Datalog als interne DSL in Rust

Mattes Bieniarz

Bachelorarbeit

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und Informatik der
Technischen Hochschule Mittelhessen

Gießen, 2025-09-18

Referent:	Prof. Dr. Michael Elberfeld
Korreferent:	Prof. Dr. Uwe Meyer

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 2025-09-18

Inhaltsverzeichnis

1 Einleitung

Datalog ist eine deklarative Logikprogrammiersprache, die ursprünglich aus dem Bereich der Datenbankforschung stammt, heute jedoch in verschiedenen Anwendungsdomänen Verwendung findet – etwa in der statischen Codeanalyse. Die Sprache zeichnet sich durch eine klare Semantik, terminierende Auswertungsstrategien und die Möglichkeit aus, rekursive Zusammenhänge in kompakter Form zu formulieren.

1.1 Problemstellung und Zielsetzung

Die zentrale Fragestellung dieser Arbeit lautet: *Wie kann eine typsichere, modulare interne Datalog-DSL in Rust mittels prozeduraler Makros realisiert werden, die Wiederverwendbarkeit und Skalierbarkeit unterstützt?*

Um diese Frage zu beantworten, wird in dieser Arbeit eine eigene interne Datalog-DSL mit dem Namen *Tofina* entworfen und implementiert. Das Hauptziel besteht darin, eine Architektur zur Modularisierung zu entwickeln, die es erlaubt, Datalog-Programme in mehrere, klar abgegrenzte Module aufzuteilen. Dabei soll sichergestellt werden, dass Typkonsistenz über Modulgrenzen hinweg gewährleistet wird und Namensräume eine saubere Trennung der Logik ermöglichen.

In dieser Arbeit wird untersucht, wie sich Datalog als *interne domänenspezifische Sprache (DSL) in Rust* implementieren lässt. Eine interne DSL ist in eine bestehende Hostsprache eingebettet. Der Vorteil interner DSLs liegt darin, dass sie direkt innerhalb der Hostsprache spezifiziert und verwendet werden können, inklusive Integration mit bestehendem Code der Hostsprache.

Konkret wird gezeigt, wie Datalog-Programme direkt innerhalb von Rust-Anwendungen mittels *prozeduraler Makros* beschrieben werden können. Dabei steht die *Modularisierbarkeit* im Fokus: Während sich viele bestehende Datalog-Systeme primär auf die Effizienz der Auswertung konzentrieren, wird in dieser Arbeit die Frage gestellt, wie sich Datalog-Programme so gestalten lassen, dass sie skalierbar, modular und typsicher sind – insbesondere dann, wenn sie über mehrere Makroaufrufe bzw. Dateien hinweg spezifiziert werden.

1.2 Einbettung in das Forschungsprojekt OPOG

Zur Einordnung: Die Arbeit ist eingebettet in das Forschungsprojekt *OPOG* – „*Op*timierung für und in Datal*og*“, in dessen Rahmen die Datalog-Engine *Tofino* entwickelt wird. Bei der in dieser Arbeit implementierten Lösung *Tofina* handelt es sich nicht um eine vollständige Datalog-Engine, sondern um das *Sprachfrontend* für Tofino. Tofina erlaubt es, modular definierte Datalog-Programme aus verschiedenen Makroaufrufen zu einem konsistenten Gesamtprogramm zusammenzusetzen, welches anschließend von der Tofino-Engine optimiert und ausgewertet wird.

Die Architektur der Datalog-Engine *Tofino* orientiert sich unmittelbar an den in OPOG verfolgten Forschungsschwerpunkten und ist klar in drei funktionale Phasen gegliedert: *Analyse*, *Optimierung* und *Evaluation*. Diese Dreiteilung erlaubt eine systematische Trennung der Zuständigkeiten und fördert sowohl Wiederverwendbarkeit als auch Erweiterbarkeit.

Die in dieser Arbeit entwickelte interne DSL *Tofina* bildet das Frontend der Engine und übernimmt die *Analysephase*. Hierbei werden Datalog-Programme, die innerhalb von Rust mittels prozeduraler Makros beschrieben werden, syntaktisch und semantisch analysiert. Das resultierende Programm wird in eine konsistente *Zwischendarstellung* überführt, die die logische Struktur, Typinformationen und Modulbeziehungen des ursprünglichen Datalog-Codes repräsentiert.

Im Anschluss an die Analyse wird diese Zwischendarstellung in der *Optimierungsphase* der Engine transformiert. Dabei kommen unter anderem Varianten der *Magic-Sets-Transformation* zum Einsatz, um die Effizienz der späteren Auswertung zu steigern. In der abschließenden *Evaluationsphase* wird das optimierte Programm schließlich ausgeführt.

Durch diese modulare Architektur kann Tofina als eigenständige Komponente entwickelt und weiterentwickelt werden, ohne direkt in die Optimierungs- oder Evaluierungslogik eingreifen zu müssen. Zugleich wird ein konsistenter Datenfluss zwischen den Phasen gewährleistet, sodass das in dieser Arbeit vorgestellte Sprachfrontend nahtlos in den Gesamtprozess der Engine eingebettet ist.

Im Unterschied zu verwandten Projekten wie *Crepe* oder *Ascent*, die komplette Datalog-Engines als interne DSLs in Rust realisieren – einschließlich Parsing und Auswertung – beschränkt sich Tofina bewusst auf das Sprachfrontend, bestehend aus syntaktischer und semantischer Analyse. Die Auswertung erfolgt durch die Tofino-Engine. Dadurch kann sich die Arbeit ganz auf Fragen der sprachlichen Gestaltung, Modularität und Typprüfung konzentrieren, ohne sich mit Aspekten der Evaluation beschäftigen zu müssen.

Im Gegensatz zu Ansätzen wie [1], in denen ein *funktionales Frontend* entwickelt wird, das nach Datalog kompiliert und anschließend von bestehenden Engines ausgeführt wird, um sowohl von deren Effizienz als auch von einer ergonomischen Sprachgestaltung zu profitieren, verfolgt diese Arbeit einen anderen Ansatz: *Tofina* bleibt vollständig bei Datalog als Basissprache. Der Fokus liegt darauf, Mechanismen zu entwickeln, mit denen Datalog-Programme *modular* in Rust beschrieben werden können. Das Frontend in dieser Arbeit deckt daher vor allem die *syntaktische und semantische Analyse* ab und führt die in verschiedenen Makros definierten Teilprogramme zu einem *globalen, konsistenten Datalog-Programm* zusammen. Die eigentliche Auswertung dieses Programms wird anschließend der Tofino-Engine überlassen.

1.3 Aufbau der Arbeit

Die Arbeit ist in sieben Abschnitte gegliedert, die inhaltlich aufeinander aufbauen – von den theoretischen Grundlagen über bestehende Systeme und technische Konzepte bis hin zur eigenen Implementierung.

Abschnitt 2 führt in die theoretischen Grundlagen der Logikprogrammierung ein und bildet damit das Fundament der Arbeit. Zunächst werden die zentralen Konzepte der deklarativen und logikbasierten Programmierung erläutert. Anschließend wird die Syntax und Semantik von Datalog vorgestellt, einschließlich der formalen Darstellung von Fakten, Regeln und deren Auswertung. Abschließend erfolgt ein Vergleich mit Prolog wobei insbesondere die Unterschiede in Bezug auf Terminierung herausgearbeitet wird. Damit wird deutlich, warum Datalog sich besonders für Anwendungsbereiche eignet, in denen garantierte Terminierung erforderlich sind.

Abschnitt 3 widmet sich den Anwendungsbereichen von Datalog und seiner Realisierung als domänenspezifische Sprache. Zunächst wird der Begriff der DSL eingeordnet und die Unterscheidung zwischen externen und internen DSLs erläutert. Anschließend wird aufgezeigt, dass viele Anwendungsdomänen Datalog um zusätzliche Sprachkonstrukte erweitern, wodurch eine Vielzahl unterschiedlicher Datalog-Dialekte entsteht. Diese Dialekte werden durch spezialisierte Engines realisiert. Im Anschluss werden exemplarisch drei weit verbreitete Datalog-Engines vorgestellt, die Datalog jeweils als externe DSL umsetzen: *Soufflé*, *CodeQL* sowie *Nemo*.

Abschnitt 4 legt den technischen Grundstein für die spätere Implementierung, indem es sich mit Rust-Makros befasst – einem zentralen Sprachfeature, das die Realisierung interner DSLs ermöglicht. Zunächst werden deklarative und prozedurale Makros erläutert. Im Anschluss werden zwei existie-

rende interne Datalog-DSLs in Rust vorgestellt: *Crepe* und *Ascent*. Beide Systeme nutzen prozedurale Makros, um Datalog-Programme innerhalb von Rust-Code zu beschreiben.

Abschnitt 5 behandelt das Konzept der Modularisierung. Zunächst wird Modularisierung aus Sicht der Softwaretechnik eingeordnet, ihre Ziele und Vorteile beschrieben sowie Prinzipien wie hohe Kohäsion, lose Kopplung und Lokalisierung von Entwurfsentscheidungen erläutert. Anschließend wird untersucht, wie Modularisierung in unterschiedlichen Programmierparadigmen realisiert wird, bevor der Fokus auf die Logikprogrammierung gelegt wird. Es werden existierende Modularisierungskonzepte in Prolog beschrieben sowie Ansätze zur Erweiterung von Datalog um objektorientierte Konzepte. Darauf aufbauend werden die Modularisierungsmechanismen von Soufflé und Ascent analysiert. Soufflé implementiert ein komponentenbasiertes System mit Vererbung und Überschreibung, während Ascent einen Mechanismus bereitstellt, um Programmfragmente zwischen Makros wiederzuverwenden, allerdings ohne Unterstützung für Namensräume

Abschnitt 6 bildet den Kern der Arbeit und präsentiert die Konzeption und Implementierung des `tofina!`-Makros. Der Abschnitt ist in zwei Hauptteile gegliedert. Zunächst wird die Basisimplementierung vorgestellt, die ohne Modularisierungsaspekte auskommt. Dabei wird die Gesamtprojektstruktur erläutert, der Aufbau der internen Zwischendarstellung des Datalog-Programms beschrieben und der Ablauf der Typinferenz und Typprüfung dargestellt. Im zweiten Teil wird die Implementierung um die Unterstützung von Modularität erweitert. Zunächst wird die Idee eines globalen Programms eingeführt, das die in verschiedenen Makros definierten Teilprogramme zusammenführt. Anschließend wird beschrieben, wie Prädikate als Schnittstellen zwischen Modulen verwendet werden können, um eine lose Kopplung zu ermöglichen. Ein durchgängiges Anwendungsbeispiel verdeutlicht den Nutzen dieser Struktur. Darauf folgt eine detaillierte Beschreibung der technischen Umsetzung: von der automatischen Erweiterung des globalen Programms über den Namespace-Mechanismus für Prädikate, der auf dem Modulsystem von Rust basiert, bis hin zur Kombination aus Compile-Zeit und Laufzeit der semantischen Analyse.

Abschnitt 7 schließt die Arbeit mit einer Zusammenfassung der erzielten Ergebnisse und einem Ausblick auf mögliche Weiterentwicklungen

2 Grundlagen der Logikprogrammierung

Datalog ist eine deklarative Programmiersprache aus dem Bereich der Logikprogrammierung, deren Syntax eine vereinfachte Version von Prolog darstellt. Im Folgenden werden zunächst die Prinzipien der *deklarativen Programmierung* erläutert, anschließend das Paradigma der *Logikprogrammierung* vorgestellt. Darauf aufbauend erfolgt eine Betrachtung von *Syntax und Semantik von Datalog*. Abschließend wird Datalog mit *Prolog* verglichen, um Gemeinsamkeiten und Unterschiede herauszuarbeiten.

Bei der deklarativen Programmierung liegt der Fokus darauf, *was* ein Programm tun soll, statt *wie* es dies im Detail umsetzt [2]. Im Gegensatz zu imperativen Sprachen, die Kontrollflüsse und Anweisungen vorgeben, beschreibt eine deklarative Sprache Zustände, Eigenschaften oder Beziehungen, die erfüllt sein müssen. Die konkrete Ausführung und Steuerung der Berechnung übernimmt dabei die zugrunde liegende Laufzeitumgebung oder Engine.

2.1 Logikprogrammierung als Paradigma

Datalog ist eine vereinfachte Variante der allgemeinen Logikprogrammierung [3]. In diesem Paradigma besteht ein Programm aus einer endlichen Menge von *Fakten* und *Regeln*. Fakten sind Aussagen über relevante Sachverhalte innerhalb einer Domäne, die als wahr angenommen werden. Regeln hin-

gegen erlauben die Ableitung neuer Fakten aus bestehenden, indem sie logische Beziehungen zwischen den Elementen beschreiben. Auf diese Weise lassen sich neue Erkenntnisse durch Schlussfolgerungen gewinnen, die auf dem bestehenden Wissensstand basieren.

Ein typisches Beispiel für den Einsatz von Datalog ist die Modellierung gerichteter Graphen. Solche Graphen lassen sich durch Fakten beschreiben, die die Existenz direkter Verbindungen zwischen Knoten ausdrücken, etwa: *von A nach B, von B nach C* usw. Mithilfe von Regeln kann darauf aufbauend die *Erreichbarkeit* zwischen Knoten definiert werden. Ein Knoten Y gilt dann als von einem Knoten X aus erreichbar, wenn entweder eine direkte Kante von X nach Y existiert oder ein weiterer Knoten Z existiert, der von X aus erreichbar ist und von dem eine direkte Verbindung zu Y besteht.

Formal werden Fakten und Regeln als sogenannte *Horn-Klauseln* dargestellt, welche folgende Form haben:

$$L_0 \vdash L_1, ..., L_n. \tag{2.1}$$

Jedes L_i ist ein sogenanntes *Literal*. Ein Literal ist entweder ein *Atom* oder die *Negation* eines Atoms. Ein Atom wiederum besteht aus einem *Prädikat* und einer festen Anzahl von *Termen*. Terme sind entweder *Konstanten* oder *Variablen*.

Eine Horn-Klausel gliedert sich in zwei Teile:
- L_0 wird als *Kopf* bezeichnet,
- $L_1, ..., L_n$ wird als *Rumpf* bezeichnet und enthält die Voraussetzungen, unter denen der Kopf gilt. Die Literale im Körper sind dabei implizit durch logisches UND ($\wedge$) verknüpft.

Ist der Körper leer, handelt es sich um einen Fakt. Enthält er mindestens ein Literal, spricht man von einer Regel.

2.2 Datalog als deklarative Logikprogrammiersprache

Datalog baut auf der allgemeinen Logikprogrammierung auf und definiert eine spezialisierte, deklarative Sprache mit klaren syntaktischen und semantischen Regeln.

Aufgrund unterschiedlicher Anwendungsbereiche haben sich verschiedene Dialekte von Datalog entwickelt, die sich in ihrer Syntax oder in Erweiterungen unterscheiden. Es gibt jedoch folgende Konventionen: Variablen werden mit Großbuchstaben geschrieben, während Konstanten und Prädikate klein geschrieben werden [2].

