Implementing Dataflow Analyses for Pegasus in Datalog

Dan Licata

May 12, 2007

1 Introduction

Recent work by Lam et al. [Lam et al., 2005, Whaley et al., 2005, Whaley and Lam, 2004] has investigated the use of the logic-programming language Datalog to code program analyses, exploiting a fast implementation of Datalog using binary decision diagrams called `bddbddb`. Analyses written in Datalog are more compact and easier to reason about than hand-coded versions, while the BDD-based implementation can make their running time competitive with hand-coded versions. However, this approach has not previously been applied to the Pegasus intermediate representation [Budiu and Goldstein, 2002]. In this report, I explore the use of Datalog to implement dataflow analyses for Pegasus.

In Datalog, programs are specified as sets of inference rules defining predicates, as in Prolog. Prolog interprets inference rules by backward-chaining: for example, to show that $P(x)$ holds, find rules of the form $P(x) \leftarrow Q(x)$ and recursively show that $Q(x)$ holds; this terminates when we get to a rule with no antecedent. Datalog, on the other hand, interprets rules by forward-chaining and saturation: starting with some base set of facts, apply any rules for which the current set of facts proves all the antecedents; iterate until saturation (no new facts can be discovered). Since dataflow analyses are typically described via iteration to a fixed point, it is very natural to implement them in Datalog.

Binary decision diagrams provide a compact and canonical representation of Boolean functions. A Datalog proposition $P(x)$ over a finite type D can be represented by a Boolean function $D \rightarrow$ Bool, and therefore in turn by a BDD. The operations of forward-chaining logic programming can be viewed as relational projections and joins, which have natural and efficient implementations in terms of BDDs.

In this project, I have:

- Implemented two dataflow analyses for Pegasus, aggressive dead-code elimination and a bit-width analysis.
- Implemented partial-redundancy elimination for a traditional intermediate representation. This implementation required a technique for encoding some forms of universal quantification in Datalog.
- Benchmarked the implementation of dead-code elimination on some example programs.

Overall, I found Datalog to be a very concise and elegant language for implementing some dataflow analyses, but I found some analyses difficult or impossible to implement. Additionally, I found `bddbddb` to be reasonably fast, but naïve interfacing with it produced analyses whose running times were on the order of seconds rather than the desired tenths of seconds. Thus, in the remainder of this report, I will both describe my Datalog code and its performance and suggest extensions of Datalog and its implementation that would overcome these shortcomings.

2 Implementing Analyses in Datalog

2.1 Aggressive Dead-code Elimination

In this section, I present a simple implementation of aggressive dead-code elimination (ADCE) as a means of introducing Datalog programming. The core of ADCE is a graph reachability problem: compute the set of nodes that are reachable from given a set of roots (c.f. mark and sweep garbage collection).

First, we define a finite type representing graph nodes, along with two propositions, `root` and `edge`:

```
Node 4
root(x:Node)
edge(x:Node, y:Node)
```

Here we inhabit the type `Node` with 4 inhabitants, notated $0, 1, 2, 3$. The proposition `root(x)` is true when `x` is a root node; the proposition `edge(x,y)` is true when there is an edge from `x` to `y`. In a Datalog program, some propositions are part of the *extensional database* (EDB), which means they are presented by explicit lists of all true facts. The propositions `root` and `edge` are examples: we define them by explicitly listing the nodes on which they hold. For example, we can define a simple graph as follows:

```
root(1).

edge(0,1).
edge(2,3).
edge(1,3).
edge(3,0).
```

Next, we define an proposition `rfr(x)` which is true when the node `x` is reachable from some root node. In a Datalog program, the propositions that are not in the EDB are said to be in the *intensional database* (IDB), which means that they are defined by inference rules. The proposition `rfr` is an example:

```
rfr(x:Node)

rfr(x) :- root(x).
rfr(x) :- edge(y,x), rfr(y).
```

A rule `conclusion :- premise1, premise2, ...` means that the conclusion is true if all the premises are true. The above two rules state that `rfr(x)` is true if `x` is a root node, and that `rfr(x)` is true if there exists a `y` such that there is an edge from `y` to `x` and `y` is reachable from a root node. Note that a free variable in the premise of a rule (e.g., `y` in the second rule) is implicitly existentially quantified.

