)–(iii) are very useful as they enable easy reasoning and debugging of Datalog programs.

Datalog programs are truly declarative: Unlike Prolog, the order of constraints and the order of predicates within a constraint is immaterial in Datalog; we may freely reorder the program without changing its semantics. The solution to a Datalog program, its minimal model, is defined without reference to any specific evaluation order or strategy. Thus, Datalog cleanly separates the *what* from the *how*, i.e. the specification is separate from its implementation.

Modern Datalog solvers are increasingly efficient [Jordan et al. 2016, 2018; Scholz et al. 2016; Subotic et al. 2018; Veldhuizen 2012], implementing many important optimizations, such as index selection, query planning, join-ordering, and parallel execution [Bancilhon et al. 1985; Graefe 1993;

Authors' addresses: Magnus Madsen, Department of Computer Science, Aarhus University, Åbogade 34, Aarhus, 8210, Denmark, magnusm@cs.au.dk; Ondřej Lhoták, Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, N2L 3G1, Canada, olhotak@uwaterloo.ca.

Gregory 1987; Ullman 1984]. In hand-crafted fixpoint computations, such optimizations must be re-discovered and re-implemented, a waste of programmer effort.

Despite these advantages, use of Datalog is not widespread. We speculate that there are at least two barriers to a broader adoption of Datalog: (i) the poor integration of Datalog into general-purpose programming languages, and (ii) the lack of mechanisms to construct modular Datalog programs. To expand on the former, use of Datalog today often involves generating constraints, storing them into a file, invoking the Datalog engine on the file, and reading the results back into memory for further processing. To expand on the latter, we are rarely interested in a *specific* Datalog program, but rather in a *family* of related Datalog programs. Since Datalog lacks a module system, programmers often resort to program generation. Such approaches, whether based on textual generation or template programming, are often inflexible and error-prone.

To overcome these issues, we propose Datalog programs as first-class values that can be constructed, composed, and solved within a functional programming language. We can leverage the features of functional programming to build modular and parametric families of Datalog programs. We can build pipelines of Datalog programs where the output facts of one program are passed as input facts to another program. We can define functions and refer to them from inside Datalog constraints. And finally, we can use parametric polymorphism to express Datalog constraints that are polymorphic in the types of their terms. In this paper, we show how to integrate first-class constraints into a statically-typed eager functional language. We describe the static and dynamic semantics, interaction with lexical scoping, and how to support stratified negation.

We identify and overcome two technical challenges in the presence of first-class constraints: (i) how can we design a modular type system in which reusable fragments of Datalog programs can be typed independently such that when they are composed, at run-time, the composition is type-safe? Specifically, the type system should support abstraction such that each module can be provided by multiple different implementations, and (ii) in the presence of negation, how can we ensure, at compile-time, that every such composition that could occur at run-time is stratified?

In summary, the contributions of this paper are:

- **(Calculus)** We show how to extend the lambda calculus with first-class Datalog constraints.
- **(Type System)** We show how a Hindley-Milner style type system for a functional programming language can be seamlessly extended to type Datalog program fragments as first-class values. Inspired by record calculi, the type system uses row polymorphism to precisely track the predicate symbols that occur in each Datalog fragment. The type system supports full type inference and we prove type safety of the system.
- **(Stratification)** We observe that there are trade-offs between modularity and type complexity in solutions for ensuring stratified negation. As a starting point, we propose one sound technique to compute stratification at compile-time.
- **(Implementation)** We implement our system in the Flix programming language.
- **(Evaluation)** We present case studies that illustrate the usefulness of our system.

2 MOTIVATION

We motivate the need for first-class Datalog constraints with several toy examples. In Section 6, we present real-world applications implemented in our system. All programs shown in this paper are valid and executable programs in our system.

2.1 Example I

As a student of history might know, Pompey the Great was the son of the wealthy equestrian Gnaeus Pompeius Strabo. Pompey had five wives with whom he had two sons and one daughter. We can capture these familial relations with the Datalog program:

```
ParentOf("Pompey", "Strabo").
ParentOf("Gnaeus", "Pompey").
ParentOf("Pompeia", "Pompey").
ParentOf("Sextus", "Pompey").
```

In ancient Rome, the lineage of a statesman was an important part of his stature. We can use Datalog to elegantly compute the ancestors of every person, i.e. the transitive closure of the ancestor relation:

```
AncestorOf(x, y) :- ParentOf(x, y).
AncestorOf(x, z) :- AncestorOf(x, y), AncestorOf(y, z).
```

The solution to this program, its minimal model, contains the fact AncestorOf(`"Sextus"`, `"Strabo"`), because Strabo was the grandfather of Sextus. We are now able to reason about family relations in ancient Rome. However, familial ties where not only by blood, as adoption was equally recognized. With this in mind, we can extend the Datalog program with the facts:

```
AdoptedBy("Augustus", "Caesar").
AdoptedBy("Tiberius", "Augustus").
```

to record that Caesar adopted Augustus who himself later adopted Tiberius. We can easily include adoptions in the ancestor relation with the addition of a single rule:

```
AncestorOf(x, y) :- AdoptedBy(x, y).
```

This example demonstrates the power of Datalog: We can extend the semantics of a program by simply adding new facts and rules! But what if we had wanted to keep the original program? We would have to maintain *two* versions of our program: one with ancestry based on biological parents and one extended to include adoptions. We could store these two programs as copies in separate files, but what happens when we discover a bug in one program? Or when we want to extend the other? We quickly run into the *multiple maintenance problem*. What we want is the ability to *construct*, *compose*, and *solve* Datalog programs at run-time. What we need are *first-class constraints*. Here is a reformulation of both programs as a single program in our extension of Flix:

```
def getParents[r](): #{ParentOf(String, String) | r} = #{
  ParentOf("Pompey", "Strabo").
  ParentOf("Gnaeus", "Pompey").
  ParentOf("Pompeia", "Pompey").
  ParentOf("Sextus", "Pompey").
}

def getAdoptions[r]() #{AdoptedBy(String, String) | r} = #{
  AdoptedBy("Augustus", "Caesar").
  AdoptedBy("Tiberius", "Augustus").
}
```

```
def heritage[r](withAdoptions: Bool): #{ParentOf(String, String),
  AncestorOf(String, String), AdoptedBy(String, String) | r} =
  let p1 = #{
    AncestorOf(x, y) :- ParentOf(x, y).
    AncestorOf(x, z) :- AncestorOf(x, y), AncestorOf(y, z).
  };
  let p2 = #{
    AncestorOf(x, y) :- AdoptedBy(x, y).
  };
  if (withAdoptions) p1 else (p1 <+> p2)

def ancestors(withAdoptions: Bool): #{AncestorOf(String, String) | r} =
  let facts = getParents() <+> getAdoptions();
  let model = solve (facts <+> heritage(withAdoptions));
  project AncestorOf model
```

This program illustrates the key idea of our work: Datalog constraints are first-class values. We can pass them around, compose them with other Datalog values, and solve them.

The functions `getParents` and `getAdoptions` return sets of facts about biological parents and adoptions. The `heritage` function encapsulates the two variants of the Datalog program. The function constructs the two Datalog program values p_1 and p_2, and returns either p_1, corresponding to the program solely with biological parents, or the composition (i.e. union) of p_1 and p_2, corresponding to the program with biological and adoptive parents. The `ancestor` function assembles the facts and rules, computes their minimal model (with `solve`), and returns the AncestorOf facts. The `getParents`, `getAdoptions`, and `ancestor` functions have a row type parameter r. If the parameter were removed, each function would return a *closed* row which could not be combined with Datalog constraints sets containing other predicate symbols.

The program illustrates the following important principles of our design:

Principle I: Datalog constraints are first-class values that can be passed around.

Principle II: Datalog constraints can be composed with other Datalog constraints to form larger Datalog programs.

Principle III: The design preserves the essence of Datalog: the constraints are declarative, they look like ordinary Datalog clauses, and they are solvable by standard techniques.

2.2 Example II

We now consider a simple reachability problem: Given a road network with speed limits on each road, we want to determine if it is possible to drive from one city to another city going at least a certain speed. We can write a function r that uses Datalog to compute if this is possible:

```
def r(g: #{Road(City, Int, City)}, src: City, dst: City, speed: Int): Bool =
  let p = #{
    Path(x, y) :- Road(x, speedLimit, y), if speedLimit > speed.
    Path(x, z) :- Path(x, y), Road(y, speedLimit, z), if speedLimit > speed.
  };
  (solve g <+> p) |= #{ Path(src, dst). }
```

The function r takes a road network in the form of a set of road facts g, a source city src, a destination city dst, and a minimum speed we want to go on each road. Each rule is equipped with a boolean-valued *filter* expression that must be true in the minimal model. The expression e_1 |= e_2 returns true if e_2 is a subset of e_1. In other words, if there is a path with the desired speed.

If we want a leisurely drive, we may wish to avoid roads which have a high speed limit. Instead of changing the *r* function, we can generalize it by equipping it with a predicate *q* that determines whether the speed limit on a road is acceptable:

```
def r(g: #{Road(City, Int, City)}, src: City, dst: City, q: Int -> Bool): Bool =
  let p = #{
    Path(x, y) :- Road(x, speedLimit, y), if q(speedLimit).
    Path(x, z) :- Path(x, y), Road(y, speedLimit, z), if q(speedLimit).
  };
  (solve g <+> p) |= #{ Path(src, dst). }
```

We can imagine that roads carry not only speed limits, but also other meta-data such as the presence of construction work, the current weather conditions, etc. We can further generalize *r* to be *polymorphic* in the type of the information associated with a road:

```
def r[a](g: #{Road(City, a, City)}, src: City, dst: City, q: a -> Bool): Bool =
  let p = #{
    Path(x, y) :- Road(x, metaData, y), if q(metaData).
    Path(x, z) :- Path(x, y), Road(y, metaData, z), if q(metaData).
  };
  (solve g <+> p) |= #{ Path(src, dst). }
```

These programs illustrate the following important principles of our design:

Principle IV: Datalog constraints may refer to expressions in the functional language.

Principle V: The types of the predicates of a Datalog constraint may be polymorphic.

2.3 Benefits of Type Safety

We can imagine a programmer accidentally writing a program like:

```
let g = #{
  Edge(Paris, 120, Lyon).
  Edge(Lyon, 110, Rome).
};
let q = #{
  Path(x, y) :- Edge(x, y).
  Path(x, z) :- Path(x, y), Edge(y, z).
};
solve (g <+> q)
```

where the arity of the predicate symbol Edge is inconsistent. In the facts of g, the Edge relation is ternary, whereas in the rules of q, the Edge relation is binary. The composition of g and q is meaningless: In a Datalog program every predicate symbol must have a fixed arity. In the presence of first-class Datalog constraints, evaluation of such a program would get stuck at run-time. In this paper, we propose a static type system, based on *row polymorphism*, to reject such programs.

As a continuation of Example II, we can also imagine a programmer accidentally writing:

```
let g = #{ Road(Paris, true, Lyon). ... };
solve r(g, Paris, Rome, speed -> speed > 60)
```

This program would get stuck inside the Datalog engine, since when the function `speed -> speed > 60` is applied to the value `true`, we get the stuck term: `true > 60`. The proposed type system also rejects such programs.

Principle VI: Type safety: Well-typed programs do not get stuck.

$$
\begin{aligned}
P \in \mathit{Programs} &= C_1, \cdots, C_n \\
C \in \mathit{Constraints} &= A_0 \Leftarrow B_1, \cdots, B_n \\
A \in \mathit{Head\ Atoms} &= p(t_1, \cdots, t_n) \\
B \in \mathit{Body\ Atoms} &= p(t_1, \cdots, t_n) \mid \mathsf{not}\, p(t_1, \cdots, t_n) \\
t \in \mathit{Terms} &= x \mid c \\
\\
c \in \mathit{Literals} &= \text{is a set of literal constants.} \\
x, y \in \mathit{VarSym} &= \text{is a set of variable symbols.} \\
p, q \in \mathit{PredSym} &= \text{is a set of predicate symbols.}
\end{aligned}
$$

Fig. 1. Datalog Syntax.

These examples demonstrate the benefit of a type system. A library based approach would suffer from such run-time errors. In addition, from a practical point of view, a language-based approach has several advantages: (i) we can determine whether a Datalog program is stratified (and compute its stratification) at compile-time (see Section 4.7), (ii) we can have proper syntax and integration with lexical scoping (as shown in the examples), and (iii) we can provide understandable type and stratification error messages.

3 THE λ_{DAT} CALCULUS

We now present λ_{DAT}, a minimal lambda calculus with first-class Datalog constraints, which is the formal foundation of our implementation. We begin with a brief recap of Datalog, then discuss the syntax, semantics, and type system of λ_{DAT}.

3.1 A Brief Recap of Datalog

We briefly discuss the syntax and semantics of Datalog. Readers who are already familiar with Datalog may wish to skip this subsection. A comprehensive introduction to Datalog is available in Ceri et al. [1989, 2012]. In this paper, we focus on *stratified* Datalog, a variant of Datalog with a restricted form of negation [Minker 1988].

Syntax. A Datalog *program P* is a collection of constraints $C_1, \cdots, C_n$. A *constraint*, also called a Horn clause, is of the form $A_0 \Leftarrow B_1, \cdots, B_n$ where A_0 is the *head atom* and each B_i is a *body atom*. A head atom $p(t_1, \cdots, t_n)$ consists of a predicate symbol p and a sequence of terms $t_1, \cdots, t_n$. A *body atom* is similar to a head atom, except it can be negated, which is written with the not keyword in front of the predicate symbol. A constraint without a body is called a *fact*. Conversely, a constraint with a body is called a *rule*. A *term* is either a variable x or a literal constant c. An atom without variables is said to be *ground*. A fact or rule with only ground atoms is said to be ground. Figure 1 shows the grammar of Datalog. As an example, the constraint:

$$\mathsf{Path}(x, z) \Leftarrow \mathsf{Path}(x, y), \mathsf{Edge}(y, z).$$

is a *rule* with predicate symbols Path and Edge and variables x, y and z. Datalog rules are implicitly *universally quantified* by their variables, hence the rule is formally written as:

$$\forall x, \forall y, \forall z.\ \mathsf{Path}(x, z) \Leftarrow \mathsf{Path}(x, y), \mathsf{Edge}(y, z).$$

In the calculus, we will write constraints with universal quantifiers. In the implementation, as shown in the previous section, the compiler will automatically determine which variables are implicitly quantified and which are bound by the lexical scope. Thus, like in traditional Datalog programs, the programmer never has to explicitly introduce variables.

$$\begin{array}{rcl}
v \in \mathit{Val} & = & c \mid \lambda x.\, e \mid \#\{C_1, \cdots, C_n\} \\
\\
e \in \mathit{Exp} & = & x \mid v \mid e\, e \mid \mathsf{let}\, x = e\, \mathsf{in}\, e \\
& \mid & e \texttt{ <+> } e \mid \mathsf{solve}\, e \mid \mathsf{subset}\, e\, e \mid \mathsf{project}\, p\, e \\
& \mid & \mathcal{S}(e) \mid \mathcal{H}(A, e, e) \\
\\
C \in \mathit{Constraints} & = & \forall(x_1 : \tau_1 \cdots x_n : \tau_n).\, A_0 \overset{e}{\Leftarrow} B_1, \cdots, B_n \\
A \in \mathit{Head\ Atoms} & = & p(t^h_1, \cdots, t^h_n) \\
B \in \mathit{Body\ Atoms} & = & p(t^b_1, \cdots, t^b_n) \mid \mathsf{not}\, p(t^b_1, \cdots, t^b_n) \\
t^h \in \mathit{Head\ Terms} & = & x \mid c \mid e \\
t^b \in \mathit{Body\ Terms} & = & x \mid c \\
\\
c \in \mathit{Literals} & = & \text{is a set of literal constants.} \\
x, y \in \mathit{VarSym} & = & \text{is a set of variable symbols.} \\
p, q \in \mathit{PredSym} & = & \text{is a set of predicate symbols.}
\end{array}$$

Fig. 2. Syntax of λ_{DAT}.

