Separate Compilation and Partial Linking: Modules for Datalog IR

David Klopp
JGU Mainz
Germany

André Pacak
JGU Mainz
Germany

Sebastian Erdweg
JGU Mainz
Germany

Abstract

In recent years, Datalog has sparked renewed interest in academia and industry, leading to the development of numerous new Datalog systems. To unify these systems, recent approaches treat Datalog as an intermediate representation (IR) in a compiler framework: Compiler frontends can lower different Datalog dialects to the same IR, which is then optimized before a compiler backend targets one of many existing Datalog engines. However, a key feature is missing in these compiler frameworks: an expressive module system.

In this paper, we present the first module system for a Datalog IR. Our modules are statically typed, can be separately compiled, and partially linked to form "bundles". Since IR modules are generated by a compiler frontend, we rely on explicit declarations of required and provided relations to maximize the decoupling between modules. This also allows modules to abstract over required relations to offer reusable functionality (e.g., computing a transitive closure) that can be instantiated for different relations in a single Datalog program. We formalize the module system, its type system, and the linking algorithm. We then describe how different usage patterns that occur in Datalog dialects (e.g., inheritance, cyclic imports) can be expressed in our IR module system. Finally, we integrate our module system into an existing Datalog compiler framework, develop a Soufflé compiler frontend that translates Soufflé components to IR modules, and demonstrate its applicability to a large Doop analysis.

CCS Concepts: • **Software and its engineering → Reusability**; **Modules / packages**; • **Theory of computation → Constraint and logic programming**.

Keywords: Datalog, module system

GPCE '24, October 21–22, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1211-1/24/10
https://doi.org/10.1145/3689484.3690737

ACM Reference Format:
David Klopp, André Pacak, and Sebastian Erdweg. 2024. Separate Compilation and Partial Linking: Modules for Datalog IR. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24), October 21–22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3689484.3690737

1 Introduction

Datalog has gained significant attention from both academia and industry in recent years. While Datalog was originally designed as a database query language, it is nowadays used for large and complex tasks, such as program analysis [8, 23], network monitoring [1], and distributed computing [2–4]. This revival of Datalog has led to various Datalog systems that improve the programmability, performance, and expressiveness of Datalog [4, 6, 7, 11, 18]. However, each system offers its unique flavor of Datalog, preventing interoperability between them. For example, Soufflé [11] extends Datalog with algebraic data types and provides a high-performance C++ evaluation engine. Formulog [7] extends Datalog with functions and SMT constraints, and it provides its own execution engine. Ascent [21] provides a macro-based library to integrate Datalog into Rust. bddbddb [24] is a simplistic Datalog dialect, that uses binary decision diagrams to evaluate programs. Unfortunately, a program written in bddbddb cannot run in Soufflé, and a Soufflé program cannot be executed using Ascent.

To target this fragmentation, a unifying framework is needed that treats Datalog as an intermediate representation (IR) instead of a programming language. The IncA framework [13] offers such a compiler framework.[1] It includes frontends for Soufflé, bddbddb [24], as well as a functional logic language [18]. These Datalog dialects are translated into a common, typed intermediate Datalog representation that can be optimized. Afterward, a compiler backend handles the execution of the IR code, usually by translation to an existing Datalog system. Although this compiler framework helps to unify Datalog compilers and supports cross-compilation, it does not yet allow for proper interoperability between program fragments written in different dialects.

The main issue is the lack of a module system for the Datalog IR. The goal of this paper is to provide a module system that is expressive enough to be a compilation target

[1] https://gitlab.rlp.net/plmz/inca-scala

for different Datalog dialects and to enable interoperability between them. This is in contrast to existing user-facing module systems that must consider programmability by a user in their design. Our module system must address the following challenges:

Separate compilation. The module system must organize Datalog definitions into modules such that separate checking and compilation is possible. In particular, it must be possible to lower and optimize modules separately.

Interoperability. Currently, there is no way to use IR code from different dialects in a single program. For example, IR code generated from Soufflé and IR code generated from a functional logic language are isolated instances that cannot interact. A module system must allow modules to interoperate, no matter the frontend that generated them.

Open modules. We need a module system that is expressive enough to embody the component architecture used by existing Datalog dialects. For example, Soufflé supports inheritance between components and different scoping levels. This necessitates the system to support open modules, which formulate external requirements.

Partial linking. A linker weaves separately compiled modules into a larger module. We follow classic module system formulations [9, 12] in allowing partial linking, where linked modules may still contain requirements to be resolved later. However, while these prior works describe linking separately from the module definitions, this amounts to build scripts in practice, which hurts program understandability. Instead, the module system should support linking as a language-integrated feature, such as import statements.

Compositional validity. When partially linking a module stepwise, we need to ensure that the module remains valid after each step. The linking of well-typed modules must yield a well-typed module, given that the requirements are met appropriately.

For example, consider we want to implement an abstract interpreter for Java in IncA's functional Datalog language. But to resolve variables, we want to reuse the configurable points-to analysis implemented by Doop [8] in Soufflé's Datalog language [11]. Figure 1 illustrates how our module system supports such interaction:

1. We compile the Soufflé modules separately to IR modules.
2. We use partial linking to bundle the compiled Soufflé modules into a single big IR module, which still contains some requirements. In our example, the Doop analysis uses this feature to allow client-side configuration of its context sensitivity.
3. We separately compile the Functional IncA modules, which import the bundled IR module generated from Soufflé code. Here, interoperability takes place.

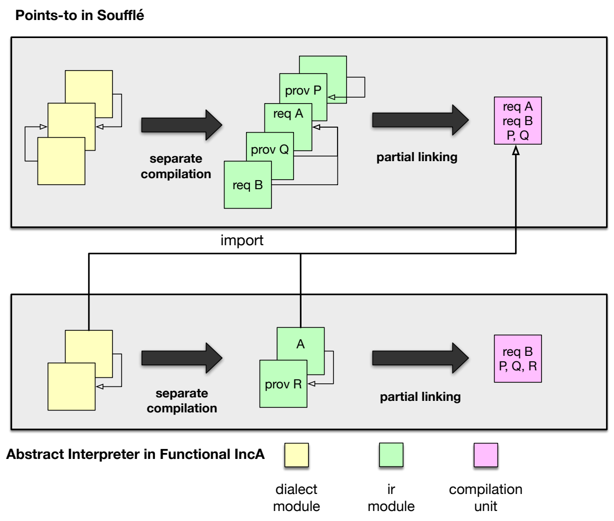

Figure 1. Separate compilation and partial linking across different Datalog dialects.

4. Finally, we use partial linking again, which resolves requirement A of the bundled IR module but leaves requirement B open. Since the Functional IncA modules import the bundled module, linking generates a new bundled IR module that contains all previous definitions.

