A Flexible Solver for Finite Arithmetic Circuits

Nathaniel Wesley Filardo and Jason Eisner

Department of Computer Science
Johns Hopkins University
3400 N. Charles St., Baltimore, MD 21218, USA
`http://cs.jhu.edu/~{nwf,jason}/`
{nwf, jason}@cs.jhu.edu

Abstract

Arithmetic circuits arise in the context of weighted logic programming languages, such as Datalog with aggregation, or Dyna. A weighted logic program defines a generalized arithmetic circuit—the weighted version of a proof forest, with nodes having arbitrary rather than boolean values. In this paper, we focus on finite circuits. We present a flexible algorithm for efficiently *querying* node values as they change under *updates* to the circuit's inputs. Unlike traditional algorithms, ours is agnostic about which nodes are tabled (materialized), and can vary smoothly between the traditional strategies of forward and backward chaining. Our algorithm is designed to admit future generalizations, including cyclic and infinite circuits and propagation of delta updates.

1998 ACM Subject Classification F.1.1 Models of Computation, I.2.3 Deduction and Theorem Proving

Keywords and phrases arithmetic circuits, memoization, view maintenance, logic programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.425

1 Introduction

The weighted logic programming language Dyna [10] is a convenient and modular notation for specifying derived data. In this paper, we begin to consider efficient algorithms for answering queries against Dyna programs. Our methods treat arithmetic circuits, and are relevant to other variants of logic programming, such as Datalog with aggregation [15, 5].

Many tasks in computer science involve computing and maintaining derived data. *Deductive databases* [21] store **extensional** (i.e., provided) data but also define additional **intensional** data specified by formulas. Algorithms in artificial intelligence or business analytics can often be written in this form [10]. The extensional data are observed facts, and the resulting cascades of intensional data arise from aggregation, record linkage, analysis, logical reasoning, statistical inference, or machine learning.

If the extensional data can change over time, keeping the intensional data up to date is called *view maintenance* or *stream processing* [23]. This pattern includes traditional *abstract data types*, which maintain derived data under operations such as "insert" and "remove." For example, a priority queue maintains the argmax of a function over an extensional set.

A Dyna program is a *declarative* specification of derived data. Like an abstract data type, it admits many correct implementations of its update and query methods. These execution strategies range from the laziest ("store the update stream and scan it when queried") to the most eager ("recompute all intensional data upon every update"). A particular strategy might trade time for space, or more time now for less time later (e.g., investing time in finding a faster query plan or maintaining an index). We seek a unified algorithm that subsumes as many reasonable strategies as possible, and which supports transitioning smoothly between

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 425–438
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
COMPUTE(j ∈ I_int)
  return f_j ({i ↦ LOOKUP(i) | i ∈ P_j})