Semantics. The model-theoretic semantics of Datalog describes the solution, i.e. minimal model, of any Datalog program independent of the mechanism used to compute it [Ceri et al. 1989, 2012; Fitting 2002; Gelfond and Lifschitz 1988, 1991; Kunen 1987]. Intuitively, a Datalog model is simply a set of facts. The minimal model is the smallest set of facts that satisfies all constraints. The model-theoretic semantics is what makes Datalog declarative: we can write a Datalog program, and someone else can write a Datalog solver, and we can independently agree on what the result ought to be. In this paper, we shall not be particularly concerned with *how* the minimal model is computed. Rather, we will model a generic Datalog solver with very mild assumptions about its behavior. Consequently, the calculus will work with any Datalog solver independent of its specific implementation details. We will model the solver as a black box that, when given a set of Datalog constraints, is permitted to repeatedly select any constraint, pick a type-safe valuation of its quantified variables, instantiate the constraint with the valuation, evaluate its terms (expressions), and add the head atom to the current constraint set.

3.2 Syntax of λ_{DAT}

We now turn to the syntax of λ_{DAT} which extends the lambda calculus with first-class constraints. The grammar of λ_{DAT} is shown in Figure 2. The language includes the usual expressions from the lambda calculus: constants, variables, lambda abstraction, function application, and let-bindings.

Values. The values of λ_{DAT} include literal constants c, lambda abstractions $\lambda x.\, e$, and *constraint sets* $\#\{C_1, \cdots, C_n\}$. A constraint set is a set of enriched Datalog constraints. In our extension of Flix, a single fact or rule can be written without #{}. We will use this in subsequent examples when there is no risk of confusion. The grammar of constraints mirrors that of Figure 1, but with three important differences: (i) the implicit universally quantified variables are made explicit along with their types, (ii) every constraint is extended with a *filter* expression on the implication arrow, and (iii) the terms of a head atom may now be expressions. The last two extensions enrich the expressive power of the constraints.

A constraint set $\#\{C_1, \cdots, C_n\}$ is a set of the constraints $C_1, \cdots, C_n$. We define two Datalog constraints C_1 and C_2 to be equal when they are syntactically identical.

$$\frac{}{(\lambda x.\, e)\, v \longrightarrow e[x \mapsto v]} \quad \text{(E-App)}$$

$$\frac{}{\text{let}\, x = v \,\text{in}\, e \longrightarrow e[x \mapsto v]} \quad \text{(E-Let)}$$

$$\frac{v_1 = \#\{C^1_1, \cdots, C^1_n\} \quad v_2 = \#\{C^2_1, \cdots, C^2_m\}}{v_1 \texttt{ <+> } v_2 \longrightarrow \#\{C^1_1, \cdots, C^1_n, C^2_1, \cdots, C^2_m\}} \quad \text{(E-Compose)}$$

$$\frac{}{\text{solve}\, \#\{C_1, \cdots, C_n\} \longrightarrow \mathcal{S}(\#\{C_1, \cdots, C_n\})} \quad \text{(E-Solve)}$$

$$\frac{\begin{array}{c} v_1 = \#\{C^1_1, \cdots, C^1_n\} \quad v_2 = \#\{C^2_1, \cdots, C^2_m\} \\ v_1 \subseteq v_2 \end{array}}{\text{subset}\, v_1\, v_2 \longrightarrow \texttt{true}} \quad \text{(E-Subset-T)}$$

$$\frac{\begin{array}{c} v_1 = \#\{C^1_1, \cdots, C^1_n\} \quad v_2 = \#\{C^2_1, \cdots, C^2_m\} \\ v_1 \not\subseteq v_2 \end{array}}{\text{subset}\, v_1\, v_2 \longrightarrow \texttt{false}} \quad \text{(E-Subset-F)}$$

$$\frac{v = \{p(v_1, \cdots, v_m) \mid p(v_1, \cdots, v_m) \in \#\{C_1, \cdots, C_n\}\}}{\text{project}\, p\, \#\{C_1, \cdots, C_n\} \longrightarrow v} \quad \text{(E-Project)}$$

$$\frac{\begin{array}{c} C_i = \forall(x_1 : \tau_1, \cdots, x_n : \tau_n).\, A \overset{e}{\Leftarrow} B_1, \cdots, B_{n'} \\ \nu \text{ is a primitive valuation of } x_i \text{ s.t. typeOf}(\nu(x_i)) = \tau_i \end{array}}{\mathcal{S}(\#\{C_1, \cdots, C_{n''}\}) \longrightarrow \mathcal{H}(\nu(A), \nu(e), \#\{C_1, \cdots, C_{n''}\})} \quad \text{(S-Rule)}$$

$$\frac{v = \{p(v_1, \cdots, v_n) \mid p(v_1, \cdots, v_n) \in \#\{C_1, \cdots, C_{n'}\}\}}{\mathcal{S}(\#\{C_1, \cdots, C_{n'}\}) \longrightarrow v} \quad \text{(S-Finish)}$$

$$\frac{A = p(v_1, \cdots, v_n)}{\mathcal{H}(A, \texttt{true}, \#\{C_1, \cdots, C_n\}) \longrightarrow \mathcal{S}(\#\{A, C_1, \cdots, C_n\})} \quad \text{(H-True)}$$

$$\frac{A = p(v_1, \cdots, v_n)}{\mathcal{H}(A, \texttt{false}, \#\{C_1, \cdots, C_n\}) \longrightarrow \mathcal{S}(\#\{C_1, \cdots, C_n\})} \quad \text{(H-False)}$$

Fig. 3. Semantics of λ_{DAT}.

Expressions. The expressions of λ_{DAT} include variables x, values v, function application $e\, e$, and let-bindings let $x = e$ in e. The calculus has four expressions for working with first-class constraints: (i) a *composition expression* e_1 <+> e_2 to compute the union of two constraint sets, (ii) a *subset expression* subset $e_1\, e_2$ to determine if one constraint set is a subset of another constraint set, (iii) a *project expression* project $p\, e$ to extract all facts of a given predicate symbol from a constraint set, and (iv) a *solve expression* solve e to compute the minimal model of a constraint set. Finally, the calculus has two internal constructs: $\mathcal{S}(e)$ and $\mathcal{H}(A, e, e)$ to model the execution of the Datalog solver. The two expressions are not considered part of the surface syntax of the language. Figure 2 shows the grammar of the expressions in λ_{DAT}.

Composing Constraints. We can compute the union of two constraints sets with the *composition expression e* <+> *e*. Unlike in Prolog, the order of constraints in Datalog is immaterial. Hence, composition is both commutative and associative[1]:

$$e_1 \texttt{ <+> } e_2 = e_2 \texttt{ <+> } e_1 \qquad \text{(||-Commutative)}$$

$$(e_1 \texttt{ <+> } e_2) \texttt{ <+> } e_3 = e_1 \texttt{ <+> } (e_2 \texttt{ <+> } e_3) \qquad \text{(||-Associative)}$$

Furthermore, repeating a Datalog constraint has no effect, so composition is idempotent:

$$e \texttt{ <+> } e = e \qquad \text{(||-Idempotent)}$$

A *key insight* is that these properties enable a design of first-class constraints which supports modular and local reasoning. A similar design for Prolog falls apart because the order of clauses is significant and simply changing the order of two clauses may cause non-termination.

[1] Under the assumption that e_1, e_2, and e_3 are pure (i.e. have no side-effects) and total (i.e. always terminate).

Computing the Minimal Model. We can compute the minimal model of a constraint set with the solve e expression. The expression evaluates to a new constraint set that consists of all the facts in the minimal model of e. The constraint set does not contain any rules.

The solve expression is idempotent since the minimal model of a set of facts is the set itself:

$$\text{solve}\,(\text{solve}\,e) = \text{solve}\,e \qquad \text{(Solve-Idempotent)}$$

Why is Solve Explicit? The solve operation *must* be explicit because in the presence of composition of constraint sets, we cannot identify a constraint set with its minimal model. If we compute the minimal model eagerly, immediately when two constraints are composed, then the *order of composition determines the minimal model.* For example, if we have the program:

```
let a = A(1).;
let b = B(1).;
let q = R(x) :- A(x), not B(x).
```

Then the eager computation of `(a <+> b) <+> q` is *not* equivalent to the eager computation of `(a <+> q) <+> b`. The minimal model of the former does not contain an R fact, but the latter does. We view such order dependence as completely antithetical to the declarative nature of Datalog. Consequently, our design requires explicit solve expressions.

Comparing Constraint Sets. We can compute whether one constraint set is a subset of another with the subset expression subset $e_1\ e_2$. The expression evaluates to `true` if all constraints in e_1 appear in e_2. The expression does *not* compute the minimal model of e_1 nor e_2. By lifting a fact into the singleton set, the subset expression can be used to determine if the minimal model of a program contains that fact.

Projecting Constraint Sets. We can use the project expression project $p\ e$ to select the facts in e which share the predicate symbol p. The project expression is useful when we have computed the minimal model of a program, and we want to select a certain type of facts for further computation.

Enriched Datalog Constraints. We enrich Datalog constraints in two ways: (i) we introduce a *filter expression* as a guard on a constraint, and (ii) we allow arbitrary expressions as head terms.

For (i), a constraint with a filter expression e is written as: $\forall(x_1 : \tau_1, \cdots, x_n : \tau_n).\ A_0 \overset{e}{\Leftarrow} B_1, \cdots, B_n$. The expression e may refer to any of the quantified variables $x_1, \cdots x_n$ of the constraint. Filter expressions enrich the semantics of a Datalog constraint by allowing an arbitrary expression to determine when a ground instance of the constraint holds. But the cost is decidability: the programmer must take adequate measures to ensure that the expression e always terminates.

For (ii), a constraint may now have arbitrary expressions as terms in its head atom. This increases the expressiveness at the cost of decidability. For example, we can now perform addition from within constraints:

```
Var(r, v1 + v2) :- Add(r, x, y), Var(x, v1), Var(y, v2).
```

But the consequence is that the Herbrand Base is no longer finite, and as such we have lost most of the important properties of Datalog. For example, we can now write a program with a constraint that never terminates:

```
P(0). P(x + 1) :- P(x).
```

With great power comes great responsibility: If the programmer chooses to use the enriched Datalog constraints, then he or she must ensure termination and that the Herbrand base remains finite.

$$
\begin{array}{rcl}
E \in \mathit{Ctx} & = & \Box \\
& | & E\, e \mid v\, E \mid E \texttt{ <+> } e \mid v \texttt{ <+> } E \mid \mathsf{let}\, x = E\, \mathsf{in}\, e \\
& | & \mathsf{solve}\, E \mid \mathsf{subset}\, E\, e \mid \mathsf{subset}\, v\, E \mid \mathsf{project}\, p\, E \\
& | & \mathcal{S}(E) \mid \mathcal{H}(A, e, E) \mid \mathcal{H}(A, E, v) \\
& | & \mathcal{H}(p(v, \cdots, E_i, \cdots, e), v, v)
\end{array}
$$

Fig. 4. Evaluation Contexts of λ_{DAT}.

Internal Solver Expressions. We model the semantics of the Datalog solver with two *internal solver expressions.* The two internal expressions are not part of the surface syntax of the language, but they are expressions that appear during evaluation.

The *select* expression $\mathcal{S}(\#\{C_1, \cdots, C_n\})$ represents an internal state where the Datalog solver is in the process of computing the fixed point of the constraint set $\#\{C_1, \cdots, C_n\}$ and needs to select and instantiate a constraint. The *extend* expression: $\mathcal{H}(A, e, \#\{C_1, \cdots, C_n\})$ represents an internal state where the Datalog solver is in the process of evaluating the head terms of an atom and its filter expression. If the filter expression evaluates to `true`, then the evaluated head atom is added to the constraint set, and the solver continues. If the filter expression evaluates to `false`, then the solver continues with the original constraint set. The purpose of the two internal solver expressions is to mimic the most fundamental operations of a Datalog solver: rule selection, rule instantiation, and evaluation of head terms. The idea is that these two expression can be substituted for an actual implementation while preserving the type safety of the calculus.

3.3 Evaluation of λ_{DAT}

We define evaluation of λ_{DAT} as a small-step operational semantics. The evaluation relation $e \longrightarrow e'$ describes when the expression e can reduce to the expression e' in a single step. We define an intrinsic (or Church-style) semantics where the evaluation rules depend on the type of the expressions: in other words, we assign no meaning to untyped programs. This is required, since for the Datalog solver to correctly instantiate a Datalog constraint, it must know the types of its quantified variables. The evaluation rules of λ_{DAT} are shown in Figure 3. We now discuss the most important rules:

The evaluation rule for constraint composition:

$$
\frac{v_1 = \#\{C^1_1, \cdots, C^1_n\} \quad v_2 = \#\{C^2_1, \cdots, C^2_m\}}{v_1 \texttt{ <+> } v_2 \longrightarrow \#\{C^1_1, \cdots, C^1_n, C^2_1, \cdots, C^2_m\}} \qquad (\text{E-Compose})
$$

states that to combine two constraint sets, we compute their union. The evaluation rule for solve:

$$
\mathsf{solve}\, \#\{C_1, \cdots, C_n\} \longrightarrow \mathcal{S}(\#\{C_1, \cdots, C_n\}) \qquad (\text{E-Solve})
$$

states that a solve expression reduces to the internal select expression.

The first evaluation rule for the subset expression:

$$
\frac{v_1 = \#\{C^1_1, \cdots, C^1_n\} \quad v_2 = \#\{C^2_1, \cdots, C^2_m\} \quad v_1 \subseteq v_2}{\mathsf{subset}\, v_1\, v_2 \longrightarrow \texttt{true}} \qquad (\text{E-Subset-T})
$$

states that a constraint set v_1 is a subset of v_2 if all constraints in v_1 also occur in v_2. The (E-Subset-F) rule is similar.

The evaluation rule for projection:

$$
\frac{v = \{p(v_1, \cdots, v_m) \mid p(v_1, \cdots, v_m) \in \#\{C_1, \cdots, C_n\}\}}{\mathsf{project}\, p\, \#\{C_1, \cdots, C_n\} \longrightarrow v} \qquad (\text{E-Project})
$$

states that given the predicate symbol p and the constraint set $\#\{C_1, \cdots, C_n\}$, the expression reduces to those ground facts in $\#\{C_1, \cdots, C_n\}$ that share the same predicate symbol. Note that the project expression returns ground facts, and implicitly strips out non-ground facts and rules.

The first evaluation rule for the internal select expression:

$$\frac{C_i = \forall(x_1 : \tau_1, \cdots, x_n : \tau_n).\, A \overset{e}{\Leftarrow} B_1, \cdots, B_{n'} \qquad \nu \text{ is a primitive valuation of } x_i \text{ s.t. typeOf}(\nu(x_i)) = \tau_i}{\mathcal{S}(\#\{C_1, \cdots, C_{n''}\}) \longrightarrow \mathcal{H}(\nu(A), \nu(e), \#\{C_1, \cdots, C_{n''}\})} \quad \text{(S-Rule)}$$

states that given a constraint set $\#\{C_1, \cdots, C_n\}$, the Datalog solver may non-deterministically select any constraint C_i and any *primitive* valuation ν of the quantified variables $x_1 : \tau_1, \cdots, x_m : \tau_m$. The valuation must assign a value of type τ_i to each quantified variable x_i and the value must be a primitive value. We do not allow complex types, such as functions and constraint sets, as terms. The evaluation rule is in Church-style since evaluation depends on the types of the quantified variables $x_1, \cdots, x_n$. Given a primitive valuation ν, the expression applies it to the head atom A and the filter expression e, and reduces to the extend expression with these components together with the original constraint set. Intuitively, the (S-Rule) models the Datalog solver when it selects a Datalog rule for evaluation, binds its quantified variables, and proceeds to evaluate the head atom.

The second evaluation rule for the internal select expression:

$$\frac{v = \{p(v_1, \cdots, v_n) \mid p(v_1, \cdots, v_n) \in \#\{C_1, \cdots, C_{n'}\}\}}{\mathcal{S}(\#\{C_1, \cdots, C_{n'}\}) \longrightarrow v} \quad \text{(S-Finish)}$$

states that the select expression $\mathcal{S}(\#\{C_1, \cdots, C_n\})$ may immediately evaluate to the facts of $\#\{C_1, \cdots, C_n\}$. Together, the (S-Rule) and (S-Finish) evaluation rules model a non-deterministic execution where constraints are repeatedly instantiated, their head atoms become part of the constraint set (if the filter expression reduces to `true`), and at some point the facts of the constraint set are returned. This over-approximates the concrete evaluation steps of any reasonable Datalog engine without modeling its exact semantics.

