Binary decision diagrams, relational algebra, and Datalog: deductive reasoning for Tcl

Kevin B. Kenny

Abstract. Future plans for aggressive optimization of the Tcl language require making assumptions about the behaviour of Tcl scripts with respect to the predictability of their operations. For example, non-local side effects from traces, modifying the core language, and variable aliasing will defeat many optimization schemes. Determining the safety of optimizations requires, in effect, proving theorems about scripts. This paper describes a deductive database - an in-memory relational database whose values all belong to finite domains with total ordering - intended to support this effort. The database is implemented atop a library for Binary Decision Diagrams (BDD's), a compact data structure representing expressions in first-order logic. This library is used to implement multiway finite-domain decision diagrams, which represent the relations of the database. The database is in turn manipulated in a 'little language' called Datalog, a limited dialect of Prolog that allows for recursive operations impossible in a traditional programming language such as SQL. This language has been used to prototype limited versions of certain critical program analyses, such as dead code elimination, calculation of reaching definitions, and data type inference.

1) Introduction

Discussions among the Tcl developers in recent years have revealed that aggressive optimization of Tcl programs will require detailed knowledge of the data types of values. Rather than changing Tcl to require declaration of variable types – a fundamental change to the language – the author of this paper is exploring the possibility of, in a useful set of cases, inferring the types of values from the contexts in which they appear.

Doing so will require quite a lot of deductive logic: essentially, proving theorems about a program's behaviour. Useful concepts will include “calling a given procedure will establish no traces, redefine no core commands, have no side effects on variables in the caller's scope, etc., given the assumption that none of those things has happened before the procedure's execution.” Even more useful will be the small conclusions that depend on these broad safety assertions: “at this point in the program, the variable X is known to contain a native integer, and code may be generated that exploits that fact.”

The detailed assumptions are similar to the analyses that sophisticated Java compilers must go through in performing “points-to” analysis: when presented with an Object, what classes can it actually be a member of? A significant recent body of effort in this sort of analysis was done in the bddbddb system [WHAL06]. This system is quite sophisticated: it is an implementation of the Datalog query langauge atop Binary Decision Diagrams, and is capable of handling quite complex compilation problems. Unfortunately, examining the code showed that it would not fit into Tcl's way of doing things without major redesigns, and instead, a deduction system for Tcl was implemented *de novo*. This paper describes the resulting system.

In Section 2, Binary Decision Diagrams (BDD's) are presented. These are compact representations for Boolean functions over arbitrary sets of variables. Section 3 describes how BDD's can be used to represent relations in a database when column values are all drawn from totally ordered finite domains. Section 4 digresses into an important detail about application performance that informs the design of the data definition language. Section 5 provides an example of manipulating a database at the 'assembly language' level of relational algebra. Section 6 discusses briefly the language used to perform logical deductions. Section 7 offers another implementation digression, discussing the handling of logical negation. Finally Section 8 contains some preliminary concrete examples of the sort of deductions that can be drawn from Tcl code and offers some directions for future development.

2) Binary decision diagrams: the “engine”

The lowest level support for deductive reasoning in this project is a C library, `tclbdd`, that implements Binary Decision Diagrams (BDD's), a data structure first described by R.E. Bryant [BRYA86]. (A gentler introduction to Binary Decision Diagrams is available as [ANDE97]. BDD's are a compact representation for Boolean expressions of an arbitrary number of variables.

To understand how BDD's work, first consider representing a Boolean function as a complete binary tree of its truth table. At each level *N* of the tree, the value of the Nth variable is checked, and a branch is chosen according to it. (This condition is sometimes stated as requiring an *ordered* BDD, or OBDD.) The leaves of the tree are the special nodes ⊥ and ⊤, representing the constant values 'false' and 'true' respectively.

A BDD can be constructed from a complete binary tree by repeatedly applying two rewrite rules to it:

1. If both edges leaving a node *M* go to the same node *N*, eliminate the node *M* and make any edges that enter it go to *N* instead.
2. If two or more nodes exist that test the same variable and have edges that go to the same pair of nodes, coalesce them into a single node.

These conditions are sometimes described as requiring a *reduced*, ordered BDD (ROBDD).

Figure 1 shows this process applied to the diagram that represents the Boolean formula, $(A \vee B) \wedge C$. In the first step, the node that tests variable C and has both edges going to ⊥ is removed. The three nodes that all test variable *C* and go to ⊥ if the value is 0 and ⊤ if it is 1 are all collapsed down into a single node. In the second step, the node that tests variable B but goes to the same node for both its values is removed.

BDD's have a number of properties that make them useful for representing complex Boolean formulas:

- While in general their size is exponential in the number of variables, for most practical problems the size has a small polynomial bound. In some sense, the functions whose BDD's grow exponentially are uninteresting. For example, the complete binary tree for the N-ary EXCLUSIVE OR function grows exponentially with N, while the BDD grows only linearly.
- Any function has a unique representation as a BDD, enabling functions to be tested for equality (or evaluated for tautology or satisfiability) in constant time.

Well-understood algorithms are available for manipulating BDD's:

- combining them using Boolean algebra, with unary, binary and ternary operators.
- applying the quantifiers ∃ and ∀.
- composition: replacing a variable in an function with another expression or renaming varaiables in a function.

