Dyna 2: Towards a General Weighted Logic Language

by

Nathaniel Wesley Filardo

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

October, 2017

Abstract

We investigate the design of an expressive, purely-declarative, weighted logic programming language, Dyna. Dyna is a decade-plus effort in pushing the boundaries of declarative programming and "executable mathematics;" it instantiates an unusual point in the design space, as it is both Turing-complete (unlike Datalog) and devoid of a specified execution order (unlike Prolog). That is, it is designed to be, at once, both highly expressive and rich in opportunities for automated optimization. This thesis contains two major thrusts. We first consider both the denotational (§2.1.2 and §3.1.4) and operational aspects (§2.2 to §2.5, §3.2 to §3.6, and §4) of Dyna. In particular, for operational semantics, we introduce (§2.2) and extend (through §2.5) our EarthBound solver for finite circuits; §3 considers the generalization to logic programs proper. We then turn our attention to the *static* analysis of this language, considering mechanisms for reasoning both about abstract notions of well-formedness of programs (§5.2) as well as more mundane concerns of realizability of programs in actual computation (§5.3 and §5.4). Along the way we endeavour to place our work in the context of the larger field of logic programming languages and present our current thoughts on future avenues of exploration.

Primary Reader: Dr. Jason Eisner
Secondary Readers: Dr. Scott Smith, Dr. David Warren (Emeritus, SUNY Stony Brook)

Acknowledgments

I am deeply indebted to the intellects and editorial proficiencies of Rachael Bennett, Dr. Thomas Filardo, Dr. Nora Zorich, Dr. Jason Eisner, Dr. Scott Smith, Dr. David S. Warren, Tim Vieira, Matthew Francis-Landau, and Adam Teichert, who all read numerous drafts of this document and/or its precursors and kindly contributed countless mathematical, structural, prosodic, and grammatical suggestions.

This thesis would not have been nearly so easily typeset without the TeX family of tools (including AMS-LaTeX, BibTeX, and LuaLaTeX) and their vibrant communities. No fewer than 55 CTAN packages are used within this thesis's source; Ti*k*Z, cleveref, hyperref, listings, and xcolor are manifestly apparent throughout; enumitem, floatrow, footmisc, and microtype deserve special attention as well. The ε-TeX and LaTeX3 efforts, as well as `latexmk`, significantly improved the *software development* aspect of authoring this thesis. And, of course, dozens of `http://tex.stackexchange.com` and `http://latex-community.org` contributors provided ready answers to difficulties encountered by the author.

Dedication

I would like to dedicate this thesis to everyone who has supported me over the years it has taken to get to this point. The support of my family, partners, and friends has been absolutely essential and I hope that the completion of this document is, in some small way, a reward for their efforts.

More broadly, I would like to thank the scientific community as a whole, and their many legacy futures upon which I have built not just this work but, indeed, my knowledge of my field. May the future in which we arrive together be the brightest we can envision.

Contents

List of Tables

List of Figures

List of Listings

Prior Publications

This thesis is, as might be expected, based on work that has already been published:

- §1, especially §1.1, is largely derived from our 2011 position paper on the Dyna project, Eisner and Filardo [50].
- §1.3 greatly extends material seen in Filardo and Eisner [63] and Filardo and Eisner [61].
- §2, up through §2.2, and part of §2.6 is based on our publication at ICLP in 2012, Filardo and Eisner [60]. §2.4.3 and §2.5 are expansions of future work in that paper.
- §3 is an expanded merger of our submissions to ICLP 2017, Filardo and Eisner [63] and Filardo and Eisner [61]. which were ultimately released as technical reports.
- §4.2 contains material taken from the author's self-published notes on computational automata (the "Automata Zoo" [57]).
- §6.2 redesigns material from the appendix of Eisner and Filardo [50].

Chapter 1

Introduction

> "Begin at the beginning," the King said, very gravely,
> "and go on till you come to the end: then stop."
>
> Lewis Carroll, *Alice in Wonderland*

1.1 Motivation and Background

The Dyna programming language was born more than a decade ago in an effort to simplify the lives of researchers working in the area of statistical artificial intelligence and natural language processing [52, 51]. The effort having been found worthwhile, work began on a second edition of the language, which is designed to overcome many of the expressive limitations and implementation artifacts of the first [50]. This thesis is the culmination of a decade of design work and presents core formalisms for efficient execution (§2 to §4) and static analysis (§4.4 and §5) of these second-edition Dyna programs.

Fundamentally, Dyna is a language for specifying a **deductive database** [116], a mutable data structure which does more than just store data for later retrieval: additionally, it **derive**s conclusions from its inputs. Such a structure is given a stream of **updates** (which change the stored data) and **queries** (which are requests to retrieve stored data and/or the conclusions drawn therefrom). The data structure responds by looking up or computing the answers to queries. Queries may be for individual items of information or for sets of related items (e.g., collections of rows within a table, in a traditional, relational database).

Dyna is a *pure* and *declarative* specification language. Programs written in Dyna have no ability to initiate interactions with their environment. Permissible answers to a query depend solely on the updates made prior to that query (and not, e.g., sensors of the environment), and updates affect only the answers to subsequent queries (and not, e.g., actuators manipulating real-world state). Dyna programs do not specify *how* the queries or updates are to be carried out. That is, a Dyna program does not explicitly specify things like choices of data structures for particular pieces of data, loop orders for traversals, or even the order in which computations are to take place. A Dyna program is, thus, *inert* unless combined with a **solver**, a computational mechanism for carrying out the computations described. Strategies ranging from the laziest ("store the update stream and scan it when queried") to the most eager ("recompute all derived data upon every update") are all possible. This partition between specification and execution is well-studied and even captured as a slogan of the logic programming community (of which Dyna is a small part): "Algorithm = Logic + Control" [109].[1] Over the years, many such language-and-solver pairs have emerged, e.g., [6, 117, 192, 124, 190] to name but a few. When so combined, the declarative program and solver form a **reactive program**, one that "maintain[s] a continuous interaction with [its] environment... at a speed which is determined by the environment" [19].[2]

Example 1: As a simple example, consider a priority queue that denotationally contains a finite set of (key, value) items, where the values are interpreted as priorities. An update will add or remove the item with a particular key, or change its value. A supported query is to ask for the current maximum-value item (if any). The fact that this item is the correct answer was not explicitly inserted by any update, but is rather the result of an eager or lazy computation that supports the query; that is, "the maximum-value item" is, itself, a (derived) item. Other queries might also be supported, e.g., for the collection of items whose keys match some pattern.

This priority queue is, presumably, pure, in that it does not take it upon itself to interact with the environment. It makes no use of a clock, random numbers, the state of the weather, or whatever missiles might be attached to the computer upon which it finds itself running. While a particular implementation of such a priority queue may be expressed in a procedural language, the relationship between items inserted (and not yet deleted) and the "maximum-value item" item, being a mathematical function, may be given declaratively.[3] ◊

Solving a Dyna program is nontrivial, both theoretically and computationally. Beyond computation of derived items from input items, derived items may depend upon other derived items. This dependence is permitted, in Dyna and other sufficiently expressive languages, to be *cyclic*, which makes answers not necessarily well-founded and so there may be *zero* answers to a program, exactly one, or *more than one* answer. Dyna does not, unlike many of its brethren, seek to prohibit the "zero" or "more than one" case. Moreover, the language specification does not concern itself with the number of answers to the program at all; the solvers simply attempt to find an answer, should one exist. This loss of well-

[1] As with all slogans, there is some fuzziness, as one could imagine embedding a simulation of a control algorithm within a sufficiently expressive logic. In some cases, e.g., Prolog [99], the control is standard and implicit, imposed by fiat by the compiler; it is, nevertheless, not the *focus* of the program's syntax, which remains firmly rooted in the logical description of data interdependence. Other systems, such as Mercury [124], have fixed aspects of their control but also attempt to derive others from *static analysis* of the program, as we shall in §5.3.

[2] Admittedly, our solver may take time—possibly even vast stretches of time—to respond to a query (or to be ready for the next update) from its environment. Nevertheless, the system is driven by *events* from the outside world; on an arbitrarily fast computer, the solver's work would always finish, and the computer idle, before the next external stimulus.

[3] There are some algorithms which make use of random numbers *internally* and yet retain their "pure interface." Neglecting such real-world concerns as the *number* of CPU instructions or wall-clock time (or power) taken to perform an operation, such algorithms are equivalent to corresponding algorithms that do not avail themselves of randomness. We beg the reader's indulgence as we sketch the notion of purity, which is, admittedly, a subject of some contention within the field.

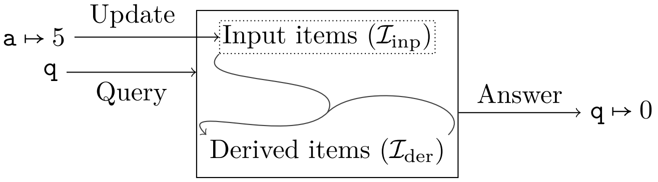

Update
a ↦ 5
q
Query
Input items ($\mathcal{I}_{\text{inp}}$)
Derived items ($\mathcal{I}_{\text{der}}$)
Answer
q ↦ 0

Figure 1.1: The map-data-structure view of a weighted logic program solver. The solver algorithm runs the loop within the box, which must compute the derived items requisite to answer the queries. A schematic representation of possible communication with the driver program (i.e., update, query, and answer) is shown; updates change the value of input items, while queries can look up the values associated with both input and derived items.

foundedness is one reason that solvers for these expressive languages may not terminate. More practical concerns are pervasive as well, even in absence of complex dependency structure. The solver may gain an asymptotic increase in efficiency if it *caches* derived items so that computational effort can be reused. However, computers have finite memory and there are costs associated with cached values that must be modified in response to updates, so the solver must juggle competing demands using some *policy*. Further, the solver must *schedule* internal computations, notably the effects of updates and the order in which (recursive) queries are to be resolved. In order to remain maximally flexible, the solver strategies we have developed and which are presented in this thesis are, to the extent possible, correct *regardless of policies and schedules employed by the solver*. While a lofty goal, such effort was motivated by an overwhelming desire to bring *machine learning* to bear within the solver, so that the solver can *adapt* its policies and schedules in a workload-responsive way. (While such machine learning concerns are out of scope for this thesis, they are very eagerly being pursued by other members of the Dyna project.)

Even so, a Dyna program within its solver is still a passive object. It will do nothing (other than initialize itself, perhaps) absent some external stimulus. We refer to the environment in which the solver finds itself as the **driver program**. It is this driver that is responsible for interaction with the procedural, chronologically-measured environment and the users therein; the driver issues updates and queries to the solver and uses the resulting answers for its own ends. A schematic representation is shown in figure 1.1.

1.1.1 Logic Programming

Dyna is, additionally, a *logic programming* language. That is, it uses the vocabulary of logical clauses to represent possibly infinite sets of theorems [83], which describe the relationships between input and derived data. The most well-known logic programming language is undoubtedly Prolog [99] and its numerous implementations, e.g., [29, 77, 192, 193]. Prolog has formed the basis of many other logic languages [1, 124, 148], the most well-known of which is a subset of Prolog termed "Datalog," which, in turn, has numerous extensions [153, 195, 82, 36, 32, 13] and implementations [190, 117] of its own.

By way of example, the Prolog rule "`a :- b, c`" indicates that the fact `a` is provable if the facts `b` and `c` are. (The syntax "`:-`" is meant to be mnemonic for ←.) More generally, rules can contain variables (capitalized identifiers), as in "`rs(X) :- r(X,Y),` `s(Y)`." This rule says that, for any x, `rs(x)` is provable if there exists a y such that there are proofs of both `r(x,y)` and `s(y)`.[4] The pair of rules "`a :- b. a :- c`" provide two possible ways of proving `a`: from either `b` or `c`. The (single) identifier to the left of the "arrow" "`:-`" is the **head** of the rule while one finds comma-separated **subgoals** to the right, which collectively form the **body**. The curious reader is invited to peek ahead to §1.4 and the references given therein.

Dyna, being, in fact, a *weighted* logic language, extends this formalism to general *expressions*, rather than logic clauses. Our rules look like "`rs(X) ⊕= r(X,Y) ⊗ s(Y)`," which indicates that, for all `x`, each `rs(x)` is associated with the pairwise ⊗ of *all* `r(x,y)` and `s(y)` for all `y` such that these items have values. The associative-commutative operator ⊕ is used to combine these multiple results, so that, for example, `rs(1)` would be assigned the value $(\texttt{r(1,1)} \otimes \texttt{s(1)}) \oplus (\texttt{r(1,2)} \otimes \texttt{s(2)}) \oplus \cdots$.[5]

More generally, one can give *recursive* definitions within the logic program. A simple example is a function which measures the length of a list. Herein, by "list," we often mean the LISP-style list structure, which is built from an "end" marker, pronounced `nil`, and a structure to hold an element and "the rest" of the list, pronounced `cons`.[6] A particular instance of `cons` is written `cons(`e`,`l`)`, where e and l represent the element and the rest of the list, respectively. There are two cases to consider: `length(nil) = 0` defines an empty list to have length 0, and `length(cons(H,T)) = 1 + length(T)` defines the length of a non-empty list to be one more than the length of its tail. More interesting recursive examples include *weighted transitive closure* (e.g., minimum cost paths within a graph); we defer treatment to "AND/OR Graphs" (in §2.1.2.1).

We have found these weighted logic rules to be a very powerful formalism for many modern AI tasks, as shown in our position paper [50]. That paper shows off the rich surface syntax of Dyna, which is designed with an eye towards making as many things as concise as possible. In this thesis, we will use simpler, but more verbose, representations and will focus on the language itself, rather than its use cases.

1.1.2 Reactive Programming

The execution model developed for the first edition of Dyna naturally incorporated a notion of *reactivity*: the inputs to the program could be changed on the fly and the outputs would update themselves correspondingly. This reactivity also goes by the names "view maintenance" and "stream processing" [131]. While this kind of behavior is found in some Datalog systems, it tends not to be native to Prolog solvers, though there is support for programmer-directed reactivity in XSB Prolog [170]. Dyna's reactivity is built-in, in the same way as in languages supporting "incremental re-computation" or "adaptivity" such

[4] For readers familiar with databases, this is a one-column view of a join of the two-column `r` table's second column with the one-column `s` table's only column.

[5] Assuming no other rules have `rs(X)` heads, anyway. If there were such, their contributions would have to be included as well, and we would require that they also used the same ⊕ to combine contributions. In the expression just given, we have performed the usual sleight of hand, using a name, such as `s(1)`, to refer to its value. Because we will manipulate both names and values in this thesis, we will use a formalism that makes it clear which role we mean.

[6] The curious reader is directed to the definition within Common LISP; see Steele [168, §2.4].

as that of Acar and Ley-Wild [4] or as in programs written using a “Functional Reactive Programming” library [54, 37].

1.2 Contrasting Dyna to Existing Logic Languages

As just said, Dyna is another entry in the family of “Prolog-inspired logic languages” but represents its own distinct point in the space. The design of Dyna borrows heavily from standard Prolog, Prolog extensions, and Datalog derivatives with aggregation, but Dyna also contains elements of our own design. We now highlight some of the key differences of the *semantic* features of Dyna from those of earlier systems.[7]

Weights Prolog programs are, when viewed as maps, *unweighted*: if an item is asserted to have a value, that value is from a singleton set. (The element of that set is often pronounced “true” or, perhaps more properly, “provable”.) As such, duplicate answers are permitted and are inconsequential from a logical semantics perspective. Prolog is thus sometimes said to be an **all-proofs** system, where each answer within a stream represents a different, successful proof of a subset of the query. Recall that computation of derived items is recursive; in Prolog, these recursive calls need not *finish* enumerating their answer streams before a proof of the original query is reported; the solver thus interleaves its two behaviors of ① exploring the space of all possible proofs of the queried items and ② emitting found answers. There may be proofs of overlapping subsets of the query or even multiple proofs of the same subset, but the stream is **monotonic**: once an item is shown to be true, subsequent discoveries will not change this fact. The duplicate work they represent is typically viewed as less than the effort it would take to remove them (and, when Prolog is not being used “extra-logically,” it may be desirable that the duplicate entries remain).