The extend expression has two evaluation rules, (H-True) and (H-False), depending on the value of the filter expression. The first rule:

$$\frac{A = p(v_1, \cdots, v_n)}{\mathcal{H}(A, \texttt{true}, \#\{C_1, \cdots, C_n\}) \longrightarrow \mathcal{S}(\#\{A, C_1, \cdots, C_n\})} \quad \text{(H-True)}$$

states that the expression $\mathcal{H}(A, \texttt{true}, \#\{C_1, \cdots, C_n\})$ reduces to the select expression where the constraint set has been extended with the fact A. Note that the side-condition requires that all terms of A are values, i.e. that A is a ground fact.

Substitution. As is standard, we use capture avoiding substitution to implement beta reduction for λ_{Dat}. The substitution rules are available in appendix.

Evaluation Contexts. As is also standard, we use evaluation contexts to allow reduction of sub-expressions [Felleisen et al. 2009]. Figure 4 shows the evaluation contexts of λ_{Dat}. We use □ to represent the hole in the expression tree. If E is an evaluation context, then $E[e]$ is an expression in which the hole □ has been replaced by the expression e. We assume that the semantics are extended with the rule:

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \quad \text{(E-Context)}$$

$$\begin{array}{rcl} \tau \in \mathit{Type} &=& \alpha \mid \iota \mid \tau_1 \rightarrow \tau_2 \mid (\tau_1, \cdots, \tau_n) \mid r \\ r, s \in \mathit{Row} &=& \rho \mid \{\,\} \mid \{\, p = (\tau_1, \cdots, \tau_n) \;\mid\; r \,\} \\ \alpha \in \mathit{TypeVar} &=& \text{is a set of type variables.} \\ \rho \in \mathit{RowVar} &=& \text{is a set of row variables.} \end{array}$$

(a) Mono Types of λ_{DAT}.

$$\begin{array}{rcll} \sigma \in \mathit{Scheme} &=& \forall \overline{\alpha}\, \forall \overline{\rho}.\, \tau & \textit{poly type} \\ \alpha \in \mathit{TypeVar} &=& \text{is a set of type variables.} & \\ \rho \in \mathit{RowVar} &=& \text{is a set of row variables.} & \end{array}$$

(b) Type Schemes of λ_{DAT}.

Fig. 5. Mono Types and Type Schemes of λ_{DAT}.

In the proofs, we will sometimes refer to a specific instantiation of this meta-rule in combination with a specific evaluation context. For example, we might refer to the context rule in reference to the first evaluation context for function application, i.e. the context rule with $E = E'\ e$ for some E'.

The evaluation contexts for λ_{DAT} are straightforward. In the case of the extend expression, the context enforces that we evaluate the constraint set first, then the filter expression, and finally the terms (expressions) of the head atom itself.

4 TYPING OF λ_{DAT}

We now discuss the type system for λ_{DAT}. The type system is based on Hindley-Milner [Damas and Milner 1982; Wright and Felleisen 1994] extended with row types. We use row types to precisely track the types of predicate symbols in constraint sets. In this way, the type system is reminiscent of type systems for extensible records [Leijen 2005]. We have proved soundness of the type system. The detailed proofs are available in appendix.

4.1 Mono and Poly Types

The type system, as is standard for Hindley-Milner style type systems, splits types into mono- and poly types. Figure 5a shows the mono types of λ_{DAT}. The mono types consist of type variables α, a set of base types denoted by ι (e.g. `Bool`), function types $\tau_1 \longrightarrow \tau_2$, tuple types $(\tau_1, \cdots, \tau_n)$, and row types r. A row type is either a row type variable ρ, an empty row $\{\}$, or a row extension $\{p = (\tau_1, \cdots, \tau_n) \mid r\}$. A row type describes the types of the predicate symbols of a constraint set. For example, the constraint set:

$$\#\{A(123).,B(\texttt{"Hello World"}).\}$$

may be assigned the row type:

$$\{A = (\texttt{Int}) \mid \{B = (\texttt{Str}) \mid \{\}\}\}$$

For brevity, we will write such types as: $\{A = (\texttt{Int}), B = (\texttt{Str})\}$. In our extension of Flix both constraint sets and row types are written using #{}. In the calculus, for clarity, we write *constraint sets* as $\#\{\cdots\}$ and *row types* as $\{\cdots\}$ without the hash #.

Figure 5b shows the poly types (or type schemes) of λ_{DAT}. A poly type is of the form:

$$\forall \alpha_1, \cdots, \alpha_n\, \forall \rho_1, \cdots, \rho_m.\, \tau$$

As can be seen, we separate regular type variables α from row type variables ρ. We use poly types to type polymorphic constraint sets. For example, the constraint set:

$$\#\{\mathsf{Path}(x, z) \Leftarrow \mathsf{Path}(x, y), \mathsf{Edge}(y, z).\}$$

is given the polymorphic type:

$$\forall \alpha_1, \alpha_2\, \forall \rho.\, \{\mathsf{Path} = (\alpha_1, \alpha_2), \mathsf{Edge} = (\alpha_2, \alpha_2) \mid \rho\}$$

The type is polymorphic in the terms of the Edge and Path atoms. Specifically, the variable x must have type α_1 whereas the variables y and z must have type α_2, because the two occurrences of the

$$\tau \cong \tau \quad (\textsc{Eq-Refl}) \qquad \frac{\forall i.\, \tau_i \cong \tau_i'}{(\tau_1, \cdots, \tau_n) \cong (\tau_1', \cdots, \tau_n')} \ (\textsc{Eq-Tuple}) \qquad \frac{\tau \cong \tau' \quad r \cong s}{\{\, p : \tau \mid r \,\} \cong \{\, p : \tau' \mid s \,\}} \ (\textsc{Eq-Head})$$

$$\frac{\tau_1 \cong \tau_1' \quad \tau_2 \cong \tau_2'}{\tau_1 \rightarrow \tau_2 \cong \tau_1' \rightarrow \tau_2'} \ (\textsc{Eq-Arrow}) \qquad \frac{\tau_1 \cong \tau_2 \quad \tau_2 \cong \tau_3}{\tau_1 \cong \tau_3} \ (\textsc{Eq-Trans}) \qquad \frac{p \neq p'}{\{\, p : \tau,\, p' : \tau' \mid r \,\} \cong \{\, p' : \tau',\, p : \tau \mid r \,\}} \ (\textsc{Eq-Swap})$$

Fig. 6. Type Equivalence in λ_{DAT}.

Path atom force the variables y and z to have the same type. But, the type is also polymorphic in the rest of the row ρ, since the constraint set can be composed with other constraint sets that (potentially) contain additional predicate symbols.

As is standard, we introduce a partial order on types [Damas and Milner 1982; Wright and Felleisen 1994]. Given two poly types σ_1 and σ_2, we say that σ_1 is more general than σ_2, written as $\sigma_1 \sqsubseteq \sigma_2$, if there is a substitution s of quantified variables of σ_1 such that $s(\sigma_1) = \sigma_2$. For example:

$$\forall \alpha.\alpha \rightarrow \alpha \sqsubseteq \texttt{Int} \rightarrow \texttt{Int}$$
$$\forall \rho.\{A = (\texttt{Int}) \mid \rho\} \sqsubseteq \{A = (\texttt{Int}), B = (\texttt{Int})\}$$
$$\forall \alpha, \rho.\{A = \alpha, B = (\texttt{Int}) \mid \rho\} \sqsubseteq \{A = (\texttt{Int}), B = (\texttt{Int}), C = (\texttt{Bool})\}$$

The least element of the partial order is $\forall \alpha.\alpha$.

4.2 Type Equality

Consider the two constraint sets:

$$\#\{A(123).\ B(456).\} \quad \text{and} \quad \#\{B(456).\ A(123).\}$$

We can assign them the two types:

$$\{A = (\texttt{Int}), B = (\texttt{Int})\} \quad \text{and} \quad \{B = (\texttt{Int}), A = (\texttt{Int})\}$$

These two types are not equivalent, since the order of the predicate symbols is different! But clearly, from a Datalog perspective, the two constraints sets are equivalent.

To fix this, we introduce an equivalence relation on types. Specifically, we consider two row types to be equivalent *up to permutation of distinct labels* following Leijen [2005]. Figure 6 specifies this equivalence relation $\cong$ on types. The [Eq-Refl] and [Eq-Trans] rules specify that the relation is reflexive and transitive. In the appendix, we prove that the relation is symmetric, hence it is an equivalence relation. The [Eq-Arrow] and [Eq-Tuple] rules specify that function and tuple types are equivalent if their constituents are. The [Eq-Head] rule specifies that two row types are equivalent if their first predicate symbols are the same, they have equivalent types, and the rest of the two rows are equivalent. The [Eq-Swap] rule specifies that two row types are equivalent if they are equivalent when the first two predicates of one of them are swapped, provided that the predicate symbols are distinct. With this in place, we have that:

$$\{A = (\texttt{Int}),\ B = (\texttt{Int})\} \cong \{B = (\texttt{Int}),\ A = (\texttt{Int})\}$$

The type system permits rows with duplicate predicate symbols. In the context of extensible records with scoped labels such types make sense and can be useful, but in the context of our work they are more of an artifact. We tolerate them because they simplify the formalism and the implementation of type inference. We refer the reader to Leijen [2005] for more details.

4.3 Examples

Before we present the type system in detail, we give some examples of how it works.

Example I. Consider the program fragment:

```
let p1 = Edge("a", "b").;
let p2 = Path("a", "b").;
let p3 = p1 <+> p2;
```

The type system will assign the two local variables p_1 and p_2 the row polymorphic types:

$$p_1 : \forall r_1.\{\mathsf{Edge} = (\texttt{Str}, \texttt{Str}) \mid r_1\}$$
$$p_2 : \forall r_2.\{\mathsf{Path} = (\texttt{Str}, \texttt{Str}) \mid r_2\}$$

The type rule for composition:

$$\frac{\Gamma \vdash e_1 : r \quad \Gamma \vdash e_2 : r' \quad r \cong r'}{\Gamma \vdash e_1 \texttt{ <+> } e_2 : r} \tag{T-Compose}$$

requires that the types of p_1 and p_2 are equivalent. We can instantiate the two polymorphic types using the type rule for variables:

$$\frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau} \tag{T-Var}$$

which allow us to replace r_1 by $\{\mathsf{Path} = (\texttt{Str}, \texttt{Str}) \mid r_3\}$ and r_2 by $\{\mathsf{Edge} = (\texttt{Str}, \texttt{Str}) \mid r_3\}$. We pick the same type variable r_3 for both instantiations; otherwise the composition rule would be inapplicable. The type rule for composition now allows us to conclude that p_1 <+> p_2 has type:

$$\{\mathsf{Edge} = (\texttt{Str}, \texttt{Str}), \mathsf{Path} = (\texttt{Str}, \texttt{Str}) \mid r_3\}$$

Using the type rule for let:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \mathsf{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2} \tag{T-Let}$$

We obtain the polymorphic type:

$$\forall r_3.\{\mathsf{Edge} = (\texttt{Str}, \texttt{Str}), \mathsf{Path} = (\texttt{Str}, \texttt{Str}) \mid r_3\}$$

for the local variable p_3.

Example II. Consider the program fragment for some labelled graph:

```
let p1 = Edge("a", 123, "b").;
let p2 = Path(x, l, z) :- Path(x, l, y), Edge(y, l, z).;
let p3 = p1 <+> p2;
```

The types of the three local variables are:

$$p_1 : \forall r_1.\{\mathsf{Edge} = (\texttt{Str}, \texttt{Int}, \texttt{Str}) \mid r_1\}$$
$$p_2 : \forall \alpha_1, \alpha_2, \alpha_3, \forall r_2.\{\mathsf{Edge} = (\alpha_1, \alpha_2, \alpha_3), \mathsf{Path} = (\alpha_1, \alpha_2, \alpha_3) \mid r_2\}$$
$$p_3 : \forall r_3.\{\mathsf{Edge} = (\texttt{Str}, \texttt{Int}, \texttt{Str}), \mathsf{Path} = (\texttt{Str}, \texttt{Int}, \texttt{Str}) \mid r_3\}$$

Note how the type of p_2 is polymorphic in terms of Edge and Path, but once p_1 and p_2 are composed in p_3, the types of both Edge and Path become ($\texttt{Str}, \texttt{Int}, \texttt{Str}$).

Example III. If we modify the previous program to the following:

```
let p1 = Edge("a", 123, "b").;
let p2 = Path(x, l1, z) :- Path(x, l1, y), Edge(y, l2, z).;
let p3 = p1 <+> p2;
```

We get the more interesting types:

$$\begin{aligned}
&p_1 : \forall r_1. && \{\mathtt{Edge} = (\mathtt{Str}, \mathtt{Int}, \mathtt{Str}) \mid r_1\} \\
&p_2 : \forall \alpha_1, \alpha_2, \alpha_2', \alpha_3, \forall r_2. && \{\mathtt{Edge} = (\alpha_1, \alpha_2, \alpha_3), \mathtt{Path} = (\alpha_1, \alpha_2', \alpha_3) \mid r_2\} \\
&p_3 : \forall \alpha_2', \forall r_3. && \{\mathtt{Edge} = (\mathtt{Str}, \mathtt{Int}, \mathtt{Str}), \mathtt{Path} = (\mathtt{Str}, \alpha_2', \mathtt{Str}) \mid r_3\}
\end{aligned}$$

Note how the type of p_2 is now polymorphic in both labels l_1 (with type α_2') and l_2 (with type α_2). Consequently, p_3 becomes polymorphic in the label l_1 (α_2').

Ill-typed Example I. The program fragment:

```
let p1 = Edge("a", "b").;
let p2 = Edge("a", 42, "b").;
let p3 = p1 <+> p2;
```

cannot be typed since the types of p_1 and p_2:

$$\forall r_1.\{\mathtt{Edge} = (\mathtt{Str}, \mathtt{Str}) \mid r_1\} \quad \text{and} \quad \forall r_2.\{\mathtt{Edge} = (\mathtt{Str}, \mathtt{Int}, \mathtt{Str}) \mid r_2\}$$

cannot be unified.

Ill-typed Example II. The program fragment:

```
let p1 = Edge("a", 12345, "b").;
let p2 = Edge("a", "abc", "b").;
let p3 = p1 <+> p2;
```

cannot be typed since the types of p_1 and p_2:

$$\forall r_1.\{\mathtt{Edge} = (\mathtt{Str}, \mathtt{Int}, \mathtt{Str}) \mid r_1\} \quad \text{and} \quad \forall r_2.\{\mathtt{Edge} = (\mathtt{Str}, \mathtt{Str}, \mathtt{Str}) \mid r_2\}$$

cannot be unified.

Ill-typed Example III. Similarly, the single constraint:

```
Path(x, 42, z) :- Path(x, "foo", y), Edge(y, z).
```

cannot be typed since the types of the Path atom in the head and body cannot be unified.

4.4 Type Rules

We now present the type rules of λ_{DAT}. The type rules are a syntax-directed formulation of the Hindley-Milner type system where instantiation occurs in the type rule for variables [T-Var] and generalization occurs in the type rule for let-bindings [T-Let] [Damas and Milner 1982; Wright and Felleisen 1994]. Figure 7 shows the type rules of λ_{DAT}. The type system has three type judgements: $\Gamma \vdash e : \tau$ for expressions, $\Gamma \vdash_c C : r$ for constraints, and $\Gamma \vdash_p A : r$ for atoms. We assume that there is a function typeOf : *Literal* ⟶ *Type* that assigns a primitive type to every constant literal.