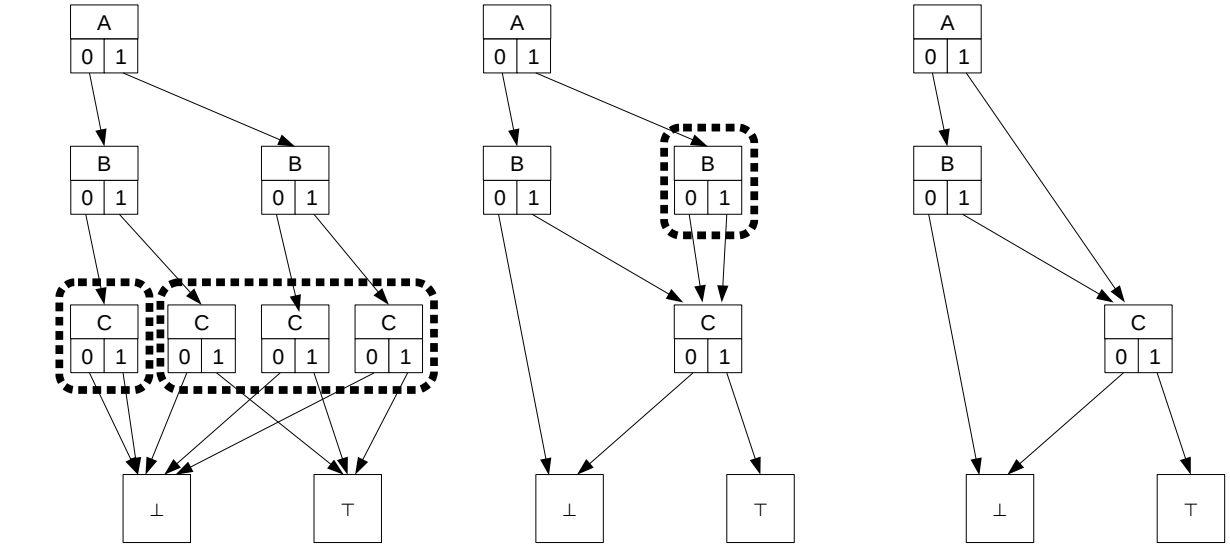

Figure 1: Reduction of a binary decision diagram

- simplification: finding a less complex expression that will yield the same value given some assumption about the variables.
- enumerating all sets of variable values that will satisfy a given expression.

In the implementation described in this paper, a set of BDD's is represented by a TclOO object. The methods that perform logical operations, quantification, simplification and enumeration are written in C for performance.

As an example, let us use TclBDD to construct the binary decision diagram for the expression, $(A \vee B) \wedge C$, query it to find the values of A and C that make the expression true for at least one value of B, and enumerate the result. The “assembly language computation for this is shown in Figure 2.

```
package require tclbdd

# Create the system
bdd::system create sys

# Name some variables
sys nthvar A 0
sys nthvar B 1
sys nthvar C 2

# Construct X=(A|B)&C
sys | temp A B; # temp = A | B
sys & X temp C; # X = temp & C

# For what values of A and C is the
# expression true for some B?
sys exists result {B} X

# Enumerate the result exhaustively
sys foreach_sat s result {
    bdd::foreach_fullsat res {0 2} $s {
      puts "A=[lindex $res 0]\
            C=[lindex $res 1]"
    }
}
```

Figure 2: Simple first-order logic

The program begins by constructing a BDD system. It names three variables *A*, *B*, and *C*, making them variable numbers 0, 1, and 2 respectively. It constructs the BDD for the given expression, and removes *B* from the BDD by applying the ∃ quantifier to it. Finally, it enumerates exhaustively all the values of *A* and *C* that make the resulting expression true. Unsurprisingly, it yields the result:

```
A=0 C=1
A=1 C=1
```

showing that the formula is satisfiable if and only if *C* is true.

For a larger case demonstrating the power of BDD's to represent complex expressions, the interested reader is referred to the test case `bdd-40.1` in the test suite for the `tclbdd` library. This test sets up the familiar “eight queens” problem: how many ways are there to place eight queens on a chessboard such that no queen attacks another. This problem is a classical example of problems that require backtracking search, but the tclbdd system solves it with ease and with no backtracking. It represents each square of the board with a Boolean variable, and computes the logical AND of all the restrictions that no two queens may be in the same row, column or diagonal. It runs in the blink of an eye, despite the fact that at some points in the calculation, it is representing over 10^{18} combinations of possibilities.

3) Finite domain decision diagrams: the “assembly language”

The next level of support for deductive reasoning in this project is the layer that implements Finite Domain Decision Diagrams (FDDD's). FDDD's are a structure introduced by Whaley and Lam in the 'bddbddb' project [WHAL05] specifically for program flow analysis. They represent tuples of values over totally ordered finite domains. In other words, every value in a tuple can be replaced with a small integer. Each value in a tuple is a member of a different named domain, and all tuples in a given set have values drawn from the same sequence of domains. (Domains in a set of tuples are analogous to columns in a table within a relational database.)

The FDDD representation derives from the BDD representation in a natural way. Each domain is associated with a set of Boolean variables in the BDD corresponding to the bits of the binary representation of its values. A domain with sixteen members, for instance, will require at least four BDD variables to represent it. A tuple is then an AND-term in the boolean expression. A BDD's Boolean expression will be true if and only if the corresponding tuple is present in the set.

The operations of relational algebra have a natural

mapping onto BDD's. The most important ones to consider are joins, replacements, negations, projections, set unions, and selection.

Joining turns out to be simply a Boolean AND operation. Replacement of one domain with another is a rewriting of the Boolean expression to replace the first domain's variables with those of the second domain, an operation that is provided by the BDD library. Negation (creating a table containing all possible rows absent from a given table) is a Boolean NOT operation. Projection (reducing a relation by removing a column) is applying existential quantification to the column's variables. The union of two sets is the logical OR of their Boolean expressions.

