Set-at-a-Time Solving in Weighted Logic Programs

NATHANIEL WESLEY FILARDO and JASON EISNER
Johns Hopkins University, Baltimore, MD
(*e-mail:* {nwf,jason}@cs.jhu.edu)

submitted 4 May 2017; revised 4 May 2017; accepted ?

Abstract

Weighted logic programs specify potentially infinite generalized computational circuits; here, we consider an extension of our solver [7] to handle these circuits. To maintain computational tractability and to improve efficiency, we extend that algorithm with set-at-a-time reasoning, wherein potentially infinite sets of structurally related steps of the solver are processed as a single step. The new algorithm manipulates representations of sets and bags of trees generated by the weighted logic program being solved. It assumes, among other operations, the ability to take unions, intersections, and differences of such sets, to test whether they are subsets of one another, and to determine their cardinality. We discuss some simple cases where this assumption is reasonable before introducing a proposal to weaken the assumption and allow for a wider class of programs.

1 Introduction

A generalized computational circuit is a directed, possibly cyclic graph in which each non-root node specifies a function. A labeling of the graph maps each node to a value. Roots (nodes with no parents) may be given any values, but the value at a non-root node must be the result of applying that node's function to its parents' values.

Circuits can be specified by weighted logic programs [5], allowing them to be infinite—including having infinite fan-in, infinite fan-out, or infinitely long cyclic or acyclic paths. The nodes of the circuit are ground terms (known as **items**). Recall that the Prolog rule "`rs(X) :- r(X,Y), s(Y)`" asserts that `rs(x)` is true if $\exists$`y` such that `r(x,y)` and `s(y)` are both true. Upper-case `X` denotes a variable which ranges over lower-case atoms such as `x` or (nested) terms such as `f(a,g(b))`. In this rule, "`:-`" acts as *disjunction* across all *conjunctive* expressions of the form "`r(x,y), s(y)`". We generalize Prolog so that instead of defining when a ground term such as `rs(x)` is true, it constrains the term's **value** (or "weight"). Our rules now look like "`rs(X)` $\oplus$`= r(X,Y)` $\otimes$ `s(Y)`," which effectively defines a vector `rs` by adding ($\oplus$) the product ($\otimes$) of the matrix `r` and the vector `s`. For each item given by instantiating the variables {`X`} in the rule head, $\bigoplus$ aggregates the values of the expressions given by instantiating the remaining variables {`Y`} in the rule body. Thus, the resulting program constrains item `rs(x)` for each index `x` to have value $\left(\bigoplus_{\texttt{y}} \texttt{r(x,y)}\right) \oplus v$, where v is the $\oplus$ aggregation of all summands contributed to `rs(x)` by other rules. The parents of `rs(x)` in the resulting circuit include `r(x,y)` and `s(y)` for all `y`, and the children of each `s(y)` include `rs(x)` for all `x`. In our rules, there is no restriction on the datatypes of values or on the functions such as $\otimes$ that combine values within a rule body, but each rule's **aggregator** $\oplus$ must be associative and commutative.

In prior work [7], we gave a flexible algorithm to handle queries and updates on finite,

possibly cyclic circuits. The algorithm therein includes a method `Compute` to compute a given node's value from its parents' values as obtained via a `Lookup` method (which finds them cached in a memo table or else recursively calls `Compute`). When the circuit is specified by a weighted logic program as above, this task generalizes to one step of backward-chained reasoning, which computes the values of a given *set* κ of nodes, such as might be specified by a non-ground term. The present paper addresses this surprisingly challenging problem, culminating in the `Compute`(κ) routine of Listing 1, which returns a representation of a *piecewise constant* map from all items in κ to their values.

Although the high-level strategy resembles Prolog's SLD resolution [11], the complexity arises from the fact that a goal or subgoal query may match the heads of multiple overlapping rules, and thus—after aggregation across rules—may return a complicated map from items to values. While we focus on backward chaining and do not cover [7]'s additional machinery for forward chaining and revision of cached answers, we conjecture that these components will readily generalize, as they are consumers of backward reasoning.

We must generalize Prolog's ability to reason about *sets* of items all at once. Prolog's solver does this using non-ground terms (partitioning them as needed by by unification against the heads of rules). For example, in a program that supports proofs of the *non-ground* terms `r(1,Y)` and `s(Y)`, the above `rs` rule would construct *infinitely many* justifications of `rs(1)` at once—all of the form `r(1,Y), s(Y)`. The same rule would combine proofs of `r(X,2)` and `s(2)` to justify infinitely many `rs(X)` items at once, including contributing an additional justification of the `rs(1)` item. Generalizing to the weighted case, our circuit semantics specifies that each item's value is given by aggregating the values of expressions corresponding to all its justifications. Because `rs(1)` has an additional justification compared to the other `rs(X)` items, it will have a different total value, meaning that when `rs(X)` appears in some other expression, it is necessary to distinguish two cases $\texttt{X} = \texttt{1}, \texttt{X} \neq \texttt{1}$. These issues do not arise in the restricted case of Prolog, nor even in semiring-weighted logic programs [16, 6], where distributivity properties make it possible to process the aggregands separately. (This separate processing is why a Prolog query leads to a stream of proved answers that may include unaggregated duplicates.)

Our treatment is intended to be applicable to any weighted logic language. Our particular formalism herein, μDyna, pronounced "micro-Dyna," is built up of little more than set theory and has a straightforward, declarative semantics, detached from any particular implementation, and devoid of metalogical escapes. We first introduce some notation (§1.1) before describing our model weighted logic language in more detail (§1.2). So armed, we begin with brief consideration of *ground* reasoning, under the assumption of a finite item universe (§2). We then motivate our non-ground reasoning with an example (§3.1) before discussing the generalizations to reasoning and giving an algorithm (§3.2-3.4). We then step back to assess what we have wrought, considering μDyna in more generality (§3.5) and some special cases of μDyna programs in which our algorithm may be tractable (§3.6). We discuss related work in §4.

1.1 Notation

Sets We adopt set- and bag- theoretic semantics throughout this paper, relying on a well-typed underlying theory. Sets are constructed and manipulated by the typical operators (e.g., $\{\ldots\}$, $\cup$, $\cap$, $\in$, $\subseteq$, $\setminus$). $\wp$ sends a set to its powerset; $\wp_{\text{fin}}$ sends a set to its set of *finite* subsets. $|\sigma|$ denotes the cardinality of σ; we take cardinalities to be limited

to $\mathbb{N}_\infty \stackrel{\text{def}}{=} \mathbb{N} \cup \{\infty\}$. (That is, naturals and only one infinity. The $(\mathbb{N}, +, 0, \cdot, 1)$ semiring extends as might be expected: $m + \infty \stackrel{\text{def}}{=} \infty$ for all $m \in \mathbb{N}_\infty$. $0 \cdot \infty \stackrel{\text{def}}{=} 0$, and $m \cdot \infty \stackrel{\text{def}}{=} \infty$ for $m \neq 0$.) The *partial* function $\text{selt}(\sigma)$ projects a singleton set to its element: $\text{selt}(\{s\}) \stackrel{\text{def}}{=} s$. We use the shorthands $\bigcup \sigma \stackrel{\text{def}}{=} \bigcup_{s \in \sigma} s$ and $\mathbb{N}_1^n \stackrel{\text{def}}{=} \{1, 2, \ldots, n\} \subsetneq \mathbb{N}$.

Bags We will use $\{\!|\ldots|\!\}$ for bag literals. $\{\!|s@m|\!\}$ denotes a bag holding exactly m copies of s, also said as s with **multiplicity** $m \in \mathbb{N}_\infty$ (with 0 being identified with absence from the bag). Multiplicities of 1 may be suppressed: $\{\!|s|\!\} = \{\!|s@1|\!\}$. As with sets, comprehension notation may be employed, quantifying over both elements and multiplicities. Bag multiplicities *add*, so $\{\!|s@m_1, s@m_2|\!\} = \{\!|s@(m_1 + m_2)|\!\}$. The traditional symbols of set-theory with a plus sign superimposed will be used for bag operations: $\uplus$, $\biguplus$, $\dot{\in}$, etc., though we overload $\varnothing$ for the empty bag as well. $\wp_+\sigma$ denotes the *set* of all sub-bags of bag σ. $U_m^{-1}\sigma$ is the bag whose elements are from σ, all with multiplicity m.