We now discuss the most important type rules:

The type rule for composition was shown in Example I. It states that e_1 and e_2 must have row types r and r', and they must be equivalent $r \cong r'$:

$$\frac{\Gamma \vdash e_1 : r \quad \Gamma \vdash e_2 : r' \quad r \cong r'}{\Gamma \vdash e_1 \mathbin{<\!\!+\!\!>} e_2 : r} \qquad \text{(T-Compose)}$$

The type rule for project states that e must have a row type where the predicate p has some type $(\tau_1, \cdots, \tau_n)$, and then the result is a new row type with the same type for the predicate p:

$$\frac{\mathrm{typeOf}(c) = \iota}{\Gamma \vdash c : \iota} \quad \text{(T-Cst)}$$

$$\frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1' \quad \tau_1 \cong \tau_1'}{\Gamma \vdash e_1\, e_2 : \tau_2} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \mathrm{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2} \quad \text{(T-Let)}$$

$$\frac{\Gamma \vdash e_1 : r \quad \Gamma \vdash e_2 : r' \quad r \cong r'}{\Gamma \vdash e_1 \texttt{ <+> } e_2 : r} \quad \text{(T-Compose)}$$

$$\frac{\Gamma \vdash e : r}{\Gamma \vdash \mathsf{solve}\, e : r} \quad \text{(T-Solve)}$$

$$\frac{\Gamma \vdash e_1 : r \quad \Gamma \vdash e_2 : r' \quad r \cong r'}{\Gamma \vdash \mathsf{subset}\, e_1\, e_2 : \texttt{Bool}} \quad \text{(T-Subset)}$$

$$\frac{\Gamma \vdash e : r \quad r \cong \{p = (\tau_1, \cdots, \tau_n) \mid r'\}}{\Gamma \vdash \mathsf{project}\, p\, e : \{p = (\tau_1, \cdots, \tau_n) \mid r''\}} \quad \text{(T-Project)}$$

$$\frac{\Gamma \vdash e : r}{\Gamma \vdash \mathcal{S}(e) : r} \quad \text{(T-Select)}$$

$$\frac{\Gamma \vdash_p A : r \quad \Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : r' \quad r \cong r'}{\Gamma \vdash \mathcal{H}(A, e_1, e_2) : r} \quad \text{(T-Extend)}$$

$$\frac{\forall i.\, (\Gamma \vdash_c C_i : r_i \wedge r_i \cong r)}{\Gamma \vdash \#\{C_1, \cdots, C_n\} : r} \quad \text{(T-Constraint-Set)}$$

$$\frac{\begin{array}{c}\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p A_0 : r \\ \forall i.\, \Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p B_i : r_i \\ \forall i.\, r \cong r_i \quad \Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash e : \texttt{Bool}\end{array}}{\Gamma \vdash_c \forall (x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0 \overset{e}{\Leftarrow} B_1, \cdots, B_n : r} \quad \text{(T-Constraint)}$$

$$\frac{\forall i.\, \Gamma \vdash t_i^h : \tau_i}{\Gamma \vdash_p p(t_1^h, \cdots, t_n^h) : \{p = (\tau_1, \cdots, \tau_n) \mid r\}} \quad \text{(T-Head-Atom)}$$

$$\frac{\forall i.\, \Gamma \vdash t_i^b : \tau_i}{\Gamma \vdash_p p(t_1^b, \cdots, t_n^b) : \{p = (\tau_1, \cdots, \tau_n) \mid r\}} \quad \text{(T-Body-Atom-1)}$$

$$\frac{\forall i.\, \Gamma \vdash t_i^b : \tau_i}{\Gamma \vdash_p \mathsf{not}\, p(t_1^b, \cdots, t_n^b) : \{p = (\tau_1, \cdots, \tau_n) \mid r\}} \quad \text{(T-Body-Atom-2)}$$

$$\begin{array}{c}\mathrm{gen}(\Gamma, \tau) = \forall \alpha_1, \cdots, \alpha_n, \forall \rho_1, \cdots, \rho_n.\tau \\ \text{where } \{\alpha_1, \cdots, \alpha_n, \rho_1, \cdots, \rho_n\} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma)\end{array}$$

Fig. 7. Type Rules of λ_{DAT}.

$$\frac{\Gamma \vdash e : r \quad r \cong \{p = (\tau_1, \cdots, \tau_n) \mid r'\}}{\Gamma \vdash \mathsf{project}\, p\, e : \{p = (\tau_1, \cdots, \tau_n) \mid r''\}} \quad \text{(T-Project)}$$

The type rule for the internal select expression states that e must have a row type r and the result has type r:

$$\frac{\Gamma \vdash e : r}{\Gamma \vdash \mathcal{S}(e) : r} \quad \text{(T-Select)}$$

The type rule for the internal extend expression states that the atom A must have row type r under the typing judgement for atoms $\vdash_p$, the filter expression e_1 must have type `Bool`, and the constraint set e_2 must have row type r'. Finally, r and r' must be equivalent, i.e. $r \cong r'$:

$$\frac{\Gamma \vdash_p A : r \quad \Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : r' \quad r \cong r'}{\Gamma \vdash \mathcal{H}(A, e_1, e_2) : r} \quad \text{(T-Extend)}$$

The type rule for a constraint set states that the type of a constraint set $\#\{C_1, \cdots, C_n\}$ is a row type r if each of the constraints C_i are typeable under the $\vdash_c$ typing judgement with some row type r_i and all of the row types are equivalent to r:

$$\frac{\forall i.\, (\Gamma \vdash_c C_i : r_i \land r_i \cong r)}{\Gamma \vdash \#\{C_1, \cdots, C_n\} : r} \qquad \text{(T-Constraint-Set)}$$

Intuitively, the rule ensures that the types of the predicates within multiple constraints have consistent types.

The type rule for a single constraint is:

$$\frac{\begin{array}{c}\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p A_0 : r \\ \forall i.\, \Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p B_i : r_i \\ \forall i.\, r \cong r_i \quad \Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash e : \mathtt{Bool}\end{array}}{\Gamma \vdash_c \forall(x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0 \overset{e}{\Leftarrow} B_1, \cdots, B_n : r} \qquad \text{(T-Constraint)}$$

It states that a constraint $\forall(x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0 \overset{e}{\Leftarrow} A_1, \cdots, A_n$ has row type r under a type environment Γ if we can type its constituents as follows: (i) the head atom A_0 must have type r under the extended environment $\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m$, (ii) each body atom A_i must have type r_i under the extended environment $\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m$, (iii) all the row types, r and each r_i must be equivalent, and (iv) the filter expression e must have type Bool under the extended environment $\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m$. Intuitively, the rule ensures that the types of the atoms within a single constraint have consistent types.

The type rule for a head atom is:

$$\frac{\forall i.\, \Gamma \vdash t_i^h : \tau_i}{\Gamma \vdash_p p(t_1^h, \cdots, t_n^h) : \{p = (\tau_1, \cdots, \tau_n) \mid r\}} \qquad \text{(T-Head-Atom)}$$

It states that the type of an atom $A = p(t_1^h, \cdots, t_n^h)$ is a row type of the form $\{p(\tau_1, \cdots, \tau_n) \mid r\}$ for some row type r provided that each expression term t_i^h has type τ_i according to the expression typing judgement $\vdash$. The type rules [T-Body-Atom-1] and [T-Body-Atom-2] are similar.

4.5 Type Safety

We have established type soundness for λ_{DAT}. We state the most important theorems here:

Theorem 4.1 (Progress). *Suppose e is a closed, well-typed expression (that is, $\vdash e : \tau$ for some τ). Then either e is a value or else there is some expression e' such that $e \longrightarrow e'$.*

Theorem 4.2 (Preservation). *If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau'$ where $\tau = \tau'$.*

The detailed proofs are available in the appendix.

4.6 Type Inference

The type system, as an instance of Hindley-Milner, supports full type reconstruction. The type system for λ_{DAT} is based on that for polymorphic extensible records [Leijen 2005]. We leave it as future work to prove its correctness, but we believe that the proofs should be easily adaptable. The relationship between records and first-class constraints is as follows: Given the Datalog constraint:

$$p_0(t_1^0, \cdots, t_l^0) \Leftarrow p_1(t_1^1, \cdots, t_{l'}^1), \cdots, p_n(t_1^n, \cdots, t_{l''}^n)$$

we can view it as a record in which the fields are the predicate symbols $p_0, \cdots, p_n$ and the types of the fields are the types of the terms, i.e. the "field" p_0 would have the tuple type $(\tau_1, \cdots, \tau_l)$, where τ_i is the type of the term t_i^0.

4.7 Stratification

Unrestricted use of negation poses problems for Datalog. Consider the following program:

```
P(x) :- not Q(x).        Q(x) :- not P(x).
```

If we assume that the program contains the constant c, then the program has *two* minimal models:

$$m_1 = \{P(c)\} \quad \text{and} \quad m_2 = \{Q(c)\}$$

This is undesirable because the minimal model is no longer unique. We can ensure a consistent semantics by restricting ourselves to *stratified* Datalog [Minker 1988]. Stratification requires that the predicate symbols of a Datalog program can be partitioned into a sequence of *strata* such that a positive predicate symbol in a stratum only depends on predicate symbols in the same or lower strata and a negative predicate symbol only depends on predicate symbols in strictly lower strata. Stratification splits a Datalog program P into a sequence of sub-programs $P_1, \cdots, P_n$ such that the output of P_i becomes the input of P_{i+1}.

Definition 4.3. (Precedence Graph) The *precedence graph* of a program P is a graph that contains:

- a *positive edge* $a \leftarrow b$ if P contains a rule where a is the head predicate and b is a positive body predicate, and similarly
- a *negative edge* $a \not\leftarrow b$ if P contains a rule where a is the head predicate and b is a negative body predicate.

We can use the precedence graph to determine whether a Datalog program is stratified:

Definition 4.4. (Stratified) A program P is *stratified* if its precedence graph contains no cycles with a negative edge.

Given a complete Datalog program, we can determine if it is stratified and compute its stratification using Ullman's Algorithm [Ullman 1988]. For a language like λ_{DAT}, we can identify three levels of modularity of a type system or static analysis. At the least modular level, we require the complete Datalog program for analysis of stratification. At the second level, we require the complete λ_{DAT} program, which contains Datalog constraint sets that will be composed at run time. Finally, at the most modular level, we may be given fragments of λ_{DAT} programs, and be required to construct a summary of each fragment such that stratification of a complete λ_{DAT} program can be computed using the summaries alone.

For the first level, stratification of the complete Datalog program can be computed using Ullman's Algorithm [Ullman 1988]. In the context of λ_{DAT}, however, such an approach would only be applicable at runtime when the complete Datalog programs are known, so stratification errors would be detected only at runtime. The strategy that we will describe in the rest of this section supports the second level of modularity, which allows analyzing Datalog constraint sets at compile time. The analysis can therefore detect stratification errors at compile time. The analysis in this section does not support the third level of modularity. At this level, there is a range of design choices in how detailed and complex the summaries of the λ_{DAT} fragments should be. An analysis is deemed modular if it reads the summaries instead of the code. At the extreme, if the full details contained in the code are recorded in the summaries, then any analysis can be made modular in name, but then analyzing the summaries will be equivalent to analyzing the original code. At this level, there is a rich design space with trade offs between precision and complexity of the summaries which we leave for exploration in future work.

Definition 4.5 (Stratification for λ_{DAT}). Given a *complete* λ_{DAT} program e, determine at compile-time whether all Datalog constraint sets that e may construct at run-time are stratified.

We propose a simple and sound algorithm that computes stratification at compile time using information from the type system. We view this as an important first step towards stratification of Flix programs, but there is still a rich design space to be explored.

Definition 4.6. (First-Class Constraint Stratification) For a whole λ_{DAT} or Flix program:

(1) Collect all constraints into a set $\mathcal{G}$, regardless of where they occur in the program.
(2) Compute the global dependency graph $\mathcal{D} = dg(\mathcal{G})$ of the constraint set $\mathcal{G}$.
(3) For each expression e with a row type $r = \{p_1 = (\cdots), \cdots, p_n = (\cdots)\}$, compute a subgraph $\mathcal{R}$ of the dependency graph with only those edges $a \leftarrow b$ or $a \nleftarrow b$ where both predicates a and b occur in the row type. Finally, use Ullman's Algorithm to determine if the dependency graph $\mathcal{R}$ is stratified, and to compute its stratification.

We do not need to consider row type variables because stratification occurs after monomorphization. For example, consider the following program fragment:

```
let r1 = B(x) :- A(x).;
let r2 = C(x) :- B(x).;
let r3 = K(x) :- A(x).;
let r4 = A(x) :- not C(x), R(x).;
```

The dependency graph $\mathcal{G}$ of this program fragment is:

$$\{B \leftarrow A, C \leftarrow B, K \leftarrow A, A \nleftarrow C, A \leftarrow R\}$$

The dependency graph is not stratified, since there is a cycle with a negative edge:

$$C \leftarrow B \leftarrow A \nleftarrow C$$

However, if we consider the expression: `r3 <+> r4` which has the row type:

$$\{A = (\cdots), C = (\cdots), K = (\cdots), R = (\cdots)\}$$

we see that its restricted dependency graph R does not have any edges for B:

$$\{K \leftarrow A, A \nleftarrow C, A \leftarrow R\}$$

and hence the constraint sets that the expression `r3 <+> r4` may evaluate to must be stratified and we can compute the stratification at compile-time. We now prove correctness of the algorithm.

Lemma 4.7. *Given any subset of constraints $\mathcal{U} \subseteq \mathcal{G}$, the dependency graph of $\mathcal{U}$ is a subgraph of the dependency graph of $\mathcal{G}$, i.e. $dg(\mathcal{U}) \subseteq dg(\mathcal{G})$ by the definition of the dependency graph.*

Theorem 4.8 (Correctness). *If algorithm 4.6 reports that an expression e is stratified, then all constraint sets that e may evaluate to must be stratified.*

Proof. An expression with row type $\{p_1 = (\cdots), \cdots, p_n = (\cdots)\}$ may evaluate to some constraint set $\mathcal{Q}$ which can be partitioned into facts $\mathcal{F}$ and rules C such that $\mathcal{Q} = \mathcal{F} \cup C$ and we have that $C \subseteq \mathcal{G}$. By inversion of the typing derivation of e, the constraint set C can only contain constraints with predicate symbols drawn from $p_1, \cdots, p_n$. By definition, the precedence graph $dg(C)$ can then only contain edges with predicates drawn from $p_1, \cdots, p_n$. By the previous lemma, the dependency graph $dg(C)$ must be a subgraph of $dg(\mathcal{G})$, but the largest subgraph of $dg(\mathcal{G})$ with edges where the predicates are drawn from $p_1, \cdots, p_n$ is $\mathcal{R}$. Hence we must have that $dg(C) \subseteq \mathcal{R}$. Now, if $\mathcal{R}$ is stratified, then $dg(C)$ must also be stratified, since if a graph has no negative cycles then a subgraph cannot have any negative cycles. □

The algorithm is an over-approximation, and it will sometimes unfairly reject programs that can never fail at run-time. For example, the program:

```
def f(b: Bool): #{...} =
  let r1 = P(x) :- A(x), not Q(x).;
  let r2 = Q(x) :- A(x), not P(x).;
  solve (if (b) r1 else r2)
```

is unfairly rejected, even though it is always stratified.

5 IMPLEMENTATION

We have implemented λ_{DAT} as an extension of the Flix programming language. Flix is a functional, imperative, and logic programming language. Flix supports algebraic data types, pattern matching, currying, higher-order functions, extensible records, channel and process-based concurrency, and now first-class Datalog constraints.

The extension required 5,000 lines of code in addition to the 55,000 lines of code already present in the compiler. The implementation effort was significant, but not difficult, as the constructs of λ_{DAT} integrate seamlessly with an ML-style language. Most changes were to the frontend of the compiler and to the intricate details of the type inference algorithm. We reuse the Datalog engine that already comes with Flix to solve Datalog constraints at run-time.

Flix is open source, ready for use, and freely available at: flix.dev

6 EVALUATION: CASE STUDIES

To demonstrate the practical value of the λ_{DAT} calculus and its implementation in Flix, we present a series of case studies. We report on programs that have already been implemented in Flix and on one program that could benefit from being implemented in Flix (or in a system based on λ_{DAT}).

6.1 Case Study: Koans

We have implemented eleven "koans" which are small programs that demonstrate how to program with first-class constraints. The koans solve practical computational problems such as: (i) Given a Git commit graph, find the pair of commits where a bug was introduced and where the bug was merged into the master branch. (ii) Given a social network graph, compute a set of friend suggestions based on the friends of my friends. (iii) Given a train and bus network, compute if there is a route from one city to another with a preference for the train. (iv) Given a list of graphs, find all pairs of graphs whose union is acyclic. The koans illustrate the interplay between functional and logic programming: the overall program is constructed from smaller functions that use first-class constraints. The koans are available at https://flix.dev/papers/flix-koans.pdf.