Once weights are more elaborate, in order to preserve the *functional dependency* in which each item has exactly one weight associated with it, such overlap must be eliminated by *aggregation*. Thus, the answer stream from a weighted logic program must not, semantically, have duplicates, making it an **all-answers** stream. This functional dependence makes weights **non-monotonic**:[8] subsequent values discovered for an item *combine* with earlier values, potentially *invalidating* observations made of those values. As a result, the ability of the solver to interleave answers and search is restricted; in the most general setting, no such interleaving will be possible and all recursive queries will have to termi-

[7] While we have designed the surface syntax of Dyna 2, the second edition of the Dyna language [50], to be a suitable language for expressing programs for deductive databases, when this thesis works with logic programs, it will do so almost exclusively in a simpler (though more verbose) core calculus, μDyna (which will be introduced in §3.1). μDyna is suitable as an early intermediate representation within a compiler; in particular, it has dispensed with almost all syntactic details while retaining the salient semantic features of the language.

[8] The word “monotonic” is unfortunately overloaded in meaning in a weighted logic system such as ours. There are “monotone functions”: those which map an ordering assertion of input values into an ordering assertion of the corresponding output values (see Functions and Maps, page 10). There is also “monotonic reasoning,” in which proven items are irrefutable: once a given item has been assigned a value, subsequent reasoning will never, in the absence of updates from without the pure program, retract or modify that value. Confusingly, the use of monotone aggregation functions does not ensure monotonic reasoning.

nate before any answers can be reported. Prior weighted systems have wrestled with this inherent non-monotonicity and come to differing conclusions, considered below.

Item Names and Values Dyna's domain of discourse is shared with Prolog: tree-shaped immutable objects. The name of its items *and the values thereof* are all chosen from a so-called "Herbrand universe" [92], which we explain, more formally, later. Datalog restricts its domain to *flat* terms, i.e., named tuples over some universe of atoms (e.g., "`s(z)`," but not "`s(s(z))`").

Program Non-Stratification Many Datalogs with aggregations (e.g., those of Ramakrishnan et al. [153], Zukowski and Freitag [195], Greco [82], and Consens and Mendelzon [36]) have continued, like Datalog, to require a *stratified* program, wherein the value of an item may never cyclically depend upon itself, even through other items' values. The Prolog extension to support negation (developed by Clark [30]) requires stratification (at least) of items whose negation is taken. Dyna 2 *does not* require program stratification, though as mentioned before, this means that the solver may never terminate or that there may be zero or more than one answers to a given program.

Weight Non-Isolation Other weighted systems, e.g., that of Cohen, Nutt, and Serebrenik [32] and the predecessor of the current effort, Dyna 1 [51], have relied on a strong partition between the logical provability of items (i.e., whether or not an item *has* a weight assigned) and the precise *value* of those items' weights. Broadly speaking, these systems do not allow the *values* of items to influence the data flow within (the same stratum of) the program. While many useful scenarios, e.g., weighted transitive closure such as the shortest path in a cyclic graph, are amenable to this restriction, it imposes an unnecessary distinction between weighted items and functions on weights. Dyna 2 does not enforce isolation of item names and weights.

Answer Stream Elements Prolog execution systems tend to be based on SLD resolution [110] (see §1.4 for more detailed discussion) which admits reasoning about *sets* of items at once: if a program contains the rule "`p(1,X,Y)`," the query "`p(A,2,B)`" will return "`p(1,2,B)`" as an answer, coding for the provability of the intersection of the rule's assertion and the query. The answer stream is thus a stream of *sets of* items.

While some Datalog algorithms do act on sets of items at once (e.g., DRed—Delete and REDerive—[85]), the definition of Datalog ensures that all answer sets are finite, permitting answer streams to be streams of *individual* items and their attendant values.

Dyna 2 adds a set-theoretic *runtime type system* to permit a richer vocabulary of sets; this thesis details at length the requirements of such a system, starting in §3.3. See §4 for more discussion of the computational representation of sets of terms.

The task at hand, then, is to somehow get the best of all possible worlds, to design a single framework that can handle set-at-a-time reasoning, in the presence of nontrivial weights and non-stratified programs and with potentially complicated interplay between weights and item provability.

1.3 Mathematical Background and Notation

This work builds upon a number of different threads from different areas of mathematics and computer science. Wherever possible, notation is taken from the most relevant field. In this section, we introduce all of our notation not specific to Dyna itself. Readers are invited to skim this section and refer back to it via the index (at the end of the document, on page 191), should the need arise.

Quantifiers Throughout this document, we use consistent notation for quantification: the quantifiers will be *subscripted* with their bound variables (and their domain constraints), there will be no delimiter between quantifiers and expressions, and logical quantifiers scope as far to the right as possible. That is, rather than "$\forall a \in \alpha \,.\, \varphi(a)$," we write "$\forall_{a\in\alpha}\, \varphi(a)$," in keeping with other operators like $\bigcup_{a\in\alpha} \varphi(a)$. (However, these other operators scope only to the end of an expression, delimited by an operation such as $=$ or $\subseteq$.) Domains of quantified variables will often be left implicit when clear from context (e.g., $\forall_i\, \varphi(x_i)$ takes i such that x_i is in scope).

Sets We adopt set- and bag- theoretic semantics throughout this thesis, relying on a well-typed underlying theory. Sets are manipulated by the typical operators (e.g., $\{\ldots\}$, $\cup$, $\cap$, $\in$, $\subseteq$, $\smallsetminus$). $\wp$ 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 $\mathbb{N}_\infty \overset{\text{def}}{=} \mathbb{N} \cup \{\infty\}$.[9] The *partial* operator $\operatorname{selt}(\sigma)$ projects a singleton set to its element: $\operatorname{selt}(\{s\}) \overset{\text{def}}{=} s$. We define the shorthands $\bigcup \sigma \overset{\text{def}}{=} \bigcup_{s\in\sigma} s$ and $\mathbb{N}_1^n \overset{\text{def}}{=} \{1, 2, \ldots, n\} \subset \mathbb{N}$.

Bags We will use $\lbag \ldots \rbag$ for bag literals. $\lbag s@m \rbag$ 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: $\lbag s \rbag = \lbag s@1 \rbag$; @ is *part of the bag notation* (i.e., a variable s cannot stand for "$t@m$"), so this abbreviation is unambiguous. $\lbag s_1@m_1, s_2@m_2 \rbag$, with $s_1 \neq s_2$, denotes a bag holding exactly m_1 copies of s_1 and m_2 copies of s_2. Bag multiplicities *add*, so $\lbag s@m_1, s@m_2 \rbag = \lbag s@(m_1 + m_2) \rbag$. As usual, comprehension notation may be employed, quantifying over both elements and multiplicities, so $\lbag s@m \mid s \in \{\mathtt{a}, \mathtt{b}\}, m \in \{1, 2\} \rbag$ is a set holding *three* copies each of a and b. Bag membership is indicated as $s \bagmember_{\geq m} \beta$ if s occurs in β at least m times, or as $s \bagmember_{=m} \beta$ when when β contains *exactly* m copies of s. $s \bagmember \beta \overset{\text{def}}{=} s \bagmember_{\geq 1} \beta$. Bag cardinality is the sum of all membership multiplicities. The traditional symbols of set-theory with a plus sign superimposed will be used for bag operations: $\uplus$, $\capplus$, $\subseteqplus$, etc., though we overload $\varnothing$ for the empty bag as well. $\wp_+\beta$ denotes the *set* of all sub-bags of bag β. The **underlying set** of a bag β is $\mathfrak{U}\beta \overset{\text{def}}{=} \{b \mid b \bagmember \beta\}$. In the other direction, $\bar{\mathfrak{U}}_m\sigma$ is the bag whose elements are from the set σ, all with multiplicity m; if not specified, $m = 1$.

Tuples We assume our theory contains n-ary **tuples**, denoted $\langle t_i \rangle_{i\in\mathbb{N}_1^n} \overset{\text{def}}{=} \langle t_1, \ldots, t_n \rangle$; we use $\vec{t}$ when n is clear from context. A **pair** is a tuple of length 2. $+\!\!\!+$ is the associative

[9] That is, naturals and only one infinity. The $\langle \mathbb{N}, +, 0, \cdot, 1 \rangle$ commutative semiring (see Algebraic Structures, page 10) extends as might be expected: $m + \infty \overset{\text{def}}{=} \infty$ for all $m \in \mathbb{N}_\infty$. $0 \cdot \infty \overset{\text{def}}{=} 0$, and $m \cdot \infty \overset{\text{def}}{=} \infty$ for $m \neq 0$.

tuple concatenation operator. The length of a tuple is denoted $\mathrm{tlen}(\langle t_1,\ldots,t_n\rangle) \overset{\text{def}}{=} n$; $\langle\rangle$ is the tuple of length 0. We define a **projection** operator, denoted $\cdot\downharpoonleft_\cdot$, to access components of (nested) tuples. $\vec{t}\downharpoonleft_k$, for $k \in \mathbb{N}_1^n$, means simply the k-th component of $\vec{t}$, i.e., t_k. More generally, we write $\vec{t}\downharpoonleft_{k_1.k_2.k_3}$ 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. We may extend a path by prefix or postfix concatenation, e.g., if $\pi = 2.3$ then $1.\pi.4 = 1.2.3.4$. The set of **positions** within a nested structure of tuples t is the set of paths π for which $t\downharpoonleft_\pi$ is defined. Formally, we should write, e.g., $\pi = \langle k_1,k_2,k_3\rangle$ for paths and $\pi + \pi'$ for concatenation, but the dot notation is standard and more easily read (save a very minor risk of occasional conflation with real numbers). Projection is, formally, defined inductively: $t\downharpoonleft_{\langle\rangle} \overset{\text{def}}{=} t$ (even for non-tuple t) and $\langle\tau_1,\ldots,\tau_n\rangle\downharpoonleft_{\langle k\rangle + \pi} \overset{\text{def}}{=} \tau_k\downharpoonleft_\pi$.

Pairs play a wide variety of roles in the formalisms in this thesis. It will be convenient to have a corresponding variety of syntaxes for pairs, beyond just $\langle a,b\rangle$. We will, therefore, often use *infix* pair constructors, namely $\mapsto$ (see Functions and Maps, below), $\hookleftarrow$, and $\Leftarrow$, in different contexts as mnemonics for the role a given pair is playing. That is, while $(a \mapsto b) = (a \hookleftarrow b) = (a \Leftarrow b) \overset{\text{def}}{=} \langle a,b\rangle$, the use of the different symbols should prove easier to read, especially when nested inside other tuples, parentheses, and/or set braces.

Sets and Bags of Tuples It will be convenient to have several shorthands available to build up and manipulate sets and bags of tuples without having to resort to comprehension notation every time.

Projection is extended to allow its first argument to be a set or bag of tuples: $\sigma\downharpoonleft_\pi \overset{\text{def}}{=} \{s\downharpoonleft_\pi \mid s \in \sigma\}$ and $\sigma\downharpoonleft_\pi^+ \overset{\text{def}}{=} \wr(s\downharpoonleft_\pi)@m \mid s \in_{=m} \sigma\int$. 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: $\sigma\downharpoonleft_\pi^@ \overset{\text{def}}{=} \wr s\downharpoonleft_\pi \mid s \in \sigma\int$.[10] In addition to projection from sets, we will make extremely heavy use of **refinement**, $\sigma[\tau/\pi] \overset{\text{def}}{=} \{s \in \sigma \mid s\downharpoonleft_\pi \in \tau\}$, which is a generalization of traditional substitution in systems based on structures over variables, as opposed to our more set-centric approach (see §4 for more discussion). After introducing some more vocabulary, we give some lemmas about projection and refinement in Lemmas for Projection and Refinement, below.

As we will often have sets described by tuples of elements sampled from other sets, we introduce a product-forming tuple operator for n-ary Cartesian products:[11]

$$\langle\sigma_1,\ldots,\sigma_n\rangle \overset{\text{def}}{=} \{\langle s_1,\ldots,s_n\rangle \mid \forall_i\, s_i \in \sigma_i\}.$$

When all $\{\sigma_i \mid i\}$ are equal, we may write σ^n.

Dependent sums, a particular encoding of disjoint unions, are written $\Sigma_{s\in\sigma}\, Y_s \overset{\text{def}}{=}$

[10] Our use of $\mathbb{N}_\infty$ for multiplicities implies some small loss of information: $\{\langle 2,r\rangle \mid r \in \mathbb{R}\}\downharpoonleft_1^@ = \wr 2@\infty\int = \{\langle 2,n\rangle \mid n \in \mathbb{N}\}\downharpoonleft_1^@$ as both of these sets are infinite (though not isomorphic to each other). However, later, we will make heavy use of a *cardinality of set subtraction* operator, and we expect to be able to recognize that $|\mathbb{R} \setminus \mathbb{N}| = \infty$ while $|\mathbb{N} \setminus \mathbb{N}| = 0$ and $|(\{\mathrm{a}\} \cup \mathbb{N}) \setminus \mathbb{N}| = 1$.

[11] Cartesian products of sets are also occasionally called "cross products," but we avoid the term due to risk of confusion with the vector space operation. In databases, the concept is also termed "cross join."

$\{\langle s,t\rangle \mid s \in \sigma, t \in Y_s\}$, where Y is a σ-indexed collection of sets.[12] We may abbreviate summation as ΣY when the domain is clear. *Every* set of pairs τ is a dependent sum of *some* indexed collection: $\tau = \Sigma_{t\in\tau\downarrow_1}(\tau[\{t\}/1]\downarrow_2)$.[13]

Functions and Maps The set of total functions from set σ to set τ is denoted $\sigma \to \tau$. σ is said to be the **domain**, and τ the **codomain**. A function f is **total** (and in $\sigma \to \tau$) iff $\forall_{s\in\sigma}\ \exists!_{t\in\tau}\ f(s) = t$. (While often mistaken for a typographic error, **iff** is short for "if and only if.") The symbol "$\to$" right-associates (as is typical for functional programming), meaning that $\alpha \to \beta \to \tau$ is $\alpha \to (\beta \to \tau)$ and not $(\alpha \to \beta) \to \tau$. When the domain is a finite set, we may use the word **map** interchangeably with "function." When the codomain is potentially dependent upon the input, we use the **dependent product** operator: $\Pi_{s\in\sigma}\,Y_s$ where Y is, as with Σ, a σ-indexed collection of *sets*. $f \in \Pi_{s\in\sigma}\,Y_s$ iff $\forall_{s\in\sigma}\ \exists!_{y_s\in Y_s}\ f(s) = y_s$.[14] ($\sigma \to \tau$ is just the special case of a constant collection, i.e., $\forall_{s\in\sigma}\,Y_s = \tau$.) When σ is clear from context, we may abbreviate this to ΠY. The domain of a function $f \in \Pi_{s\in\sigma}\,Y_s$ is denoted $\mathrm{dom}(f) \overset{\text{def}}{=} \sigma$. Function evaluation always uses parentheses in this thesis, e.g., $f(x)$, rather than mere juxtaposition, e.g., $f\ x$, as would be traditional for functional programming.

The reuse of product notation for sets of functions is justified, or perhaps excused, by the observation that $\langle\tau_1,\ldots,\tau_n\rangle$ is isomorphic to $\Pi_{i\in\mathbb{N}_1^n}\,\tau_i$: "$\langle f(1),\ldots,f(n)\rangle$" can be read as sending a function $f \in \Pi_{i\in\mathbb{N}_1^n}\,\tau_i$ to a tuple or *defining* such a f from a tuple.[15] We will, therefore, sometimes use tuples of length k as (dependent) functions with domains $\mathbb{N}_1^k$.