Tuples We assume our theory contains n-ary **tuples**, denoted $\langle t_i \rangle_{i \in \mathbb{N}_1^n} \stackrel{\text{def}}{=} \langle t_1, \ldots, t_n \rangle$; we use $\vec{t}$ when n is clear from context. A **pair** is a tuple of length 2. $+\!\!+$ is the associative tuple concatenation operator. The length of a tuple is denoted $\text{tlen}(\langle t_1, \ldots, t_n \rangle) \stackrel{\text{def}}{=} n$; $\langle\rangle$ is the tuple of length 0. For ease of reading, this paper will use color-matched brackets for deeply nested tuple structures, e.g., $\langle \langle\rangle, \langle \langle a \rangle, b \rangle \rangle$. We define a **projection** operator, denoted $\cdot\downarrow$, to access components of (nested) tuples. $\vec{t}\downarrow_k$, for $k \in \mathbb{N}_1^n$, means simply the k-th component of $\vec{t}$, i.e., t_k. More generally, we write $\vec{t}\downarrow_{k_1.k_2.k_3}$ to mean the k_3-th component of the k_2-th component of the k_1-th component of $\vec{t}$. The subscript $k_1.k_2.k_3$ is called a **path** and can in general be any tuple of positive integers; π denotes such a path.[1] We will often define mnemonic NAMEs for particular common path prefixes.

Sets of Tuples Dependent sums are written $\Sigma_{s \in \sigma} Y_s \stackrel{\text{def}}{=} \{\langle s, t_s \rangle \mid s \in \sigma, t_s \in Y_s\}$, where Y is a σ-indexed collection of sets. (Dually, *every* set of pairs is a dependent sum of *some* indexed collection.) As we will often have sets described by tuples of elements sampled from a product of other sets, we introduce a product-forming tuple operator, $\langle \sigma, \tau \rangle \stackrel{\text{def}}{=} \{\langle s, t \rangle \mid s \in \sigma, t \in \tau\}$. Projection is extended to sets: $\sigma\downarrow_\pi \stackrel{\text{def}}{=} \{s\downarrow_\pi \mid s \in \sigma\}$. Since restricting focus to a particular path in a *set* of tuples may yield a smaller set, we have a **bag-view projector** as well: $\sigma\downarrow_\pi^{@} \stackrel{\text{def}}{=} \{\!|s\downarrow_\pi \mid s \in \sigma|\!\}$.[2] In addition to projection from sets, we will make extremely heavy use of **refinement**, $\sigma[\tau/\pi] \stackrel{\text{def}}{=} \{s \in \sigma \mid s\downarrow_\pi \in \tau\}$.

Functions The set of total functions from set σ to set τ is denoted $\sigma \to \tau$. When the codomain is potentially dependent upon the input, we use the dependent product operator: $\Pi_{s \in \sigma} Y_s$ where Y is a σ-indexed collection of *sets*.[3] ($\sigma \to \tau$ is just the special case of a constant collection, i.e., $\exists_\tau \forall_{s \in \sigma} Y_s = \tau$.) The domain of a function $f \in \Pi_{s \in \sigma} Y_s$ is denoted $\text{dom}(f) \stackrel{\text{def}}{=} \sigma$. Functions may be written as sets of tuples, i.e., $\{s \mapsto t \mid \cdots\}$, where s is from the domain and t is its corresponding element of the (dependent) codomain and $\mapsto$ is simply an infix pair constructor (i.e., $(a \mapsto b) \stackrel{\text{def}}{=} \langle a, b \rangle$). We also use this notation in

[1] Formally, we should write e.g., $\pi = \langle k_1, k_2, k_3 \rangle$, but the dot notation is standard and risks less confusion. Projection is defined inductively: $t\downarrow_{\langle\rangle} \stackrel{\text{def}}{=} t$ (even for non-tuple t) and $\langle \tau_1, \ldots, \tau_n \rangle\downarrow_{\langle k \rangle +\!\!+ \pi} \stackrel{\text{def}}{=} \tau_k\downarrow_\pi$.

[2] Our restriction on multiplicities implies $\{\langle 2, r \rangle \mid r \in \mathbb{R}\}\downarrow_1^{@} = \{\langle 2, n \rangle \mid n \in \mathbb{N}\}\downarrow_1^{@} = \{\!|2@\infty|\!\}$.

[3] For readers unfamiliar with the notation and alarmed by the apparent reuse of numeric product notation for something completely different, a worthwhile exercise is to demonstrate, for all sets α and α-indexed collections of sets Y, that $|\Pi_{a \in \alpha} Y_a| = \Pi_{a \in \alpha} |Y_a|$, where the quantifier on the left is our set-theoretic one and that on the right is the numeric product operator operating on cardinalities. We avoid the term "function" for Y to skirt the question of its codomain.

quantification, e.g., $\{\varphi(s,t) \mid s \mapsto t \in f\}$, to range over the domain of a function (so, e.g., $\mathrm{dom}(f) = \{s \mid s \mapsto t \in f\}$). Functions can be constructed out of other notation by use of the usual **argument placeholder**, "$\cdot$": e.g., if a is a σ-indexed family of objects from a set τ then by "$a_\cdot$" (subscripted with $\cdot$) we mean the function $\{s \mapsto a_s \mid s \in \sigma\} \in (\sigma \to \tau)$.

Terms We assume a Herbrand universe $\mathcal{H}$ from the underlying collection of symbols $\mathcal{F}$; elements of $\mathcal{H}$ are called **(ground) terms**. Explicitly, $t \in \mathcal{H}$ is composed of a **functor** with fixed arity $n \in \mathbb{N}$, denoted $\mathtt{f}/n \in \mathcal{F}$, and a tuple of n terms: $t = \mathtt{f}\langle t_1, \ldots, t_n\rangle$; functors with arity 0 form the base case. A **non-ground term** is a subset of $\mathcal{H}$.[4] We use the product-forming tuple operator as a shorthand for non-ground terms: $\mathtt{f}\langle \tau_1, \ldots, \tau_n\rangle \stackrel{\text{def}}{=} \{\mathtt{f}\langle t_1, \ldots, t_n\rangle \mid \forall_i t_i \in \tau_i\}$. Projection is extended, within its inductive definition, to work on trees as well as on tuples by ignoring any functors along the path: e.g., $\langle \mathtt{t}\langle t_1, t_2\rangle, x\rangle\downarrow_{1.2} = t_2$; this applies to the extension to sets of tuples as well. We will avail ourselves of a symbol $\textsc{null} \notin \mathcal{H}$ to indicate the absence of a value assigned to an item. Let $\mathcal{H}^+ \stackrel{\text{def}}{=} \wp_+ U_\infty^{-1} \mathcal{H}$ be the set of all bags of terms and, with NULLs, let $\mathcal{H}' \stackrel{\text{def}}{=} \{\textsc{null}\} \cup \mathcal{H}$ and $\mathcal{H}'^+ \stackrel{\text{def}}{=} \wp_+ U_\infty^{-1} \mathcal{H}'$.

