Implementing Datalog

CS173 Capstone Abstract

Sam Hecht

Introduction

Datalog is a declarative logic programming language. Its primary use is as a query language for deductive databases. Some of its applications include:[1]:

- Data integration – identifying subsets of large datasets that satisfy various properties
- Program analysis – following the flow of both data and control, pointer analysis, and static code structure to optimize code and discover bugs
- Declarative networking – many network protocols can be very naturally expressed using Datalog

A Datalog program consists of facts, such as

1. `parent(dan, tom).` – Dan is Tom's parent
2. `parent(tom, bob).` – Tom is Bob's parent,

and rules, such as

1. `ancestor(X, Y) :- parent(X, Y).` – X is Y's ancestor if X is Y's parent
2. `ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).` – X is Y's ancestor if X is the parent of someone, Z, and Z is Y's ancestor.

Using this program, you can query something such as `ancestor(dan, X)`, that is, "*Who are all the X such that dan is their ancestor?*" Datalog will respond to this query with `X = tom` and `X = bob`.

How Datalog handles queries

Datalog answers queries by searching for proofs of them. In the above example, Datalog would construct the following proof to find the answer `X = bob`:

Proof. `ancestor(dan, X)` *unifies* with Rule 2 with `X = dan, Y = X`. Applying this translation, we now must prove `parent(dan, Z), ancestor(Z, X)`. `parent(dan, Z)` is proven by Fact 1, with `Z = tom`. Applying this translation to the rest of the rule, `ancestor(tom, X)` can be proven with `X = bob` by using the Rule 1 and Fact 2. □

This gives the essence of how Datalog proves claims. Given a query, Datalog finds a rule/fact that can *unify* with the query. That is, by applying some translation to each, the query and the *head* of the rule/fact become the same. This translation defines what variables need to be replaced with what, including other variables. Once finding a valid unification, Datalog applies the translation to the *body* of the rule/fact and then queries each term of the body against the program in attempt to find their proofs (facts are simply rules with no body, and thus are immediately provable). If successful, Datalog returns the translation used to prove the original query.

[1] For more information, see *Datalog and Emerging Applications: An Interactive Tutorial*, Shan Shan Huang, Todd J. Green, and Boon Thau Loo

The Project

The goal of the project was to learn more about programming language constructs, in particular, continuations and recursion, as well as formal logic, by applying them in the implementation of a Datalog interpreter. When computing a value with a single-threaded machine, it is sometimes necessary to "pause" a computation, perhaps to check the value of a button (on a webpage, e.g.) or return an initial value (returning the first solution found for a Datalog query). To be able to resume computation, you need to create a continuation, i.e., a lambda containing the rest of the computation that can be stored for later use.

To guarantee the interpreter discovers all possible solutions to a query, virtually every function has to return a continuation containing a recursive call so that the search will be exhaustive. The search space of a given program and query can be thought of as a tree. The root of the tree is the original query. The children of the query are all the heads of rules that unify with query. This is a disjunctive level of the tree (the query can be proven with rule *a or* rule *b*). Each of those nodes have a child for each term in the body of the rule, respectively. This is a conjunctive level of the tree (to prove a rule, all terms in the body must be satisfied). Proving each of these terms requires querying the program, each of which spawns a new proof subtree.

Each level of the tree alternates between disjunctive and conjunctive. The leaves of the tree are facts; although, not every branch necessarily terminates. At the disjunctive layers, any node can hypothetically lead to a solution to the query. When a solution is found for some node, the solution is returned and a continuation is constructed that will continue the search, i.e., descend the tree from the other nodes.

`Datalog.scala` contains the main class of the interpreter. The main method of the object runs various small tests of the program, with a more substantial test in
`AcademicTree.scala`. Unfortunately, the `Datalog` class only takes in the parse tree of a Datalog program, and I did not write a parser. See `Term.scala` for the classes used to represent Datalog programs.