Integrating Attribute Grammars with Datalog - CSCI 8760

Ian Kariniemi – karin010

1 Introduction

How does one describe the semantics of a programming language?

Some approaches might describe the semantics of programming language terms with a set of inference rules or as a set of mathematical objects in a semantic domain [7]. These approaches provide formalisms that can be reasoned about but are not generally intended to be run on a physical machine.

In contrast, programming language semantics for general-purpose languages will often define the semantics of their languages by an implementation running on hardware. Attribute Grammars fit in the middle of these two realms, as they are based on the declarative formalism of context-free grammars [4], but are also amenable for writing compilers.

Datalog, a declarative logic language with similar syntax to Prolog, is a language that's also based on a formalism, which in this case is horn clauses [2]. Datalog programs are roughly analogous to a set of constraints, and the meaning of a Datalog program is a valid solution to that set of constraints [2]. Datalog's termination guarantees and declarative structure make it apt for expressing program analysis problems, and modern implementations of Datalog can solve these problems with speed on par with hand-optimized implementations [8].

We've implemented a novel extension to our attribute grammar system, Silver [10], that allows compiler writers to integrate Datalog statements into the language definition, translating a programmer-defined analysis into a query to a Datalog engine. This has

the benefit of providing Silver language developers a means of more easily specifying static analysis problems involving fixpoint algorithms.

In the rest of the paper, we'll expand on our discussion of Datalog in 2, motivating our extension which we discuss in 3, then we'll compare our approach with other work in 4, and discuss future work in 5.

2 Datalog

Like Prolog, Datalog programs have a finite set of facts and rules, where facts are assertions about the world like `foo(``bar'', ``baz'')`, and rules are sentences describing how to deduce facts from other facts [2]. A clause has the following form

```
L_0 :- L_1,...,L_n
```

Clauses without a body are facts, and Clauses with a body are rules.

Unlike Prolog, Datalog can return a *set* of facts, a notion that comes from its origins as a "deductive database" language [2]. Datalog does not allow complex terms (terms that have nested terms within them, such as `suc(suc(nat))`) but does allow a limited form of recursion. Each fact in a Datalog program must be ground, and each variable that occurs in the head of a rule must also occur in the body of the same rule.

With negation, additional restrictions are added. For example, rules that have a variable appearing in a negative literal of a clause's body should also have that same variable appear in a positive literal of the clause. In addition, the clauses have to be *stratifiable* or have to be able to be partitioned into a collection of sets that each have a number assigned to them. A stratification is an assignment of those numbers to disjoint sets of clauses, and it judges recursion to be well-behaved if the following is true.

1. If a fact P can be deduced from a positive fact Q (through the application of a rule where Q can be assigned as part of the body and P as the head), then the stratification number of P needs to be greater than or equal to the stratification number of Q.

2. If a fact P can be deduced from a negated fact Q (through the application of a

rule where ¬Q is in the body and P is the head), then the stratification number of P must be strictly greater than the stratification number of Q.

This allows a form of negation and recursion that disallows negative cycles and also produces what is called a *minimal model*, a singular set of facts that a Datalog program can produce after iteration until a fixed point. Valid stratifications will put sets of clauses in dependency order, with the clauses that are not dependent on any internal facts being closer to zero. Notably, Datalog programs will also not terminate. This makes it a helpful abstraction for expressing static analysis problems that might involve mutually recursive equations, which could otherwise be difficult for programmers to reason about if implemented imperatively [8].

2.1 Motivation

Datalog programs can also be remarkably succinct and immediately understandable in user-defined terms.

A simple example is live variable analysis, as defined in [9, 1]. We take from the presentation in [9], defining it in the following way.

Let *in(b)* be the set of variables live immediately before block *b*, and let *out(b)* be the set of variables live immediately after block *b*. Let *def(b)* be the set of assigned variables in *b*, and *use(b)* be the set of used variables in *b*. The *def(b)* and *use(b)* relate to the *in(b)* and *out(b)* sets in the following way.

$in(b) = use(b) \cup (out(b)\ def(b))$
$out(b) = \bigcup_{x \in succ(b)} in(x)$

This expression can be written as the following in Datalog.

```
In(b,v) :- Use(b,v).
In(b,v) :- Out(b,v), !Def(b,v).
Out(n,v) :- Succ(n,m), Use(m,v)
```

Writing this imperatively would be more complex, requiring one to think about the order of evaluation of terms, noticing when the set of `in` and `out` values have all stopped changing and avoiding the creation of infinite loops.