Aggregation Functions Our programs need to reduce an arbitrary bag of results (i.e., terms and NULLs, which we call **aggregand**s) to a single term. We call the functions which do so **aggregators**: functions $f \in \mathcal{H}'^+ \to \mathcal{H}'$ which obey $f(\varnothing) = \textsc{null}$, $\forall_{a \in \mathcal{H}'} f(\{\!|a|\!\}) = a$, and $\forall_{\sigma,\sigma'} f(\sigma \uplus \sigma') = f(\{\!|f(\sigma)|\!\} \cup \sigma')$, which ensure that f acts like reduction by an associative-commutative binary operator with identity element NULL. We add a further requirement $f(\{\!|\textsc{null}@\infty|\!\}) = \textsc{null}$ to ensure that all NULL aggregands are ignored.[5]

1.2 μDyna Normal-Form Programs

We define μDyna, a minimal, set-theoretic, "administrative" normal-form [9] of weighted logic programs. A μDyna program consists of several components: ① its set of items, $\mathcal{I} \subseteq \mathcal{H}$; ② a map from items to their aggregation operators, $\mathrm{aggr} \in \mathcal{I} \to (\mathcal{H}'^+ \to \mathcal{H}')$; and ③ a *bag* of ($\mu$Dyna) rules, $\{\!|\rho_r \mid r \in \Xi|\!\}$, where Ξ is a finite *set of rule indices*. A μDyna rule has three major parts: a **head** (an item name), a **result**, and a **body**. The **body** is a tuple of **subgoals**, which are pairs of a *key* and a *value* (in that order, i.e., $\langle \mathrm{key}, \mathrm{value}\rangle$), and which request the value of the item named by the key.[6] A rule **grounding**, then, is a nested tuple over these components: $\langle\langle \mathrm{head}, \mathrm{result}\rangle, \langle \mathrm{subgoal}_1, \ldots, \mathrm{subgoal}_n\rangle\rangle$. Each grounding of a μDyna rule reads as an instruction: "aggregate the *result* into the *head* if each *subgoal's key* has been assigned the corresponding *value*"; groundings which satisfy this condition are called **rule answers**. The set of rule answers will vary if items' values change (e.g., during a solver's execution or in response to updates external to the solver). Generalizing, a μDyna rule ρ_r is a set containing *all possible groundings* of this rule, from which the rule answers will be selected. Our example of a weighted logic language rule

[4] The identification of a non-ground term with its set of groundings is perhaps unusual; most alternative expositions add an explicit notion of *variable* object within terms and a notion of binding contexts which map variables to trees and/or other variables. Set refinement generalizes variable substitution.

[5] Taking the special value NULL to be the identity element of every aggregator ensures that every aggregator *has* an identity. It also ensures that the sum of no elements (namely NULL) is different from the sum of $\{\!|5, 0, -5|\!\}$ (namely 0). The former is ignored in subsequent aggregations, yielding a different result if the subsequent aggregator is a different operator such as max.

[6] Readers familiar with Prolog may think of a subgoal $\langle k, v\rangle$ as another rendering of v `is` k. In μDyna, *every* subgoal is an `is/2` subgoal, though one which evaluates against the program rules rather than a built-in database and which is not restricted to the Prolog mode "`-Number is +Expr`", where `-` and `+` mean free and ground structure, respectively.

from above, `rs(X) ⊕= r(X,Y) ⊗ s(Y)`, is now rendered as

$$\rho = \{\langle \overbrace{\langle \underbrace{\mathtt{rs}\langle x\rangle}_{\text{HEAD}}, \underbrace{z}_{\text{RES}}\rangle}^{\text{HR}}, \overbrace{\langle \mathtt{r}\langle x, y\rangle \mapsto \underbrace{r}_{\text{SG.1.2}}, \mathtt{s}\langle y\rangle \mapsto s, \underbrace{\otimes\langle r, s\rangle}_{\text{SG.3.1}} \mapsto z\rangle}^{\text{SG}} \rangle \mid r, s, x, y, z \in \mathcal{H}\}.$$

Recall that $\mapsto$ is simply an infix pair constructor; we use it here as a mnemonic between subgoal key and value even though there is no functional dependence in ρ. We have annotated the rule with several paths and given mnemonics to particular prefixes, $\text{HR} \stackrel{\text{def}}{=} 1$, $\text{HEAD} \stackrel{\text{def}}{=} 1.1$, $\text{RES} \stackrel{\text{def}}{=} 1.2$, and $\text{SG} \stackrel{\text{def}}{=} 2$, to help clarify later operations. Variables used more than once within the set element constructor give rise to *covariance* between different positions within a rule: above, the RES and SG.3.2 projections are equated (by reuse of z). Our formal theory *does not* use variables; they are merely notation to help specify sets.

Formally, sets ρ_r used as μDyna rules obey five constraints: ① projections along HEAD, RES, and SG are defined for all elements of the set; ② the head and result are terms, i.e., $\forall_{t\in\rho_r,\pi\in\{\text{HEAD},\text{RES}\}} t\downarrow_\pi \in \mathcal{H}$; ③ the number of subgoals in r, denoted n_r, is *constant* across all groundings of the rule, i.e., $\forall_{r\in\Xi,s\in(\rho_r\downarrow_{\text{SG}})} \text{tlen}(s) = n_r$; ④ each subgoal is itself a pair of two terms, i.e., $\forall_{t\in\rho_r,i\in\mathbb{N}_1^{n_r},j\in\{1,2\}} t\downarrow_{\text{SG}.i.j} \in \mathcal{H}$;[7] and ⑤ the subgoals and head determine the grounding, i.e., $\forall_{\alpha\subseteq\rho_r} |\alpha\downarrow_{\text{SG}}| = |\alpha\downarrow_{\text{HEAD}}| = 1 \Rightarrow |\alpha| = 1$ (and, in particular, that $|\alpha\downarrow_{\text{RES}}| = 1$). These clearly hold for the example above: ① these projections clearly exist, ② $\mathtt{rs}\langle x\rangle$ and z are terms, ③ there are exactly three subgoals in any grounding, ④ subgoal keys and values are terms, and ⑤ the reuses of x and z together imply the stronger statement $\forall_{\alpha\subseteq\rho} |\alpha\downarrow_{\text{SG}}| = 1 \Rightarrow |\alpha| = 1$.

A **rule query** $\vec{t}$ for a rule r is a n_r-tuple of items. A rule query gives rise to a set of **pre-answers** by refining the subgoal keys: $\theta_r^{\vec{t}} \stackrel{\text{def}}{=} \rho_r[\{t_1\}/\text{SG}.1.1]\cdots[\{t_{n_r}\}/\text{SG}.n_r.1]$. $\vec{t}$ is **trivial** if $\theta_r^{\vec{t}} = \varnothing$. For example, $\langle \mathtt{r}\langle 1, 2\rangle, \mathtt{s}\langle 3\rangle\rangle$ is a trivial rule query for the rs rule, because $2 \neq 3$. Given an item valuation function $\mathcal{S} \in \mathcal{I} \to \mathcal{H}'$, one can filter pre-answers to the set of rule answers, $\epsilon_{r,\mathcal{S}}^{\vec{t}} \stackrel{\text{def}}{=} \theta_r^{\vec{t}}[\mathcal{S}(t_1)/\text{SG}.1.2]\cdots[\mathcal{S}(t_{n_r})/\text{SG}.n_r.2]$.[8] If any $\mathcal{S}(t_i) = \text{NULL}$, then ϵ is $\varnothing$. As all subgoal projections have been refined to singletons within rule answers, the constraints on μDyna rules imply that $\forall_{h,r,\vec{t},\mathcal{S}} |\epsilon_{r,\mathcal{S}}^{\vec{t}}[\{h\}/\text{HEAD}]| \leq 1$.

2 Ground Reasoning

Let us begin by considering the special case in which there are only finitely many items. In this case, we could imagine that `Lookup` takes a possibly-infinite set of terms κ—an (item) **query**—and returns an **answer** consisting of a (finite!) map from those items in κ to their corresponding values. Formally, $\texttt{Lookup} \in \Pi_{\kappa\subseteq\mathcal{H}} \bigcup_{\alpha\in\wp_{\text{fin}}(\kappa\cap\mathcal{I})} (\alpha \to \mathcal{H})$. Let us assume that we can ensure that `Lookup` gives us a *coherent* view of items' values, so that if we call it repeatedly with overlapping κs, the resulting maps will agree on the overlapping region; in a real implementation, this may require some kind of *concurrency control*. To interpret ρ's influence on the item h against the backdrop provided by `Lookup`, we would compute $\rho[\{h\}/\text{HEAD}]$ and then visit each subgoal in turn, projecting its key for `Lookup`, and then

[7] It is, therefore, impossible to define rules which explicitly match on NULL and we need not define that, by default, NULL passes through subgoals, e.g., by asserting that $1 + \text{NULL} = \text{NULL}$. Any attempt to refine a subgoal value to $\{\text{NULL}\}$ will immediately empty the set of groundings. Thus, while NULL is an identity of aggregation, it is an *annihilator* of the conjunction of rules' subgoals.

