15-819K: Logic Programming

Lecture 26

# Datalog

Frank Pfenning

December 5, 2006

In this lecture we describe Datalog, a decidable fragment of Horn logic. Briefly, Datalog disallows function symbols, which means that the so-called Herbrand universe of ground instances of predicates is finite. Datalog has applications in databases and, more recently, in program analysis and related problems. We also sketch a promising new way to implement Datalog via its bottom-up semantics using BDDs to represent predicates.

## 26.1 Stratified Negation

Datalog arises from Horn logic via two restrictions and an extension. The most important restriction is to disallow function symbols: terms must be variables or be drawn from a fixed set of constant symbols. The second restriction is that any variable in the head of a clause also appears in the body. Together these mean that all predicates are decidable via a simple bottom-up, forward chaining semantics, since there are only finitely many propositions that can arise. These propositions form the so-called *Herbrand universe*.

If all domains of quantification are finite, we can actually drop the restriction on variables in clause heads, since a head such as $p(x)$ just stands for finitely many instances $p(c_1), \ldots, p(c_n)$, where $c_1, \ldots, c_n$ is an enumeration of the elements of the domain of $p$.

In either case, the restriction guarantees decidability. This means it is possible to add a sound form of constructive negation, called *stratified negation*. For predicates $p$ and $q$ we say $p$ *directly depends on* $q$ if the body of a clause with head $p(\mathbf{t})$ contains $q(\mathbf{s})$. We write $p \geq q$ for the reflexive and transitive closure of the direct dependency relation. If $q$ does *not* depend on $p$ then we can decide any atom $q(\mathbf{s})$ without reference to the predicate $p$. This allows us to write clauses such as

$$p(\mathbf{t}) \leftarrow \ldots, \neg q(\mathbf{s}), \ldots$$

without ambiguity: first we can determine the extension of $q$ and then conclude $\neg q(\mathbf{s})$ for ground term $\mathbf{s}$ if $q(\mathbf{s})$ was not found to be true.

If the domains are infinite, or we want to avoid potentially explosive expansion of schematics facts, we must slightly refine our restrictions from before: any goal $\neg q(\mathbf{s})$ should be such that $\mathbf{s}$ is ground when we have to decide it, so it can be implemented by a lookup assuming that $q$ has already been saturated.

A Datalog program which is stratified in this sense can be saturated by sorting the predicates into a strict partial dependency order and then proceeding bottom-up, saturating all predicates lower in the order before moving on to predicates in a higher stratum.

Programs that are not stratified, such as

$$p \leftarrow \neg p.$$

or

$$\begin{aligned} p &\leftarrow \neg q. \\ q &\leftarrow \neg p. \end{aligned}$$

do not have such a clear semantics and are therefore disallowed. However, many technical variations of the most basic one given above have been considered in the literature.

## 26.2 Transitive Closure

A typical use of Datalog is the computation of the transitive closure of a relation. We can also think of this as computing reachability in a directed graph given the definition of the edge relation.

In the terminology of Datalog, the *extensional data base* (EDB) is given by explicit (ground) propositions $p(\mathbf{t})$. The *intensional data base* (IDB) is given by Datalog rules, including possible stratified uses of negation.

In the graph reachability example, the EDB consists of propositions $\mathsf{edge}(x, y)$ for nodes $x$ and $y$ defining the edge relation. The $\mathsf{path}$ relation, which is the transitive closure of the edge relation, is defined by two rules which constitute the IDB.

$$\begin{aligned} \mathsf{path}(x, y) &\leftarrow \mathsf{edge}(x, y). \\ \mathsf{path}(x, y) &\leftarrow \mathsf{path}(x, z), \mathsf{path}(z, y). \end{aligned}$$

## 26.3 Liveness Analysis, Revisited

As another example of the use of Datalog, we revisit the earlier problem of program analysis in a small imperative language.

$$\begin{array}{lcl} l & : & x = op(y, z) \\ l & : & \mathsf{if}\ x\ \mathsf{goto}\ k \\ l & : & \mathsf{goto}\ k \\ l & : & \mathsf{halt} \end{array}$$

We say a variable is *live* at a given program point $l$ if its value will be read before it is written when computation reaches $l$. Following McAllester, we wrote a bottom-up logic program for liveness analysis and determined its complexity using prefix firings as $O(v{\cdot}n)$ where $v$ is the number of variables and $n$ the number of instructions in the program.