In this paper, we present a module system that solves the above challenges for a Datalog IR. Besides definitions, modules can contain three new declarations. A declaration `require A : T` indicates that a module needs a Datalog relation `A` with signature `T`. The requirement must be resolved when linking the surrounding module. A declaration `provide B : T` indicates that a module exports a Datalog relation `B` with signature `T`. Relation `B` must be defined in the surrounding module. And finally, a declaration `import M as q with {A = R}` triggers linking: All provided declarations of `M` become available with prefix `q` in the surrounding module. However, such an import is only valid when providing bindings for all requirements of `M`. We formalize the syntax, the static semantics, and the linking of our module system.

We also study how common usage patterns of source-language module systems can be encoded in our IR module system. In particular, we investigate how cyclic imports, inheritance, and nested modules can be represented in our module system. This demonstrates the expressiveness of the module system. We have implemented the module system by extending the IncA compiler framework for Datalog. The implementation checks modules, compiles them separately, and links them in accordance with the formalization. Finally, we have added support for compiling Soufflé and its complex components to Datalog IR modules, which enabled us to do separate compilation, partial linking, interoperability with an existing Doop analysis as indicated in Figure 1.

```
module Graph
declare edge: Int × Int
provide edge: Int × Int
edge(1,2). edge(2,3). edge(3,4).

module Path
require E: Int × Int
declare path: Int × Int
provide path: Int × Int
path(x,y) :- E(x,y).
path(x,y) :- E(x,z), path(z,y).
```

Figure 2. Datalog modules that can be separately compiled.

In summary, we present the following contributions:

- We design a module system for Datalog IR that supports separate compilation, partial linking and allows modules to abstract over relations (Section 2).
- We formalize the static semantics and linking process of the module system (Section 3).
- We describe how complex usage patterns can be encoded in our module system (Section 4).
- We implement our module system in the IncA compiler framework. Finally, we showcase its applicability with two case studies: first, we compile a Doop analysis written in Soufflé into multiple Datalog IR modules that we link to a bundle, and second, we use a Soufflé points-to analysis in Functional IncA to implement an escape analysis (Section 5).

2 Modular Logic Programming in Datalog

Datalog programs define relations through inference rules. Datalog rules take the form $R(t) :- a_1, \ldots, a_n$, where $R(t)$ is called the head of the rule and the atoms a_i make up the body of the rule. Semantically, relation R contains a tuple (t) if all atoms a_i hold and all free variables in term t are bound by the atoms. Rules without atoms describe facts that hold unconditionally. A Datalog program is a collection of rules.

In this section, we introduce the constructs of our module system by example and highlight a few key design aspects.

2.1 Open Datalog Modules and Separate Compilation

We enrich Datalog with notation for modules as illustrated in Figure 2. Each module has a name and body, which consists of entries. Module `Graph` in Figure 2 has 5 module entries. First, the module declares a relation `edge` with its type.[2] Second, the module marks `edge` as provided, meaning that other modules will be able to query `edge`. Third, the module defines `edge` through three Datalog rules, providing three facts.

Based on this graph, we want to compute the transitive closure of `edge` in a relation `path`. Without modules, `edge` and `path`

[2]We consider explicitly typed Datalog programs so that we can validate them during separate compilation. Most Datalog systems used in practice have type annotations or types can be easily inferred.

```
module App
import Graph as g
import Path as p with { E=g.edge }
declare cyclic: Int
cyclic(x) :- p.path(x,x).
```

(a) Before linking

```
module App
declare g.edge: Int × Int
g.edge(1,2).  g.edge(2,3).  g.edge(3,4).
declare p.path: Int × Int
p.path(x,y) :- g.edge(x,y).
p.path(x,y) :- g.edge(x,z), p.path(z,y).
declare cyclic: Int
cyclic(x) :- p.path(x,x).
```

(b) After linking

Figure 3. Imports lead to linking, which yields a new module with inlined dependencies.

have to be defined together and compiled together. Instead, we can use modules to isolate their definitions as shown in Figure 2. Module `Path` requires a relation `E` with its expected type. Requirements are not imports: They are not resolved right away but remain abstract until linking. However, the requirement provides sufficient information that the remainder of the `Path` module can be defined and validated. Here, module `Path` defines a relation `path` that computes the transitive closure over the abstract `E`. We call modules like `Path` that have unresolved requirements *open modules*. In contrast, `Graph` is a *closed module*.

Modules `Path` and `Graph` can be defined in isolation, validated in isolation, and compiled in isolation. For example, we would detect if `path` used `E` inconsistently, and an optimizing Datalog compiler could select a specialized backing data structure for the transitive closure computation implemented by `path`, which can improve performance considerably [20]. Moreover, the module system makes Datalog more expressive: We effectively defined a parametric transitive closure relation over `E`, which is not possible in standard Datalog.

2.2 Language-Integrated Linking

Datalog modules need to be linked before the code can be executed. In our module system, linking is triggered through import declarations that occur in a module. This is in contrast to prior approaches in theory [9, 12] and practice (e.g., the C language), where separate compilation requires build scripts to enumerate sets of modules to be linked. Build scripts are known to be burdensome for software maintenance [15], which is why we rather include imports.

Figure 3 shows a linking example. Module `App` imports the two modules `Graph` and `Path` we defined above in Figure 2. Each import declares the name of the imported module as

```
module SubgraphPath
import Graph as g
require inSubgraph: Int
declare subEdge: Int × Int
subEdge(x,y) :- g.edge(x,y),inSubgraph(x),inSubgraph(y).
import Path as p with { E=subEdge }
provide p.path: Int × Int
```

(a) Before linking

```
module SubgraphPath
declare g.edge: Int × Int
g.edge(1,2).  g.edge(2,3).  g.edge(3,4).
require inSubgraph: Int
declare subEdge: Int × Int
subEdge(x,y) :- g.edge(x,y),inSubgraph(x),inSubgraph(y).
declare p.path: Int × Int
p.path(x,y) :- subEdge(x,y).
p.path(x,y) :- subEdge(x,z), p.path(z,y).
provide p.path: Int × Int
```

(b) After linking

Figure 4. Imports lead to linking, which yields a new module with inlined dependencies.

well as a local name used to qualify the relations of the imported module. Since `Graph` is a closed module, we can import it as is. Through the import, all provided relations of `Graph` become available locally, but with prefix `g` as a qualifier. For `Graph`, we only obtain one relation: `g.edge`.

The import of `Path` is more complex, because `Path` is an open module. We designed our module system to enforce that every import provides bindings for all requirements of the imported module. Module `Path` has one required relation `E`, which must be assigned as part of the import declaration. Indeed, we bind `E=g.edge`, thus linking `path` to `g.edge`. We can then use `p.path` locally, since it was provided by `Path`. In our example, we define a new relation `cyclic` that collects all nodes that have a path to themselves.

The compilation of a module with imports triggers linking. Linking can be thought of as a preprocessor step, where imported modules are being inlined and relations are renamed accordingly. Indeed, it is possible to import the same module multiple times, for example, to compute the transitive closure `path` over different graphs. Crucially, the result of linking is yet again a module, which can later be linked with other modules compositionally. We show the result of linking for our example at the bottom of Figure 3. While relations `edge` and `path` have been renamed to include the local module name as a prefix, requirements have been eliminated and substituted according to the bindings of the import declaration.