[8] There is no analog of pre-answers sitting between (item) queries and (item) answers: pre-answers emerge due to the *conjunction* of queries, i.e., interactions among the subgoals of a rule. In this paper, they will not appear computationally.

use each point in the resulting map to refine the rule before visiting the next subgoal or, should there be none left, arriving at a rule answer. Procedurally, we would define a recursive function `REFINERULESUFFIX`$(\sigma \subseteq \rho_r, i \in \mathbb{N}_1^{n_r+1})$ which interpreted a *suffix* of a rule subset σ, i.e., all subgoals at and after the `i`-th position, and called some function `CONTRIBRULEANSWER`$(t \in \rho_r)$ on each obtained rule answer. (Throughout our code listings, we will use special formatting for `PROCEDURES`, `reserved words`, and type annotations.) That is, we would write:

```
1 def REFINERULESUFFIX(σ ⊆ ρ_r, i ∈ ℕ_1^{n_r+1})
2   if σ = ∅ then return
3   else if i = n_r + 1 then CONTRIBRULEANSWER(selt(σ))
4   else foreach (k ↦ v) ∈ LOOKUP(σ↓_{SG.i.1}) do
5     REFINERULESUFFIX(σ[{⟨k, v⟩}/SG.i], i + 1)
```

The constraints on μDyna rules ensure that the call to selt$(\cdot)$ will succeed: the HEAD and SG projections of σ have been brought to singletons. Moreover, `CONTRIBRULEANSWER` will be called only finitely many times, as each nested loop has finite domain.

While correct, the procedure above works only for one item h at a time! Rather than having to iterate our (finite, but possibly large) item set, we would surely much rather let the facts guide us, à la SLD resolution. If the rules of our program all obey **range restriction** [1], the stronger constraint that subgoals alone determine the grounding, i.e., $\forall_{\alpha \subseteq \rho} |\alpha\downarrow_{\text{SG}}| = 1 \Rightarrow |\alpha| = 1$,[9] then we can skip the initial selection of the head h and still be assured that selt$(\cdot)$ succeeds. We now see how to execute a single step of backward reasoning for some query set $\kappa \subseteq \mathcal{I}$: visit each rule r, compute $\sigma = \rho_r[\kappa/\text{HEAD}]$, and invoke `REFINERULESUFFIX`$(\sigma, 1)$, letting `CONTRIBRULEANSWER` accumulate all obtained results. Connecting this back to our sets, we see that `REFINERULESUFFIX` computes the set of rule answers $\epsilon^{\vec{t}}_{r,\mathcal{S}}$, by interleaving the refinements given in the definitions of θ and ϵ, with `LOOKUP` as $\mathcal{S}$.

We can render the actions of `REFINERULESUFFIX` as a search tree. Non-leaf nodes represent invocations of `LOOKUP` on subgoals, and their outgoing edges represent answer item/value pairs; leaves are either $\varnothing$ or represent rule answers. In this light, our assumption that `LOOKUP` returns finite maps now ensures that every node has *finite branching factor* (and, thus, finitely many leaves) and range restriction ensures that each leaf corresponds to at most one rule answer.

3 Non-ground Reasoning

3.1 A Motivating Example

We now seek a *set-at-a-time* execution strategy, which attempts to reason about *sets* of similarly-behaving items at once. For example, the rule `f(X,Y) ⊕= 1` defines an aggregand for each of the infinitely many terms of $\texttt{f}\langle\mathcal{H}, \mathcal{H}\rangle$, but the pattern is so simple that all these terms can be considered *at once*.

Example 1. This kind of *bulk handling* extends across rules, too. The pair of rules `f(1,Y) ⊕= 3` and `f(X,2) ⊕= 4` will contribute $3 \oplus 4$ to the aggregated value of item `f(1,2)`,

[9] The usual notion of range restriction is that variables appearing in the head must also appear in a subgoal; the requirement given here is a trivial generalization to our set-based, weighted setting. The current requirement excludes, e.g., $\{\langle\langle \texttt{f}\langle x, y\rangle, v\rangle, \langle \texttt{g}\langle x\rangle \mapsto v\rangle\rangle \mid \cdots\}$.

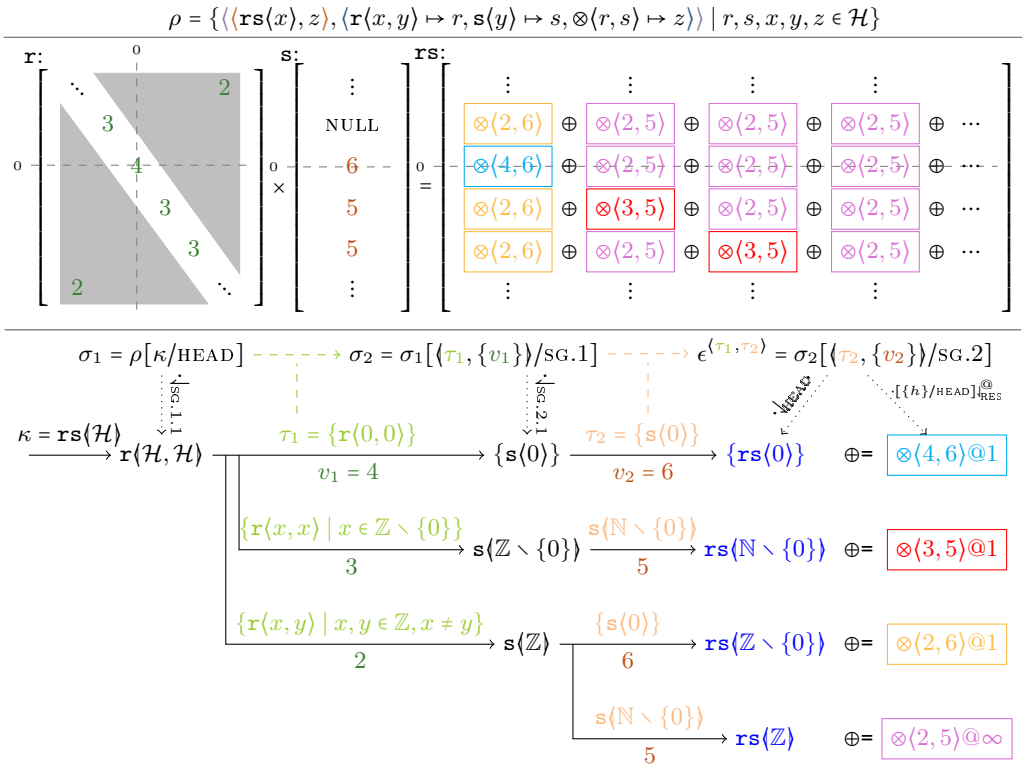

Fig. 1: Our rule `rs(X) ⊕= r(X,Y) ⊗ s(Y)`, shown in μDyna form at the top, can perform the computation shown in the middle—the product of an infinite matrix (all of whose off-diagonal elements are 2) with an infinite vector. To obtain the answer, we call COMPUTE with query κ = `rs`$\langle\mathcal{H}\rangle$. The bottom of the figure shows a search tree that computes the aggregands of the answers. The root represents the initial query of the first subgoal `r`, and the edges from the root correspond to the branches of answers returned by LOOKUP. Each such edge leads to a new node with some refined query of the second subgoal `s`, and the edges from that node again correspond to answers. Each such edge leads to a leaf that specifies some subset of the head `rs` and contributes some aggregand at some multiplicity (colored box) to all items in that subset. (We elide the handling of the third subgoal, $\otimes$, as it is only queried on singleton sets and so no branching is possible.) Thus, the leaves (at right) correspond to rule answers (ϵ sets). The rule answers must be further partitioned and aggregated (see §3.3) to yield the answers to the original query `rs`$\langle\mathcal{H}\rangle$, namely $\{$`rs`$\langle 0\rangle \mapsto \bigoplus\{\!\{\otimes\langle 4,6\rangle@1, \otimes\langle 2,5\rangle@\infty\}\!\}$, `rs`$\langle\mathbb{N}\setminus\{0\}\rangle \mapsto \bigoplus\{\!\{\otimes\langle 3,5\rangle@1, \otimes\langle 4,6\rangle@1, \otimes\langle 2,5\rangle@\infty\}\!\}$, `rs`$\langle\mathbb{Z}\setminus\mathbb{N}\rangle \mapsto \bigoplus\{\!\{\otimes\langle 4,6\rangle@1, \otimes\langle 2,5\rangle@\infty\}\!\}\}$. The shape of the search tree is determined by the answers from LOOKUP, which returns disjoint slices of the `r` matrix $\{$`r`$\langle 0,0\rangle \mapsto 4, \{$`r`$\langle x,x\rangle \mid x \in \mathbb{Z}\setminus\{0\}\} \mapsto 3, \{$`r`$\langle x,y\rangle \mid x,y \in \mathbb{Z}, x \neq y\} \mapsto 2\}$ and the `s` vector $\{\{$`s`$\langle 0\rangle\} \mapsto 6, \{$`s`$\langle t\rangle \mid t \in \mathbb{N}\setminus\{0\}\} \mapsto 5\}$.