This time we take a different approach, mapping the problem to Datalog. The idea is to extract from the program propositions in the initial EDB of the following form:

- $\mathsf{read}(x, l)$. Variable $x$ is read at line $l$.
- $\mathsf{write}(x, l)$. Variables $x$ is written at line $l$.
- $\mathsf{succ}(l, k)$. Line $k$ is a (potential) successor to line $l$.

The succ predicate depends on the control flow of the program so, for example, conditional jump instructions have more than one successor. In addition we will define by rules (and hence in the IDB) the predicate:

- $\mathsf{live}(x, l)$. Variable $x$ may be live at line $l$.

Like most program analyses, this is a conservative approximation: we may conclude that a variable $x$ is live at $l$, but it will never actually be read. On the other hand, if $\mathsf{live}(x, l)$ is not true, then we know for sure that $x$ can never be live at $l$. This sort of information may be used by compilers in register allocation and optimizations.

First, we describe the extraction of the EDB from the program. Every program instruction expands into a set of assertions about the program lines and program variables.

$$
\begin{array}{lcl}
l : x = op(y, z) & \leftrightarrow & \left\{ \begin{array}{l} \mathsf{read}(y, l) \\ \mathsf{read}(z, l) \\ \mathsf{write}(x, l) \\ \mathsf{succ}(l, l+1) \end{array} \right. \\[2ex]
l : \mathsf{if}\ x\ \mathsf{goto}\ k & \leftrightarrow & \left\{ \begin{array}{l} \mathsf{read}(x, l) \\ \mathsf{succ}(l, k) \\ \mathsf{succ}(l, l+1) \end{array} \right. \\[2ex]
l : \mathsf{goto}\ k & \leftrightarrow & \mathsf{succ}(l, k) \\[2ex]
l : \mathsf{halt} & \leftrightarrow & \mathit{none}
\end{array}
$$

Here we assume that the next line $l+1$ is computed explicitly at translation time.

Now the whole program analysis can be defined by just two Datalog rules.

$$
\begin{array}{lcl}
\mathsf{live}(w, l) & \leftarrow & \mathsf{read}(w, l). \\
\mathsf{live}(w, l) & \leftarrow & \mathsf{live}(w, k), \mathsf{succ}(k, l), \neg\mathsf{write}(w, l).
\end{array}
$$

The program is stratified in that live depends on read, write, and succ but not vice versa. Therefore the appeal to negation in the second clause is legitimate.

This is an extremely succinct and elegant expression of liveness analysis. Interestingly, it also provides a practical implementation as we will discuss in the remainder of this lecture.

## 26.4 Binary Decision Diagrams

There are essentially two "traditional" ways of implementing Datalog: one is by a bottom-up logic programming engine, the other using a top-down logic programming engine augmented with tabling in order to avoid non-termination. Recently, a new mechanism has been proposed using *binary decision diagrams*, which has been shown to be particularly effective in large scale program analysis.

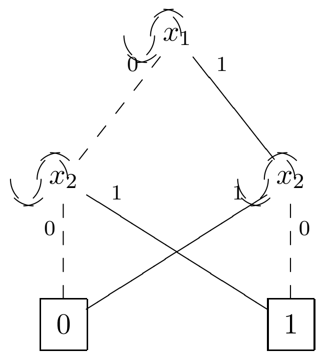
We briefly review here binary decision diagrams (BDDs). To be more precise, we will sketch the basics of *reduced ordered binary decision diagrams* (ROBDDs) which are most useful in this context for reasons we will illustrate below.