These two rules constitute a complete Datalog implementation of reachability. In `bddbddb`, computation occurs according to a forward-chaining operational semantics: starting from the propositions in the EDB, inference rules are repeatedly applied to derive new facts, which are added to the database; computation terminates when all true propositions are in the database (at which point we say that the database is *saturated*). In this example, the initial database contains the root nodes, and computation saturates the database with `rfr(x)` for all nodes `x` that are reachable from the root set.

With Prolog-style backward-chaining logic programming, these rules might not terminate on a graph with cycles (depending on the order in which the rules are applied). With forward-chaining logic programming, these rules can be run effectively. Indeed, because computation only adds facts to the database, the

computation arising from any set of inference rules is guaranteed to terminate when Datalog is restricted to only finite types—there is an upper bound on the number of true facts.[1] Consequently, any Datalog program is automatically a decision procedure.

A simple interface for using the above Datalog program to perform ADCE for Pegasus requires only a small bit of glue code:

- Before analysis, number the operations in the Pegasus graph and print out the edge relation and root set (consisting of the return, store, etc. nodes).
- After analysis, read in the reachable nodes and remove the unreachable nodes from the graph.

Relative to a hand-coded version, there is no need to manually specify the details of a graph search algorithm such as depth-first search, maintaining a visited list to account for cycles.

2.2 Partial-Redundancy Elimination; Encoding Universal Quantification

Partial-redundancy elimination (PRE) is an optimization that consolidates expressions that occur redundantly along some but not all paths in a program graph. For a traditional intermediate representation, PRE is implemented using four iterative dataflow analyses. However, these analyses are not necessary for Pegasus, as PRE can be implemented by inspecting the Pegasus graph [Budiu and Goldstein, 2002].

In this section, I present a simple Datalog implementation of PRE for a traditional IR. Aside from a technical complication, this implementation is a straightforward transcription of the definition of PRE from a standard textbook [Muchnick, 1997], with the added benefit that it can be run directly to investigate examples. The implementation also illustrates a technical device for encoding a limited form of universal quantification in Datalog; this device may be useful for implementing analyses for Pegasus as well.

Partial redundancy elimination assumes that the program is represented as a graph whose nodes are basic blocks and whose edges represent control flow. Further, it assumes that the program has been labeled with the *locally transparent* and *locally anticipatable* expressions in each basic block. An expression is locally transparent in a block if the block does not assign to the variables in the expression; an expression is locally anticipatable in a block if there is a computation of the expression in the block and the expression can be moved to the beginning of the block. In Datalog, we represent the program with the following EDB:

```
entry(b:Blk)            % entry node
exit(b:Blk)             % exit node
node(b:Blk)             % real node
edge(b1:Blk, b2:Blk)    % control edge from b1 to b2
locAnt(b:Blk, e:Exp)    % e is locally anticipatable in b
locTrans(b:Blk, e:Exp)  % e is locally transparent in b
```

The type `Blk` represents basic blocks; the type `Exp` represents expressions. The first two propositions identify the entry and exit nodes of a procedure; the third identifies all of the inhabitants of type `Blk` that actually represent basic blocks; this proposition is necessary because `bddbddb` rounds the number of inhabitants of a type up to the next power of two. The proposition `edge` represents the edges of the graph.