while also contributing 3 to each item in $\{$`f`$\langle 1, y\rangle \mid y \in \mathcal{H}\setminus\{2\}\}$ and 4 to each item in $\{$`f`$\langle x, 2\rangle \mid x \in \mathcal{H}\setminus\{1\}\}$. ◊

A more dramatic example is shown in Figure 1: the product of a infinite matrix with an infinite vector, exploiting the fact that both have simple definitions. The matrix `r` is defined by the cases $\{$`r`$\langle 0,0\rangle\} \mapsto 4$, $\{$`r`$\langle x,x\rangle \mid x \in \mathbb{Z}\setminus\{0\}\} \mapsto 3$, and $\{$`r`$\langle x,y\rangle \mid x,y \in \mathbb{Z}, x \neq y\} \mapsto 2$, where $\tau \mapsto v$ means that v is the value of each $t \in \tau$. Similarly, the vector `s` is defined by $\{$`s`$\langle 0\rangle\} \mapsto 6$ and $\{$`s`$\langle y\rangle \mid y \in \mathbb{N}\setminus\{0\}\} \mapsto 5$. For simplicity, assume $\otimes$ is total. In this case, our `rs` rule should answer non-ground rule queries as follows:

① $\{\langle \langle \mathtt{rs}\langle 0\rangle, \otimes\langle 4,6\rangle\rangle, \langle \mathtt{r}\langle 0,0\rangle \mapsto 4, \mathtt{s}\langle 0\rangle \mapsto 6, \cdots\rangle\rangle\}$, from the query $\langle\{\mathtt{r}\langle 0,0\rangle\}, \{\mathtt{s}\langle 0\rangle\}, \ldots\rangle$;
② $\{\langle \langle \mathtt{rs}\langle x\rangle, \otimes\langle 3,5\rangle\rangle, \langle \mathtt{r}\langle x,x\rangle \mapsto 3, \mathtt{s}\langle x\rangle \mapsto 5, \cdots\rangle\rangle \mid x \in \mathbb{N} \setminus \{0\}\}$;
③ $\{\langle \langle \mathtt{rs}\langle x\rangle, \otimes\langle 2,6\rangle\rangle, \langle \mathtt{r}\langle x,0\rangle \mapsto 2, \mathtt{s}\langle 0\rangle \mapsto 6, \cdots\rangle\rangle \mid x \in \mathbb{Z} \setminus \{0\}\}$;
④ $\{\langle \langle \mathtt{rs}\langle x\rangle, \otimes\langle 2,5\rangle\rangle, \langle \mathtt{r}\langle x,y\rangle \mapsto 2, \mathtt{s}\langle y\rangle \mapsto 5, \cdots\rangle\rangle \mid x \in \mathbb{Z}, y \in \mathbb{N} \setminus \{x\}\}$;

The other two relevant rule queries give $\varnothing$, as $\{0\} \cap (\mathbb{N} \setminus \{0\}) = \varnothing$. We can read out the contributions of the ground answers contained in each non-ground rule answer: ① $\mathtt{rs}\langle 0\rangle$ gets $\{\!|\otimes\langle 4,6\rangle@1|\!\}$. ② Each $\mathtt{rs}\langle \mathbb{N} \setminus \{0\}\rangle$ gets $\{\!|\otimes\langle 3,5\rangle@1|\!\}$.[10] ③ Each $\mathtt{rs}\langle \mathbb{Z} \setminus \{0\}\rangle$ gets $\{\!|\otimes\langle 2,6\rangle@1|\!\}$. ④ For each $x \in \mathbb{Z}$, the item $\mathtt{rs}\langle x\rangle$ gets $\{\!|\otimes\langle 2,5\rangle@\infty|\!\}$—an *infinite* bag of aggregands (one for each $y \in \mathbb{N} \setminus \{x\}$).

Figure 1 shows how these answers are computed. Recall that our `REFINERULESUFFIX` strategy in §2 simply enumerated individual items that matched a rule's subgoals, in order to deduce aggregands for individual items that matched the rule's head. However, this is inadequate to compute the infinite example above. Figure 1 must enumerate several *sets* of related subgoal items, such as $\{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{N} \setminus \{0\}\}$. Furthermore, each set may produce contributions to multiple head items, and we must combine all contributions to each head item, with the results given in the figure caption. The next two sections explain how this is done in general.

3.2 Non-ground Rule Answers

A **non-ground rule query** for the rule r is a n_r-tuple of *sets of* items, $\vec{\tau}$. The corresponding set of pre-answers is the union of all pre-answers possible for different queries formed by element-wise choices from $\vec{\tau}$, or, more simply, just the refinement by each element in turn: $\theta_r^{\vec{\tau}} \stackrel{\text{def}}{=} \rho_r[\tau_1/\text{SG}.1.1]\cdots[\tau_{n_r}/\text{SG}.n_r.1]$. In general, calling `LOOKUP` on a subgoal will return a finite map with elements $\tau \mapsto f$, where $\tau \subseteq \mathcal{I}$ is a *set* of items and $f : \tau \to \mathcal{H}'$ is a valuation *function* on that set.[11] (Recall that in the ground setting of §2, `LOOKUP` merely returned a finite key-value map with individual items as keys.)

Suppose that $\tau_i \mapsto f_i$ is one of the elements returned by `LOOKUP` on the i-th subgoal, for each $i \in \mathbb{N}_1^{n_r}$. Then the rule query $\vec{\tau} = \langle \tau_1, \ldots, \tau_{n_r}\rangle$ has a **non-ground rule answer** of

$$\epsilon \;=\; \epsilon_{r,\vec{f}}^{\vec{\tau}} \stackrel{\text{def}}{=} \rho_r[\{t \mapsto f_1(t) \mid t \in \tau_1\}/\text{SG}.1]\cdots[\{t \mapsto f_{n_r}(t) \mid t \in \tau_{n_r}\}/\text{SG}.n_r] \;\subseteq\; \theta_r^{\vec{\tau}}.$$

This is essentially a set of ground rule answers (which, in principle, could be individually processed by `CONTRIBRULEANSWER`). It contributes to each $h \in \epsilon\downarrow_{\text{HEAD}}$ the values $\epsilon[\{h\}/\text{HEAD}]\downarrow_{\text{RES}}^{@}$. Since the value $f_i(t)$ may covary with t, our expression for ϵ takes care to refine subgoal i (that is, $\text{SG}.i$) by a set of *pairs* $\langle t, f_i(t)\rangle$ that captures this covariance. We will disallow this covariance in the next section.

3.2.1 Three Simplifying Assumptions

Recall from the introduction that `LOOKUP` may call `COMPUTE`. `COMPUTE` will issue rule queries against the rules of the program—via recursive `LOOKUP` calls to their subgoals—and then will combine the heads of the resuting rule answers ϵ (via §3.3 below) to obtain its own return value. We now observe that `LOOKUP` will return a *piecewise constant* map if the recursive `LOOKUP` calls do so and the resulting rule answers have sufficiently simple heads.

[10] Preserving the covariance of x between the head and body is vital: if projected separately, we would be at risk of claiming infinitely many contributions to each of infinitely many items!
[11] We use $\mathcal{H}'$ because some or all items in τ_i may have value `NULL`.