Ein zentrales Konzept in Datalog ist die Fähigkeit, rekursive Zusammenhänge zu formulieren. Besonders anschaulich lässt sich dies anhand von Graphstrukturen zeigen. Ein gerichteter Graph kann dabei durch Fakten beschrieben werden, die direkte Kanten zwischen Knoten angeben. Darauf aufbauend lassen sich Regeln formulieren, die aus diesen direkten Verbindungen weiterführende Beziehungen ableiten – etwa die Erreichbarkeit zwischen zwei Knoten. Eine solche Regel beschreibt die sogenannte *transitive Hülle* des Graphen:

```datalog
edge(a, b).
edge(b, c).
edge(c, d).

reachable(X, Y) :- edge(X, Y).
reachable(X, Y) :- edge(X, Z), reachable(Z, Y).
```

Dieses Beispiel verdeutlicht, wie mit einfachen, logisch strukturierten Regeln komplexe Beziehungen, insbesondere rekursive, modelliert werden können.

In Datalog müssen alle Literale mit demselben Prädikat die gleiche Anzahl von Termen besitzen, die sogenannte *Stelligkeit* (Arity) des Prädikats.

Um sicherzustellen, dass ein Datalog-Programm terminiert und keine unendlichen Faktenmengen generiert, gelten folgende Bedingungen [3]:
- Jeder Fakt darf ausschließlich Konstanten enthalten.
- Jede Variable, die im Kopf einer Regel vorkommt, muss auch im Körper derselben Regel verwendet werden.

Semantisch lässt sich eine Datalog-Klausel durch eine Formel der Prädikatenlogik erster Ordnung beschreiben [2]. Die Literale im Körper werden dabei mit einem logischem UND verknüpft und implizieren das Literal im Kopf. Formal lässt sich die Klausel aus Regel (2.1) folgendermaßen ausdrücken:

$$L_1 \wedge ... \wedge L_n \rightarrow L_0$$

Die Auswertung eines Datalog-Programms erfolgt mittels einer Bottom-up-Strategie, auch Fixpunktberechnung genannt. Dabei werden iterativ neue Fakten aus den vorhandenen Fakten und Regeln abgeleitet, solange neue Fakten entstehen können. Dieser Prozess endet, wenn keine weiteren Fakten mehr ableitbar sind, wodurch garantiert wird, dass die Auswertung terminiert.

Im Folgenden wird Datalog mit Prolog verglichen.

2.3 Vergleich mit Prolog

Prolog steht für „*Pro*grammieren in *Log*ik“ und ist eine der ältesten und bedeutendsten Programmiersprachen der Logikprogrammierung [4, 5].

Datalog ist eng mit Prolog verwandt, teilt viele grundlegende Prinzipien und syntaktische Elemente, unterscheidet sich jedoch in wesentlichen Punkten, die Einfluss auf die Anwendbarkeit und Auswertung haben.

Syntaktisch basieren beide Sprachen auf der Logikprogrammierung und verwenden Horn-Klauseln zur Darstellung von Fakten und Regeln. Die Syntax von Datalog ist sogar eine Teilmenge der Prolog-Syntax, jede Menge von Datalog Klauseln kann von einem Prolog Interpreter geparsed und ausgeführt werden.

Datalog und Prolog unterscheiden sich jedoch in ihrer *Semantik* [3]. Datalog wurde bewusst so eingeschränkt, dass Programme immer terminieren und nur eine endliche Menge an Fakten erzeugen. Prolog kennt solche Garantien nicht und kann in Endlosschleifen laufen. Prolog nutzt ein top-down-orientiertes Auswertungsverfahren, bei dem die Reihenfolge der Regeln und Literale im Regelkörper die Ausführung beeinflussen kann. Datalog hingegen basiert auf einer rein deklarativen Semantik mit einer Bottom-up-Fixpunktberechnung, bei der die Reihenfolge der Regeln in Datalog keine Rolle spielt. In Datalog darf jeder Fakt nur Konstanten enthalten, und alle Variablen im Kopf einer Regel müssen auch im Körper vorkommen. Prolog kennt solche syntaktischen Beschränkungen nicht.

Diese Eigenschaften machen Datalog besonders gut geeignet für Anwendungen, in denen eine vollständige und terminierende Auswertung wichtig ist, wie in Datenbankabfragen und Analysewerkzeugen.

3 Anwendungsbereiche von Datalog und externen DSLs

Bei Datalog handelt es sich um eine *domänenspezifische Sprache (DSL)*. Im Gegensatz zu Allzweckprogrammiersprachen wie Rust, Java oder Python sind DSLs auf einen bestimmten Anwendungsbereich – eine sogenannte Domäne – zugeschnitten. Das ermöglicht eine kompaktere, verständlichere und oft

auch effizientere Ausdrucksweise. Typische Beispiele für DSLs sind etwa SQL für Datenbankabfragen oder HTML für die Strukturierung von Webseiten. Bei domänenspezifischen Sprachen unterscheidet man grundsätzlich zwei Hauptansätze [6]: *Externe DSLs* sind eigenständige Sprachen mit eigener Syntax, Semantik und gegebenenfalls eigenen Werkzeugen wie IDEs oder Compilern. *Interne DSLs* werden innerhalb einer allgemeinen Programmiersprache implementiert und nutzen deren Syntax und Laufzeitumgebung. Die Sprache, in der die DSL eingebettet ist, wird als *Host-Sprache* bezeichnet.

Typische Anwendungsgebiete von Datalog sind unter anderem Datenbankabfragen, insbesondere wenn rekursive Abfragen benötigt werden, bei denen klassischen SQL-Lösungen an ihre Grenzen stoßen [3]. Ein weiteres zentrales Einsatzfeld ist die Programmanalyse, etwa zur Durchführung von Datenfluss-, Kontrollfluss- und Zeigeranalysen sowie zur Untersuchung statischer Code-Strukturen oder sicherheitsrelevanter Eigenschaften [2, 7]. Auch im Bereich der Netzwerktechnik findet Datalog Anwendung: Im sogenannten *declarative networking* lassen sich Netzwerkprotokolle durch Regln modellieren, die rekursive Beziehungen zwischen Knoten in Netzwerken beschreiben [8]. Darüber hinaus wird Datalog in der Künstlichen Intelligenz eingesetzt, insbesondere in regelbasierten, argumentativen Systemen, um effizient relevante Schlussfolgerungen aus Regeln erster Ordnung zu ziehen [9].

Je nach Anwendungsdomäne wird Datalog nicht nur als Sprache verwendet, sondern häufig an die jeweilige Domäne angepasst oder erweitert – etwa um Negation, Typannotationen, Aggregationsfunktionen sowie arithmetische und Vergleichsoperatoren. Auf diese Weise entstehen verschiedene Datalog-Dialekte, die meist durch spezialisierte Datalog-Engines realisiert werden. Im Folgenden werden Datalog-Engines vorgestellt, die eigene Datalg-Dialekte als externe oder interne DSLs realisieren.

3.1 Soufflé

Soufflé ist eine leistungsfähige Datalog-Engine, die ursprünglich für die statische Codeanalyse entwickelt wurde [7, 10]. Inzwischen wird sie auch in anderen Bereichen eingesetzt, etwa im Reverse Engineering, in der Netzwerkanalyse und der Datenanalyse. Soufflé stellt sowohl einen Interpreter als auch einen Compiler zur Verfügung. In beiden Fällen werden Datalog-Programme aus `.dl`-Dateien eingelesen. Der Interpreter eignet sich für kleinere Programme und ermöglicht eine direkte Ausführung ohne Zwischenschritte. Für größere oder rechenintensivere Programme ist der Compiler besser geeignet: Er übersetzt das Datalog-Programm in optimierten, parallelisierten C++ Code, der effizient auf Multi-Core-Systemen ausgeführt werden kann. Der Kompilierungsvorgang ist zwar aufwändiger, ermöglicht jedoch eine Verbesserung der Laufzeitleistung bei komplexeren Analysen.

Die Hauptelemente eines Soufflé-Programms sind Direktiven, Relationen, Fakten und Regeln. *Direktiven* sind spezielle Steueranweisungen, die mit einem Punkt beginnen. Sie beeinflussen das Verhalten des Programms, etwa hinsichtlich Ein- und Ausgabe. *Relationen* werden mit der Direktive `.decl` deklariert. Dabei wird das Prädikat gemeinsam mit den Typen seiner Argumente angegeben. Zum Beispiel: `.decl edge(x: number, y: number)`. Relationen können mit den Direktiven `.input` und `.output` als Eingabe- bzw. als Ausgaberelation kennzeichnen und können nach entsprechend vor bzw. nach der Auswertung importiert oder exportiert werden – etwa: `.input edge` und `.output reachable`.

Soufflé ist statisch typisiert und erfordert die explizite Angabe von Typen in jeder Relationsdeklaration. Diese Typisierung wird bereits zur Kompilier- oder Interpretationszeit geprüft, was eine frühzeitige Erkennung von Fehlern ermöglicht. Es werden die folgenden primitiven Datentypen unterstützt: `number`, `unsigned`, `float` und `symbol`. Mit der Direktive `.type` lassen sich neue Typen definieren. So können *Alias-Typen* erstellt werden, die sich identisch zu ihrem Basistyp verhalten und überall dort verwendet werden können, wo auch der ursprüngliche Typ zulässig ist – beispielsweise: `.type myNumber = number`. Darüber hinaus unterstützt Soufflé *Subtypen*, die mit

`.type <Subtype> <: <Basetype>` deklariert werden. Subtypen können überall dort eingesetzt werden, wo auch der Basistyp erlaubt ist – umgekehrt jedoch nicht. Dies ermöglicht wie in Codebeispiel 1 gezeigt eine präzisere Typprüfung und hilft dabei, logische Widersprüche frühzeitig zu erkennen.

```
1  .type even <: number                                    soufflé
2  .type odd <: number
3
4  .decl A(x: even)
5  .decl B(x: odd)
6
7  A(x) :- B(x).       // Fehler: Typkonflikt
```

Codebeispiel 1: *Verwendung von Subtypen in Soufflé.* In diesem Beispiel werden zwei disjunkte Subtypen des Basistyps `number` definiert: `even` und `odd`. Die Relationen `A` und `B` verwenden jeweils einen dieser Subtypen in ihrer Typdefinition. Die Regel `A(x) :- B(x).` führt zu einem Typfehler, da `x` in beiden Relationen denselben Typ haben muss – hier jedoch inkompatible Subtypen verwendet werden. Solche Konflikte können frühzeitig erkannt und Fehler in der Programmspezifikation aufgedeckt werden.

Mit *Union Types* lassen sich – ähnlich wie bei Enums – mehrere Subtypen zu einem gemeinsamen Typ kombinieren: `.type <new-union> = <subtype-1> | <subtype-2> | ... | <subtype-k>`. Die enthaltenen Typen müssen dabei denselben Basistyp besitzen. Für komplexere Datenstrukturen unterstützt Soufflé sogenannte *Record Types*: `.type <new-record> = [ <name_1>: <type_1>, ..., <name_k>: <type_k> ]`. Diese sind vergleichbar mit Structs und ermöglichen das Bündeln mehrerer Werte, die unterschiedliche – auch rekursive – Typen haben können. Damit lassen sich strukturierte Informationen innerhalb eines einzelnen Records modellieren.

Soufflé erweitert Datalog um verschiedene funktionale Konstrukte, die die Ausdrucksstärke der Sprache deutlich erhöhen. Dazu gehören arithmetische und textuelle Funktionen, Vergleichsoperatoren, Negation, Einschränkungen (*Constraints*) sowie Aggregatsfunktionen. Ein zentrales Konzept bei diesen Erweiterungen ist das *Grounding*: In jeder Regel müssen alle Variablen *grounded* sein – das heißt, sie müssen mindestens einmal als Argument eines positiven Prädikats im Rumpf der Regel vorkommen. Nur so ist garantiert, dass die Auswertung endlich bleibt und das Programm terminiert. *Arithmetische Ausdrücke* wie `x + 1` oder `x * y` dürfen nur verwendet werden, wenn alle beteiligten Variablen eingeschränkt sind – also auf einen endlichen Wertebereich begrenzt. Andernfalls besteht die Gefahr unendlicher Rekursion, wie etwa bei den beiden Regeln `A(1)` und `A(i + 1) :- A(i)`, wobei `A` ohne Einschränkung die gesamte Menge der natürlichen Zahlen $\mathbb{N}$ annimmt. *Einschränkungen* sind Ausdrücke im Rumpf einer Regel, die einen Wahrheitswert liefern. Dafür unterstützt Soufflé die Vergleichsoperatoren `>`, `<`, `=`, `!=`, `>=` und `<=`. Über solche Constraints lassen sich Variablen effektiv beschränken. So kann etwa durch eine angepasste Regel wie `A(i + 1) :- A(i), i < 10` sichergestellt werden, dass das Programm terminiert. *Aggregatsfunktionen* stehen zur Verfügung, mit denen sich Werte innerhalb eines Datalog-Programms zusammenfassen lassen. Dazu zählen `min`, `max`, `sum` und `count`. Auch `symbol`-Werte können mit Funktionen wie `cat`, `ord` oder `strlen` verarbeitet werden.

Darüber hinaus ist Soufflé durch benutzerdefinierte Funktionen erweiterbar, die von Entwickler:innen in C oder C++ implementiert werden können. Diese Funktionen werden in einer gemeinsam genutzten Bibliothek gespeichert, die zur Auswertungszeit geladen wird.

3.2 CodeQL und Nemo

CodeQL ist eine weitere Anwendung einer Datalog-Engine im Bereich der statischen Codeanalyse [11]. Dabei wird Quellcode in relationale Fakten umgewandelt, die anschließend mit einer deklarativen Abfragesprache analysiert werden. Die verwendete Sprache QL wurde ursprünglich von Semmle entwickelt und ist heute Teil von GitHub bzw. Microsoft. CodeQL wird unter anderem von GitHub genutzt, um im Rahmen von CI/CD-Pipelines automatische statische Codeanalysen durchzuführen und so frühzeitig potenzielle Fehler oder Sicherheitsprobleme zu identifizieren.

Ein weiteres Beispiel für eine externe Datalog-Engine ist *Nemo*, entwickelt an der Technischen Universität Dresden und vollständig in Rust implementiert [12–14]. Im Gegensatz zu Soufflé verfolgt Nemo einen reinen interpreterbasierten Ansatz: Datalog-Programme werden zur Laufzeit aus `.rls`-Dateien eingelesen und vollständig im Arbeitsspeicher ausgewertet – Nemo ist somit eine reine *In-Memory-Engine*.

Nemo erlaubt die Verwendung von *Internationalized Resource Identifiers* (IRIs) als Konstanten innerhalb von Relationen – vergleichbar mit symbolischen Namen in klassischen Datalog. Darüber hinaus bietet Nemo eine Vielzahl integrierter Funktionen, insbesondere für Zeichenkettenverarbeitungen (`REGEX`, `CONTAINS`, `LCASE`), arithmetische Berechnungen (`+`, `-`, `POW(x, y)`, `BITOR`$(x_1, \ldots, x_n)$) sowie Typkonvertierung (`INT`, `DOUBLE`, `FLOAT`, `IRI`). Über die Direktiven `@import` und `@export` lassen sich Daten in verschiedenen Formaten wie z.B. `csv` laden bzw. speichern.