The propositions `locAnt` and `locTrans` define the locally anticipatable and transparent expressions in each block. In the textbook presentation of PRE, `locAnt` maps a block `b` to the set of expressions that are locally anticipatable `b`. Here, we have reformulated `locAnt` as a relation on both a block `b` and

[1] As we discuss below, the fact that all Datalog programs saturate also depends on a restriction on the use of negation.

an expression `e`, with the intention that `locAnt(b,e)` iff `e` is in the set of expressions that are locally anticipatable at `b`. The representation of `locTrans` is analogous. Because our Datalog program can manipulate only finite types, dealing directly with the set membership function is easier than representing sets of expressions explicitly.

2.2.1 Globally Anticipatable Expressions, Take 1

PRE is implemented using a succession of dataflow analyses, the first of which computes the *globally anticipatable* expression in each block. An expression is globally anticipatable on entry to a block if (1) every path from that point includes a computation of the expression and (2) the computation may be placed at any point along these paths. The following dataflow equations define this analysis:

$$\begin{aligned} ant(\texttt{exit}) &= \emptyset \\ ant(i) &= locAnt(i) \cup (locTrans(i) \cap (\bigcap_{j \in succ(i)} ant(j))) \end{aligned}$$

As with `locAnt` and `locTrans` above, we reformulate `ant` as a relation between blocks and expressions, where `ant(b,e)` if `e` is in the set of expressions that are globally anticipatable at `b`. The straightforward translation of the above dataflow equations is as follows:

```
ant(b,e)  :- !exit(b),locAnt(b,e).
ant(b1,e) :- !exit(b),locTrans(b1,e),
             (forall b2. succ(b1,b2) -> ant(b2,e)).
```

The syntax `!` negates a proposition. The `bddbddb` implementation permits *stratified negation*: the propositions in a program must be stratifiable in such a way that a rule for a proposition only negates propositions from a previous stratum. Operationally, this means that it is possible to first compute all true instances of the negated proposition, and then determine the truth of a negation by checking that a particular fact is not in the database. In this example, all that is necessary is that the EDB proposition `exit` be assigned to a stratum previous to the IDB proposition `ant`; because the EDB propositions are defined explicitly, this is clearly possible.

These two rules say that an expression is globally anticipatable at a block either if the block is not the exit and the expression is locally anticipatable or if the block is not the exit, the expression is locally transparent in the block, and the expression is globally anticipatable at all successors of the block. The universal quantification in the premise of the second rule corresponds to the fact that the dataflow equations use an intersection over all successors ($\bigcap_{j \in succ(i)} ant(j)$).

Logically, these two rules define the analysis. Unfortunately, Datalog does not permit universal quantification in the premise of a rule! Thus, we cannot use these rules directly. Fortunately, it is possible to encode some uses of universal quantification, including this one.

2.2.2 Encoding Universal Quantification

When restricted to finite domains and stratified negation, every Datalog proposition is decidable; this permits even an intuitionist to use classical reasoning. Consequently, we can encode some uses of universal quantification using existential quantification and negation.

Specifically, rather than defining `ant(b,e)` directly, we will inductively define its negation, which we will call `notant(b,e)`. Then the original proposition `ant(b,e)` can be defined by `!notant(b,e)`, which implies `ant(b,e)` (because classically $(!!A) \supset A$). The result of applying this device to the definition of `ant(b,e)` is following Datalog code:

```
notant(b:Blk, e:Exp)    % not globally anticipatable
notant(b,_)  :- exit(b).
notant(b,e)  :- !locAnt(b,e), !locTrans(b,e).
notant(b1,e) :- !locAnt(b1,e), edge(b1,b2), notant(b2,e).

ant(b:Blk, e:Exp)       % globally anticipatable
ant(b,e) :- !notant(b,e).
```

In this example, the one use of universal quantification in the definition of `ant` above has been replaced by an existential quantification in the definition of `notant` (because $!\forall x.A \equiv \exists x.!A$). Moreover, no uses of universal quantification have been introduced, so the definition of `notant` is a valid Datalog program. In general, this device will successfully eliminate uses of universal quantification when the proposition uses universal quantification but not also existential quantification; otherwise, the original existentials will produce universals in the output. Because dataflow analyses typically either meet or join over adjacent nodes, many of them satisfy this condition.

