ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δηλωτική Ανάλυση Δεικτών σε Διαφορετικές Υλοποιήσεις της Datalog

Αναστάσιος Ι. Αντωνιάδης

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΑΝΟΥΑΡΙΟΣ 2014

NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

THESIS

Declarative Points-To Analysis on Different Datalog Engines

Anastasios I. Antoniadis

Supervisors: **Yannis Smaragdakis,** Associate Professor NKUA

ATHENS

JANUARY 2014

THESIS

Declarative Points-To Analysis on Different Datalog Engines

Anastasios I. Antoniadis

R.N..: 1115200600031

SUPERVISORS: **Yannis Smaragdakis,** Associate Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δηλωτική Ανάλυση Δεικτών σε Διαφορετικές Υλοποιήσεις της Datalog

Αναστάσιος Ι. Αντωνιάδης
Α.Μ.: 1115200600031

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια η Datalog έχει βρει νέα εφαρμογή στην δηλωτική ανάλυση προγραμμάτων. Σε αυτή την πτυχιακή εργασία παρουσιάζουμε ένα πρωτότυπο framework για ανάλυση δεικτών (ανεξάρτητη συμφραζομένων) σε προγράμματα Java, η οποία είναι μια κατηγορία στατικής ανάλυσης προγραμμάτων που εκτιμά πού μπορεί να 'δείξει' κάθε μεταβλητή του προγράμματος για κάθε πιθανή εκτέλεση του κώδικα. Το framework αυτό χρησιμοποιεί τη Datomic, μια κατανεμημένη βάση δεδομένων η οποία χρησιμοποιεί μια γλώσσα επερωτήσεων βασισμένη στη Datalog. Η ίδια ανάλυση δεικτών ανεξαρτήτως συμφραζομένων υλοποιήθηκε στη διάλεκτο Datalog^{LB}, η οποία χρησιμοποιεί την μηχανή LogicBlox, προκειμένου να χρησιμοποιηθεί ως μέτρο σύγκρισης. Ο στόχος μας είναι να αξιολογήσουμε το σύστημα βάσης δεδομένων της Datomic για χρήση σε δηλωτικές αναλύσεις προγραμμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση Προγραμμάτων.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Datalog, Datomic, ανάλυση δεικτών ανεξάρτητη συμφραζομένων

ABSTRACT

In recent years, Datalog has found new application in declarative program analysis. In this thesis we present a prototype framework for context-insensitive points-to analysis of Java programs, which is a category of static program analysis that evaluates where each variable of a program can 'point-to' for each possible execution of the code. This framework uses Datomic, a distributed database which implements a Datalog-based query language. The same prototype context-insensitive analysis has been implemented using the Datalog^{LB} dialect, which uses the LogicBlox engine in order to be used as a benchmark for comparison. Our aim is to evaluate the performance of the Datomic database system for the purpose of declarative program analysis.

SUBJECT AREA: Static Program Analysis

KEYWORDS: Datalog, Datomic, context-insensitive points-to analysis

to Labros and Michael…

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Yannis Smaragdakis for providing his expertise and guidance, which were valuable for the preparation and completion of this work and most of all for his patience throughout the whole process.

Also, I would like to thank PhD candidates George Kastrinis and George Balatsouras for their contributions to this work. Their assistance helped me surpass the many obstacles I encountered during my work both in understanding Datalog^{LB} and Datomic in order to complete this work and I am grateful to them.

Contents

List of Figures

List of Tables

PREFACE

This thesis aims to evaluate the Datomic database system for the purpose of pointer analysis. It was developed in Athens, Greece and Geneva, Switzerland between July, 2012 and January, 2014. The first period of this work was associated with studying the Datalog^{LB} dialect, understanding the Doop framework for pointer analysis and implementing a prototype context-insensitive analysis in Datalog^{LB}. The second period was associated with studying and experimenting with Datomic and using it to build a prototype context-insensitive analysis framework in order to evaluate the database's capabilities.

1. Introduction

An important trend in recent program analysis literature is the expression of analyses declaratively, for clearer specification and easier modifiability [3, 6, 7, 9]. In particular the usage of Datalog for the definition of program analysis specifications has drown researchers' attention, due to its ability to specify mutually recursive relations [1, 2, 4, 5, 8]. In this case we are interested in pointer analysis of Java programs, which is a category of static program analysis that evaluates where each variable of a program can 'point-to' for each possible execution of the code.

This thesis aims to evaluate the Datomic database system, which uses a Datalog-based query language for the purpose of conducting declarative program analysis. In order to perform the evaluation we have built two prototype context-insensitive pointer analysis implementations, one in Datalog^{LB} and the other in Datomic and the evaluation of Datomic is done by comparing its execution times and memory usage to those of Datalog^{LB}.

The rest of the thesis is organized as follows:

- In Chapter 2 we give a background of points-to analysis in Datalog and then we present the highlights of the Datalog^{LB} dialect.
- In Chapter 3 we present an overview of Datomic, its features and capabilities.
- In Chapter 4 we present and explain all the basic steps of the Java application performin the analysis in Datomic.
- In Chapter 5 we perform an evaluation of Datomic by comparing the execution times and memory usage of the context-insensitive analysis implemented in our prototype Datomic framework to those of Datalog^{LB}.
- In Chapter 6 we present our conclusions.

2. Background

2.1 Points-To Analysis in Datalog

Datalog is a declarative logic-based programming language which is often used as a query language for deductive databases. In recent years, Datalog has found new application in the domain of program analysis, due to its ability to define recursive relations. Relations are the main Datalog data type and computation consists of inferring the contents of all relations from a set of input relations. Mutual recursion is the source of complexity in program analysis. Due to the fact that recursive definitions are easier to specify in Datalog, the language is very convenient for the specification of complex program analysis algorithms.

In the case of points-to analysis for Java programs, it is to easy to represent the actions of Java program as relations, stored as database tables. In particular, consider the following two relations, `AssignHeapAllocation(?heap, ?var)`[1] and `Assign(?to, ?from)`. The former relation represents all occurrences of an instruction "`a = new A();`" in a Java program, where a heap object is allocated and assigned to a variable.

The aforementioned relations are the outcome of a pre-processing step which takes a Java program as input and produces the relation contents which will be the input facts. This kind of relations which are produced directly from the input Java program, are known in Datalog terminology as the EDB (Extensional Database) predicates. EDB predicates normally are used to hold the facts that are explicitly entered by the user with fact assertions.

In particular, for `AssignHeapAllocation(?heap, ?var)` relation, a static abstraction of the heap is captured in variable `?heap`–it can be concretely represented as, for instance, a fully qualified class name and the allocation's bytecode instruction index. In the same manner, the Assign relation contains an entry for each assignment between two Java program (reference) variables.

Following the pre-processing step a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

```
VarPointsTo(?heap, ?var) <- AssignHeapAllocation(?heap, ?var).
VarPointsTo(?heap, ?to) <- Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

Each Datalog program consists of a series of rules, also known in Datalog semantics as the IDB(Intensional Database) rules, that are used to establish facts about derived relations (such as `VarPointsTo`, which is the points-to relation, meaning it links every program variable, `?var`, with every heap object abstraction, `?heap`, it can point to) from a conjunction of previously established facts (i.e., the body of the rule).

[1]We follow the convention of capitalizing the first letter of relation names, while writing variable names in lower case and prefixing them with a question-mark.

```
V is a set of program variables
H is a set of heap abstractions (i.e., allocation sites)
M is a set of method identifiers
S is a set of method signatures (including name, type signature)
F is a set of fields
I is a set of instructions (mainly used for invocation sites)
T is a set of class types
N is the set of natural numbers

ASSIGNHEAPALLOC (var : V, heap : H, inMeth : M)              # var=new ...
ASSIGNLOCAL( to : V, from : V, inMeth : M)                   # to=from
LOAD (to : V, base : V, fld : F)                             # to=base.fld
STORE (base : V, fld : F, from : V)                          # base.fld=from
VCALL (base : V, sig : S, invo : I, inMeth : M)              # base.sig(..)
SCALL (meth : M, invo : I, inMeth : M)                       # Class.meth(..)
FORMALPARAM (i : N, meth : M, arg : V)
ACTUALPARAM (i : N, invo : I, arg : V)
RETURNVAR (ret : V, meth : M)
ASSIGNRETURNVALUE (invo : I, var : V)
THISVAR (meth : M, var : V)
HEAPTYPE (heap : H, type : T)
VARTYPE (var: V, type: T)
METHODLOOKUP (type : T, sig : S, meth : M)

VARPOINTSTO (heap : H, var : V)
CALLGRAPHEDGE (invo : I, meth : M)
FLDPOINTSTO (heap: H, fld: F, baseH: H)
ASSIGN (type :T, from : V, to : V)
REACHABLE (meth : M)
```

Figure 2.1: The domain, input relations (representing program instructions – with the matching program pattern shown in a comment – and type information) and output relations for a context-insensitive analysis.

```
ASSIGN (?type, ?from, ?to) <-
      CALLGRAPHEDGE (?invo, ?meth),
      FORMALARG (?meth, ?i, ?to),
      ACTUALARG (?invo, ?i, ?from),
      VARTYPE (?from, ?type).

ASSIGN (type, from, to) <-
      CALLGRAPHEDGE (?invo, ?meth),
      RETURNVAR (?from, ?meth),
      ASSIGNRETURNVALUE(?invo, ?to),
      VARTYPE(?from, ?type).

VARPOINTSTO (?heap, ?var) <-
      REACHABLE (?meth),
      ASSIGNHEAPALLOC (?var, ?heap, ?meth).

VARPOINTSTO (?heap, ?to) <-
      ASSIGN (?type, ?from, ?to),
      VARPOINTSTO (?heap, ?from).

VARPOINTSTO (?heap, ?to) <-
      REACHABLE(?meth),
      ASSIGNLOCAL(?from, ?to, ?meth),
      VARPOINTSTO(?heap, ?from).

VARPOINTSTO (?heap, ?to) <-
      REACHABLE(?meth),
      LOAD (?base, ?fld, ?to, ?meth),
      VARPOINTSTO (?base, ?baseH),
      FLDPOINTSTO (?baseH, ?fld, ?heap).

FLDPOINTSTO (?heap, ?fld, ?baseH) <-
      REACHABLE (?meth),
      STORE (?from, ?base, ?fld, ?meth),
      VARPOINTSTO (?heap, ?from),
      VARPOINTSTO (baseH, base).

REACHABLE (?toMeth),
VARPOINTSTO (?this, ?heap),
CALLGRAPHEDGE (?invo, ?toMeth) <-
```

```
      REACHABLE (?inMeth),
      VCALL (?base, ?sig, ?invo, ?inMeth),
      VARPOINTSTO (?base, ?heap),
      HEAPTYPE (?heap, ?heapT),
      METHODLOOKUP (?heapT, ?sig, ?toMeth),
      THISVAR (?toMeth, ?this).