2.3 Partial and Compositional Linking

In the previous example, we used linking to build an executable application. However, linking can also be used to form what we call *bundles*. A bundle combines multiple modules into a single unit that can be used as a library in other modules. To retain expressivity, bundles should still be able to abstract over other relations. And bundles should be able to selectively re-export definitions. Our module system supports such bundles, even though it does not have a dedicated language feature for it.

Consider the example in Figure 4. We define a module `SubgraphPath` that computes the transitive closure over a subgraph. The exact subgraph is not known but will be provided by the required `inSubgraph` relation. We then define a relation `subEdge`, which selects those edges, where both nodes are in the subgraph. Only then we import the `Path` module, using the `subEdge` relation to bind `E`. Finally, the `SubgraphPath` module re-provides `p.path` to its clients.

The linked `SubgraphPath` module illustrates bundling. While the linking process is the same as before, the resulting module has two novel aspects. First, the linked module contains a requirement `inSubgraph`, which means we have an open module after linking. Second, the linked module contains a provide declaration, which means we can import the module elsewhere to use the provided relation. Both aspects clearly indicate: a bundle is a library, not an executable application.

Two features of our module system enable bundling: partial and compositional linking. Partial linking occurs when imports depend on unresolved requirements, such as the import of `Path` in `SubgraphPath`, which depends on `inSubgraph`. We say partial linking because the linker leaves these requirements unresolved. However, this is only useful if we can later combine the result of partial linking with another module to produce a closed module eventually. To this end, we rely on compositional linking: The result of linking is a regular module, which can be imported elsewhere. For example, we can use the `SubgraphPath` bundle in another module:

```
module App
declare subgraph: Int
subgraph(1). subgraph(2).
import SubgraphPath as sp { inSubgraph = subgraph }
```

This module is closed since it does not contain any more requirements. Moreover, we can statically validate that the module is executable.

2.4 Compositional Validation

Each module has an implicit module interface that we infer from the module's definition. The module interface comprises the required and provided relations with their types. Local definitions are not part of the interface. For example, the modules from above have the following interfaces:

$$\begin{array}{rcl}\texttt{Graph} & : & \langle \varepsilon \mid edge : Int \times Int \rangle \\ \texttt{Path} & : & \langle E : Int \times Int \mid path : Int \times Int \rangle \\ \texttt{SubgraphPath} & : & \langle inSubgraph : Int \mid p.path : Int \times Int \rangle\end{array}$$

In the module interface $\langle R, P \rangle$, we first list the required relations R and then the provided relations P. As we will see

$$
\begin{array}{lll}
\text{(rules)} & r & ::= p(\overline{X}) :- \overline{a}. \\
\text{(atoms)} & a & ::= p(\overline{t}) \mid t = t \mid t \neq t \\
\text{(terms)} & t & ::= X \mid v \\
\text{(predicate)} & p & ::= n \mid n.p \\
\multicolumn{3}{c}{n \in \textbf{Name} \quad X \in \textbf{Var} \quad T \in \textbf{Type} \quad v \in \textbf{Value}} \\
\\
\text{(module)} & M & ::= \textbf{module}\ n\ \overline{E} \\
\text{(entry)} & E & ::= \textbf{declare}\ sig \mid \textbf{rule}\ r \mid \\
 & & \quad \textbf{require}\ sig \mid \textbf{provide}\ sig \mid \\
 & & \quad \textbf{import}\ n\ \textbf{as}\ n\ \textbf{with}\ \{\overline{p = p}\} \\
\text{(signatures)} & sig & ::= p : T \times \cdots \times T
\end{array}
$$

Figure 5. Core Datalog IR with modules.

in the following section, the interface of a module is sufficient for checking the validity of an import. In particular, it is not necessary to expose the full definition of a module.[3]

3 Formalization of the Module System

The goal of this paper is to provide a module system for a Datalog IR. The module system needs to be expressive enough that existing Datalog dialects can use it as a compilation target. In this section, we formalize the syntax of modules, define inference rules for compositional module validation, and provide the linking routine.

3.1 Syntax of the Module System

We have shown examples of Datalog modules in Section 2, but have not formally introduced the syntax. We remedy this omission now. The top half of Figure 5 shows the grammar of a standard Datalog. Datalog rules $p(\overline{X}) :- \overline{a}.$ are inference rules with a head $p(\overline{X})$ and a rule body $\overline{a}$. A rule body consists of a sequence of atoms $\overline{a}$, each of which is either a predicate call $p(\overline{t})$, an equality constraint $t = t$, or an inequality constraint $t\ != t$. A term t is either logic variable X or a constant value v. The only difference from standard Datalog is that we explicitly allow qualified predicate names p to support qualified imports later.

The remainder of Figure 5 introduces the syntax of Datalog modules. Each module carries a name n and a sequence of entries $\overline{e}$. There are 5 different kinds of entries. First, a module can declare the signature of a locally defined relations using **declare** *sig*. We require explicit declarations with their type to support static validation of modules. Note that many current Datalog systems (e.g., IncA and Soufflé) require similar signatures, too. Second, a module can define standard Datalog rules using **rule** *r*. Third, an entry **require** *sig* requests a relation with the given signature to be provided during linking. Fourth, **provide** *sig* exports the relation with its signature given that it is locally defined. Finally, language-integrated linking is triggered by an entry **import** m **as** n **with** $\{\overline{p = p}\}$, which imports a module named m as n while resolving requirements of m following the assignments $\overline{p = p}$. The examples in the previous section followed and exemplified this syntax for Datalog modules.

[3]This enables higher-order modules that abstract over modules similar to SML functors, but we have not explored this option yet.

3.2 A Type System for Modules

A valid module must satisfy three key properties:

1. The Datalog rules in the module must be well-typed.
2. Provided relations must be locally available.
3. Required relations must be bound during linking.

We designed a type system to validate modules with respect to these properties. However, we leave the type checking of standard Datalog rules undefined, since this is orthogonal to the module system. Moreover, we assume that no accidental variable shadowing occurs, such as re-declarations of relations with conflicting signatures, which can be easily checked and fixed through renaming.

To validate a module, we operate in two phases. First, we collect key information about the module: the required relations Γ^{in}, the provided relations Γ^{out}, and the locally defined relations Γ^{loc}. The required and provided relations constitute a module's public interface $\langle\Gamma^{in}, \Gamma^{out}\rangle$. The local definitions Γ^{loc} are only necessary locally during validation. We formalize the first phase of module validation in Figure 6. We visit each entry of a module using judgment $\Gamma^{in}, \Gamma^{out}, \Gamma^{loc} \vdash e \dashv \Gamma^{in}, \Gamma^{out}, \Gamma^{loc}$, which adds signatures to the appropriate context. The only interesting case is the rule for imports: We add all provided relations Γ^{out}_m of the imported module m to the local definitions Γ^{loc}, but only after qualifying their name. Here, $m : \langle\Gamma^{in}, \Gamma^{out}\rangle|\Gamma^{loc}$ yields the public interface and local definitions of module m, as defined by the last inference rule in Figure 6.