Eine Besonderheit liegt in der hohen Typenflexibilität: Werte beliebiger Typen können an jeder Stelle eines Ausdrucks verwendet werden – ohne feste Typvorgaben. Regeln werden dabei nur auf jene Fakten angewendet, für die die beteiligten Ausdrücke typkompatibel sind. Ein Beispiel dazu ist in Codebeispiel 2 dargestellt.

```nemo
mydata(a, b) .
mydata("hello", 42) .

resultA(?N + 10) :- mydata(_, ?N) .
resultB(?D) :- mydata(?X, _), ?D = DATATYPE(?X) .
```

Codebeispiel 2: *Beispiel für typflexible Relation in Nemo.* In diesem Beispiel werden zwei Fakten für `mydata` definiert. Der erste nutzt abstrakte Konstanten (die intern als IRIs interpretiert werden), der zweite verwendet einen String und eine Ganzzahl. Die Regel `resultA` addiert `10` zum zweiten Argument aus `mydata`. Die Regel `resultB` bestimmt den Datentyp des ersten Arguments mithilfe der integrierten Funktion `DATATYPE`. Variablen werden in Nemo durch ein vorangestelltes Fragezeichen (`?`) gekennzeichnet. Die Auswertung führt in Nemo zu folgenden Ergebnissen: `resultA` enthält den einzelnen Fakt `resultA(52)`, da nur der Ausdruck `?N + 10` nur für die Ganzzahl `42` definiert ist. `resultB` enthält zwei Fakten mit den Typen `http://www.w3.org/2001/XMLSchema#anyURL` (für `a`) und `http://www.w3.org/2001/XMLSchema#string` (für `"hello"`). Diese Typbezeichner entsprechen den internen XML-Schema-Repräsentationen von Nemo für IRIs und Strings.

4 Makros in Rust – Grundlage interner DSLs

Es existieren bereits Ansätze, Datalog direkt als *interne DSL in Rust* umzusetzen. Insbesondere die Projekte *Crepe* und *Ascent* zeigen, wie sich mithilfe von Makros Datalog-Programme elegant im Rust-Code formulieren lassen. Beide Varianten verwenden Rust-Makros zur Definition von Fakten und Regeln. Welche Arten von Makros Rust bereitstellt und wie diese technisch funktionieren, wird im Folgenden erläutert.

4.1 Arten von Makros in Rust

Die folgenden Inhalte basieren auf den offiziellen Rust-Dokumentationen [15, 16]. Makros in Rust ermöglichen es, zur Compile-Zeit Code zu generieren. Damit lassen sich wiederkehrende Muster abstrahieren, neue syntaktische Konstrukte schaffen und domänenspezifische Sprachkonstruktionen (DSLs) realisieren.

In Rust gibt es zwei Hauptarten von Makros: *deklarative Makros*, auch als `macro_rules!`-Makros bekannt, und *prozedurale Makros*. Deklarative Makros basieren auf Muster- beziehungsweise Strukturerkennung von Code und ersetzen diesen durch neuen Code. Prozedurale Makros arbeiten auf Tokenstreams als Eingabe: Sie nehmen Code entgegen, verarbeiten ihn und generieren einen neuen Tokenstream als Ausgabe. Prozedurale Makros lassen sich in drei Typen unterteilen:

- *Benutzerdefinierte `#[derive]`-Makros*, die verwendet werden, um mittels des `derive`-Attributs an Structs und Enums deren Funktionalität zu erweitern,
- *Attribut-ähnliche Makros*, mit denen benutzerdefinierte Attribute erstellt werden können, die an beliebigen Elementen, wie Funktionen, verwendet werden können und in der Anwendung nicht auf Structs und Enums beschränkt sind,
- *Funktions-ähnliche Makros*, die wie ein Funktionsaufruf aussehen und mit den als Argumente übergebenen Token operieren.

Deklarative Makros funktionieren ähnlich wie ein Rust-`match`-Ausdruck: Dieser nimmt einen Wert entgegen und vergleicht ihn mit Mustern (Patterns). Bei deklarativen Makros handelt es sich bei dem Wert um Rust-Quellcode, der mit Codestrukturen abgeglichen wird. Wenn der übergebene Code mit einer verglichenen Struktur übereinstimmt, wird er durch den mit dem Muster verknüpften Code ersetzt. Dabei werden keine konkreten Typen oder Werte verglichen, wie es bei einem `match`-Ausdruck der Fall ist, sondern beispielsweise, ob der übergebene Code ein Ausdruck (Expression), eine Anweisung (Statement) oder ein Typ ist. Deklarative Makros werden in Rust häufig verwendet, um Code eleganter und kürzer auszudrücken, wie etwa das `vec!`-Makro: Hier kann man `vec![1, 2, 3]` schreiben, was intern durch die Deklaration eines Vektors mit mehreren `push`-Aufrufen ersetzt wird, und schließlich der Vektor zurückgegeben wird.

Prozedurale Makros müssen in einem gesonderten *Crate* definiert werden. In Rust ist ein Crate die kleinste Codeeinheit, die der Compiler zu einer Zeiteinheit betrachtet. Crates können Module enthalten, die in verschiedenen Dateien definiert werden können und gemeinsam kompiliert werden. Es gibt zwei Formen von Crates: *Binary Crates*, die Programme mit einer `main`-Funktion sind und zu ausführbaren Dateien compiliert werden, und *Libary Crates*, die keine `main`-Funktion enthalten, sondern Funktionalitäten bereitstellen, die in mehreren Projekten wiederverwendet werden können. Ein *Package* bündelt ein oder mehrere Crates und stellt eine Sammlung von Funktionalitäten dar. Ein Package kann beliebig viele Binary Crates enthalten, aber höchstens ein Libary Crate. Zudem enthält ein Package eine Konfigurationsdatei, die `Cargo.toml` heißt und Informationen über das Package beinhaltet. Um prozedurale Makros zu definieren, muss in der `Cargo.toml`-Datei angegeben werden, dass es sich bei dem Crate im Package um `proc-macro` Crates handelt. In Codebeispiel 3 wird der Aufbau einer solchen Datei gezeigt. Es können mehrere prozedurale Makros in einem Crate definiert werden, jedoch enthält ein Package mit `proc-macro` nur ein Crate, da es sich dabei um ein Libary Crate handelt.

```toml
Cargo.toml
[package]
name = "tofina_macros"
…
[lib]
proc-macro = true
…
[dependencies]
syn = "2.0"
quote = "1.0"
…
```

Codebeispiel 3: *Konfigurationsdatei `Cargo.toml` eines prozeduralen Makros.* Im `[package]`-Abschnitt werden grundlegende Metadaten wie der Name des Packages definiert. Im `[lib]`-Abschnitt gibt `proc-macro = true` an, dass es sich um ein `proc-macro`-Crate handelt. Unter `[dependencies]` werden die externen Abhängigkeiten angegeben – typischerweise `syn` und `quote`, die zum Parsen und Generieren von Rust-Code verwendet werden.

Um ein prozedurales Makro zu definieren, wird eine Funktion mit einem Attribut versehen, das angibt, um welche Art von prozeduralem Makro es sich handelt. Diese Funktion nimmt als Eingabe einen `TokenStream` entgegen und gibt ebenfalls einen `TokenStream` zurück. Ein `TokenStream` ist eine Sequenz von Tokens, also Codefragmenten, die vom Makro verarbeitet werden.

Benutzerdefinierte `derive`-Makros werden verwendet, um Structs und Enums um zusätzliche Funktionalitäten zu erweitern. Dabei wird ein Makro definiert, das ein `Trait` – also eine Art Interfaces in Rust – für eine bestimmte Funktionalität implementiert. Dieses Makro wird dann per Annotation mit dem `derive`-Attribut an das Struct oder Enum angehängt. Um ein benutzerdefiniertes `derive`-Makro zu definieren, wird eine Funktion mit der Annotation `#[proc_macro_derive(<Name des derive>)]` erstellt. Im Funktionsrumpf wird dann die Logik implementiert, die für das Struct oder Enum generiert wird. So kann eine Funktionalität einmal definiert und an mehreren Datenstrukturen durch einfaches Angeben von `#[derive(<Name des derive>)]` verwendet werden, anstatt das Trait manuell mehrfach zu implementieren. Zum Beispiel wird das `derive`-Makro `Hash` mittels `#[derive(Hash)]` vor die Definition einer Datenstruktur geschrieben und implementiert automatisch das Hash-Trait, ohne dass die Entwickler:innen diese Logik selbst schreiben müssen. Bei der Implementierung prozeduraler Makros wird häufig das Crate `syn` verwendet, um den eingelesenen `TokenStream` in eine AST-Struktur (Abstract Syntax Tree) zu überführen, auf der dann gearbeitet wird. Mit dem Crate `quote` wird der manipulierte AST anschließend zurück in einen `TokenStream` übersetzt.

Attribut-ähnliche Makros ähneln benutzerdefinierten `derive`-Makros, generieren aber keinen Code für das `derive`-Attribut. Stattdessen werden sie verwendet, um neue Attribute zu erstellen, die auch an anderen Elementen, etwa Funktionen, angewendet werden können. In der Definition des Attribut-ähnlichen Makros wird eine Funktion mit `#[proc_macro_attribute]` annotiert. Diese Funktion erhält zwei Parameter: `attr: TokenStream`, der Inhalt des annotierten Attributs, und `item: TokenStream`, der Rumpf des Elements, an das das Attribut gehängt ist, zum Beispiel eine Funktionssignatur mit Rumpf. Ansonsten funktionieren sie wie benutzerdefinierte `derive`-Makros: Sie operieren auf dem AST des TokenStreams und generieren daraufhin neuen Code.

Funktions-ähnliche Makros definieren Makros, die wie Funktionsaufrufe aussehen. Die Makro-Definition wird mit dem Attribut `#[proc_macro]` versehen, nimmt einen `TokenStream` als Eingabe entgegen und gibt einen `TokenStream` als Ausgabe zurück. Innerhalb des Makros wird auf diesem TokenStream gearbeitet und daraus neuer Code generiert – ähnlich wie bei `macro_rules!`-Makros. Allerdings

werden hier nicht nur die Struktur des Eingabecodes mit Mustern abgeglichen und ersetzt, sondern die Eingabe wird innerhalb des Makros mit Rust-Code manipuliert. So kann beispielsweise das Trait `parse` aus dem `syn`-Crate verwendet werden, um zu definieren, wie eine bestimmte Datenstruktur geparst wird. Mit der Implementierung des Traits `ToTokens`, der die Funktion `to_tokens` bereitstellt und einen `TokenStream` zurückgibt, wird festgelegt, welcher Rust-Code aus der eigenen Datenstruktur generiert werden soll. Somit ist es möglich mit prozeduralen Makros unter Implementierung Traits `Parse` und `ToToken` festzulegen, wie eine Datenstruktur benutzerdefiniert geparst wird und daraus Code generiert wird. Dadurch kann eine benutzerdefinierte Syntax geparst und eine DSL in Rust erstellt werden. Zum Beispiel kann ein Makro definiert werden, das eine SQL-ähnliche DSL parst und Rust-Code generiert. Ein Aufruf eines solchen Funktions-ähnlichen Makros mit benutzerdefinierter SQL-Syntax könnte folgendermaßen aussehen: `sql!(SELECT * FROM users WHERE id=1);`. Die Definition des Makros ist in Codebeispiel 4 dargestellt:

```rust
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
    let ast = parse_macro_input!(input as SqlAst);
    quote! { #ast }.into()
}
```

Codebeispiel 4: *Definition eines Funktions-ähnlichen Makros.* Die Funktion `sql` ist mit `#[proc_macro]` annotiert und definiert damit ein prozedurales Makro. Sie nimmt ein `TokenStream` als Eingabe entgegen, der mittels `parse_macro_input!(input as SqlAst)` in eine benutzerdefinierte Datenstruktur (`SqlAst`) geparst wird. Dafür muss für `SqlAst` der Trait `Parse` mit der `parse`-Funktion implementiert sein. In der `parse`-Funktion erfolgt eine syntaktische Analyse, bei der die Tokens z.B. als Schlüsselwörter oder Identifikatoren klassifiziert werden. Jedes Token enthält einen *Span*, der positionsbezogene Informationen enthält und für präzise Fehlermeldungen genutzt werden kann. Abschließend wird über `quote!` die `to_tokens`-Implementierung von `SqlAst` aufgerufen, um den Ziel-Code zu generieren. Da `syn` und `quote` mit `proc_macro2` arbeiten – einem Wrapper für `proc_macro` –, muss das Ergebnis mit `.into()` in einen `proc_macro::TokenStream` konvertiert werden, um den erwarteten Rückgabewert des Makros zu entsprechen.

Makros bieten Möglichkeiten, die über reguläre Funktionen hinausgehen. Während Funktionen zur Laufzeit ausgeführt werden, agieren Makros bereits zur Compile-Zeit: Sie erhalten den Quellcode als Tokenstream, verarbeiten ihn und generieren daraus neuen Code, noch bevor der Compiler Typsystem oder Semantik analysiert. Im Gegensatz zu Funktionen, die eine feste Signatur mit einer klaren Anzahl und Typisierung von Parametern besitzen, können Makros eine variable Anzahl an Eingaben verarbeiten. Da sie auf der Tokenebene operieren, ist ihre Eingabeform deutlich flexibler und syntaktisch vielseitiger. Prozedurale Makros ermöglichen es beispielsweise, automatisch Methoden oder Trait-Implementierungen für benutzerdefinierte Datentypen zu generieren – eine Fähigkeit, die regulären Funktionen nicht zur Verfügung steht. Allerdings ist die Definition solcher Makros auch komplexer, da sie selbst aus Code bestehen, der anderen Code erzeugt.

4.2 Interne Datalog DSLs mittels Rust-Makros

Crepe ist eine Datalog-Engine, vollständig in Rust implementiert. Sie realisiert Datalog als interne DSL über das prozedurale Makro `crepe!` [17] und unterstützt grundlegende Funktionalitäten wie *semi-naive Evaluation* und *stratifizierte Negation*. Ein Datalog-Programm wird innerhalb eines Rust-Makros formuliert und erzeugt im Hintergrund entsprechende Datenstrukturen und Evaluationslogik in Rust.

In Crepe werden Relationen im Makro direkt als native *Rust-Structs* deklariert – etwa mit den Rust-Typen `char`, `i32` oder `String`. Über Annotationen wie `@input` und `@output` wird festgelegt, ob

für die jeweilige Relation Ein- bzw. Ausgabefunktionalitäten generiert werden sollen. Innerhalb der Regelrümpfe sind auch arithmetische Ausdrücke zulässig. Die Datalog-Regeln werden in ein Rust-Programm übersetzt, das sämtliche für die Evaluation benötigten Strukturen und Logik automatisch generiert.