6.2 Case Study: PuppetMaster an Actor Library with Declarative Actor Supervision

PuppetMaster is an actor library for Flix. An *actor* is a light-weight process that executes concurrently with—and in isolation from—other processes. Every actor has its own unique mailbox, an unbounded queue of incoming messages. Actors communicate by sending immutable messages to each other's mailboxes. An *actor system* is a collection of actors together with policies that govern how actors are started, how actors are stopped, and how to respond when an actor crashes. In PuppetMaster, startup, shutdown, and supervision policies are expressed as first-class Datalog constraints. The "input" (i.e. ground facts) of a policy is the state of every actor in the system and the "output" (i.e. minimal model) of a policy is a set of actions to be executed by the actor system.

For example, the very simple policy (where ActorPolicy is a type alias omitted for brevity):

```
def immediatelyStartAllPolicy(): ActorPolicy = #{
    Start(actor) :- Actor(actor).
}
```

specifies that every actor should be started immediately. A more sophisticated policy is:

```
def defaultStartPolicy(): ActorPolicy = #{
    Waiting(x) :- DependsOn(x, y), not ActorState(y, Running).
    Start(x) :- Actor(x), not Waiting(x).
}
```

which allows an actor to start once all its dependencies have entered the `Running` state. An even more sophisticated policy is:

```
def defaultSupervisionPolicy(): ActorPolicy = #{
  Path(x, y) :- DependsOn(x, y).
  Path(x, z) :- Path(x, y), DependsOn(y, z).
  Waiting(x) :- Path(x, w), ActorState(w, NonResumablyCrashed).
  Resume(x) :- ActorState(x, ResumablyCrashed), not Waiting(x).
}
```

which allows a resumably crashed actor x to continue execution unless one of its transitive dependencies have non-resumably crashed. The PuppetMaster library ships with a collection of such startup, shutdown, and supervision policies, but the programmer may also define his or her own policies using first-class constraints.

6.3 Case Study: A Prototype Points-To Analysis for Python

We have implemented a prototype points-to analysis for Python with a special focus on precise object initialization. The points-to analysis is parameterized by the choice of context- and heap-sensitivity in the form of two functions, merge and record, following Smaragdakis et al. [2011]:

```
def analysis(merge: (String, octx, String, ctx) -> ctx,
                 record: (String, ctx) -> octx, empty_ctx : ctx): #{...}
```

where the analysis function is polymorphic in the type of the call context `ctx` and the type of the heap context `octx`. The merge and record functions are used within the analysis rules:

```
Reachable(attrObject, merge(baseObject, baseOCtx, invo, callerCtx)) :-
  VCall(base, attr, invo, inMeth),
  VarPointsTo(base, callerCtx, baseObject, baseOCtx),
  AttrPointsTo(baseObject, baseOCtx, attr, attrObject, octx),
  ObjectIsFunction(attrObject),
  Reachable(inMeth, callerCtx).
```

This design allows us to easily change and experiment with different types of context- and heap sensitivity. We can instantiate the analysis with no context or heap sensitivity by simply using the Unit type and passing in two constant functions. We can also instantiate the analysis with 2-call-1-heap sensitivity by defining appropriate algebraic data types and passing in appropriate record and merge functions.

Case Study: The IFDS & IDE Program Analyses. The Interprocedural Finite Distributive Subset (IFDS) algorithm [Reps et al. 1995] solves context-sensitive interprocedural dataflow analysis problems by computing reachability in a graph. The input to the algorithm is a graph called the exploded supergraph, and the algorithm works by constructing two sets of additional edges, called path edges and summary edges, according to rules about existing edges. For example, whenever there is a path edge from node n_1 to node n_2 and a supergraph edge from node n_2 to node n_3, the algorithm adds a new path edge from n_1 to n_3. There are additional rules, some of them more complicated, but they all have a similar form. Thus, it appears natural to express the rules of the IFDS algorithm as Datalog rules and to use a Datalog solver to compute the solution.

However, the exploded supergraph is generally very large and the IFDS algorithm explores only a subgraph that is reachable from the entry point of the program. It would be expensive to realize the whole exploded supergraph as a data structure, including unreachable parts of the graph. Instead, the graph is usually specified implicitly as a successor function that can compute, for each node n that is found to be reachable, the set of edges leading out from n to its successor nodes. The exploded supergraph would be too large to realize as a Datalog relation, so it is impractical to implement the IFDS algorithm in pure Datalog. In our hybrid language, the exploded supergraph can be expressed using functions in the functional fragment of the language, which can be called on demand by the rules of the IFDS algorithm expressed declaratively in the relational fragment of the language. Since the functions can be first-class this allows us to implement a generic IFDS framework once, with parameters that can later be instantiated to specific program analyses.

The Interprocedural Distributive Environment (IDE) algorithm [Sagiv et al. 1996] extends IFDS by adding representations of functions on a lattice to the edges of the exploded supergraph. Whenever a rule of the algorithm creates a new path edge or summary edge, it decorates the new edge with a new function created by composing functions on the edges that triggered the rule. When the sets of edges are represented as Datalog relations, it is easy to decorate each edge with a function by adding a new attribute to the relation to store the function. However, the rules for creating new edges no longer map to pure Datalog: each rule needs to perform a general computation to determine the function for the new edge from the functions of the existing edges. Our language makes it possible to express this by allowing a Datalog rule to call into the functional fragment of the language, which can express the general computations needed to determine the function for the new edge. As with IFDS, we can implement a single parametric IDE framework and later instantiate it with multiple IDE analyses.

6.4 Case Study: The Doop Program Analysis Framework

Doop[2] is a fast, scalable, context-sensitive, subset-based points-to analysis for Java implemented in Datalog [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2013, 2014]. Doop models many features of the Java programming language, including reflection and exceptions. The Doop Framework comes with a wide variety of configuration options for tuning the precision and performance of the points-to analysis, e.g. the choice of context sensitivity, heap sensitivity, reflection support, exception support, and a multitude of other options. The implementation uses the C preprocessor to selectively include (or exclude) rules based on the configuration options.

For example, if the `FEATHERWEIGHT_ANALYSIS` option is disabled, the C preprocessor excludes the following rule that models static field accesses:

```
#ifndef FEATHERWEIGHT_ANALYSIS
StaticFieldPointsTo(?hctx, ?value, ?signature) <-
  ReachableStoreStaticFieldFrom(?from),
  OptStoreStaticField(?signature, ?from),
  VarPointsTo(?hctx, ?value, _, ?from).
#endif
```

As another example, if the `REFLECTION` option is enabled, the C preprocessor includes the following rule that models reflective calls:

```
#ifdef REFLECTION
AnyCallGraphEdge(?from, ?to) :- ReflectiveCallGraphEdge(_, ?from, _, ?to).
#endif
```

[2] https://bitbucket.org/yanniss/doop/src/master/

Similarly, the REFLECTION_DYNAMIC_PROXIES option controls the modelling of reflective calls through proxy objects:

```
#ifdef REFLECTION_DYNAMIC_PROXIES
AnyCallGraphEdge(?from, ?to) :- ProxyCallGraphEdge(_, ?from, _, ?to).
#endif
```

These examples demonstrate that it is common to selectively include or exclude rules in Doop. In fact, a search across the Doop repository reveals 268 uses of ifdef and 189 uses of ifndef.

An interesting flag is EXCEPTIONS_CS, which controls whether context sensitivity is used when modelling the data flow of exceptions:

```
#ifdef EXCEPTIONS_CS
ThrowPointsTo(?hctx, ?heap, ?ctx, ?method) <-
    Throw(?ref, ?var),
    VarPointsTo(?hctx, ?heap, ?ctx, ?var),
#else
ThrowPointsTo(?hctx, ?heap, ?method) <-
    Throw(?ref, ?var),
    VarPointsTo(?hctx, ?heap, _, ?var),
#endif
```

Note that in the first rule, the ThrowPointsTo predicate symbol takes *four* arguments whereas in the second rule it takes only three arguments. Thus it is critical, whether ThrowPointsTo is enabled or not, that any use of ThrowPointsTo is also guarded by EXCEPTIONS_CS to ensure that the arities always match up. Another example of the same pattern is:

```
#ifdef EXCEPTIONS_CS
ThrowPointsTo(?hctx, ?heap, ?callerCtx, ?callerMethod) <-
   CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?tomethod),
   ThrowPointsTo(?hctx, ?heap, ?calleeCtx, ?tomethod),
#else
ThrowPointsTo(?hctx, ?heap, ?callerMethod) <-
   CallGraphEdge(_, ?invocation, _, ?tomethod),
   ThrowPointsTo(?hctx, ?heap, ?tomethod),
#endif
```

These examples illustrate that there is a need to selectively include or exclude certain rules where even the arity of the predicate symbols might differ. The Flix type system ensures that for any first-class constraint set constructed at run-time the arities of predicate symbols always match.

Doop uses macros to parameterize the choice of context- and heap sensitivity:

```
#define MergeMacro(callerCtx, invocation, hctx, value, calleeCtx) \
  Context(calleeCtx), \
  CtxFromRealCtx[RealContext2FromContext[callerCtx], invocation] = calleeCtx

#define MergeMacro(callerCtx, invocation, hctx, heap, calleeCtx) \
  Context(calleeCtx), \
  CtxFromRealCtx[RealHContext1FromHContext[hctx], \
                 DeclaringClass:Allocation[RealHContext2FromHContext[hctx]], \
                 heap] = calleeCtx

// .. and many more ...
```

Here the syntax $A[x] = y$ can be understood as a function call that is true if $A(x)$ evaluates to y. The two macro definitions above are examples of different context sensitivity policies. Similarly, there are different definitions of the MergeStartupMacro macro:

```
#define MergeStartupMacro(hctx, heap, calleeCtx) \
  Context(calleeCtx), \
  ContextFromRealContext[RealHContextFromHContext[hctx], heap] = calleeCtx

#define MergeStartupMacro(hctx, heap, calleeCtx) \
  Context(calleeCtx), \
  ContextFromRealContext[heap, heap] = calleeCtx

// .. and many more ...
```

and of the MergeThreadStartMacro macro:

```
#define MergeThreadStartMacro(hctx, heap, callerCtx, newCtx) \
  Context(newCtx), \
  ContextFromRealContext[RealHContextFromHContext[hctx], heap] = newCtx

#define MergeThreadStartMacro(hctx, heap, callerCtx, newCtx) \
  Context(newCtx), \
  ContextFromRealContext[heap] = newCtx

// .. and many more ...
```

In Flix, we can express such macros as arguments to higher-order functions that return constraint sets parameterized by these functions, as seen in Example II. The commonMacros file contains 11 such macros and the repository has over 800 uses of #define.

The Doop Framework demonstrates that there is a need for parametricity of Datalog programs. However, there are at least three problems with a meta-programming approach based on textual generation: (i) we cannot be sure that all of the programs output by the C preprocessor system are valid Datalog programs, (ii) we cannot change any option at runtime; we have to recompile the entire program from scratch, and (iii) we cannot determine if the program is stratified without actually constructing a specific program for a specific configuration. The λ_{DAT} calculus (and its implementation in Flix) overcomes these issues. We can use first-class Datalog constraints to model the selective inclusion of rules. We can model macro functions, such as MergeMacro, using ordinary functions. And finally, the type system of λ_{DAT} ensures that any Datalog program constructed at run-time is well-formed and stratified.

7 RELATED WORK

Datalog Integration. Arntzenius and Krishnaswami [2016] present Datafun, a typed functional programming language with constructs for fixpoint computations. The key feature of Datafun is to track monotonicity with types. If a function f is deemed monotone by the type system, this guarantees that the fixpoint of f exists and can be computed by the fix operator. Datafun is closer to a functional language than λ_{DAT}. In Datafun, the programmer writes functions and computes with these, whereas in λ_{DAT} the programmer writes constraints, composes them, and solves them. The integration in Datafun is tight: functions such as map and filter are given monotonicity types. It is also expressive: the cross product and transitive closure can be expressed as generic functions. However, Datafun programs are not solvable by standard techniques, such as semi-naive evaluation. Consequently, the extra power of Datafun comes at a cost. Most recently Arntzenius and Krishnaswami [2019] have studied semi-naive evaluation in the context of Datafun.

Madsen et al. [2016] present Flix, a programming language that extends the Datalog semantics from *constraints on relations* to *constraint on lattices*. Unlike Datalog relations, which are always finite, lattices may have an infinite number of elements. Termination is still ensured, provided that the lattices have finite height. To define the components of the lattices, e.g. the least upper bound and greatest lower bound, Flix allows Datalog predicates to refer to functions defined in a functional language. While this allows the logic part of Flix to refer to the functional part of Flix, the opposite direction is not possible. In the current work, λ_{DAT} allows integration in both ways: Datalog constraints can refer to expressions and expressions can evaluate to constraints.

Template Programming. Programmatic generation of Datalog programs is commonplace, whether based on simple string concatenation or with the use of a macro pre-processors. Recently, several macro-based meta programming languages for Datalog have appeared.

Souffle is an efficient and scalable Datalog engine that comes with its own template programming language [Scholz et al. 2016; Souffle Authors 2018]. In Souffle, a Datalog program can be organized as a set of *components* which are collections of Datalog constraints. A component can be instantiated, which copies all constraints and predicates within it, giving fresh names to all its predicate symbols. The fresh predicate symbols are then accessible through a handle to the instance. Components also support a simple form of inheritance that allows reuse of constraints and gives the ability to override (i.e. remove) constraints from a super-component. In λ_{DAT}, the combination of first-class constraints and predicate symbols allows us to emulate components, if so desired.

Template programming is a powerful technique, but it has at least two downsides: (i) programming with templates is difficult and error-prone, and (ii) template expansion can only depend on information that is available at compile time.

Logic Programming. Mercury is a strongly typed functional and logic programming language [Henderson et al. 1996; Somogyi et al. 1996, 1995]. In Mercury, a predicate has a *mode* that determines which arguments must explicitly be passed to it and how many times the predicate can succeed, i.e. whether it is deterministic or non-deterministic. This enables a mix of functional and logic programming: A function requires all its parameters and returns a deterministic result, whereas a Prolog-style goal requires only some of its arguments and may return a non-deterministic result. In λ_{DAT}, the integration between functions and predicates is less tight: We are free to call functions inside Datalog constraints, but we cannot call a Datalog predicate as an expression. Instead, we must put that predicate into a Datalog program and explicitly solve it. In comparison to Mercury, λ_{DAT} is closer to a meta-language for Datalog.

Constraint Logic Programming. Constraint Logic Programming (CLP) extends logic programming with a decidable background theory, such as lists, trees, or linear arithmetic [Cohen 1990; Jaffar and Lassez 1987; Jaffar and Maher 1994; Li and Mitchell 2003]. A CLP(X) program is a set of Horn clauses with formulae over the background theory X. During evaluation, term unification is augmented with a decision procedure for the underlying theory, e.g. an SMT solver or other specialized solver.

Datalog Extensions. Many Datalog extensions have been proposed. We detail some of them below.

Alvaro et al. [2010] present Dedalus, an extension of Datalog with time. In Dedalus, every fact is equipped with a timestamp $\mathcal{T}$ and said to hold at that instant. Dedalus rules come in two types. A *deductive* rule derives a new fact at timestamp $\mathcal{T}$ from facts already established at timestamp $\mathcal{T}$. An *inductive* rule, on the other hand, derives a new fact at timestamp $\mathcal{T} + 1$ from facts already established at timestamp $\mathcal{T}$. An important use case for Dedalus is to describe distributed systems.

Alvaro et al. [2011] present the Bloom programming language, a distributed programming language based on Datalog and built on the ideas of Dedalus. A Bloom program is a collection of rules that operate on facts with timestamps. Bloom programs are re-evaluated whenever new messages arrive, i.e. over the network or due to some local event (such as a timeout). In later work, Conway et al. [2012] extend Bloom with predicate symbols that are given a lattice interpretation, similar to the first work on Flix [Madsen et al. 2016].

Avgustinov et al. [2016] present QL, a programming language that combines logic programming with elements of object-oriented programming. QL supports classes and methods, but recast in logical terms. For example, a class is simply a set of values described by a collection of logic formulae. Sub-typing between classes is then logical implication between sets. While the QL language is rich in features, it still compiles to plain Datalog and is solvable using standard Datalog techniques.