After we have gathered all relevant information about the relations of a module, we can enter the second phase of module validation and check the module definition. We define module validity through inference rules in Figure 7. Rule V-Module first computes the public interface and local definitions of a given module. It then checks two properties. First, the relations Γ^{out} provided by the module must be subsumed by the locally available definitions Γ^{loc} and their signatures must match. Second, all module entries must be valid, assuming both locally defined relations Γ^{loc} and externally required relations Γ^{in} can be used. The remainder of the rules in Figure 7 validate module entries ($\Gamma \vdash e \checkmark$). For **declare** *sig*, **require** *sig*, and **provide** *sig*, no extra check is necessary: we used these entries in the first phase of module validation. For **rule** *r*, we rely on a standard typing relation for Datalog rules, but provide the appropriate context consisting of required and local relations. Recall that local relations include imported ones, which thus may also be used in rules.

$$\frac{}{\Gamma^{in}, \Gamma^{out}, \Gamma^{loc} \vdash \textbf{declare}\ p : \overline{T} \dashv \Gamma^{in}, \Gamma^{out}, (\Gamma^{loc}; p : \overline{T})}\ \text{C-Decl} \qquad \frac{}{\Gamma^{in}, \Gamma^{out}, \Gamma^{loc} \vdash \textbf{rule}\ r \dashv \Gamma^{in}, \Gamma^{out}, \Gamma^{loc}}\ \text{C-Rule}$$

$$\frac{}{\Gamma^{in}, \Gamma^{out}, \Gamma^{loc} \vdash \textbf{require}\ p : \overline{T} \dashv (\Gamma^{in}; p : \overline{T}), \Gamma^{out}, \Gamma^{loc}}\ \text{C-Req} \qquad \frac{}{\Gamma^{in}, \Gamma^{out}, \Gamma^{loc} \vdash \textbf{provide}\ p : \overline{T} \dashv \Gamma^{in}, (\Gamma^{out}; p : \overline{T}), \Gamma^{loc}}\ \text{C-Prov}$$

$$\frac{m : \langle \_, \Gamma^{out}_m \rangle | \_ \qquad \Gamma^{loc}_m = \{n.p : \overline{T} \mid p : \overline{T} \in \Gamma^{out}_m\}}{\Gamma^{in}, \Gamma^{out}, \Gamma^{loc} \vdash \textbf{import}\ m\ \textbf{as}\ n\ \textbf{with}\ \{\overline{p = p}\} \dashv \Gamma^{in}, \Gamma^{out}, \Gamma^{loc} \cup \Gamma^{loc}_m}\ \text{C-Import}$$

$$\frac{(\Gamma^{in}_1, \Gamma^{out}_1, \Gamma^{loc}_1) = (\emptyset, \emptyset, \emptyset) \qquad \forall i.\ \ \Gamma^{in}_i, \Gamma^{out}_i, \Gamma^{loc}_i \vdash e_i \dashv \Gamma^{in}_{i+1}, \Gamma^{out}_{i+1}, \Gamma^{loc}_{i+1}}{\textbf{module}\ n\ e_1 \cdots e_n : \langle \Gamma^{in}_{n+1}, \Gamma^{out}_{n+1} \rangle | \Gamma^{loc}_{n+1}}\ \text{C-Module}$$

Figure 6. Collect the required relations Γ^{in}, provided relations Γ^{out}, and locally defined relations Γ^{loc} of a module.

$$\frac{\textbf{module}\ n\ e_1 \cdots e_n : \langle \Gamma^{in}, \Gamma^{out} \rangle | \Gamma^{loc} \qquad \Gamma^{out} \subseteq \Gamma^{loc} \qquad \forall i.\ \ \Gamma^{loc} \cup \Gamma^{in} \vdash e_i \checkmark}{\textbf{module}\ n\ e_1 \cdots e_n\ \checkmark}\ \text{V-Module}$$

$$\frac{}{\Gamma \vdash \textbf{declare}\ sig \checkmark}\ \text{V-Decl} \qquad \frac{\Gamma \vdash_{rule} p(\overline{x}) {:-}\ \overline{a}. \checkmark}{\Gamma \vdash \textbf{rule}\ p(\overline{x}) {:-}\ \overline{a}. \checkmark}\ \text{V-Rule} \qquad \frac{}{\Gamma \vdash \textbf{require}\ sig \checkmark}\ \text{V-Req} \qquad \frac{}{\Gamma \vdash \textbf{provide}\ sig \checkmark}\ \text{V-Prov}$$

$$\frac{m : \langle \Gamma^{in}, \_ \rangle | \_ \qquad \forall i.\ \ \Gamma^{in}(p_i) = \Gamma(q_i) \qquad \text{keys}(\Gamma^{in}) \setminus \{p_1, \ldots, p_k\} = \emptyset}{\Gamma \vdash \textbf{import}\ m\ \textbf{as}\ n\ \textbf{with}\ \{p_1 = q_1, \ldots, p_k = q_k\} \checkmark}\ \text{V-Import}$$

Figure 7. Validate the provided relations and entries of a module, in particular ensuring imports satisfy all requirements.

Finally, we validate import entries in V-Import. This is where linking takes place, and thus we need to validate that the imported module is correctly instantiated. We first obtain the required relations Γ^{in} of the imported module m. Then, for each binding $p_i = q_i$ in the import declaration, we validate that the type $\Gamma^{in}(p_i)$ required by m matches the local type $\Gamma(q_i)$ of the assigned relation. Moreover, the import declaration must provide bindings for all required relations in Γ^{in}. If these conditions are met, linking will succeed.

One important point to notice about module validation is that it is compositional: While checking a module, we only ever use the interface of other modules but not the definition. We can see this in rules C-Import and V-Import, which use the provided and required relations of the imported module only. Therefore, module validation supports separate compilation as desired.

3.3 Linking Modules

Our linking algorithm operates on a group of modules that are linked to form bundles or an executable program. Each group of modules is identified by one primary module and a set of library modules. The primary module contains import statements, while library modules in the same group are free of imports. The library modules in a group are given by the import statements in the primary module. During linking, we resolve and eliminate the imports in the primary module, producing a single import-free module with the same name as the primary module. Consider the following example, where we aim to compile and link five modules:

```
module M1                    module M2
  import M2 as m2              import M4 as m4
  import M3 as m3              import M5 as m5
module M3                    module M4
  import M2 as m2            module M5
```

First, we order all modules topologically based on their imports. Following this order, we first separately compile `M5` and `M4`, followed by `M2`, `M3` and finally `M1`. Each time we encounter a module with an import statement during compilation, we trigger linking. For example, no linking happens during the compilation of `M4` and `M5`, since they are import-free. Compilation of `M2`, `M3` and `M1` triggers linking. When compiling and linking `M2`, we use `M2` as a primary module and `M5` and `M4` as library modules in the group. After linking, `M2` is free of imports and we link the primary module `M3` with the linked `M2` module as library module. The topological order ensures that all library modules are consistently free of imports. Note that the existence of a topological order precludes cyclic imports, which we guarantee before linking. Since this is a severe