LOOKUP(j ∈ I)
  v ← M[j]
  if v = UNK then v ← COMPUTE(j)
  maybe M[j] ← v
  return v
```

Listing 1 Internals of basic backward chaining with optional memoization. $\mathcal{M}$ stores values for extensional items and initially stores UNK for intensional items.

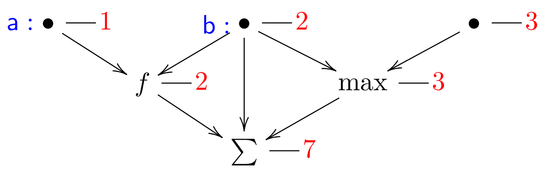

Figure 1 An example arithmetic circuit on the natural numbers, showing the function for each intensional item (f, max, $\sum$, where $f(a, b) = b^a$) and the symbol • for each extensional item. Item values are shown in red, and selected item names in blue.

them. This allows different strategies to be selected for different parts of the program (static analysis) or for different workloads (dynamic analysis).

In the present work, we discuss the development of a generic algorithm for finding and maintaining solutions to finite *arithmetic circuits*, which are a subset of Datalog and Dyna programs. Our algorithm offers several degrees of freedom, which will allow us to compare various static and adaptive strategies in future. The algorithm can choose any initial guess for the circuit's solution; its agenda of pending computations may be processed in any order; and it contains `maybe` directives where the algorithm has an additional free choice. Thus, our algorithm smoothly interpolates among traditional strategies such as forward chaining, backward chaining, and backward chaining with memoization.

2 Arithmetic Circuits

An **arithmetic circuit** [4] is a finite directed acyclic graph on nodes $\mathcal{I}$ with edges $\mathcal{E}$. We refer to the nodes as **items**. We denote item j's set of **parents** by $P_j \stackrel{\text{def}}{=} \{i \mid (i \to j) \in \mathcal{E}\}$, and its set of **children** by $C_j \stackrel{\text{def}}{=} \{k \mid (j \to k) \in \mathcal{E}\}$. Transitive parents are called **ancestors**, and transitive children are called **descendants**. We use the term **generalized arithmetic circuit** for the more general case where the graph may be infinite and/or cyclic.

Each item $i \in \mathcal{I}$ has a **value** in some set $\mathcal{V}$. Figure 1 shows a small arithmetic circuit over integer values. The **root** or **input** items, those without parents, are denoted $\mathcal{I}_{\text{ext}}$ ("extensional") and receive their values from the environment.[1] The remaining items are denoted $\mathcal{I}_{\text{int}}$ ("intensional") and derive their values by rule from their parents' values. Each item $j \in \mathcal{I}_{\text{int}}$ is equipped with a function f_j to combine its parents' values. The input to f_j is not merely an unordered collection of the parents' values. Rather, to specify which parent has which value, it is a *map* $P_j \to \mathcal{V}$, consisting of a collection of pairs $i \mapsto v$.

A Datalog or pure Prolog program can be regarded as a concise specification of a **generalized boolean circuit**, which is the case where $\mathcal{V} = \{\text{TRUE}, \text{FALSE}\}$. The items $\mathcal{I}$ correspond to propositional terms of the logic program, and clauses of the logic program describe how to discover the parents or children of a given item (on demand). Specifically, each grounding of a clause corresponds to an AND node whose parents are the body items, and whose child is an OR node corresponding to the head item. This kind of circuit is called an AND/OR graph.

[1] The literature varies as to whether extensional items are called roots or leaves, whether they are regarded as ancestors or descendants, and whether they are drawn at the top or the bottom of a figure. We treat them as roots and ancestors and draw them at the top. So edges and information flow downward in our drawings. As a result, "bottom-up" reasoning (forward chaining) actually proceeds from the top of the drawing down.

Datalog is sometimes extended to allow limited use of NOT nodes as well.

Arithmetic circuits are the natural analogue of boolean circuits for *weighted* logic programming languages, such as Datalog with aggregation [15, 5] and our own Dyna [11, 9].

Suppose we are given a generalized arithmetic circuit, along with a map $\mathcal{S} : \mathcal{I}_{\text{ext}} \to \mathcal{V}$ that specifies the extensional data. A **solution** to the circuit is an extension $\overline{\mathcal{S}}$ of this map over all of $\mathcal{I}$ such that $\overline{\mathcal{S}}[j] = f_j\left(\left\{i \mapsto \overline{\mathcal{S}}[i]\right\} \mid i \in P_j\right)$ for each $j \in \mathcal{I}_{\text{int}}$. In the traditional case where the circuit is finite and acyclic a solution $\overline{\mathcal{S}}$ always exists and is unique. This paper considers only that case, which also ensures that our algorithms always terminate. However, we will avoid using methods that rely strongly on finiteness or acyclicity. This makes our methods relevant to the harder problem of solving *generalized* arithmetic circuits (see section 7), as needed for the general case of weighted logic programming languages.

3 Backward Chaining

We begin with some basic strategies for querying an item's solution value $\overline{\mathcal{S}}[j]$, based on **backward chaining** from the item to its ancestors. We construct a map $\mathcal{M}$ from items to their solution values, known as the **memo table** or **chart**. For each extensional item $j \in \mathcal{I}_{\text{ext}}$, we initialize $\mathcal{M}[j]$ to $\mathcal{S}[j]$ $(= \overline{\mathcal{S}}[j])$. For each intensional items $j \in \mathcal{I}_{\text{int}}$, the solution value $\overline{\mathcal{S}}[j]$ is initially *unknown*, so we initialize $\mathcal{M}[j]$ to the special object UNK $\notin \mathcal{V}$. We may regard the map $\mathcal{M} : \mathcal{I} \to \mathcal{V} \cup \{\text{UNK}\}$ as a *partial* map $\mathcal{I} \to \mathcal{V}$ that stores actual values for only some items—initially just the extensional items.

We define mutually recursive functions LOOKUP and COMPUTE as in Listing 1. A user may query the solution with LOOKUP(j). This returns $\mathcal{M}[j]$ if it is known, but otherwise calls COMPUTE(j) to compute j's value using f_j, which in turn requires LOOKUPs at j's parents.

Pure Backward Chaining The simplest form of backward chaining simply recurses through ancestors until LOOKUP reaches the roots. Line 7 is never used in this case, so $\mathcal{M}$ never changes and intensional items remain as UNK. Clearly LOOKUP(j) returns $\overline{\mathcal{S}}[j]$.

Unfortunately, pure backward chaining can have runtime exponential in the size of the circuit. Each call to LOOKUP(j) will in effect enumerate all *paths* to j. For example, consider a circuit for computing Fibonacci numbers, where each item $\mathsf{fib}(n)$ for $n \geq 2$ is the sum of its parents $\mathsf{fib}(n-1)$ and $\mathsf{fib}(n-2)$. Then LOOKUP($\mathsf{fib}(n)$) has runtime that is exponential in n, with $\mathsf{fib}(n-t)$ being repeatedly computed $\mathsf{fib}(t)$ ($\approx O(1.618^t)$) times during the recursion.

Optional Memoization To avoid such repeated computation, a call to LOOKUP(j) can **memoize** its work by caching the result of COMPUTE(j) in $\mathcal{M}[j]$ for use by future calls, via line 7 of Listing 1. This is the backward-chaining version of dynamic programming. It generalizes the node-marking strategy that depth-first search uses to avoid re-exploring a subgraph. However, the **maybe** keyword in line 7 indicates that the memoization step is not required for correctness; it merely commits space in hopes of a future speedup. LOOKUP($\mathsf{fib}(n)$) can even achieve $O(n)$ expected runtime without memoizing all recursive LOOKUPs: instead it can memoize LOOKUPs on a systematic subset of items, or on a random subset of calls.

4 Reactive Circuits: Change Propagation

Our goal is to design a dynamic algorithm for arithmetic circuits that supports not just **queries** but also **updates**. It must handle a stream of operations of the form QUERY(j) for

```
RunAgenda()
  until 𝒜 = ∅
    pop i : ⇽v from 𝒜
    if v = UNK then v ← Compute(i)
    if v ≠ ℳ[i] then   % else discard
      ℳ[i] ← v
      foreach j ∈ C_i
        w ← UNK
        maybe w ← Compute(j)
        Update(j, w)
```

Listing 2 The core of an agenda-driven, tuple-at-a-time variant of the traditional forward chaining algorithm. $\mathcal{M}$ is initialized to an arbitrary but total guess and remains total (no UNK values) thereafter. Hence, though Compute calls Lookup, Lookup never recurses back to Compute.