If the recursive `Lookup` calls could instead return *arbitrary* item valuation functions f_i, then it would presumably be hard to compute ϵ and hard to aggregate ϵ's heterogenous contributions to head items. (For example, imagine that $f_i = \{\mathsf{g}\langle x\rangle \mapsto 2 * x \mid x \in \mathbb{Z}\}$.) We therefore restrict to cases of the sort illustrated by Figure 1: each f_i is a constant function returning some v_i, so that the result of `Lookup` is piecewise constant with finitely many pieces. Despite being a special case, this is still a *strict generalization* of our ground reasoning story. Under this assumption, letting v_i be the value associated with all τ_i of a rule query, our set of rule answers has a much simpler definition: $\epsilon^{\vec{\tau}}_{r,\vec{v}} \stackrel{\text{def}}{=} \theta^{\vec{\tau}}_r[\{v_1\}/\text{SG}.1.2]\cdots[\{v_{n_r}\}/\text{SG}.n_r]$. There is no difficult preservation of covariance here: we are directly refining the subgoal value positions, for all corresponding keys. In this context, `Lookup` now has type $\Pi_{\kappa\subseteq\mathcal{H}} \bigcup_{K\in\text{fp}(\kappa\cap\mathcal{I})} \Pi_{\alpha\in K}(\{\text{NULL}\} \cup \bigcap_{t\in\alpha} \tau_t)$, where $\text{fp}(\beta)$ is the set of all *finite partitions* of the set β: $B \in \text{fp}(\beta)$ iff all of $\bigcup B = \beta$, $|B| < \infty$, and $\forall_{\beta_1,\beta_2}\beta_1 \cap \beta_2 = \varnothing$.

We next insist that the rule answers $\epsilon = \epsilon^{\vec{\tau}}_{r,\vec{v}}$ in the previous paragraph have simple heads, which we will be able to combine across rules (§3.3 below) to give a new piecewise function. Recall that for ground reasoning (§2), we required "range-restricted" rules so that each rule answer would have a single item as its head: that is, bringing $\rho_r\downarrow_{\text{SG}}$ to a singleton would bring the entire set to a singleton. Now that we are prepared to reason about sets of terms at once, this is no longer necessary and we can write `f(X,Y) ⊕= g(Y).` However, we will assume a different condition, **non-ground range restriction**: any rule answer $\epsilon = \epsilon^{\vec{\tau}}_{r,\vec{v}}$ that we compute must treat all its head items $\eta = \epsilon\downarrow_{\text{HEAD}}$ identically, contributing the same aggregands $\{\!|v@m|\!\}$ to each of them. (Formally, $\exists_{v\in\mathcal{H},m\in\mathbb{N}_\infty} \forall_{h\in\eta}\, \epsilon[\{h\}/\text{HEAD}]\downarrow^{@}_{\text{RES}} = \{\!|v@m|\!\}$.) This will allow §3.3 to easily determine which sets of head items receive which sets of aggregands.[12]

More trivially, we also need all head items in η to share an aggregator. To ensure this, we require that each rule r specify an aggregator consistent with all its possible heads: all items $\mathcal{I} \cap \rho_r\downarrow_{\text{HEAD}}$ must use this aggregator. This ensures that $\text{aggr}(\eta) \stackrel{\text{def}}{=} \text{selt}(\{\text{aggr}(h) \mid h \in \eta\})$ is well-defined when we use it to construct a query answer in Listing 1 below.

3.3 Combining Results

Recall that in §2, we imagined collecting ground rule answers by calling a procedure `contribRuleAnswer` on each one. We now generalize this to the non-ground case. If rules obey non-ground range restriction, then a rule query with non-empty answer ϵ can be read as an instruction: "contribute, to *each* $h \in \eta = \epsilon\downarrow_{\text{HEAD}}$, $m = \text{selt}(\{|\epsilon[\{h\}/\text{HEAD}]| \mid h \in \eta\})$ copies of $v = \text{selt}(\epsilon\downarrow_{\text{RES}})$." We further *assume the existence* of a procedure `ruleToInstr` which takes a non-empty ϵ and extracts η and $\{\!|v@m|\!\}$.

Given two rule answers, ϵ_1 and ϵ_2, with corresponding HEAD projections, η_i, and contributions, $\{\!|v_i@m_i|\!\}$, their combined contributions should be that $(\eta_1 \setminus \eta_2)$ gets only $\{\!|v_1@m_1|\!\}$, that $(\eta_2 \setminus \eta_1)$ gets only $\{\!|v_2@m_2|\!\}$, and that $(\eta_1 \cap \eta_2)$ gets contributions from both, i.e., $\{\!|v_1@m_1, v_2@m_2|\!\}$. (Recall Example 1.) Generalizing, if we

[12] However, in constrast to range restriction, non-ground range restriction is no longer a simple syntactic condition on the μDyna program. This is because it requires "any rule answers that we compute" to have a simple head, but those rule answers are not determined by the rule alone, but also by the particular rule queries. So actually our assumption is a joint constraint on the rules of the program and the behavior of `Lookup`. Static analysis of such properties of a logic program is the purview of *mode analysis* [13]; we leave all static analysis to future work.

```
1  def COMPUTE(κ ⊆ 𝓘)
2    c ← ∅  % initialize accumulator: no contributions to any item
3    foreach r ∈ Ξ do
4      REFINERULESUFFIX(ρ_r[κ/HEAD], 1)
5
6      def REFINERULESUFFIX(σ ⊆ ρ_r, i ∈ ℕ_1^{n_r+1}) ∈ ⟨⟩
7        if σ = ∅ then return                          % no contributions here, or
8        elif i = n_r + 1 then CONTRIBRULEANSWER(σ)    % some answers to process, or
9        else foreach (τ ↦ v) ∈ LOOKUP(σ↓_{SG.i.1}) do
10         REFINERULESUFFIX(σ[⟨τ, {v}⟩/SG.i], i + 1)  % refine and move to next subgoal
11
12     def CONTRIBRULEANSWER(ϵ ⊆ ρ_r) ∈ ⟨⟩  % extract answers and accumulate (§3.3)
13       c ← DISJOIN(c, η, β) where ⟨η, β⟩ = RULETOINSTR(ϵ)
14
15   return {η ↦ aggr(η)(β) | (η ↦ β) ∈ c}
16
17 LOOKUP ∈ Π_{κ⊆ℋ} ⋃_{K∈fp(κ∩𝓘)} Π_{α∈K}({NULL} ∪ ⋂_{t∈α} τ_t)  % answer a subgoal query
18 RULETOINSTR ∈ Π_{ϵ⊆ρ_r} ⟨{ϵ↓_HEAD}, ℘_+ U_∞^{-1}(ϵ↓_RES)⟩  % extract head and result bag from rule
```

Listing 1: Non-ground COMPUTE. LOOKUP(τ) is assumed to return an assignment of values (now inclusive of NULL) to each element of a *f*inite *p*artitioning (fp) of τ. CONTRIBRULEANSWER is prepared to deal with *multiple* answers at once, provided that RULETOINSTR can extract a head and result bag such that all results apply to each head. It uses DISJOIN to maintain the invariant that all elements of the accumulator dom(c) are *disjoint*.

have already accumulated some number of rule answers into a map c, upon the arrival of another set of heads η and bag of contributions $\beta = \{\!|v@m|\!\}$, one must construct a new entry in the map for any novel items in η and then split every existing entry κ into $\kappa \cap \eta$ and $\kappa \setminus \eta$ (we may safely omit the $\varnothing$ bin). Procedurally,

```
def DISJOIN(c, η, β)  % Add all of β to each h ∈ η across all of c
  return {(η ∖ ⋃(dom(c))) ↦ β | η ∉ ⋃(dom(c))}     % new bin for new terms;
    ∪{(κ ∖ η) ↦ τ | (κ ↦ τ) ∈ c, κ ∉ η}           % split existing bins: differences ...
    ∪{(κ ∩ η) ↦ β ⊎ τ | (κ ↦ τ) ∈ c, κ ∩ η ≠ ∅}   % ... and intersections
```

This procedure forms the core of our CONTRIBRULEANSWER procedure for non-ground backward reasoning.