Selection is a special case of join. A set is constructed containing the values to be sought, and the newly constructed set is joined with the set being searched to yield the rows with the desired values.

There is also a special equality relation between any pair of domains. The relation `A==B` contains exactly those pairs of tuples that have the same values in domain `A` as in domain `B`. Joining this relation to any other relation has the effect of performing a self-join.

4) A digression: ordering of variables

It turns out that one critical feature in the design of a FDDD database is the ordering of the variables in a BDD. A poor ordering can easily result in a BDD that grows exponentially in size with the number of entities being represented, while a good ordering may be nicely linear.

Consider for instance, a table, `suc`, containing two four-bit fields `a` and `b`. A tuple `suc(a,b)` is present if and only if `a=b+1`.

If the BDD is constructed using the naïve variable order `(a2, a1, a0, b2, b1, b0)`, listing the bits of the values from most significant to least significant, then the size of the BDD is 44 nodes. (Its first three levels are a complete binary tree over the values of `a`.) If instead, the variables are interleaved, `(a2, b2, a1, b1, a0, b0)`, the size drops to 17 nodes, and there are no more than three nodes at each level. The difference is even more striking as the size of the domains grows: with eight-bit columns, there are 764 nodes in the concatenated representation but only 37 in the interleaved.

Alas, the problem of determining the optimum variable ordering is NP-hard [BOLL96]. Fortunately, there are a number of good heuristics. So far, in the experiments that the author has conducted with this project, a good ordering was obtainable simply by interleaving the bits of columns that were thought to be “closely related” and occasionally reversing their endian-ness.

The database definition for the FDDD allows specifying the domains using three commands, `domain`, `interleave` and `concatenate`. The `domain` command defines a single domain, accepting its size in bits and endian-ness. The `interleave` and `concatenate` commands each accept any number of partial database definition (the result of `domain`, `interleave`, or `concatenate`), and produce a new database definition by taking their arguments and interleaving or concatenating them in the bit ordering. Thus, the definition of the database containing the (properly interleaved) `suc` relation will look like:

```
database create db \
    [interleave \
        [domain a 8 bigendian] \
        [domain b 8 bigendian]]
db relation suc a b
```

5) The FDDD assembly language

Given this infrastructure, what the FDDD package provides is a set of methods that compile BDD code to manipulate the database. Because this is still “assembly language” level, the code is still verbose and somewhat unreadable. Nevertheless, it would be good to walk through one complete example to show the sorts of operations that are available at the FDDD level.

A) Creating a database

We will use as an example a database containing information about the ancestry of a certain well-known family. There will be three columns, `p1`, `p2`, and `p3`, each four bits wide, and all interleaved. There will be one predefined relation, `parentOf(p1,p2)` which contains a tuple (a,b) if and only if *a* is a parent of *b*. Figure 3 shows how the database is defined.

```
bdd::fddd::database create db \
    [bdd::fddd::interleave \
       [bdd::fddd::domain p1 4] \
       [bdd::fddd::domain p2 4] \
       [bdd::fddd::domain p3 4]]
db relation parentOf p1 p2
```

Figure 3: Creating a database for ancestry

Since the database can accept only small integers as column values, we create a mapping between personal names and small integers: `$p($name)` gives the integer for `$name`, and `[lindex $people $i]` gives the name of person `$i`. The simple code in Figure 4 is the usual design pattern for setting up a finite domain.

```
set i 0
set people {
    Andrew Anne Beatrice Charles Edward
    Elizabeth Eugenie George Harry
    James Louise William
}
foreach x $people {
    set p($x) $i
    incr i
}
```

Figure 4: Naming objects in a finite domain

Next, we load the `parentOf` relation. The database objject provides a `[loader]` method that emits a Tcl command that will import a row. (Most of the FDDD methods work by emitting Tcl code, rather than by performing a requested action directly.) Figure 5 gives the code.

```
interp alias {} parentOf {} \
    {*}[db loader parentOf]
parentOf $p(Elizabeth) $p(Charles)
parentOf $p(Elizabeth) $p(Anne)
parentOf $p(Elizabeth) $p(Andrew)
parentOf $p(Elizabeth) $p(Edward)
parentOf $p(Charles) $p(Harry)
parentOf $p(Charles) $p(William)
parentOf $p(Andrew) $p(Beatrice)
parentOf $p(Andrew) $p(Eugenie)
parentOf $p(Edward) $p(Louise)
parentOf $p(Edward) $p(James)
parentOf $p(William) $p(George)
```

Figure 5: Loading a relation

Now we want to create a `grandparentOf` relation that contains a tuple (a,b) if *a* is a grandparent of *b*: that is, if *a* is a parent of some value *c*, and *c* is a parent of *b*.

Expressing this in terms of the lowest level relational primitives is a bit awkward. The most effective way to approach it appears to be:

1. Create a new relation `t1`, that will contain a tuple (a,c) if *a* is a parent of c.
2. Create a new relation, `t2`, that will contain a tuple (c,b) if *c* is a parent of *b*.
3. Join the two relations. The result, `t3`, will have a tuple (a,c,b) for every combination of (a,c) from `t1` and (c,b) from `t2`.
4. Project away the common column *c*, leaving a relation `grandparentOf`, containing a tuple (a,b) if *a* is a grandparent of *b*.