BDDs provide an often compact representation for Boolean functions. We will use this by viewing *predicates* as Boolean functions from the arguments (coded in binary) to either $1$ (when the predicate is true) or $0$ (when the predicate is false).

As an example consider the following Boolean function in two variables, $x_1$ and $x_2$.

$$\mathsf{xor}(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

We order the variables as $x_1, x_2$ and then present a diagram in which the variables are tested in the given order when read from top to bottom. When a variable is false (0) we follow the dashed line downward to the next variable, when it is true (1) we follow the solid line downward. When we reach the constant $0$ or $1$ we have determined the value of the Boolean function on the given argument values.

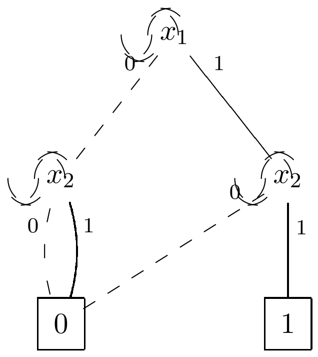


For example, to compute $\mathsf{xor}(1,1)$ we start at $x_1$ follow the solid line to the right and then another solid line to the left, ending at $0$ so $\mathsf{xor}(1,1) = 0$.

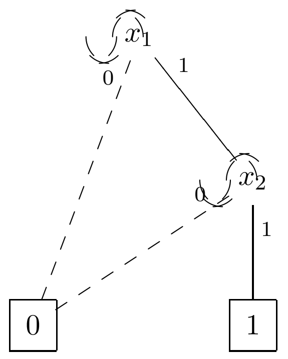
As a second simple example consider

$$\mathsf{and}(x_1, x_2) = x_1 \wedge x_2$$

If we make all choices explicit, in the given order of variables, we obtain

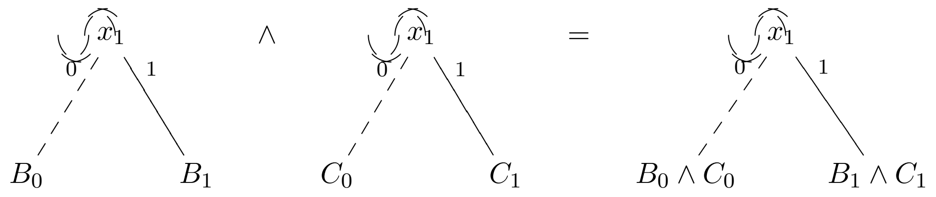


However, the test of $x_2$ is actually redundant, so we can simplify this to the following reduced diagram.



If we perform this reduction (avoiding unnecessary tests) and also share identical sub-BDDs rather than replicating them, then we call the OBDD *reduced*. Every Boolean function has a unique representation as an ROBDD once the variable order is fixed. This is one of the properties that will prove to be extremely important in the application of ROBDDs to Datalog.

Many operations on ROBDDs are straightforward and recursive, followed by reduction (at least conceptually). We will see some more examples later and now just consider conjunction. Assume we have two Boolean functions $B(x_1, \mathbf{x})$ and $C(x_1, \mathbf{x})$, where $\mathbf{x}$ represents the remaining variables. We notate $B(0, \mathbf{x}) = B_0$ and $B(1, \mathbf{x}) = B_1$ and similarly for $C$. We perform the following recursive computation



where the result may need to be reduced after the new BDD is formed. If the variable is absent from one of the sides we can mentally add a redundant node and then perform the operation as given above. On the leaves we have the equations

$$\boxed{0} \wedge B = B \wedge \boxed{0} = \boxed{0}$$

and

$$\boxed{1} \wedge B = B \wedge \boxed{1} = B$$

Other Boolean operations propagate in the same way; only the actions on the leaves are different in each case.