Bembenek and Chong [2018] present FormuLog, an extension of Datalog with logical formulae. FormuLog permits terms to be constructed from boolean connectives and using functions from some underlying theory. Using FormuLog, it becomes possible to express program analyses such as symbolic execution and model checking. To solve FormuLog programs, an SMT solver is used to reason about the specific logical formulae.

While all of these extensions increase the power of Datalog, the primary contribution of the current work is an embedding of Datalog as first-class values within a functional programming language. The extension of the expressive power of Datalog by allowing expressions as head terms is a secondary contribution. In addition, we can imagine situations where one would want the expressive power of Dedalus or FormuLog combined with first-class constraints.

Type Systems for Datalog. Zook et al. [2009] present a type system for LogicBlox, a Datalog-based platform for enterprise planning. The type system is based on the notion of integrity constraints. An integrity constraint is a special form of rule that if ever instantiated indicates an error. For example, an integrity constraint might demand that every `Child(x)` is also a `Person(x)`: if there is a child that is not a person then there is a violation. Such integrity constraints can be viewed as specifying a type system. It is straightforward to check such constraints at run-time, i.e. during the fixpoint computation, and to raise an error, corresponding to a form of dynamic type checking. The authors present a static type system that can eliminate many of these run-time checks.

Schäfer and de Moor [2010] propose a type system for statically checking integrity constraints in the style of Zook et al., but for a richer language of type constraints, and with a type inference algorithm that is sound and optimal.

The work of Zook et al. [2009] and Schäfer and de Moor [2010] is orthogonal to our work: We embed Datalog programs inside a functional programming language and we want to ensure that composition of such programs is well-typed w.r.t. the arity and type of the terms. We can imagine an extension of our type system that also takes integrity constraints into account. We leave it as interesting future work to explore this direction.

8 CONCLUSION

We have proposed the idea of first-class Datalog constraints to enable the construction, composition, and evaluation of Datalog programs within a functional language.

We have proposed a modular type system, based on Hindley-Milner, in which reusable fragments of Datalog programs can be typed independently while guaranteeing that their composition is type-safe. The type system allows reuse and abstraction via polymorphism. We have proven safety of the system. We have also proposed a sound technique for computing stratification of λ_{DAT} programs at compile-time. The technique ensures that every Datalog program constructed at run-time is stratified. Our implementation is freely available as as part of the Flix programming language.

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Proc. Conference on Innovative Data Systems (CIDR)*.

Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. 2010. Dedalus: Datalog in time and space. In *International Datalog 2.0 Workshop*.

Michael Arntzenius and Neel Krishnaswami. 2019. Seminaïve evaluation for a higher-order functional language. *Proc. of ACM on Programming Languages* Principles of Programming Languages (POPL) (2019).

Michael Arntzenius and Neelakantan R Krishnaswami. 2016. Datafun: a functional Datalog. In *Proc. International Conference on Functional Programming*.

Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented queries on relational data. In *Proc. European Conference on Object-Oriented Programming (ECOOP 2016)*.

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. Principles of Database Systems (PODS)*. https://doi.org/10.1145/6012.15399

Aaron Bembenek and Stephen Chong. 2018. FormuLog: Datalog for static analysis involving logical formulae. *arXiv preprint arXiv:1809.06274* (2018).

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1640089.1640108

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What You Always Wanted to Know About Datalog (and Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (1989). https://doi.org/10.1109/69.43410

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 2012. *Logic programming and databases.* Springer Science & Business Media.

Jacques Cohen. 1990. Constraint Logic Programming Languages. *Commun. ACM* (1990). https://doi.org/10.1145/79204.79209

Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proc. Symposium on Cloud Computing (SoCC)*. https://doi.org/10.1145/2391229.2391230

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proc. Symposium on Principles of Programming Languages (POPL)*.

O de Moor, G Gottlob, T Furche, and AJ Sellers (Eds.). 2011. *Datalog Reloaded – First International Workshop, Datalog 2010.* https://doi.org/10.1007/978-3-642-24206-9

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.*

Melvin Fitting. 2002. Fixpoint Semantics for Logic Programming a Survey. *Theoretical Computer Science (TCS)* (2002). https://doi.org/10.1016/S0304-3975(00)00330-3

Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Proc. International Conference on Logic Programming (ICLP/SLP)*.

Michael Gelfond and Vladimir Lifschitz. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* (1991). https://doi.org/10.1007/BF03037169

Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys (CSUR)* (1993). https://doi.org/10.1145/152610.152611

Steve Gregory. 1987. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley.

Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. 2006. codeQuest: Scalable Source Code Queries with Datalog. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/11785477_2

Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, et al. 2014. Demonstration of the Myria big data management service. In *Proc. International Conference on Management of Data.*

Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, Chris Speirs, Tyson Dowd, Ralph Becket, and Mark Brown. 1996. The Mercury language reference manual. (1996).

Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and Emerging Applications: An Interactive Tutorial. In *Proc. Management of Data (SIGMOD)*. https://doi.org/10.1145/1989323.1989456

Joxan Jaffar and Jean-Louis Lassez. 1987. Constraint Logic Programming. In *Proc. Principles of Programming Languages (POPL)*. https://doi.org/10.1145/41625.41635

Joxan Jaffar and Michael J Maher. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming* (1994). https://doi.org/10.1016/0743-1066(94)90033-7

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *International Conference on Computer Aided Verification.*

Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2018. Two concurrent data structures for efficient datalog query processing. In *Proc. Symposium on Principles and Practice of Parallel Programming.*

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *Proc. International Conference on Programming Language Design and Implementation (PLDI)* (2013).

Ross D King. 2004. Applying Inductive Logic Programming to Predicting Gene Function. *AI Magazine* (2004).

Kenneth Kunen. 1987. Negation in Logic Programming. *Journal of Logic Programming* (1987). https://doi.org/10.1016/0743-1066(87)90007-0

Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive Program Analysis as Database Queries. In *Proc. Principles of Database Systems (PODS)*. https://doi.org/10.1145/1065167.1065169

Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming* (2005).

Ninghui Li and John C Mitchell. 2003. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proc. Practical Aspects of Declarative Languages (PADL)*. https://doi.org/10.1007/3-540-36388-2_6

Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* (2009).

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Programming Language Design and Implementation (PLDI)*.

Jack Minker. 1988. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann.

Raymond J Mooney. 1996. Inductive Logic Programming for Natural Language Processing. In *International Conference on Inductive Logic Programming*.

Christos H. Papadimitriou. 1985. A note the expressive power of Prolog. *Bulletin of the European Association for Theoretical Computer Science (EATCS)* (1985).

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. Principles of Programming Languages (POPL)*. https://doi.org/10.1145/199448.199462

Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science (TCS)* (1996). https://doi.org/10.1016/0304-3975(96)00072-2

Max Schäfer and Oege de Moor. 2010. Type inference for datalog with complex type hierarchies. In *Proc. Symposium on Principles of Programming Languages (POPL)*.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proc. International Conference on Compiler Construction (CC)*.

Jiwon Seo, Stephen Guo, and Monica S Lam. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *International Conference Data Engineering (ICDE)*.

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proc. International Conference on Management of Data*.

Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based pre-processing for points-to analysis. In *Proc. International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*.

Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*. https://doi.org/10.1145/1926385.1926390

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proc. Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1925844.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. *Proc. International Conference on Programming Language Design and Implementation (PLDI)*.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming* (1996).

Zoltan Somogyi, Fergus J Henderson, and Thomas Charles Conway. 1995. Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications* (1995).

Souffle Authors. 2018. Souffle. https://souffle-lang.github.io/ [Online; accessed 18-October-2018].

Pavle Subotic, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. (2018).

Jeffrey D Ullman. 1984. *Principles of Database Systems*. Galgotia publications.

Jeffrey D. Ullman. 1988. Principles of Database and Knowledge-Base Systems.

Todd L Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481* (2012).

Andrew K Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and computation* (1994).

David Zook, Emir Pasalic, and Beata Sarna-Starosta. 2009. Typed datalog. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*.

A SUBSTITUTION

$$y[x \mapsto v] = v \;\text{ if } x = y \qquad \text{(S-VAR-1)}$$

$$y[x \mapsto v] = y \;\text{ if } x \neq y \qquad \text{(S-VAR-2)}$$

$$c[x \mapsto v] = c \qquad \text{(S-CST)}$$

$$(\lambda x.\, e)[x \mapsto v] = \lambda x.\, [x \mapsto v] \qquad \text{(S-ABS)}$$

$$(\#\{C_1, \cdots, C_n\})[x \mapsto v] = \#\{C_1[x \mapsto v], \cdots, C_n[x \mapsto v]\} \qquad \text{(S-CONSTRAINT-SET)}$$

$$(e_1\, e_2)[x \mapsto v] = e_1[x \mapsto v]\, e_2[x \mapsto v] \qquad \text{(S-APP)}$$

$$\begin{array}{c}(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2)[x \mapsto v] = \\ \mathsf{let}\, x = e_1[x \mapsto v] \,\mathsf{in}\, e_2[x \mapsto v]\end{array} \qquad \text{(S-LET)}$$

$$(e_1 \texttt{ <+> } e_2)[x \mapsto v] = e_1[x \mapsto v] \texttt{ <+> } e_2[x \mapsto v] \qquad \text{(S-COMPOSE)}$$

$$(\mathsf{solve}\, e)[x \mapsto v] = \mathsf{solve}\, e[x \mapsto v] \qquad \text{(S-SOLVE)}$$

$$(\mathsf{subset}\, e_1\, e_2)[x \mapsto v] = (\mathsf{subset}\, e_1[x \mapsto v]\, e_2[x \mapsto v]) \qquad \text{(S-SUBSET)}$$

$$(\mathsf{project}\, p\, e)[x \mapsto v] = (\mathsf{project}\, p\, e[x \mapsto v]) \qquad \text{(S-PROJECT)}$$

$$(\mathcal{S}(e))[x \mapsto v] = \mathcal{S}(e[x \mapsto v]) \qquad \text{(S-SOLVER)}$$

$$\begin{array}{c}(\mathcal{H}(A, e_1, e_2))[x \mapsto v] = \\ (\mathcal{H}(A[x \mapsto v], e_1[x \mapsto v], e_2[x \mapsto v]))[x \mapsto v]\end{array} \qquad \text{(S-EVAL-HEAD)}$$

$$(\forall(\overline{x_i : \tau_i}).\, (A_0 \overset{e}{\Leftarrow} B_1, \cdots, B_n))[x \mapsto v] = \forall(\overline{x_i : \tau_i}).\, A_0[x \mapsto v] \overset{e[x \mapsto v]}{\Leftarrow} B_1[x \mapsto v], \cdots, B_n[x \mapsto v] \qquad \text{(S-CONSTRAINT)}$$

$$(p(t^h_1, \cdots, t^h_n))[x \mapsto v] = p(t^h_1[x \mapsto v], \cdots, t^h_n[x \mapsto v]) \qquad \text{(S-HEAD-ATOM)}$$

$$(p(t^b_1, \cdots, t^b_n))[x \mapsto v] = p(t^b_1[x \mapsto v], \cdots, t^b_n[x \mapsto v]) \qquad \text{(S-BODY-ATOM-1)}$$

$$\mathsf{not}\,(p(t^b_1, \cdots, t^b_n))[x \mapsto v] = \mathsf{not}\, p(t^b_1[x \mapsto v], \cdots, t^b_n[x \mapsto v]) \qquad \text{(S-BODY-ATOM-2)}$$

Fig. 8. Substitution in λ_{DAT}.

B PROOF OF TYPE SOUNDNESS

B.1 Proof of Progress

THEOREM B.1 (PROGRESS). *Suppose e is a closed, well-typed expression (that is, $\vdash e : \tau$ for some τ), then either e is a value or else there is some expression e' such that $e \longrightarrow e'$.*

PROOF. By induction on the derivation of $e : \tau$.

Case [T-VAR]: Impossible since the expression e is closed.

Case [T-CST]: Immediate since we know that the expression e is a value.

Case [T-ABS]: We know that $e = \lambda x.\, e_1$ which is a value.

Case [T-CONSTRAINT-SET]: We know that $e = \#\{C_1, \cdots, C_n\}$ which is a value.

Case [T-APP]: We know that $e = e_1\, e_2$. By the induction hypothesis either e_1 is a value or it can take a step, and likewise for e_2. If e_1 can take a step, then by [E-CONTEXT] the whole expression e can take a step, and likewise for e_2. Otherwise both e_1 and e_2 must be values and then the canonical forms lemma assures us that e_1 must be of the form $\lambda x.\, e'$. Thus [E-APP] applies, and we can take a step.

Case [T-Let]: We know that $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$. By the induction hypothesis either e_1 is a value or it can take a step. If e_1 can take a step, then by [E-Context] the whole expression e can take a step. Otherwise e_1 must be a value. Thus [E-Let] applies, and we can take a step.

Case [T-Compose]: We know that $e = e_1$ <+> e_2. By the induction hypothesis either e_1 is a value or it can take a step, and likewise for e_2. If e_1 can take a step, then by [E-Context] the whole expression e can take a step, and likewise for e_2. Otherwise both e_1 and e_2 must be values and then the canonical forms lemma assures us that e_1 must be of the form $\#\{C^1_1, \cdots, C^1_n\}$ and e_2 must be of the form $\#\{C^2_1, \cdots, C^2_m\}$. Thus [E-Compose] applies, and we can take a step.

Case [T-Solve]: We know that $e = \mathsf{solve}\, e_1$. By the induction hypothesis either e_1 is a value or it can take a step. If e_1 can take a step, then by [E-Context] the whole expression e can take a step. Otherwise e_1 must be a value and then the canonical forms lemma assures us that it must be of the form $\#\{C_1, \cdots, C_n\}$. Thus [E-Solve] applies, and we can take a step.

Case [T-Subset]: We know that $e = \mathsf{subset}\, e_1\, e_2$. By the induction hypothesis either e_1 is a value or it can take a step, and likewise for e_2. If e_1 can take a step, then by [E-Context] the whole expression e can take a step, and likewise for e_2. Otherwise both e_1 and e_2 must be values and then the canonical forms lemma assures us that e_1 must be of the form $\#\{C^1_1, \cdots, C^1_n\}$ and e_2 must be of the form $\#\{C^2_1, \cdots, C^2_m\}$. Thus either [E-Subset-T] or [E-Subset-F] applies since either $\#\{C^1_1, \cdots, C^1_n\}$ is a subset of $\#\{C^2_1, \cdots, C^2_m\}$ or it is not, and either way we can take a step.

Case [T-Project]: We know that $e = \mathsf{project}\, p\, e_1$. By the induction hypothesis either e_1 is a value or it can take a step. If e_1 can take a step, then by [E-Context] the whole expression e can take a step. Otherwise e_1 must be a value and then the canonical forms lemma assures us that it must be of the form $\#\{C_1, \cdots, C_n\}$. Thus [E-Project] applies, and we can take a step.

Case [T-Select]: We know that $e = \mathcal{S}(e_1)$. By the induction hypothesis either e_1 is a value or it can take a step. If e_1 can take a step, then by [E-Context] the whole expression e can take a step. Otherwise e_1 must be a value and then the canonical forms lemma assures us that it must be of the form $\#\{C_1, \cdots, C_n\}$. If the constraint set is empty then [S-Finish] applies, and we can take a step. If the constraint set is non-empty then both [S-Rule] and [S-Finish] apply, and we can take a non-deterministic step.

Case [T-Extend]: We know that $e = \mathcal{H}(A, e_1, e_2)$. By the induction hypothesis either e_1 is a value or it can take a step, and likewise for e_2. If e_1 can take a step, then by [E-Context] the whole expression e can take a step, and likewise for e_2. We know that $A = p(t^h_1, \cdots, t^h_n)$. By the induction hypothesis, for each i, either t^h_i is a value or it can take a step. If t^h_i can take a step then by [E-Context] the whole expression e can take a step. Otherwise, we have that $A = p(v_1, \cdots, v_n)$ and e_1 and e_2 are values. The canonical forms lemma assures that e_1 is either `true` or `false` and that e_2 is of the form $\#\{C_1, \cdots, C_n\}$. Thus [H-True] or [H-False] applies, and we can take a step.