When a function's codomain is a subset of its domain (or all of its dependent codomains are subsets of its domain), it is said to be an **automorphism**. Given an automorphsim $f \in \Pi_\sigma\,Y$, $\sigma' \subseteq \sigma$ is **closed under** f iff $\{f(s) \mid s \in \sigma'\} \subseteq \sigma'$. The notions of automorphism and closure thereunder extend to functions of multiple arguments, so we may, for example, speak of closure of a subset of $\mathbb{R}$ under addition in general, not just of its closure under addition by a particular constant.

We may transform (via the *covariant* powerset (categorical) functor, $\wp$) functions $f \in \sigma \to \tau$ to functions $\wp f \in \wp\sigma \to \wp\tau$ defined on sets of elements: $(\wp f)(\alpha) \overset{\text{def}}{=} \{f(a) \mid a \in \alpha\}$.

[12] For readers who are unfamiliar with the notation and may be concerned by the apparent re-use of numeric summation notation for something completely different, a worthwhile exercise is to demonstrate, for all sets α and α-indexed collections of sets Y, that $|(\Sigma_{a\in\alpha}\,Y_a)| = \Sigma_{a\in\alpha}|Y_a|$, where the quantifier on the left is our set-theoretic one and that on the right is the more traditional summation operator operating on cardinalities. We avoid referring to objects Y as "functions" to stave off the question of their codomain (i.e., the set of all sets?); fortunately, we have no need to use these objects in any first-class capacity, so we may keep them in the meta-language.

[13] Despite the reuse of the word "sum," dependent sums are unrelated to the notion of the "direct sum" of, e.g., groups. Notably, a "direct sum" requires that the sets being summed be equipped with a designated "point" element, while no such point is required for dependent sums. Further, for any *finite* number of objects, their direct sum corresponds with their direct (i.e., Cartesian) product; an *infinite* direct sum is a proper sub-object of the corresponding infinite direct product. The curious reader is encouraged to see chapter 8 of Hungerford [97]. Concretely, the direct sum of $\mathbb{Z}$ and $\mathbb{R}$, equipped with their typical additive group structures in which the 0 elements are the corresponding points, is (isomorphic to) $\langle\mathbb{Z},\mathbb{R}\rangle$ equipped with the product of the additive group structures, while the dependent sum of $\sigma_{\mathrm{z}} = \mathbb{Z}$ and $\sigma_{\mathrm{r}} = \mathbb{R}$ is (isomorphic to) $\Sigma_{i\in\{\mathrm{z},\mathrm{r}\}}\,\sigma_i = \{\langle\mathrm{z},s\rangle \mid s \in \mathbb{Z}\} \cup \{\langle\mathrm{r},s\rangle \mid s \in \mathbb{R}\}$. The conflict between different disciplines' terminology is unfortunate, but we shall not speak of direct sums again.

[14] There is no convenient set comprehension definition for a dependent product, thus this biconditional.

[15] Thus, as with Σ, $|\Pi_{a\in\alpha}\,Y_a| = \Pi_{a\in\alpha}|Y_a|$ relates functions and numeric product.

Similarly, $\wp_+$ transforms functions to take and return bags. Dependent functions $f \in \Pi_{s\in\sigma} Y_s$ transform to dependent functions $\wp f \in \Pi_{\alpha\in\wp\sigma}\{\{y_a \mid a \in \alpha\} \mid \forall_{a\in\alpha}\, y_a \in Y_a\}$. Our extensions of projection of terms—$t{\downarrow_\pi}$—to act on sets—$\tau{\downarrow_\pi}$—and bags—$\sigma{\downarrow^+_\pi}$—are of this form (but bag-view projection—$\sigma{\downarrow^@_\pi}$—is not); omitting $\wp$ in such projections reduces clutter.

Functions may be identified with sets of pairs obeying a functional dependence between first and second projections (i.e., $\forall_{a\in f\downarrow_1} |f[\{a\}/1]{\downarrow_2}| = 1$). In this context, we will use the infix pair constructor $\mapsto$, so that functions render as $\{s \mapsto t \mid \cdots\}$, rather than $\{\langle s,t\rangle \mid \cdots\}$, where s stands for an element of the domain and t its corresponding element of the (dependent) codomain. We also use this notation in 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 (well-typed) indexed collections and other notation by use of an **argument placeholder**, "$\cdot$": e.g., if a is a $\langle\mathbb{N},\mathbb{N}\rangle$-indexed collection of objects then by "$a_{\cdot,3}$" we mean the function $\{n \mapsto a_{n,3} \mid n \in \mathbb{N}\}$. If multiple placeholders appear in such a construction, they represent different arguments and apply left to right; i.e., $(a_{\cdot,\cdot})(x)(y) = a_{x,y}$. We will be more explicit when we need repeated use of arguments, e.g., $\{n \mapsto a_{n,n} \mid n \in \mathbb{N}\}$.

Function **composition** is written as is typical, with $(g \circ f)(x) \overset{\text{def}}{=} g(f(x))$ when $x \in \mathrm{dom}(f)$ and $f(x) \in \mathrm{dom}(g)$. Function composition itself gives rise to two functions, **pre-composition** (by g, i.e., $\cdot \circ g$) and **post-composition** (by f, i.e., $(f \circ \cdot)$). Combined with our view of tuples as equivalent to dependent functions, this means that $f \circ \langle x_1, \ldots, x_n\rangle = \langle f(x_1), \ldots, f(x_n)\rangle$, providing convenient notation for mapping a function over a tuple, much as $(\wp f)$ maps a function across a set. We occasionally compose functions made with placeholders; despite the repeated use of $\cdot$, the usual evaluation order applies: $(f_\cdot \circ \cdot)(g)(x) = (f_\cdot \circ g)(x) = (f_\cdot)(g(x)) = f_{g(x)}$, *not* $f_g(x)$. Functions can also be combined with the **right-biased merge** operator, $\triangleleft$: $(f \triangleleft g)(x)$ is $g(x)$ if $x \in \mathrm{dom}(g)$ and is $f(x)$ otherwise.

To pass multiple arguments to a function, we generally prefer to use a **curried** form: a higher-order function, e.g., $\alpha \to \beta \to \tau$. Occasionally, however, it will be useful to use the **uncurried** form: $\langle\alpha,\beta\rangle \to \tau$. These sets of functions are isomorphic, and this generalizes to any number $n \in \mathbb{N}$ of arguments. For any indexed collection Y, the isomorphism between the dependent functions $f \in \Pi_n(Y_n \to \tau)$ and $g \in (\Sigma_n Y_n) \to \tau$ is witnessed by (un)currying: $f(n)(y) \equiv g(\langle n,y\rangle)$

The use of dependent functions means that functions do not have a unique type. Every f with domain σ has type $\Pi_{s\in\sigma}\{f(s)\}$, of which f is the sole element; moreover, if $f \in \Pi_\sigma Y$, then $f \in \Pi_\sigma Y'$, when $\forall_{s\in\sigma} Y_s \subseteq Y'_s$. However, we will not allow our functions to implicitly restrict their domains, so $\mathrm{dom}(f)$ is always well defined. However, a function $f \in \Pi_{s\in\sigma} Y_s$ can be **upcast** to another type by narrowing its domain: $f|_\alpha$, with $\alpha \subseteq \sigma$, is in $\Pi_{a\in\alpha} Y_a$ and is such that $\forall_{a\in\alpha} f|_\alpha(a) = f(a)$.

Algebraic Structures A **semigroup** $\langle\sigma,\oplus\rangle$ consists of a set (σ; the semigroup is said to be "**over** σ") and an **associative** binary operator $\oplus \in \sigma \to \sigma \to \sigma$ (i.e., $\forall_{a,b,c\in\sigma}(a \oplus b) \oplus c = a \oplus (b \oplus c)$). A **monoid** $\langle\sigma,\oplus,\textcircled{0}\rangle$ extends a semigroup to have a **identity element** $\textcircled{0} \in \sigma$ such that $\forall_{s\in\sigma}\, \textcircled{0}\oplus s = s\oplus\textcircled{0} = s$. A semigroup or monoid is **commutative** if $\forall_{a,b\in\sigma}\, a\oplus b = b\oplus a$.

A **semiring** (or, less ambiguously, **rig**; see footnote 16, below) $\langle\sigma,\oplus,\textcircled{0},\otimes,\textcircled{1}\rangle$

consists of a set (σ), a commutative monoid over σ ($\langle \sigma, \oplus, \textcircled{0} \rangle$) and another monoid, also over σ ($\langle \sigma, \otimes, \textcircled{1} \rangle$). Not any choice of monoids will suffice; the semiring must obey two additional properties: ① **distributivity** of $\otimes$ over $\oplus$, i.e., $\forall_{a,b,c \in \sigma}\, a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ ("right distributivity") and $\forall_{a,b,c \in \sigma} (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$ ("left"); and ② $\forall_{s \in \sigma}\, \textcircled{0} \otimes s = s \otimes \textcircled{0} = \textcircled{0}$, i.e., that the additive identity, $\textcircled{0}$, is a **absorbing element** of $\otimes$.[16] The $\oplus$ operator is said to provide the "additive" aspect of the semiring structure, while $\otimes$ provides the "multiplicative" aspect.[17] A semiring is dubbed **commutative** if its *multiplicative* monoid is (additionally) commutative.

Adding a notion of **inverse**, $-s$, of each element s of a monoid produces a **group** $\langle \sigma, \oplus, \textcircled{0}, -\cdot \rangle$ when inverses obey $\forall_{s \in \sigma}\, \textcircled{0} = s \oplus (-s) = (-s) \oplus s$. Similarly, augmenting a semiring with *additive* inverse, i.e., promoting its additive monoid to a group, produces a **ring** and introduces a notion of "subtraction," with $a - b$ being defined as $a \oplus (-b)$.[18] When the monoid operator is written $\otimes$ or is otherwise "multiplicative," inverses are typically denoted using "exponential" notation, e.g., as "s^{-1}."

A (weak) **partial order**, $\langle S, \cdot \leq \cdot \rangle$, on a set S is a reflexive, anti-symmetric, transitive, *partial* relationship between elements of S. Reflexivity is the condition that every element relates to itself: $\forall_{s \in S}\, s \leq s$. Transitivity of $\leq$ means that it includes composition with itself: $\forall_{a,b,c \in S} (a \leq b \wedge b \leq c) \Rightarrow (a \leq c)$. Anti-symmetry means that no pair of distinct elements mutually relate, or, equivalently, that two putative elements that do mutually relate must be the same element: $\forall_{s,s' \in S} (s \leq s' \wedge s' \leq s) \Rightarrow (s = s')$. Elements $s, s' \in S$ are **comparable** (with regard to $\leq$) if either $s \leq s'$ or $s' \leq s$; partiality means that, potentially, not all pairs of elements are comparable. If, instead, all elements are comparable, the structure is a **total order**, **linear order**, or **chain**.[19]

A (meet) **semilattice** $\langle S, \cdot \wedge \cdot \rangle$ is a commutative semigroup whose **meet** operator, $\wedge$, is also **idempotent**, i.e., $\forall_{s \in S}\, s \wedge s = s$. Meet is sometimes called the **greatest lower bound** operator.[20] Semilattices give rise to partial orders: take $s \leq s'$ whenever $s = s \wedge s'$. Subset inclusion, $\subseteq$, is a partial order among sets and corresponds to the semilattice formed

[16] Some authors reserve the word "semiring" for a pair of *semigroups*, i.e., $\langle \sigma, \oplus, \otimes \rangle$, over a common σ and obeying distributivity. "Rig" is unambiguous, but, as we shall not avail ourselves of these structures without identity elements, we shall follow the natural language processing community in assigning the term "semiring" to rigs.

[17] The terms "additive" and "multiplicative" are standard and come from the fact that addition and multiplication form a semiring on many classes of numbers, e.g., the natural numbers: $\langle \mathbb{N}, +, 0, *, 1 \rangle$. However, in some cases these terms simply instill confusion, as in the "tropical semiring" $\langle \mathbb{R} \cup \{\infty\}, \min, \infty, +, 0 \rangle$, where the "multiplicative" aspect is *addition* [164]. When confusion is possible, we will either use the appropriate operator symbol or refer to "semiring addition" ($\oplus$) or "semiring multiplication" ($\otimes$).

[18] Thus, we can explain the term "rig" as a "ring without negatives." Some authors use "rng," pronounced "rung," to mean "ring without multiplicative identity," i.e., an additive group and multiplicative semigroup $\langle \sigma, \oplus, \textcircled{0}, \otimes \rangle$ obeying distributivity and additive-identity-absorption of $\otimes$. Thus, rigs are rngs are semirings of footnote 16.

[19] One can, equivalently, think of the partial relation $\leq$ as a total function in $S \to S \to \{\texttt{TRUE}, \texttt{FALSE}, \star\}$ where incomparable pairs $\langle s, s' \rangle$ are those for which $s \leq s' = s' \leq s = \star$. Otherwise, for comparable pairs, either $s \leq s' = s' \leq s = \texttt{TRUE}$ or reversing the order must flip the truth result, i.e., $\{s \leq s', s' \leq s\} = \{\texttt{TRUE}, \texttt{FALSE}\}$ A total ordering is then a partial order for which $\cdot \leq \cdot \in S \to S \to \{\texttt{TRUE}, \texttt{FALSE}\}$.

[20] A *join* semilattice defines a **join** operator, $\vee$, also called the **least upper bound**, but is identical in all but name. A **lattice** has both a meet and a join commutative semigroup which, together, obey the **absorption laws**, $\forall_{s,s'}\, s \vee (s \wedge s') = s$ and $\forall_{s,s'}\, s \wedge (s \vee s') = s$. We do not use lattices in our theory, but may occasionally discuss them for contrast.

by set intersection, $\cap$.[21]

A function $f \in \sigma \to \tau$ whose domain and codomain are both equipped with partial orders is **monotone** from $\langle \sigma, \leq \rangle$ to $\langle \tau, \leq^\dagger \rangle$ (though usually these are clear from context) if f preserves ordering: $\forall_{s,s' \in \sigma} (s \leq s') \Rightarrow (f(s) \leq^\dagger f(s'))$. In the special case that $\leq$ and $\leq^\dagger$ are the same relation (and defined at least on $\langle \sigma, \sigma \rangle \cup \langle \tau, \tau \rangle$), we may say that f is $\leq$-monotone. In general, a function $f \in \sigma \to \tau$ is a **homomorphism** of an algebraic structure if both σ and τ are instances of that structure and if f preserves the salient attributes thereof. Monotone functions could equivalently be called "partial order homomorphisms." The tuple length operator, $\mathrm{tlen}(\cdot)$, is a monoid homomorphism from $\langle \tau^*, \langle\rangle, \mathbin{+\!\!+} \rangle$ to $\langle \mathbb{N}, 0, + \rangle$, where by τ^* we mean $\langle\rangle \cup \langle \tau \rangle \cup \langle \tau, \tau \rangle \cup \cdots$,[22] because it preserves both the identity element, i.e., $\mathrm{tlen}(\langle\rangle) = 0$, and the binary operator, i.e., $\mathrm{tlen}(\vec{t} \mathbin{+\!\!+} \vec{s}) = \mathrm{tlen}(\vec{t}) + \mathrm{tlen}(\vec{s})$.

Lemmas for Projection and Refinement Because our projection and refinement operators are not often encountered in other documents, we pause here to record several immediate facts about them. The reader should feel free to skip this paragraph if they feel comfortable with the definitions.

① Projection at a fixed path is $\subseteq$-monotone: $\forall_\pi\, \sigma \subseteq \tau \Rightarrow \sigma{\downharpoonleft_\pi} \subseteq \tau{\downharpoonleft_\pi}$.

② A refinement of a set is a subset of that set: $\tau[\sigma/\pi] \subseteq \tau$.