Figure 6 shows the code that performs these four steps. Note that the `replace`, `join`, and `project` methods all return bursts of code that in turn perform the requested actions. For this reason, they are substituted into a script that is then evaluated.

```
db relation t1 p1 p3
db relation t2 p3 p2
db relation t3 p1 p2 p3
db relation grandparentOf p1 p2
eval [subst {
    [db replace t1 parentOf p3 p2]
    [db replace t2 parentOf p3 p1]
    [db join t3 t1 t2]
    [db project grandparentOf t3]
}]
```

Figure 6: Creating a 'grandparentOf' relation

We now have enough information in the database to answer the question, “who are the grandchildren of Elizabeth?” To pose the question we do the following:

1. Create a relation `t4`, consisting of the single value **Elizabeth** in the column `p1`.
2. Join that relation to the grandparentOf relation, yielding the desired result.
3. Enumerate the values in the result.

```
# Create a singleton relation to hold 'Elizabeth'
db relation t4 p1
interp alias {} x {} {*}[db loader t4]
x $p(Elizabeth)

# Create a relation holding Elizabeth's grandchildren
db relation result p1 p2
eval [subst {
    [db join result grandparentOf t4]
}]

# Enumerate Elizabeth's grandchildren
db enumerate row result {
    puts [lindex $people \
              [dict get $row p2]]
}
```

Figure 7: Who are Elizabeth's grandchildren?

The program that carries out these steps is shown in Figure 7. When run, it produces the result,

```
Harry
Beatrice
Louise
Eugenie
James
William
```

6) Datalog: a high level language for deductive reasoning

Now that we have a relational database in hand, we need a way to manipulate and query it. As we have seen, the FDDD library provides low-level manipulators (join, project, union, and so on), but it is lacking in both power and user-friendliness. We need something better for the purpose of program analysis. It is tempting to say that for a relational database, there is only one language that makes sense: SQL. Nevertheless, for the application of program flow analysis, SQL would be a horrible choice. The issue is that most questions to be answered with flow analysis are fundamentally graph-theoretic. Their answers, generally speaking, depend on transitive closures, or recursive queries.

The classic example that SQL has trouble with is the relation, "*a* is an ancestor of *b*." In the example from Section 5, we can do "a is a grandparent of b" fairly easily, as shown in Figure 8.

```
SELECT x.parent AS grandparent,
       y.child AS grandchild
FROM parentOf x
LEFT JOIN parentOf y
ON y.parent = x.child
```

Figure 8: 'Grandparent' relation in SQL

The answer to 'who are Elizabeth's descendants?' is less straightforward. In standard SQL-99 (which is widely ignored by database vendors), a query like Figure 9 could do the job.

The syntax is awkward, and the standard SQL version is not widely available (although various databases implement their own, equally awkward, versions of recursive query).

Instead, the project embeds an implementation of the Datalog database manipulation language. [CERI89] Datalog is a subset of Prolog, intended to support efficient manipulation of relational data structures. A Datalog program comprises some set of facts, rules, and queries. A

```
WITH RECURSIVE temp(anc, desc) AS (
  SELECT parent, child FROM parentOf WHERE parent = 'Elizabeth'
  UNION
  SELECT anc, child FROM temp JOIN parentOf ON parent = desc
) SELECT desc FROM temp
```

Figure 9: Recursive query in SQL

```
db relation ancestorOf p1 p2
proc descendantsOf {ancestor} [bdd::datalog::compileProgram db {
    variable p
    variable people
    set anc $p($ancestor)
    set result {}
} {
    ancestorOf(p1, p2) :- parentOf(p1, p2).
    ancestorOf(p1, p2) :- ancestorOf(p1, p3), parentOf(p3, p2).
    ancestorOf($anc, p2)?
} d {
    lappend result [lindex $people [dict get $d p2]]
} {
    return $result
}]
```

Figure 10: Datalog program, embedded in Tcl, to solve the 'ancestorOf' relation

fact is simply an assertion that something is true about a specific relation:

```
parent($a, $b).
```

A rule gives a way to deduce new facts from what is known. Ancestry can be specified in two short rules:

```
ancestorOf(p1, p2) :- parentOf(p1, p2).
ancestorOf(p1, p2) :-
  ancestorOf(p1, p3), parentOf(p3, p2).
```

And a query simply reports information to a calling program:

```
ancestorOf($anc, p2)?
```

Recursion is implicit: any rule that depends, directly or indirectly on itself, is iterated to a fixpoint.

The Datalog compiler, of course, has to include a little bit of glue to interface the Datalog and Tcl languages. It's fairly simple, and designed for writing Tcl procedures. Each Datalog program can refer to the values in Tcl variables, and each Datalog program is augmented with an initialization block (a Tcl script executed before the Datalog program runs), a Tcl variable that will be used to hold a row of a result (expressed as a dict), a Tcl script that is executed once per query result, and a finalization block (a Tcl script execute once after the Datalog program terminates). A sample Tcl script wrapping the above three lines of Datalog looks like Figure 10.

Given the procedure in Figure 10, the Tcl command:

```
puts [descendantsOf Elizabeth]
```

lists all of Elizabeth's children, grandchildren and great-grandchildren:

```
Andrew Harry Beatrice Louise Edward
Eugenie Anne James Charles William
George
```

7) Another digression: handling negation

Datalog, as originally envisioned, had no negated terms: there was no way to say “A is true if B is false.” The lack of negation stemmed from two things: first, a relational database typically has no way of dealing with the combinatorial explosion of enumerating nonexistent rows, and second, allowing uncontrolled negation would lead to problems without a fixpoint:

```
A(x) :- ~A(x).
```

or problems without a unique fixpoint:

```
A(x) :- ~B(x). B(x) :- ~A(x).
```

Nevertheless, negation is needed for a great many tasks. For instance, let us consider the question, “who is an only child?” The predicate, “a is an only child” is most easily formulated as “a has a parent, but has no siblings,” as shown in Figure 11.