Das `crepe!`-Makro erzeugt dabei für jede deklarierte Relation ein entsprechendes Rust-`struct`, auf das automatisch verschiedene Traits (`Copy`, `Clone`, `Eq`, `Hash`) mittels `#[derive(...)]` angewendet werden. Zusätzlich wird ein zentrales Struct namens `Crepe` generiert. Dieses enthält für jede `@input`-Relation ein Feld, dessen Typ ein `Vec<T>` ist – wobei `T` das jeweils deklarierte Relations-Struct repräsentiert. Somit verwaltet das `Crepe`-Struct die Eingabedaten für das Datalog-Programm. Zur Auswertung wird für das `Crepe`-Struct eine private Methode `run` erzeugt, welche alle im Makro definierten Regeln in Rust-Code implementiert und ausführt. Das Ergebnis – d.h. die Ausgabefakten aller mit `@output` annotierten Relationen – wird als Tupel zurückgegeben.

Auf diese Weise lassen sich Datalog-Regeln im Makro deklarativ formulieren und aus einer regulären Rust-Anwendung heraus mit Daten befüllen und auswerten. Die komplette Ausführungslogik wird durch das Makro automatisch erzeugt – inklusive aller Datenstrukturen und der Implementierung der Datalog-Evaluation. Ein Beispiel für die Berechnung der transitiven Hülle in Crepe ist in Codebeispiel 5 dargestellt.

```rust
use crepe::crepe;

crepe! {
    @input
    struct Edge(i32, i32);

    @output
    struct Reachable(i32, i32);

    Reachable(x, y) <- Edge(x, y);
    Reachable(x, z) <- Edge(x, y), Reachable(y, z);
}

fn main() {
    let mut runtime = Crepe::new();
    runtime.extend([Edge(1, 2), Edge(2, 3), Edge(3, 4)]);
    let (reachable,) = runtime.run();
    assert!(reachable.contains(&Reachable(1, 4)))
}
```

Codebeispiel 5: *Nutzung des `crepe!`-Makros zur Berechnung der transitiven Hülle.* In diesem Beispiel wird im `crepe!`-Makro eine Eingaberelation `Edge(i32, i32)` deklariert, welche gerichtete Kanten zwischen Knoten beschreibt, sowie eine Ausgaberelation `Reachable(i32, i32)` zur Darstellung der Erreichbarkeitsrelation. Anschließend werden zwei Regeln definiert: Eine Basisregel, die jede direkte Kante als erreichbar markiert, und eine rekursive Regel, die die transitive Hülle bildet, indem sie Erreichbarkeit über Zwischenschritte schließt. In der `main`-Funktion wird eine neue Instanz des automatisch generierten `Crepe`-Structs erzeugt und in der Variable `runtime` gespeichert. Da `Edge` im Makro mit `@input` annotiert wurde, ist für das `Crepe`-Struct der `Extend`-Trait für Edge implementiert. Dadurch kann die `extend`-Methode verwendet werden, um eine Liste von `Edge`-Instanzen an `runtime` anzuhängen. Anschließend wird über `runtime.run()` die Datalog-Evaluation gestartet, wobei alle definierte Regeln ausgeführt und die mit `@output` markierten Relationen berechnet werden. Das Ergebnis ist in diesem Fall ein Tupel mit der einzigen Ausgabe-Relation `Reachable`. Abschließend wird per `assert!` überprüft, ob ein Pfad von Knoten `1` nach `4` berechnet wurde.

Ascent geht über die Funktionalität von Crepe hinaus und stellt nicht nur ein einzelnes, sondern mehrere Makros zur Nutzung von Datalog in Rust bereit [18, 19]. Das Makro `ascent!` ist funktional vergleichbar mit dem `crepe!`-Makro: Es definiert Relationen und Regeln und generiert aus den Deklarationen ausführbaren Rust-Code.

Darüber hinaus bietet Ascent mit dem Makro `ascent_run!` eine Variante, die direkt beim Aufruf ausgeführt wird. Der zentrale Vorteil gegenüber `ascent!` liegt darin, dass `ascent_run!` Zugriff auf lokale Variablen des umgebenden Rust-Codes hat. So können externe Variablen innerhalb des Makroaufrufs verwendet werden, was eine flexiblere Einbindung in Rust-Anwendung ermöglicht. Ergänzend erlaubt Ascent mit dem Makro `ascent_run_par!` auch parallelisierte Ausführungen. Dieses generiert parallelen Rust-Code zur effizienten Evaluation der Datalog-Regeln auf Multi-Core-Systemen.

Ein weiteres zentrales Merkmal von Ascent ist das Prinzip *BYODS (Bring Your Own Data Structures to Datalog)* [20]. Während Datalog-Relationen in den meisten Engines intern mit festgelegten, nicht konfigurierbaren Datenstrukturen umgesetzt werden, erlaubt Ascent Entwickler:innen explizit anzu-

geben, mit welcher Datenstruktur eine Relation Implementiert werden soll. Wie in Codebeispiel 6 dargestellt.

```rust
use ascent_byods_rels::trrel_uf;
ascent! {
    relation edge(i32, i32);

    #[ds(trrel_uf)] // Makes the relation transitive
    relation path(i32, i32);

    path(x, y) <-- edge(x, y);
}
```

Codebeispiel 6: *Beispielanwendung von BYODS in Ascent.* In diesem Beispiel wird mithilfe der Relationen `edge` und `path` die transitive Hülle definiert. Die Relation `edge` wird zunächst mit den Typen `(i32, i32)` deklariert. Anschließend folgt die Relation `path`, die über das Attribut `#[ds(trrel_uf)]` annotiert ist. Die dabei verwendete Datenstruktur `trrel_uf` stammt aus dem Crate `ascent_byods_rels` und implementiert eine transitive Relation basierend auf einer Union-Find-Struktur. Durch diese Annotation genügt eine einzige Regel `path(x, y) <-- edge(x, y);`, um `path` automatisch transitiv zu machen – eine explizite rekursive Regel (z.B. `path(x, z) <-- path(x, y), path(y, z);`) ist nicht erforderlich.

5 Modularität in DSLs: Anforderungen und Konzepte

Bei der Entwicklung umfangreicher Datalog-Programme – etwa im Kontext statischer Codeanalyse – stellt sich schnell die Frage, wie sich komplexe Logikprogramme sinnvoll strukturieren lassen. Insbesondere bei internen (Datalog-)DSLs, die mithilfe von proc-macros in Rust realisiert werden, ergibt sich dabei eine grundlegende technische Herausforderung: Rust-Makros operieren isoliert. Jeder Makroaufruf verarbeitet – unabhängig von anderen – ausschließlich seinen eigenen Eingabetokenstream und generiert daraus zur Compile-Zeit Rust-Code. Ein Zugriff auf Informationen aus anderen Makros ist dabei nicht möglich.

Diese Isolation erschwert die Umsetzung modularer DSLs, bei denen Informationen aus verschiedenen Modulen zusammengeführt und typkonsistent verarbeitet werden sollen. Es fehlt an einer gemeinsamen Sicht auf das globale Gesamtprogramm – etwa auf definierte Regeln oder Typen aus anderen Modulen.

Diese Problematik zeigt sich exemplarisch bei der internen Datalog-DSL Crepe: Hier wird das gesamte Datalog-Programm innerhalb eines einzigen Makroaufrufs spezifiziert. Das bedeutet, dass sämtliche Fakten, Regeln und Prädikate in einem einzigen Makroaufruf – uns somit in einer einzigen Datei – definiert werden müssen. Eine solche monolithische Struktur verhindert die sinnvolle Trennung logisch abgegrenzter Einheiten und schränkt die Wiederverwendbarkeit, Wartbarkeit und insbesondere die Skalierbarkeit deutlich ein.

Im Folgenden wird zunächst erläutert, was unter Modularisierung zu verstehen ist. Darauf aufbauend werden unterschiedliche Ansätze zur Modularisierung in bestehenden Datalog-Engines vorgestellt, insbesondere anhand der Beispiele Soufflé und Ascent.

5.1 Modularität – eine Einordnung

Modularität ist ein Prinzip aus der Softwaretechnik und dem Systems Engineering. Ziel modularer Systeme ist es, ein komplexes Gesamtsystem in kleinere, überschaubare und klar abgegrenzte Einheiten – sogenannte *Module* – zu zerlegen. Diese Zerlegung dient vor allem der Erfüllung *nicht-funktionaler Anforderungen* wie Wartbarkeit, Testbarkeit, Erweiterbarkeit und Wiederverwendbarkeit.

Der Begriff *Modul* leitet sich vom lateinischen *modulus* ab, was so viel bedeutet wie „kleine Maßeinheit“ oder allgemein „Einheit“. In technischen Disziplinen bezeichnet ein Modul eine abgegrenzte Einheit innerhalb eines größeren Systems. Diese begriffliche Grundlage findet sich auch in der Softwaretechnik wieder, wird jedoch je nach Kontexten unterschiedlich interpretiert.

Oft wird *Modul* synonym mit Begriffen wie Komponente, Einheit oder Subsystem verwendet. Allgemein versteht man darunter eine Einheit, die aus mehreren Komponenten besteht, spezifische Schnittstellen besitzt und einem oder mehreren nicht-funktionalen Zielen dient.

In [21] wird vorgeschlagen, Module nicht als Teilprogramme, sondern als organisatorische Verantwortungsbereiche zu verstehen. In dieser Auffassung isoliert ein Modul Designentscheidungen, insbesondere solche, die sich mit hoher Wahrscheinlichkeit in Zukunft ändern könnten. Ein Modul trägt demnach die Verantwortung für eine spezifische Aufgabe, deren Implementierung sich unabhängig von anderen Modulen anpassen lässt. Ziel ist es, Änderungen lokal vorzunehmen, ohne unbeabsichtigt Auswirkungen auf andere Systemteile zu riskieren.

Der Fokus von Modulen liegt auf der strukturellen und organisatorischen Funktion zur Erhöhung der Entwurfsflexibilität – nicht notwendigerweise einer konkreten funktionalen Aufgabe. Funktionale Anforderungen werden durch Systeme, Teilsysteme und Systemelemente erfüllt. Module sind hingegen durch definierte Schnittstellen von anderen Systemteilen abgegrenzt.

Gute Modularität basiert auf mehreren zentralen Prinzipien. Eines davon ist das sogenannte *Localization of Design Decisions* – die gezielte Isolation von Entwurfsentscheidungen, insbesondere solcher, die sich wahrscheinlich ändern werden. Dabei wird nicht spezifiziert, wo genau diese Entscheidungen implementiert werden sollen, sondern lediglich, dass sie nicht über mehrere Komponenten hinweg verteilt oder redundant realisiert werden dürfen. Verstöße gegen dieses Prinzip führen zu dupliziertem Code und erschweren somit die Wartbarkeit des Systems [22].

Ergänzend dazu das Prinzip der *High Cohesion* (hohe Kohäsion), welches zusammengehörige Verantwortlichkeiten innerhalb eines Moduls bündelt. Eine Komponente sollte idealerweise nur eine klare Aufgabe erfüllen. So wird die Verständlichkeit erhöht und Änderungen erfolgen lokal.

Das Prinzip der *Low Coupling* (lose Kopplung) steht dem gegenüber. Hierbei wird angestrebt, dass die Abhängigkeiten zu anderen Modulen möglichst gering gehalten werden. Explizite Abhängigkeiten sollen im Code sichtbar und nachvollziehbar sein, implizite oder unnötige Abhängigkeiten werden vermieden.

Ein Weiteres Prinzip ist das *Modular Reasoning* – modulares Denken. Es besagt dass Entwickler:innen in der Lage sein sollten, ein Modul vollständig zu verstehen, indem sie nur dessen eigene Implementierung sowie die öffentlichen Schnittstellen referenzierter Module betrachten. Die internen Details anderer Module sollten hierfür nicht bekannt sein müssen.

Die Anwendung modularer Designprinzipien verfolgt vorrangig das Ziel, verschiedene nicht-funktionale Eigenschaften eines Systems zu verbessern. Zu den typischen Vorteilen modularer Systeme zählen [21, 23]:

- *Wartbarkeit*: Änderungen können lokal durchgeführt werden, ohne unbeabsichtigte Auswirkungen auf andere Teile des Systems.
- *Testbarkeit*: Einzelne Module lassen sich isoliert testen, was die Erkennung von Fehlerquellen erleichtert.
- *Wiederverwendbarkeit*: Einmal entwickelte Module können in anderen Systemen oder Kontexten über Schnittstellen wiederverwendet werden.
- *Verständlichkeit*: Eine klare Modulstruktur reduziert die Komplexität und erleichtert das Verständnis des Gesamtsystems.
- *Parallele Entwicklung*: Teams können unabhängig an getrennten Modulen arbeiten, was die Entwicklungszeit verkürzt.
- *Flexibilität*: Systeme lassen sich leichter an unterschiedliche Anforderungen oder Produktvarianten anpassen.

Diese Vorteile ergeben sich meist dann, wenn die Prinzipien wie hohe Kohäsion, lose Kopplung und Isolation von Designentscheidungen konsequent umgesetzt werden. Jedoch ist wichtig hervorzuheben, dass Modularität diese Eigenschaften nicht garantiert, sondern lediglich begünstigt. Ein modular aufgebautes System kann dennoch schwer wartbar oder schlecht testbar sein, wenn die zugrundeliegenden Prinzipien nicht sorgfältig angewendet wurden. Umgekehrt erschwert das Fehlen modularer Strukturen die Erreichung dieser Ziele erheblich.

In [23] wird darauf hingewiesen, dass Modularität nicht immer zur Verbesserung der Funktionalität eines Systems beiträgt. Sie kann unter Umständen sogar negative Auswirkungen auf Performance oder Komplexität haben. Zum Beispiel erhöht sich die Systemkomplexität durch Schnittstellen. Langfristig führen ihre Vorteile jedoch oft indirekt zu einer Verbesserung funktionaler Eigenschaften durch erhöhte Testbarkeit und Erweiterbarkeit.

Die Konzepte *Modularität*, *Abstraktion* und *Verkapselung* sind zentrale Prinzipien in der Softwareentwicklung. Deren konkrete Ausprägung unterscheiden sich jedoch deutlich zwischen verschiedenen *Programmierparadigmen* [22]. Zwar existieren gemeinsame Ziele, jedoch variieren sowohl die Umsetzung als auch die zugrundeliegende Konzepte.

In der *objektorientierten Programmierung* ist Modularisierung stark mit der Kapselung von Daten und Verhalten innerhalb von Klassen verbunden. Module werden hier durch Sprachmittel wie *Packages*, *Klassen* oder *Objekte* unterstützt. Die Prinzipien der hohen Kohäsion werden durch Methoden unterstützt. Zugriffsmodifikationen wie `private`, `protected` oder `public` erlauben Kontrollmechanismen über die Sichtbarkeit und unterstützen so die Abstraktion und Verkapselung.

Funktionale Programmierung unterstützt Modularisierung durch den Einsatz von reinen Funktionen ohne Seiteneffekte. Die Funktionen hängen hierbei nur von den Eingabewerten ab und sind können unabhängig getestet, verstanden und wiederverwendet werden. Die Abwesenheit von geteiltem Zustand reduziert Kopplung, während starke Typisierung und Funktionstransparenz Kohäsion fördert.