③ Refinement at a fixed path is $\subseteq$-monotone in both of its arguments: $\sigma \subseteq \sigma' \Rightarrow \sigma[\tau/\pi] \subseteq \sigma'[\tau/\pi]$ and $\tau \subseteq \tau' \Rightarrow \sigma[\tau/\pi] \subseteq \sigma[\tau'/\pi]$.

④ Multiple refinements commute: $\sigma[\tau/\pi_1][\alpha/\pi_2] = \sigma[\alpha/\pi_2][\tau/\pi_1]$.

⑤ Refining before projecting at the same path is the same as refining and intersecting: $\sigma[\tau/\pi]{\downharpoonleft_\pi} = \sigma{\downharpoonleft_\pi} \cap \tau$. In particular, $\sigma[\tau/\langle\rangle] = \sigma \cap \tau$.

⑥ Multiple refinements intersect at the same path: $\alpha[\beta/\pi][\beta'/\pi] = \alpha[\beta \cap \beta'/\pi]$.

Properties of Functions of Bags Let S be a set of bags. A function $f \in \Pi_{\sigma \in S}\, Y_\sigma$ is

- **idempotent** if the multiplicity of elements does not alter the result, i.e., if $\forall_{\sigma,\sigma' \in S}\, \mathfrak{U}\sigma = \mathfrak{U}\sigma' \Rightarrow f(\sigma) = f(\sigma')$. ($Y_\sigma$ and $Y_{\sigma'}$ are not required to be equal.) Union of sets is idempotent, for example.

- **selective** if $\forall_\sigma\, Y_\sigma \subseteq \mathfrak{U}\sigma$: the output is *selected* from the input. Minimization is selective (and idempotent), for example. Set union, while idempotent, is not selective.

[21] Strictly, as we assume sets of elements to relate and meet, we should be careful that these constructions take place within any set of sets S in our underlying theory, and not between arbitrary sets.

[22] τ^* is the *least fixed point* of the equation $\tau^* = \langle\rangle \cup \{\langle t \rangle \mathbin{+\!\!+} \vec{s} \mid t \in \tau, \vec{s} \in \tau^*\}$, by tuple-based analogy to the Kleene star operator. It contains, exclusively, *finite* tuples whose components are elements of τ, and is equivalently defined as the smallest set such that $\langle\rangle \in \tau^*$, $\langle \tau \rangle \subseteq \tau^*$, and $\forall_{\vec{s},\vec{t} \in \tau^*}\, \vec{s} \mathbin{+\!\!+} \vec{t} \in \tau^*$.

- an **AC-reducer** if $\forall_{\sigma,\sigma',\sigma\uplus\sigma'\in S}\, f(\sigma \uplus \sigma') = f(\wr f(\sigma) \int \uplus \sigma')$, wherein we assume that S is suitably closed ($\forall_{\sigma,\sigma',\sigma\uplus\sigma'\in S} \wr f(\sigma) \int \uplus \sigma' \in S$). The primary use case for AC-reducers are domains $\wp_+\bar{\mathfrak{U}}_\infty\tau$, for some set τ, which are, intrinsically, so closed.[23]

Terms and Sets of Terms We construct a **Herbrand universe** $\mathcal{H}_\mathcal{F}$ from the underlying collection of symbols $\mathcal{F}$ [92]; elements of $\mathcal{H}_\mathcal{F}$ are, interchangeably, called (ranked) **trees** or (ground) **terms**. Explicitly, $t \in \mathcal{H}_\mathcal{F}$ is composed of a **functor** with fixed arity n, denoted $\mathtt{f}^{/n} \in \mathcal{F}$,[24] and a tuple of n terms: $t = \mathtt{f}\langle t_1, \ldots, t_n\rangle$, where $t_i \in \mathcal{H}_\mathcal{F}$. These t_i are called **immediate subterms** of t, and $\mathtt{f}^{/n}$ is called the **outermost functor** (or **root functor**) of t. Trees ground out at **leaves**, for which $n = 0$. A **non-ground term** (synonymously, **type**) is a subset of $\mathcal{H}$.[25] When $\mathcal{F}$ is clear from context, we will denote the corresponding universe simply as $\mathcal{H}$; unless otherwise stated, we assume "anonymous" functors of all arities, letting us include arbitrary-length tuples of terms into the term universe. Non-ground term products will be denoted like products of sets: $\mathtt{f}\langle\tau_1,\ldots\rangle \overset{\text{def}}{=} \{\mathtt{f}\langle t_1,\ldots\rangle \mid \forall_i\, t_i \in \tau_i \subseteq \mathcal{H}\}$. Projection is extended, within its inductive definition, to work on trees and sets thereof by ignoring any functors along the path: e.g., $\langle \mathtt{t}\langle\tau_1,\tau_2\rangle, \mathbb{N}\rangle\downarrow_{1.2} = \tau_2$.

Well-typed Typesetting At many points in this document, we will need to discuss several different sorts of objects at once. To provide a kind of mnemonic guide for the reader, we will use different font faces and alphabets for these different sorts. Our notation will tend to use s, t, a, b for base elements (very often, trees), $\sigma, \tau, \alpha, \beta$ for sets of elements (as typical when describing a programming language type system), and S, T, A, B for sets of sets of elements. (In short, $t \in \tau \in T$.) Tuples over sets of base elements will, naturally, be rendered as $\vec{\sigma} = \langle\sigma_1, \ldots, \sigma_n\rangle$. Rather than insisting upon lowercase Greek for all sets, when contrast is important and, especially, when forming sets of *tuples of sets*, we will use the Fraktur font (e.g., $\mathfrak{R}$, $\mathfrak{M}$). Fraktur also serves as something of a "grab bag" for miscellaneous symbols, though we attempt to limit their scope. Not all lower-case Greek letters are sets; notably, π will be used for paths.

Natural numbers and other elements of indexing sets will tend to be the lowercase Roman letters n, m, i, j, k. Throughout, we will use r as an index into the program's linear order of rules, Ξ (so chosen simply because it resembles a program source's horizontal line after line).

[23] AC-reducers $\tilde{g} \in ((\wp_+\bar{\mathfrak{U}}_\infty\tau) \smallsetminus \{\varnothing\}) \to \tau$ which additionally obey $\forall_{t\in\tau}\, \tilde{g}(\wr t \int) = t$ are in bijective correspondence with associative, commutative functions $g \in \tau \to \tau \to \tau$: $\tilde{g}$ is a catamorphic image of g and, in the other direction, $g(a)(b) = \tilde{g}\wr a, b\int$. If g has an identity $e \in \tau$, then $\tilde{g}$ can be extended so that $\varnothing \mapsto e$. Functions g are also said to be idempotent (if $\forall_{t\in\tau}\, g(t)(t) = t$) or selective (if $\forall_{s,t\in\tau}\, g(s)(t) \in \{s, t\}$); these properties are equivalent to their namesakes of $\tilde{g}$. Selectivity of an AC-reducer implies its idempotency.

[24] Yet again we encounter terminological difficulties. In the context of Prolog, "functor" means a symbol-with-arity and is almost entirely unrelated to the categorical notion of "functor" meaning a homomorphism between categories. We shall only rarely have reason to refer to the latter, and so will use "categorical functor" to avoid confusion. Prolog functors are also sometimes called **function symbols**, though we consider that terminology likely confusing. Readers familiar with Prolog literature may balk at our use of $\mathtt{f}^{/n}$ rather than the more traditional notation, $\mathtt{f}/n$; the latter clashes too often with division for the author's comfort. One can think of $\cdot^{/\cdot}$ as yet another piece of pair-forming syntax, if desired.

[25] The identification of a non-ground term *with* its set of groundings is perhaps unusual. We contrast our approach with the more usual representation involving explicit variables in §4.

Smallcapital letters will be used for a variety of purposes, including special symbols and mnemonics for common path prefixes.

In addition to fonts, this thesis uses color as a visual aid to help the reader relate objects; as with fonts, coloration is merely assistive and will never carry essential semantic content. Certain kinds of objects will be made to stand out by giving them consistent coloration throughout the document, related objects within figures may share colors, and grouping syntax (parentheses, braces, etc.) may be color-matched when deeply nested (e.g., $\{\langle\langle\rangle,\langle\langle a\rangle,(b+c)\cdot d\rangle\rangle\}$).

Certain regions of prose will have their ends denoted with symbols to the right of the last line. Proofs will be so marked with "□," while examples will use "◊."

Pseudocode Language Throughout this document, we write our algorithms in a simple, imperative, strongly-typed, lexically-scoped procedural language with pattern matching. Its syntax is indentation-sensitive and includes conditionals, loops, procedures, and imperative data structures and references. Code blocks are occasionally explicitly denoted with braces and use semicolons to separate statements within a block.[26] When pattern matching, a value may be discarded by use of an underscore (_). Evaluation order of arguments to procedures is not specified; while the results of execution may not be identical across varying orders, our listings will remain correct.

We will use special formatting for `procedureNames`, `reserved words`, and type annotations. Some procedures are `Exported` from their listings, and would be linked against others as part of a real implementation, while other procedures are more `local`.

We will generally refer to lines with their containing listing, e.g., "line 3 of listing 1.2," but, if clear from context, we may occasionally suppress the listing number and just give the line number. Un-captioned listings within the main body of the prose are called, and referenced as, "blocks;" such creatures are typeset in rough analogy with equations, with a counter on the right, though we choose to place it at the top of the listing rather than the middle.

Example 2: To give a feel for our pseudocode language, here is a simple example:

B. 1.1

```
def Collatz($i \in \mathbb{N}$) $\in \mathbb{N}$
  $s$ ← 0
  while $i \neq 1$ do
    $s$ ← $s + 1$
    let $\langle q, r\rangle$ = divmod($i$,2)
    $i$ ← if r = 0 then $q$ else $3i + 1$
  return $s$
```

This block would be cross-referenced as "block 1.1." ◊

[26] The use of indentation sensitivity is inspired by Haskell's two-dimensional syntax, which "desugars," i.e., is deterministically and mechanically translated, to a more explicit braces-and-semicolon syntax also available to, but rarely used in, source files. The curious reader is directed to §9.3 of the Haskell 98 report [142].

1.4 Aside: Prolog and The Logic of Logic Programming

There is a long and storied history of using clausal logic to represent computer programs. A full re-telling of the story is surely out of scope, but an excellent first-hand account of the early efforts is Kowalski's [111]. Perhaps the earliest work which proposes logic not just for analysis, but rather *construction*, of computation is, unsurprisingly, by Alonzo Church, in 1957, discussing using "recursive arithmetic" to synthesize logic circuits [28].[27] The effort to synthesize *programs* from logic began in earnest in the late 1960s with Green [83]. Prolog emerged in the early 1970s; an excellent retrospective by its designers, Alain Colemerauer and Philippe Roussel, is available [34]. By 1974, the logical foundations of the core of Prolog, including the notion of "selective linear definite clause resolution" (SLD resolution), had been formalized in Kowalski [110] and Van Emden [179]. (These authors collaborated to produce Van Emden and Kowalski [178] in 1976, which we use as inspiration for much of the next several paragraphs.) SLD resolution was shown, in 1980, to be a sound and complete resolution technique for Prolog programs [11].

Let us spend a little ink describing SLD resolution (a longer exposition can be found in Van Emden and Kowalski [178, §3 to §4]). A grounding of a Prolog rule (without negation or calls to extra-logical facilities) "$\texttt{h :- b}_1\texttt{, …, b}_k$" can be interpreted as a **definite Horn clause**, i.e., a disjunction with exactly one *non-negated* literal: $h \vee \neg b_1 \vee \cdots \vee \neg b_k$.[28] This gives the correct interpretation as an implication: if all b_i are TRUE, the clause can only be satisfied by making h TRUE as well, but if any b_i is FALSE, the clause is satisfied regardless of the value assigned to h. The task at hand, then, is to decide whether some initial, **goal**, query, a Horn clause containing *only* negated literals, $\neg q_1 \vee \cdots \vee \neg q_n$, can be *refuted* by the *conjunction* of a program's rules' groundings: $(h_1 \vee \neg b_{1,1} \vee \cdots \vee \neg b_{1,k_1}) \wedge (h_2 \vee \neg b_{2,1} \vee \cdots) \wedge \cdots$. If no $\{h_i \mid i\}$ equals q_1, there is no reason to believe that q_1 is TRUE, and so we cannot refute the goal. On the other hand, suppose that q_1 equals h_i for some i (there may be several such i). Then we may be able to refute our goal by (recursively) considering the refutation of $\neg b_{i,1} \vee \cdots \vee \neg b_{i,k_i} \vee \neg q_2 \vee \cdots \vee \neg q_n$, obtained by observing that the truth of $\neg q_1 = \neg h_i$ is implied by the refutation of the body corresponding to h_i. This may not be refutable, so we should ensure that we try all possible i. Axiomatic rules, i.e., those without bodies or, equivalently, those whose clausal interpretation is just a positive literal, serve to shorten the goal clause under consideration.[29] A successful refutation is a series of resolution steps that results in attempting to refute an empty goal clause: the empty disjunct is FALSE, and reading the resolver steps backwards gives a *linear* (the "L" in "SLD") proof that the goal clause is therefore also FALSE.

It was not strictly *necessary* that we restricted our attention to the leftmost literal in the goal clause; in fact, it would be logically coherent to pick any subset of the goal

[27] The "Historical Note" section at the end of Church [28] contains a number of early references which discuss the use of logics and algebras to analyse circuits. While out of scope for the present document, it is mentioned here as an attractive tangent for the historically-minded reader.

[28] A **Horn clause** is, in general, a disjunction with *at most one* non-negated literal. The class is named after Alfred Horn, who recognized their significance in 1951 [96]. The "D" in "SLD" emphasises that its program input is a conjunction of definite Horn clauses.

[29] Primitive arithmetic operations are handled as, essentially, an infinite collection of axiomatic literals and so achieve the same shortening of the goal clause.

disjuncts for resolution (though picking the empty set does not get one very far). However, this *selective* (the "S" in "SLD") attention makes the algorithm simpler to describe without altering its ultimate conclusions. As presented, the goal clauses in fact form a *stack*, in which we are always attempting to resolve the most-recently introduced literal. This, also, is not fundamental to the algorithm's operation, $\vee$ being associative and commutative.

The use of *equality* in the procedure given above is sufficient for the case of propositional logic without variables or for a grounded Prolog program, but Prolog actually considers goal clauses and rules with variables. In set notation, the clausal form of a rule is, as might be imagined, somewhat akin to a μDyna rule (§3.1): a union of sets of the form $\{\mathtt{h}\langle\ldots\rangle \vee \neg\mathtt{b}_1\langle\ldots\rangle \vee \cdots \mid \cdots\}$. Each clause within the set must have the same number of literal terms; without loss of generality, we put the non-negated literal first. The resolution step then merges the goal clause set γ and rule clause set ρ into the new goal clause set $\{\vec{b} \vee \vec{g} \mid (\neg h \vee \vec{g}) \in \gamma, (h \vee \vec{b}) \in \rho\}$, where we have taken the liberty of using $\vec{b}$ and $\vec{q}$ to represent the rest of the string of disjuncts from the rule body and goal clauses, respectively.[30] As one often cares about the values of variables in irrefutable queries, one might imagine using a union of sets of reversed definite Horn clauses, $\{\neg\mathtt{q}_1\langle\ldots\rangle \vee \cdots \vee \neg\mathtt{q}_k\langle\ldots\rangle \vee \mathtt{q}_0\langle x_1, \ldots, x_k\rangle \mid \cdots\}$ where $\vec{x}$ captures the variables in question. A successful refutation is then one in which the goal clause is no longer empty but instead a set of positive $\mathtt{q}_0{}^{/k}$ standing alone.