Algorithm 1 Algorithm to link a group of modules

```
1:  Input:
2:      module $N\ \overline{imp}, \overline{req}, \overline{prov}, \overline{decl}, \overline{rule}$   primary module
3:      $Mods$                    set of import-free library modules
4:  procedure LINK($P$, $Mods$)
5:      $Entries \leftarrow \overline{req} \cup \overline{prov} \cup \overline{decl} \cup \overline{rule}$
6:      for import $m$ as $n$ with $\{\sigma\} \in \overline{imp}$ do
7:          module $m$ _, _, _, $\overline{decl}_m, \overline{rule}_m \leftarrow$ get($m$, $Mods$)
8:          $Entries \leftarrow Entries \cup \{$prefix$(n, D) \mid D \in \overline{decl_m}\}$
9:          $Entries \leftarrow Entries \cup$
10:             $\{$prefix$(n,$ subst$(\sigma, R)) \mid R \in \overline{rule}_m\}$
11:     return module $N$ $Entries$
```

limitation, given that various Datalog dialects support cyclic imports, we will show how to resolve it in the next section.

We define our linking algorithm in 1. It expects two inputs: a group's primary module P and a set of import-free library modules *Mods* contained in the current group. When we link a module, we preserve all entries in the primary module *except* for the imports (line 5) by initializing *Entries* with all entries of P, but excluding $\overline{imp}$. Hence, we do not alter the public interface of the primary module during linking. This entails that open modules stay open and closed modules stay closed. Now, we resolve all imports by inlining corresponding entries. We proceed in three steps for each imported module:

(i) We collect and **prefix** all local declarations in the imported module with the local module qualifier n.
(ii) We use **subst** to replace predicate names referencing required signatures with the right-hand side of the import's binding.
(iii) Finally, we **prefix** the predicate names in each rule of step (ii) with the local module qualifier n. Hence, the rules now match their declared signature again.

To produce the linked module, we create a new module with the name of the primary module and incorporate all entries accumulated during the linking process.

To show that our linking is indeed correct, we proof two key properties: (i) a linked module is free of imports, and (ii) linking does not break module validity.

Definition 3.1. Let P be a module, and *Mods* be a set of import-free modules, that is $\forall m \in Mods.\ \textbf{imports}(m) = \emptyset$. We call the tuple ($P$, *Mods*) a *group of modules* with *primary module P* and library modules *Mods*, if all imports in P are resolvable by *Mods*, that is $\textbf{modNames}(\textbf{imports}(P)) \subseteq \textbf{names}(Mods)$.

Theorem 3.2. *Let* $(P, Mods)$ *be a group of modules, then* $\textsc{Link}(P, Mods) = P'$ *with* $\textbf{imports}(P') = \emptyset$.

Proof. The proof follows directly from line 5, where we include all entries except imports from P in P', and lines 8-9 where we only add entries from import-free modules. □

To show that linking does not break module validity, we first formulate and proof two lemmas.

Lemma 3.3. *Let* $(P, Mods)$ *be a group of modules, and* $\textsc{Link}(P, Mods) = P'$*, then the interface of P is equal to the interface of P', that is:* $\Gamma^{in}_P = \Gamma^{in}_{P'} \wedge \Gamma^{out}_P = \Gamma^{out}_{P'}$.

Proof. We copy all entries, including the interface of P (line 5), and ignore the interface of other modules (lines 8-9). □

Lemma 3.4. *Let* $(P, Mods)$ *be a group of valid modules, and* $\textsc{Link}(P, Mods) = P'$*, then the local context of* P' *extends the local context of P, that is* $\Gamma^{loc}_P \subseteq \Gamma^{loc}_{P'}$.

Proof. Let $p : \overline{T} \in \Gamma^{loc}_P$, then line 5 guarantees that $p : \overline{T} \in \Gamma^{loc}_{P'}$. If $n.p : \overline{T} \in \Gamma^{loc}_P$, that is $n.p$ is imported, then $\exists m \in Mods.\ p : \overline{T} \in \Gamma^{out}_m \subseteq \Gamma^{loc}_m$. Line 8 copies all local signature $\overline{decl}_m$ from Γ^{loc}_m prefixed with the same n to P'. □

Finally, we can now formulate the theorem that linking does not break module validity.

Theorem 3.5. *Let* $(P, Mods)$ *be a group of modules, such that* $P\ \checkmark$*, and all modules in Mods are valid, that is* $\forall m \in Mods.\ m\ \checkmark$*, then* $\textsc{Link}(P, Mods) = P'$ *is valid, that is* $P'\ \checkmark$.

Proof. To show that P' is valid, V-Module must hold. Given lemma 3.3 and 3.4 it follows: $\Gamma^{out}_{P'} = \Gamma^{out}_P \subseteq \Gamma^{loc}_P \subseteq \Gamma^{loc}_{P'}$. That is, to prove V-Module holds, it is enough to show that all entries of P' are valid. In particular, Theorem 3.2 guarantees that P' is import-free, meaning V-Import is never required. Additionally, the rules V-Decl, V-Req, and V-Prov all hold unconditionally, thus we only need to show that V-Rule holds for all rules of P'. Consider rule $r \in \textbf{rules}(P')$. Either r was originally defined in P, in which case r is still valid, since we check it unmodified under the extended context $\Gamma^{in}_{P'} \cup \Gamma^{loc}_{P'} \supseteq \Gamma^{in}_P \cup \Gamma^{loc}_P$, or r originates from a module $m \in Mods$. In this case, we need to show that r is still valid if we copy all local entries, substitute requirements and prefix them (lines 8-9). Since we copy *all* rules $\overline{rule}_m$ with their corresponding signatures $\overline{decl}_m$ with the same prefix n, they stay valid, as long as they don't reference requirements. If a rule references a requirement of m, typing is still valid because we replace calls of required relations in m with calls to local or required relations in P using **subst** and the bindings σ. Since P is valid, we know that for all requirements in m a valid binding and as such a valid signature exists in P. Given lemma 3.3 and 3.4, we also know that all required and all local signatures in P must also exists in P', after prefixing. □

This concludes our formalization of the module system, including the module checker and linking process.

```
module A                        module B
require Q: Int                  require R: Int
declare R: Int                  declare Q: Int
provide R: Int                  provide Q: Int
R(x) :- Q(x).                   Q(x) :- R(x).

module C
import A as a { Q = b.Q }
import B as b { R = a.R }
```

Figure 8. Modules A and B are mutually dependent through C, but do not have a cyclic dependency on each other.

4 Translation of Usage Patterns

Several Datalog dialects are already equipped with component systems that support a variety of usage patterns not directly supported by our module system. In this section, we demonstrate how to utilize requirements and language-integrated linking to express these patterns.