3.4 An Algorithm

We now have all the pieces of Listing 1, our algorithm for non-ground backward reasoning.

3.5 Beyond Expression Trees

While the `rs` rule discussed above is a typical μDyna rule, it does not illustrate the full potential and complexity of rules. The algorithm presented in Listing 1 will, however, produce the correct answer for any μDyna program obeying the simplifying assumptions above. In particular, it handles cases where a *value* returned for one subgoal constrains the *key* of another subgoal, or vice-versa. Such cases are not allowed in some other weighted

logic programming languages in which rules primarily describe how to combine keys and values "come along for the ride," e.g., [2, 6].

User-Defined Operations on Values The μDyna translation of the `rs` rule is shown at the top of Figure 1. The third subgoal $\otimes\langle r, s\rangle$ takes the product of the values r, s returned by the first two subgoals. In typical weighted logic programming languages (including Datalog with aggregation), values can only be combined by built-in operations. In μDyna, however, $\otimes\langle r, s\rangle$ is simply a key built from values. Its own value is obtained via `Lookup` as for any other subgoal; so $\otimes$ could be user-defined.

Value Chaining Nested function evaluation of the form `g(f(X),Y)` is possible via a μDyna rule with subgoal structure $\{\cdots, \mathtt{f}\langle x\rangle \mapsto f, \mathtt{g}\langle f, y\rangle \mapsto g, \cdots \mid f, g, x, y\}$. Note that the value f returned by the first subgoal appears within the key $\mathtt{g}\langle f, y\rangle$ of the second subgoal. Alternatively, if we consider these subgoals in the other order, $\{\cdots, \mathtt{g}\langle f, y\rangle \mapsto g, \mathtt{f}\langle x\rangle \mapsto f, \cdots \mid f, g, x, y\}$, we have a subgoal `f(X)` whose *value* f is constrained by the *key* returned by lookup on the previous subgoal.

Value-Head Covariance In μDyna, subgoal values may influence the graph structure of the circuit. This happens when the head item's key is determined by a subgoal's value. For example, the rule $\{\langle\langle \mathtt{f}\langle a\rangle, 1\rangle, \langle \mathtt{a}\langle\rangle \mapsto a\rangle\rangle \mid a \in \mathcal{H}\}$ contributes the value 1 to an item determined by *the value of* $\mathtt{a}\langle\rangle$. A more complex example is $\{\langle\langle \mathtt{f}\langle a\rangle, 1\rangle, \langle \mathtt{a}\langle x\rangle \mapsto a\rangle\rangle \mid a, x \in \mathcal{H}\}$, whereby $\mathtt{f}\langle 3\rangle$ gains an aggregand of 1 for each value of x such that, according to `Lookup`, $\mathtt{a}\langle x\rangle$ has value 3.

3.6 Set Manipulations

A practical implementation of our algorithm requires a computational representation of sets of terms, such as regular tree automata, which are closed under the set operations we use and whose cardinality can be computed.

Unfortunately, to represent sets with covariance such as $\{\mathtt{r}\langle x, x\rangle \mid x\}$—or any μDyna rule with repeated variables—we require tree automata *with equality constraints* [3]. This extension destroys the nice computational properties, meaning that some sets cannot be constructed or cannot be counted in finite time.

However, there are useful settings where our algorithm can be executed without running into these problems.

Bounded-Depth Rules Recall that a μDyna rule is formally a set of nested tuples, typically representing all allowed instantiations of a template such as `rs(X) ⊕= r(X,Y) ⊗ s(Y)`. If the variables in this template are restricted to bounded-depth terms (e.g., depth 0 in the case of Datalog programs), then the rule consists of bounded-depth tuples. In this case, all sets that arise in our algorithm should be representable using *acyclic* tree automata with equality and disequality (where the disequalities arise from set difference). The operations on such sets are tractable.

Tree Automata With Bounded-Depth (Dis)Equalities More generally, for some programs, we can similarly guarantee that all sets that arise can be represented using tree automata with equality and disequality constraints that only mention nodes close to the root. For example, this is true for a rule like `rs(X) ⊕= r(X,Y) ⊗ s(Y)` if `X`, `Y` are allowed to range freely over $\mathcal{H}$ (rather than being typed variables that are restricted to terms of a complicated type such as "lists of all-equal elements," which requires deep equality checks).

3.7 Lowering Set-Theoretic Requirements

A particular sticking point in our work has been that tree automata with equality are not closed under set difference. A natural question was whether we could eliminate our need for this operation, which we have used here to construct the piecewise constant valuation function (§3.2.1) by partitioning its domain into *non-overlapping* sets (§3.3).

As we show in a companion paper also submitted to this venue [8], there indeed exists an alternative design. The idea is to encode the piecewise function as a series of partial functions with *overlapping* domains. These functions partially override one another, where functions with smaller domains "win" on their domain: that is, of all the partial functions that contain a given item name in their domain, there is one with strictly smallest domain, and it defines the corresponding value. The sole remnant of subtraction in this system is a need to compute *cardinality of the difference of two sets*; no explicit representation of the difference set is needed.

For sets represented by tree automata with equality constraints, cardinality is Turing-complete, and thus is not in general computable even for a single set. However, by developing appropriate heuristics that cover common cases, we can enlarge the class of programs on which we can guarantee terminating execution.

4 Related Work

4.1 Answer Subsumption

Many Prolog systems have been extended with mechanisms termed "Answer Subsumption" to provide aggregation of (projections of) answers from the solver [14]. These extensions allow Prolog programs to compute shortest paths, for example, even in the case of cyclic input graphs (without negative-weight cycles, as is usual). For example, in XSB (adapted from [15, Example 2]), the program

```
1 :-table p(_,_,lattice(min/3)).
2 p(X,Y,1) :- e(X,Y).    p(X,Y,D) :- p(X,Z,D1), p(Z,Y,D2), D is D1 + D2.
```

will, given a set of `e`/2 facts, assign each an additive cost of 1 and compute their min-weighted transitive closure.[13] However, only recently was a formal semantics proposed for these extensions [15]. This semantics works stratum-by-stratum within a Prolog program, computing the fixed point of a modified immediate-consequence operator which first computes the *Prolog* consequences and then *adds* any answers that are a consequence of the aggregation. Having computed that fixed point, the subsumed answers are all discarded, leaving only the order-maximal answers to be presented to the next stratum.

Answer subsumption thus has fundamentally different semantics than the privileged functional dependency of weighted logic languages like μDyna. First, the reliance on the Prolog fixed-point operator implies that non-idempotent aggregators, such as `sum`, are not admissible: the programs "`p(1). p(1).`" and "`p(1).`", which have identical answer

[13] In idiomatic μDyna, one would not use an indexing position to hold the weight and instead would rely on the privileged functional dependence between item and value. A weighted transitive closure would be written as the pair of rules $\{\langle\langle \mathtt{p}\langle x, y\rangle, 1\rangle, \langle\langle \mathtt{e}\langle x, y\rangle, \mathtt{true}\langle\rangle\rangle\rangle \mid \cdots\}$ and $\{\langle\langle \mathtt{p}\langle x, z\rangle, v\rangle, \langle\langle \mathtt{p}\langle x, y\rangle, l\rangle, \langle \mathtt{p}\langle y, z\rangle, r\rangle, \langle l + r, v\rangle\rangle\rangle \mid \cdots\}$. Of course, one of the selling points of μDyna is that it is *possible* to mix values into key positions, and the rule $\{\langle\langle \mathtt{p}\langle x, y, v\rangle, \mathtt{true}\langle\rangle\rangle, \langle\langle \mathtt{p}\langle x, y\rangle, v\rangle\rangle\rangle \mid \cdots\}$ will derive `p`/3 items equivalent to those of the Prolog program from the `p`/2 items derived above.

sets, must also have identical subsumed answers.[14] Weighted logic languages use bag semantics, rather than set semantics, to interpret multiple justifications of a particular aggregand for the same key. Second, the post-processing to remove answers means that justification of the surviving answers is less obvious. In a pure Prolog program (without the use of answer subsumption) or a weighted logic program, an item's value is always a fixed function of its parent items' values; with answer subsumption, one must refer back to subsumed answers for justification. Consider this program (also adapted from [15]):

```
1 :-table p(lattice(max/3)).
2 p(0).  p(1) :- p(0).
```

which has answer $\{\mathtt{p}\langle 1\rangle\}$. The justification for $\mathtt{p}\langle 1\rangle$ is that $\mathtt{p}\langle 0\rangle$ *was* true during computation but has been subsumed. The related idiomatic μDyna program, consisting of the rules $\{\langle\langle\mathtt{p}\langle\rangle, 0\rangle, \langle\rangle\rangle\}$ and $\{\langle\langle\mathtt{p}\langle\rangle, 1\rangle, \langle\langle\mathtt{p}\langle\rangle, 0\rangle\rangle\rangle\}$, with $\mathtt{p}\langle\rangle$ aggregated by `max`, makes the ill-founded recursion more explicit.[15] Third, we believe that the semantics of answer subsumption are incompletely specified if one allows variables in indexing argument positions, as it would be necessary to invoke a notion of set subtraction, à la DISJOIN, when a ground-indexed answer was to (partially!) subsume a non-ground-indexed answer. However, such operations are, at least, not readily apparent in [15] nor obviously available to typical WAM-based representations of non-ground terms.