The first problem, that of the combinatorial explosion, is not an issue for BDD's. The BDD of a relation's complement is exactly the same size as that of the relation itself. The second problem, the possibility of constructing a system without a fixpoint, needs to be solved with some rigor for what negation means. The current implementation provides *stratified* negation semantics, which is fairly mainstream for Datalog implementations. In stratified negation, each rule is assigned a stratum number. A rule

```
db relation siblingOf p1 p2
db relation hasSibling p1
db relation onlyChild p1
proc onlyChildren {} [bdd::datalog::compileProgram db {
    variable p
    variable people
    set result {}
} {
    siblingOf(p1,p2) :- parentOf(p3, p1), parentOf(p3, p2), p1 != p2.
    hasSibling(p1) :- siblingOf(p1,_).
    onlyChild(p1) :- parentOf(_,p1), !hasSibling(p1).
    onlyChild(p1)?
} d {
    lappend result [lindex $people [dict get $d p1]]
} {
    return $result
}]
puts [onlyChildren]
```

Figure 11: Who is an only child?

that depends only on facts has a stratum of zero. A rule X that depends on another rule Y must have a stratum number that is at least Y's stratum number. If it depends on the negation of rule Y, its stratum must be strictly greater than Y's stratum number. This scheme avoids dependency cycles involving negation.

In the example from Figure 11, stratification is straightforward. The `parentOf` relation, being a ground term, is at stratum 0. The `siblingOf` relation, being dependent only on ground terms, is at stratum 1. The `hasSibling` relation depends only on non-negated stratum-1 terms and is also at stratum 1. The `onlyChild` relation depends on a negated stratum-1 term and is relegated to stratum 2. The content of the relations is computed in order by stratum. Negated terms are computed using the “closed world hypothesis” in which a term not known to be true is assumed to be false. The resulting model is guaranteed to be logically consistent.

Stratified negation may prove not to be sufficient, and the author suspects that the package will need to provide an option for *well-founded* negation, [GELD91] which allows for recursion through negation, as long as the result yields a model in which if any literals are true, their complements are false, and vice versa. (There may be literals whose value is undetermined by the program.)

8) Where is this going?

The current status of the project is that the BDD and FDDD libraries and the Datalog compiler are all available in reasonably complete form, with test suites and manual pages, from the author's Fossil repository at https://chiselapp.com/user/kbk/repository/tclbdd/. The remainder of this paper is considerably more speculative, and reports on the results of early “proof of concept” experiments with Datalog and the analysis of Tcl programs.

In most cases, aggressive optimization will depend on inferring the data types of values in Tcl programs. Without some sort of type inference, Tcl's processing of code includes large amounts of run-time type identification, type coercion (“shimmering”), and packaging of values into Tcl_Obj structures. All of this can be avoided, for example, in numeric-intensive code if we can prove facts like “A is an integer at point B in the code.”

As a simple example, the first experiment is the `cos` procedure shown in Figure 12, which computes the cosine of a number using a Maclaurin series approximation. This example has the property that in an ideal world, the types of all objects would be identified perfectly, and the procedure could be compiled entirely down to machine code.

The first part of the experiment is to retrieve the bytecode that Tcl's compiler generates for the `cos` procedure, and convert it to a form more amenable to analysis: in this instance, three-address code. The conversion logic onsists mostly of a `[switch]` command and bookkeeping to keep track of the depth of the Tcl execution stack, and is

```
proc cos {x {n 16}} {
  set j 0
  set s 1.0
  set t 1.0
  set i 0
  while {[incr i] < $n} {
      set t [expr {-$t*$x*$x / [incr j] / [incr j]}]
      set s [expr {$s + $t}]
  }
  return $s
}
```

Figure 12: Numeric-intensive procedure - optimization example

not shown here. The initial converted program, prior to any optimization, appears as xxx. In it, `{var X}` denotes a named variable; `{temp N}` denotes a value on the stack, and `{literal V}` denotes a constant value. The conversion is sloppy and straightforward. For instance, `[set j 0]` was translated by the bytecode compiler to "push `0` to the top of the stack; pop the stack and put the popped value in variable `j`," and this translation came forward literally into the three-address code.

There are then some ground facts that are asserted about the operations:

- `reads(pc,v)` – The instruction at `pc` reads the value of `v`.
- `writes(pc,v)` – The instruction at `pc` writes the value of `v`.
- `isCopy(pc)` – The instruction at `pc` is a copy.
- `noSideEffect(pc)` – The instruction at `pc` is free of unknown side effects (such as aliasing variables, establishing traces, evaluating scripts).
- `seq(pc1,pc2)` – The instruction at `pc` is immediately followed by `pc2` on at least one execution path.

These facts will be used for the analyses that follow.

There are a few optimizations that can be done early, in order to reduce the sheer volume of code that sophisticated analyses must process. The first of these is copy propagation – the removal of useless data motion. For example, the sequence:

```
copy {temp 0} {literal 0}
copy {var j} {temp 0}
```

can (provided that `{temp 0}` is not used elsewhere) be replaced by:

```
copy {var j} {literal 0}
```

Performing copy propagation is ordinarily a fairly major task in an optimizer, requiring sophisticated data structures and voluminous code. In Datalog, it's mostly a matter of identifying that a statement `st` reads a given value `v`, that value `v` is a copy of value `v2`, and that every definition of either `v` or `v2` that reaches `st` goes through either a copy `v:=v2` or `v2:=v`. The Datalog code is more complex than anything we've seen yet, but is still less than a page of code. It appears in Figure 14 on page 10. The Tcl action for the code is trivial: rewrite the statement at `st`, replacing `v` with `v2`.

