Dataflow Analysis and Datalog

Avi Saven

April 8th 2020

Dataflow Analysis Until Now: Parameterization

- So far, iterative data flow analysis are defined by a series of parameters:
  1. The direction of the analysis
  2. The values the analysis is defined over
  3. The meet operator for combining values from different paths
  4. A family transfer functions.

Dataflow Analysis Until Now: Liveness Analysis

1. Direction of the analysis: Backward
2. The values the analysis is defined over: the variables
3. How nodes are met: Union VALUES
4. Transfer function:

$$\text{LIVEOUT}(n) = \bigcup_{m \in \text{succ}(n)} \text{UE}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{DEF}(m)})$$

Dataflow Analysis Until Now: Solving

- Dataflow analyses are then executed using an iterative algorithm
  1. Start at the entry node (or end, based on direction)
  2. For each node each node, reducing using the meet operator, and then applying the transfer function.
  3. Repeating until there's no more change

Limitations

- Embedded in compilers, difficult to transfer analyses between projects
- Takes significant compiler time and space.
- No inherent parallelism in the execution
  - LLVM's dataflow framework is sequential
- Implementation of Analyses is far removed from definitions
- Analyses done using only the CPU

Datalog

- A declarative language which is a subset of Prolog.
- Not Turing Complete.
- Used for databases and dataflow analyses
- Various compilers/interpreters for datalog
  - ABCDatalog
  - bddbddb
  - Soufflé
  - Many others

Datalog: Atoms

- The most fundamental unit of datalog is the *atom*, of form $p(X_1, X_2, \ldots, X_n)$
- p is the *predicate*
  - The textbook describes the predicate as "a symbol that represents a type of statement such as 'a definition reaches the beginning of a block' "
- $X_1, X_2, \ldots, X_n$ are the terms of the predicate.
  - These can be either variables or constants.
- A *ground atom* is a predicate which has only constants in its arguments

Datalog: Atoms

- Atoms represent a true or a false value
- To represent a falsity, one writes $!p(X_1, X_2, \ldots, X_n)$

Datalog: Atom Examples

- `edge("X", "Y")`
  - There is an edge between node `X` and `Y`
- `def("X", 1)`
  - Node `X` defines variable 1
- `use("Y", A)`
  - Node `Y` uses a variable parameterized by A
  - Placeholder
  - Unknown that is filled in

Datalog: Predicates

- Predicates can be thought of as a relation, similar to how databases represent its information.
  - These relations can be turned into tables.
- Consider the following graph:

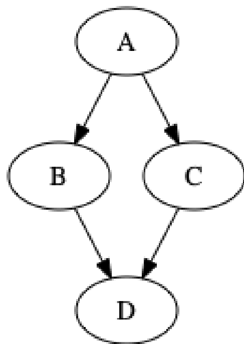


Figure 1: graph

Datalog: Predicates

- We can represent that graph with the following Datalog predicates:

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

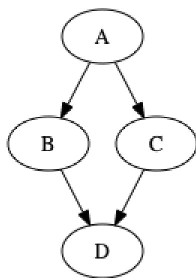
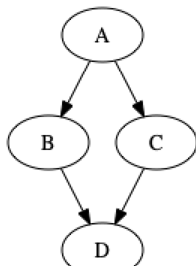


Figure 2: graph

Datalog: Predicates

- These can be represented in the following table

from	to
a	b
a	c
b	d
c	d



Datalog: Predicates

- Consider the atom `edge("a", X)`, which values of `X` make this atom true?
  - `"b"`
  - `"c"`

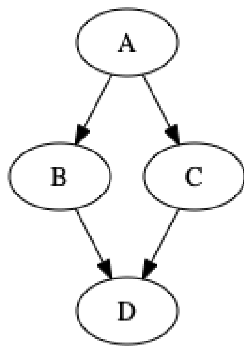


Figure 4: graph

Datalog: Rules

- Rules are of the form (where H and B_i are atoms)

$$H :\!- B_1, B_2, \ldots B_n.$$

- H is the "head" and $B_1, B_2, \ldots B_n$ is the body of the rule
- `:-` can be read as "if"
- Note, the textbook uses a different notational form than what we will use (which is what Soufflé uses).
  - Instead of ",", they use "&"
  - We can also represent the notion of OR by using ";"
    - however this is syntactic sugar for having same relation have multiple rules
  - All rules will end in a period.