□

B.2 Proof of Preservation

Theorem B.2 (Preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : \tau'$ *where* $\tau = \tau'$.

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. At each step, we assume that the property holds for all sub-derivations. We proceed by case analysis on the final rule in the derivation.

Case [T-Var]: If the last rule in the derivation is [T-Var], we know that the expression must be a variable, and it cannot take a step, hence the requirements of the theorem are trivially satisfied.

Case [T-Cst]: If the last rule in the derivation is [T-Cst], we know that the expression must be a value, and it cannot take a step, hence the requirements of the theorem are trivially satisfied.

Case [T-Abs]: If the last rule in the derivation is [T-Abs], we know that the expression must be a lambda, and it cannot take a step, hence the requirements of the theorem are trivially satisfied.

Case [T-Constraint-Set]: If the last rule in the derivation is [T-Constraint-Set], we know that the expression must be a constraint set, and it cannot take a step, hence the requirements of the theorem are trivially satisfied.

Case [T-App]: If the last rule in the derivation is [T-App], we know that the expression must be of the form $e_1\, e_2$ for some e_1 and e_2. We must also have sub-derivations with the conclusions $\Gamma \vdash e_1 : \tau_1 \longrightarrow \tau$ and $\Gamma \vdash e_2 : \tau_1'$, and we know that $\tau_1 \cong \tau_1'$. We must prove that if e takes a step it produces an expression e' of type τ. Now, looking at the evaluation contexts and evaluation rules with $e_1\, e_2$ on the left-hand side, we find there are three cases by which $e \longrightarrow e'$:

Subcase [E-App-Ctx-1]: In $e_1\, e_2$, we have that $e_1 \longrightarrow e_1'$ and $\Gamma \vdash e_1 : \tau_1 \longrightarrow \tau$. Applying the induction hypothesis yields: $\Gamma \vdash e_1' : \tau_1 \longrightarrow \tau$. Applying [T-App], we conclude $e_1'\, e_2 : \tau$.
Subcase [E-App-Ctx-2]: Similar.
Subcase [E-App]: If $e \longrightarrow e'$ by [E-App] then we know that $e_1 = \lambda x.\, e_b$, $e_2 = v$, and $e' = e_b[x \mapsto v]$. We must show that $e_b[x \mapsto v] : \tau$. By inversion on the typing of the lambda abstraction, we have that $\Gamma, x : \tau_1 \vdash e_b : \tau$. We also have that $v : \tau_1'$, but $\tau_1 \cong \tau_1'$, hence $v : \tau_1$. We must now show that $e' : \tau$, which holds by the substitution lemma.

Case [T-Let]: If the last rule in the derivation is [T-Let], we know that the expression must be of the form $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ for some e_1 and e_2. We must also have sub-derivations with the conclusions: $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x : \mathsf{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2$. We must prove that if e takes a step it produces an expression e' of type τ_2. Now, looking at the evaluation contexts and evaluation rules with $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ on the left-hand side, we find there are two cases by which $e \longrightarrow e'$:

Subcase [E-Let-Ctx]: In $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$, we have that $e_1 \longrightarrow e_1'$ and $\Gamma \vdash e_1 : \tau_1$. Applying the induction hypothesis yields $\Gamma \vdash e_1' : \tau_1$. Applying [T-Let], we conclude $\mathsf{let}\, x = e_1' \,\mathsf{in}\, e_2 : \tau_2$.
Subcase [E-Let]: If $e \longrightarrow e'$ by [E-Let] then we know that e_1 is a value v and that $e' = e_2[x \mapsto v]$. We must show that $e_2[x \mapsto v] : \tau_2$ which holds by the substitution lemma.

Case [T-Compose]: If the last rule in the derivation is [T-Compose], we know that the expression must be of the form e_1 <+> e_2 for some e_1 and e_2. We must also have sub-derivations with the conclusions $\Gamma \vdash e_1 : r$, $\Gamma \vdash e_1 : r'$, and $r \cong r'$. We must prove that if e takes a step it produces an expression of type r. Now, looking at the evaluation contexts and evaluation rules with e_1 <+> e_2 on the left-hand side, we find there are three cases by which $e \longrightarrow e'$:

Subcase [E-Compose-Ctx-1]: In e_1 <+> e_2, we have that $e_1 \longrightarrow e_1'$ and $\Gamma \vdash e_1 : r$. Applying the induction hypothesis yields: $\Gamma \vdash e_1' : r$. Applying [T-Compose], we conclude e_1' <+> $e_2 : r$.

Subcase [E-Compose-Ctx-2]: Similar.
Subcase [E-Compose]: If $e \longrightarrow e'$ by [E-Compose] then we know that $e_1 = \#\{C_1^1, \cdots, C_n^1\}$, $e_2 = \#\{C_1^2, \cdots, C_m^2\}, \Gamma \vdash e_1 : r, \Gamma \vdash e_2 : r'$, and $r \cong r'$. We must show that $\#\{C_1^1, \cdots, C_n^1, C_1^2, \cdots, C_m^2\}$ has type r. The only rule that applies to constraint sets is [T-Constraint-Set], from which we know that $\forall i.C_i^1 : r_i^1 \land r_i^1 \cong r$ and $\forall i.C_i^2 : r_i^2 \land r_i^2 \cong r'$. By transitivity of $\cong$, $\forall i, r_i^2 \cong r$. By applying [T-Constraint-Set] we get $\#\{C_1^1, \cdots, C_n^1, C_1^2, \cdots, C_m^2\} : r$, as required.

Case [T-Solve]: If the last rule in the derivation is [T-Subset], we know that the expression must be of the form solve e_1 for some e_1. We must also have a sub-derivation with conclusion $\Gamma \vdash e_1 : r$. We must prove that if e takes a step it produces an expression of type r. Now, looking at the evaluation contexts and evaluation rules with solve e_1 on the left-hand side, we find there are two cases by which $e \longrightarrow e'$:

Subcase [E-Solve-Ctx]: In solve e_1, we have that $e_1 \longrightarrow e_1'$ and $\Gamma \vdash e_1 : r$. Applying the induction hypothesis yields: $\Gamma \vdash e_1' : r$. Applying [T-Solve], we conclude solve $e_1' : r$.
Subcase [E-Solve]: If $e \longrightarrow e'$ by [E-Solve] then $e' = \mathcal{S}(\#\{C_1, \cdots, C_n\})$ and we know that $\#\{C_1, \cdots, C_n\} : r$. By [T-Select] we have: $\mathcal{S}(\#\{C_1, \cdots, C_n\}) : r$.

Case [T-Subset]: If the last rule in the derivation is [T-Subset], we know that the expression e must be of the form subset $e_1\, e_2$ for some e_1 and e_2. We must also have sub-derivations with conclusions: $\Gamma \vdash e_1 : r$, $\Gamma \vdash e_2 : r'$, and $r \cong r'$. We must prove that if e takes a step it produces an expression of type `Bool`. Now, looking at the evaluation contexts and evaluation rules with subset $e_1\, e_2$ on the left-hand side, we find there are four cases by which $e \longrightarrow e'$:

Subcase [E-Subset-Ctx-1]: In subset $e_1\, e_2$, we have that $e_1 \longrightarrow e_1'$ and $\Gamma \vdash e_1 : r$. Applying the induction hypothesis yields: $\Gamma \vdash e_1' : r$. Applying [T-Subset], we conclude: subset $e_1'\, e_2$: `Bool`.
Subcase [E-Subset-Ctx-2]: Similar.
Subcase [E-Subset-T]: If $e \longrightarrow e'$ by [E-Subset-T] then $e' =$ `true` and by the canonical forms lemma `true` must have type `Bool`.
Subcase [E-Subset-F]: Similar.

Case [T-Project]: If the last rule in the derivation is [T-Project], we know that the expression e must be of the form project $p\, e_1$ for some e_1. We must also have a sub-derivation with the conclusion: $\Gamma \vdash e_1 : \{p = (\tau_1, \cdots, \tau_n) \mid r\}$. We must prove that if e takes a step to e' then $e' : \{p = (\tau_1, \cdots, \tau_n) \mid r'\}$. Now, looking at the evaluation contexts and evaluation rules with project $p\, e_1$ on the left-hand side, we find there are two cases by which $e \longrightarrow e'$:

Subcase [E-Project-Ctx]: In project $p\, e_1$, we have that $e_1 \longrightarrow e_1'$ and $\Gamma \vdash e_1 : \{p = (\tau_1, \cdots, \tau_n) \mid r\}$. Applying the induction hypothesis yields: $\Gamma \vdash e_1' : \{p = (\tau_1, \cdots, \tau_n) \mid r\}$. Applying [T-Project], we conclude project $p\, e_1' : \{p = (\tau_1, \cdots, \tau_n) \mid r\}$.
Subcase [E-Project]: If $e \longrightarrow e'$ by [E-Project] then $e =$ project $p\, \#\{C_1, \cdots, C_n\}$ and e' is a constraint set containing those constraints $\#\{C_1, \cdots, C_n\}$ that are facts (i.e. have no body) and have p as their predicate. By inversion of the typing $\Gamma \vdash e_1 : \{p = (\tau_1, \cdots, \tau_n) \mid r\}$ and inversion of the [T-Constraint] and [T-Head-Atom] rules, each such constraint must be of the form $p(t_1^h, \dots t_n^h)$ where $\forall i.\Gamma \vdash t_i^h : \tau_i$. Then by [T-Head-Atom] with r instantiated as r' and by [T-Constraint], each of these constraints has type $\{p = (\tau_1, \cdots, \tau_n) \mid r'\}$. Finally, by [T-Constraint-Set], the overall constraint set e' has this type as well, as required.

Case [T-Select]: If the last rule in the derivation is [T-Select], we know that the expression e must be of the form $\mathcal{S}(e_1)$. We must also have a sub-derivation with the conclusion $\Gamma \vdash e_1 : r$. We must prove that if $e : r$ takes a step to e' then $e' : r$. Now, looking at the evaluation contexts and evaluation rules with $\mathcal{S}(e_1)$ on the left-hand side, we find there are three cases to consider:

Subcase [E-Select-Ctx]: In $\mathcal{S}(e_1)$, we have that $e_1 \longrightarrow e_1'$ and $\Gamma \vdash e_1 : r$. Applying the induction hypothesis yields: $\Gamma \vdash e_1' : r$. Applying [T-Select], we conclude: $\Gamma \vdash \mathcal{S}(e_1') : r$.
Subcase [S-Rule]: If $e \longrightarrow e'$ by [S-Rule] then $e_1 = \#\{C_1, \cdots, C_n\}$, $\Gamma \vdash e_1 : r$, and $e' = \mathcal{H}(v(A), v(e), \#\{C_1, \cdots, C_n\})$ where there is some $C_i = \forall(x_1 : \tau_1, \cdots, x_m : \tau_m).A \overset{e}{\Leftarrow} B_1, \cdots, B_n$ and v is a primitive valuation that respects the types τ_i. By inversion of [T-Constraint-Set] C_i has type r' where $r \cong r'$. Then by inversion of [T-Constraint] e must have type Bool and A has type r' where A is the head atom of the constraint C_i.

We want to apply [T-Extend] to get the desired result. To do so, we must show three things: (i) $v(A)$ has type r', (ii) $v(e)$ has type Bool, and (iii) $\#\{C_1, \cdots, C_n\}$ has type r with $r' \cong r$.

Case (i) we know that A has type r, then by the substitution lemma $v(A)$ has type r.
Case (ii) we know that e has type Bool, then by the substitution lemma $v(e)$ has type Bool.
Case (iii) is immediate from the fact that $\#\{C_1, \cdots, C_n\}$ has type r.
Subcase [S-Finish]: If $e \longrightarrow e'$ by [S-Finish] then $e_1 = \#\{C_1, \cdots, C_n\}$, $\Gamma \vdash e_1 : r$, and $e' = \#\{C_1, \cdots, C_n\}$. The former two facts tells us that $\Gamma \vdash \#\{C_1, \cdots, C_n\} : r$ which gives us the required result.

Case [T-Extend]: If the last rule in the derivation is [T-Extend], we know that the expression e must be of the form $\mathcal{H}(A, e_1, e_2)$ of type r for some r, A, e_1, and e_2. We must have sub-derivations with the conclusions: $\Gamma \vdash_c A : r, \Gamma \vdash e_1 :$ Bool$, \Gamma \vdash e_2 : r'$, and $r \cong r'$. We must prove that if e takes a step to e' then $e' : r$. Now, looking at the evaluation contexts and evaluation rules with $\mathcal{H}(A, e_1, e_2)$ on the left-hand side, we find there are five cases to consider:

Subcase [E-Extend-Ctx-1]: In $\mathcal{H}(A, e_1, e_2)$, we have that $e_2 \longrightarrow e_2'$ and $\Gamma \vdash e_2 : r'$. Applying the induction hypothesis yields: $\Gamma \vdash e_2' : r'$. Applying [T-Extend], we conclude: $\Gamma \vdash \mathcal{H}(A, e_1, e_2') : r$.
Subcase [E-Extend-Ctx-2]: Similar.
Subcase [E-Extend-Ctx-3]: For some i, we have that:

$$\mathcal{H}(p(t_1, \cdots, t_i, \cdots, t_n), e_1, e_2) \longrightarrow \mathcal{H}(p(t_1, \cdots, t_i', \cdots, t_n), e_1, e_2)$$

We know that $\Gamma \vdash t_i : \tau_i$. Applying the induction hypothesis yields $\Gamma \vdash t_i' : \tau_i$. By the hypotheses of e and by applying [T-Extend], we conclude $\Gamma \vdash \mathcal{H}(A, e_1, e_2') : r$.
Subcase [H-True]: If $e \longrightarrow e'$ by [H-True] then we know that $e_2 = \#\{C_1, \cdots, C_n\}$ and $e' = \mathcal{S}(\#\{A, C_1, \cdots, C_n\})$. By inversion of $\Gamma \vdash e_2 : r'$, every $\Gamma \vdash_c C_i : r_i \cong r'$. Moreover, $\Gamma \vdash_c A : r$. Since $r \cong r'$ all of the r_i's are congruent to r. By [T-Constraint-Set], $\#\{A, C_1, \cdots, C_n\}$ has type r, and by [T-Select] e' has type r.
Subcase [H-False]: If $e \longrightarrow e'$ by [H-False] then we know that $e' = \mathcal{S}(e_2)$. We also know that $e_2 : r$. Applying [T-Select], we conclude that $e' : r$.

□

B.3 Generalization

Lemma B.3 (Generalization). *If* $\Gamma, x : \sigma' \vdash e : \tau$ *and* $\sigma \sqsubseteq \sigma'$, *then* $\Gamma, x : \sigma \vdash e : \tau$.

Proof. The proof is by induction on the derivation of $\Gamma, x : \sigma' \vdash e : \tau$. All cases except [T-Var] and [T-Let] are resolved directly with the induction hypothesis. In the [T-Var] case, we know that $\Gamma, x : \sigma \vdash x : \tau'$ for some τ' such that $\sigma' \sqsubseteq \tau'$. We need to show that $\Gamma, x : \sigma' \vdash x : \tau'$. This follows using the [T-Let] rule and the fact that $\sigma \sqsubseteq \sigma' \sqsubseteq \tau'$. In the [T-Let] case, we know that

$\Gamma, x : \sigma' \vdash e_1 : \tau_1$ and $\Gamma, x : \sigma', x' : \text{gen}(\Gamma, x : \sigma', \tau_1) \vdash e_2 : \tau$ for some e_1, e_2, τ_1. Since $\sigma \sqsubseteq \sigma'$, it is also the case that $\text{gen}(\Gamma, x : \sigma, \tau_1) \sqsubseteq \text{gen}(\Gamma, x : \sigma', \tau_1)$. We apply the induction hypothesis to infer $\Gamma, x : \sigma \vdash e_1 : \tau_1$. We then apply the induction hypothesis twice, on both $x : \sigma'$ and $x' : \text{gen}(\Gamma, x : \sigma', \tau_1)$, to infer $\Gamma, x : \sigma, x' : \text{gen}(\Gamma, x : \sigma, \tau_1) \vdash e_2 : \tau$. Then by [T-Let], we infer the required fact $\Gamma, x : \sigma \vdash e : \tau$. □

B.4 Substitution

Theorem B.4 (Substitution). *The typing judgments* $\vdash$, $\vdash_c$, *and* $\vdash_p$ *satisfy the implications:*

- *If* $\Gamma, x_0 : \forall\overline{\alpha}, \forall\overline{\rho}.\tau_0 \vdash e : \tau$, *and* $x_0 \notin dom(\Gamma)$, *and* $\Gamma \vdash v_0 : \tau_0$, *and* $(\overline{\alpha} \cup \overline{\rho}) \cap ftv(\Gamma) = \emptyset$, *then* $\Gamma \vdash e[x_0 \mapsto v_0] : \tau$.
- *If* $\Gamma, x_0 : \forall\overline{\alpha}, \forall\overline{\rho}.\tau_0 \vdash_c C : r$, *and* $x_0 \notin dom(\Gamma)$, *and* $\Gamma \vdash_c v_0 : \tau_0$, *and* $(\overline{\alpha} \cup \overline{\rho}) \cap ftv(\Gamma) = \emptyset$, *then* $\Gamma \vdash_c C[x_0 \mapsto v_0] : r$.
- *If* $\Gamma, x_0 : \forall\overline{\alpha}, \forall\overline{\rho}.\tau_0 \vdash_p A : r$, *and* $x_0 \notin dom(\Gamma)$, *and* $\Gamma \vdash_p v_0 : \tau_0$, *and* $(\overline{\alpha} \cup \overline{\rho}) \cap ftv(\Gamma) = \emptyset$, *then* $\Gamma \vdash_p A[x_0 \mapsto v_0] : r$.