```
Update(j ∈ ℐ, w ∈ 𝒱 ⊔ {UNK})
  delete 𝒜[j]
  if w ≠ ℳ[j] then   % else discard
    𝒜[j] ← ⇽w
```

Listing 3 Updates requested by the user or by RunAgenda are enqueued on the agenda $\mathcal{A}$ as replacement updates.

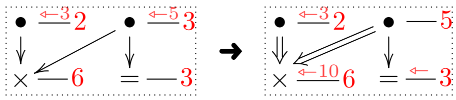

Figure 2 An example iteration of the loop in RunAgenda. We apply the update $\underline{\leftarrow 5}$ to the right parent, which makes the children inconsistent with their parents, and enqueue new updates that will fix the inconsistencies. Double arrows indicate the edges used to Compute the new value in the replacement update: 10 is 2×5.

any j, which returns $\overline{\mathcal{S}}[j]$, and Update$(i,v)$ for $i \in \mathcal{I}_{\text{ext}}$, which modifies $\mathcal{S}[i]$ to $v \in \mathcal{V}$.[2]

In the case of pure backward chaining, we only have to maintain the stored extensional data, as intensional values are not stored, but are derived from the extensional data on demand. In our terminology from above, Update(i,v) can just set $\mathcal{M}[i] \leftarrow v$, and Query$(j)$ can just call Lookup(j).

However, handling updates is harder once we allow memoization of intensional values. The memos in $\mathcal{M}$ grow **stale** as external inputs change, yet Lookup would continue to return outdated results based on these memos. That is, updating i may make its intensional descendants inconsistent; this must be rectified before subsequent queries are answered. We therefore need some mechanism for restoring consistency in $\mathcal{M}$, by propagating changes to memoized descendants.

Formally, we say that $j \in \mathcal{I}_{\text{ext}}$ is **consistent** iff Lookup$(j)= \mathcal{S}[j]$, and that $j \in \mathcal{I}_{\text{int}}$ is **consistent** iff Lookup(j) = Compute(j). Notice that un-memoized intensional items (those with $\mathcal{M}[j] = \text{UNK}$) are always consistent. We call $\mathcal{M}$ consistent if all items are consistent—in this case Lookup(j) will return the solution $\overline{\mathcal{S}}[j]$ as desired. Equivalently, the memo table $\mathcal{M}$ is consistent iff each extensional memo is correct and each intensional memo is in agreement with its visible ancestors. Here i and k are said to be **visible** to each other whenever there is a directed path from i to its descendant k that goes only through un-memoized (UNK) items. Thus, calling Compute(k) eventually recurses to Lookup(i) at each visible parent i.

5 Pure Forward Chaining

An alternative solution strategy, **forward chaining**, propagates updates. We will use it in section 6 to solve the update problem. First we present forward chaining in its pure form.

Pure forward chaining *eagerly* fills in the *entire* chart $\mathcal{M}$, starting at the roots and visiting children after their parents. Eventually $\mathcal{M}$ converges to $\overline{\mathcal{S}}$. Forward chaining algorithms include natural-order recalculation in spreadsheets [29] and semi-naive bottom-up evaluation for Datalog [28]. We use the "tuple-at-a-time" algorithm of Listing 2. It uses an **agenda** $\mathcal{A}$

---

[2] We also wish to support *continuous queries*, in which the user may request (asynchronous) notifications when specified items change value. This is, however, beyond the scope of the current paper.

that enqueues *future* **updates** to the chart [17, 11]. $\mathcal{A}$ contains at most one update for each item i, which we denote $\mathcal{A}[i]$, and supports modification or deletion of this update.[3]

Our updates are **replacement updates** of the form $i : \underline{\leftarrow v}$ (where $i \in \mathcal{I}$ and $v \in \mathcal{V}$).[4] Iteratively, until the agenda is empty, our forward chaining algorithm **pops** (selects and removes) any update $i : \underline{\leftarrow v}$ from the agenda, and **applies** it to the chart by setting $\mathcal{M}[i] \leftarrow v$. The algorithm then **propagates** this update to i's children, by **pushing** an appropriate update $j : \underline{\leftarrow w}$ onto the agenda for each child j. This push operation overwrites any previous update to j, so we write it as $\mathcal{A}[j] \leftarrow \underline{\leftarrow w}$.

The new value w is obtained by `Compute(j)`, meaning it is recomputed from the values at j's parents (including the changed value at i). If $\mathcal{M}[j]$ already had value w, the update is immediately discarded and does not propagate further. Ordinarily, w is `Compute`d in line 9 when the update is constructed and pushed. But if line 9 is optionally skipped, the update specifies w as UNK, meaning to compute the new value only when the update is popped and actually applied (line 4). Such a **refresh update** $j : \underline{\leftarrow \text{UNK}}$ may be abbreviated as $j : \underline{\leftarrow}$ and simply says to refresh j's value so it is consistent.[5] In any case, *an inconsistent item always has an update pending on the agenda*, which will eventually make it consistent.[6]

Figure 2 shows one step of pure forward chaining. In our visual notation for circuits, we draw the state of item i as $i : f_i \;\text{—}\; \mathcal{M}[i]$, where i (if present) names the item, f_i is the item's function (or $\bullet$ if $i \in \mathcal{I}_{\text{ext}}$), and $\mathcal{M}[i]$ is the current memo if any. If an update to i is waiting on the agenda, we display it over i's line as $i : f_i \overset{\leftarrow v}{\text{—}} \mathcal{M}[i]$, omitting the new value v if it is UNK. Since information flows downward in our drawings, being *above* i's line indicates that the update has yet to be applied to $\mathcal{M}[i]$. (In section 6.1 we will introduce a below-the-line notation.) Our textual update notation $i : \underline{\leftarrow v}$ is intended to resemble the drawing.

The process can be started from any total (UNK-free) initial chart $\mathcal{M}$, provided that the initial agenda $\mathcal{A}$ is sufficient to correct any inconsistencies in this $\mathcal{M}$. $\mathcal{A}$ is always sufficient if it updates *every* item: so the **conservative initialization strategy** defines each $\mathcal{A}[i]$ to be $i : \underline{\leftarrow \mathcal{S}[i]}$ for extensional i, and either $j : \underline{\leftarrow \texttt{Compute}(j)}$ or $j : \underline{\leftarrow}$ for intensional j. However, just as Listings 2–3 discard unnecessary updates, we can also omit as unnecessary any initial updates to items that are consistent in the initial $\mathcal{M}$. So we may wish to choose our initial $\mathcal{M}$ to be mostly consistent. For example, under the **NULL initialization strategy**, we initialize $\mathcal{M}[i]$ to a special value NULL $\in \mathcal{V}$ for all $i \in \mathcal{I}$. Provided that each function f_j outputs NULL whenever all its inputs are NULL, each intensional j is initially consistent and hence requires no initial update.[7]

The user method `Query(j)` is now defined as `RunAgenda(); return Lookup(j)`. This runs

[3] The agenda can be implemented as a simple dictionary. However, using an adaptable priority queue [14] can speed convergence, if one orders the updates topologically or by some informed heuristic [18, 12].