The negation of a proposition defined by inference rules can be derived using the following algorithm:

1. Form the *iff-completion* of the proposition. The iff-completion of a proposition is one proposition that summarizes all of the inference rules defining the proposition. In this example, the iff-completion is a disjunction of the two rules:

$$\begin{aligned} ant(b,e) \quad \equiv \quad & (!exit(b) \wedge locAnt(b,e)) \vee \\ & (!exit(b) \wedge locTrans(b1,e) \wedge (\forall b2.succ(b1,b2) \supset ant(b2,e))) \end{aligned}$$

   The iff-completion of a general Datalog program must also account for constrained variables in the conclusion of a rule (e.g., if we had a rule `ant(0,e)` that constrained the first argument to be a particular constant) and implicit existential quantification in the premises of the rule.

2. Negate the iff-completion and push the negation down to the leaves.

3. Put the resulting proposition into disjunctive normal form, recovering inference rules.

I manually ran this algorithm for `ant`, producing the definition of `notant` above. We will use this device several more times in the definition of PRE.

2.2.3 Implementation of PRE

Aside from the need to encode universal quantification, the definition of PRE is a straightforward transcription of the lectures notes, using the techniques I have described above. Figure 1 presents a complete implementation. Two of the three remaining analyses require the universal quantification device, as they are defined by intersecting the sets associated with adjacent nodes.

I have used this Datalog code to run the example from the textbook and lecture notes.[2]

[2]By the way, on slide 27 of the PRE notes, $ISOLout(entry)$ should be $\{a+1\}$; this typo is in Muchnick, too.

```
notant(b:Blk, e:Exp)    % not globally anticipatable
notant(b,_)  :- exit(b).
notant(b,e)  :- !locAnt(b,e), !locTrans(b,e).
notant(b1,e) :- !locAnt(b1,e), edge(b1,b2), notant(b2,e).

ant(b:Blk, e:Exp)       % globally anticipatable (ANTin)
ant(b,e) :- !notant(b,e).

earl(b:Blk, e:Exp)      % earliest (EARLin)
earl(b,_) :- entry(b).
earl(i,e) :- edge(j,i), !locTrans(j,e).
earl(i,e) :- edge(j,i), !ant(j,e), earl(j,e).

notdelay(i:Blk, e:Exp)  % not delay
notdelay(i, e) :- entry(i), !ant(i,e).
notdelay(i, e) :- entry(i), !earl(i,e).
notdelay(i, e) :- !ant(i,e), edge(j,i), locAnt(j,e).
notdelay(i, e) :- !ant(i,e), edge(j,i), notdelay(j,e).
notdelay(i, e) :- !earl(i,e), edge(j,i), locAnt(j,e).
notdelay(i, e) :- !earl(i,e), edge(j,i), notdelay(j,e).

delay(i:Blk, e:Exp)     % delay (DELAYin)
delay(i,e) :- node(i), !notdelay(i,e).

latest(i:Blk,e:Exp)     % latest (LATE)
latest(i,e) :- delay(i,e), locAnt(i,e).
latest(i,e) :- delay(i,e), edge(i,j), !delay(j,e).

notisol(i:Blk, e:Exp)   % not isolated (not ISOLout)
notisol(i,_) :- exit(i).
notisol(i,e) :- edge(i,j), !latest(j,e), locAnt(j,e).
notisol(i,e) :- edge(i,j), !latest(j,e), notisol(j,e).

opt(i:Blk, e:Exp)       % optimal (OPT)
opt(i,e) :- latest(i,e), notisol(i,e).

redn(i:Blk, e:Exp)      % redundant (REDN)
redn(i,e) :- locAnt(i,e), !latest(i,e), notisol(i,e).
```

Figure 1: Implementation of PRE for Traditional IR

2.3 Bit-Width Analysis

Because Datalog provides no built-in support for arithmetic, it is difficult to implement analyses that require much computation with numbers, such as constant propagation. However, it is possible to code up arithmetic explicitly by defining the necessary functions extensionally in the EDB. This technique becomes intractable when large numbers are required, but it is usable when the numbers are small enough. In this section, I present Datalog code for a simple bit-width analysis that determines how many bits are necessary to represent the values that flow on the wires of a Pegasus graph. This analysis is feasible because only numbers $0, \ldots, 31$ are necessary.