By defining this in Datalog, we not only have a more straightforward representation of the algorithm but also provide the restrictions of Datalog's evaluation semantics, forcing the user to define a set of equations that will terminate.

Silver as a compiler has the advantage of being tightly integrated with the grammar and structure of the language through its Attribute grammar formalism while also containing sufficient power to work as a general-purpose Turing complete language. Our extension to Silver takes the best of both worlds, Datalog's termination guarantees and the freedom of a general-purpose programming language, assigning the best-suited portions to the appropriate realm.

3 Approach

We orient our discussion of the Silver extension around an example, a reachable definitions analysis for a small imperative language.

An example program is below:

```
1  main () {
2      Integer y;
3      y = 2;
4      Integer x;
5      x = 1 + y;
6      y = 4;
7      while (x < 4)
8          x = x+1;
9      print("execution finished\n");
10 }
```

An observant viewer will notice that the assignment to y on line 6 is not used, and since the function returns, it can't possibly be available to use afterward.

For our implementation, we defined a reachable definitions analysis that notes the dead assignment and removes it in an optimization pass, using the Datalog extension to find it.

A component of that analysis is to discover the furthest extent of assignments in the control flow graph of a program. With that information, one can examine variable references and note what definitions could be used at that reference, conservatively establishing that those definitions could be used, and should not be optimized out.

In our compiler for this language, we organize its structure mainly in terms of two nonterminal types, expressions (arithmetic expressions without side effects) and statements. Each statement above has values for the attributes `succ`, a list of references to other statements, and `assign`, a list of string names that are assigned at that statement.

We also define that each statement has the attributes `in` and `out`, both typed as a list of pairs, with the first element being a statement reference, and the second element being the name assigned in the the statement being referenced. Each pair tells us the reachable assignments going in and out, but the values are filled in due to the Datalog query.

Our example looks like the following:

```
datalog! AssignmentReachability{
  on Stmt, Root;
  computes inside rootStmt;
  with inputs: Stmt.succ, Stmt.assigns;
       outputs: Stmt.in, Stmt.out;
  in(StmtCurrent, StmtDef, Variable) :-
      out(StmtPrev,StmtDef,Variable),
      succ(StmtPrev,StmtCurrent),
      !assigns(StmtPrev,Variable).
  out(StmtCurrent,StmtCurrent,Variable) :- assigns(StmtCurrent,Variable).
  out(StmtCurrent,StmtDef,Variable) :- in(StmtCurrent,StmtDef,Variable).
}
```

The user has to consider where the results of the query should propagate to (nonterminals of type `Stmt` and `Root`), at which productions the reachability analysis should run, which attributes go into the Datalog engine (`succ` and `assign`), and which attributes come out of the Datalog engine (`in` and `out`). Defining the output attributes allows the

extension to fill in the values for these attributes. Lastly, the user defines the Datalog rules used to calculate the reachable assignments for each statement.

An essential element is the translation between Datalog facts and Silver attribute values.

Translation of Silver Facts

To represent the state of the Silver elements for Datalog, we need to interpret the values of a Silver attribute as a set of facts.

This set has to be sufficiently descriptive to represent the relationships between Silver elements, but does not have to be a complete description of the value, as Datalog is only concerned with the logical connections between Silver elements.

For these relationships, the important aspects are that Silver elements have a consistent identifier, and that the attribute value itself (which might refer to other Silver values) can be described as a logical relationship. For identifiers, If we're writing an analysis of variable liveness, there should only be one identifier for the same variable in Datalog. This is accomplished internally by using typeclasses, and for a set of Silver types typeclass instances have been implemented that provide a consistent identifier for Silver values, including references.

For describing attributes as logical relationships, we convert an attribute named `foo`, occurring on nonterminal with identifier `top`, in the following way depending on the attribute's type.

- Boolean attributes are converted either to an empty list (for false values), or a list containing one element (`[foo(top)]`)
- For a list of elements, where each element (named here as `baz` followed by a number) has an identifier typeclass instance, the list is converted to `[foo(top, baz1), foo(top,baz2), foo(top,baz3), ...]`.
- For a list of elements of length `n`, where each element is a tuple with arity `m`, and each element in the tuple (named here as `baz` followed by a number for the list element and a number for the element within the tuple), the list is converted to

```
[foo(top, baz1_1, ..., baz1_m), foo(top,baz2_1, ..., baz2_m), ..., foo(top,
bazn_1, ..., bazn_m)]
```

Propagation of Facts

Internally, the input facts have to be propagated up the tree before the Datalog engine is run with these facts as input, and after the query, the output facts have to be propagated down to fill in the Silver output attributes.