Variables not negated have a different character than those under negation. The Prolog rule "`a(A) :- e(E)`" is interpreted (by the above resolution procedure) as *all* $\mathtt{a}\langle\mathcal{H}\rangle$ being TRUE if *there exists* $e \in \mathcal{H}$ such that $\mathtt{e}\langle e\rangle$ is TRUE (i.e., $\neg\mathtt{e}\langle e\rangle$ is refutable). Thus, the goal (query) "`f(X)`" will yield a stream of $\alpha \subseteq \mathcal{H}$ for which $\{\neg\mathtt{f}\langle a\rangle \mid a \in \alpha\}$ were all refuted. It is not, in this scheme, possible to have a subgoal that can be discharged only if *all of* $\{\neg\mathtt{f}\langle a\rangle \mid a \in \mathcal{H}\}$ can be refuted.

This procedural view is perhaps unsatisfying and, in any case, not obviously connected to the fixed-point view of the meaning of our programs throughout this thesis (notably, §2.5.1 and §3.1.4). The connecting insight is from Van Emden and Kowalski [178, §6] and is that one can define, for a program P, an operator T_P, which, given a set of TRUE items (i.e., literals, terms), computes the set of items which must be "immediately" TRUE due to the rules of P. Recast into our notation, T_P reads as

$$T_P(\tau) = \{h \mid \exists_{\rho \in P} (h \vee \neg b_1 \vee \cdots \vee \neg b_n) \in \rho, \forall_i\, b_i \in \tau\}.$$

That is, $h \in T_P(\tau)$ if there exists a rule ρ in the program which contains a ground instance with head h and for which all subgoals are true in τ. The semantics of the program is then taken to be the smallest τ such that $\tau = T_P(\tau)$, assuming it exists.[31] Such a τ is called the **minimal Herbrand model** of P. Readers curious about fixed-point semantics of logic programs are encouraged to read Fitting [67].

[30] This generalizes the typical, variable-based notion of finding the most-general unifying substitution of the goal literal q_1 and head h and applying that substitution to the new goal clause. See §4.2.1 for more discussion.

[31] Equivalently, without appealing to "smallest," $\tau = \bigcap T$ with $T = \{\tau' \subseteq \mathcal{H} \mid T_P(\tau') \subseteq \tau'\}$, assuming $T \neq \varnothing$.

1.4.1 Non-monotonic Reasoning: Negation

Early in its life, Prolog added *negated subgoals* to rules. The programmer's intent in the rule "`p :- q, \+ r, s`", wherein `\+` denotes negation, is to prove `p` if `q` and `s` can be proven and `r` cannot be. The *procedural* interpretation of such a subgoal is that it is successfully satisfied (and the solver may move on to the next subgoal) if the solver fails to satisfy the negated subgoal. Thus, it is given the name **negation as failure** (NAF) [30]. (The analogous capability in Dyna is discussed in example 28, in §3.1.3.)

An awkward feature of NAF is that a free variable within a negated subgoal retains its existential flavor, but is now scoped under a negation, and will not be refined. Thus, "`a :- \+ e(E)`" constitutes a proof of $\mathtt{a}^{/0}$ if *no* $\mathtt{e}^{/1}$ item can be proven. If $\mathtt{d}^{/1}$ brings its argument to ground, "`a :- d(E), \+ e(E)`" will behave as expected: it proves $\mathtt{a}^{/0}$ iff there exists some $e \in \mathcal{H}$ such that $\mathtt{d}\langle e\rangle$ is TRUE (i.e., $\neg\mathtt{d}\langle e\rangle$ can be refuted) and $\mathtt{e}\langle e\rangle$ is not. However, "`a :- \+ e(E), d(E)`," which ought to be logically equivalent, yields fewer proofs: the negated subgoal behaves as just described, only succeeding when *no* $\mathtt{e}^{/1}$ can be proven. Going forward, we assume that every negated subgoal is ground so that the procedural interpretation coincides with our intuition.

Even so assuming, the logical semantics of such negations were open until 1978, when Clark [30], demonstrated a "program completion" transformation that rewrites a Prolog program's rules' implications into a first-order formula with *bi*-conditionals: an item is TRUE if *and only if* there is a rule that proves it, justifying the use of exhaustive search to prove the item's negation [see 30, Theorem 2]. The resolution procedure so obtained eventually came to be known as "SLD with Negation as Failure" (SLDNF).

Clark [30], additionally, shows that the database completion semantics describe a *constructive* ("intuitionistic") logic, in which the law of the excluded middle (i.e., the axiom $\forall_\phi\, \phi \vee \neg\phi$ where ϕ ranges over *all propositions*, including itself) is not justified; in practice, this implies non-termination of certain programs under SLDNF, but, logically, it means that there are terms neither assigned truth nor falsity by the program. Clark gives the example of attempting to resolve $\mathtt{p}\langle\mathtt{a}\langle\rangle\rangle$ against a program consisting of "`p(X) :- p(f(X))`" (with no other $\mathtt{p}^{/1}$-headed rules): SLDNF will fail to terminate (always with a single, ever bigger, literal on its goal stack), while first-order logic refuses to demonstrate truth or falsity of $\mathtt{p}\langle\mathtt{a}\langle\rangle\rangle$ given only the bi-conditional $\forall_x\, \mathtt{p}\langle x\rangle \Leftrightarrow \mathtt{p}\langle\mathtt{f}\langle x\rangle\rangle$ [adapted from 30, (13)].

Another, fundamentally unweighted semantics of negation, and possibly the most well-known, is that of Gelfond [74], presented in 1987 and further developed as the "Stable Model Semantics" [75]; this formulation of negation forms the basis of a different branch of work in logic programming, termed "Answer Set Programming" (ASP).

However, while Prolog remains an unweighted language, in the sense used herein, yet a different semantics for negation was given in 1985 [66], explicitly using a *three-valued* logic, which designates predicates as true, false, or *undefined.* The "undefined" value captures both inconsistency (e.g., `p` iff not `p`) and tautology (e.g., `p` iff `p`, with no other assertions about `p`). Other three-valued semantics emerged, including "Constructive Negation" [187] and the "Well-Founded Semantics" [180]. The Well-Founded semantics has an efficient implementation, using a resolution procedure called "Linear resolution with Selection function for General logic programs" (SLG) [26, 27] and was even extended to work with aggregation [105]. More recently, when considering a variant of Datalog (a subset of Prolog) which allows one to not just *test for* negation but to explicitly *assert* negated heads, a series of *four-valued* logics have been considered [119]. The utility of a larger space of values within the semantics of logic programs provides, we feel, additional justification for persuing *weighted* logic programs.

1.5 Aside: Inexact Values

Most of this thesis takes place in some abstract, ethereal plane where all computations are exact, memory is infinite, time is of no issue, and any adversaries are precisely constrained in their capabilities. The challenges addressed by this thesis remain interesting, even in light of their idealized nature. However, we feel that we must at least nod in the direction of some issues likely to be faced by those who will come after us, and who, following in the footsteps of Prometheus, or perhaps of Icarus, would seek to wrestle mathematics into the real world and onto carefully drawn lines in ultra-pure sand. Towards that end, we will consider **inexact values**, quantities for which something as seemingly banal as *equality* is a fuzzy notion.

Example 3 (*Floating Point Approximations*): A **floating point** numeric system is an approximation of the real numbers where one uses a fixed number of bits to represent each of the "significand" (also called the "mantissa") and "exponent" of a number. Very briefly, every positive real number $r \in (0, \infty) \subset \mathbb{R}$ can be *normalized* so that $r = (1 + s) * 2^x$ for some exponent $x \in \mathbb{N}$ and significand $s \in [0, 1) \subset \mathbb{R}$ (that is, the unit interval inclusive of 0 and exclusive of 1). 17.125 (in base 10) is 10001.001 in base 2, and so admits normalization with s, in base 2, being 0.0001001 (or 0.703125, in base 10) and $x = 4$. If we restrict to using 6 bits for significand and 3 for exponent, however, we are forced to approximate 17.125 as either 17 or 17.25. The net effect of such an encoding is to give relatively fine approximations near 0 and coarser approximations as magnitudes become larger.[32] While that is sensible enough, the result is that many properties we might depend upon in numeric analysis are no longer true. Addition is not, for example, associative, as $(17 + .125) + .125$ is 17 (assuming we round down) while $17 + (.125 + .125)$ is surely 17.25. We do not wish to conclude that .25 and .125 and 0 are equal quantities; so, we must abandon our notion of strict, structural (i.e., bit-wise) equality when approximating real numbers this way. Thus, addition on floating point numbers is not, properly, an AC-reducer, though it is convenient to occasionally *pretend* that it is, but we must not allow such approximations within the values computed by a program to adversely affect the solver algorithm itself.

Readers curious for more details of floating point systems are invited to read Bryant and O'Hallaron [24, §2.4]; an exceptionally enthusiastic reader should be directed to the IEEE-754 standards document [98], which specifies surely the most pervasive of machine encodings of floating point numbers. ◊

Example 4 (*Interval Arithmetic*): Alternatively, one may approximate a real number by giving *bounds* on its possible value; a typical mechanism is to specify an *interval* containing the number. π (here, the ratio of a circle's circumferance to diameter in Euclidean geometry,

[32] Within IEEE-754, even finer approximations near 0 are obtained by use of *denormalized* representations where the leading 1 bit is no longer implied for the smallest possible exponent. There are also explicit representations of (signed) infinity and "Not A Number."

not a path within a tree) is, for example, in $[3.141, 3.142]$. One can work out an entire theory of arithmetic for intervals (e.g., Warmus [188] and Moore [129]) or a generalization which works with *finite unions* of intervals (e.g., Dreyer [45, ch. 2]).

Because an interval essentially behaves as an existential—the number intended is *somewhere* within the given bounds—equality becomes an almost entirely one-sided notion. Certainly $[1, 2]$ and $[3, 4]$ represent *unequal* quantities, but a number represented as $[1, 4]$ might, but need not equal, a number represented by any of $[0, 5]$ (super-interval), $[2, 3]$ (sub-interval), $[2, 5]$ (overlapping interval), or even $[1, 4]$ (identical interval). The only time equality can be assured is when the interval represents a singleton: $[2, 2]$ is surely reflexively equal. ◊

The introduction of such inexact values into the domains manipulated by the solver algorithms of this thesis introduces a variety of subtle problems. We address these by constraining the behavior of the solver (e.g., in §2.3.3.1 and §2.4.1) or by rejecting programs which would necessitate the solver's pondering of equality of inexact quantities (§5.5).[33]

[33] There is another proposal on the table, as well, which aims to address some of the challenges of inexact values by relaxing the notion of *functional dependence* in some cases. In this design, there may be *multiple* results for some expressions which use queries in "inverse modes;" that is, while there is only one value $x/2$ for each x, there may be multiple "y such that $y * 2 = x$." This author does not believe himself sufficiently versed with the proposal to speak intelligently about it, but, in the interest of completeness, notes that it has been proffered as another tool in the effort to manage inexact values.

Chapter 2

Finite Circuit Inference and Lightweight Extensions

> [A]utomatic programming always has been a euphemism for programming with a higher-level language than was then available to the programmer. Research in automatic programming is simply research in the implementation of higher-level programming languages.
>
> David Lorge Parnas. *Software Aspects of Strategic Defense Systems*, 1985 [140].

2.1 Arithmetic Circuits

2.1.1 Precursors

The first step in our gradual buildup to Dyna is to consider an extremely simplified design of the core solver for Dyna programs. We will gradually add embellishments in this chapter before turning our attention, in the next chapter, to extending our design to work with Dyna itself. This section is little more than warm-up and formalization of well-established concepts. Readers familiar with the notions of expression trees (and forests) may wish to skim or skip over to the beginning of §2.1.2. Readers familiar with computational circuits more generally may wish to skim all the way to the beginning §2.2, which reviews the algorithmic machinery for *solving* circuits before introducing our own EarthBound algorithm.

2.1.1.1 Expression Trees

An **expression tree** is a tree whose collection of underlying functors-with-arity, ϕ, comes equipped with a **ϕ-algebra**: a pair of a **carrier** set, χ, and an **interpretation function**, $e \in \Pi_{\mathtt{f}/n \in \phi}(\chi^n \to \chi)$.[34] The interpretation of an expression tree under a given algebra is denoted $[\![\cdot]\!]_e$ (or just $[\![\cdot]\!]$ when e is clear from context) and is done by bottom-up traversal: all immediate subterms of t must have been interpreted before t itself can be. That is,

[34] Algebra interpretation functions are often given, instead, as the (dependently un-Curried) isomorphic $(\Sigma_{\mathtt{f}/n \in \phi}\chi^n) \to \chi$. We prefer the Curried form as the partial applications correspond to the interpretation of particular symbols in ϕ.

$[\![\mathtt{f}\langle t_1,\ldots,t_n\rangle]\!] \overset{\text{def}}{=} e(\mathtt{f}^{/n})(\langle[\![t_1]\!],\ldots,[\![t_n]\!]\rangle)$. Unpacking the recursion, the nullary nodes are first interpreted as, e.g., $e(\mathtt{f}^{/0})(\langle\rangle)$; then, nodes whose immediate subterms are all nullary are interpreted, using the values from χ assigned to said subterms; etc.

Example 5: Suppose $\phi = \{\mathtt{neg}^{/1}, \mathtt{add}^{/2}, \mathtt{mul}^{/2}\} \cup \{n^{/0} \mid n \in \mathbb{N}\}$. One possible algebra for this structure is a ring over $\chi = \mathbb{Z}$: $e(\mathtt{neg}^{/1})(\langle x\rangle) = -x$, $e(\mathtt{add}^{/2})(\langle x_1, x_2\rangle) = x_1 + x_2$, $e(\mathtt{mul}^{/2})(\langle x_1, x_2\rangle) = x_1 * x_2$, and $e(n^{/0})(\langle\rangle) = n$. The tree $\mathtt{mul}\langle\mathtt{add}\langle 1, \mathtt{neg}\langle 3\rangle\rangle, \mathtt{neg}\langle 2\rangle\rangle$ would then interpret to having value $(1 + (-3)) * (-2) = 4$. ◊

Example 6: The Herbrand universe $\mathcal{H}$ itself is a carrier of an algebra with $\phi = \mathcal{F}$. The function e is trivial: $e(\mathtt{f}^{/n})(\langle t_1,\ldots,t_n\rangle) = \mathtt{f}\langle t_1,\ldots,t_n\rangle$. That is, e builds a term from its functor and immediate subterms, and is, in fact, when uncurried, witness to an isomorphism between terms and the pair of functor and (tupled) subterms. This algebra is (up to isomorphism) special; it is the *initial* $\mathcal{F}$-algebra, as *any other* algebra on $\mathcal{F}$ can be obtained from this initial algebra by suitable algebra homomorphism. We shall not dwell on the point, but the curious reader is directed to section 2.2 of Pierce [145] for a gentle introduction and/or to Meijer, Fokkinga, and Paterson [122] for the canonical text which brought such categorical considerations to the attention of functional programmers. ◊

2.1.1.2 Expression Forests

Nothing fundamental to interpretation changes if we replace the trees above with nodes in a directed, acyclic graph (DAG). Letting $\mathcal{I}$ denote the set of nodes (also called **items**), an **expression forest**[35] is a DAG wherein ① a function $l^{\mathrm{s}}_{\cdot} \in \mathcal{I} \to \phi$ labels each node with a symbol (and arity); ② for each $j \in \mathcal{I}$, if $l^{\mathrm{s}}_{j} = \mathtt{f}^{/n}$, then j has n in-edges; and ③ each node j equips its set of n in-edges with a linear ordering, which we denote as a n-tuple, $\vec{P}_j$. Algebraic interpretation of this structure continues in the same vein as for trees: given $j \in \mathcal{I}$, $[\![j]\!] = e(l^{\mathrm{s}}_{j})([\![\cdot]\!] \circ \vec{P}_j) = e(l^{\mathrm{s}}_{j})(\langle[\![i_1]\!],\ldots,[\![i_n]\!]\rangle)$ where i_1 through i_n are the n nodes with edges to j, ordered appropriately. The algorithmic order in which nodes become available for interpretation (because all of their in-neighbors have been interpreted) is now called a **topological sort** because it provides a linear ordering on $\mathcal{I}$ such that all edges point to later-ordered nodes.