REACHABLE (?toMeth),
CALLGRAPHEDGE (?invo, ?toMeth) <-
      SCALL (?toMeth, ?invo, ?inMeth),
      REACHABLE (?inMeth).
```

Figure 2.2: Datalog rules for the points-to analysis and call-graph construction.

Figure 2.1 shows the domain of our points-to analysis (i.e., the different value sets that constitute the space of the computation), its input relations, the intermediate and output relations.

For the purpose of a context-insensitive analysis we ignore any kind of context. Figure 2.2 shows the points-to analysis and call-graph computation. The rule syntax is simple: the left arrow symbol (<-) separates the inferred facts (i.e., the head of the rule) from the previously established facts (i.e., the body of the rule). For instance, the first rule states that, if we have computed a call-graph edge between invocation site `?invo` and method `?meth`, then we infer an assignment to the i-th formal parameter of `?meth` from the i-th actual parameter at `?invo`, for every i. The type `?type` of the assignment relation is the type of the `?from` variable.

A more thorough explanation of the contents of both figures follows:

- The input relations correspond to the intermediate language for our analysis. They are logically grouped into relations that represent instructions and relations that represent name-and-type information. For instance, the ASSIGNHEAPALLOC relation represents every instruction that allocates a new heap object, heap, and assigns it to local variable var inside method `?inMeth`. (Note that every local variable is defined in a unique method, hence the `?inMeth` argument is also implied by var but is included to simplify later rules.) There are similar input relations for all other instruction types (ASSIGNLOCAL, LOAD, STORE, VCALL, and SCALL). Similarly, there are relations that encode pertinent symbol table information. Most of these are self-explanatory but some deserve explanation. METHODLOOKUP matches a method signature to the actual method definition inside a type. HEAPTYPE matches an object to its type, i.e., is a function on its first argument. (Note that we are shortening the term “heap object” to just “heap” and represent heap objects as allocation sites throughout.) VARTYPE matches a variable to its type, i.e., is a function on its first argument just like VARTYPE. ASSIGNRETURNVAR is also a function on its first argument (a method invocation site) and returns the local variable at the call-site that receives the method call’s return value.

- There are five output or intermediate computed relations (VARPOINTSTO, CALLGRAPHEDGE, FLDPOINTSTO, ASSIGN, REACHABLE). The main output relations are VARPOINTSTO and CALLGRAPH, encoding our points-to and call-graph results. The VARPOINTSTO relation links a variable (var) to a heap object (heap). Other intermediate relations (FLDPOINTSTO, ASSIGN, REACHABLE) correspond to standard concepts and are introduced for conciseness.

The rules of Figure 2.2 show how each input instruction leads to the inference of facts for the five output or intermediate relations. The most complex rule is the second-to-last, which handles virtual method calls (input relation VCALL). The rule says that if a reachable method of the program has an instruction making a virtual method call over local variable base (this is an input fact), and the analysis so far has established that base can point to heap object heap, then the called method is looked up inside the type of heap and several further facts are inferred: that the looked up method is reachable, that it has an edge in the call-graph from the current invocation site, and that its this variable can point to heap.

The declarative nature of Datalog allows for very concise specifications of analyses. In Figure 2.3 we demonstrate an excerpt of the logic for the Java cast checking – answering to the question “can type A be cast to type B?”. The Datalog rules presented are almost an exact transcription of the Java Language Specification.

```
/**
* - If S is an ordinary (nonarray) class, then:
*
* o If T is a class type, then S must be the same class as T, or a subclass of T.
*/
CheckCast(?s, ?s) <- ClassType(?s).
CheckCast(?s, ?t) <- Subclass(?t, ?s).


/**
* o If T is an interface type, then S must implement interface T.
*/
CheckCast(?s, ?t) <-
     ClassType(?s),
     Superinterface(?t, ?s).


/**
* - If S is an interface type, then:
*
* o If T is a class type, then T must be Object
*/
CheckCast(?s, t) <-
     InterfaceType(?s),
     Type:Value(t:"java.lang.Object").
/**
```

```
* o If T is an interface type, then T must be the same interface
* as S or a superinterface of S
*/
CheckCast(?s, ?s) <-
     InterfaceType(?s).

CheckCast(?s, ?t) <-
     InterfaceType(?s),
     Superinterface(?t, ?s).
```

Figure 2.3: Excerpt of Datalog code for Java cast checking. The Java Language Specification text for each rule is included in its comment section.

2.2 LogicBlox Datalog dialect and engine:

This version of Datalog allows ‘‘stratified negation’’, that is, negated clauses, as long as the negation is not part of a recursive cycle. It also allows specifying that some relations are functions, that is, the variable space is partitioned into domain and range variables, and there is only one range value for each unique combination of values in domain variables.

Some highlights of the language are:

2.2.1 Rules

Datalog^{LB} rules are specified using a <- notation (instead of the traditional “:-"), as in the example below:

```
VarPointsTo(?var, ?heap) <-
    Reachable(?meth),
    AssignHeapAllocation(?var, ?heap, ?meth);
    Assign(?to, ?from),
    VarPointsTo(from, heap).

MethodLookup[?name, ?descriptor, ?type] = ?method <-
    MethodImpl[?name, ?descriptor, ?type] = ?method.

MethodLookup[?name, ?descriptor, ?type] = ?method <-
    DirectSuperclass[?type] = ?supertype,
    MethodLookup[?name, ?descriptor, ?supertype] = ?method,
    !MethodImpl[?name, ?descriptor, ?type].
```

In this example, `;` indicates disjunction while `!` is used for negation. Predicate and variable names may use lower/upper case freely. The first rule computes the `VarPointsTo` predicate, essentially as the union of two conjunctive queries. The second rule computes the `MethodLookup` predicate by copying data from the `MethodImpl` predicate. Finally, the third rule computes the `MethodLookup` predicate by looking the method up in the `?supertype` if it is not implemented in the `?type` and `?supertype` is the direct superclass of `?type`, with negation interpreted under the stratified semantics.

2.2.2 Entities

The main building-blocks of the Datalog^{LB} type system are entities, i.e., specially declared unary predicates corresponding to some concrete object or abstract concept. The Datalog^{LB} type system also includes various primitive types (e.g., numeric types, strings etc.). For example, the following Datalog^{LB} program declares (using a -> notation) that `MethodSignatureRef` is an entity:

```
MethodSignatureRef(?x) -> .
```

2.2.3 Refmodes

Refmodes are used in circumstances where it is necessary to define a key to identify each entity. A refmode predicate is normally declared at the same time an entity type is declared.

```
MethodSignatureRef(x), MethodSignatureRef:Value(x:s) ->
string(s).
```

2.2.4 Types

Entities can be arranged in subtyping hierarchies, e.g., the following example declares that ClassType is a subtype of Type:

```
ClassType(x) -> Type(x).
```

As expected, subtypes inherit the properties of their supertypes and can be used wherever instances of their supertypes are allowed by the type system. The -> notation can also be used to specify runtime integrity constraints.

2.2.5 Functional Predicates

If a predicate is functional and not a refmode predicate, its arguments should be declared using Datalog^{LB}'s functional notation, in which the arguments that functionally determine the final argument (the keyspace) are placed in square-brackets, followed by

the equals operator and the final argument. Those arguments not in the keyspace of a predicate are said to form its valuespace.

```
MethodSignature:Type[?signature] = ?type ->
    MethodSignatureRef(?signature),
    Type(?type).
```

The above syntax describes the following explicit declaration and constraint:

```
MethodSignature:Type(?signature,?type) ->
    MethodSignatureRef(?signature),
    Type(?type).