[4] It will be explained shortly why an underline appears in the notation for this type of update.

[5] Why are there two kinds of updates? Both have potential advantages. Refresh updates ensure that j is only recomputed once, even if the parents change repeatedly before the update pops. On the other hand, ordinary updates have the chance of being discarded immediately, which avoids the expense of pushing and popping any update at all; and if they are not discarded, their priority order can be affected by knowledge of w. Later algorithms in this paper cache item values temporarily, with the result that the cost of computing w may vary depending on when `Compute(j)` is called. Finally, delta updates (section 7) must be computed at push time.

[6] A consistent item might also have an update pending—a refresh that is not yet known to be unnecessary.

[7] In a logic programming setting, updating $\mathcal{M}[j]$ from NULL to non-NULL may be regarded as "proving j." Forward chaining proves the extensional items from the initial agenda, and then propagation causes it to prove some or all of the intensional items. In *unweighted* logic programming, NULL may be interpreted as "not proven" and identified with `FALSE`. Both `AND` and `OR` functions then have the necessary property. Similarly, in *weighted* logic programming, NULL means "no proven value." Here Dyna [9] again uses functions that guarantee the necessary property, by extending arithmetic functions (which generalize `AND`) as well as aggregation functions (which generalize `OR`) over the domain that includes NULL.

the agenda to completion and then returns $\mathcal{M}[j]$. As for the user method $\text{UPDATE}(i, v)$, the user is permitted to call Listing 3 in this case, thereby pushing a new update onto the agenda. Forward chaining processes all such updates at the start of the next query. This does not require recomputing the whole circuit.

It may be instructive at this point to contemplate the physical storage of the map $\mathcal{M} : \mathcal{I} \to \mathcal{V} \cup \{\text{UNK}\}$ (where $\text{NULL} \in \mathcal{V}$). A large circuit may be compactly represented by a much smaller logic program (section 2). In this case one might also hope to store $\mathcal{M}$ compactly in space $o(|\mathcal{I}|)$, using a sparse data structure such as a hash table. The "natural" storage strategy is to treat UNK as the default value in the case of backward chaining, but to treat NULL as the default value in the case of forward chaining. In each case this means that initialization is fast because intensional items are not *initially* stored. Backward chaining then adds items to the hash table only if they are queried (and memoized), while forward chaining adds them only if they are provable (see footnote 7). The *final* storage size of $\mathcal{M}$ may differ in these two cases owing to the different choice of default. It can be more space-efficient—particularly in our hybrid strategy below—to choose different defaults for different types of items, reflecting the fact that some type of item is "usually" UNK or NULL (or even 0). One stores the pair $(i, \mathcal{M}[i])$ only when $\mathcal{M}[i]$ differs from the default for i. The datatype used to store $\mathcal{M}[i]$ does not need to be able to represent the default value.

6 Mixed Chaining With Selective Memoization

Both pure algorithms above are fully reactive, but sometimes inefficient. Backward chaining may redo work. Forward chaining requires storage for all items, and updates fully before answering a query. Yet each has advantages. Backward chaining visits only the nodes that are needed for a given query; forward chaining visits only the nodes that need updating.

A hybrid algorithm should combine the best of both, visiting nodes only as necessary and using $\mathcal{M}$ to materialize some useful subset of them. Our core insight is that

- The job of backward chaining is to *compute values for which the memo is missing* (UNK).
- The job of forward chaining is to *refresh any memos that are present but potentially stale.*
- Pure backward chaining is the case where *all memos are missing.* So a query triggers a cascade of backward computation; but forward chaining is unnecessary (no *stale* memos).
- Pure forward chaining is the dual case where *all memos are present.* So an initial or subsequent update triggers a cascade of forward computation; but backward chaining is unnecessary (no *missing* memos). We regard the arbitrarily initialized chart of section 5 as a complete but potentially stale memo table.

We will develop a hybrid algorithm that can memoize any subset of the intensional items. This subset can change over time: memos are optionally created while answering queries by backward chaining, and can be freely created or flushed at any time. Why bother? Computing only values that are needed for a given query can reduce asymptotic time and space requirements, a fact exploited by the magic sets technique [25]. Furthermore, materializing some or all of these values only *temporarily* can reduce the cost of storing and maintaining many memos. For example, [30] thereby solve the arithmetic circuit for the forward-backward algorithm in $O(\log n)$ rather than $O(n)$ space, while increasing runtime only from $O(n)$ to $O(n \log n)$.

6.1 Updates vs. Notifications

The essential (and novel) challenge here is to make forward chaining work with an incomplete memo table $\mathcal{M}$. Intuitively, we merely need to propagate updates as usual down through unmemoized regions of the circuit, so that they reach and refresh any stale memos below.

However, updates in such a region have a different nature. When we update the memo for an item i, each *visible* unmemoized descendant j remains *consistent* (in the terminology of section 4). After all, the result of calling LOOKUP(j) would *already* reflect the change to i.

Thus, what we propagate to j is not really an update but a **notification**. It does not say "change the value of j," but rather "the value of j has already [implicitly] changed." Crucially, this notification must be propagated to the descendants of j. When it finally reaches i's visible *memoized* descendants k—which became inconsistent the moment that i's memo was updated—it will trigger updates there to repair the inconsistencies.[8]

The agenda $\mathcal{A}$ now contains two kinds of **messages**: $\mathcal{A}[j]$ may be either an update to j or a notification from j. Recall from section 5 that an update to j is graphically displayed *above* the line. A notification from j is drawn as $j : f \underset{\leftarrow}{\underline{\quad}}$, with the change displayed *below* the line to indicate that it has already descended through item j. In this paper, the change is always a replacement by an unspecified (UNK) value, written textually as $j : \overline{\leftarrow}$.[9]

6.2 Push-Time Updates and Invalidations