Copy propagation has done absolutely nothing to reduce code size, but once it's been done, there will be (we hope) a fair number of statements that are dead – they do nothing but write values that are never read. Dead code elimination is the next step in the process: find those points! It begins with the analysis of reaching definitions: "the assignment of a value `v` at location `st` is potentially read at location `st2`". This one can be done in three Datalog statements (Figure 13).

```
flowsTo(_, st, st2) :- seq(st, st2).
flowsTo(v, st3, st2) :-
    flowsTo(v, st3, st),
    !writes(st, v),
    flowsTo(v, st, st2).
reaches(v, st, st2) :-
    writes(st, v),
    flowsTo(v, st, st2),
    reads(st2,v).
```

Figure 13: Reaching definitions

With reaching definitions available, dead code analysis is also straightforward. A statement is live if it has

```
% Determine for each pair of variables what statements copy
% between variables of the pair, in either direction.

isCopyBetween(st,v,v2) :- isCopy(st), reads(st, v2), writes(st, v).
isCopyBetween(st,v,v2) :- isCopy(st), reads(st, v), writes(st, v2).

% copyTransparent(st, st2, v, v2) means 'there is a path from st to
% st2 on which no code writes to either v or v2'.

copyTransparent(st, st2, _, _) :- seq(st, st2).
copyTransparent(st, st2, v, v2) :- copyTransparent(st, st3, v, v2),
                                   !writes(st3, v), !writes(st3, v2),
                                   noSideEffect(st3),
                                   copyTransparent(st3, st2, v, v2).

% writesOneOf(st, v, v2) means 'st writes either v or v2'

writesOneOf(st, v, _) :- writes(st, v).
writesOneOf(st, _, v) :- writes(st, v).

% nonCopyReaches(v, v2, st2) means 'on at least one code path,
% an assignment to either v or v2 that is not a copy between them
% reaches the statement st2'

nonCopyReaches(v, v2, st2) :- writesOneOf(st, v, v2),
                              !isCopyBetween(st, v, v2),
                              copyTransparent(st, st2, v, v2).

% copyReaches(v, v2, st2) means 'on at least one code path, a
% copy v := v2 reaches statement st2 without any intervening code
% changing v or v2'

copyReaches(v, v2, st2) :- isCopy(st), reads(st, v2), writes(st, v),
                           copyTransparent(st, st2, v, v2).

% A statement is a candidate for copy propagation if it reads a
% variable v, variable v obtains its value by copying variable v2 on
% at least one code path, and every reaching definition of variable
% v and variable v2 goes through either v := v2 or v2 := v before
% reaching the statement.

copyPropagatable(st, v, v2) :- reads(st, v),
                               copyReaches(v, v2, st),
                               !nonCopyReaches(v, v2, st).

copyPropagatable(st, v, v2)?
```

Figure 14: Copy propagation in Datalog

uncontrolled side effects, if it writes a value that is read by a live statement, or if it does something other than generate a value. All statements that assign to an unused value are dead and can be removed from the instruction sequence.

The effect of copy propagation and dead code elimination, together with the related but simpler operation of reverse copy propagation, in which a copy of an instruction's result, rather than of its operand, is removed, is to decrease code volume by a factor of two. (This decrease consists almost entirely of eliminating temporary variables introduced by the brutally simple translation of stack-oriented code.) Figure 15 on page 11 shows the transformation that takes place for the `cos` procedure.