MethodSignature:Type(?signature,?type1),
    MethodSignature:Type(?signature, ?type2) ->
    ?type1 = ?type2.
```

With the functional notation, however, this constraint is implicit. That is, due to the fact that by using the functional notation, the predicate's functional nature is automatically declared.

The prototype Datalog^{LB} context-insensitive analysis framework demonstrated in this thesis is based on Doop, a declarative points-to analysis framework for Java programs implemented by Bravenboer et al [1, 2, 8].

3. Overview of Datomic

Datomic is a distributed database of flexible, time-based facts, supporting queries and joins, with elastic scalability. Datomic is a non-relational database providing a logical query language–Datalog, for the purpose of bringing declarative data manipulation to the application and it runs on the JVM (Java Virtual Machine).

3.1 Architecture

In a Datomic-based system, the application (or a part of the application) is a Peer. A Peer is a process that manipulates a database using the Datomic Peer library. Any process can be a Peer and the Datomic-specific code written in an application is run in the Peer(s).

Peers read facts from the Storage Service. The facts the Storage Service returns never change, so Peers do extensive caching. Each Peer's cache represents a partial copy of all the facts in the database. The Peer cache implements a least-recently used policy for discarding data, making it possible to work with databases that won't fit entirely in memory. Once a Peer's working set is cached, there is little or no network traffic for reads.

Peers write new facts by asking the Transactor to add them to the Storage Service. The Transactor processes these requests using ACID transactions, ensuring they succeed or fail atomically and do not interfere with one another. The Transactor notifies all Peers about new facts so that they can add them to their caches.

Peers can query and access data locally using a database value. Database values are constructed when code in a Peer requests them. By default, a database value is constructed from the most recent set of facts a Peer has. However, it is also possible to construct a value for a database at a particular moment in the past by using the facts stored as of that time. This is possible because old facts are immutable, remaining unchanged over time. Database values share underlying data structures, differing only as much as is necessary to represent changes. This structural sharing makes building new database values very efficient in terms of both time and space.

The ability to query and access data locally has a profound effect on the code in a Peer. Query results are directly accessible as simple data structures without having to deal with any added abstractions.

Values are immutable and provide a stable, consistent view of data for as long as a Peer needs one. The code in a Peer can also access multiple database values simultaneously, making it possible use different values to process different requests, and to compare values from different points in time.

3.2 Data Model

Datomic does not model data as documents, objects or rows in a table. Instead, data is represented as a collection of immutable facts called “Datoms”. A datom consists of the following four pieces:

1. Entity
2. Attributes
3. Value

4. Transaction timestamp

A more specific demonstration of a Datomic database structure would be that of a flat set of datoms.

Table 1: Datomic database structure

Entity	Attribute	Value	Timestamp
21005	:Var/name	java.lang.Object.toString/$r4	400
21005	:Var/type	java.lang.StringmoveToFront/r1	400
21006	:Var/name	java.lang.Object.getClass/@this	421
21006	:Var/type	java.lang.StringBuffer	421
21007	:Var/name	java.lang.Object	421
21007	:Var/type	java.lang.Object	421

An important characteristic of Datomic is that the time essence is built-in, since every datom retains its transaction. Transactions are totally ordered, first-class entities. By default Datomic retrieves the most recent database value.

Table 2: Datomic database structure - most recent value

Entity	Attribute	Value	Timestamp
21005	:Var/name	java.lang.Object.toString/$r4	400
21005	:Var/type	java.lang.StringmoveToFront/r1	400
21006	:Var/name	java.lang.Object.getClass/@this	421
21006	:Var/type	java.lang.StringBuffer	421
21007	:Var/name	java.lang.Object	421
21007	:Var/type	java.lang.Object	421

Due to the fact that time is built-in it is also possible to get the database value as-of a previous point in time.

Table 3: Datomic database structure - as-of 400 value

Entity	Attribute	Value	Timestamp
21005	:Var/name	java.lang.Object.toString/$r4	400
21005	:Var/type	java.lang.StringmoveToFront/r1	400
21006	:Var/name	java.lang.Object.getClass/@this	421

21006	:Var/type	java.lang.StringBuffer	421
21007	:Var/name	java.lang.Object	421
21007	:Var/type	java.lang.Object	421

3.3 Schema

As described in section 3.3 the facts that a Datomic database stores are represented by datoms. Each datom is an addition or retraction of a relation between an entity, an attribute, a value, and a transaction. The set of possible attributes a datom can specify is defined by a database's schema.

Each Datomic database has a schema that describes the set of attributes that can be associated with entities. A schema only defines the characteristics of the attributes themselves. It does not define which attributes can be associated with which entities. Decisions about which attributes apply to which entities are made at the application level.

This gives applications a great degree of freedom to evolve over time. For example, an application that wants to model a person as an entity does not have to decide up front whether the person is an employee or a customer. It can associate a combination of attributes describing customers and attributes describing employees with the same entity. An application can determine whether an entity represents a particular abstraction, customer or employee, simply be looking for the presence of the appropriate attributes.

3.3.1 Schema Attributes

Schema attributes are defined using the same data model used for application data. That is, attributes are part of the Datomic meta model, which specifies the characteristics (i.e., attributes) of the attributes themselves meaning attributes are themselves entities with associated attributes. Datomic defines a set of built-in system attributes that are used to define new attributes.

Every new attribute is described by three required attributes (the rest of section 3.4.1 is copied from the Datomic Reference found in Datomic Development Resources [3]):

- `:db/ident` specifies the unique name of an attribute. It's value is a namespaced keyword with the lexical form `:<namespace>/<name>`. It is possible to define a name without a namespace, as in `:<name>`, but a namespace is preferred in order to avoid naming collisions. Namespaces can be hierarchical, with segments separated by ".", as in `:<namespace>.<nested-namespace>/<name>`. The `:db` namespace is reserved for use by Datomic itself.
- `:db/valueType` specifies the type of value that can be associated with an attribute. The type is expressed as a keyword. Allowable values are listed below.

  `:db.type/keyword` – Value type for keywords. Keywords are used as names, and are interned for efficiency. Keywords map to the native interned-name type in languages that support them.

  `:db.type/string` – Value type for strings.

`:db.type/boolean` – Boolean value type.

`:db.type/long` – Fixed integer value type. Same semantics as a Java `long`: 64 bits wide, two's complement binary representation.

`:db.type/bigint` – Value type for arbitrary precision integers. Maps to `java.math.BigInteger` on Java platforms.

`:db.type/float` – Floating point value type. Same semantics as a Java `float`: single-precision 32-bit IEEE 754 floating point.

`:db.type/double` – Floating point value type. Same semantics as a Java `double`: double-precision 64-bit IEEE 754 floating point.

`:db.type/bigdec` – Value type for arbitrary precision floating point numbers. Maps to `java.math.BigDecimal` on Java platforms.

`:db.type/ref` – Value type for references. All references from one entity to another are through attributes with this value type.

`:db.type/instant` – Value type for instants in time. Stored internally as a number of milliseconds since midnight, January 1, 1970 UTC. Maps to `java.util.Date` on Java platforms.

`:db.type/uuid` – Value type for UUIDs. Maps to `java.util.UUID` on Java platforms.

`:db.type/uri` – Value type for URIs. Maps to `java.net.URI` on Java platforms.

`:db.type/bytes` – Value type for small binary data. Maps to byte array on Java platforms.

- `db/cardinality` – specifies whether an attribute associates a single value or a set of values with an entity. The values allowed for `:db/cardinality` are:

  `:db.cardinality/one` – the attribute is single valued, it associates a single value with an entity

  `:db.cardinality/many` – the attribute is multi-valued, it associates a set of values with an entity

  Transactions can add or retract individual values for multi-valued attributes.

Apart from these three required attributes there are some optional attributes which can be associated with an attribute definition:

- `:db/doc` – specifies a documentation string
- `:db/unique` – specifies a uniqueness constraint for the values of an attribute. Setting an attribute `:db/unique` also implies `:db/index`. The values allowed for `:db/unique` are:

  `:db.unique/value` – the attribute value is unique to each entity; attempts to insert a duplicate value for a different entity id will fail

  `:db.unique/identity` – the attribute value is unique to each entity and "upsert" is enabled; attempts to insert a duplicate value for a temporary entity id will cause all attributes associated with that temporary id to be merged with the entity already in the database.

`:db/unique` defaults to nil.

- `:db/index` – specifies a boolean value indicating that an index should be generated for this attribute. Defaults to false.
- `:db/fulltext` – specifies a boolean value indicating that a fulltext search index should be generated for the attribute. Defaults to false.
- *:*`db/isComponent` – specifies that an attribute whose type is `:db.type/ref` refers to a subcomponent of the entity to which the attribute is applied. When an entity is retracted with `:db.fn/retractEntity`, all subcomponents are also retracted. When an entity is touched, all its subcomponent entities are touched recursively. Defaults to nil.
- `:db/noHistory` – specifies a boolean value indicating whether past values of an attribute should not be retained. Defaults to false.

3.4 Entities

Every datom in Datomic includes a database-unique entity id. Entity ids are assigned by the transactor, and never change.

It is possible to request new entity ids by specifying a temporary id (tempid) in transaction data. The `Peer.tempid` method creates a new temporary id, and the `Peer.resolveTempid` method can be used to query a transaction return value for the actual id assigned. Internally, entity ids encode the partition an entity belongs to. An entity's partition may be useful in some cases, and can be discovered by calling the `Peer.part` method.

As mentioned above, all entities in a database have an internal key, the entity id. It is possible to use `:db/unique` and `:db/index` together to define an attribute to represent an external key. An entity may have any number of external keys, however, external keys must be single attributes, multi-attribute keys are not supported.

3.5 Transactions

Datomic represents transaction requests as data structures which provides the ability to better build requests programatically.

A transaction is simply a list of lists and/or maps, each of which is a statement in the transaction. Each list of a transaction represents either the addition or retraction of a specific fact about an entity, attribute, and value, as shown below:

```
[:db/add entity-id attribute value]
[:db/retract entity-id attribute value]
```

Each map a transaction contains is equivalent to a set of one or more `:db/add` operations. The map must include a specific `:db/id` key, identifying the entity which data is being added to (as described below). It may include any number of attribute-value pairs.

```
{:db/id entity-id
    attribute value
    attribute value
    …
}
```

However, internally, the map structure gets transformed to a list structure where each attribute-value pair becomes a `:db/add` list, using the entity-id value associated with the `:db/id` key.

```
[:db/add entity-id attribute value]
[:db/add entity-id attribute value]
. . .
```

The map structure is used as a convenience when adding data. Datomic uses an object-oriented form to present data at application-level as each entity has an id by which it can be addressed and a set of attributes which represent its state and behaviour. The attribute keys in the map may be either keywords or strings. `:db/retract` works similarly but we will not discuss it here.

In a transaction it is fundamental that all the statements must specify the entity id they apply to. An entity id may take one of three possible values:

- a temporary id for a new entity being added to the database
- an existing id for an entity that already exists in the database
- an identifier for an entity that already exists in the database

Temporary ids are generated by calling the `datomic.Peer.tempid` method. The first argument to `Peer.tempid` is the name of the partition where the new entity will reside. The three partitions built into Datomic are:

- `:db.part/db` – Schema partition. It is used only for schema entities, such as attributes and partitions.
- `:db.part/tx` – Transaction partition. It is used only for transaction entities, which are automatically created for each committed transaction.
- `:db.part/user` – User partition. It is used for application entities.

For instance in order to generate new temporary id in the `:db.part/user` partition, the following statement is required:

```
temp_id = Peer.tempid(":db.part/user");
```

By default, each call to `Peer.tempid` generates a unique temporary id, however it is worth mentioning that there is an overloaded version of `Peer.tempid` which takes a negative number as an argument and returns a temporary id based on that number. If multiple invocations of `Peer.tempid` are called with the same partition and number are

called, each invocation will return the same temporary id, making it extremely useful for the construction of transactions which add references between entities.

When a transaction containing temporary ids is processed, each unique temporary id is mapped to an actual entity id. If a given temporary id is used more than once in a given transaction, all instances are mapped to the same actual entity id.

In general, unique temporary ids are mapped to new entity ids. However, there is one exception. When a fact about a new entity with a temporary id is added and one of the attributes is specified as `:db/unique :db.unique/identity`, the system will "upsert" i.e., it will map the temporary id to an existing entity if one exists with the same attribute and value (update) or will make a new entity if one does not exist (insert). All further adds in the transaction that apply to that same temporary id are applied to the "upserted" entity.

Finally, in order to add, modify or retract data about existing entities in a transaction, it is necessary to know their respective entity ids. These can be retrieved by querying the database for an external key.

3.5.1 Adding data to a new entity

In order to add data to a new entity, a transaction must be built using `:db/add` implicitly or (explicitly with the list structure), a temporary id and the attributes and values to be added.

For instance in order to add an entity with two attributes, `:Var/name` and `:Var/type`:

```
[{:db/id #db/id[:db.part/user]
  :Var/name "java.lang.Object.finalize/@this"
  :Var/type "java.lang.Object"}]
```

The same transaction can be constructed using Java code:

```
temp_id = Peer.tempid(":db.part/user");
tx = Util.list( Util.map( ":db/id", tempid,
                ":Var/name", "java.lang.Object.finalize/@this",
                ":Var/type", "java.lang.Object")
);
```

Note that there is no requirement and restrictions about which attributes are added to which entities, this is left entirely up to the application. This provides a great deal of flexibility as the system evolves.

3.6 Queries

In general, a Datalog system would have a global fact database and set of rules. Datomic's query engine instead takes databases (and as a matter of fact, many other data sources) and rule sets as inputs.

The basic job of a query is, given a set of variables and a set of clauses, to find (the set of) all of the (tuples of) variables that satisfy the clauses. A most basic query in Datomic would have the following pattern:

```
[:find variables :where clauses]
```

As an example, consider the data of Table 1:

```
[   [21005 :Var/name java.lang.Object.toString/$r4]
    [21006 :Var/name java.lang.String.moveToFront/r1]
    [21007 :Var/name java.lang.Object.getClass/@this]
    [21005 :Var/type java.lang.StringBuffer]
    [21006 :Var/type java.lang.Object]
    [21007 :Var/type java.lang.Object] ]
```

A query invocation would take the following form:

```
Peer.q(query, inputs...);
```

A query could be formulated like this:

```
[:find ?e :where [?e :Var/Type "java.lang.Object"]]
```

This query has only one variable, `?e`, and one clause `[?e :Var/Type "java.lang.Object"]` and will take one input, expected to be a set of tuples with at least three components. This first kind of clause is called a data clause. By convention data clauses are shown in square brackets and other kinds of clauses in parentheses, but both designate lists. A data clause consists of constants and/or variables, and a tuple satisfies a clause if its constants match. The variables, in particular, get bound to the corresponding part of the matching tuple. All of this matching happens by position.

In this case we have the following two matches:

Table 4: First match of query with data

Query	**Data**
?e	21006
:Var/type	:Var/type
java.lang.Object	java.lang.Object

Table 5: Second match of query with data

Query	Data
?e	21007
:Var/type	:Var/type
java.lang.Object	java.lang.Object

So, the result of the above query would be:

```
[[21006, 21007]]
```

Another example, demonstrating the entity id binding and unification in order to retrieve an attribute value, would be the following:

```
[:find ?name : where
    [?e :Var/name ?name]
    [?e :Var/type "java.lang.Object"] ]
```

returning:

```
[   [java.lang.String.moveToFront/r1],
    [java.lang.Object.getClass/@this]   ]
```

This second query has two variables `?name` and `?e` and two clauses `[?e :Var/name ?name] [?e :Var/type "java.lang.Object"]`, and will take one input, expected to be a set of tuples with at least three components.

3.6.1 Unification

In the case demonstrated above, we have two clauses and both of them use the variable `?e`. When a variable name is used more than once, it must represent the same value in every clause in order to satisfy the set of clauses. Another perspective is that the reuse of `?e` causes an implicit self-join on the single data source. All of the values of `?e` in a single match are said to unify.

3.6.2 Anonymous (Placeholder) Variables

A single placeholder variable '_' can be used to match certain components of the tuples in a query, for which the user does not care about, in order to get to the positions of their interest. '_' matches anything, but does not unify with itself.

For instance, the following query retrieves all the variable types:

```
[:find ?type :where [_ :Var/type ?type] ]
```

3.6.3 Querying a database

The first thing needed in order to query the database is to get its value from the connection:

```
Database db = conn.db();
```

This is a true value, it is not going to change. If db is used for several queries it is guaranteed that the answers are based upon exactly the same data from a single point in time. As already mentioned the database itself acts as a relation of 4-tuples of `[entity attribute value transaction]`.