There are further operations which we sketch below in the example as we need them.

## 26.5 Datalog via ROBDDs

When implementing Datalog via ROBDDs we represent every predicate as a Boolean function. This is done in two steps: based on the type of the argument, we find out how many distinct constants can appear in this argument (say $n$) and then represent them with $\log_2(n)$ bits. The output of the function is always $1$ or $0$, depending on whether the predicate is true ($1$) or false ($0$). In this way we can represent the initial EDB as a collection of ROBDDs, one for each predicate.

Now we need to apply the rules in the IDB in a bottom-up manner until we have reached saturation. We achieve this by successive approximation, starting with the everywhere false predicate or some initial approximation based on the EDB. Then we compute the Boolean function corresponding to the body of each clause and combine it disjunctively with the current approximation. When the result turns out to be equal to the previous approximation we stop: saturation has been achieved. Fortunately this equality is easy to detect, since Boolean functions have unique representations.

We discuss the kind of operations required to compute the body of each clause only by example. Essentially, they are relabeling of variables, Boolean combinations such as conjunction and disjunction, and projection (which becomes an existential quantification).

As compared to traditional bottom-up strategy, where each fact is represented separately, we iterate over the whole current approximation of the predicate in each step. If the information is regular, this leads to a lot of sharing which can indeed be observed in practice.

## 26.6 An Example of Liveness Analysis

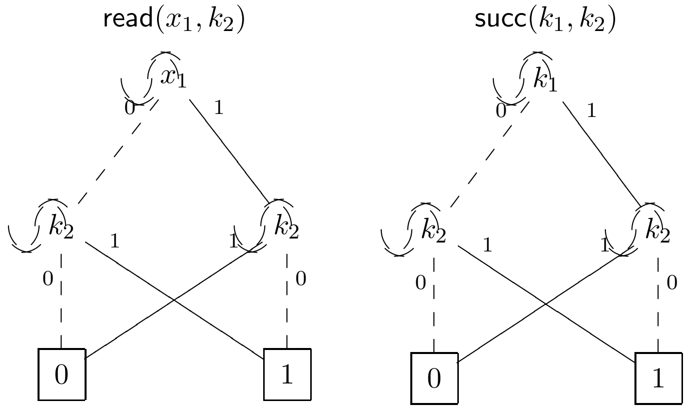
Now we walk through a small example of liveness analysis in detail in order to observe the BDD implementation of Datalog in action. Our program is very simple.

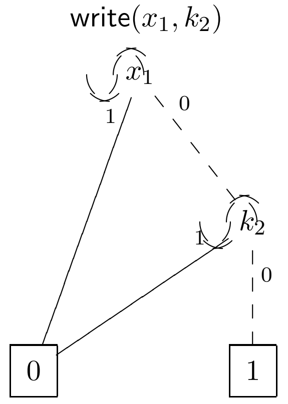
$$\begin{array}{lcl} l_0 & : & w_0 = w_1 + w_1 \\ l_1 & : & \mathsf{if}\ w_0\ \mathsf{goto}\ l_0 \\ l_2 & : & \mathsf{halt} \end{array}$$

To make things even simpler, we ignore line $l_2$ an analyse the liveness of the two variables $w_0$ and $w_1$ at the two lines $l_0$ and $l_1$. This allows us to represent both variables and program lines with a single bit each. We use $0$ for $l_0$ and $1$ for $l_1$ and similarly for the variables $w_0$ and $w_1$. Then the EDB we extract from the program is

| | |
|---|---|
| $\mathsf{read}(1,0)$ | we read $w_1$ at line $l_0$ |
| $\mathsf{read}(0,1)$ | we read $w_0$ at line $l_1$ |
| $\mathsf{succ}(0,1)$ | line $l_1$ succeeds $l_0$ |
| $\mathsf{succ}(1,0)$ | line $l_0$ succeeds $l_1$ (due to the goto) |
| $\mathsf{write}(0,0)$ | we write to $w_0$ at line $l_0$ |

Represented as a BDD, these predicates of the EDB become the following three diagrams. We write $x$ for arguments representing variables, $k$ for arguments representing line numbers, and index them by their position in the order.