Datalog: Rules Examples

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

- Read as:
  - "There is a relation `reaches(X, Y)` if there is a relation `edge(X, Y)`"
  - "There is a relation `reaches(X, Z)` if there is a relation `edge(X, Y)` and a relation `reaches(Y, Z)`"
- Can also be written

```
reaches(X, Y) :- edge(X, Y) ;
                 edge(X, Z), reaches(Z, Y).
```

Datalog: Liveness Analysis

- Consider the traditional equation for liveness analysis

$$\text{LIVEOUT}(n) = \bigcup_{m \in \text{succ}(n)} \text{UE}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{DEF}(m)})$$

- How can this be changed to a declarative program?

Datalog: Liveness Analysis

Consider the following graph:

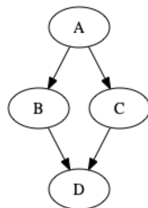


Figure 5: graph

and consider the following facts on this graph:

- A, B, C, and, D define variable `x`
- D upwardly exposes `x`

Which nodes is `x` alive at?

Datalog: Liveness Analysis

We could use the following analysis (written in Soufflé's syntax):

```
.decl edge(from: symbol, to: symbol)
.input edge

.decl ue(block: symbol, var: symbol)
.input ue

.decl def(block: symbol, var: symbol)
.input def
```

- We start by declaring input data and relations

Datalog: Liveness Analysis

```
.decl live(block: symbol, var: symbol)

live(n, v) :- edge(n, m), ue(m, v).
live(n, v) :- edge(n, m), live(m, v), !def(m, v).

.output live
```

- Nearly a direct translation from the equation.
- Express not using sets but declaring per-variable
  - Not iterating through each successor, but rather declaring the atom `edge(n, m)`
    - This has a grounded term `n` which is input from the head atom `liveout(n, v)`
    - `m` is filled in by the engine to any value that would make `edge(n, m)` true

Datalog: Liveness Analysis

- Soufflé facts are written as tab separated .facts files, where the predicate is the name of the file
- From the previous example:

```
$ ls facts
def.facts  edge.facts ue.facts
$ souffle -F facts ./liveness.dl
$ cat live.csv
B   x
C   x
```

Datalog: How is it executed?

- Many ways to execute datalog, no one prescriptive way
- One is described in Chapter 12.3.4
  - An iterative algorithm, very reminscient of the iterative dataflow analysis algorithm
  - Separates predicates into IDB and EDBs, which declares whether or not its input data or derived from rules.
  - See textbook for more details
- In this lecture previously, and the assignments, we'll be using Soufflé.

Datalog: Soufflé

- For our purposes we are going to use Soufflé (https://souffle-lang.github.io), a Datalog runtime from Oracle
- Soufflé translates your datalog into highly-parallel C++ which can then be executed natively on your processor.
- Uses various transformations such as the Magic Set transformation (described in the Soufflé documentation) and data structures to improve runtime.

Datalog: Soufflé

- Soufflé requires some extra information about the datalog in order to compile it
  - Before writing your rules, you have to declare your atoms, where each term receives a type
  - If the relation comes as an input, or is outputted, you must declare this as well
- Soufflé includes two fundamental types:
  - `symbol` is a string
  - `number` is an signed 32 bit number (note: Soufflé can be compiled to support 64 bit numbers as well)
  - Further documentation about the type system is provided in the documentation - it is relatively complex and there's an object-oriented type system included, however most analyses don't require such sophisticated types.

Demo.