Proof. By mutual induction on the derivation of the premises:

- $\Gamma, x_0 : \forall\overline{\alpha}, \forall\overline{\rho}.\tau_0 \vdash e : \tau$,
- $\Gamma, x_0 : \forall\overline{\alpha}, \forall\overline{\rho}.\tau_0 \vdash_c C : r$
- $\Gamma, x_0 : \forall\overline{\alpha}, \forall\overline{\rho}.\tau_0 \vdash_p A : r$

We proceed by cases on the final typing rule used in the typing derivation.

Case [T-Cst]: Immediate from the definition of (S-Cst).

Case [T-Constraint-Set]: By the premises of the rule, we know that $e = \#\{C_1, \cdots, C_n\}$, and $\tau = r$, and:

$$\forall i.(\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash_c C_i : r_i \wedge r_i \cong r) \tag{1}$$

Using the induction hypothesis on (1) yields:

$$\forall i.(\Gamma \vdash_c C_i[x_0 \mapsto v_0] : r_i \wedge r_i \cong r) \tag{2}$$

By the definition of substitution, the expression:

$$\#\{C_1[x_0 \mapsto v_0], \cdots, C_n[x_0 \mapsto v_0]\}$$

is equivalent to:

$$(\#\{C_1, \cdots, C_n\})[x_0 \mapsto v_0]$$

By (2) and [T-Constraint-Set] the type of this expression is r, as required.

Case [T-Var]: This case can be proved in the same way as in [Wright and Felleisen, 1994] (see Lemma 4.4).

Case [T-Abs]: This case can be proved in the same way as in [Wright and Felleisen, 1994] (see Lemma 4.4).

Case [T-App]: By the premises of the rule, we know that: $e = e_1\, e_2$ and:

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_1 : \tau' \longrightarrow \tau \tag{3}$$

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_2 : \tau' \tag{4}$$

Using the induction hypothesis on (3) and (4) yields:

$$\Gamma \vdash e_1[x_0 \mapsto v_0] : \tau' \longrightarrow \tau \tag{5}$$

$$\Gamma \vdash e_2[x_0 \mapsto v_0] : \tau' \tag{6}$$

By the definition of substitution, the expression:

$$(e_1[x_0 \mapsto v_0])\,(e_2[x_0 \mapsto v_0])$$

is equivalent to:

$$(e_1\,e_2)[x_0 \mapsto v_0]$$

By (5) and (6) and [T-App], the type of this expression is τ, as required.

Case [T-Let]: This case can be proved in the same way as in [Wright and Felleisen, 1994] (see Lemma 4.4).

Case [T-Compose]: By the premises of the rule, we know that: $e = e_1$ <+> e_2, and $r \cong r'$, and:

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_1 : r \tag{7}$$

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_2 : r' \tag{8}$$

Using the induction hypothesis on (7) and (8) yields:

$$\Gamma \vdash e_1[x_0 \mapsto v_0] : r \tag{9}$$

$$\Gamma \vdash e_2[x_0 \mapsto v_0] : r' \tag{10}$$

By the definition of substitution, the expression:

$$(e_1[x_0 \mapsto v_0]) \texttt{<+>} (e_2[x_0 \mapsto v_0])$$

is equivalent to:

$$(e_1 \texttt{<+>} e_2)[x_0 \mapsto v_0]$$

By (9) and (10) and [T-Compose], the type of this expression is r.

Case [T-Solve]: By the premises of the rule, we know that $e = \mathsf{solve}\, e_1$ and:

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_1 : r \tag{11}$$

Using the induction hypothesis on (11) yields:

$$\Gamma \vdash e_1[x_0 \mapsto v_0] : r \tag{12}$$

By the definition of substitution, the expression:

$$\mathsf{solve}\,(e_1[x_0 \mapsto v_0])$$

is equivalent to:

$$(\mathsf{solve}\,(e_1))[x_0 \mapsto v_0]$$

By (12) and [T-Solve], the type of this expression is r, as required.

Case [T-Subset]: By the premises of the rule, we know that: $e = \mathsf{subset}\, e_1\, e_2$, $r \cong r'$, $\tau = \texttt{Bool}$, and:

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_1 : r \tag{13}$$

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_2 : r' \tag{14}$$

Using the induction hypothesis on (13) and (14) yields:

$$\Gamma \vdash e_1[x_0 \mapsto v_0] : r \tag{15}$$

$$\Gamma \vdash e_2[x_0 \mapsto v_0] : r' \tag{16}$$

By the definition of substitution, the expression:

$$\mathsf{subset}\,(e_1[x_0 \mapsto v_0])\,(e_2[x_0 \mapsto v_0])$$

is equivalent to:

$$(\mathsf{subset}\, e_1\, e_2)[x_0 \mapsto v_0]$$

By (15) and (16) and [T-Subset], the type of this expression is `Bool`, as required.

Case [T-Project]: By the premises of the rule, we know that $e = \mathsf{project}\, p\, e_1$ and:

$$\tau = \{p = (\tau_1, \cdots, \tau_n) \mid r'\}$$

and

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_1 : \{p = (\tau_1, \cdots, \tau_n) \mid r\} \tag{17}$$

Using the induction hypothesis on (17) yields:

$$\Gamma \vdash e_1[x_0 \mapsto v_0] : \{p = (\tau_1, \cdots, \tau_n) \mid r\} \tag{18}$$

By the definition of substitution, the expression:

$$\mathsf{project}\, p\, (e_1[x_0 \mapsto v_0])$$

is equivalent to:

$$(\mathsf{project}\, p\, e_1)[x_0 \mapsto v_0]$$

By (18) and [T-Project], the type of this expression is $\{p = (\tau_1, \cdots, \tau_n) \mid r'\}$, as required.

Case [T-Select]: By the premises of the rule, we know that $e = \mathcal{S}(e_1)$, and $\tau = r'$ and:

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_1 : r \tag{19}$$

Using the induction hypothesis on (19) yields:

$$\Gamma \vdash e_1[x_0 \mapsto v_0] : r \tag{20}$$

By the definition of substitution, the expression:

$$\mathcal{S}(e_1[x_0 \mapsto v_0])$$

is equivalent to:

$$(\mathcal{S}(e_1))[x_0 \mapsto v_0]$$

By (20) and [T-Select], the type of this expression is r, as required.

Case [T-Extend]: By the premises of the rule, we know that $e = \mathcal{H}(A, e_1, e_2)$ and:

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash_c A : r \tag{21}$$
$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_1 : \mathtt{Bool} \tag{22}$$
$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_2 : r' \tag{23}$$
$$r \cong r' \tag{24}$$

Using the induction hypothesis on (21), (22), and (23) yields:

$$\Gamma \vdash_c A[x_0 \mapsto v_0] : r \tag{25}$$
$$\Gamma \vdash e_1[x_0 \mapsto v_0] : \mathtt{Bool} \tag{26}$$
$$\Gamma \vdash e_2[x_0 \mapsto v_0] : r' \tag{27}$$

By the definition of substitution, the expression:

$$\mathcal{H}(A[x_0 \mapsto v_0], e_1[x_0 \mapsto v_0], e_2[x_0 \mapsto v_0])$$

is equivalent to:

$$(\mathcal{H}(A, e_1, e_2))[x_0 \mapsto v_0]$$

By (24), (25), (26), (27), and [T-Extend], the type of this expression is r, as required.

Case [T-Constraint]: By the premises of the rule, we know that:

$$e = \forall (x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0 \overset{e_g}{\Leftarrow} B_1, \cdots, B_n$$

and:

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p A_0 : r \tag{28}$$

$$\forall i.\, \Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p B_i : r_i \tag{29}$$

$$\Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash e_g : \texttt{Bool} \tag{30}$$

$$\forall i.\, r \cong r_i \tag{31}$$

By the convention that we can always rename variables, we may assume that $\forall i > 0.\, x_0 \neq x_i$. Applying the permutation lemmas for $\vdash_p$ and $\vdash$, we may rearrange:

$$\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash_p A_0 : r \tag{32}$$

$$\forall i.\, \Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash_p B_i : r_i \tag{33}$$

$$\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash e_g : \texttt{Bool} \tag{34}$$

Applying the induction hypotheses for $\vdash_p$ and $\vdash$ on (32), (33), and (34), we obtain:

$$\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p A_0[x_0 \mapsto v_0] : r \tag{35}$$

$$\forall i.\, \Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p B_i[x_0 \mapsto v_0] : r_i \tag{36}$$

$$\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash e_g[x_0 \mapsto v_0] : \texttt{Bool} \tag{37}$$

By the definition of substitution, the expression:

$$\forall (x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0[x_0 \mapsto v_0] \overset{e_g[x_0 \mapsto v_0]}{\Leftarrow} B_1[x_0 \mapsto v_0], \cdots, B_n[x_0 \mapsto v_0]$$

is equivalent to:

$$(\forall (x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0 \overset{e_g}{\Leftarrow} B_1, \cdots, B_n)[x_0 \mapsto v_0]$$

By (31), (35), (36), (37), and [T-Constraint], the type of this expression is r, as required.

Case [T-Head-Atom]: By the premises of the rule, we know that $A = p(t^h_1, \cdots, t^h_n)$, and $\tau = \{p = (\tau_1, \cdots, \tau_n) \mid r\}$, and:

$$\forall i.\, \Gamma, x_0 : \overline{\alpha}, \overline{\rho}.\tau_0 \vdash t^h_i : \tau_i \tag{38}$$

Using the induction hypothesis on (38) yields:

$$\forall i.\, \Gamma \vdash t^h_i[x_0 \mapsto v_0] : \tau_i \tag{39}$$

By the definition of substitution, the head atom:

$$p(t^h_1[x_0 \mapsto v_0], \cdots, t^h_n[x_0 \mapsto v_0])$$

is equivalent to:

$$(p(t^h_1, \cdots, t^h_n))[x_0 \mapsto v_0]$$

By (39) and [T-Head-Atom] the type of this head atom is $\{p = (\tau_1, \cdots, \tau_n) \mid r\}$, as required.

Case [T-Body-Atom-1]: Similar.

Case [T-Body-Atom-2]: Similar. □

Lemma B.5 (Permutation). *We prove the following implications for the three typing judgments:*

- *If $\Gamma \vdash e : \tau$ and Δ is a permutation of Γ, then $\Delta \vdash e : \tau$.*
- *If $\Gamma \vdash_c C : r$ and Δ is a permutation of Γ, then $\Delta \vdash_c C : r$.*
- *If $\Gamma \vdash_p A : r$ and Δ is a permutation of Γ, then $\Delta \vdash_p A : r$.*

Proof. Straightforward mutual induction on typing derivations. □

Lemma B.6 (Weakening). *We prove the following implications for the three typing judgments:*

- *If* $\Gamma \vdash e : \tau$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : \tau' \vdash e : \tau$.
- *If* $\Gamma \vdash_c C : r$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : \tau' \vdash C : r$.
- *If* $\Gamma \vdash_p A : r$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : \tau' \vdash A : r$.

Proof. Straightforward mutual induction on typing derivations. □

Lemma B.7 (Type Substitution). *If* $\Gamma \vdash e : \tau$ *and* S *is type substitution then* $S(\Gamma) \vdash e : S(\tau)$.

Proof. Straightforward mutual induction on typing derivations. □

B.5 Canonical Forms

Lemma B.8 (Canonical-Forms). *The lemma tells us the shape of values of certain types:*

(i) If v *is a value of type* `Bool`*, then* $v = $ `true` *or* $v = $ `false`.
(ii) If v *is a value of type* $\tau_1 \longrightarrow \tau_2$*, then* $v = \lambda x : \tau_1\, e$.
(iii) If v *is a value of type* $\{\}$ *or* $\{\, p = (\tau_1, \cdots, \tau_n) \mid r\,\}$ *then* v *is of the form* $\#\{C_1, \cdots, C_n\}$.

Proof. There are 3 type rules that apply to values: [T-Cst], [T-Abs] and [T-Constraint-Set].

- For part (i), [T-Abs] and [T-Constraint-Set] cannot apply because the primitive type `Bool` is not a function type nor a row type. Thus, [T-Cst] requires the value to be a boolean.
- For part (ii), [T-Cst] and [T-Constraint-Set] cannot apply because an arrow type is not a primitive type nor a row type. Thus, [T-Abs] requires the value to be lambda abstraction.
- For part (iii), [T-Cst] and [T-Abs] cannot apply because a row type is not a primitive type nor a function type. Thus, [T-Constraint-Set] requires the value to be a constraint set.

□

B.6 ≅ is an Equivalence Relation

Theorem B.9 (≅ is an Equivalence Relation). *We must show that the relation* $\cong$ *is reflexive* ($\tau \cong \tau$)*, symmetric* $\tau_1 \cong \tau_2 \Leftrightarrow \tau_2 \cong \tau_1$*, and transitive* $\tau_1 \cong \tau_2 \wedge \tau_2 \cong \tau_3 \Longrightarrow \tau_1 \cong \tau_3$.

Lemma B.10. *The relation* $\cong$ *is reflexive.*

Proof. Immediate from [Eq-Refl]. □

Lemma B.11. *The relation* $\cong$ *is symmetric.*

Proof. By induction on the derivation rules.

Case [Eq-Refl]: Immediate.

Case [Eq-Arrow]: We know that $\tau_1 \cong \tau_1'$ and $\tau_2 \cong \tau_2'$. For symmetry, we must show that $\tau_1' \longrightarrow \tau_2' \cong \tau_1 \longrightarrow \tau_2$. By the induction hypothesis we know that $\tau_1' \cong \tau_1$ and $\tau_2' \cong \tau_2$. Applying [Eq-Arrow] yields the desired result.

Case [Eq-Tuple]: We know that $\forall i.\ \tau_i \cong \tau_i'$. For symmetry, we must show that $(\tau_1', \cdots, \tau_n') \cong (\tau_1, \cdots, \tau_n)$. By the induction hypothesis we know that $\forall i.\ \tau_i' \cong \tau_i$. Applying [Eq-Tuple] yields the desired result.

Case [Eq-Trans]: We know that $\tau_1 \cong \tau_2$ and $\tau_2 \cong \tau_3$. For symmetry, we must show that $\tau_3 \cong \tau_1$. By the induction hypothesis we know that $\tau_2 \cong \tau_1$ and $\tau_3 \cong \tau_2$. Applying [Eq-Trans] yields the desired result.

Case [Eq-Head]: We know that $\tau \cong \tau'$ and $r \cong s$. For symmetry, we must show that $(\!| p : \tau' \mid s |\!) \cong (\!| p : \tau \mid r |\!)$. By the induction hypothesis we know $\tau' \cong \tau$ and $s \cong r$. Applying [Eq-Head] yields the desired result.

Case [Eq-Swap]: Immediate. □

Lemma B.12. *The relation $\cong$ is transitive.*

Proof. Immediate from [Eq-Trans]. □