The resulting code is shown in Figure 3. Our code also takes the opportunity to exploit notifications even for memoized items. In the old Listing 3, UPDATE(j, w) always enqueued $j : \underline{\leftarrow w}$ for later. Our new UPDATE(j, w) in Listing 4 can still choose that option provided that j is memoized (Line 4:8, a **pop-time update**), but its default is to APPLY the update immediately (Line 4:14, a **push-time update**). If so, it pushes only the notification $j : \overline{\leftarrow}$ and there is no need to APPLY the update at pop time. What *does* still happen at pop time is propagation: it is not until we pop an update *or* a notification to j (Line 5:5) that we visit j's children (Line 6:2).[10]

What happens if Line 6:4 is optionally skipped (so that $w =$ UNK)? Then the resulting UPDATE is a refresh update as before (section 5) if processed at pop time. However, if processed at push time, it is an **invalidation** update that deletes a memo instead of correcting it. Propagating invalidations can clear out stale portions of the chart at lower computational cost. Separately, the FLUSH method can also be called by the user or by FREELYMANIPULATEM (Listing 5) to delete individual memos without the need to propagate.

Like the forward chaining algorithm, the hybrid algorithm may start from any initial chart $\mathcal{M}$—but intensional items j now have the option of $\mathcal{M}[j] =$ UNK. The initial agenda does not contain any notifications, but as before, it must include enough updates to correct any inconsistencies in the initial chart. Since unmemoized intensional UNK items are always consistent by definition (section 4), the initial agenda never needs to have updates for them. For example, the **UNK initialization strategy** initializes just as in backward chaining (section 3), with extensional items set correctly and everything else initially UNK.

[8] This algorithm has a more complicated invariant than that of section 5: When k is inconsistent, the agenda contains *either* an update at k (as in section 5) *or* a notification at some visible ancestor of k.

[9] However, in general, *any* change that can appear in an update could appear in a notification, e.g., a more specific replacement $\overline{\leftarrow w}$, or a delta $\overline{\oplus w}$ (see section 7).

[10] Why allow both pop-time and push-time updates? Pop-time updates are required for correctness in certain settings involving delta updates (section 7). Also, pop-time updates include refresh updates, which are useful in avoiding premature computation of the new value w (footnote 5). On the other hand, push-time updates ensure fresher lookup results by immediately updating $\mathcal{M}[j]$ to a new value (or invalidating it to UNK). If the same update is deferred to pop time, then any calls to LOOKUP(j) while the update is waiting on the agenda will unfortunately get a stale memo for j, resulting in stale descendants that must be updated after the update pops.

6.3 Correctness: Avoiding A Subtle Bug

Returning to the setting of section 6.1, again suppose that i was updated, making its descendant k inconsistent, and that j is an unmemoized intermediate item on an i-to-k path.

Updating some *other* visible descendant of j (i.e., other than k) may cause j to get recursively looked up and optionally memoized before its notification arrives. If j gets memoized, it will receive an update rather than a notification. But the COMPUTE(j) that computes the update value will get the same answer as the COMPUTE(j) that computed the memo. That is, the memo $\mathcal{M}[j]$ was not stale but already reflected the change to i. This causes a subtle bug: forward chaining will discard the apparently unnecessary update, rather than propagating it on downward to k. Thus, k may remain inconsistent forever.

To prevent this bug, memoizing j must also enqueue a notification that the memo at j has been updated. The correct behavior is illustrated in Figure 4. This notification reflects the past update to i; it restores the invariant mentioned in footnote 8, and it will propagate down to k as desired. Such a notification must be enqueued when memoizing any item j such that COMPUTE(j) recursed to some item that had a notification on the agenda. The functions in Listing 7 return (as a second value) a flag that is TRUE if this condition holds, and enqueue the required notification at Line 7:22.

6.4 Efficiency: Obligation Tracking

Recall our challenge in section 4: backward chaining with optional memoization was a good algorithm, but to support UPDATE, we needed change propagation to refresh stale memos.

In our hybrid algorithm, we can use the UNK initialization strategy (section 6.2) to recover backward chaining. Change propagation will now be handled correctly.

Unfortunately, our propagation of notification through unmemoized regions is overly aggressive. For example, if *no intensional items have been memoized*, then change propagation should be completely unnecessary—this is the pure backward chaining case of section 4—and yet our algorithm will visit all descendants of an UPDATEd item! Our method visits all children of an updated item to check whether they too may need updating. In pure forward chaining, we can stop propagating (discard the update) at a child whose value is consistent; but for an UNK child the value is unknown, so we conservatively keep propagating.

In general, we should propagate down along an edge only when this may eventually reach a memoized descendant. This requires **obligation tracking**: for every item in the circuit, we desire to know if it has descendant memos which must be visited if its value changes.

We define the predicate $\text{obl}(\mathcal{A}[i], j)$ (used on Line 6:2) to mean that i is **obligated to** inform its child j of the update $\mathcal{A}[i]$. By definition, this is so if j is memoized or is in turn obligated to any of j's children. As a result, obligation of items in an arithmetic circuit $\mathcal{C}$ is naturally expressed as a boolean circuit $\mathcal{C}_{\text{obl}}$ that determines transitive reachability. Roughly speaking, $\mathcal{C}_{\text{obl}}$ has the same topology as $\mathcal{C}$ but with the edge direction reversed.

We can be even more precise about determining obligation. Specifically, in the recursive definition, i is *not* obligated to its child j if there is a notification at i or a refresh update at j. In these cases, j and its descendants are guaranteed to get refreshed anyway, so it is not necessary to propagate messages to j from i or its ancestors. One can again express this tighter definition as a boolean circuit $\mathcal{C}_{\text{obl}}$, whose boolean inputs are updated as $\mathcal{M}$ and $\mathcal{A}$ evolve. Line 6:2 then queries this $\mathcal{C}_{\text{obl}}$ using our algorithm.

We can maintain $\mathcal{C}_{\text{obl}}$ in turn using $(\mathcal{C}_{\text{obl}})_{\text{obl}}$, or by falling back to a *cheaper* obligation tracking strategy at this stage. For example, obligation tracking is cheap on a circuit that uses a memoization and flushing policy such that the memoized items always have memoized