```
;;when given a db source, finds the names of all the attributes
[:find ?name :where
             [_ :db.install/attribute ?a]
             [?a :db/ident ?name] ]
```

While this query is intended to be used against a database, its data clauses contain only three elements, not four. This is not a problem due to the fact that data clauses always omit any trailing components we don't care about, in this particular case the transaction information.

3.6.4 Expression Clauses

Expression clauses allow native Java or Clojure functions to be used inside of Datalog queries. User-defined or library functions can be used as predicates or as transformation functions. Functions or methods used in expression clauses must be pure.

There are two forms of expression clauses:

```
[(predicate …)]
[(function …) bindings]
```

The first item in an expression clause is a list designating a function or method call. If no bindings are provided, the function is presumed to be a predicate returning a boolean truth value. A predicate can be used to filter out results:

```
[:find ?e :where [?e :age ?a] [(< ?a 30)]]
```

Variables can be supplied as arguments to the predicate and the function will be called on their bound values.

Functions behave in the same manner, except that their return values can in turn bind other variables:

```
[:find ?e ?months :where [?e :age ?a] [(* ?a 12) ?months]]
```

3.6.5 Bindings

Bindings can vary, from single scalar to a tuple of results, a collection of results or a full relation (collection of tuples).

Table 6: Binding Patterns

Pattern	**Binds**
?a	Scalar
[?a ?b]	Tuple
[?a …]	Collection
[[?a ?b]]	Relation

3.7.6 Multiple Inputs

Queries can take multiple inputs, and as soon as they do, an :in clause must be specified to describe and name them:

```
[[:find ?e :in $data ?age :where [$data ?e :age ?age]]
```

The above query would be called like this:

```
Peer.q(query, data, 42);
```

The `:in` clause above indicates that the query expects two inputs and they will be referred as `$data` and `?age`. Inputs named with a leading `$` are data sources and can be matched using data clauses.

Inputs involving variables are binding patterns, and directly bind those variables. All of the binding patterns accepted for function returns listed above are also accepted for inputs. As a consequence, the user can take scalars, tuples, collections, and relations as inputs and bind their components to variables for use in the query.

3.7 Rules

Datomic Datalog offers users the ability to merge sets of `:where` clauses into named rules. These rules make query logic reusable, and also provide composability, meaning that portions of a query's logic can be bound at query time.

A rule is a named group of clauses that can be plugged into the `:where` section of a query.

Below we present the Datalog rules of Figure 1.1 in Datomic Datalog, as a set of rules:

```
[   [ (VarPointsTo ?heap ?var)
    (AssignHeapAllocation ?heap ?var) ]

[   (VarPointsTo ?heap ?to)
    (Assign ?to ?from)
    (VarPointsTo ?heap ?from) ]

[   (AssignHeapAllocation ?heap ?var)
    [?e :AssignHeapAllocation/heap ?heap]
    [?e :AssignHeapAllocation/var ?var] ]

[   (Assign ?to ?from)
    [?e :Assign/to ?to]
    [?e :Assign/from ?from] ] ]
```

As with transactions and queries, rules are described using data structures. A rule is a list of lists. The first list in the rule is the `head`, naming the rule and specifying its parameters. The rest of the lists are clauses that make up the body of the rule.

In the first rule, the rule-name is `VarPointsTo`, the variables `?heap` and `?var` are input arguments, and the body is a single rule invocation testing whether another rule, namely `AssignHeapAllocation`, is satisfied by `?heap` and `?var`.

In the second rule the body is composed of two rule invocations, the first being `(Assign ?to ?from)` and the second being a reuse of `VarPointsTo` with different input arguments–`?to` and `?from`, creating a recursive rule.

The last two rules have bodies consisting of two data clauses each. For instance, in the case of the third rule, the output is a list of attribute value pairs, each representing an entity which satisfies the two data clauses. In particular, it returns the ?heap ?var pairs for which an entity with entity id `?e` exists with value `?heap` for its `:AssignHeapAllocation/heap` attribute and value `?var` for its `:AssignHeapAllocation/var` attribute. Rule number four follows the same logic.

The example shown above demonstrates Datomic's way of combining individual rule definitions into sets of rules. A set of rules is simply another list, containing a number of rule definitions.

In order to use a rule set in a query the following two things are necessary:

- First, the rule set has to be passed as an input source and be referenced in the `:in` section of a query using the '`%`' symbol.

- Second, one or more rules have to be invoked from the `:where` section of a query. This is done by adding a rule invocation clause. Rule invocations have this structure:

```
(rule-name rule-arg*)
```

A rule invocation is a list containing a rule-name and one or more arguments, either variables or constants, as defined in the rule head. It's idiomatic to use parentheses instead of square brackets to represent a rule invocation in literal form, because it makes it easier to differentiate from a data clause. However, it is not a requirement.

For instance in the following rule's body:

```
[   (VarPointsTo ?heap ?var)
    (AssignHeapAllocation ?heap ?var) ]
```

`(AssignHeapAllocation ?heap ?var)` is an invocation of the rule with rule-name "`AssignHeapAllocation`" with two rule-args, `?heap` and `?var`.

As with other where clauses, it is possible to specify a database before the rule-name to scope the rule to that database. Databases cannot be used as arguments in a rule.

```
($db rule-name rule-arg*) //not allowed
```

Furthermore, as shown in our example rules also make it possible to define different logical paths to the same conclusion (i.e., logical OR). The `VarPointsTo` rule has two definitions, the first testing whether a heap allocation assignment of `?heap` to `?var` exists and the second testing whether there is an assignment of `?from` to `?to`, with `?from` potentially pointing to `?heap`.

In the example above, the body of each rule consists solely of other rule invocations. However, rules can contain any type of clause: data, expression, or other rule invocations.

3.8 Storage Services

Datomic offers several options for persistent data storage plus the option to use the memory as a storage service. For each particular case it is necessary to start the Transactor with the appropriate properties file and then connect the Peer Library to the Transactor with a Storage Service specific URI.

Among the Storage Service options are:

- SQL database

- DynamoDB
- Riak
- Couchbase
- Infinispan memory cluster
- Cassandra
- Dev (free local storage)
- Memory

An application can be moved from one Storage Service to another simply by switching the connection string used by peers and the properties file used to start the Transactor. All are fully API-compatible.

4. Context-Insensitive Points-To Analysis in Datomic

The first part of the analysis was implemented in Datalog^{LB}. After running an analysis in Datalog^{LB}, we use a shell script, executing queries to retrieve each input fact in the workspace and to redirect the output to corresponding files, in order to obtain the input fact files for Datomic from the produced workspace of our context-insensitive analysis in Datalog^{LB}.

```
function generate_facts() {

    dest=/home/destination/path
    rm -rf $dest/*.facts

    $bloxbatch -db $database -query InstructionRef > $dest/InstructionRef.facts

    ...

    $bloxbatch -db $database -query MethodLookup > $dest/MethodLookup.facts
    $bloxbatch -db $database -query AssignCompatible > $dest/AssignCompatible.facts

}
```

Figure 4.1: The shell script used to obtain the input facts from the Datalog^{LB} workspace

The resulting input fact files for Datomic are taken as input by our Java application which performs the context-insensitive analysis. This application has three tasks. First, to read the generated input facts from the input files and convert them to Datomic seed data. Second, to create the Datomic database on the selected Storage Service (in our case the memory), parse the Schema which provides the characteristics of the attributes used for our analysis and import the converted seed data to files the Datomic database. The final task is the actual execution of the context-insensitive points-to analysis in Datomic.

4.1 Analysis Schema

The schema specifies all the necessary attributes for our analysis. Figure 4.2 shows an excerpt of our schema of attributes:

```
{:db/id #db/id[:db.part/db]
 :db/ident :MethodSignatureRef/value
 :db/valueType :db.type/string
 :db/cardinality :db.cardinality/one
 :db.install/_attribute :db.part/db }


{:db/id #db/id[:db.part/db]
 :db/ident :HeapAllocation-Type/heap
 :db/valueType :db.type/ref
 :db/unique :db.unique/value
 :db/cardinality :db.cardinality/one
 :db.install/_attribute :db.part/db }


{ :db/id #db/id[:db.part/db]
  :db/ident :HeapAllocation-Type/type
  :db/valueType :db.type/ref
```

```
  :db/cardinality :db.cardinality/one
  :db.install/_attribute :db.part/db
}
```

Figure 4.2: Excerpt of the Datomic analysis Schema

This part of the schema declares three attributes. `:db/ident` specifies the unique names of these attributes `MethodSignatureRef/value`, `HeapAllocation-Type/heap` and `HeapAllocation-Type/type`. All three attributes have cardinality value `:db.cardinality/one` meaning they associate a single value of the attribute with an entity. `HeapAllocation-Type/heap` has the `:db/unique` attribute set as `:db.unique/value` meaning that the attribute value is unique to each entity. `MethodSignatureRef/value` has its value type set as `:db.type/string` while the other two have their value types set as `:db.type/ref` which is the value type for references to other entities.

4.2 Input Facts Conversion

As shown in Figure 4.1 some of the input attributes have primitive value types such as `:db.type/string`. For such cases the conversion to Datomic entities is very simple. For `MethodSignatureRef` entities we declare the corresponding class having the same name (`MethodSignatureRef`). We decided to model entities this way as it is the exact interpretation of the object-oriented form Datomic uses to model them in transactions.

For each Datomic entity type we have declared a corresponding class and each entity is modeled as an object of its corresponding class. In total we have 58 Java classes representing Datomic entity types. For entities with dashes in their names such as `HeapAllocation-Type` the corresponding class name is the entity name without the dash (e.g., `HeapAllocationType`).

Figure 4.3 shows the `MethodSignatureRef` class:

```
public class MethodSignatureRef {
    private int id;
    private String value = null;

    public MethodSignatureRef(int id, String value) {
        this.id = id;
        this.value = value;
    }

    public String getValue() {

        return value;

    }

    public int getID() {

        return this.id;

    }
}
```

Figure 4.3: The MethodSignatureRef class

Figure 4.4 shows how we read the input facts from a file and then create and store the created class objects in memory. For each line of the input file `MethodSignatureRef.facts` we create a `MethodSignatureRef` object and add it to a data structure (an `ArrayList` in particular). The `MethodSignatureRef` class has two private members, the `id` (an `int` representing its temporary_id) and the value (a `String` representing the value of its `MethodSignatureRef/value` attribute).

This is the simple scenario where an entity has no attributes referring to other entities:

```
try(BufferedReader br = new BufferedReader(new FileReader("input-
facts/MethodSignatureRef.facts"))) {

     String line;
     while ((line = br.readLine()) != null) {
          line = line.trim();
          MethodSignatureRef m = new MethodSignatureRef(id.getID(), line);
          methodSignatureRefFactsList.add(m);

     }
     br.close();

}
```

Figure 4.4: Excerpt of the code reading from the MethodSignatureRef.facts input file, creating MethodSignatureRef class objects and adding them to an ArrayList.

We also have declared a class named `FactsID` and we create only one object of it, which produces unique temporary ids (one per entity), by decrementing a negative number, using a synchronized method named `getID()`. `getID()` has to be synchronized because we use multithreading to speed up the facts conversion procedure.