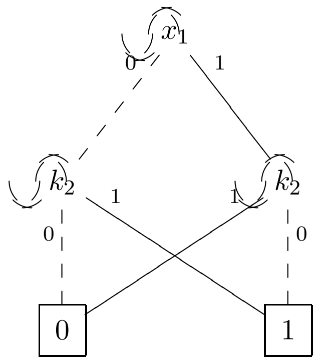


Now recall the IDB rules.

$$\begin{array}{lcl} \mathsf{live}(w,l) & \leftarrow & \mathsf{read}(w,l). \\ \mathsf{live}(w,l) & \leftarrow & \mathsf{live}(w,k), \mathsf{succ}(k,l), \neg\mathsf{write}(w,l). \end{array}$$

We initialize live with read, and use the notation $\mathsf{live}[0](x_1,k_2)$ to indicate it is the initial approximation of live.

$\mathsf{live}[0](x_1,k_2)$



This takes care of the first rule. The second rule is somewhat trickier, partly because of the recursion and partly because there is a variable on the right-hand side which does not occur on the left. This corresponds to an existential quantification, so to be explicit we write

$$\mathsf{live}(w,l) \leftarrow \exists k.\, \mathsf{live}(w,k), \mathsf{succ}(k,l), \neg\mathsf{write}(w,l).$$

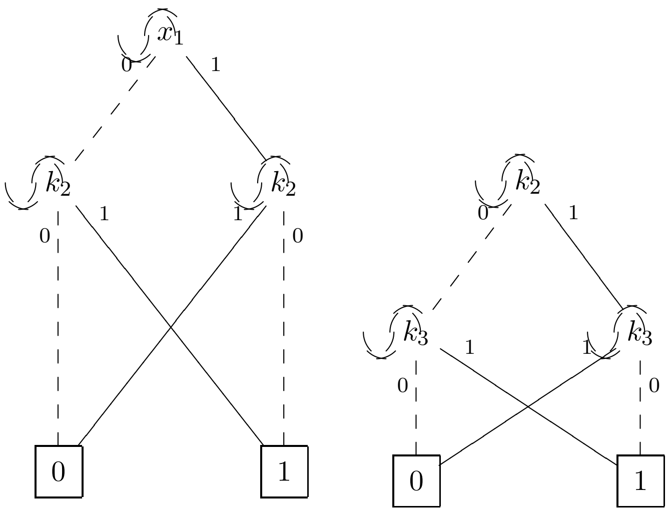
In order to compute the conjunction $\mathsf{live}[0](w,k) \wedge \mathsf{succ}(k,l)$ we need to relabel the variables so that the second argument for live is the same as the first argument for succ. To write the whole relabeled rule, also using logical notation for conjunction to emphasize the computation we have to perform:

$$\mathsf{live}(x_1,k_3) \leftarrow \exists k_2.\, \mathsf{live}(x_1,k_2) \wedge \mathsf{succ}(k_2,k_3) \wedge \neg\mathsf{write}(x_1,k_3).$$

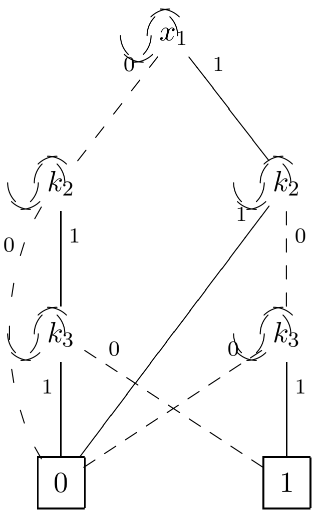
We now proceed to calculate the right hand side, given the fully computed definitions of succ and write and the initial approximation $\mathsf{live}[0]$. The intent is then to take the result and combine it disjunctively with $\mathsf{live}[0]$.