First, we define the necessary arithmetic operations. In this example, we will require two arithmetic operations: maximum and truncated addition (defined by $tadd(x, y) = min(x + y, 31)$). Thus, we will define an EDB containing explicit definitions of these functions as relations:

```
Width 32

max(n1:Width, n2:Width, n3:Width)
tadd(n1:Width, n2:Width, n3:Width)
# ... 32^2 rules for each ...
```

Next, we must consider how to represent the Pegasus graph. For this analysis, we require more detailed information than for ADCE: we must model the flow of values along wires. We can do so by defining a type of *wires* along with propositions defining the relationships between them. Specifically, we use the following propositions:

- `const(w:Width, out:Wire)` A constant of width `w` flows along the output wire. This proposition is used to represent constants and unknowns, which must conservatively be assumed to use all 32 bits. Unknowns include the arguments to a function, and, because we are not doing a memory-sensitive analysis, values loaded from memory.
- `addOp(in1:Wire, in2:Wire, out:Wire)` The (unsigned) sum of the inputs flows over the outputs. This proposition is used to represent additions.
- `mulOp(in1:Wire, in2:Wire, out:Wire)` The (unsigned) multiplication of the inputs flows over the outputs. This proposition is used to represent multiplication.
- `copy(w1:Wire, w2:Wire)` The input value is copied to the output. This proposition is used to represent pass-through instructions such as eta, hold, noop, and reg.
- `merge2(in1:Wire, in2:Wire, out:Wire)` One of the two inputs flows over the output. This proposition is used to model join points such as mux and mu (n-ary muxes and mus can be expanded into a succession of merges).

Note that this representation omits token and crt wires—presumably a token does not need be represented at run-time, or, if it does, is known to require only 1 bit. Additionally, this representation elides the predicate inputs to a node, since these have no effect on the width of the value that flows from the node. (A predicate wire is still declared to be the output of the node producing its value so that we determine the number of bits necessary for it.)

Using this representation of the graph, we define a proposition `width(w:Wire, n:Width)`, which means that the values flowing over `w` may use the $0, \ldots, n^{th}$ bits of a 32-bit word.

```
width(w:Wire, n:Width)

width(_,0).
width(out,n) :- const(n, out).
width(out,n) :- addOp(in1, in2, out), width(in1,n1), width(in2, n2),
                max(n1, n2, n3), tadd(n3, 1, n).
width(out,n) :- mulOp(in1, in2, out), width(in1,n1), width(in2, n2),
                tadd(n1, n2, n).
width(out,n) :- copy(in, out), width(in,n).
width(out,n) :- merge2(in1, in2, out), width(in1, n1), width(in2, n2),
                max(n1, n2, n).
```

Because we assume dead code elimination and constant propagation are implemented as separate passes, the first rule asserts that every wire uses the 0-bit (at least one non-zero number flows over it). The next rule asserts that a constant uses the bits it is declared to use. The next two rules define the width of addition and multiplication: an addition may use one more bit than the maximum of the inputs; a multiplication may use the sum of the inputs. The rule for copy propagates the input value, and the rule for merging takes the maximum.

The propositions `merge2`, `const`, and `copy` seem to be sufficient for all Pegasus operations besides the primops; to handle all Pegasus primops, it may be necessary to implement additional arithmetic operations in Datalog. Thus, while we can implement this analysis without primitive support for arithmetic, it would be more pleasant with such support.