4.1 Cyclic Imports

Many module systems support cyclic dependencies between modules, which are not allowed in the module system presented in this paper. We disallow cyclic imports because our linking algorithm requires a topological order based on the import dependencies between modules. For example, the following code with a cyclic dependency between A and B is prohibited in our module system:

```
module A                        module B
import B as b                   import A as a
declare R: Int                  declare Q: Int
provide R: Int                  provide Q: Int
R(x) :- b.Q(x).                 Q(x) :- a.R(x).
```

However, not all hope is lost, as we can express cyclic import dependencies using an intermediate module that introduces a mutual dependency between modules instead. In particular, we use our requirement feature to implement this dependency without cycles. Figure 8 shows how we can express our running example with a mutual dependency. Instead of having A and B directly depend on each other, we introduce an intermediate module C that imports both. Thus, we eliminate the direct dependency cycle. Since A and B still need access to Q and R respectively, we introduce requirements for those relations that we resolve in module C.

At first glance, it may seem that we have merely shifted the problem. We still reference b.Q in the import of A and a.R in the import of B. However, it is important to note that these bindings are simply equalities between relations, which are resolved by inlining during linking.

4.2 Inheritance

Inheritance is a common feature in various Datalog dialects. For instance, QL offers an object-oriented dialect for

```
.comp BaseConfig {
    .decl R(x:number)
    R(41).
}
.comp ConfigOne: BaseConfig { R(42). }
.comp ConfigTwo: BaseConfig { R(43). }
.init c1 = ConfigurationOne
.init c2 = ConfigurationTwo
```

(a) Soufflé program

```
module BaseConfig               module ConfigOne
declare R: Int                  require super$R: Int
provide R: Int                  declare R: Int
R(41).                          provide R: Int
                                R(x) :- super$R(x).
                                R(42).

module ConfigTwo                module Main
require super$R                 import BaseConfig as BC
declare R: Int                  import ConfigOne as c1
provide R: Int                    with { super$R = BC.R }
R(x) :- super$R(x).             import ConfigTwo as c2
R(43).                            with { super$R = BC.R }
```

(b) Module system

Figure 9. Multiple Soufflé components inherit relations from a base component.

Datalog supporting classes and inheritance [6]. Similarly, Soufflé includes a sophisticated component system that supports extending components. Figure 9 shows such an example with three components in Soufflé. We define one super component, BaseConfig that declares a single relation R with fact R(41). Subsequently, we define and instantiate two child components, namely ConfigOne and ConfigTwo which extend BaseConfig. Each child component inherits relation R and extends it with one additional fact. Executing the program, we expect the following output: c1.R: $\{41, 42\}$ and c2.R: $\{41, 43\}$.

Using required relations, we can express the same inheritance structure within our module system. We create separate modules for each Soufflé component and one main module to represent the entire program. Both modules representing ConfigOne and ConfigTwo require the relation R that they inherit from the parent. We mark these requirements with a super prefix to indicate inheritance. Additionally, both modules also redefine relation R with two bodies. The first body calls the inherited relation, while the second body defines the additional fact for R.

For each component instantiation in the original Soufflé program, we import the corresponding module in the Main module. We use the name given on instantiation as the local module name during import. For example, we import module ConfigOne with the local name c1. Note that to resolve the

```
.decl Q(x:number)
Q(41).
.comp Config {
    .decl R(x:number)
    R(x) :- Q(x).
    R(42).
    .comp NestedConfig {
      .decl S(x:number)
      S(x) :- R(x).
      S(x) :- Q(x).
    }
    .init n = NestedConfig
}
.init c = Config
```

(a) Soufflé program

```
module Main
import Config as c
  with { outer$Q = Q }
declare Q: Int
Q(41).
```

```
module NestedConfig
require outer$R: Int
require outer$Q: Int
declare S: Int
provide S: Int
S(x) :- outer$R(x).
S(x) :- outer$Q(x).
```

```
module Config
import NestedConfig as n with {
  outer$Q = outer$Q, outer$R = R
}
require outer$Q: Int
declare R: Int
provide R: Int
R(x) :- outer$Q(x).
R(42).
```

(b) Module system

Figure 10. Relations from the outer scope of a Soufflé program are passed down into nested component.

requirements for the inherited relations, we also need to import the `BaseConfig` module with an arbitrary local name.

This approach is also flexible enough to support nested inheritance hierarchies. In such cases, we pass the requirements along the inheritance hierarchy from one module to the next. Using the same design, we can also mimic Soufflé's multiple inheritance, where a single component inherits from more than one parent component. Instead of requiring relations from just one parent component, we additionally require relations from a second parent.

4.3 Scoping

Scoping plays a crucial role in organizing code in Datalog dialects. For instance, Soufflé implicitly exposes all relations declared in a program to all its components. Each nested component can also access all content from the outer component. In Figure 10, the relation `Q`, defined in the outermost scope of a Soufflé program, is available in the components `Config` and `NestedConfig`. Consequently, `Q` can be referenced in the component's internal rule `R` or `S` respectively.

We encode this scoping behaviour with requirements. Modules, corresponding to a component, require all relations from the outer scope, that we resolve on import. For example, the `Main` module imports module `Config` and fulfills the requirement for `outer$Q` with its local definition. `NestedConfig` requires everything from the outermost main scope, but also all relations from the `Config` scope. That is, each nesting level propagates all requirements from the parent scope and adds new requirements for relations in the current scope. Specifically, the `Config` module propagates relation `Q` from the main scope and also propagates its new definition `R` to `NestedConfig`.

We have seen how we can express even complex Datalog dialect features in our module system. In the next section, we extend the IncA compiler framework with our module system and use it to link a large case study in Doop.

5 Implementation and Case Study

IncA is a multi-level IR compiler framework, which mirrors the architecture of LLVM [14], but adapts it for Datalog. Datalog dialects can implement compiler frontends to convert their code into an extensible Datalog IR. Independently of the frontend, this IR code is transformed, analyzed and optimized in multiple interleaved passes. Finally, compiler backends translate the IR code for various Datalog engines and execute it. The rest of this section demonstrates how we implemented and use our module system in IncA.

5.1 Extend the IncA Datalog IR

At its core, IncA consists of a typed core Datalog IR, that is extensible with any number of IR extensions. We extend IncA's core Datalog IR with the syntax described in Figure 5. That is, a Datalog program consists of a set of modules, each containing a set of module entries. Thus, Datalog frontends produce one or more modules that are linked to an executable program or bundles. Note that compiler backends reject open modules because they are not executable.

We extend the type system of IncA's core IR to support our module system by following the rules in Figure 6 and Figure 7. However, our implementation includes one additional check to guarantee that local names of modules are unique to prevent name clashes. For readability purposes, we left this detail out of the presentation in Figure 7.

Finally, we add our linking algorithm from Section 3.3 to IncA's compilation pipeline. Technically, we leverage an existing mechanism within IncA's compilation pipeline to accomplish this, namely *lowerings*. Lowerings allow IR extensions to be expressed in terms of other IR extensions by desugaring them. We implement our linking algorithm as a lowering which desugars imports by inlining relations.