The first conjunction has the form $\mathsf{live}(x_1,k_2) \wedge \mathsf{succ}(k_2,k_3)$. Now, on each side one of the variables is not tested. We have lined up the corresponding BDDs in order to represent this relabeling.

$\mathsf{live}[0](x_1, k_2) \quad \wedge \quad \mathsf{succ}(k_2, k_3)$



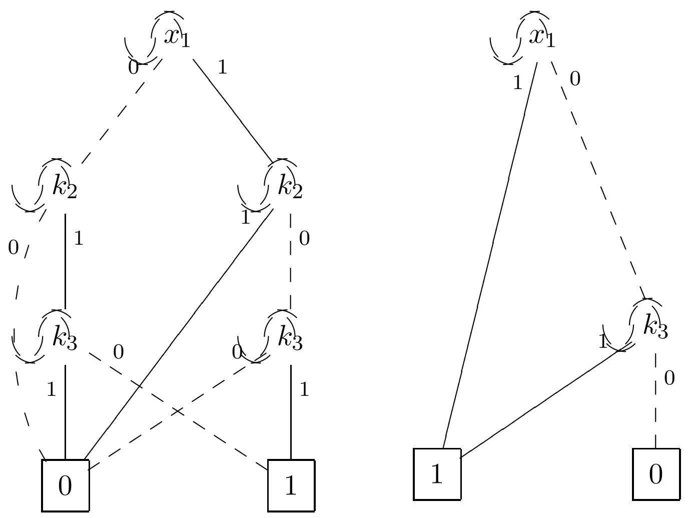
Computing the conjunction according to our recursive algorithm yields the following diagram. You are invited to carry out the computation by hand to verify that you understand this construction.



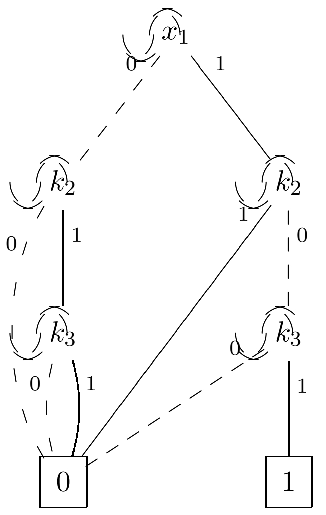
Now we have to conjoin the result with $\neg\mathsf{write}(x_1, k_3)$. We compute the negation simply by flipping the terminal 0 and 1 nodes at bottom of the

BDD, thus complementing the results of the Boolean function.

$$(\mathsf{live}[0](x_1, k_2) \wedge \mathsf{succ}(k_2, k_3)) \quad \wedge \qquad \neg\mathsf{write}(x_1, k_3)$$



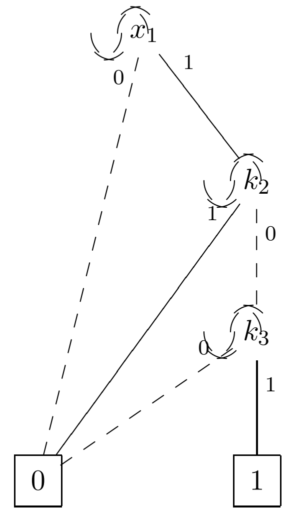
After the recursive computation we obtain the diagram below.



This diagram clearly contains some significant redundancies. First we notice that the test of $k_3$ on the left branch is redundant. Once we remove this node, the test of $k_2$ on the left-hand side also becomes redundant. In an efficient implementation this intermediate step would never have been computed in the first place, applying a technique such as hash-consing and immediately checking for cases such as the one here.