```
Query(i ∈ 𝓘)
  RunAgenda()
  (v, ·) ← Lookup(i)    % · will be FALSE
  return v

Update(j ∈ 𝓘, w ∈ 𝓥 ⊔ {UNK})
  maybe
    if (𝓐[j] ≠ ⥢) ∧ (𝓜[j] ≠ UNK) then
      delete 𝓐[j]
      if (w = UNK) ∨ (𝓜[j] ≠ w) then
        𝓐[j] ← ←̲w̲
      return
    % else fall through
  Apply(j, w)

Flush(j ∈ 𝓘_int)
  if 𝓐[j] = ←̲·̲ then 𝓐[j] ← ⥢
  𝓜[j] ← UNK
```

Listing 4 User interface methods. (A user call to Update must have $j \in \mathcal{I}_{\text{ext}}, w \in \mathcal{V}$.)

```
Propagate(i ∈ 𝓘)
  foreach j ∈ C_i such that obl(𝓐[i], j)
    w ← UNK
    maybe (w, ·) ← Compute(j)
    Update(j, w)
  delete 𝓐[i]

% Convert update to notification
HandleUpdate(i : ←̲v̲)
  v_cur ← 𝓜[i]   % will not be UNK
  maybe if v = UNK then
    (v, ·) ← Compute(i)
  if v ≠ v_cur then   % else discard
    foreach j ∈ C_i maybe Lookup(j)
    maybe v ← UNK
    Apply(i, v)

Apply(j ∈ 𝓘_int, w ∈ 𝓥 ⊔ {UNK})
  𝓜[j] ← w
  𝓐[j] ← ⥢
```

Listing 6 Forward chaining internals.

```
RunAgenda()
 until 𝓐 = ∅
  FreelyManipulateM()
  peek u from 𝓐
  case u of
    i : ⥢ → Propagate(i)
    · : ←̲·̲ → HandleUpdate(u)

FreelyManipulateM()
 done ← FALSE
 until done
  foreach i ∈ 𝓘_int maybe Lookup(i)
  foreach i ∈ 𝓘_int maybe Flush(i)
  maybe done ← TRUE
```

Listing 5 Nondeterministic high-level control.

```
% Derive j's value from parents
Compute(j ∈ 𝓘_int)
  foreach i ∈ P_j
    (v_i, m_i) ← LookupFromBelow(i)
  FreelyManipulateM()
  return (f_j({i ↦ v_i | i ∈ P_j}),
          max_{i∈P_j} m_i)

% Interaction with forward chaining
LookupFromBelow(i ∈ 𝓘)
  m_c ← (𝓐[i] = ⥢)
  (v, m_t) ← Lookup(i)
  return (v, m_c ∨ m_t)