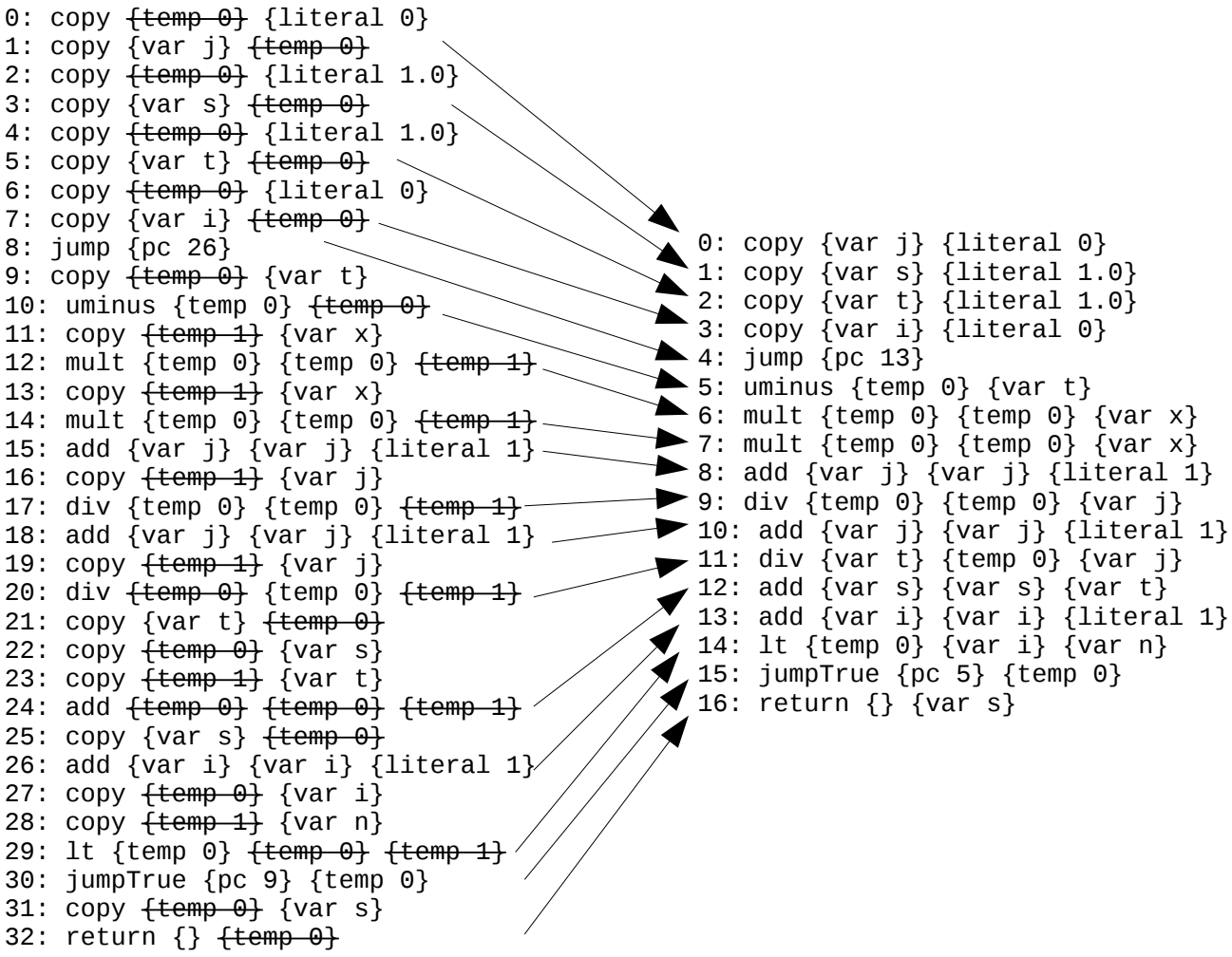

Figure 15: Code improved by copy propagation

The next significant transformation is to reduce the program to Static Single Assignment (SSA) form[CYTR91]. In this form, all assignments of a value have distinct names. If more than one assignment of a value reaches a given point in the code, the multiple reaching assignments are replaced by a pseudo-function φ whose arguments enumerate the reaching definitons. This form allows reading out directly all the reaching definitions of a given value, or all the places that a given value reaches, and is critical to type analysis. The actual requirements for the placement of φ-functions are somewhat complicated (the reader is referred to [CYTR91] for the details), but one fairly short Datalog program serves to identify the points at which φ-functions need to be inserted and another serves to replace references to values with references to the appropriate φ results. The resulting transformed program can be seen in Figure 16.

We are finally ready to perform type analysis on this program. The SSA form gives us more or less complete information about data flows, and now all analysis of types can be abstracted without detailed reference to the program's control flow.

```
0: copy {var j 0} {literal 0}
1: copy {var s 1} {literal 1.0}
2: copy {var t 2} {literal 1.0}
3: copy {var i 3} {literal 0}
4: jump {pc 13}
5: uminus {temp 0 5} {var t phi 13}
6: mult {temp 0 6} {temp 0 5} {var x input}
7: mult {temp 0 7} {temp 0 6} {var x input}
8: add {var j 8} {var j phi 13} {literal 1}
9: div {temp 0 9} {temp 0 7} {var j 8}
10: add {var j 10} {var j 8} {literal 1}
11: div {var t 11} {temp 0 9} {var j 10}
12: add {var s 12} {var s phi 13} {var t 11}
13: phi {var j phi 13} {var j 0} {var j 10}
14: phi {var s phi 13} {var s 1} {var s 12}
15: phi {var t phi 13} {var t 2} {var t 11}
16: phi {var i phi 13} {var i 3} {var i 13}
17: add {var i 13} {var i phi 13} {literal 1}
18: lt {temp 0 14} {var i 13} {var n input}
19: jumpTrue {pc 5} {temp 0 14}
20: return {} {var s phi 13}
```

Figure 16: [cos] procedure in SSA form

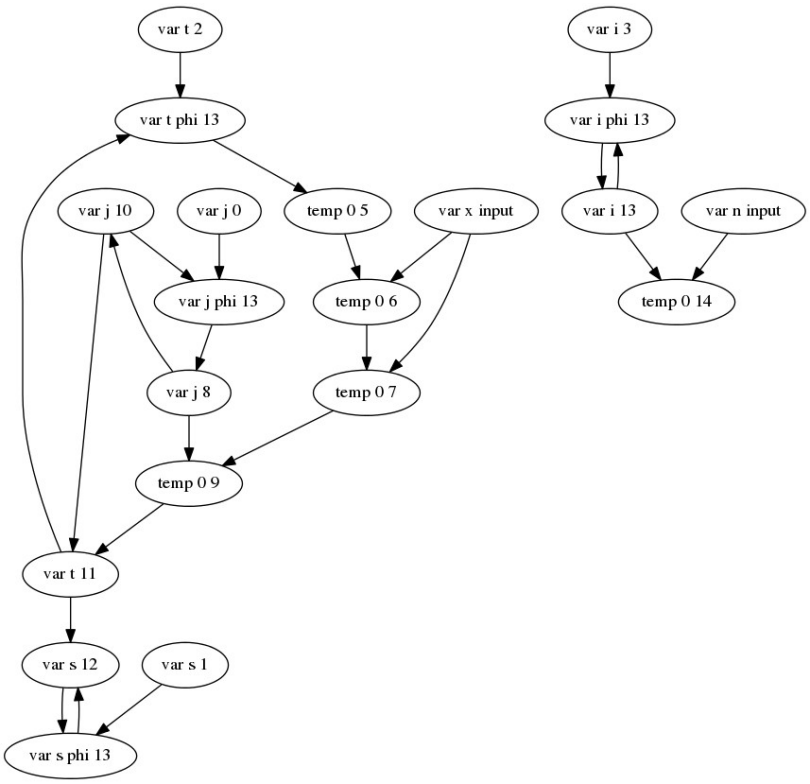

Figure 17: Variable dependencies abstracted from control flow

Figure 17 shows the dependency graph for the values in the `[cos]` procedure.