Bei der *Logikprogrammierung* stehen Prädikate, Regeln und Fakten zur Verfügung. Diese betonen Verhaltensabstraktion und Datenabstraktion, bieten jedoch nur eingeschränkte Mittel zur Modularisierung und Verkapselung. Dennoch kann Modularität erreicht werden, indem jedes Prädikat eine einzelne Idee oder Verantwortung repräsentiert (hohe Kohäsion), jede potenziell veränderliche Designentscheidung klar einer Menge von Regeln zugeordnet ist (lokalisierte Entscheidungen). Ein besonderer Vorteil der Logikprogrammierung ist die ausgeprägte Kontrollabstraktion: Die zugrundeliegende Ausführung bleibt weitgehend verborgen und erlaubt eine deklarative Spezifikation des Systemverhaltens.

5.2 Modularität in der Logikprogrammierung

Bereits in Prolog gibt es Modularisierungskonzepte. In [24] wird beschrieben, dass Standard Logikprogrammierung – basierend auf Horn-Klauseln – nicht für die Programmierung im großen Stil geeignet ist. Dafür wird für λProlog ein Modularisierungskonzept vorgestellt, das auf der Trennung von Namensräumen (Scoping mittels Namespaces) basiert. Dabei werden Programme in einzelne Module unterteilt, die separat kompiliert werden können. Module verwalten hier eigene Namen und Prozeduren und können andere Module importieren und deren Prozeduren zur Laufzeit nutzen. Um Namenskonflikte zu vermeiden, verwendet das System beim Importieren eine sogenannte Renaming-Funktion, die dafür sorgt, dass Konstanten aus importierten Modulen kontextabhängig umbenannt werden. So wird sichergestellt, dass Module konfliktfrei geladen werden können.

Auch in Datalog gibt es Modularisierungsversuche, wie etwa in [25] beschrieben. Dabei werden Konzepte aus der objektorientierten Programmierung – insbesondere *Vererbung* – auf Datalog übertragen. Eine Besonderheit liegt darin, dass Definitionen in untergeordneten Modulen (bzw. *Subklassen*) nicht zwangsläufig die gleichnamigen Definitionen aus übergeordneten Modulen (*Superklassen*) überschreiben, sondern die Vererbung als Erweiterung eingesetzt wird: Mehrere Regeln, die dasselbe Prädikat referenzieren können koexistieren und die Bedeutung dieses Prädikats definieren.

CodeQL ist ein beispiel für einen Objektorientierten Datalog-Dialekt: Hier werden Klassen erstellt welche sich in Modulen befinden und auf mehrere `.ql`- bzw. `qll`-Dateien für Bibliotheken verteilen lassen.

Das Konzept der Vererbung ist auch in Soufflé umgesetzt. Dabei können Komponenten von einer oder mehreren Super-Komponenten erben. Wobei die Elemente der Super-Komponente an die Sub-Komponente weitergegeben werden. Komponenten sind in Soufflé ein Mechanismus zur Modularisierung realisiert. Komponenten erlauben es, Soufflé-Programme in wiederverwendbare Einheiten zu gliedern, die Relationen, Typdefinitionen, Fakten und Regeln enthalten können. Programme können über mehrere Dateien hinweg strukturiert und über `.include`-Anweisungen eingebunden werden. Relationen können von Sub-Komponenten Komponenten überschrieben werden, wenn diese in einer Super-Komponente als `overridable` deklariert ist.

Eine Verwendung von Komponenten in Soufflé ist in Codebeispiel 7 dargestellt.

```
soufflé
.comp MyComponent {
  .type myType = number
  .decl TheAnswer(x:myType)    // component relation
  TheAnswer(42).               // component fact
}

.init myInstance1 = MyComponent
myInstance1.TheAnswer(33).

.decl Test(x:number)
Test(x) :- myInstance1.TheAnswer(x).
.output Test
```

Codebeispiel 7: *Deklaration und Initialisierung von Komponenten in Soufflé.* Eine Komponente wird mit `.comp` deklariert und erhält eine Typdefinition für `myType`, eine Relation-Deklaration, die dem Prädikat `TheAnswer` den Typ `myType` zuweist und einen Fakt mit dem Wert `42`. Instanzen von Komponenten werden über `.init` erzeugt und erhalten einen Namen sowie die Zuweisung zu einer Komponente.

In Codebeispiel 8 wird gezeigt wie Soufflé Komponenten Initialisierungen intern verarbeitet.

```
soufflé
.type myInstance1.myType = number
.decl myInstance1.TheAnswer(x:myType)    // relation of myInstance1
myInstance1.TheAnswer(42).               // fact of myInstance1
myInstance1.TheAnswer(33).

.decl Test(x:number)
Test(x) :- myInstance1.TheAnswer(x).
.output Test
```

Codebeispiel 8: *Interne Darstellung von Komponenten in Soufflé.* Intern expandiert die Soufflé-Engine Instanzen von Komponenten in eigenständige Namensräume. Dabei werden alle in der Komponente definierten Typen, Fakten und Regeln mit dem Präfix des Instanz-Namens versehen, um Namenskonflikte zu vermeiden.

Komponenten in Soufflé können verschachtelt sein, das heißt: Sie können weitere innere Komponenten enthalten. Regeln und Fakten, die innerhalb einer verschachtelten Komponente definiert werden, deren Prädikate jedoch in einer äußeren Komponente deklariert sind, werden von Soufflé intern direkt in die äußere Komponente verschoben. Die innere Komponente verhält sich dabei wie eine lokale Erweiterung der äußeren, ohne eigene Namensräume zu erstellen.

Soufflé unterstützt *generische Programmierung* durch die Parametrisierung von Komponenten mit Typ-Platzhaltern. Dabei wird in der Komponenten-Deklaration kein konkreter Typ verwendet, sondern ein Platzhalter. Erst bei der Initialisierung der Komponente wird dieser Platzhalter durch einen konkreten Typ ersetzt. Dies ermöglicht es, generische, wiederverwendbare Komponenten zu definieren, die in verschiedenen Kontexten mit unterschiedlichen Typen verwendet werden können. Dadurch wird die Code-Wiederverwendbarkeit erhöht und Code-Duplizierung vermieden.

Auch in der internen Datalog-DSL Ascent existieren erste Ansätze zur Modularisierung. Dafür stellt Ascent verschiedene Makros zur Verfügung, unter anderem `ascent_source!` und `include_source!`. Mit `ascent_source!` lassen sich wiederverwendbare Codefragmente definieren, die sich in anderen Modulen mithilfe von `include_source!` einbinden lassen.

Dabei wird im `ascent_source!`-Makro dem Codefragment ein Name zugewiesen. Dieser Name kann später über einen Pfad im `include_source!`-Makro referenziert werden, um das entsprechende Fragment in ein Ascent-Programm einzubinden.

In Codebeispiel 9 wird gezeigt, wie `ascent_source!` und `include_ascent!` zusammen verwendet wird, um Ascent modular in Rust zu nutzen.

```rust
mod base {
    ascent::ascent_source! { tc:
        relation edge(Node, Node);
        relation path(Node, Node);
        path(x, y) <-- edge(x, y);
        path(x, z) <-- edge(x, y), path(y, z);
    }
}

fn main() {
    type Node = usize;
    let res = ascent::ascent_run! {
        include_source!(base::tc);
        edge(1, 2), edge(2, 3), edge(3, 4);
    };
    assert!(res.path.contains(&(1, 4)));
}
```

Codebeispiel 9: *Modularität in Ascent mittels `ascent_source!`- und `include_source!`-Makros.* Im Modul `base` werden in einem `ascent_source!`-Makro Relationen und Regeln für die Transitive Hülle unter dem Namen `tc` (transitive closure) definiert. Unter Angabe eines Pfads zu `tc` kann das Codefragmente anschließend im `include_source!`-Makro eingebunden werden.

Dieser Mechanismus ermöglicht eine Form der Modularisierung, insbesondere im Hinblick auf Wiederverwendbarkeit.

Allerdings werden dabei keine separaten Namensräume erzeugt, alle eingebundenen Prädikate liegen in dem selben Namensraum. Werden in zwei eingebundenen `ascent_source!`-Fragmenten gleichnamige Prädikate mit unterschiedlichen Typen verwendet, führt dies zur Compile-Zeit zu einem Namenskonflikt.

6 Tofina: Eine interne Datalog-DSL in Rust

In diesem Kapitel wird die eigene Implementierung des Frontends der Tofino-Engine vorgestellt. Damit ist die Analysephase gemeint, die mithilfe eines Proc-Makros realisiert wurde. Zwar liegt der Fokus der Arbeit auf dieser Komponente, zur Einordnung wird jedoch auch die Gesamtarchitektur der Engine erläutert – bestehend aus Analyse, Optimierung und Ausführung.

Die Darstellung gliedert sich in zwei Abschnitte: Zunächst wird die Basisimplementierung ohne Modularisierungsaspekte beschrieben, anschließend folgt die Erweiterung um Modularisierungsaspekte.

Zu Beginn wird der Aufbau der Engine beschrieben sowie Struktur der internen Zwischendarstellung des verarbeiteten Datalog-Programms.

Im Anschluss wird der Aufbau des `tofina!`-Proc-Makros detailliert dargestellt. Der Fokus liegt dabei auf der Analysephase innerhalb des Makros, die syntaktische und semantische Analyse umfasst – insbesondere Typinferenz und grundlegende Typprüfung – sowie die anschließende Codegenerierung.

Der zweite Abschnitt widmet sich der Modularisierung. Zunächst wird konzeptionell beschrieben, wie ein modulares Datalog-System aussehen könnte. Anhand eines konkreten Beispiels wird die Relevanz modularer Strukturen veranschaulicht. Im Anschluss wird erläutert, wie der bisherige Ansatz

um Modularisierungsaspekte erweitert wurde und wie bestehende Herausforderungen gelöst werden konnten – insbesondere durch importierte Prädikate und Prüfmechanismen, die weder zur Compile-Zeit noch zur regulären Laufzeit vollständig greifen.

6.1 Basisimplementierung ohne Modularität

Im Rahmen des *OPOG*-Forschungsprojekts wurde eine dreiteilige Architektur für die Datalog-Engine *Tofino* entwickelt, bestehend aus Analyse, Optimierung und Ausführung. Diese Struktur orientiert sich an klassischen Compiler-Architekturen und erlaubt eine klare Trennung der funktionalen Phasen der Programmausführung.

Im Gegensatz zu anderen Implementierungen interner Datalog-DSLs in Rust – etwa *Crepe* und *Ascent* – übernimmt das Makro in *Tofino* ausschließlich die Rolle eines Sprach-Frontends. Während bei Crepe und Ascent das Makro sowohl das Parsen als auch die Generierung ausführbarer Strukturen integrieren, beschränkt sich `tofina!`, ein Proc-Macro zur Verarbeitung des im Rahmen dieser Arbeit entwickelten Datalog-Dialekts Tofina, auf das Parsen des Datalog-Programms und die Erzeugung einer Rust-Datenstruktur, die das Programm in Form eines *abstrakten Syntaxgraphen* (ASG) repräsentiert.

Das Makro generiert also nicht die Funktionen oder Logik zur Weiterverarbeitung oder Ausführung des Programms. Die Optimierung und Evaluation des ASG erfolgen vollständig außerhalb des Makros – zur Laufzeit und unabhängig von der Makroexpansion.

Die Entkopplung erlaubt eine saubere Trennung von Compile-Zeit-Parsing und Laufzeitverarbeitung und bietet Flexibilität für spätere Erweiterungen.

Die Architektur der Engine ist in Abbildung 1 dargestellt:

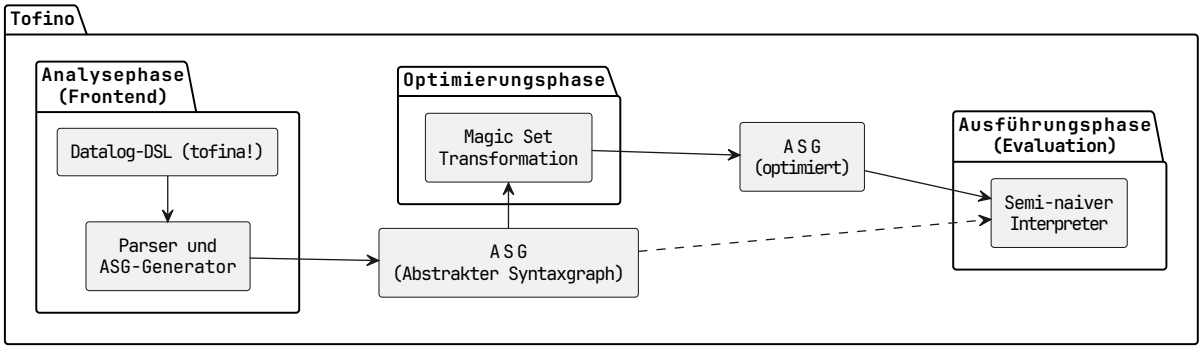

Abbildung 1: *Architektur der Datalog-Engine Tofino.* Die Engine besteht aus drei Phasen: *Analyse*, *Optimierung* und *Ausführung*. In der Analysephase wird über das Makro `tofina!` ein Datalog-Programm geparst und in einen abstrakten Syntaxgraphen (ASG) überführt. Der ASG dient als zentrale Zwischendarstellung und wird in der anschließenden Optimierungsphase weiterverarbeitet, etwa durch Anwendung der Magic-Set-Transformation. In der Ausführungsphase erfolgt die Evaluation des (ggf. optimierten) Programms. Optional kann der ASG auch direkt ohne Optimierung ausgeführt werden (gestrichelte Linie).

Die Abbildung zeigt, wie der ASG als zentrale Zwischendarstellung zwischen den drei Phasen der Engine fungiert.

Die konkrete Struktur der Zwischendarstellung ist in Abbildung 2 dargestellt. Der abstrakte Syntaxgraph (ASG) beschreibt die interne Repräsentation eines Datalog-Programms in Tofino.