However, it turns out that some compilation steps are in conflict with partial linking. In particular, some optimizations are only fully applicable when the final Datalog program is known. To enable specialized handling of these cases, we extend IncA to differentiate closed-world and open-world compilation. Closed-world compilation assumes complete knowledge of the entire Datalog program, whereas open-world compilation only operates on a partial Datalog program. For example, consider an optimization that removes all relations from a Datalog module that are never queried. During an open-world compilation, we cannot yet determine if provided relations are queried. This decision can only be made once a program is fully linked. Nevertheless, we still provide an optimization for open-world compilation to remove only non-provided relations that are never queried.

In the remainder of this section, we demonstrate its applicability to a large Doop analysis written in Soufflé.

5.2 Case Study: Points-To Analysis in Doop

Doop is a highly configurable points-to framework for JVM bytecode written in Soufflé's Datalog dialect. We are able to automatically translate an existing large context-insensitive points-to analysis to modules in IncA's IR. To this end, we modify IncA's Soufflé frontend to produce modules, following the strategies described in Section 4.

Figure 11 illustrates the analysis structure and generated IR modules. The original Soufflé program spans ~7000 LOC across 5 components, exploiting scoping and inheritance. `ContextInsensitiveConfiguration` inherits all relations defined in `AbstractConfiguration`. Besides inheritance, the program also uses nested scoping. That is, each component can access all declarations from the main program scope. At the same time, relations in the main scope can also access the `basic` and `config` instances through the `mainAnalysis` instance.

We translate this program into 6 modules, one for each component and one main module representing the primary program. Each module requires all relations from the main module. `ContextInsensitiveConfiguration` also requires all inherited relations from `AbstractConfiguration`. Additionally, `BasicContextSensitivity` re-provides relations imported from `Basic` and `ContextInsensitiveConfiguration` to the main scope.

We link the program in the topological order of the modules. First, we separately lower, optimize, and type check the open modules `Basic`, `AbstractConfiguration`, and `ContextInsensitiveConfiguration`. Then, we proceed with `BasicContextSensitivity`, which remains an open module as it still requires relations from the outer scope. Finally, we generate the executable program by linking the main module and resolving all remaining dependencies.

5.3 Case Study: Interoperability

As indicated in Figure 1, our module system enables interoperability between Datalog dialects. To showcase this, we implement an escape analysis in Functional IncA by utilizing an existing points-to analysis written in Soufflé.

```
.comp Basic { ... }
.comp AbstractConfiguration { ... }
.comp ContextInsensitiveConfiguration:
    AbstractConfiguration { ... }
.comp BasicContextSensitivity {
  .init basic = Basic
  .init configuration = ContextInsensitiveConfiguration
  ...
}
.init mainAnalysis = BasicContextSensitivity ...
```

(a) Soufflé program

```
module Basic
... // require all relations from outer scope
module AbstractConfiguration
... // require all relations from Main scope
module ContextInsensitiveConfiguration
... // require Main & AbstractConfiguration relations
module BasicContextSensitivity
... // require all relations from Main scope
import Basic as basic with {/*Main Scope*/}
import AbstractConfiguration as A with {/*Main Scope*/}
import ContextInsensitiveConfiguration as config
    with {/*Main Scope & Inherited from A*/}
// re-provide everything from basic and config
provide basic.X
provide config.X
...
module Main
import BasicContextSensitivity as mainAnalysis
  with {/*all relations from this scope*/}
...
```

(b) Module system

Figure 11. Structure of the context-insensitive Doop analysis written in Soufflé.

Functional IncA is a functional language that does not use Datalog relations in its frontend dialect, but still compiles down to Datalog IR. In contrast, the Soufflé dialect closely aligns with Datalog and operates directly on Datalog relations. To integrate these two languages, we adapt Functional IncA to import Datalog relations by name and module. To this end, we extend Functional IncA's parser, type checker, and compiler. We treat imported relations as a set of tuples on which we operate with set comprehensions. Consider an excerpt from our escape analyses in Functional IncA:

```
import PointsTo as pt:
   AssignHeapAllocation(String, String, String), ...

def mayEscape(): Set[(String, Boolean)] =
  { (obj, mayOutlive(obj, method)) |
    (obj, _, method) in pt.AssignHeapAllocation }
```

We import the `AssignHeapAllocation` relation from our Soufflé module into Functional IncA. This relation holds all heap allocations in the program that we want to analyze. To map Soufflé types to Functional IncA types, we require explicit type annotations upon import. The function `mayEscape` determines if each heap allocation outlives the method in which it is defined. To do so, we use the helper function `mayOutlive`.

To create an executable program, we follow the steps outlined earlier. First, we separately compile the Soufflé point-to analysis to multiple IR modules and link them to a single big IR bundle. Then, we compile the Functional IncA module to an IR module and link it with the Soufflé IR bundle, yielding the executable Datalog program. We can execute the program with any Datalog engine supported by IncA.

6 Related Work

We design, formalize, and implement a typed module system for Datalog intermediate representations. To the best of our knowledge, we are the first to study such a module system. However, prior research on module systems, especially in logic programming languages and Datalog, does exist and will be discussed in this section.

6.1 Module Systems in the Broader Literature

Cardelli [9] provides a theoretical framework for System F, focusing on modularization and linking through program fragments. These fragments are syntactically well-formed program terms that support separate compilation (simplified to type checking) and partial linking. They define linking as a process generating an output program from a collection of fragments and a description of how these are combined. This description is specified in a configuration language, different from the module language, and formalized by linksets. Linking either produces a complete program or an *incomplete* library for further linking. Our modules align with the concept of fragments, as they are collections of relations that are separately compiled and checked. However, our linking process is integrated into the language via import statements, eliminating the need for a separate configuration language. In our system, open modules are akin to libraries, while closed modules correspond to complete programs.

While module systems are prevalent in nearly all widely used programming languages, such as OCaml, Go, Java, Python, or C++, a detailed comparison to all of them is not practical. Instead, we focus on a comparison with one of the most comprehensive module systems found in Standard ML (SML). SML [17] is a general-purpose functional programming language with an advanced module system that separates linking and library creation from the core language. There are three core concepts in SML's module system: structures, signatures, and functors. A structure is a collection of declarations with concrete implementations. Structures are instantiated by binding them to a name and using qualified names to access their contents. The second concept, signatures, describes the structure's interface, consisting of names and types for all entities and substructures. A functor, the third concept, maps between structures to parameterize modules. Given an existing structure that adheres to a signature, a functor creates a new structure from it. Our Datalog IR module system resembles SML structures, as our modules are collections of relations that are instantiated with local names through imports. A module's public interface is similar to the signature of a structure, as type checking an IR module requires only the public interface. These interfaces enable us to write parametric modules that abstract over modules, similar to how SML functors abstract over modules. However, contrary to SML, our linking process is integrated into the Datalog IR language.