% Derive i's value from memo or parents
Lookup(i ∈ 𝓘)
  if 𝓜[i] ≠ UNK then
    return (𝓜[i], FALSE)
  (v,m) ← Compute(i)
  maybe
    𝓜[i] ← v
    if m then 𝓐[i] ← ⥢
  return (v, m)
```

Listing 7 Backward chaining internals.

Figure 3 The internals of our initial mixed-chaining algorithm, which combines forward and backward reasoning and supports arbitrary Flushes during execution.

parents. In that case, i is obligated to its child j only when j is memoized, which can be checked directly without an auxiliary circuit. Also, obligation tracking tolerates one-sided error: it is always safe for an obligation query to conservatively return TRUE, which at worst just results in unnecessary propagation. This leads to cheap approximate obligation tracking strategies, such as always returning TRUE, or coarse-to-fine approximations where the circuits $\mathcal{C}, \mathcal{C}_{\text{obl}}, (\mathcal{C}_{\text{obl}})_{\text{obl}}, \ldots$ are progressively smaller because a node in one circuit corresponds to a *set* of nodes in the previous circuit and is TRUE if *any* of them are obligated.

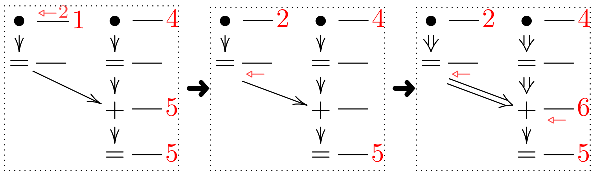

Figure 4 Backward chaining may need to enqueue notifications. After the top left update ("i") propagates to a notification $\overleftarrow{\leftarrow}$ at its child, the + item ("j") is flushed. Backward chaining from the + item (through un-memoized items) memoizes an up-to-date result of 6. Because backward chaining encountered a $\overleftarrow{\leftarrow}$, the memoization enqueues another $\overleftarrow{\leftarrow}$ at the + item, which ensures that its child ("k") will later be updated from 5 to 6.

6.5 Related Work

The recent constraint solver Kangaroo [24] was independently motivated by similar concerns. Like us, it mixes backward and forward chaining. In Kangaroo, queries seek out relevant updates—the reverse of our obligation approach, in which updates seek out relevant memoized queries. We are more selective about storage than Kangaroo, which stores memos at *all* nodes of the circuit.[11] On the other hand, Kangaroo is more selective about runtime. While it may have more memos, it updates only stale memos that are relevant to current queries, whereas our current algorithm updates all stale memos.

Previous mixed-chaining algorithms have been simpler. For functional programming, Acar et al. [1, 3] answer queries by backward chaining with *full* memoization; they update these memos by forward chaining of *replacement* updates. The same strategy is used for Prolog by Saha and Ramakrishnan [26, 27], who contrast it with the "DRed" strategy that forward-chains *invalidation* updates [16]. The "magic sets" transformation for Datalog [25] can be seen as a variant of these strategies. It uses only forward chaining, but restricted to items that would have been visited by backward chaining from the given query. All of these strategies memoize every computed item. In contrast, we are more economical with space.

Acar et al. [2] do separately consider *selective* memoization, but do not handle updates in this more challenging case (see section 6). A different selective strategy [19] relies primarily on *unmemoized* backward chaining. It first performs forward chaining on a given sub-circuit to identify and memoize a subset of TRUE values. However, this relies on the special property of Datalog that a TRUE node of a sub-circuit is also TRUE in the full circuit.

We believe that our framework can naturally be extended with richer computational strategies (see section 7). This is because it integrates fully selective memoization with a mixed chaining strategy, and because it has a general notion of an agenda of pending work, which can support a variety of update types, prioritization heuristics, and parallelizations.

7 Extensions

Some of the following extensions to our hybrid algorithm of section 6 are not too difficult, and we sketch them here. We defer full treatments to a longer version of this paper.

Richer Vocabulary of Updates For simplicity, this paper has focused on replacement updates i : $\underline{\leftarrow v}$. However, our prototype of Dyna [11] actually used agenda-based forward chaining with **delta updates** such as i : $\underline{\oplus\, v}$ for some operator $\oplus$. Applying this update at

[11] Selective memoization is an added reason for mixed chaining. Our forward chaining sometimes invokes backward chaining, in order to re-COMPUTE the value of a stale item with an unmemoized parent.

pop time increments the old memo $\mathcal{M}[i]$ to $\mathcal{M}[i] \oplus v$. Similarly, Dijkstra's shortest-path algorithm [7] chooses to use forward chaining with push-time delta updates, which are applied immediately and push delta notifications $i : \overline{\oplus v}$ onto the agenda (where $\oplus$ is min). A delta update at i is sometimes cheap to propagate to j, compared to a replacement update. This is because one can sometimes avoid a full call to Compute(j) at Line 6:4—which looks up or computes all the parents of j—by exploiting arithmetic properties such as distributivity of f_j over $\oplus$, or associativity and commutativity of $\oplus$ if $f_j = \oplus$.[12] Also, associativity and commutativity of $\oplus$ updates can be used to simplify the agenda data structure.

Circuit Transformation Even replacement updates can sometimes be propagated to j without a full call to Compute(j). Consider the case where f_j aggregates a large set of parents P_j using an associative binary operator. We can statically or dynamically transform the circuit to replace the direct edges from P_j to j with a binary **aggregation tree**. As this tree is just part of the circuit, it can use any strategy. In particular, if we maintain memos at the tree's internal nodes, then we can propagate a change from $i \in P_j$ to j in time $O(\log |P_j|)$.

Circuits can also be rearranged into more efficient forms by refactoring arithmetic expressions. It is possible to carry out such rearrangements by transforming the weighted logic program from which the circuit is derived [8]. But in principle, one might also rearrange the circuit locally as inference proceeds.

Aborting Backward Chaining by Guessing Our algorithm can be extended to handle cyclic arithmetic circuits.[13] Pure forward chaining can propagate updates around cycles indefinitely in hopes that the memos will converge [11]. If so, it finds a fixed-point solution $\overline{\mathcal{S}}$. But backward chaining does not work on the same circuit: it can recurse around the cycles forever without ever making progress by creating a memo. There is an interesting solution.

In general, we can interrupt any long backward-chaining recursion by allowing Compute(j) to optionally *guess* an arbitrary memo for $\mathcal{M}[j]$ (perhaps null). In this case we must enqueue a refresh update $j : \underline{\leftarrow}$, which serves as a continuation. Popping this update later will resume backward chaining and check that our guess at j is consistent with j's ancestors (perhaps including j itself, cyclically). If not, it will use the agenda to propagate a fix by forward chaining (perhaps cyclically until convergence). If j is already obligated to any children, we must also enqueue a notification $j : \overline{\leftarrow}$ to alert them that guessing $\mathcal{M}[j]$ may have changed it from the previous value of Lookup(j).