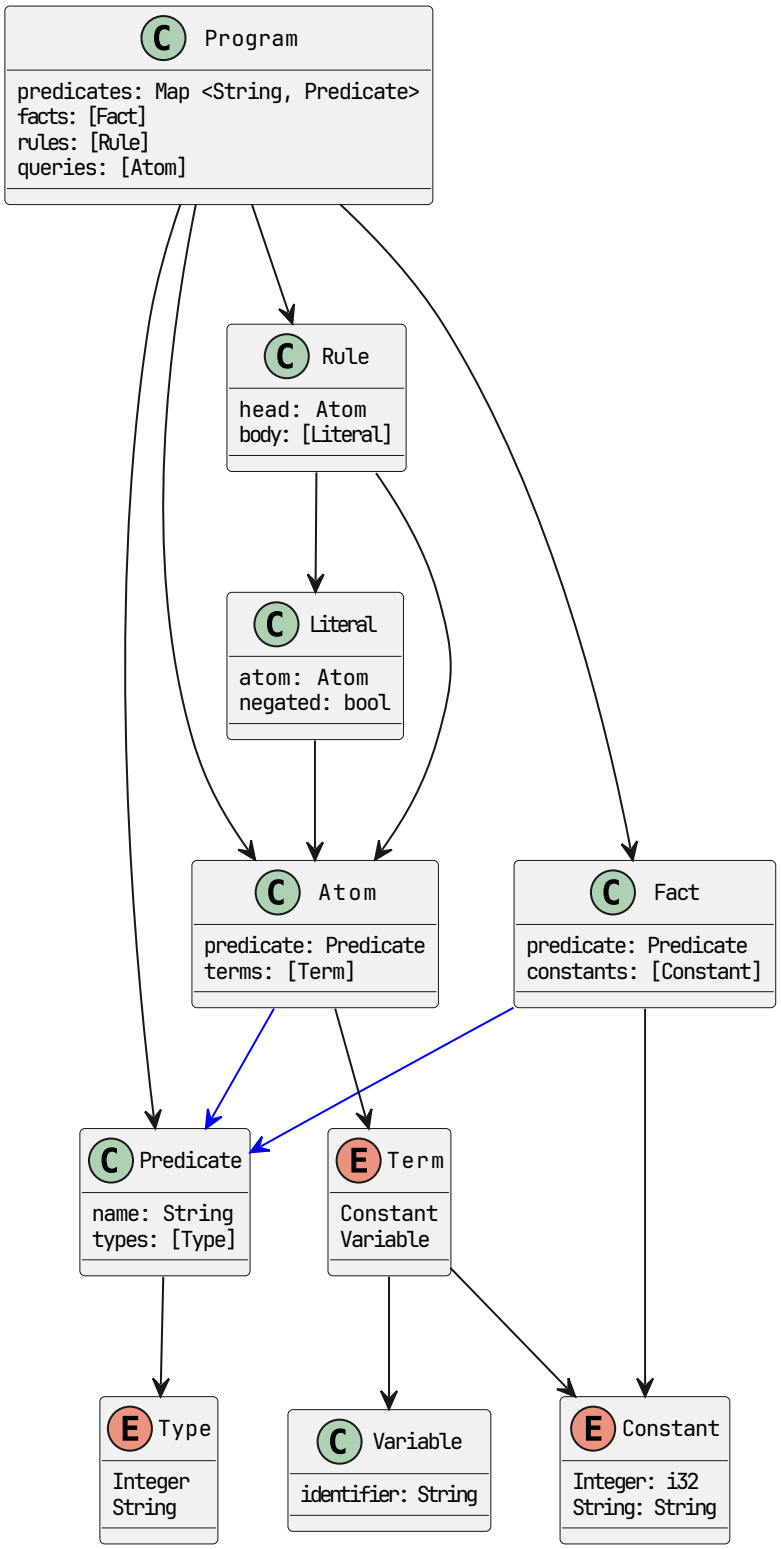

Abbildung 2: *Abstrakter Syntaxgraph (ASG) in Tofino.* Die Abbildung zeigt die Struktur des abstrakten Syntaxgraphen (ASG) als zentrale Zwischendarstellung eines Datalog-Programms in Tofino. Ein `Program` besteht aus einer Menge von `Fact`-, `Rule`- und `Atom`-Elementen sowie einer `Map`, die Prädikatsnamen (`String`) auf `Predicate`-Instanzen abbildet. Die Linien zwischen den Typen stellen strukturelle Beziehungen dar. Blaue Pfeile kennzeichnen referenzielle Verbindungen, bei denen mehrere Elemente auf dieselbe `Predicate`-Instanz verweisen können. Attribute mit eckigen Klammern wie `[Fact]` oder `[Term]` bezeichnen Listen von Elementen.

Ein `Program` besteht aus einer Menge von Regeln, Fakten und Abfragen sowie einer Map, die allen im Programm verwendeten Prädikatsnamen jeweils eine `Predicate`-Instanz zuordnet.

Der Regelkopf wird durch ein einzelnes `Atom` repräsentiert, der Regelrumpf durch eine Liste von `Literal`. Ein `Literal` ist dabei ebenfalls ein `Atom`, jedoch ergänzt um ein Negations-Flag.

Sowohl `Atom` als auch `Fact` enthalten eine Referenz auf eine `Predicate`-Instanz. Während `Atom` zusätzlich eine Liste von `Term` enthält – also entweder `Constant` oder `Variable` –, besteht ein `Fact` ausschließlich aus Konstanten, also konkreten Werten. Diese können derzeit vom Typ `i32` oder `String` sein.

Die Signatur eines `Predicate` besteht in Tofino nicht nur aus dem Namen und der Stelligkeit (Arity), sondern aus einer Liste von Typen, aus der sich die Arity ableiten lässt. Unterstützt werden derzeit die Typen `Integer` und `String`. Die Anzahl und Reihenfolge der Terme bzw. Konstanten in `Atom` und `Fact` müssen mit der Typenliste des referenzierten Prädikats übereinstimmen.

Die Abbildung zeigt außerdem, dass einzelne `Predicate`-Instanzen mehrfach verwendet werden können. So verweist beispielsweise sowohl der Fakt `edge(1, 2)` als auch die Regel `reachable(x, y) :- edge(x, y)` auf dieselbe `Predicate`-Instanz `edge`.

Diese geteilte Nutzung einer Instanz wird in der Implementierung durch sogenannte Smartpointer wie `Rc<T>` (Reference Counted, für Single-Threading) oder `Arc<T>` (Atomically Reference Counted, für Multi-Threading) realisiert. Sie ermöglichen *multiple ownership*, ohne dass das referenzierte Objekt dupliziert werden muss.

Gerade diese referenziellen Verbindungen unterscheiden den ASG von einem klassischen abstrakten Syntaxbaum (AST), wie er typischerweise im Compilerbau verwendet wird. In einem AST besitzt jeder Knoten genau einen Elternknoten. Im dargestellten ASG hingegen kann ein Knoten – konkret eine `Predicate`-Instanz – von mehreren Komponenten gleichzeitig verwendet werden.

Zwar verweisen im Diagramm mehrere Komponenten – etwa `Rule` und `Literal` – auf den Typ `Atom`, doch handelt es sich bei den konkreten Vorkommen im Regelkopf und im Rumpf jeweils um eigene Instanzen. Diese Mehrfachverwendung ist daher rein strukturell und nicht referenziell – und begründet nicht den Unterschied zum AST.

Um sicherzustellen, dass die Anzahl und Reihenfolge der Terme bzw. Konstanten in `Atom` und `Fact` mit der Typenliste des jeweils referenzierten Prädikats übereinstimmen, wird nach dem erfolgreichen Parsen des Datalog-Programms eine *semantische Analyse* durchgeführt. Dabei werden zwei zentrale Aufgaben erfüllt, die *Typinferenz*, bei der fehlende Typinformationen automatisch aus dem Kontext abgeleitet werden, sowie die anschließende *Typprüfung*, die sicherstellt, dass alle Prädikate konsistent und korrekt verwendet werden.

Diese Schritte erfolgen innerhalb der Makroexpansion nach Abschluss der syntaktischen Analyse – also nachdem alle Regeln, Fakten und Abfragen syntaktisch erkannt, aber noch vor der Codegenerierung in ASG-Strukturen überführt wurden.

Die Tofina-DSL verwendet ein Typisierungssystem, das auf *implizieter, aber statisch aufgelöster Typisierung* basiert: Typinformationen müssen beim Schreiben des Programms nicht expliziet angegeben werden, sofern sie sich *aus dem Kontext zuverlässig ableiten lassen.*

Die gesamte Typinferenz geschieht dabei zur Compile-Zeit im Makro selbst – ohne dass es zur Laufzeit zu dynamischen Typumwandlungen (Castings) oder Typprüfung kommt. Das bedeutet, dass keine manuelle Typannotation erforderlich ist, es entstehen keine Laufzeitkosten, und Fehlermeldungen zur Typkonsistenz erfolgen bereits beim Kompilieren.

Damit unterscheidet sich unser Ansatz von bestehenden Datalog-Dialekten wie Crepe, Ascent oder auch externen Tools wie Soufflé oder Nemo, bei denen Typen der Prädikate explizit anzugeben sind.

Der Ablauf der Typinferenz erfolgt in mehreren Schritten, die im Folgenden erläutert werden:

1. *Ableitung aus Fakten*
   Fakten enthalten ausschließlich Konstanten, deren Typen direkt bestimmt werden können. Beispielsweise kann aus dem Fakt `edge(1, 2)` ergibt sich für `edge` das Prädikat: `Predicate { name: "edge", types: [Integer, Integer] }`

2. *Typableitungen aus Konstanten in Regeln*
   Auch in Regeln können Konstanten vorkommen, aus denen Typen direkt abgeleitet werden. Beispiel: in `ancestors_of_alice("Alice", Y) <- parent("Alice", Z), ancestor(Z, Y)` liefert `"Alice"` den Typ `"String"` für Position 0 von `ancestors_of_alice` und `parent`.

3. *Typ-Constraints aus Variablen*
   Variablen liefern keine direkten Type, verknüpfen aber Positionen in verschiedenen Prädikaten. Beispiel: `reachable(X, Y) <- edge(X, Z), reachable(Z, Y)` in dieser Regel treten die Variablen `X`, `Y` und `Z` mehrfach auf, wobei eine Variable in einer Regel von genau einem Typ ist. Daraus ergeben sich folgende *Constraints* (Einschränkungen):
   - aus `X` ergibt sich `reachable[0] == edge[0]`,
   - aus `Y` ergibt sich `reachable[1] == reachable[1]` (trivial)
   - und aus `Z` ergibt sich `edge[1] == reachable[0]`.

   Insgesamt ergeben sich Gruppen von Argumentpositionen, deren Typen übereinstimmen müssen: `reachable[0] == edge[0] == edge[1]`. Jede dieser Positionen wird dabei durch ein Tupel der Form `(Prädikatsname, Artumentindex)` dargestellt, z.B. `("edge", 1)` für das zweite Argument von `edge`. Alle Positionen, die über gemeinsame Variablen miteinander verbunden sind, werden zu einer Gruppe zusammengefasst. Diese Gruppen werden intern als Mengen solcher Tupel repräsentiert und dienen in einem späteren Schritt der Typ-Propagation (Typverbreitung) über alle verknüpfunten Positionen hinweg.

4. *Typ-Propagation über verknüpfte Positionen*
   Nach dem Sammeln aller Typconstraints werden diese ausgewertet. Für jede Gruppe verknüpfter Positionen wird geprüft, ob mindestens eine der Positionen bereits über einen konkreten Typ verfügt. Ist dies der Fall, wird dieser Typ auf alle übrigen Positionen innerhalb der Gruppe übertragen. Ist der Typ jedoch wiedersprüchlich zu einem bereits bestehenden Typ an einer anderen Position, wird ein Fehler ausgegeben. Beispiel: wenn `edge[0] = Integer` ist und `reachable[0]` über eine gemeinsame Variable `X` mit `edge[0]` verbunden ist, so wird `reachable[0]` ebenfalls auf `Integer` gesetzt.

5. *Finale Prüfung auf vollständige Typisierung*
   Nach Abschluss der Typverteilung wird für jede Position in jedem Prädikat überprüft, ob ein Typ abgeleitet werden konnte. Ist dies nicht der Fall, wird ein Kompilierfehler ausgegeben, der angibt, für welche Prädikatsposition keine Typableitung möglich war.

Nach Abschluss der Typinferenz werden alle extrahierten Informationen in konkrete Strukturen des ASG überführt.

Dabei ist zu beachten, dass ein und dasselbe Prädikat innerhalb eines Programms an mehreren Stellen referenziert werden kann. Bei der Codegenerierung muss daher sichergestellt werden, dass alle Verwendungen tatsächlich dieselbe Instanz referenzieren.

Um dies zu ermöglichen, wird nicht unmittelbar eine `Program`-Instanz erzeugt, sondern stattdessen eine *Block Expression* generiert – also ein Rust-Codeblock `{ ... }`, der mehrere Anweisungen enthal-

ten kann und den Wert des letzten Ausdrucks (sofern kein Semikolon gesetzt ist) als Rückgabewert annimmt.

Innerhalb dieses Blocks wird für jedes Prädikat zunächst eine lokale Variable erzeugt. Diese kann anschließend an allen relevanten Stellen im Programm wiederverwendet werden. Auf diese Weise wird sichergestellt, dass alle Referenzen auf ein Prädikat tatsächlich auf dieselbe Instanz zeigen, ohne dass redundante Instanzen erzeugt oder Typinformationen erneut angegeben werden müssen.

In Codebeispiel 10 ist der generierte Rust-Code dargestellt, der die ASG-Struktur für ein einfaches Datalog-Programm repräsentiert.

```rust
{
  let pred_edge = Rc::new(Predicate {
    name: "edge".to_string(),
    types: vec![Type::Integer, Type::integer]
  });

  Program {
    predicates: {
      let mut map = HashMap::new();
      map.insert(pred_edge.name.clone(), Rc::clone(&pred_edge));
      map
    },
    facts: HashSet::from([
      Rc::new(Fact {
        predicate: Rc::clone(&pred_edge),
        constants: vec![Constant::Integer(1), Constant::Integer(2)],
      }),
    ]),
…                                    …
  }
}
```

Codebeispiel 10: *Generierte Zwischendarstellung für ein einfaches Datalog-Programm.* Das Beispiel zeigt den erzeugten Rust-Code für ein Programm, das aus einem einzigen Fakt `edge(1, 2)` besteht. Zunächst wird eine lokale Variable für das Prädikat `edge` erzeugt, anschließend wird diese Instanz an allen relevanten Stellen wiederverwendet. Die Markierung verdeutlichen, wo die Prädikat-Instanz erzeugt und wo sie im weiteren Verlauf referenziert wird. Für die Darstellung wurde hier `Rc` verwendet, alternativ wäre auch `Arc` möglich.

Bereits mit der bisherigen Implementierung ist es möglich, mehrere separat erzeugte Teilprogramme nach der Makroexpansion manuell zu einem gemeinsamen Programm zusammenzuführen. Hierzu wurde für die `Program`-Struktur eine eigene `merge`-Methode implementiert, die zwei vollständig spezifizierte ASG-Instanzen miteinander kombiniert. Dabei wird über die `predicates`-Map geprüft, ob Prädikate mit identischem Namen auch kompatible Typinformationen besitzen. Ist dies der Fall, werden die Prädikate sowie die zugehörigen Komponenten (`facts`, `rules`, `queries`) zusammengeführt.

6.2 Erweiterung zur Unterstützung von Modularität

Das einfache Zusammenführen von Datalog-Teilprogrammen – etwa durch das manuelle Aufrufen der `merge`-Methode zum Mergen mehrerer `Program`-Instanzen in Rust – stellt noch keine ausreichende Modularisierung dar. Zwar können damit Programmfragmente in unterschiedliche Makro-Aufrufe ausgelagert werden, jedoch entsteht dabei keine Wiederverwendbarkeit: Jedes Modul enthält seine eigene vollständige Definition der verwendeten Prädikate, was eine konsistente Verwendung über mehrere Module hinweg erschwert.

Für echte Modularisierung sind *Schnittstellen* erforderlich. Diese Schnittstellen müssen *typisiert* sein, um sicherzustellen, dass sich alle beteiligten Module auf einheitliche Typinformationen einigen. Im Fall von Datalog liegen diese Typinformationen in den Prädikaten vor.