Additionally, this Datalog program computes more facts than are necessary if, at the end of the day, we only care about the maximum `n` for which `width(w,n)`. For example, this program may spend time and space deriving `width(w,23)` when we already have derived `width(w,31)`. See below for speculation about a more efficient solution.

3 Efficiency

I have collected a bit of data in order to evaluate whether the performance of these dataflow analyses is reasonable. All of these runs were conducted on a Pentium 4 3GHz processor with 1GB memory. `bddbddb` is implemented in Java and was run with the Sun JVM build `1.5.0_06-b05`. These experiments used a version of `bddbddb` obtained from sourceforge on April 23, 2007.

I present data for the ADCE code presented above. I ran the code using three separate invocations of the JVM: the first two read in text files containing the definitions of `root` and `edge` as explicit lists of tuples and output text files describing the BDD representations of these relation. The third invocation reads in these BDD representations and performs the actual computation.

The tables in Figure 2 present the running time of these invocations, as collected by the Linux command `time`, for five example programs of various sizes. The first four programs are from the `Sample/` directories of the CASH repository. The fifth is somewhat synthetic—it is the code of `matmult` copied (and permuted slightly) enough times to be 200 lines long. In addition to the running times, I also list the program size (number of nodes in the Pegasus graph) and the true instances of each relation (for `root` and `rfr`, this number can be at most the number of nodes; for `edge`, it can be at most that squared). In the final table, the solve time column is an attempt to see what fraction of the time was spent doing the actual computation, as opposed to reading and printing; it contains the wall-clock time between the start and end of the actual computation (collected using `currentTimeMillis` before and after).

Make `root(x)` BDD (read in tuples, make BDD, print BDD):

	program size	user time (s)	sys time (s)	true instances
ccp-example3	150	.30	.04	4
adce-example	54	.41	.03	4
matmult	231	.25	.06	8
adpcm	868	.27	.04	11
matmult200	3080	.53	.01	63

Make `edge(x,y)` BDD (read in tuples, make BDD, print BDD):

	program size	user time (s)	sys time (s)	true instances
ccp-example3	150	.57	.06	286
adce-example	54	.51	.07	83
matmult	231	.59	.02	398
adpcm	868	1.1	.03	1797
matmult200	3080	4.94	.05	5418

Compute reachability `rfr(x)` and print results:

	program size	user time (s)	sys time (s)	solve time (s)	true instances
ccp-example3	150	0.74	0.08		150
adce-example	54	0.77	0.07		48
matmult	231	0.72	0.06		231
adpcm	868	0.74	0.07	0.232	868
matmult200	3080	1.15	0.10	0.539	2875

Running the empty Datalog program takes 0.250 to 0.350 seconds (user+sys).

Figure 2: Running time of ADCE

Discussion. These numbers are promising but not yet satisfactory, if the goal for an optimization is 0.1 seconds. However, there are many potential avenues for improvement:

- BDD algorithms are highly sensitive to the structure of the decision tree, which is determined by an order on the variables. I have not yet attempted to find good variable orders, beyond what `bddbddb` does by default.
- These data were collected using `bddbddb` with a Java BDD implementation. `bddbddb` seems to be set up to run a highly-tuned C BDD implementation called BuDDy, but I was unable to get this functionality to work (and the author of `bddbddb` has not yet responded to my query about it).
- Interacting with `bddbddb` in the same process as the compiler would remove the I/O and JVM startup overhead. The solve time numbers in the final table show that, in these examples, only a third to a half of the running time was actual computation.
- For large Pegasus graphs, the time to build the large `edge` BDD dominates. (This trend was corroborated by other experiments on randomly generated graphs as well.) In addition to using a faster BDD implementation and interfacing with the BDD library directly rather than going through text files, this time could be brought down by exploiting the available information better. Specifically, `bddbddb` currently "or"s each tuple in the relation with the previous BDD. However, there is no need to build the BDD using an online algorithm, and it may be more efficient to, e.g., incorporate all successors of a given node at once.