```
public class FactsID {
     private int id;

     public FactsID( int id ) {

         this.id = id;

     }

     public synchronized int getID() {
         int temp_id = id--;
         return temp_id;

     }


     public int printID() {

         return id;

     }
}
```

Figure 4.5: The FactsID class which generates temporary ids

The most crucial part of the conversion is the conversion of relationships like `HeapAllocation-Type` because both attributes for such an entity have value type

`:db.type/ref`, which means that they are references to other entities. As a result, an object of the class `HeapAllocationType` (which represents a `HeapAllocation-Type` entity) needs to have two more members other than `id`. One of them is a reference to a `Type` object and the other is a reference to a `HeapAllocationRef` object.

The `HeapAllocationType` class is shown in Figure 4.6:

```
class HeapAllocationType {

    HeapAllocationRef heapAllocationRef = null;
    Type type = null;
    int id = 0;


    public HeapAllocationType(int id, HeapAllocationRef har, Type t) {

         this.id = id;
         heapAllocationRef = har;
         type = t;

    }


    public int getID() {

         return this.id;

    }


    public Type getType() {

         return type;

    }


    public HeapAllocationRef getHeapAllocationRef() {
         return heapAllocationRef;
    }
}
```

Figure 4.6: The HeapAllocationType class representing HeapAllocation-Type entities

In general, all the references to other entities from the attribute(s) of an entity are modeled in this way during the fact conversion phase (class members referring to other classes' objects). To make things more complicated some entity attributes are references to other entities which in turn have attributes which are references to other entities and need to be modeled accordingly.

The Java code in Figure 4.7 demonstrates how such a scenario is handled in the input facts conversion part of our analysis (for `HeapAllocation-Type` entities):

```
Type type = null;
CallGraphEdgeSourceRef c = null;

for(Type type1 : typeFactsList) {

     if(type1.getValue().equals(m.group(3))) {
          type = type1;
          break;
```

```
     }
}
if (type == null)
//Exit with error if reference not found

for(CallGraphEdgeSourceRef c1 : cList) {
     if (c.getInstructionRef().getInstruction().equals(m.group(1))) {
         c = callGraphEdgeSourceRef1;
         break;
     }
}
if (c == null)
//Exit with error if reference not found
HeapAllocationRef h = new HeapAllocationRef(id.getID(), c);
heapFactsList.add(h);
HeapAllocationType hAllocType = new HeapAllocationType(id.getID(), h, type);
heapTypeFactsList.add(hAllocType);
```

Figure 4.7 Conversion of HeapAllocation-Type facts

```
public class Type {
     private String value = null;
     private int id;

     public Type( int id, String value ) {
          this.id = id;
          this.value = value;
     }

     public String getValue() {
          return this.value;
     }

     public int getID() {
          return this.id;
     }
}
```

Figure 4.8 The Type class

```
public class HeapAllocationRef {
     private CallGraphEdgeSourceRef x = null;
     private int id;

     public HeapAllocationRef( int id, CallGraphEdgeSourceRef x ) {
           this.x = x;
           this.id = id;
     }

     public int getID() {
           return this.id;
     }

     public CallGraphEdgeSourceRef getCallGraphEdgeSourceRef() {
           return this.x;
```

```
    }
}
```

Figure 4.9 The HeapAllocationRef class

The best way to explain what the code in Figure 4.7 does is through an example. Consider the following line in the file HeapAllocation-Type.facts:

```
America/Adak, java.lang.String
```

When this line is read and matched to our regular expression, `m.group(1)` has the value "`America/Adak`" and `m.group(3)` has the value "`java.lang.String`".

First, we need to search for the `Type` object whose `value` (see Figure 4.8) member is equal to "`java.lang.String`" in the `typeFactsList` and make the `type` Java variable refer to it, but for `HeapAllocationRef` the situation is more complicated, because each `HeapAllocationRef/x` attribute refers to a `CallGraphEdgeSourceRef` object whose `CallGraphEdgeSourceRef/x` atrribute in turn refers to an `InstructionRef` which has only one attribute, `InstructionRef/x` with value type `:db.type/string`. So we search `cFactsList` (containing all the `CallGraphEdgeSourceRef` objects) for the correct `CallGraphEdgeSourceRef` object which has a reference to an `InstructionRef` object which has a member of type `String` and value equal to "`America/Adak`". After the correct `CallGraphEdgeSourceRef` object (referred to by `c`) is found, we create a `HeapAllocationRef` object (referred to by `h`), which has a member named `x` (see Figure 4.9) which is a reference of type `CallGraphEdgeSourceRef`, referring to `c` and add it to `heapFactsList`. Finally, we create a `HeapAllocationType` object with references to the objects referred to by the `type` and `h` Java references and add it to `heapTypeFactsList`.

The final step of the conversion, is the generation of files containing a map for each entity mapping its `:db/id` attribute to its temporary id, and its other attributes to their corresponding values (in the case of `:db.type/ref` value types the values of attributes are again temporary ids).

Figure 4.10 shows how the files containing the seed data are generated. The first `writer.println` statement of each `try` block writes the line containing the mapping of `:db/id` to the temporary id of the entity and the other `writer.println()` statement in the first `try` block and two `writer.println()` statements in the second `try` block write the mapping of the entities' attributes to their values. In this example all attribute values are references, thus the attribute values are the temporary ids of the referenced entities:

```
try(PrintWriter writer = new PrintWriter( new BufferedWriter( new FileWriter(
"HeapAllocationRef.dtm", false)));) {

     for( HeapAllocationRef key : heapFactsList) {
          writer.println("{:db/id #db/id[:db.part/user " + key.getID() + "]");
          writer.println(" :HeapAllocationRef/x #db/id[:db.part/user " +
```

```
                                key.getCallGraphEdgeSourceRef().getID() + "]}");
     }
     writer.close();
}
try(PrintWriter writer = new PrintWriter( new BufferedWriter( new FileWriter(
"HeapAllocation-Type.dtm", false)));) {

    for( HeapAllocationType key: heapTypeFactsList ) {
         writer.println("{:db/id #db/id[:db.part/user "+key.getID() + "]");
         writer.println(" :HeapAllocation-Type/heap #db/id[:db.part/user " +
                                key.getHeapAllocationRef().getID() + ]");

         writer.println(" :HeapAllocation-Type/type #db/id[:db.part/user " +
                                key.getType().getID()+"]}");
    }
    writer.close();
}
```

Figure 4.10: Excerpt of the Java code responsible for the generation of Datomic seed data files

As shown in Figure 4.10 the `ArrayLists` containing the class objects representing each entity are traversed and their attributes are written in the file using a map structure.

Figure 4.11 shows how a generated Datomic seed data file looks:

```
{:db/id #db/id[:db.part/user -95700]
 :HeapAllocation-Type/heap #db/id[:db.part/user -95540]
 :HeapAllocation-Type/type #db/id[:db.part/user -93294]}

{:db/id #db/id[:db.part/user -96247]
 :HeapAllocation-Type/heap #db/id[:db.part/user -96244]
 :HeapAllocation-Type/type #db/id[:db.part/user -93294]}

{:db/id #db/id[:db.part/user -96755]
 :HeapAllocation-Type/heap #db/id[:db.part/user -96752]
 :HeapAllocation-Type/type #db/id[:db.part/user -93294]}
```

Figure 4.11: Datomic seed data file HeapAllocationType.dtm

This is how the mapping is performed in the above example: `-95700` is the temporary id of the `HeapAllocation-Type` entity which will be added to the database and its `HeapAllocation-Type/heap` attribute has the `:db/id` of the entity found in the user partition (`:db.part/user`) with temporary id `-95540` as its value (reference to entity). In the same manner its `HeapAllocation-Type/type` attribute has the `:db/id` of the entity found in the user partion (`:db.part/user`) with temporary id `-93294` as its value (again, reference to entity). All the temporary ids will be resolved to actual ids in the database when the transaction occurs.

4.3 Read Schema Data and Import Seed Data

After creating all the seed data files, we need to create a database in memory and then connect to it. This is done using:

```
String uri = "datomic:mem://analysis";
Peer.createDatabase(uri);
Connection conn = Peer.connect(uri);
```

Next, we need to parse the schema and add it to the database using a transaction:

```
Reader schema_rdr = new FileReader("../schema_and_seed_data/schema.dtm");
List schema_tx = (List) Util.readAll(schema_rdr).get(0);
Object txResult = conn.transact(schema_tx).get();
```

Finally, we merge all the files containing the Datomic seed data to one big file (this is done because it is necessary to add all the entities with one transaction in order to correctly resolve the temporary ids of all references in the seed data to actual ids in the database) and we read that file and insert the data with a transaction:

```
data_rdr = new FileReader("../schema_and_seed_data/seed-data.dtm");
data_tx = (List) Util.readAll(data_rdr).get(0);
txResult = conn.transact(data_tx).get();
```

After this step we are ready to perform our analysis on the seed data.

4.4 Analysis Implementation

In order to properly evaluate Datomic our analysis was implemented both iteratively and recursively.

4.4.1 Iterative Context-Insensitive Points-To Analysis

For the iterative context-insensitive analysis we first execute the queries which will output the entity ids of the main method declaration, the implicitly reachable methods and the reachable class initializer methods which are the initially reachable methods.

Figure 4.12 shows this part of the code:

```
Collection results = Peer.q("[:find ?method :where" +
"[?m :MainMethodDeclaration/method ?method]]", conn.db());
```

```
results = Peer.q("[:find ?method :where" +
"[?i :ImplicitReachable/sig ?Method]]", conn.db());


results = Peer.q("[:find ?clinit :where" +
"[?ic            :InitializedClass/classOrInterface           ?class]"           +
"[?ci                :ClassInitializer/type               ?class]"               +
"[?ci                      :ClassInitializer/method                    ?clinit]]",
conn.db());
```

Figure 4.12 Initial queries of the iterative analysis

The above queries use data clauses as explained in chapter 2. The first query retrieves the entity id of the `MethodSignatureRef` entity referred to by the `MainMethodDeclaration/method` attribute of the sole `MainMethodDeclaration` entity in the database. The entity id retrieved will be inserted into the database as value to the attribute `Reachable/method` of a `Reachable` entity (`Reachable/method` has value type `:db.type/ref`). In the same way the second query retrieves the entity ids of the `MethodSignatureRefs` of all the implicitly reachable methods and then `Reachable` entities are inserted into the database using them as attribute values. The third query behaves similarly, retrieving the entity ids of the class initializer methods' `MethodSignatureRefs` for each initialized class.

For instance Figure 4.13 shows how we insert each new `Reachable` entity to the database:

```
for (Object result : results) {
          List tx = Util.list(Util.map(":db/id", Peer.tempid(":db.part/user"),
                              ":Reachable/method", ((List) result).get(0)));
     try {
          Object txResult = conn.transact(tx).get();
     }
     catch (InterruptedException | ExecutionException ex) {
         ex.toString();
         System.exit(-1);
     }
}
```

Figure 4.13: Insertion of query results to the database

For each result in the results Collection we create a `Util.list tx` containing a single `Util.map` which maps the attribute `:db/id` to a newly generated temporary id in the user partition (`:db.part/user`) and we map the attribute `:Reachable/method` to the result's value which is the first and only element of the `List result`. Then we call `conn.transact()` with `tx` as argument to add the new entity to the database.

After this step a loop starts where we call queries to the database returning lists of entity ids which will be used as attribute values for new entities and then we add those new entities to the database. The loop terminates when fix point is reached (no further new entities to insert to the database are found).

Figure 4.14 shows some of the queries used:

```
results = Peer.q("[:find ?heap ?var :where"+
                 "[?r :Reachable/method ?inmethod]" +
                 "[?a :AssignNormalHeapAllocation/inmethod ?inmethod]"+
                 "[?a :AssignNormalHeapAllocation/heap ?heap]" +
                 "[?a :AssignNormalHeapAllocation/var ?var]]",
                 conn.db());

results = Peer.q("[:find ?type ?actual ?formal :where"+
                 "[?c :CallGraphEdge/tomethod ?method]" +
                 "[?c :CallGraphEdge/invocation ?invocation]"+
                 "[?f :FormalParam/method ?method]" +
                 "[?f :FormalParam/index ?index]" +
                 "[?f :FormalParam/var ?formal]" +
                 "[?a :ActualParam/invocation ?invocation]" +
                 "[?a :ActualParam/index ?index]" +
                 "[?a :ActualParam/var ?actual]" +
                 "[?formal :Var/type ?type]]",
                 conn.db());

results = Peer.q("[:find ?invocation ?tomethod :where" +
                 "[?r :Reachable/method ?inmethod]" +
                 "[?s :StaticMethodInvocation/inmethod ?inmethod]" +
                 "[?s :StaticMethodInvocation/invocation ?invocation]" +
                 "[?s :StaticMethodInvocation/signature ?signature]" +
                 "[?m :Method/signature ?signature]" +
                 "[?m :Method/declaration ?tomethod]]",
                 conn.db());
```

Figure 4.14: Excerpt of analysis queries

The first query returns a list of 2-element lists containing entity ids of `HeapAllocationRef` and `Var` entity pairs which form `VarPointsTo` relations. For a reachable `?inmethod` and an assignment from `?var` to `?heap` in `?inmethod` we have to insert a `VarPointsTo` entity with `?heap` and `?var` as attribute values. The `VarPointsTo` entity represents the `VarPointsTo` relation between `?var` and `?heap`. The transaction performed afterwards will insert one `VarPointsTo` entity for each 2-element list where the value of the `VarPointsTo/heap` attribute of the entity is equal to the first element of the 2-element list and the value of the `VarPointsTo/var` attribute of the entity is equal to the second element of the list.

The second query returns a list of 3-element lists of assignments from `?actual` to `?formal` where `?type` is the value of the `:Var/Type` attribute of `?formal`. Given an `?invocation` to `?method` there is an assignment from the `?actual` parameter at `?index` of the `?invocation` to the `?formal` parameter at `?index` of the `?method`.. `?actual` and `?formal` are `Var` entity ids and the value of the `:Assign/type` attribute of the `Assign` entity is the same as the value of the `:Var/type` attribute of `?formal`. The transaction which follows this query will insert one `Assign` entity for each 3-element list where the value of its `Assign/type` attribute is equal to the first element of the 3-element list, the value of its `Assign/from` attribute is equal to the second element of the 3-element list and the value of its `Assign/to` attribute is equal to the third element of the 3-element list.

The third query returns a list of 2-elements lists containing entity ids of `MethodInvocationRef` and `MethodSignatureRef` entity pairs, forming a

`CallGraphEdge` relations (there is a call graph edge from `?invocation` to `?method`). Given a reachable method `?inmethod` and an `?invocation` of the method with signature `?signature` inside `?inmethod`, where the method with signature `?signature` is declared as `?tomethod`, a 2-element list of `?invocation` and `?tomethod` values is returned. The value of `?invocation` is the entity id of a `MethodInvocationRef` entity and the value of `?tomethod` is the entity id of a `MethodSignatureRef` entity. Then we have to insert a `CallGraphEdge` entity with the `?invocation` and `?tomethod` values as attribute values.

In order to correctly reach fix point we have to keep track of the already inserted entities and subtract them from the new found ones before adding the new found ones to the database. The analysis terminates when we find zero new entities to be added in the outputs of all queries.

4.4.2 Recursive Points-To Analysis

For the recursive part of the analysis, things are much simpler. We only need to provide the query with a set of rules (using the `:in` argument as mentioned in section 3.8) and use one of the rules in the `:where` part of the query. For instance:

```
results = q( "[:find ?varValue ?z " +
             ":in $ % " +
             ":where (VarPointsTo ?heap ?var)" +
             "[?var :VarRef/name ?varValue]" +
             "[?heap :HeapAllocationRef/x ?x]" +
             "[?x :CallGraphEdgeSourceRef/x ?y]" +
             "[?y :InstructionRef/x ?z]]",
             conn.db(),
             rules );
```

Figure 4.15 Query the database using the VarPointsTo rule

In order to evaluate `VarPointsTo` all the rules will be evaluated recursively till fix point is reached. This query returns a list of 2-element lists of variable name and instruction pairs (both of `:db.type/string`).

The rules are provided by a file:

```
Reader rulesReader = new FileReader("resources/analysis.edn");
Object rules = Util.readAll(rulesReader).get(0);
```

Then the object `rules` can be used in the as input to the "`%`" argument in the `:in` part of a query. The form of a rules file is that shown in Figure 4.15:

```
[ (VarPointsTo ?heap ?var)
  (Reachable ?inmethod)
  [?a :AssignNormalHeapAllocation/inmethod ?inmethod]
  [?a :AssignNormalHeapAllocation/heap ?heap]
  [?a :AssignNormalHeapAllocation/var ?var] ]


[ (Assign ?type ?actual ?formal)
  (CallGraphEdge ?invocation ?method)
```

```
  [?f :FormalParam/method ?method]
  [?f :FormalParam/index ?index]
  [?f :FormalParam/var ?formal]
  [?a :ActualParam/invocation ?invocation]
  [?a :ActualParam/index ?index]
  [?a :ActualParam/var ?actual]
  (VarType ?formal ?type) ]


[ (Reachable ?method)
  [?m :MainMethodDeclaration/method ?method] ]

[ (Reachable ?method)
  [?i :ImplicitReachable/sig ?method] ]

[ (Reachable ?clinit)
  [?ic :InitializedClass/classOrInterface ?class]
  [?ci :ClassInitializer/type ?class]
  [?ci :ClassInitializer/method ?clinit] ]


[ (Reachable ?tomethod)
  (Reachable ?inmethod)
  [?s :StaticMethodInvocation/inmethod ?inmethod]
  [?s :StaticMethodInvocation/invocation _]
  [?s :StaticMethodInvocation/signature ?signature]
  (MethodDeclaration ?signature ?tomethod) ]
```

Figure 4.16: Excerpt of the recursive analysis rule set

As seen, each rule has its own head and the bodies of all the rules can consist either of data clauses or other rules or a combination of both (we don't use any expression clauses in our analysis). The rules in Figure 4.16 work in the same way the corresponding queries work in the iterative analysis. The main difference is that the rules will be computed recursively till fix point is reached. For instance in order to compute the `(VarPointsTo ?heap ?var)` rule, the `Rechable` rule must be computed which requires the computation of all the `Rechable` rules. In the case of the last rule, in order to compute `(Reachable ?tomethod)` we need to compute `(Reachable ?inmethod)` and this gives us an example of recursive computation. Basically, what the rule says is that if we have computed a reachable method `?inmethod` and there is a static method invocation in ?inmethod to a method with signature `?signature`, then the method `?tomethod` declared with signature `?signature` is reachable.

In Datomic no data is added to the database during the computation of the rules, so despite having evaluated all the output relations (`VarPointsTo`, `Reachable`, `CallGraphEdge` etc.) after the execution of a query, we need to execute one query per rule to get each rule's output.

As an example after the first query we have to execute another one in order to retrieve the reachable methods:

```
results = q( "[:find ?method " +
             ":in $ % " +
             ":where (Reachable ?method) ]" ,
```