Die Idee zur Realisierung modularer Schnittstellen besteht darin, Prädikate als Schnittstellen zu verwenden, indem ein Prädikat *extern* definiert wird – etwa als Funktion, die ein Prädikat zurückgibt: `fn edge() -> Rc<Predicate> { ... }` – und in verschiedenen Makro-Aufrufen darauf referenziert wird, anstatt das Prädikat mehrfach lokal zu definieren.

So könnte beispielsweise anstatt einer lokalen Definition des Prädikats in Codebeispiel 10, Zeile 2 ein Funktionsaufruf erfolgen: `let pred_edge: Rc<Predicate> = edge();` dafür müsste bei dem Makro-Aufruf lediglich der *Pfad* zu dieser Funktion übergeben werden.

Dabei entsteht jedoch die Herausforderung, dass für importierte Prädikate zur Makroexpansion die Typinformationen nicht bekannt sind. Da Rust-Makros isoliert arbeiten und nur Zugriff auf den übergebenen TokenStream haben, ist zu diesem Zeitpunkt lediglich der Pfad zur Funktion bekannt, jedoch nicht deren Rückgabewert mit den Typinformationen.

Die Typinferenz muss so angepasst werden, dass akzeptiert wird, dass sich bestimmte Typinformationen nicht zur Compile-Zeit herleiten lassen – insbesondere wenn sie nur aus dem Kontext eines importierten Prädikats hervorgehen.

Um dennoch eine konsistente semantische Analyse zu gewährleisten, muss zusätzlich zu dem ASG-Code auch *Validierungscode* generiert werden, der zur Laufzeit prüft, ob die importierten Prädikate die erwartete Arity und Typen besitzen. Es muss jedoch nicht explizit geprüft werden, ob eine solche Funktion existiert, oder den korrekten Rückgabetyp hat, solche Sicherstellungen werden bereits vom Rust-Compiler übernommen. So können also Prädikate mittels Funktionen definiert werden und in mehreren Makroaufrufen verwendet werden.

Um den Nutzen von modularem Datalog zu veranschaulichen wird zunächst in Abbildung 3 eine vereinfachte Codeanalyse dargestellt, bei der Prädikate in verschieden Modulen definiert werden. Angenommen es soll eine Codeanalyse zur Bestimmung von unerreichbarem Code durchgeführt werden – für eine fiktive funktionale Programmiersprache, die aus Funktionen und Funktionsaufrufen besteht. Die Funktion `main` dient als Einstiegspunkt in das Programm. Alle Funktionen, die von `main` aus nicht erreichbar sind, gelten als toter Code und können entsprechen markiert oder durch eine Fehlermeldung hervorgehoben werden.

Das Modul `program_facts` enthält die Fakten über das Programm – etwa welche Funktionen deklariert wurden und welche Funktionsaufrufe definiert sind. Das Modul `call_analysis` ist dafür verantwortlich zu bestimmen, welche Funktionsaufrufe gültig sind (sowohl *Caller* als auch *Callee* müssen deklarierte Funktionen sein). Darauf aufbauend bestimmt die Regel `reachable` von welchen Funktionen aus welche anderen Funktionen erreichbar sind. Das Modul `verify_program` beinhaltet die Regeln zur Bestimmung, welche Funktionen von der `main`-Funktion aus erreichbar sind sowie Funktionen die als unerreichbar markiert werden.

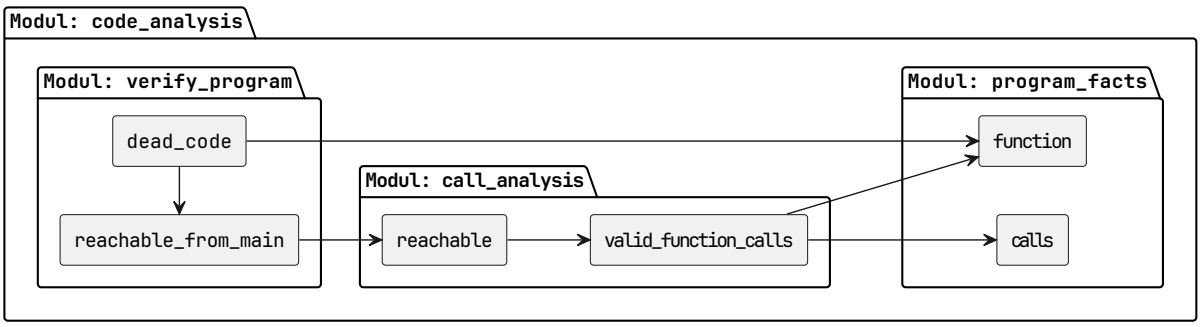

Abbildung 3: *Modular definierte Prädikate in einer Datalog-basierten Codeanalyse.* Das Diagramm zeigt, wie Fakten und Regeln (`dead_code`, `reachable_from_main`, `reachable`, `valid_function_call`, `function`, `calls`) in verschiedenen Modulen definiert und über Modulgrenzen hinweg importiert und verwendet werden. Die gerichteten Pfeile stellen Importabhängigkeiten zwischen Prädikaten dar – beispielsweise nutzt `valid_function_call` die in `program_facts` definierten Prädikate `function` und `calls`.

Nachdem gezeigt wurde, wie modular definierte Prädikate verwendet werden können, folgt nun die Beschreibung der Umsetzung.

Die Funktionen zur Definition der Prädikate können direkt in der Codegenerierung erzeugt werden. Ein Makroaufruf erzeugt dabei für jedes Prädikat eine eigene Funktion. Auf diese Weise entstehen automatisch Schnittstellen, die wiederum in anderen Makroaufrufen verwendet werden können.

Um die in den `tofina!`-Makros definierten Teilprogramme zu einem vollständigen Datalog-Programm zusammenzuführen – ohne dass manuell die `merge`-Methode aufgerufen werden muss – wird eine *globale* `Program`-Instanz verwendet. Ziel ist es, dass Makroaufrufe in verschiedenen Rust-Modulen jeweils Teilaspekte des Datalog-Programms spezifizieren, und diese automatisch zu einem vollständigen Gesamtprogramm zusammengefügt werden.

Dies wird dadurch erreicht, dass in der Codegenerierung zusätzlicher Code erzeugt wird, der das jeweils lokal definierte Teilprogramm in den globalen ASG einbettet. Zu diesem Zweck wird neben den Funktionen zur Definition der Prädikate eine weitere Funktion `register_program()` generiert. Diese erzeugt eine lokale `Program`-Instanz, wie in Codebeispiel 10 gezeigt, und übergibt diese per Aufruf von `merge` an die globale statische Instanz.

Damit diese Funktion nicht manuell aufgerufen werden muss, wird sie mit dem Attribut `#[ctor]` annotiert [26]. Dies bewirkt, dass die Funktion beim Start des Programms – noch vor Ausführung der `main`-Funktion – automatisch ausgeführt wird. So registriert sich jedes lokal definierte Teilprogramm beim Start selbstständig im globalen Datalog-Programm.

Ein Problem, das bei dieser Art der Modularisierung auftritt, ist die potenzielle Namenskollision von Prädikaten. Wird beispielsweise in zwei verschiedenen Makroaufrufen das Prädikat `item` verwendet – etwa einmal im Modul `a` mit dem Typ `Integer`, und einmal im Modul `b` mit dem Typ `String` – so führt dies zu einem Konflikt beim Zusammenführen.

Um dieses Problem zu lösen, werden *Namespaces* (Namensräume) eingeführt. Jeder Makroaufruf erhält implizit einen Namensraum basierend auf dem Rust-Modul, in dem er sich befindet. Das bedeutet: Das Prädikat `item`, das im Modul `a` definiert wird, erhält den vollqualifizierten Namen `a::item`, während das Prädikat im Modul `b` als `b::item` bezeichnet wird. Damit sind beide Prädikate unterscheidbar und können koexistieren, selbst wenn sie unterschiedliche Typinformationen haben.

Die Umsetzung erfolgt über das Makro `module_path!`, welches den aktuellen Modulpfad zur Compile-Zeit expandiert [27]. In der Codegenerierung für die Prädikat-Funktionen wird ein Aufruf von

`module_path!` als Präfix für den Prädikats-Namen erstellt. Wird also innerhalb eines Makroaufrufs im Modul `a` das Prädikat `item` definiert, so erhält die generierte Funktion den Namen `item`, welche ein Prädikat mit dem Namen `a::item` zurück gibt.

Der Ablauf der Registrierung eines Teilprogramms wird am Beispiel des Moduls `call_analysis` aus Abbildung 3 anhand eines Sequenzdiagramms in Abbildung 4 dargestellt.

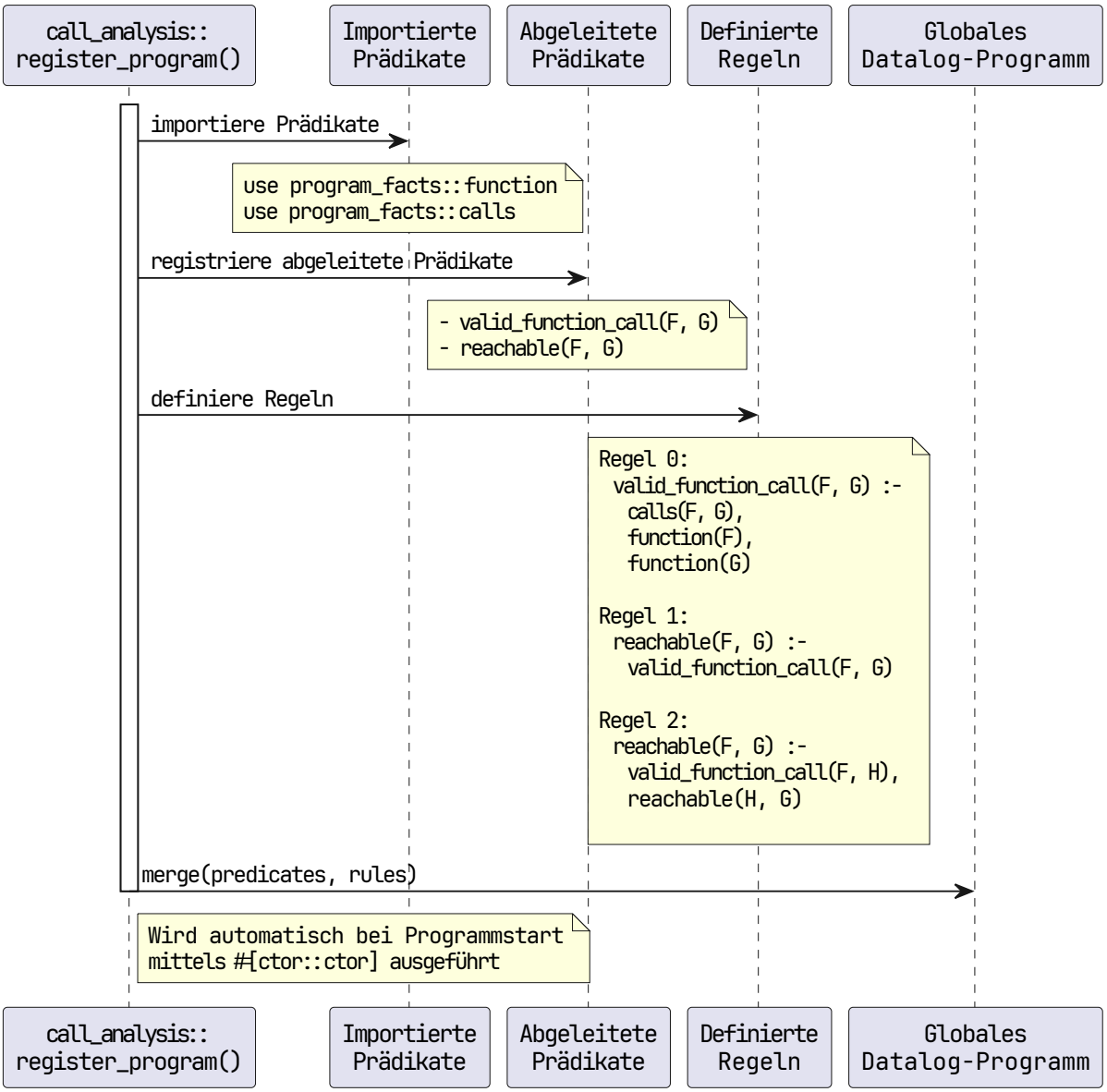

Abbildung 4: *Ablauf der automatisch ausgeführten Funktion `register_program()` im Modul `call_analysis`*. Die Funktion `register_program()` wird als Teil der Makro-Expansion generiert und beim Programmstart durch das Attribut `#[ctor::ctor]` automatisch ausgeführt. Sie erstellt ein lokales Datalog-Programm aus den im Modulkontext definierten Komponenten – einschließlich importierter und abgeleiteter Prädikate, Regeln, Fakten und Abfragen – und führt dieses anschließend mit der globalen statischen Programm-Instanz zusammen.

Für die Prüfung der modularen Prädikate wird in der Codegenerierung ein automatisches Testmodul erzeugt. Für jedes Makro, das importierte Prädikate verwendet, wird ein Rust-Modul `mod test` generiert, das mit dem Attribut `#[cfg(test)]` versehen ist. Innerhalb dieses Testmoduls wird für jedes importierte Prädikat eine eigene Funktion erstellt, die mit `#[test]` annotiert ist.

Aus [28]: Im automatisch generierten Testmodul steht das Attribut `cfg` für Konfiguration und teilt Rust mit, dass das folgende Element nur bei einer bestimmten Konfigurationsoption eingebunden werden soll. In diesem Fall ist die Konfigurationsoption `test`, die von Rust zum Kompilieren und Ausführen von Tests bereitgestellt wird. Durch die Verwendung des Attributs `cfg` kompiliert Cargo den Testcode nur, wenn die Tests aktiv mit `cargo test` ausgeführt werden. Dies schließt alle Hilfsfunktionen ein, die sich in diesem Modul befinden, zusätzlich zu den mit `#[test]` annotierten Funktionen.

In Codebeispiel 11 ist der generierte Testcode dargestellt, für das importierte Prädikat `calls` im Modul `call_analysis` aus Abbildung 3.

```rust
#[cfg(test)]
mod test {
  use super::*;
  #[test]
  fn test_imported_pred_calls() {
    let calls: Rc<Predicate> = program_facts::calls();
    assert_eq!(calls.arity(), 2);
    assert_eq!(calls.types[0], Type::String);
    assert_eq!(calls.types[1], Type::String);
  }
  // ...
}
```

Codebeispiel 11: *Automatisch generierte Laufzeittests für importierte Prädikate.* Mittels `use super::*` werden alle im umgebenen Modul (dem Makro-Kontext) definierten Komponenten verfügbar gemacht. Die Testfunktion ruft das importierte Prädikat `calls` aus dem Modul `program_facts` auf und prüft per `assert_eq!`, ob Arity und Typen den erwarteten Werten entsprechen.

Somit wird eine Möglichkeit zur Sicherstellung der Konsistenz und Kompatibilität importierter Prädikate geboten – welche nicht vollständig zur Compile-Zeit geprüft werden können – durch Laufzeit Validierung, ohne die reguläre Laufzeit des Programms zu belasten.