After removing both redundant tests, we obtain

$$\mathsf{live}(x_1, k_2) \wedge \mathsf{succ}(k_2, k_3) \wedge \neg\mathsf{write}(x_1, k_3)$$

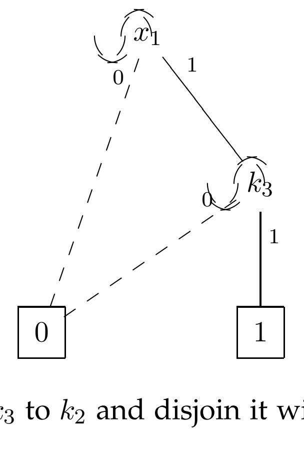


Recall that the right-hand side is

$$\exists k_2.\, \mathsf{live}(x_1, k_2) \wedge \mathsf{succ}(k_2, k_3) \wedge \neg\mathsf{write}(x_1, k_3).$$

It remains to account for the quantification over $k_2$. In general, we compute $\exists x_1.\, B(x_1, \mathbf{x})$ as $B(0, \mathbf{x}) \vee B(1, \mathbf{x})$. In this example, the disjunction appears at the second level and is easy to compute.

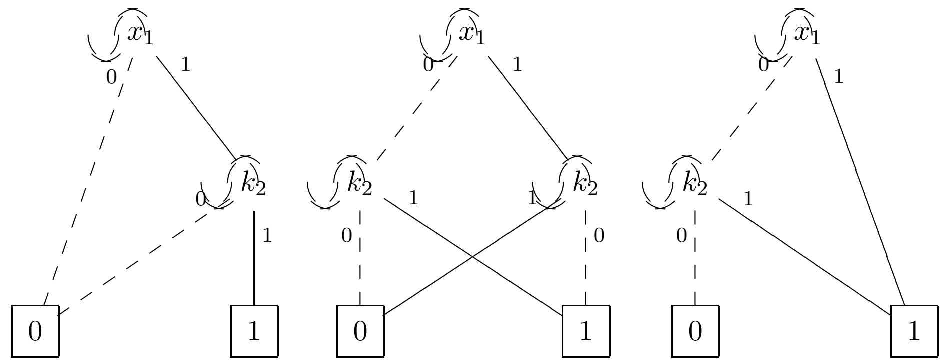
$$\mathsf{rhs}(x_1, k_3) = \exists k_2.\, \mathsf{live}(x_1, k_2) \wedge \mathsf{succ}(k_2, k_3) \wedge \neg\mathsf{write}(x_1, k_3)$$



Now we rename again, $k_3$ to $k_2$ and disjoin it with $\mathsf{live}[0](x_1, k_2)$ to obtain

$\mathsf{live}[1](x_1, k_2)$.

$$\mathsf{rhs}(x_1, k_2) \quad \vee \quad \mathsf{live}[0](x_1, k_2) \quad = \quad \mathsf{live}[1](x_1, k_2)$$



At this point we have gone through one iteration of the definition of live. Doing it one more time actually does not change the definition any more ($\mathsf{live}[2](x_1, k_2) = \mathsf{live}[1](x_1, k_2)$) so the database has reached saturation (see Exercise 26.1).

Let us interpret the result in terms of the original program.

$$\begin{array}{lcl} l_0 & : & w_0 = w_1 + w_1 \\ l_1 & : & \mathsf{if}\ w_0\ \mathsf{goto}\ l_0 \\ l_2 & : & \mathsf{halt} \end{array}$$

The line from $x_1$ to $1$ says that variable $w_1$ is live at both locations (we do not even test the location), which we can see is correct by examining the program. The path from $x_1 = 0$ through $k_2 = 1$ to $1$ states that variable $w_0$ is live at $l_1$, which is also correct since we read its value to determine whether to jump to $l_0$. Finally, the path from $x_1 = 0$ through $k_2 = 0$ to $0$ encodes that variables $w_0$ is not live at $l_0$, which is also true since $l_0$ does not read $w_1$, but writes to it. Turning this information into an explicit database form, we have derived

$$\begin{array}{l} \mathsf{live}(w_1, l_0) \\ \mathsf{live}(w_1, l_1) \\ \mathsf{live}(w_0, l_1) \\ \neg\mathsf{live}(w_0, l_0) \quad \textit{(implicitly)} \end{array}$$

where the last line would not be explicitly shown but follows from its absence in the saturated state.