In this more general framework, we can envision shared sub-structure (i.e., nodes with out-degree greater than one) as well as multiple heads (i.e., more than one node with out-degree zero).

Example 7: Confined to trees, evaluating an expression of the form "(1 + 2) + (1 + 2)" would necessarily involve three additions. However, an equivalent expression forest, perhaps represented as "$\mathtt{let}\ x\ \mathtt{=}\ \mathtt{1}\ \mathtt{+}\ \mathtt{2}\ \mathtt{in}\ x\ \mathtt{+}\ x$," can be evaluated with *two* additions, by reusing the intermediate result. This kind of transformation is wildly popular as part of optimizing compilers' operation, within "common subexpression elimination" [31] or "global value numbering" [106] phases. ◊

[35] In some existing literature, notably that of parsing within natural language processing, starting with [172], "forest" often implies a DAG with a single root but with the possibility of reused intermediates. Here, we mean to allow many roots; in fact, disjoint expression trees, if considered together, form, in our taxonomy, a perfectly acceptable expression forest.

2.1.2 Arithmetic Circuits Proper

An expression forest as specified is a *static* structure: once specified, all that remains is to compute the interpretation. However, in many cases, we will want to use the same structure with different inputs (i.e., we may wish to vary the nullary nodes' values while leaving the rest of the structure intact); these inputs may, for example, represent the current values of some physical state as it evolves through time (possibly under the influence of the leaves of the computation). Towards that end, we reduce the domain of $l^{\mathrm{s}}_{\cdot}$, removing some subset of the nullary nodes, denoted $\mathcal{I}_{\mathrm{inp}}$ and called **input** nodes; the remainder of the items we call **derived** and denote as $\mathcal{I}_{\mathrm{der}}$. When we wish to interpret the resulting structure, we will need a map which provides values to our input nodes, i.e., a map $\mathrm{inp} \in \mathcal{I}_{\mathrm{inp}} \to \chi$.[36] We call the resulting structure—a forest missing its inputs—an **arithmetic circuit** [21].[37]

One may wonder if $l^{\mathrm{s}}_{\cdot}$ and ϕ are fundamental, as their only use within $[\![\cdot]\!]$ is composed with e. Indeed, we can divorce our notion of computation from the symbolic vocabulary ϕ with which we started and simply associate every node $j \in \mathcal{I}_{\mathrm{der}} \overset{\mathrm{def}}{=} \mathcal{I} \setminus \mathcal{I}_{\mathrm{inp}}$ with a function $\chi^{n_j} \to \chi$ where $n_j \in \mathbb{N}$ is the in-degree of j; that is, we now take $e \in \Pi_{j \in \mathcal{I}_{\mathrm{der}}}(\chi^{n_j} \to \chi)$. Equivalently, we take $\mathcal{I}_{\mathrm{der}}$ itself *as* the set of symbols ϕ. We call nodes in $\mathcal{I}_{\mathrm{der}}$ derived as their values are the result of computation from other nodes' values.

Relative Nomenclature Let $\mathcal{E}$ be the set of directed edges, with an edge $e \in \mathcal{E}$ from i (the **source** of e) to j (the **target** of e) being denoted $e = \langle i, j \rangle$. Each item j has a set of **parents**, $P_j \overset{\mathrm{def}}{=} \{i \mid \langle i, j \rangle \in \mathcal{E}\}$ (i.e., j's in-neighbors), and a set of **children** by $C_j \overset{\mathrm{def}}{=} \{k \mid \langle j, k \rangle \in \mathcal{E}\}$ (i.e., j's out-neighbors). Transitive parents are called **ancestors**, and transitive children are called **descendants**. Derived items with no children are, then, **leaves**; items, derived or input, with no parents are **roots**.[38]

Solutions All told, then, the value assigned to an item is

$$[\![j]\!] = \begin{cases} \mathrm{inp}(j) & j \in \mathcal{I}_{\mathrm{inp}} \\ e(j)(\langle [\![i_1]\!], \ldots, [\![i_{n_j}]\!] \rangle) & j \in \mathcal{I}_{\mathrm{der}} \end{cases} \tag{2.1}$$

where, as before, i_1 through i_{n_j} are the in-neighbors of j, ordered appropriately. We call the function $[\![\cdot]\!]$, which extends inp from $\mathcal{I}_{\mathrm{inp}}$ to $\mathcal{I}$, the **solution**; it is guaranteed to uniquely exist while the circuits remain finite and acyclic, as they will, up until §2.5.

[36] The partition of specifications—originally, of logic programs—into input and derived items appears to have been first articulated within Reiter [156]. That and other literature uses the terms **extensional** and **intensional** to refer to what we term input and derived, respectively.

[37] The word "circuit" in this context, while standard, ultimately comes from electrical engineering, wherein one may discuss a "logic circuit" in referring to an arrangement of logic gates for carrying out a particular, finite-depth, acyclic computation. Just as electrical circuits can be extended to execute cyclic computations, so too can our arithmetic circuits; see §2.5.

[38] The literature varies as to whether input 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. The sole standardized thing seems to be that roots are at the top and leaves at the bottom, which suggests a somewhat distorted view of nature. Anyway, we treat inputs as roots and ancestors and draw them at the top. Edges point and information flows downward in our drawings. As a result, "bottom-up" reasoning (forward-chaining) actually proceeds from the top of the drawing down.

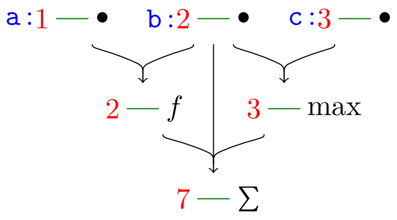


Figure 2.1: An example arithmetic circuit with $\chi = \mathbb{N}$, showing the function for each derived item (max, $\sum$, and $f(\langle a, b\rangle) = b^a$; for AC-reducers, the order of parent items is immaterial) and the symbol • for each input item. Item values are shown in red, and selected item names in blue. Ultimately, this circuit computes $\sum\wr \mathtt{b}^{\mathtt{a}}, \mathtt{b}, \max\wr \mathtt{b}, \mathtt{c}\wr\wr = \sum\wr 2, 2, 3\wr$.

Example 8: Figure 2.1 shows a small arithmetic circuit, with three input nodes and three derived nodes. This circuit uses $\mathbb{N}$ as its space of values, χ. Two of the derived nodes therein use different AC-reducers (recall "Properties of Functions of Bags," in §1.3) while one does not, as a demonstration that arbitrary functions may be used as well. ◊

2.1.2.1 Special Cases

and/or Graphs A Datalog program [25, 72] can be regarded as a concise specification of a **boolean circuit**, which is the special case where χ is a doubleton set, say, {TRUE, FALSE}. The items $\mathcal{I}$ correspond to propositional terms of the logic program and conjunctions thereof, 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' corresponding OR nodes, and whose child is an OR node corresponding to the head item. This kind of circuit is called an AND/OR graph, and the AND and OR functions are defined on χ in the standard way: AND is TRUE iff all of its inputs are TRUE, and OR is TRUE iff any of its inputs are TRUE. Datalog is sometimes extended to allow limited use of NOT nodes as well.

Example 9: A simple Datalog program might compute a traditional database join and projection of two relations. For example, the Datalog **rule** "`rs(Y,Z) :- r(J,X,Y), s(J,Z)`" can be read as specifying entries in the $\mathtt{rs}^{/2}$ relation by joining $\mathtt{r}^{/3}$ with $\mathtt{s}^{/2}$, equating the first columns of both (by reusing the variable `J`), and discarding—projecting away—the second column of `r` and the first columns of both relations (the variables `J` and `X` appear only in the body). We can give a circuit interpretation of this rule thus:

① Every potential row of the $\mathtt{r}^{/3}$, $\mathtt{s}^{/2}$, and $\mathtt{rs}^{/2}$ relations are made manifest as items. If, for example, all columns range over the finite set $\mathbb{N}_1^{10}$: $\{\mathtt{r}\langle j, x, y\rangle \mid j, x, y \in \mathbb{N}_1^{10}\} \subseteq \mathcal{I}$, $\{\mathtt{s}\langle j, x\rangle \mid j, x \in \mathbb{N}_1^{10}\} \subseteq \mathcal{I}$, and $\{\mathtt{rs}\langle y, z\rangle \mid x, y \in \mathbb{N}_1^{10}\} \subseteq \mathcal{I}$. All $\mathtt{r}^{/3}$ and $\mathtt{s}^{/2}$ items are input (and so are in $\mathcal{I}_{\text{inp}}$, not just $\mathcal{I}$) and all $\mathtt{rs}^{/2}$ items are derived (in $\mathcal{I}_{\text{der}}$).

② Additionally, we create derived items $\{\mathtt{join}\langle j, x, y, z\rangle \mid j, x, y, z \in \mathbb{N}_1^{10}\} \subseteq \mathcal{I}_{\text{der}}$.[39] These items are intended to represent the availability of a row in the join of the two relations, and as such we add edges to the item $\mathtt{join}\langle j, x, y, z\rangle$ from both the items $\mathtt{r}\langle j, x, y\rangle$ and $\mathtt{s}\langle j, z\rangle$. The function associated with each $\mathtt{join}^{/4}$ item is logical AND: the item only represents a row in the join if there are corresponding rows in both source relations.

[39] Really, here, $\mathtt{join}^{/4}$ stands for a symbol not otherwise used in the program. If there are multiple rules, they must each be given a different set of AND nodes.

③ Last, we draw edges from each $\texttt{join}\langle j, x, y, z\rangle$ to $\texttt{rs}\langle y, z\rangle$ and set the function at each $\texttt{rs}^{/2}$ item to be logical OR: a row exists in the result of the projection of the join if there was *some* row in the join that projects appropriately.

The edges of this graph are described by taking advantage of *structural naming* within $\mathcal{I}$; the variables j, x, y, and z correspond to the Datalog variables of the same name.

If we suppose that the input items $\texttt{r}\langle 1, 2, 3\rangle$ and $\texttt{s}\langle 1, 4\rangle$ are given the value of logical TRUE, then we see that $\texttt{join}\langle 1, 2, 3, 4\rangle$ will have value TRUE (being the AND of its parents' values) and that $\texttt{rs}\langle 3, 4\rangle$ will as well (by OR). ◊

Semiring-weighted Circuits Directly generalizing AND/OR graphs, one can use an arbitrary semiring for χ, using its multiplication operator for AND and addition for OR. The resulting family of structures are well-studied in parsing [79, 84] and form the basis of the first version of Dyna [52, 51, 48].

AC-reducing nodes Often, the functions at some (or even all) items of a circuit are AC-reducers, and, as such, have no need to order their parents. That is, these nodes are content to obtain a *bag* of their parents' values. The formalism herein will address the more general case, in which the child cares about which parent has what value, but awareness of AC-reducers is a possibly useful optimization (e.g., §2.3.3.1 takes advantage of such to dynamically transform the circuit).

In the early sections of this chapter, we consider only a *full recomputation* of an item's value in light of any change from any parent. For exact values, no additional work is necessary. When one wishes to act on inexact values (§1.5), ensuring that AC-reduction behaves as a function often requires extra processing. One such mechanism would be to *sort* the bag of parents' values and then reduce them *in sorted order*, thereby ensuring that every invocation of the reduction function obtains the same result. However, supporting *efficient revision* of these aggregates is likely to require keeping the association of parents and values in the case of inexact values; see §2.4.1.

2.1.2.2 The Hypergraph Perspective

A different, but isomorphic, graphical notation for arithmetic circuits comes from consideration of semiring-weighted circuits in which the additive operator is AC-reducing. We present this *B-hypergraph* structure here to orient readers already familiar with it and because we find it a more natural presentation for *generalized* arithmetic circuits (§2.5) and the further generalization to weighted logic programs (§3.1).

A directed **hypergraph** (properly, directed **hypermultigraph** or **hyperquiver**) $\langle \mathcal{N}, \mathcal{E}, \text{src}, \text{targ}\rangle$ is a generalization of a graph in which edges may have multiple sources and targets (the "hyper-" prefix) and in which there may be multiple hyperedges between the same sources and targets (as in a multigraph). Formally, a directed hypergraph is composed of a set of nodes, $\mathcal{N}$ and *an indexing set* for hyperedges, $\mathcal{E}$, with functions src and targ mapping $e \in \mathcal{E}$ to the source set and target set, respectively. For the purposes of this document we work exclusively with directed **B-hypergraphs** [73] with *ordered, finite source collections*; that is, each $e \in \mathcal{E}$ identifies a *single* target node, $\text{targ}(e) \in \mathcal{N}$, and a

```
def COMPUTE($j \in \mathcal{I}_{\mathrm{der}}$) $\in \chi$
  return $e(j)$(LOOKUP $\circ \vec{P}_j$)

def LOOKUP($j \in \mathcal{I}$) $\in \chi$
  $v \leftarrow \mathcal{M}(j)$
  if $v = $ UNK then $v \leftarrow$ COMPUTE($j$)
  maybe $\mathcal{M}(j) \leftarrow v$
  return $v$
```

Listing 2.1: Internals of basic backward-chaining with optional memoization. $\mathcal{M}$ stores values for input items and initially stores UNK for derived items. "maybe $\langle$*block*$\rangle$" indicates that "$\langle$*block*$\rangle$" may or may not be executed, arbitrarily, according to some external policy.

finite tuple of source nodes, $\forall_e \, \exists_{n \in \mathbb{N}, n>0} \, \mathrm{src}(e) \in \mathcal{N}^n$. For brevity, we leave "directed" implied throughout.

The notion of irreflexive path reachability in a graph may be extended to hypergraphs. A node $i \in \mathcal{N}$ can irreflexively reach a node $k \in \mathcal{N}$ if either ① there exists a hyperedge $e \in \mathcal{E}$ with $i \in \mathrm{src}(e)$ and $k = \mathrm{targ}(e)$, or ② there exists both a hyperedge $e \in \mathcal{E}$ and node $j \in \mathcal{N}$ such that $i \in \mathrm{src}(e)$, $j = \mathrm{targ}(e)$, and j can (irreflexively) reach k. A hypergraph is acyclic if there is no node that can irreflexively reach itself. While, for present purposes of considering arithmetic circuits, we restrict to acyclic hypergraphs, we note for future use that the hypergraph formalism also allows *self-loops* in which i is both the target and a source of some edge e.

Extending arithmetic circuits over to this B-hypergraph formalism, we require that every node has in-degree of either zero ($\mathcal{I}_{\mathrm{inp}}$) or one ($\mathcal{I}_{\mathrm{der}}$). (That is, all *expressions* are captured within the *hyperedges*; *common* expressions are manifest as $\mathcal{I}_{\mathrm{der}}$ items with out-degree greater than one.) The graph must, as stated, be acyclic. The interpretation function continues to work only on items; for the moment, hyperedges are merely decorative and are not assigned evaluation functions; $[\![j]\!]$ is defined as in equation (2.1), where, for the recursive case, take $e \in \mathcal{E}$ be the unique hyperedge index such that $\mathrm{targ}(e) = j$ and take $\langle i_1, \ldots, i_{n_j} \rangle = \mathrm{src}(e)$. The relative nomenclature continues to apply: given a hyperedge e, the target $\mathrm{targ}(e)$ is a child of each source $\{\mathrm{src}(e){\downarrow_n} \mid n \in \mathbb{N}\}$. Formally, $C_j = \{\mathrm{targ}(e) \mid e \in \mathcal{E}, \exists_n \, j = \mathrm{src}(e){\downarrow_n}\}$. Letting $\vec{P}_j = \mathrm{src}(e)$ when e is the unique $e \in \mathcal{E}$ such that $\mathrm{targ}(e) = j$, the parents of j are $P_j = \{\vec{P}_j{\downarrow_n} \mid n \in \mathbb{N}\}$. Because ordering of parents is important, $\vec{P}_j$ is the more useful quantity in the following.