6.2 Module Systems in Logic Programming

Miller [16] defines a pure theory of module systems for logic programming languages using nested implications. To this end, they define a logic language that extends the syntax of horn clauses with implications in goals and in bodies of clauses. They do not address linking or partial compilation in their work. We implemented our module system as part of the IncA compiler framework and demonstrated its usability with two case studies. While their system structures multiple predicates within a module, it lacks mechanisms for information hiding. Our modules are also a collection of relations, but we support information hiding by selectively exporting relations. Similar to us, they support parameterization by using variables as predicates in module definitions. However, their approach does not detail how these variables are resolved. They support an import mechanism through implication. That is, if a body of a relation is attempted as a goal, it will come with the module as a hypothesis. In our system, imports are resolved by linking, which inlines imported relations using a local module name. Our modules are parametric in relations through requirements that we resolve with imports. We do not need external linking mechanisms.

Hill [10] also presents a theory of a module system for a typed logic programming language called Gödel. A module is a collection of predicates and consists of a public (export) and a private (local) interface. Public predicates are accessible outside the module, while private ones are restricted to the module. Modules can be imported into one another, bringing in all public symbols from the imported module. Our modules are a collection of relations as well. We explicitly provide relations to other modules and import them by inlining. Using local names, we can import the same module multiple times, enabling reusable, parametric modules. Gödel also supports parametric modules, allowing parameterization by types and relations, which are resolved upon import. However, Gödel does not support importing the same module multiple times within the same namespace. Their work also does not address compositional validity or partial linking.

Sannella and Wallen [22] present a theoretical module system for Prolog. Although Prolog differs from Datalog in semantics, its surface language of horn clauses with atoms and terms is quite similar. Even though our module system purely focuses on Datalog, there are still similarities. The core abstraction of their approach is called *structure*. Structures contain declarations of functions and relations and can export those to other structures. It is also possible to inherit declarations from other structures or make a structure parametric in other structures. Since standard Datalog does not support functions, our design does not capture them. However, our modules can export relations using `provide` and also `require` other relations. As discussed in Section 4, this form of parameterization can express various patterns, including inheritance. Their structures and our modules both are accessed with qualified names. Additionally, structures include signatures that specify their content. However, since their theory is untyped, their signatures only include the arity of relations. While we always explicitly require type annotations for relations and requirements, structures can infer their signatures. Since our module system is designed for IRs and is not user-facing, type inference is not a necessity.

6.3 Datalog Dialects with Module Systems

QL [6] is an object-oriented frontend language that compiles to Datalog and is evaluated by a custom engine to perform static analysis on various programming languages. QL supports different module types, particularly parameterized modules, which accept multiple predicate names as input that must be resolved upon instantiation. That is, a module can be reused by instantiating it multiple times with different predicates. Similarly, our module system allows parametrization over relations by defining requirements. Our modules can be imported multiple times with different requirements. Even though QL directly translates to Datalog, it is unclear how and if modules translate to Datalog as well.

LogicQL [5] is another mostly pure Datalog dialect with its own evaluation engine. Modules consist of relations and type definitions with support for information hiding. In particular, a module can export declarations for other modules to use. LogicQL supports neither the parametrization of modules nor of relations. Similarly, our modules are collections of relations that support information hiding by providing certain relations to other modules. Additionally, we enhance modules by making them parametric through requirements.

DDLog [19] is an incremental execution engine and dialect of Datalog. It supports a module system inspired by Haskell and Python, that allows importing types, functions, or relations. Our formalization of the module system only focuses on relations, since our targeted Datalog IR has no notion of functions or algebraic data. DDLog allows importing modules to the same namespace or keeping the namespace separate. We always use distinct namespaces for modules.

Soufflé [11] is a high-performance C++ engine with a datalog frontend language that supports modularization through components. Components can contain relations, type declarations and definitions, or other nested components. They can be instantiated multiple times by assigning a unique name to them. Access to a component's content requires a fully qualified name. Similarly, our modules encapsulate relations, can be imported multiple times with local names, and require fully qualified names for access. While Soufflé's components support inheritance and scoping, our modules do not have built-in support for these features. However, as shown in Section 4, our design is flexible enough to accommodate these patterns. Soufflé also allows components with parametric type arguments. We consider type parametrization orthogonal to our modules. Instead, we suggest that in a compiler framework, extensions could provide type-parametric relations, rather than type parametric modules. Extending our module system to support parametric relations should be straightforward based on our current formalization. Before execution, Soufflé flattens and inlines components to produce standard Datalog code. Our linking algorithm also inlines relations to produce a single executable Datalog program.

7 Conclusion

We propose the first module system for Datalog IRs to support interoperability between Datalog dialects. Modules are uniquely named collections of relations that can be separately type checked, compiled, and partially linked to form bundles. They can both, provide relations to other modules and require relations from other modules. Requirements must be resolved when linking the surrounding module, which we trigger with a language-integrated import statement. This approach enables parametric modules, which naturally extend the expressiveness of standard Datalog. We formalize the static semantics and linking algorithm of our module system and demonstrate how we support cyclic imports, inheritance, and scoping. We then integrate our module system into the existing Datalog framework IncA, and implement a Soufflé frontend to compile complex Soufflé programs with inheritance and scoping into multiple IR modules. Finally, we use our implementation to compile and link an existing large context-insensitive points-to analysis written in Doop into an executable Datalog program. To show interoperability, we implement an escape analysis in Functional IncA using a Soufflé points-to analysis.

Acknowledgments

This work has been funded by the German Research Foundation (DFG) (Project numbers 451545561 & 508316729) and the European Research Council (ERC) under the European Union's Horizon 2023 (Grant Agreement ID 101125325).

References

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

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

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

[4] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281.

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[6] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

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

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

[9] Luca Cardelli. 1997. Program Fragments, Linking, and Modularization. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 266–277. https://doi.org/10.1145/263699.263735

[10] P. M. Hill. 1993. A parameterised module system for constructing typed logic programs. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 2* (Chambery, France) *(IJCAI'93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 874–880.

[11] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[12] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 773–792. https://doi.org/10.1145/2384616.2384673

[13] David Klopp, Sebastian Erdweg, and André Pacak. 2024. A Typed Multi-Level Datalog IR and its Compiler Framework. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 327 (oct 2024), 29 pages. https://doi.org/10.1145/3689767

[14] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[15] Shane McIntosh. 2011. Build system maintenance. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 1167–1169. https://doi.org/10.1145/1985793.1986031

[16] Dale Miller. 1986. A Theory of Modules for Logic Programming.

[17] Robin Milner, Mads Tofte, and Robert Harper. 1990. *The definition of Standard ML.* MIT Press, Cambridge, MA, USA.

[18] André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 222)*. Schloss Dagstuhl, 7:1–7:28.

[19] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog*. https://api.semanticscholar.org/CorpusID:169040311

[20] Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 264 (oct 2023), 26 pages. https://doi.org/10.1145/3622840

[21] Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. 2022. Seamless deductive inference via macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3497776.3517779

[22] D.T. Sannella and L.A. Wallen. 1992. A calculus for the construction of modular prolog programs. *The Journal of Logic Programming* 12, 1 (1992), 147–177. https://doi.org/10.1016/0743-1066(92)90042-2

[23] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Programming Language Design and Implementation (PLDI)*. ACM.

[24] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (Tsukuba, Japan) *(APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 97–118. https://doi.org/10.1007/11575467_8

Received 2024-06-18; accepted 2024-08-15