```
            conn.db(),
            rules );
```

5. Evaluation

The evaluation was performed on a system running Linux Debian 7.2 64-bit. Below we present the system specifications:

- 6-core processor clocked at 3.9GHz
- 16GB of DDR3 RAM clock at 2133MHz

Additionally a 30GB swapfile was created in the File System (running on an SSD) in order to satisfy the memory requirements of Datomic. We use small benchmark programs so our analysis is mostly performed on the Java library, using the JRE 1.3.

5.1 Evaluation Method

The evaluation method was rather simple. The first step required running the Datalog^{LB} our context-insensitive analysis on a jar file and measuring the analysis time. After that, we ran the iterative analysis implemented in Datomic on the input facts obtained from Datalog^{LB} analysis workspace and measured its execution time. Finally, we ran the recursive analysis implemented in Datomic with the same input facts. The Datomic analysis is a direct translation of the Datalog^{LB} analysis rules with the same input facts so, as expected, it achieves the same precision. The two concerns of our evaluation were the execution time of the analysis part and its memory consumption.

5.2 Evaluation Results

Table 7: Execution times

Java program	Datalog^{LB} analysis runtime	Datomic iterative analysis runtime	Datomic recursive analysis runtime
Empty.jar	6.31s	1653.63s	Stopped at the 2-hour mark
Arrays.jar	6.14s	1665.34s	Stopped at the 2-hour mark
InstanceField.jar	6.17s	1662.38s	Stopped at the 2-hour mark
New.jar	6.24s	1658.83s	Stopped at the 2-hour mark
VirtualMethod.jar	6.22s	1669.31s	Stopped at the 2-hour mark
VirtualMethodParam.jar	6.17s	1668.47s	Stopped at the 2-hour mark

As Table 7 suggests our Datomic analysis implementation was orders of magnitude slower than the Datalog^{LB} analysis in all cases. The iterative Datomic analysis turned out to be more than 100 times slower than the Datalog^{LB} analysis in all cases while for the recursive Datomic analysis even for a trivial scenario such as the Empty.jar no output was received even from the first query after 18 hours. The memory consumption proved equally worrying as for instance the Datalog^{LB} analysis would peak at 1GB while the Datomic iterative analysis peaked at 13.8GB after inserting all the computed output facts to the database. The recursive analysis in Datomic, expectedly, was even more memory-consuming as it reached 30GB memory usage before the 2-hour mark. Even

before query execution the size of the Datomic database would consume between 2 and 2.5GB of memory, even for Empty.jar.

5.3 Discussion

As indicated in section 5.2, our recursive analysis in Datomic has displayed two major weaknesses, the first one being its execution times and the second being its heavy memory consumption. Our efforts to identify the problems within the implementation of our Java application performing the analysis and make improvements to it did not yield any significant results.

Potential flaws of our analysis implementation could be found in the attribute schema used by our application or the use of poor query optimization strategies in our rule set and queries. Even in our iterative analysis implementation some of the more complicated queries have demonstrated very weak performance.

Leaving the reasonable possibility of flaws in our implementation aside, we will try to address some potential weaknesses of Datomic, which we consider relative to the performance we witnessed.

5.3.1 Execution Times

As explained in section 4.4.2 Datomic does not store the results of rule evaluations to the database this possibly indicates the potential lack of semi-naive evaluation. The semi-naive evaluation strategy guarantees that no rule firing as a whole will be duplicated in subsequent iterations, meaning that already computed facts in the evaluation of a rule won't be recomputed.

Furthermore, another assumption is that Datomic probably does not implement materialized views. A materialized view is a database object that contains the results of a query. For example, it may be a local copy of data located remotely, or may be a subset of the rows and/or columns of a table or join result, or may be a summary based on aggregations of a table's data. Materialized views provide more efficient access and can drastically improve query times.

It is also worth mentioning that Datomic does not allow rules with multiple headers for cases of rules with the same body, meaning that they have to be defined and evaluated separately. For instance, the computation of the rules `(VarPointsTo ?heap ?this)`, `(CallGraphEdge ?invocation ?tomethod)`, `Reachable(?tomethod)` from a reachable virtual method invocation must be done separately, leading to inefficient computation in spite of these particular rules having the same body.

Moreover, the flexibility of the Datomic schema where the entities are mapped to attributes at the application-level may be costly, as the usage of multiple data clauses to match the attribute values of an entity may hinder the performance of queries.

Finally, in our implementation we have a lot of references to other entities in order to correctly represent our relations. It is possible that entity references increase the complexity of queries significantly.

5.3.2 Memory Consumption

First of all, Datomic runs on the JVM which is not very efficient in terms of memory, especially for such memory-intensive applications.

Datomic does not do string interning. String interning is a method of storing only one copy of each distinct string value, which must be immutable. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are stored in a string intern pool.

In our implementation we have avoided the repetition of the same string by using references to attribute values, but in most likelihood a workaround like this is not as sufficient as using string interning. As explained in section 5.3.1, our implementation uses a lot of references in order to convert the relations of Datalog^{LB} to Datomic entities which is an indication that Datomic's data model is not optimal for declaration of points-to analysis specifications, leading to heavy memory consumption.

As expected, a large initial database size increases the query memory usage, however we could not address any other reasons leading to the 30GB memory usage in our recursive analysis.

6. Conclusions

This thesis presented an evaluation of the Datomic database system for the purpose of conducting context-insensitive points-to analysis. In order to perform the evaluation we implemented a prototype context-insensitive points-to analysis both in Datomic and Datalog^{LB} and we compared the execution times and memory consumption for each engine.

Based on our measurements which showed slower execution times in Datomic by orders of magnitude and heavy memory consumption, we conclude that for the time being Datomic is not competitive enough as a tool to perform pointer analysis, at least for the schema and rule definitions tested in this work.

ABBREVIATIONS

JVM	Java Virtual Machine
ACID	Atomicity, Consistency, Isolation, Durability
JRE	Java Runtime Environment

Appendices

The appendices are structured as follows. In Appendix A the rule set of our Datomic recursive analysis can be found. In Appendix B we provide the rules for the Datalog^{LB} analysis.

A. Datomic Recursive Analysis

A.1 Recursive Analysis Rule Set

```
[
[ (VarType ?var ?type)
  [?var :Var/type ?type] ]

[ (ThisVar ?method ?this)
  [?thisVar :ThisVar/method ?method]
  [?thisVar :ThisVar/var ?this] ]

[ (HeapAllocationType ?heap ?type)
  [?heapAllocationType :HeapAllocation-Type/heap ?heap]
  [?heapAllocationType :HeapAllocation-Type/type ?type] ]

[ (MethodDeclaration ?signature ?method)
  [?methodDeclaration :Method/signature ?signature]
  [?methodDeclaration :Method/declaration ?method] ]

[ (AssignCompatible ?target ?source)
  [?assignCompatible :AssignCompatible/target ?target]
  [?assignCompatible :AssignCompatible/source ?source] ]

[ (ComponentType ?arrayType ?componentType)
  [?componentType :ComponentType/arrayType ?arrayType]
  [?componentType :ComponentType/componentType ?componentType] ]

[ (Reachable ?method)
  [?mainMethodDeclaration :MainMethodDeclaration/method ?method] ]

[ (Reachable ?method)
  [?implicitReachable :ImplicitReachable/sig ?method] ]

[ (Reachable ?clinit)
  [?initializedClass :InitializedClass/classOrInterface ?class]
  [?classInitializer :ClassInitializer/type ?class]
  [?classInitializer :ClassInitializer/method ?clinit] ]

[ (Reachable ?tomethod)
  (Reachable ?inmethod)
  [?virtualMethodInvocation :VirtualMethodInvocation/inmethod ?inmethod]
  [?virtualMethodInvocation :VirtualMethodInvocation/invocation ?invocation]
  [?virtualMethodInvocation :VirtualMethodInvocation/signature ?signature]
  [?virtualMethodInvocation :VirtualMethodInvocation/base ?base]
  [?method :Method/signature ?signature]
  [?method :Method/simplename ?simplename]
  [?method :Method/descriptor ?descriptor]
  (VarPointsTo ?heap ?base)
  (HeapAllocationType ?heap ?type)
  [?methodLookup :MethodLookup/simplename ?simplename]
  [?methodLookup :MethodLookup/descriptor ?descriptor]
```

```
  [?methodLookup :MethodLookup/type ?type]
  [?methodLookup :MethodLookup/method ?tomethod]
  (ThisVar ?tomethod _ ) ]

[ (Reachable ?tomethod)
  (Reachable ?inmethod)
  [?specialMethodInvocation :SpecialMethodInvocation/inmethod ?inmethod]
  [?specialMethodInvocation :SpecialMethodInvocation/invocation ?invocation]
  [?specialMethodInvocation :SpecialMethodInvocation/base ?base]
  (VarPointsTo _ ?base)
  [?specialMethodInvocation :SpecialMethodInvocation/signature ?signature]
  (MethodDeclaration ?signature ?tomethod)
  (ThisVar ?tomethod _ ) ]

[ (Reachable ?tomethod)
  (Reachable ?inmethod)
  [?staticMethodInvocation :StaticMethodInvocation/inmethod ?inmethod]
  [?staticMethodInvocation :StaticMethodInvocation/invocation _]
  [?staticMethodInvocation :StaticMethodInvocation/signature ?signature]
  (MethodDeclaration ?signature ?tomethod) ]

[ (CallGraphEdge ?invocation ?tomethod)
  (Reachable ?inmethod)
  [?virtualMethodInvocation :VirtualMethodInvocation/inmethod ?inmethod]
  [?virtualMethodInvocation :VirtualMethodInvocation/invocation ?invocation]
  [?virtualMethodInvocation :VirtualMethodInvocation/signature ?signature]
  [?virtualMethodInvocation :VirtualMethodInvocation/base ?base]
  (VarPointsTo ?heap ?base)
  [?method :Method/signature ?signature]
  [?method :Method/simplename ?simplename]
  [?method :Method/descriptor ?descriptor]
  (HeapAllocationType ?heap ?type)
  [?methodLookup :MethodLookup/simplename ?simplename]
  [?methodLookup :MethodLookup/descriptor ?descriptor]
  [?methodLookup :MethodLookup/type ?type]
  [?methodLookup :MethodLookup/method ?tomethod]
  (ThisVar ?tomethod _ ) ]

[ (CallGraphEdge ?invocation ?tomethod)
  (Reachable ?inmethod)
  [?specialMethodInvocation :SpecialMethodInvocation/inmethod ?inmethod]
  [?specialMethodInvocation :SpecialMethodInvocation/invocation ?invocation]
  [?specialMethodInvocation :SpecialMethodInvocation/base ?base]
  (VarPointsTo _ ?base)
  [?specialMethodInvocation :SpecialMethodInvocation/signature ?signature]
  (MethodDeclaration ?signature ?tomethod)
  (ThisVar ?tomethod _) ]

[ (CallGraphEdge ?invocation ?tomethod)
  (Reachable ?inmethod)
  [?staticMethodInvocation :StaticMethodInvocation/inmethod ?inmethod]
  [?staticMethodInvocation :StaticMethodInvocation/invocation ?invocation]
  [?staticMethodInvocation :StaticMethodInvocation/signature ?signature]
  (MethodDeclaration ?signature ?tomethod) ]

[ (VarPointsTo ?heap ?var)
  (Reachable ?inmethod)
  [?a :AssignNormalHeapAllocation/inmethod ?inmethod]
  [?a :AssignNormalHeapAllocation/heap ?heap]
  [?a :AssignNormalHeapAllocation/var ?var] ]

[ (VarPointsTo ?heap ?var)
```

```
  (Reachable ?inmethod)
  [?a :AssignAuxiliaryHeapAllocation/inmethod ?inmethod]
  [?a :AssignAuxiliaryHeapAllocation/heap ?heap]
  [?a :AssignAuxiliaryHeapAllocation/var ?var] ]

[ (VarPointsTo ?heap ?var)
  (Reachable ?inmethod)
  [?a :AssignContextInsensitiveHeapAllocation/inmethod ?inmethod]
  [?a :AssignContextInsensitiveHeapAllocation/heap ?heap]
  [?a :AssignContextInsensitiveHeapAllocation/var ?var] ]

[ (VarPointsTo ?heap ?this)
  (Reachable ?inmethod)
  [?virtualMethodInvocation :VirtualMethodInvocation/inmethod ?inmethod]
  [?virtualMethodInvocation :VirtualMethodInvocation/invocation ?invocation]
  [?virtualMethodInvocation :VirtualMethodInvocation/signature ?signature]
  [?virtualMethodInvocation :VirtualMethodInvocation/base ?base]
  (VarPointsTo ?heap ?base)
  [?method :Method/signature ?signature]
  [?method :Method/simplename ?simplename]
  [?method :Method/descriptor ?descriptor]
  (HeapAllocationType ?heap ?type)
  [?methodLookup :MethodLookup/simplename ?simplename]
  [?methodLookup :MethodLookup/descriptor ?descriptor]
  [?methodLookup :MethodLookup/type ?type]
  [?methodLookup :MethodLookup/method ?tomethod]
  (ThisVar ?tomethod ?this) ]

[ (VarPointsTo ?heap ?this)
  (Reachable ?inmethod)
  [?specialMethodInvocation :SpecialMethodInvocation/inmethod ?inmethod]
  [?specialMethodInvocation :SpecialMethodInvocation/invocation ?invocation]
  [?specialMethodInvocation :SpecialMethodInvocation/base ?base]
  (VarPointsTo ?heap ?base)
  [?specialMethodInvocation :SpecialMethodInvocation/signature ?signature]
  (MethodDeclaration ?signature ?tomethod)
  (ThisVar ?tomethod ?this) ]