While this may seem very tedious even in this small example, it has in fact shown itself to be quite efficient even for very large programs. However, a number of optimization are necessary to achieve this efficiency, as mentioned in the papers cited below.

## 26.7 Prospectus

BDDs were successfully employed in model checking for finite state systems. In our setting, this would correspond to a linear forward chaining process where only the rules are unrestricted and facts are linear and thus subject to change on each iteration. Moreover, the states (represented as linear contexts) would have to satisfy some additional invariants (for example, that each proposition occurs at most once in a context).

The research on Datalog has shown that we can use BDDs effectively for saturating forward chaining computations. We believe this can be generalized beyond Datalog if the forward chaining rules have a subterm property so that the whole search space remains finite. We only have to find a binary coding of all terms in the search space so that the BDD representation technique can be applied.

This strongly suggests that we could implement an interesting fragment of LolliMon which would encompass both Datalog and some linear logic programs subject to model checking, using BDDs as a uniform engine.

## 26.8 Historical Notes

The first use of deduction in databases is usually ascribed to a paper by Green and Raphael [3] in 1968, which already employed a form of resolution. The connection between logic programming, logic, and databases became firmly established during a workshop in Toulouse in 1977; selected papers were subsequently published in book form [2] which contained several seminal papers in logic programming. The name *Datalog* was not coined until the 1980's.

BDDs were first proposed by Randy Bryant [1]. Their use for implementing Datalog to statically analyze large programs was proposed and shown to be effective by John Whaley and collaborators [4], with a number of more specialized and further papers we do not cite here. The resulting system, bddbddb (BDD-Based Deductive DataBase) is available on SourceForge[1].

[1]`http://bddbddb.sourceforge.net/`

## 26.9 Exercises

**Exercise 26.1** *Show all the intermediate steps in the iteration from* $\mathsf{live}[1](x_1, k_2)$ *to* $\mathsf{live}[2](x_1, k_2)$ *and confirm that saturation has been reached.*

**Exercise 26.2** *Consider a 4-vertex directed graph*

$$\begin{tikzcd}
v_0 \arrow[d] & v_3 \arrow[l] \\
v_1 \arrow[r] & v_2 \arrow[ul]
\end{tikzcd}$$

*Represent the* edge *relation as a BDD and saturate the database using the two rules for transitive closure*

$$\begin{aligned}
&\mathsf{path}(x, y) \leftarrow \mathsf{edge}(x, y).\\
&\mathsf{path}(x, y) \leftarrow \mathsf{path}(x, z), \mathsf{path}(z, y).
\end{aligned}$$

*similer to the way we developed the liveness analysis example. Note that there are 4 vertices so they must be coded with 2 bits, which means that* edge *and* path *each have 4 boolean arguments, two bits for each vertex argument.*

**Exercise 26.3** *Give an encoding of another program analysis problem by showing (a) the extraction procedure to construct the initial EDB from the program, and (b) the rules defining the EDB.*

## 26.10 References

[1] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[2] Hervé Gallaire and Jack Minker, editors. *Logic and Data Bases*. Plenum Press, 1978. Edited proceedings from a workshop in Toulouse in 1977.

[3] C. Cordell Green and Bertram Raphael. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 23rd ACM National Conference*, pages 169–181, Washington, D.C., August 1968. ACM Press.

[4] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, pages 97–118, Tsukuba, Japan, November 2005. Springer LNCS 3780.