The current procedure for type analysis has not yet been formulated in Datalog, because initial experiments regarding type inference were conducted before the Datalog compiler was available. Instead, it works by analyzing strongly connected components of the graph of dependencies among values.

What is implemented so far is a simple type algebra incorporating the types, `int`, `entier`, `double`, `boolean` and `string` (together with a `numeric` type that represents the union of `entier` and `double`, and an `int&boolean` type that represents the intersection of `int` and `boolean`, the values `0` and `1`. Error: Reference source not found shows the hierarchy.

Working through the variables in dependency order gives the following set of conclusions:

1. `{var t 2}` is a double.
2. `{var x input}` is of unknown type (without further examination of the calling context).
3. `{var j 0}` is an integer.
4. `{var j 8}`, `{var j 10}` and `{var j phi 13}` form a dependency loop. Loops are handled by assuming the most restrictive data type possible for each value and then relaxing the type constraints that lead to inconsistencies. This iteration deduces that all three of these values are also integers.
5. A similar analysis deduces that `{temp 0 5}`, `{temp 0 6}`, `{temp 0 7}`, `{temp 0 8}`, `{var t 11}`, and `{var t phi 13}` , which form a dependency loop, are all doubles.
6. `{var s 1}` is a double.
7. The loop of variables `{var s 12}` and `{var s phi 13}` are also doubles.
8. `{var i 3}` is both an integer and a boolean.
9. The loop of variables `{var i 13}` and `{var i phi 13}` are integers.
10. `{var n input}` is of unknown type (without examination of the calling context).
11. `{temp 0 14}` is a boolean.

This analysis actually constitutes a fairly complete type extraction for the given procedure. About the only way to

improve it would be to add retrograde analysis. We could detect, for instance, that {var x input} flows only into a context that will yield an error if it is not a number, and generate specialized code to convert it only once. Similarly, we could detect that {var n input} is always, only, compared with a number, and generate specialized code assiming that it, too, is numeric.

9) Conclusions

The Datalog compiler, and the underlying inference engine, presented in this paper are in relatively polished form. They should be of value to systems that wish to analyze large data sets over finite domains, and can support complex recursive queries over those data sets. Needless to say, no software is ever entirely “complete” in terms of its feature set, and this library is no exception. Specific areas that could be improved include optimization of relational queries (the system as implemented does not make use, for instance, of combinations of Boolean formulas with quantifiers), the support of imperative languages for BDD and FDDD manipulation, and the extension of Datalog to well-founded semantics and retractions.

The application of the library to Tcl code analysis, by comparison, is in its infancy. The example put forth in Section 8 merely scratches the surface as a “proof of concept” and indicates what may be possible in terms of identifying the types of values.

If this type identification can be done competently, there is reason to hope that a useful subset of Tcl code can be reduced to machine code by a Just-In-Time compiler. Such a reduction could yield a tremendous (perhaps as many as 30- or 40-fold) performance gain in numeric- or list-intensive Tcl code, and would eliminate one major reason for escape into C.

References

[ANDE97] Andersen, Henrik Reif. ", An Introduction to Binary Decision Diagrams." Lecture notes, IT University of Copenhagen, 1997, http://aima.eecs.berkeley.edu/~russell/classes/cs289/f04/readings/Andersen:1997.pdf

[BOLL96] Bollig, Beate, and Ingo Wegener. " Improving the Variable Ordering of OBDDs Is NP-Complete." IEEE Transactions on Computers 45:9 (1996) 993-1002. http://dx.doi.org/10.1109/12.537122

[BRYA86] Bryant, R.E.. " Graph-Based Algorithms for Boolean Function Manipulation." IEEE Transactions on Computers BC-35:8 (1986) 677-691. http://dx.doi.org/10.1109/TC.1986.1676819

[CERI89] Ceri, S., C. Gottlob, L. Tanca. " What You Always Wanted to Know about Datalog (and Never Dared to Ask)." IEEE Transactions on Knowledge and Data Engineering 1:1 (1989) 146-166.

[CYTR91] Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N.; and Zadeck, F. Kenneth. " Efficiently computing static single assignment form and the control dependence graph." ACM Trans. on Prog. Lang. and Sys. 13: (1991) 451-490.

[GELD91] Van Gelder, Allen, Kenneth A. Ross and John S. Schlipf. " The Well-Founded Semantics of General Logic Programs." Journal of the ACM 38:3 (1991) 620-650. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.6788&rep=rep1&type=pdf

[WHAL05] Whaley, John, Dzintars Avots , Michael Carbin , Monica S. Lam . " Using Datalog with binary decision diagrams for program analysis." Proc. 3rd Asian Symp. on Programming Languages and Systems (ASPLAS '05) : (2005) . http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.8258&rep=rep1&type=pdf

[WHAL06] Whaley, John, bddbddb, 2006, http://bddbddb.sourceforge.net/