4.2 Other Weighted Logic Languages

The weighted logic languages we are aware of work solely with ground answers, making no attempts at set-at-a-time reasoning. These languages include several Datalog derivatives with aggregation [2, 4, 10, 12] and the predecessor of our current effort, Dyna [6].

Conclusion

We have formalized a strategy for set-at-a-time reasoning within a weighted logic program solver. The approach can handle all finite circuits and many practical infinite ones, although it will be unable to proceed if the execution of a particular program produces overly complex valuation functions that are not piecewise constant (see §3.2.1) or whose constant regions have cardinality that is too hard to compute (see §3.6). Concurrent work [8] picks up where we have left off and investigates one avenue for representing these valuation functions more neatly.

Acknowledgements

We are deeply indebted to the editorial proficiencies and intellects of Rachael Bennett, Dr. Thomas Filardo, Dr. Nora Zorich, Dr. Scott Smith, Tim Vieira, and Matthew Francis-Landau, who all read numerous early drafts of this document and kindly contributed countless structural, prosodic, and grammatical suggestions to the text.

[14] Much of the complexity of [15] is to extend beyond selective aggregations (i.e., those $\oplus$ for which $a \oplus b \in \{a, b\}$); it is for this reason that their iteration to fixed point involves repeated addition of facts computed from lattice joins. See Example 3 therein.

[15] It is also possible to explicitly plumb "unless subsumed" subgoals into rules of a μDyna program to emulate answer-subsumption semantics. The resulting program is *cyclic*, so while items' values are still functions of their parents', the path by which a solver arrived (or not) at a particular fixed point is no longer readily apparent, intermediate values as lost as the subsumed $\mathtt{p}\langle 0\rangle$ answer. One could unroll the cycle with time-stamp indices, if the entire trace is essential to have on-hand in-program.

References

[1] Stefan Brass. "Range Restriction for General Formulas". In: *Proceedings of the 23rd Workshop on (Constraint) Logic Programming*. 2009.

[2] Sara Cohen, Werner Nutt, and Alexander Serebrenik. "Algorithms for Rewriting Aggregate Queries Using Views". In: *Proc. of ADBIS-DASFAA*. Springer-Verlag, 2000, pp. 65–78. DOI: 10.1007/3-540-44472-6_6.

[3] Hubert Comon et al. *Tree Automata Techniques and Applications*. Online publication. 2007. URL: http://tata.gforge.inria.fr/.

[4] M.P. Consens and A.O. Mendelzon. "Low-Complexity Aggregation in GraphLog and Datalog". In: *Theoretical Computer Science* 116.1 (1993), pp. 95–116.

[5] Jason Eisner and Nathaniel W. Filardo. "Dyna: Extending Datalog for modern AI". In: *Datalog Reloaded*. Ed. by Tim Furche et al. Vol. 6702. LNCS. Springer, 2011. DOI: 10.1007/978-3-642-24206-9_11.

[6] Jason Eisner, Eric Goldlust, and Noah A. Smith. "Compiling Comp. Ling.: Weighted Dynamic Programming and the Dyna Language". In: *Proc. of HLT-EMNLP*. Association for Computational Linguistics, 2005, pp. 281–290.

[7] Nathaniel Wesley Filardo and Jason Eisner. "A Flexible Solver for Finite Arithmetic Circuits". In: *Technical Communications of the 28th ICLP*. Ed. by Agostino Dovier and Vítor Santos Costa. Vol. 17. Leibniz International Proceedings in Informatics (LIPIcs). 2012, pp. 425–438.

[8] Nathaniel Wesley Filardo and Jason Eisner. "Default Reasoning in Weighted Logic Programs". In: In submission to ICLP'17; see http://www.cs.jhu.edu/~nwf/ilcp17-2.1.pdf. 2017.

[9] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *Proc. of PLDI*. PLDI '93. ACM, 1993, pp. 237–247. DOI: 10.1145/155090.155113.

[10] Sergio Greco. "Dynamic Programming in Datalog with Aggregates". In: *IEEE Transactions on Knowledge and Data Engineering* 11.2 (1999), pp. 265–283. DOI: 10.1109/69.761663.

[11] Robert Kowalski. *Predicate Logic as Programming Language*. Memo 70. Department of Artificial Intelligence, Edinburgh University, 1974.

[12] Abhijeet Mohapatra and Michael Genesereth. *Aggregation in Datalog under set semantics*. Tech. rep. 2012.

[13] David Overton. "Precise and Expressive Mode Systems for Typed Logic Programming Languages". PhD thesis. University of Melbourne, 2003.

[14] Terrance Swift and David S. Warren. "Tabling with Answer Subsumption: Implementation, Applications and Performance". In: *Proceedings of the 12th European Conference on Logics in Artificial Intelligence*. JELIA'10. Springer-Verlag, 2010, pp. 300–312.

[15] Alexander Vandenbroucke et al. "Tabling with Sound Answer Subsumption". In: *Theory and Practice of Logic Programming* 16.5-6 (2016), pp. 933–949. DOI: 10.1017/S147106841600048X.

[16] Neng-Fa Zhou and Taisuke Sato. "Toward a High-Performance System for Symbolic and Statistical Modeling". In: *Proc. of the IJCAI Workshop on Learning Statistical Models from Relational Data*. 2003, pp. 153–159.

Appendix A A Richer Query Interface

As an optimization, one could pass values, as well as keys, to `Lookup`, to take advantage of its access to cached values. Suppose, for example, that the value at $\text{SG}.i.2$ has otherwise been refined prior to the invocation of $\texttt{refineRuleSuffix}(\sigma, i)$, due to covariances in the rule or perhaps just by the rule itself (yielding, say, a subgoal with projection $\langle \mathtt{a}\langle \ldots \rangle, \{1\} \rangle$). Absent this optimization, the algorithms given will rely on refinement to exclude keys whose values mismatch σ. In practice, this additional information passed to `Lookup` additionally allows for certain kinds of "inverse" modes where, e.g., the value and one argument to addition are ground and the other argument is only partially constrained.

In both code listings, we would replace the calls to $\texttt{Lookup}(\sigma{\downarrow_{\text{SG}.i.1}})$ with $\texttt{Lookup}(\sigma{\downarrow_{\text{SG}.i}})$ (note the shorter projection path). In the ground reasoning of §2, `Lookup` would have the type $\Pi_{\Sigma_{k \in \kappa} \tau_k \subseteq \langle \mathcal{H}, \mathcal{H} \rangle} \bigcup_{\alpha \in \wp_{\text{fin}}(\kappa \cap \mathcal{I})} \Pi_{k \in \alpha} \tau_k$. In the non-ground reasoning of Listing 1, it would be $\Pi_{\Sigma_{k \in \kappa} \tau_k \subseteq \langle \mathcal{H}, \mathcal{H} \rangle} \bigcup_{K \in \text{fp}(\kappa \cap \mathcal{I})} \Pi_{\alpha \in K} (\{\text{NULL}\} \cup \bigcap_{t \in \alpha} \tau_t)$.