The user does have to have some awareness of this, but it is limited to providing the set of nonterminal types that facts will be extracted from and propagated down to. In the example we showed above, that set is {`Stmt`, `Root`}. If a compiler developer would like to have a nonterminal participate in this analysis (or allow facts to pass through that nonterminal), they have to provide the type of that nonterminal.

4 Related Work

Jastadd [9, 6] has an existing approach that combines attribute grammars with declarative definitions of static analyses, but it does so via explicit circular attributes. The user marks the attributes that are circular (in our example, `in` and `out` would be circular), and the attributes are evaluated until they reach a fix-point.

Jastadd users do not have to explicitly mark the dependent set of attributes, as in our approach. Instead, their attribute evaluation engine discovers the dependencies dynamically at runtime.

While Silver is also demand driven, this approach was not adopted, and we instead chose to implement this behavior via a domain-specific language extension to Silver, with Datalog as our language for expressing these fix-point semantics.

An advantage of our approach of using Datalog is that we gain termination guarantees. Jastadd does not check whether the circular attributes users' provide form a lattice of finite height or that the functions which update attributes on each iteration are monotonic [6]. Because our approach requires the user to define update equations in Datalog, a Turing incomplete language, infinite loops are not possible to express

(excluding bugs in the implementation of the Datalog engine, or extensions to the Datalog spec) [2].

Other approaches also embed Datalog in functional contexts. Flix, a a multi-paradigm language that also compiles to the JVM, supports first-class Datalog constraints [5], and the type system verifies that their Datalog programs are consistent and stratifiable. Silver does not do this kind of verification of Datalog programs.

5 Future Work

Our approach could be extended in several ways. Firstly, we could use our extension to provide pointer analysis to AbleC, our extensible language framework for the C language [3].

In addition, this approach of embedding a declarative set of constraints and mapping from Silver could be extended to other kinds of constraint solvers, such as SAT, or SMT solvers. An avenue of exploration that would be particularly useful for static analysis would be abstract interpreters.

The integration with Datalog could be refined, adding to our extension analysis of the Datalog inputs provided to ensure consistency and stratifiability, as is done in [5].

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. "What you always wanted to know about Datalog(and never dared to ask)". In: *IEEE transactions on knowledge and data engineering* 1.1 (1989), pp. 146–166.

[3] Ted Kaminski et al. "Reliable and Automatic Composition of Language Extensions to C: The AbleC Extensible Language Framework". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3138224. URL: https://doi.org/10.1145/3138224.

[4] Donald E. Knuth. "Semantics of context-free languages". In: *Mathematical Systems Theory* 2.2 (June 1968), pp. 127–145. ISSN: 1433-0490. DOI: 10.1007/bf01692511. URL: http://dx.doi.org/10.1007/BF01692511.

[5] Magnus Madsen and Ondřej Lhoták. "Fixpoints for the Masses: Programming with First-Class Datalog Constraints". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428193. URL: https://doi.org/10.1145/3428193.

[6] Eva Magnusson and Görel Hedin. "Circular Reference Attributed Grammars - their Evaluation and Applications". In: *Electronic Notes in Theoretical Computer Science* 82.3 (2003). LDTA'2003 - Language descriptions, Tools and Applications, pp. 532–554. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(05)82627-1. URL: https://www.sciencedirect.com/science/article/pii/S1571066105826271.

[7] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[8] Bernhard Scholz et al. "On fast large-scale program analysis in datalog". In: *Proceedings of the 25th International Conference on Compiler Construction*. 2016, pp. 196–206.

[9] Emma Söderberg et al. "Declarative Intraprocedural Flow Analysis of Java Source Code". In: *Electronic Notes in Theoretical Computer Science* 238 (Oct. 2009), pp. 155–171. DOI: 10.1016/j.entcs.2009.09.046.

[10] Eric Van Wyk et al. "Silver: An extensible attribute grammar system". In: *Science of Computer Programming* 75.1-2 (2010), pp. 39–54.