2.2 Finite Circuit Inference

2.2.1 Backward-Chaining

We begin with some basic strategies for querying an item's solution value $[\![j]\!]$, based on **backward-chaining** from the item to its ancestors. Backward-chaining is so called because it begins reasoning at a query—an item (or, later, set of items) whose values are of interest—by moving *backwards* towards ancestors, and ultimately roots, whose values are known. We construct a map $\mathcal{M}$ from items to their solution values. $\mathcal{M}$ is known as the **memo table** or **chart**. For each input item $i \in \mathcal{I}_{\mathrm{inp}}$, we initialize $\mathcal{M}(i)$ to $\mathrm{inp}(i)$ ($= [\![i]\!]$). For each derived item $j \in \mathcal{I}_{\mathrm{der}}$, the solution value $[\![j]\!]$ is initially *unknown*, so we initialize $\mathcal{M}(j)$ to the special object $\textsc{unk} \notin \chi$. We may regard the map $\mathcal{M} : \mathcal{I} \to \chi \cup \{\textsc{unk}\}$ as a *partial* map $\mathcal{I} \rightharpoonup \chi$ that stores actual values for only some items—initially just the input items.

We define mutually recursive functions Lookup and Compute as in listing 2.1. 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 $e(j)$, which in turn requires Lookup-s at j's parents. Line 2 of listing 2.1 maps Lookup over $\vec{P}_j$ as if Lookup were a function, rather than an imperative procedure; we trust the reader forgives us our shorthand, as the algorithm is correct regardless of the actual sequence of procedure calls used.

Pure Backward-Chaining The simplest form of backward-chaining simply always recurses through ancestors until Lookup reaches the roots, and never memoizes any derived values. That is, line 7 of listing 2.1 always declines its option to execute, so $\mathcal{M}$ never changes and derived items remain as $\textsc{unk}$. Clearly, Lookup(j) returns $[\![j]\!]$.

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 (a finite prefix of the) Fibonacci numbers, where each item $\texttt{fib}\langle n\rangle$ for $n \geq 2$ is the sum of its parents $\texttt{fib}\langle n-1\rangle$ and $\texttt{fib}\langle n-2\rangle$. Then Lookup($\texttt{fib}\langle n\rangle$) has runtime that is exponential in n, with $\texttt{fib}\langle n-t\rangle$ being repeatedly computed $\texttt{fib}\langle t\rangle$ ($\in O(((1+\sqrt{5})/2)^t)$) 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 2.1. 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 sub-graph. However, the `maybe` keyword in line 7 of listing 2.1 indicates that the memoization step is not required for correctness; it merely commits space in hopes of a future speedup. Lookup($\texttt{fib}\langle n\rangle$) can even achieve $O(n)$ expected runtime without memoizing all recursive Lookup-s: instead it can memoize Lookup-s on a systematic subset of items. For another example, Zweig and Padmanabhan [196] use systematic, structured memoization of a *dynamic* subset of items to solve the arithmetic circuit for the forward-backward algorithm (see, e.g., Rabiner and Juang [150]) in $O(\log n)$ rather than $O(n)$ space, while increasing runtime only from $O(n)$ to $O(n \log n)$.

2.2.2 Reactive Circuits: Change Propagation

Our goal is to design a dynamic algorithm for arithmetic circuits that supports not just *queries* but also updates to the input. It must handle a stream of operations of the form Query(j) for any j, which returns $[\![j]\!]$,[40] and Update(i,v) for $i \in \mathcal{I}_{\text{inp}}$, which modifies $\text{inp}(i)$

[40] One could imagine relaxing the universality of the Query interface, so that only *some* items' values may be demanded. It is possible that such a relaxation could allow for more efficient solvers, especially in the case of streaming inputs: the solver would not need to memorize the stream, merely ensure that it could respond to any of the summary statistics computed by the circuit. Such a change would even allow our finitely-minded solvers of this chapter to work on arbitrary-length streams: despite the infinite nature of the circuit (to accommodate the unknown stream length), only finitely many nodes would be relevant at once.

	Message	Mnemonic	Section
Updates	$\underline{\leftarrowtriangle v}$	Replacement	§2.2.3
	$\underline{\leftarrowtriangle}$	Refresh	
	$\underline{\oplus d}$	Delta	§2.4.3
Notifications	$\overline{\leftarrowtriangle \text{UNK}}$ (abbr. $\overline{\leftarrowtriangle}$)	Invalidation	§2.2.3.3
	$\overline{\leftarrowtriangle; \text{was}\, v}$	Replacement	§2.4.1
	$\overline{\leftarrowtriangle; \oplus d}$	Pure Delta	§2.4.3
	$\overline{\leftarrowtriangle; \text{was}\, v; \oplus d}$	Fully-informative Delta	
	$\overline{\sigma \leftarrowtriangle}$ or $\overline{\sigma \leftarrowtriangle; \oplus d}$	Partially propagated	§2.4.2

Table 2.1: The forward-chaining messages of this chapter. For each, we give the graphical notation, the textual mnemonic, and section of prose introducing this message type. The notation is designed to mesh with figures 2.1 and 2.2; the over-/under-line in the message corresponds to the line shown with each item. As information flows down the page, an update, $i : \underline{\leftarrowtriangle}$, has yet to apply to the value of its associated item, i, while a notification (to be introduced first in §2.2.3.3), $i : \overline{\leftarrowtriangle}$, has already been applied to i's value but is now poised to influence the values of i's *children*.

to $v \in \chi$. (In §2.4.3, we show how to manage *incremental*, "delta" revisions to input items, such as "increase by 1," as well.)

In the case of pure backward-chaining, we only have to maintain the stored input data, as derived values are not stored, but are derived from the input 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 derived 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 (derived) 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 $j \in \mathcal{I}_{\text{inp}}$ is **consistent** iff Lookup(j) $= [\![j]\!] = \text{inp}(j)$, and that $j \in \mathcal{I}_{\text{der}}$ is **consistent** iff Lookup(j) = Compute(j). Notice that un-memoized derived 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 $[\![j]\!]$ as desired. Equivalently, the memo table $\mathcal{M}$ is consistent iff each input memo is correct and each derived 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 (hyper)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.

2.2.3 Pure Forward-Chaining

An alternative solution strategy, **forward-chaining**, reasons *forward* from ancestors that are known towards descendants that are not. It does so by applying updates until all items (or, at least, all items of interest) know their values. We will use it in §2.2.4 to solve the update problem and will iteratively increase the expressive power of our forward chaining

```
def RunAgenda()
  until $\mathcal{A} = \varnothing$
    pop $i : \underline{\leftarrow v}$ from $\mathcal{A}$
    if $v = \text{UNK}$ then $v \leftarrow$ Compute($i$)
    if $v \neq \mathcal{M}(i)$ then  % else discard
      $\mathcal{M}(i) \leftarrow v$
      Apply($i$)

def Apply($i \in \mathcal{I}$)
  foreach $j \in C_i$ do
    $w \leftarrow$ UNK
    maybe $w \leftarrow$ Compute($j$)
    Update($j$, $w$)

def Update($j \in \mathcal{I}$, $w \in \chi \cup \{\text{UNK}\}$)
  delete $\mathcal{A}(j)$
  if $w \neq \mathcal{M}(j)$ then  % else discard
    $\mathcal{A}(j) \leftarrow \underline{\leftarrow w}$
```

Listing 2.2: The core of an agenda-driven, item-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 (not shown, but as in listing 2.1) calls Lookup, Lookup never reenters Compute.

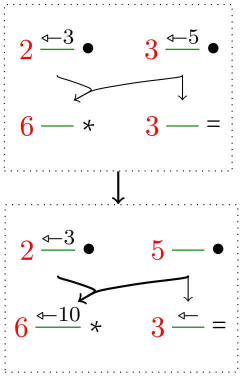


Figure 2.2: An example iteration of the loop in RunAgenda. We apply the update $\underline{\leftarrow 5}$ to the right parent, making its children inconsistent with their parents, and enqueue new updates that will fix the inconsistencies. The thick hyperedge is used to Compute the new value in the replacement update: $10 = 2 * 5$.

algorithms. Table 2.1 shows the full taxonomy of the internal *messages* used within the several forward chaining algorithms we develop; it may be helpful to refer back to this table throughout the next few sections. First, however, 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 (a subset of) their parents. Eventually $\mathcal{M}$ converges to $[\![\cdot]\!]$. Forward-chaining algorithms include natural-order recalculation in spreadsheets [194] and semi-naïve bottom-up evaluation for Datalog [175]. We use the "item-at-a-time" (sometimes also called "tuple-at-a-time") semi-naïve algorithm of listing 2.2. It uses an **agenda** $\mathcal{A}$ that enqueues *future* updates to the chart [104, 51]. $\mathcal{A}$ contains at most one update for each item i, which we denote $\mathcal{A}(i)$, and supports modification or deletion of this update.[41] $\mathcal{A}$ is also a priority queue, supporting a **pop** operation, which selects and removes an arbitrary update. Priorities are *implicit* in the algorithms we describe here; we presume some *orthogonal* mechanism is computing the relative utility of scheduled work (as discussed in, e.g., Vieira et al. [183]). The algorithms herein remain *correct* regardless of prioritization, so long as every update stored into $\mathcal{A}$ is either returned by some pop operation or explicitly deleted

[41] The agenda can be implemented as a simple dictionary. However, using an adaptable priority queue [80] can speed convergence, if one orders the updates topologically or by some informed heuristic [107, 53].

in the solver code shown. The return value of the pop function applied to $\mathcal{A}$ is a pair of an update message u *at* an item $i \in \mathcal{I}$, previously enqueued by a statement of the form $\mathcal{A}(i) \leftarrow u$; we write this pair as $i : u$ rather than $\langle i, u \rangle$.

Our updates are **replacement updates** of the form $i : \underline{\leftarrow v}$ (where $i \in \mathcal{I}$ and $v \in \chi$). Iteratively, until the agenda is empty (the initial conditions will be discussed momentarily), our forward-chaining algorithm pop 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 creates updates at i's children, by **pushing** (enqueuing) an appropriate update $j : \underline{\leftarrow w}$ onto the agenda, by calling Update on line 13 of listing 2.2. This push operation overwrites any previous update to j, so we delete any pending update before pushing (see line 16 of listing 2.2). This iteration back and forth between popping and queueing updates is what makes our algorithm "semi-naïve:" it will potentially recompute some item only given reason to do so. By contrast, a "fully-naïve" algorithm recomputes *all* items until they have all stopped changing.

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 no updates are enqueued at the children of j (though these children may already have updates pending). Ordinarily, w is Compute-ed in line 12 of listing 2.2 when the update is constructed and pushed. But if that line 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 of listing 2.2). Such a **refresh update** $j : \underline{\leftarrow}$ simply says to refresh j's value so it is consistent.

Both kinds of updates 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 overhead 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.

Figure 2.2 shows one step of pure forward-chaining. In our visual notation for circuits, we draw the state of item $\mathtt{i}$, with no pending update, as $i : \mathcal{M}(i) \,\text{—}\, e(i)$, where i (if present) names the item, $e(i)$ is the item's function (or $\bullet$ if $i \in \mathcal{I}_{\text{inp}}$), 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 $i : \mathcal{M}(i) \overset{\leftarrow v}{\text{—}} e(i)$, 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 $\mathcal{M}(i)$. Our textual update notation $i : \underline{\leftarrow v}$ is intended to resemble the drawing.

In any case, the system enforces a straightforward invariant:

Invariant 1 (*Pure Forward-Chaining*)**:** *An inconsistent item always has an update pending on the agenda.*

This update ensures that the inconsistent item will eventually be made consistent. (This is not a biconditional; a consistent item might also have an update pending, e.g., a refresh that is not yet known to be unnecessary.)

The process can be started from any total (i.e., UNK-free) initial chart $\mathcal{M}$, provided that the initial agenda $\mathcal{A}$ is sufficient to correct any inconsistencies in this $\mathcal{M}$. Totality of

$\mathcal{M}$ ensures that the calls to Compute in listing 2.2 will never extend upwards in the circuit beyond Lookup of parent values: the test for $v = $ UNK on line 6 of listing 2.1 is always false, and so Lookup never calls Compute. This ensures that the algorithm need not consider the question of the value of an un-memoized item that has a pending update; this is, in some sense, the key challenge of mixed chaining, which we address in §2.2.4.

Returning to the case of pure forward chaining, $\mathcal{A}$ is always sufficient if it updates *every* item: so the **conservative initialization strategy** defines each $\mathcal{A}(i)$ to be $\leftarrow \mathrm{inp}(i)$ for input $i \in \mathcal{I}_{\mathrm{inp}}$, and either $j : \leftarrow$ Compute(j) or $\leftarrow$ for derived $i \in \mathcal{I}_{\mathrm{der}}$. However, just as listing 2.2 discards unnecessary updates (at line 17), 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 $\in \chi$ for all $i \in \mathcal{I}$. Provided that each function $e(j)$ outputs NULL whenever all its inputs are NULL, each derived j is initially consistent and hence requires no initial update. Emphasizing the “logic” in “logic programming,” updating $\mathcal{M}(j)$ from NULL to non-NULL v may be regarded as “proving j to have value v.”

The user method Query(j) is now defined as RunAgenda(); return Lookup(j). This runs the agenda to completion and then returns $\mathcal{M}(j)$. As for the user method Update(i,v), the user is permitted to call the generic Update of listing 2.2 for input items i and with $v \in \chi$ (i.e., $v \neq$ UNK), 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, though it will potentially visit items irrelevant to that query. Running the agenda to completion is a deficiency we shall correct by stages in §2.2.4.4 and §2.4.2. Moreover, we shall endeavor to opportunistically shed work from the agenda whenever possible; see §2.2.4.3, §2.3.5, and §2.5.4.

2.2.3.1 Aside: Connecting Back to Logic

We can connect the boolean AND/OR-circuits derived from a Datalog program P (recall “AND/OR Graphs,” in §2.1.2.1) back to the logical semantics of (Prolog) programs discussed in §1.4. While we could use any two distinct values in χ to represent truth or non-truth of an item, we will use TRUE and NULL, respectively. In light of the NULL initialization strategy above, the use of NULL for one of the boolean values is convenient as both AND and OR have the requisite property: given only non-truth inputs, both return non-truth, and given only truth inputs, both return truth. We break the symmetry and interpret NULL as non-truth so that the system initializes and behaves as Datalog and Prolog, initially assuming all derived items are not true.[42]

In particular, using the T_P operator, we view our circuit as computing $j \in T_P(\{i \in P_j \mid \text{Lookup}(i) = \text{TRUE}\})$ at each item j with parents P_j. When forward chaining terminates, $\{i \in \mathcal{I} \mid \text{Lookup}(i) = \text{TRUE}\}$ will be a fixed point of T_P. Under the NULL initialization strategy for a cyclic circuit, our solver can be thought of as computing $T_P(T_P(\cdots T_P(\mathcal{I}_{\mathrm{inp}})\cdots))$.

[42] We avoid the term “false” to leave the door open for *three-valued* logics which carry not only explicit proofs of truth but also explicit proofs of falsity, and so may have items which have not yet been proven either way; in such a system, NULL would represent this lack of a proof either way. Recall §1.4.

2.2.3.2 Aside: Storage of $\mathcal{M}$

It may be instructive at this point to contemplate the physical storage of the map $\mathcal{M} : \mathcal{I} \to \chi \cup \{\text{UNK}\}$ (where NULL $\in \chi$). A large circuit may be compactly represented by a much smaller logic program (as will be introduced in detail in §3.1). 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 derived 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. 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.