4 Related Work and Conclusion

Lam et al. [2005], Whaley et al. [2005] and Whaley and Lam [2004] describe the implementation of `bddbddb`, as well as Datalog pointer-analyses for Java and C and several other static analyses. See these papers for citations of the vast amount of prior work on Datalog, logic programming with BDDs, and other applications of BDDs. Muchnick [1997] defines PRE and other dataflow analyses. Budiu and Goldstein [2002] describe the semantics of the Pegasus internal representation. To the best of my knowledge, the application of a BDD-based implementation of Datalog to Pegasus is novel to the present work. Additionally, I at least independently re-invented the technique for encoding universal quantification used in PRE (I wouldn't be surprised if others have used the same device), and I have not seen PRE implemented using forward-chaining logic programming in the literature.

In summary, I claim that Datalog is a lovely language for writing dataflow analyses when possible: the iterative computation of forward-chaining logic programming implements the iterative aspects of these algorithms automatically. In general, analyses that associate each block or statement or node with either true or false (e.g., ADCE) or with a set of abstract values (e.g., all the PRE analyses) seem to have natural implementations. In part, this is because these analyses never need to rescind the truth of a proposition that has already been deemed true.[3]

The analyses that seem difficult to implement are those that make use of arithmetic, which Datalog provides no native support for, and those that use other kinds of lattices besides the two mentioned above. For example, our implementation of bit-width analysis computes redundant information because it is not

[3]For set-based analyses, the set of values associated with a statement does change as the analysis runs. However, in the examples in this report, we were able to decompose the set into the membership of individual elements, which individually do not change over time.

summarizing information using the proper lattice structure, and its definition is complicated by having to define arithmetic explicitly. As another example, conditional constant propagation suffers from both of these problems as well. For CCP, the lack of arithmetic seems deadly since large numbers are required. Additionally, implementing CCP in the inefficient style used for bit-width (i.e., not summarizing multiple constants as "unknown", but computing all possible values) would require computing many more extra facts.

These limitations suggest useful avenues for future research. First, it may be possible to design a constraint logic programming language based on Datalog, providing primitive support for arithmetic. Second, it may be possible to design a linear logic programming language based on Datalog, which would permit propositions whose truth changes over time. This could be used to implement analyses where the lattice value associated with a statement changes over time, such as good versions of CCP and bit-width analysis. However, the meaning of saturation is less clear when knowledge is not monotonic. The LolliMon language [López et al., 2005] is an initial attempt at forward-chaining linear logic programming, but its current implementation is not nearly as fast as the implementation of Datalog using BDDs.

The present report suggests several other avenues for future work as well. First, it would be nice to automate the encoding of universal quantification, so that the program can be written in the straightforward syntax using `forall`. More ambitiously, it may be possible to add stratified universal quantification and implication to Datalog: because all types are finite, a "forall" can be thought of as generating one premise for each element of the type; stratified implication would restrict attention to those elements of the type that satisfy a previously-computed proposition. Additionally, an extension of Datalog with first-order terms could enable the implementation of dataflow-based transformations as well as analyses.

While these extensions would be very useful for implementing dataflow analyses, it remains to be seen whether they can be integrated with Datalog without losing the benefits of BDD-based forward-chaining logic programming.

Acknowledgments. I would like to thank Jake Donham for initial discussions about this project, Rob Simmons for discussions about Datalog, and John Whaley for answering questions about `bddbddb`.

References

Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Department of Computer Science, Carnegie Mellon University, 2002.

Monica A. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *24th SIGMOD-SIGACT-SIGARTS Symposium on Principles of Database Systems*, June 2005.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Amy Felty, editor, *International Symposium on Principles and Practice of Declarative Programming*, pages 35–46, Lisbon, Portugal, 2005. ACM Press.

Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems: Third Asian Symposium,*, Tsukuba, Japan, November 2005.

John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.