Fine-Grained Obligation Suppose j is an or node whose parent i has value true, or a $\times$ node whose parent i has value 0. As long as i has this value, j is *insensitive* to its other parents, who should not be obligated to propagate their updates to j. This generalizes the **watched variable trick** from the satisfiability community [22].

On-Demand Propagation Our current algorithm calls RunAgenda at the start of every Query, which refreshes all stale memos—including those that are not relevant to this query. This can be especially inefficient for cyclic or infinite circuits. We would prefer to propagate only the currently relevant updates, as in Kangaroo [24].

[12] This is slightly tricky when $\oplus$ is not idempotent, but solved in [11].

[13] Our current definition of obligation is overly broad in the cyclic case. It can create *self-supporting obligation*, where updates are unnecessarily propagated around cycles without actually refreshing any memos, merely because each item believes it is obligated to the next. Restoring efficiency in this case has been considered by [20].

Continuous Queries and Snapshots A **continuous query** of item i in an arithmetic circuit is a request to be notified (e.g., via callback) whenever UPDATES have caused QUERY(i) to change. Continuous queries are also used in databases and from functional reactive programming [13, 6]. Some users may also like to be notified of any updates that reach i as our algorithm runs, allowing them to **peek** at intermediate states $\mathcal{M}[i]$.

Programs We are actively working to extend the algorithms presented here to work not on arithmetic circuit descriptions directly but on Prolog-like weighted rules of Datalog with Aggregation [15, 5] and Dyna [9]. These programs can describe *infinite* generalized arithmetic circuits with value-dependent structure and with infinite fan-in or fan-out. A query, update, or memo may now be specified using a pattern that makes it apply to infinitely many items. This is the most challenging extension we have discussed.

8 Conclusion

We have developed a dynamic algorithm for solving arithmetic circuits and maintaining the solution under updates to the inputs. The solver can smoothly mix backward and forward chaining, while selectively memoizing results (and flushing memos). Different chaining and memoization strategies can be used as needed for different parts of the circuit, which does not affect correctness but can potentially improve time or space efficiency. Our framework also provides a basis for several extensions.

Acknowledgements This research was funded in part by the JHU Human Language Technology Center of Excellence. We would further like to thank John Blatz for useful early discussions, and an anonymous reviewer for calling our attention to Kangaroo [24].

References

1 Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proc. of POPL*, pages 247–259, 2002.

2 Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proc. of POPL*, pages 14–25, 2003.

3 Umut A. Acar and Ruy Ley-Wild. Self-adjusting computation with Delta ML. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

4 A. Borodin and I. Munro. *The computational complexity of algebraic and numeric problems.* Elsevier, 1975.

5 Sara Cohen, Werner Nutt, and Alexander Serebrenik. Algorithms for rewriting aggregate queries using views. In *Proc. of ADBIS-DASFAA*, pages 65–78, London, UK, 2000. Springer-Verlag.

6 Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

7 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

8 Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In Shuly Wintner, editor, *Proc. of FG 2006: The 11th Conference on Formal Grammar*, pages 45–85. CSLI Publications, 2007.

9 Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Tim Furche, Georg Gottlob, Oege de Moor, and Andrew Sellers, editors, *Datalog 2.0*, volume (to be published) of *LNCS*. Springer, 2011.

10 Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI (full version). Technical report, Johns Hopkins University, 2011. Available at `dyna.org/Publications`. A condensed version appeared as [9].

11 Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. of HLT-EMNLP*, pages 281–290, Vancouver, October 2005. Association for Computational Linguistics.

12 Gal Elidan, Ian Mcgraw, and Daphne Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*, 2006.

13 Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

14 Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in JAVA*. Wiley, 1998.

15 Sergio Greco. Dynamic programming in datalog with aggregates. *IEEE Transactions on Knowledge and Data Engineering*, 11(2):265–283, 1999.

16 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD Conference*, pages 157–166. ACM Press, May 1993.

17 Martin Kay. Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck Jones, and B. L. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Kaufmann, Los Altos, CA, 1986. First published in 1980 as Xerox PARC Technical Report CSL-80-12 and in the Proceedings of the Nobel Symposium on Text Processing, Gothenburg.

18 Dan Klein and Christopher D. Manning. A* parsing: Fast exact Viterbi parse selection. In *Proc. of HLT-NAACL*, 2003.

19 Thomas Labish. Developing a combined forward/backward-chaining system for logic programs in a hybrid expertsystem shell. Master's thesis, Universität Kaiserlautern, June 1993. In German.

20 Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. *IEEE 25th International Conference on Data Engineering*, 2009.

21 J.W. Lloyd and R.W. Topor. A basis for deductive database systems. *The Journal of Logic Programming*, 2(2):93 – 109, 1985.

22 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535, Las Vegas, NV, USA, June 2001. ACM.

23 Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

24 M. A. Hakim Newton, Duc Nghia Pham, Abdul Sattar, and Michael Maher. Kangaroo: An efficient constraint-based local search system using lazy propagation. In *17th International Conference on Principles and Practice of Constraint Programming*, pages 645–659, Perugia/Italy, September 2011.

25 Raghu Ramakrishnan. Magic templates: a spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3-4):189–216, 1991.

26 Diptikalyan Saha. *Incremental Evaluation of Tabled Logic Programs.* PhD thesis, Stony Brook University, December 2006.

27 Terrance Swift and David Scott Warren. XSB: Extending Prolog with tabled logic programming. *CoRR*, abs/1012.5123, 2010. Under consideration for publication in Theory and Practice of Logic Programming.

28 Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

29 Alan G. Yoder and David L. Cohn. Domain-specific and general-purpose aspects of spreadsheet languages. In *Proceedings of the Workshop on Domain-Specific Languages*, 1997.

30 G. Zweig and M. Padmanabhan. Exact alpha-beta computation in logarithmic space with application to map word graph construction. In *Proceedings of ICSLP*, 2000.