[ (VarPointsTo ?heap ?to)
  (Reachable ?inmethod)
  [?assignLocal :AssignLocal/inmethod ?inmethod]
  [?assignLocal :AssignLocal/to ?to]
  [?assignLocal :AssignLocal/from ?from]
  (VarPointsTo ?heap ?from) ]

[ (VarPointsTo ?heap ?to)
  (Reachable ?inmethod)
  [?loadInstanceField :LoadInstanceField/inmethod ?inmethod]
  [?loadInstanceField :LoadInstanceField/sig ?fieldsig]
  [?loadInstanceField :LoadInstanceField/base ?base]
  [?loadInstanceField :LoadInstanceField/to ?to]
  (VarPointsTo ?heapbase ?base)
  (InstanceFieldPointsTo ?heapbase ?fieldsig ?heap) ]

[ (VarPointsTo ?heap ?to)
  (Reachable ?inmethod)
  [?loadStaticField :LoadStaticField/inmethod ?inmethod]
  [?loadStaticField :LoadStaticField/sig ?fieldsig]
  [?loadStaticField :LoadStaticField/to ?to]
  (StaticFieldPointsTo ?fieldsig ?heap) ]
```

```
[ (VarPointsTo ?heap ?to)
  (Assign ?type ?from ?to)
  (VarPointsTo ?heap ?from)
  (HeapAllocationType ?heap ?heaptype)
  (AssignCompatible ?type ?heaptype) ]

[ (VarPointsTo ?heap ?to)
  (Reachable ?inmethod)
  [?loadArrayIndex :LoadArrayIndex/inmethod ?inmethod]
  [?loadArrayIndex :LoadArrayIndex/to ?to]
  [?loadArrayIndex :LoadArrayIndex/base ?base]
  (VarPointsTo ?heapbase ?base)
  (ArrayIndexPointsTo ?heapbase ?heap)
  (VarType ?to ?type)
  (HeapAllocationType ?heapbase ?heapbasetype)
  [?componentType :ComponentType/arrayType ?heapbasetype]
  [?componentType :ComponentType/componentType ?basecomponenttype]
  (AssignCompatible ?type ?basecomponenttype) ]

[ (Assign ?type ?from ?to)
  (Reachable ?inmethod)
  [?assignCast :AssignCast/inmethod ?inmethod]
  [?assignCast :AssignCast/type ?type]
  [?assignCast :AssignCast/from ?from]
  [?assignCast :AssignCast/to ?to] ]


[ (Assign ?type ?actual ?formal)
  (CallGraphEdge ?invocation ?method)
  [?formalParam :FormalParam/method ?method]
  [?formalParam :FormalParam/index ?index]
  [?formalParam :FormalParam/var ?formal]
  [?actualParam :ActualParam/invocation ?invocation]
  [?actualParam :ActualParam/index ?index]
  [?actualParam :ActualParam/var ?actual]
  (VarType ?formal ?type) ]

[ (Assign ?type ?return ?local)
  (CallGraphEdge ?invocation ?method)
  [?assignReturnValue :AssignReturnValue/invocation ?invocation]
  [?assignReturnValue :AssignReturnValue/to ?local]
  [?returnVar :ReturnVar/method ?method]
  [?returnVar :ReturnVar/var ?return]
  (VarType ?local ?type) ]

[ (StaticFieldPointsTo ?fieldsig ?heap)
  (Reachable ?inmethod)
  [?storeStaticField :StoreStaticField/inmethod ?inmethod]
  [?storeStaticField :StoreStaticField/signature ?fieldsig]
  [?storeStaticField :StoreStaticField/from ?from]
  (VarPointsTo ?heap ?from) ]

[ (InstanceFieldPointsTo ?heapbase ?fieldsig ?heap)
  (Reachable ?inmethod)
  [?storeInstanceField :StoreInstanceField/inmethod ?inmethod]
  [?storeInstanceField :StoreInstanceField/base ?base]
  [?storeInstanceField :StoreInstanceField/from ?from]
  [?storeInstanceField :StoreInstanceField/signature ?fieldsig]
  (VarPointsTo ?heapbase ?base)
  (VarPointsTo ?heap ?from) ]

[ (ArrayIndexPointsTo ?heapbase ?heap)
```

```
  (Reachable ?inmethod)
  [?storeArrayIndex :StoreArrayIndex/inmethod ?inmethod]
  [?storeArrayIndex :StoreArrayIndex/from ?from]
  [?storeArrayIndex :StoreArrayIndex/base ?base]
  (VarPointsTo ?heapbase ?base)
  (VarPointsTo ?heap ?from)
  (HeapAllocationType ?heap ?heaptype)
  (HeapAllocationType ?heapbase ?heapbasetype)
  [?componentType :ComponentType/arrayType ?heapbasetype]
  [?componentType :ComponentType/componentType ?componenttype]
  (AssignCompatible ?componenttype ?heaptype) ]
]
```

B. Datalog^{LB} Analysis

B1. Analysis Rules

```
VarPointsTo(?heap, ?var) <-
    AssignNormalHeapAllocation(?heap, ?var, ?inmethod),
    Reachable( ?inmethod).

VarPointsTo( ?heap, ?var) <-
    AssignAuxiliaryHeapAllocation(?heap, ?var, ?inmethod),
    Reachable(?inmethod).

VarPointsTo(?heap, ?var) <-
    AssignContextInsensitiveHeapAllocation(?heap, ?var, ?inmethod),
    Reachable( ?inmethod).

VarPointsTo( ?heap, ?to ) <-
    VarPointsTo( ?heap, ?from ),
    Assign( ?type, ?from, ?to ),
    HeapAllocation:Type[?heap] = ?heaptype,
    AssignCompatible(?type, ?heaptype).

VarPointsTo(?heap, ?to ) <-
    Reachable( ?inmethod ),
    AssignLocal( ?from, ?to, ?inmethod ),
    VarPointsTo( ?heap, ?from ).


Assign( ?type, ?from, ?to ) <-
    Reachable(?inmethod),
    AssignCast(?type, ?from, ?to, ?inmethod).

Assign( ?type, ?actual, ?formal ) <-
    FormalParam[?index, ?method] = ?formal,
    ActualParam[?index, ?invocation] = ?actual,
    Var:Type[?formal] = ?type,
    CallGraphEdge( ?invocation, ?method ).

Assign( ?type, ?return, ?local ) <-
    ReturnVar( ?return, ?method ),
    CallGraphEdge( ?invocation, ?method ),
    Var:Type[?local] = ?type,
    AssignReturnValue[?invocation] = ?local.

ArrayIndexPointsTo( ?baseheap, ?heap ) <-
    Reachable( ?inmethod ),
    StoreArrayIndex(?from, ?base, ?inmethod),
    VarPointsTo( ?baseheap, ?base ),
    VarPointsTo( ?heap, ?from),
    HeapAllocation:Type[?heap] = ?heaptype,
    HeapAllocation:Type[?baseheap] = ?baseheaptype,
    ComponentType[?baseheaptype] = ?componenttype,
    AssignCompatible(?componenttype, ?heaptype).

VarPointsTo( ?heap, ?to ) <-
    Reachable( ?inmethod ),
    LoadArrayIndex( ?base, ?to, ?inmethod ),
    VarPointsTo( ?baseheap, ?base ),
    ArrayIndexPointsTo( ?baseheap, ?heap ),
    Var:Type[?to] = ?type,
    HeapAllocation:Type[?baseheap] = ?baseheaptype,
```

```
    ComponentType[?baseheaptype] = ?basecomponenttype,
    AssignCompatible(?type, ?basecomponenttype).

VarPointsTo( ?heap, ?to) <-
    Reachable(?inmethod),
    LoadInstanceField(?base, ?signature, ?to, ?inmethod ),
    VarPointsTo( ?baseheap, ?base ),
    InstanceFieldPointsTo( ?heap, ?signature, ?baseheap ).

InstanceFieldPointsTo( ?heap, ?signature, ?baseheap) <-
    Reachable(?inmethod),
    StoreInstanceField(?from, ?base, ?signature, ?inmethod),
    VarPointsTo( ?heap, ?from),
    VarPointsTo( ?baseheap, ?base).

VarPointsTo( ?heap, ?to ) <-
    Reachable( ?inmethod ),
    LoadStaticField( ?signature, ?to, ?inmethod ),
    StaticFieldPointsTo( ?heap, ?signature ).

StaticFieldPointsTo( ?heap, ?signature ) <-
    Reachable( ?inmethod ),
    StoreStaticField( ?from, ?signature, ?inmethod),
    VarPointsTo( ?heap, ?from).

Reachable( ?tomethod ),
CallGraphEdge( ?invocation, ?tomethod ) <-
    Reachable( ?inmethod ),
    StaticMethodInvocation( ?invocation, ?signature, ?inmethod ),
    MethodDeclaration[?signature] = ?tomethod.

Reachable( ?tomethod ),
CallGraphEdge( ?invocation, ?tomethod ),
VarPointsTo( ?heap, ?this ) <-
    Reachable( ?inmethod ),
    VirtualMethodInvocation( ?invocation, ?signature, ?inmethod ),
    VirtualMethodInvocation:Base[?invocation] = ?base,
    VarPointsTo( ?heap, ?base ),
    HeapAllocation:Type[?heap] = ?type,
    MethodSignature:SimpleName[?signature] = ?simplename,
    MethodSignature:Descriptor[?signature] = ?descriptor,
    ThisVar[?tomethod] = ?this,
    MethodLookup[?simplename, ?descriptor, ?type] = ?tomethod.

Reachable( ?tomethod ),
CallGraphEdge( ?invocation, ?tomethod ),
VarPointsTo( ?heap, ?this ) <-
    Reachable( ?inmethod ),
    SpecialMethodInvocation:In(?invocation, ?inmethod),
    SpecialMethodInvocation:Base[?invocation] = ?base,
    VarPointsTo( ?heap, ?base ),
    SpecialMethodInvocation:Signature[?invocation] = ?signature,
    ThisVar[?tomethod] = ?this,
    MethodDeclaration[?signature] = ?tomethod.

Reachable( ?method ) <-
    MainMethodDeclaration( ?method ).

Reachable( ?method ) <-
    ImplicitReachable( ?method ).

Reachable( ?clinit ) <-
```

```
    InitializedClass(?class),
    ClassInitializer[?class] = ?clinit.

Stats:Runtime(?value, ?attr) ->
    decimal[64](?value),
    string(?attr).
```

References

[1] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2009. ACM.

[2] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In L. Dillon, editor, *ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis*, New York, NY, USA, July 2009.

[3] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems -a case study. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, 1996.

[4] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Asian Symposium on Programming Languages and Systems*, 2005.

[5] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with Datalog. *In Proc. European Conf. on Object-Oriented Programming (ECOOP), pages 2–27*. Spinger, 2006.

[6] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1–12*, New York, NY, USA, 2005. ACM.

[7] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

[8] M.Bravenboer and Y.Smaragdakis. Using Datalog for fast and easy program analysis. In *Datalog 2.0*.

[9] W. C. Benton and C. N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. *In PPDP '07: Proc. of the 9th ACM SIGPLAN int. conf. on Principles and practice of declarative programming*, 2007.

[10] Datomic Development Resources – http://docs.datomic.com/

[11] Datalog^{LB} Programmers Guide

[12] Datomic Google group – https://groups.google.com/forum/#!forum/datomic

[13] Datomic/mbrainz-sample github repository – https://github.com/Datomic/mbrainz-sample/blob/master/