6.3 Anwendungsbeispiele

Im Folgenden werden zwei Beispiele vorgestellt, die den praktischen Einsatz der Tofina-DSL demonstrieren. Beide bauen auf den zuvor beschriebenen Konzepten der Modularisierung auf und zeigen, wie diese in Rust mithilfe des `tofina!`-Makros umgesetzt werden können. Das erste Beispiel greift das eingangs skizzierte Anwendungsszenario einer modularen Codeanalyse auf und zeigt, wie sich eine solche Analyse konkret mit Tofina realisieren lässt. Das zweite Beispiel verdeutlicht den Einsatz von Namensräumen zur Vermeidung von Prädikatskonflikten in voneinander unabhängigen Modulen.

Das folgende Beispiel zeigt die Implementierung einer modularen Codeanalyse in Tofina, wie sie zuvor konzeptionell in Abbildung 3 beschrieben wurde. Drei Module spezifizieren dabei jeweils die Faktenbasis, die Analyse und die Verifikation. Über die Verwendung von importierten Prädikaten und automatischer Registrierung entsteht ein konsistentes, modular aufgebautes Datalog-Programm. Die einzelnen Module sind in Codebeispiel 12, Codebeispiel 13 und Codebeispiel 14 dargestellt:

code_analysis/program_facts.rs rust

```rust
tofina_macros::tofina! {
  // Declared functions
  function("main");
  function("foo");
  function("bar");        // <- dead_code
  // Function-calls(Caller, Callee)
  calls("main", "foo");
  calls("foo", "baz");    // <- invalid_call
}
```

Codebeispiel 12: *Modul zur Erfassung grundlegender Programminformationen.* Im Modul `program_facts` werden Fakten über die deklarierten Funktionen – `function` – sowie über direkte Funktionsaufrufe – `calls` – definiert. Ungültige Aufrufe (z. B. zu nicht deklarierten Funktionen wie `baz`) sowie nicht referenzierte Funktionen (z. B. `bar`) werden später identifiziert.

code_analysis/call_analysis.rs rust

```rust
use super::program_facts;

tofina_macros::tofina! {
  use program_facts::calls;
  use program_facts::function;

  valid_function_call(F, G) <- calls(F, G), function(F), function(G);

  reachable(F, G) <- valid_function_call(F, G);
  reachable(F, G) <- valid_function_call(F, H), reachable(H, G);
}
```

Codebeispiel 13: *Modul zur Analyse gültiger und transitiver Funktionsaufrufe.* Im Modul `call_analysis` wird mithilfe der aus `program_facts` importierten Prädikate `calls` und `function` überprüft, ob definierte Funktionsaufrufe gültig sind – also ob sowohl aufrufende als auch aufgerufene Funktionen tatsächlich existieren. Dies geschieht durch das Prädikat `valid_function_call`. Darauf aufbauend wird mit dem Prädikat `reachable` analysiert, welche Funktionen auch transitiv (indirekt) erreichbar sind.

code_analysis/verify_program.rs rust

```rust
use super::{call_analysis, program_facts};
tofina_macros::tofina! {
  use call_analysis::reachable;
  use program_facts::function;
  // Functions reachable from main
  reachable_from_main(F) <- reachable("main", F);
  // Functions unreachable from main are dead_code
  dead_code(F) <- function(F), !reachable_from_main(F);
  ?dead_code(x);
}
```

Codebeispiel 14: *Modul zur Identifikation von totem Code basierend auf Erreichbarkeit.* Im Modul `verify_program` wird mithilfe der Prädikate `reachable` (importiert aus `call_analysis`) sowie `function` (aus `program_facts`) untersucht, welche Funktionen von `main` aus erreichbar sind. Funktionen, die nicht mit `reachable_from_main` erreichbar sind, gelten als toter Code – `dead_code` – und werden mit `?dead_code(x);` ausgegeben.

Codebeispiel 15 demonstriert den Einsatz von Namensräumen in Tofina anhand eines einfachen Tabellenmodells. Es illustriert, wie gleichnamige Prädikate aus verschiedenen Modulen über Aliase konfliktfrei gemeinsam verwendet werden können. Dies wird insbesondere dann relevant, wenn verschiedene Komponenten unabhängig entwickelt werden, aber ähnliche Begriffe (z. B. `value`) verwenden.

table.rs rust

```rust
mod rows {
    tofina_macros::tofina! {
        value(1);
        value(2);
    }
}

mod columns {
    tofina_macros::tofina! {
        value("A");
        value("B");
    }
}

mod grid {
    use super::{columns::value as column, rows::value as row};
    tofina_macros::tofina! {
        use row;
        use column;

        cell(R, C) <- row(R), column(C);
    }
}
```

Codebeispiel 15: *Modulübergreifende Nutzung gleichnamiger Prädikate mittels Namensräumen.* Das Beispiel zeigt, wie gleichnamige Prädikate mithilfe von Rust-Namespaces (Modulen) eindeutig referenziert werden können. Die Datei `table.rs` enthält drei Module: `rows`, `columns` und `grid`. Sowohl `rows` als auch `columns` definieren ein Prädikat `value` mit jeweils unterschiedlichen Fakten bzw. Typen. Im Modul `grid` werden diese Prädikate mithilfe von Aliasnamen (`row`, `column`) importiert, wodurch innerhalb des `tofina!`-Blocks Namenskollisionen vermieden werden. Das Prädikat `cell(R, C)` beschreibt das kartesische Produkt der Mengen von Zeilen- und Spaltenwerten, d. h. alle gültigen Zellkoordinaten der Tabelle. Trotz der Aliasverwendung werden intern die vollqualifizierten Namen `table::rows::value`, `table::columns::value` und `table::grid::cell` verwendet.

7 Fazit und Ausblick

In dieser Arbeit wurde mit *Tofina* eine modulare, domänenspezifische Datalog-Sprache für Rust entwickelt. Die Sprache wird als *interne DSL* realisiert und basiert auf einem prozeduralen Makro *tofina!*, das die Definition modularer Datalog-Programme innerhalb der Rust-Modulstruktur erlaubt. Ziel war die konzeptionelle Entwicklung sowie die Implementierung einer modularen internen Datalog-DSL in Rust.

Das zentrale Element ist das prozedurale Makro `tofina!`, das benutzerdefinierte Datalog-Programme zur Compilezeit analysiert und in ausführbaren Rust-Code überführt. Für jedes definierte Prädikat wird eine Funktion erzeugt, über die andere Module darauf zugreifen können. Durch automatische Registrierung mittels `#[ctor]` werden alle Datalog-Teilprogramme zur Laufzeit zu einem konsistenten Programm zusammengeführt. Zur Prüfung der Typverträglichkeit importierter Prädikate generiert

Tofina automatisch Testfunktionen. Diese werden nur bei Ausführung von `cargo test` ausgeführt und beeinflussen nicht die reguläre Laufzeit.

Im Vergleich zu bestehenden Systemen ergeben sich unterschiedliche Schwerpunkte: Während *Ascent* auf modular strukturierte *Programmeinheiten* setzt, importiert Tofina gezielt *einzelne Prädikate* als explizite Schnittstellen. Durch vollqualifizierte Namen wird sichergestellt, dass es in Tofina zu keinen Namenskonflikten kommt, was eine robuste Modularisierung auf Ebene der Prädikate erlaubt. Tofina übernimmt die Idee von *Namensräumen* von *Souffle*, kann jedoch im Gegensatz zu Soufflés *Components* keine komplexeren Abstraktionen wie *Vererbung* oder *Subtypen* abbilden.

Einschränkungen ergeben sich aus der aktuell begrenzten Ausdrucksstärke: Weder Aggregationen, arithmetische Ausdrücke noch Vergleichsoperatoren werden derzeit unterstützt. Auch auf Seiten der Typisierung bietet Tofina bislang keine Möglichkeit, benutzerdefinierte Typen einzubinden. Während Crepe und Ascent generische Rust-Typen verarbeiten, ist Tofina aktuell auf die Datentypen `String` und `Integer` beschränkt, da die interne Repräsentation (ASG) nur diese Typen in Form eines Enums unterstützt. Dies führt jedoch dazu dass zusätzliche semantische Prüfungen erforderlich sind, da die ASG-Struktur auch die Modellierung ungültiger Programme zulässt.

Es bestehen mehrere konkrete Ansatzpunkte für zukünftige Weiterentwicklungen:

- *Fehlermeldungen mit Spans:* Aktuell erzeugt das Makro keine präzisen Fehlermeldungen, da keine `Span`-Informationen verwendet werden. Eine Verbesserung der Fehlermeldungen würde die Benutzbarkeit erhöhen.
- *Sichtbarkeitsregeln für Prädikate:* Derzeit werden alle Prädikatsfunktionen mit öffentlichen Zugriffsmodifikatoren `pub` generierten.
- *Typdeklarationen:* Bisher werden Typen ausschließlich aus Konstanten abgeleitet. Eine Erweiterung der DSL um explizite Typannotationen könnte die Verständlichkeit und Typsicherheit erhöhen.
- *Typisierung und Traits:* Durch die Einführung von Traits für zulässige Datentypen könnten die aktuellen Einschränkungen auf String und Integer aufgehoben werden, indem durch einen Trait definiert wird, welche Eigenschaften ein Datentyp erfüllen muss, um mit der Engine verarbeitet werden zu können.

Insgesamt zeigt Tofina, dass sich modularisierte Datalog-Programme mittels prozeduralen Makros als interne DSL in Rust realisieren lassen.

Literatur

[1] André Pacak, „Datalog as a (non-logic) Programming Language“, 2024. Zugegriffen: 15. September 2025. [Online]. Verfügbar unter: https://openscience.ub.uni-mainz.de/handle/20.500.12030/10930

[2] Todd J. Green, Shan Shan Huang, Boon Thau Loo, und Wenchao Zhou, „Datalog and Recursive Query Processing“, 2012. doi: 10.1561/1900000017.

[3] Stefano Ceri, Georg Gottlob, und Letizia Tanca, „What You Always Wanted to Know About Datalog (And Never Dared to Ask)“. S. 146–166, 1989. doi: 10.1109/69.43410.

[4] Bruce J. MacLennan, *PRINCIPLES OF PROGRAMMING LANGUAGES: Design, Evaluatoin, and Implementation*. S. 445–491. Zugegriffen: 16. September 2025. [Online]. Verfügbar unter: https://web.eecs.utk.edu/~bmaclenn/POPL/ch13.pdf

[5] Philipp Körner *u. a.*, „Fifty Years of Prolog and Beyond“, 2022. doi: 10.1017/S1471068422000102.

[6] Markus Voelter, *DSL Engineering*. Zugegriffen: 16. September 2025. [Online]. Verfügbar unter: https://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf

[7] Bernhard Scholz, Herbert Jordan, Pavle Subotić, und Till Westmann, „On Fast Large-Scale Program Analysis in Datalog“. S. 196–206, 2016. doi: 10.1145/2892208.2892226.

[8] Shan Shan Huang, Todd Jeffrey Green, und Boon Thau Loo, „Datalog and Emerging Applications: An Interactive Tutorial“. Juni 2011. doi: 10.1145/1989323.1989456.

[9] Martin Diller, Sarah Alice Gaggl, Philipp Hanisch, Giuseppina Monterosso, und Fritz Rauschenbach, „Grounding Rule-Based Argumentation Using Datalog“. 14. August 2025. doi: 10.48550/arXiv.2508.10976.

[10] „Soufflé - Logic Defined Static Analysis“. Zugegriffen: 17. September 2025. [Online]. Verfügbar unter: https://souffle-lang.github.io/

[11] „CodeQL“. Zugegriffen: 15. September 2025. [Online]. Verfügbar unter: https://codeql.github.com/

[12] *Nemo: A fast in-memory rule engine*. Zugegriffen: September 2025. [Online]. Verfügbar unter: https://github.com/knowsys/nemo

[13] Alex Ivliev *u. a.*, „Nemo: First Glimpse of a New Rule Engine“, 2023. doi: 10.4204/EPTCS.385.35.

[14] „Nemo: Graph Rule Engine“. Zugegriffen: 17. September 2025. [Online]. Verfügbar unter: https://knowsys.github.io/nemo-doc/

[15] Steve Klabnik und Carol Nichols, „The Rust Programming Language“. Zugegriffen: 12. September 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/book/

[16] The Rust Project Developers, „The Rust Reference: Macros“. Zugegriffen: 16. September 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/reference/macros.html

[17] *Crepe: Datalog compiler embedded in Rust as a procedural macro*. Zugegriffen: 16. September 2025. [Online]. Verfügbar unter: https://github.com/ekzhang/crepe/

[18] *Ascent: Logic programming in Rust*. Zugegriffen: 16. September 2025. [Online]. Verfügbar unter: https://github.com/s-arash/ascent

[19] Arash Sahebolamri, Thomas Gilray, und Kristopher Micinski, „Seamless Deductive Inference via Macros“. S. 77–88, April 2022. doi: 10.1145/3497776.3517779.

[20] A. Sahebolamri, L. Barrett, S. Moore, und K. Micinski, „Bring Your Own Data Structures to Datalog“. doi: 10.1145/3622840.

[21] D. L. Parnas, „On the criteria to be used in decomposing systems into modules“, *Communications of the ACM*, Bd. 15, Nr. 12, S. 1053–1058, Dez. 1972, doi: 10.1145/361598.361623.

[22] Stephen Clyde und Jorge Edison Lascano, „Unifying Definitions for Modularity, Abstraction, and Encapsulation as a Step Toward Foundational Multi-Paradigm Software Engineering Principles“. S. 105–113, Oktober 2017. [Online]. Verfügbar unter: https://www.researchgate.net/publication/331452921_Unifying_Definitions_for_Modularity_Abstraction_and_Encapsulation_as_a_Step_Toward_Foundational_Multi-Paradigm_Software_Engineering_Principles

[23] Mahmoud Efatmaneshnik, Shraga Shoval, und Li Qiao, „A Standard Description of the Terms Module and Modularity for Systems Engineering“, Mai 2020. doi: 10.1109/TEM.2018.2878589.

[24] Gopalan Nadathur und Guanshan Tong, „Realizing Modularity in Prolog“, September 1997. [Online]. Verfügbar unter: https://www.researchgate.net/publication/2829937_Realizing_Modularity_in_Prolog

[25] Foto Afrati, Isambo Karali, und Theodoros Mitakos, „Inheritance in Object Oriented Datalog: A Modular Logic Programming Approach“, Dezember 1997. [Online]. Verfügbar unter: https://www.researchgate.net/publication/2316768_Inheritance_in_Object_Oriented_Datalog_A_Modular_Logic_Programming_Approach

[26] „Crate ctor“. Zugegriffen: 13. September 2025. [Online]. Verfügbar unter: https://crates.io/crates/ctor

[27] „Macro module_path“. Zugegriffen: 16. September 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/std/macro.module_path.html

[28] „Test Organization“. Zugegriffen: 16. September 2025. [Online]. Verfügbar unter: https://doc.rust-lang.org/book/ch11-03-test-organization.html