2.2.3.3 Updates vs. Notifications

The algorithm of §2.2.3 ensures that, when an update to an item i is popped, an update is (considered for) pushing to the agenda for each of i's children, immediately after i's memo table entry is modified. (Changes to i's value that cause its child j's COMPUTE-d value to match its memoized value can, in fact, *discard* updates from the agenda and will not cause an update to be pushed; recall the behavior of UPDATE.) We say that updates (at i) have a **pop-time** effect on the memo table (specifically, at $\mathcal{M}(i)$). However, the immediacy of **propagation**, i.e., of creating updates at child items, is not fundamental to forward-chaining, so long as it, like the updates it spawns, eventually happens. Zooming in on the act of propagation itself, the loop at line 10 of listing 2.2 acts to provide **notification**s to each child j that i's value has *already* changed; these notifications result in updates being queued at child items (line 13 of listing 2.2). We may instead queue these pre-propagation notifications, too, just as we queue updates. From the perspective of the memo table, notifications are seen to have a **push-time** effect: the value changes at the same time as a notification is pushed to the agenda.

Little of significance changes if one rephrases the pure forward chaining algorithm in terms of notifications, rather than updates; pseudocode is given in listing 2.3. The system's invariant would be:

Invariant 2 (*Pure Forward-Chaining with Notifications*)**:** *Inconsistent items always have at least one parent with a notification pending on the agenda.*

Thus, when a notification at i is popped, each child of i will re-COMPUTE its value and, if this differs from its current memoized values, will update the memo table and queue notifications to their children. This variant of forward-chaining keeps the most recent value for any item in the memo table (and available for LOOKUP), unlike the update-based machinery discussed earlier. The approaches have a computational trade-off here, just as there was between replacement and refresh update messages earlier. If PROPAGATE-ion from an item i is

```
def RunAgenda()
  until $\mathcal{A} = \varnothing$
    pop $i : \overline{\leftarrowtriangle}$ from $\mathcal{A}$
    foreach $j \in C_i$ do Update(j)

def Update($j \in \mathcal{I}$)
  $v' \leftarrow$ Compute($j$)
  if $v' \neq \mathcal{M}(j)$ then Apply($j$, $v'$)

def Apply($j \in \mathcal{I}$, $v' \in \chi$)
  $\mathcal{M}(j) \leftarrow v'$
  $\mathcal{A}(j) \leftarrow \overline{\leftarrowtriangle}$
```

Listing 2.3: An alternative presentation of forward-chaining, now using *notifications* rather than the *updates* of listing 2.2. While that algorithm invokes Apply on the popped item i and Update on each of its children, here, both Update and Apply are invoked on *the children* of the popped item.

expensive (e.g., i has many children), it may be advantageous to leave a notification pending while allowing children to see these up-to-date values, in hopes that when propagate-ion does happen, a large fraction of children will not change their values. On the other hand, because children may have seen any memoized value for an item, once enqueued, a notification cannot be removed except by propagate-ion, unlike updates, which can be removed when they are observed to make no change to their item's value.

As might be expected, one need not pick sides. The agenda $\mathcal{A}$ now contains two kinds of **messages**: an update *to* j or a notification *from* j. In fact, each item j may, in general, have up to one of each kind of message pending.[43] Recall that an update to j is graphically displayed *above* the line: $j : \underline{\leftarrowtriangle v}$. A notification from j is drawn as $j : \overline{\leftarrowtriangle}$, with the change displayed *below* the line to indicate that it has already descended through item j. (For the moment, we restrict ourselves to notifications without additional metadata, which merely convey *that* their item's value has changed, though other options will become available in §2.4.) Unlike updates, notifications do not carry the current (i.e., newer) value; if the system is caching it, it will be in the memo table.[44] Were we to permit notifications to carry old values, we would have to modify backward-chaining to check two places (namely, the agenda, for a notification, and the memo table). Even though we will find, in §2.2.4.2, that backward-chaining must probe the agenda for notifications, the possibility of values being cached in two places still seems of little gain. Later, §2.4.1 and §2.4.2 will introduce additional machinery and will, indeed, have backward-chaining *optionally* look for values in notifications, but these values are intended to be *ephemeral* within operations of the forward-chaining components of the system.

Logically, there is a kind of strict alternation between updates and notifications: a notification dispatched from an item propagates to become updates at its children, which

[43] The restriction to at-most-one message of each sort avoids concerns about messages (of the same type and at the same item) re-ordering on the agenda. See §2.4.4.

[44] In an implementation, however, one may wish to augment a notification *during propagation* (but not during storage on the agenda) with the current value, if known. That is, it may be lower overhead, computationally, to inform children of the new value of their parents rather than to make them call back to Lookup their values. Further, an implementation is free to conflate the actual storage of $\mathcal{M}$ and $\mathcal{A}$, if that is useful; some items may have memoized values *only* when they also have messages on the agenda.

then each propagate through their items to become notifications. Either of these phases may be done eagerly (i.e., skipping being queued on the agenda) at any point: updates can apply and propagate to become updates (while updating the *parent* value, as in listing 2.2) and notifications can propagate and apply to become notifications (while updating the *children* values, as in listing 2.3).[45] The pure forward-chaining algorithms above emerge as universal policies to always eagerly perform one or the other.

If a solver uses both kinds of messages (but still keeps $\mathcal{M}$ total), its correctness invariant becomes a disjunct of the two we have given for the systems using just one or the other. Specifically:

Invariant 3 (*Pure Forward-Chaining with Both Updates and Notifications*)**:** *Any inconsistent item j has at least one of an update pending (at j) or a parent with a pending notification.*

Having both an update at j and a notification at its parent i is a little redundant, but not incorrect. In the following development of a mixed-chaining solver with selective memoization, we shall discover that we *must* use both kinds of messages if we wish to be able to remove arbitrary entries from the memo table.

2.2.3.4 Aside: 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 Update-s have caused Query(i) to change. Essentially, they are requests by the driver program that it be treated as a child of i. Continuous queries are also used in databases and in functional reactive programming [54, 37]. 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 of $\mathcal{M}(i)$.

A **snapshot** is a view of the arithmetic circuit that will not be modified by subsequent Update-s to input items, but which is guaranteed to (eventually) converge to a solution consistent with all updates made up to the point at which it was taken. There are several open challenges associated with snapshots, including an efficient mechanism for specifying exactly what updates are or are not to be considered when acquiring a snapshot (i.e., "What time is it, anyway?" in a potentially distributed system) and a notion of **stability** of answers when there exists more than one solution (as there may be after we allow circuits to be cyclic in §2.5).

A solver may offer various levels of support for snapshots. The most primitive offering is *none*: the onus is on the driver program not to issue updates until it has queried all that it might wish to know. An intermediate offering might be the ability to maintain several snapshots, representing points in a *linear* timeline of updates. In this scheme, Query-s would be issued against an existing snapshot or the current state, Update-s would apply to the current state, and there would be a MakeSnapshot procedure which snapshotted the (singular) current state, yielding a snapshot handle. A rather sophisticated offering would

[45] Because the propagation and application of a notification yielding another notification at a child item does indeed update said child item's value, "notifications," especially those handled in this eager way, were sometimes called "push-time updates" in our earlier work, Filardo and Eisner [60]. We have come to view this as a confusing term and avoid this phrase, as well as its counterpart "pop-time update," for which we now just use "update."

be to allow a forked timeline, in which any snapshot can be extended by Update-s, yielding a new circuit state, any of which may themselves be converted into additional snapshots. This sophisticated interface would be useful for *counter-factual exploration* on the part of the driver program.

2.2.4 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 may 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 two directions of chaining have dedicated roles in the system:

- Backward-chaining *computes values for which the memo is missing* (UNK).
- Forward-chaining *refreshes any memos that are present but potentially stale.*

Our pure strategies emerge, then, as consequences of universal memoization policies:

- Pure backward-chaining is the case where *all derived 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 §2.2.3 as a complete but potentially stale memo table.

We develop a hybrid algorithm that can memoize any subset of the derived 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. Computing only values that are needed to answer a given query can reduce asymptotic time and space requirements, a fact exploited by the magic sets technique [152] (see §2.2.4.3).

The essential 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 un-memoized regions of the circuit, so that they reach and refresh any stale memos below. However, if an item j is not memoized, it is no longer sensible to speak of a pending *update* to its value! Any attempt to invoke Lookup(j) will *by supposition* recurse to Compute and therefore see the effect of this update. Thus, we have another invariant of our algorithm:

Invariant 4 (*Mixed-Chaining Updates*)**:** *Updates exist only at memoized items.*

Notifications, however, may be pending at both memoized and un-memoized items. However, the *implicit* change to j must still be propagated to its (visible) descendants, which will be done by propagating these notifications through regions of un-memoized items. When a notification propagates to a memoized item, it will, at least logically, become an update and repair the inconsistency. All told, our solver's primary invariant must be revised once more:

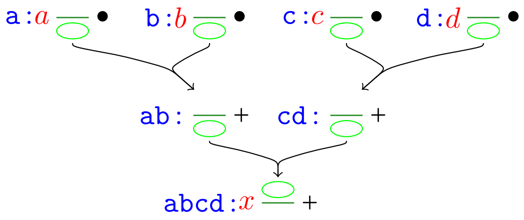


Figure 2.3: An example circuit showing a pictorial interpretation of Invariant 5. If the item `abcd` is inconsistent (i.e., the values returned by COMPUTE(`abcd`) and LOOKUP(`abcd`) are not equal, or, equivalently, $(a + b) + (c + d) \neq x$), then Invariant 5 states that the green circled locations must contain at least one agenda message. Invariant 4 prohibits updates at the un-memoized `ab` and `cd` items.

Invariant 5 (*Mixed-Chaining*)**:** *Any inconsistent item j has at least one of an update pending (at j) or a* visible ancestor *with a pending notification.*

This invariant has a convenient pictorial interpretation, as shown in figure 2.3: if j is inconsistent, a *message* must exist somewhere within j's visible region of the circuit, which spans the slots for an update above its line, for notifications below the lines of its memoized visible ancestors, and for any messages at un-memoized visible ancestors (but we restrict to using only notifications, as per Invariant 4).

2.2.4.1 A Mixed-Chaining Solver with Selective Memoization

The core of our mixed-chaining algorithm, which we call EARTHBOUND ("Encompassing Algorithm for Reactive Truth-maintenance, Homogenizing Backward-chaining, Optional-storage, Updates, and Notifications, as Desired"),[46] is shown in figures 2.4 and 2.5. The listings of EARTHBOUND will use a `maybe` construct to expose multiple strategies; read `maybe` as a conditional statement whose outcome (executing the block or not) depends on some *implicit* or *external* components of the implementation. In general, the merits of executing such optional code—that is, the memory requirements and computational latency of the current operation *and future* operations—depends on the solver's current state and the future workload. In practice, we envision using an adaptive policy for making each `maybe` decision required by the solver; see Vieira et al. [183] for more details. Before getting bogged down in detail, let us provide a high-level synopsis:

- The two listings in figure 2.4 contain the functions used for forward-chaining. Were $\mathcal{M}$ total, these listings would describe a forward-chaining solver using both updates and notifications, as per §2.2.3.3.
- Listing 2.6 shows the backward-chaining core, a generalization of listing 2.1 to mixed-chaining. LOOKUP and COMPUTE remain mutually recursive, but we now distinguish between LOOKUP-s, which begin their work *at* the item in question, while COMPUTE more properly visits each item with a LOOKUPFROMBELOW, which includes a check to see if any

[46] "Truth-maintenance" is another name for reactive solving of (boolean) circuits; similarly, the database literature refers to "view maintenance" for circuits defined using relational calculus (see, e.g., Ahmad et al. [6] and Koch, Lupei, and Tannen [108]). While a reasonable acronym unto itself, the name EARTHBOUND was inspired by Pink Floyd's *Learning to Fly* from *A Momentary Lapse of Reason* [146], because our algorithm spends much of its time calling LOOKUPFROMBELOW, unable to "keep [its] eyes from the circling skies." The identity of the first-person referent of "Tongue-tied and twisted, just an earthbound misfit, I" is left to the reader's interpretation.

```
1  % Enqueue update or notification
2  def UPDATE($j \in \mathcal{I}$, $w \in \chi \cup \{\text{UNK}\}$) $\in \langle\rangle$
3    % Update messages only possible if
4    % existing memo (Invariant 4)
5    % & no notification (Invariant 6)
6    if $(\mathcal{M}(j) \neq \text{UNK}) \wedge (\mathcal{A}(j) \neq \overline{\leftarrow})$ then
7      % optionally, apply now
8      maybe
9        delete $\mathcal{A}(j)$
10       if $(w = \text{UNK}) \vee (\mathcal{M}(j) \neq w)$ then
11         $\mathcal{A}(j) \leftarrow \underline{\leftarrow w}$
12       % else update made no change
13       return
14     % else fall through
15     % (promote update to notification)
16   APPLY($j$, $w$)
17
18 % Apply update, create notification
19 def APPLY($j \in \mathcal{I}_{\text{der}}$, $w \in \chi \cup \{\text{UNK}\}$) $\in \langle\rangle$
20   $\mathcal{M}(j) \leftarrow w$
21   $\mathcal{A}(j) \leftarrow \overline{\leftarrow}$
```

Listing 2.4: Forward-chaining internals. UPDATE and APPLY will be called by the agenda loop (listing 2.5).

```
1  def RUNAGENDA()
2   until $\mathcal{A} = \varnothing$
3    FREELYMANIPULATEM()
4    peek $u$ from $\mathcal{A}$
5    case $u$ of
6      $i : \overline{\leftarrow}$ → PROPAGATE($i$)
7      $\_ : \underline{\leftarrow \_}$ → HANDLEUPDATE($u$)
8
9  % Convert update to notification
10 def HANDLEUPDATE($i : \underline{\leftarrow v}$)
11   delete $\mathcal{A}(i)$ % clear old message
12   $v_{\text{cur}} \leftarrow \mathcal{M}(i)$  % will not be UNK
13   maybe $v \leftarrow$ UNK % in-line FLUSH
14   maybe $\langle v, \_\rangle \leftarrow$ COMPUTE($i$)
15   if $v \neq v_{\text{cur}}$ then APPLY($i$, $v$)
16
17 % Route notification to children
18 def PROPAGATE($i \in \mathcal{I}$) $\in \langle\rangle$
19   foreach $j \in C_i$ do
20     $w \leftarrow$ UNK
21     maybe $\langle w, \_\rangle \leftarrow$ COMPUTE($j$)
22     UPDATE($j$, $w$)
23   delete $\mathcal{A}(i)$ % finished propagating
```

Listing 2.5: Agenda loop. Notifications persist on the agenda during PROPAGATE-ion so that dependencies amongst child items are *marked* correctly (§2.2.4.2).

Figure 2.4: The forward-chaining components of the first version of EarthBound, our mixed-chaining solver. At this point in the development of the algorithm, the only form of notifications used within the system are $\underline{\leftarrow}$; the remainder of table 2.1 will be introduced in §2.4 and will necessitate a few changes throughout these listings.

notification is pending at this parent. This check is necessary for reasons we study in detail shortly (§2.2.4.2).

- Listing 2.7 contains the functions, in addition to UPDATE of listing 2.4, exposed to the user of the solver. As before, QUERY is used to look up the value of an item. A user may call UPDATE(j,w) so long as $j \in \mathcal{I}_{\text{inp}}$ and $w \in \chi$. FLUSH is used to remove an entry from $\mathcal{M}$. FREELYMANIPULATEM is not, by itself, a very *useful* function, but it serves to emphasize that the solver will remain correct under arbitrary memoization policies, including arbitrary removal of memos between other calls into the solver.

Inspection of the possible flows of control within listings 2.4 to 2.6 will reveal that EarthBound is *primarily* a forward-chaining one. Forward-chaining is a consumer of backward-chaining when constructing update objects (see line 21 of listing 2.5 and line 14

```
% Derive $j$'s value from parents
def Compute($j \in \mathcal{I}_{\texttt{der}}$) $\in \langle \chi, \texttt{bool} \rangle$
  foreach $i \in P_j$ do
    $\langle v_i, m_i \rangle$ ← lookupFromBelow($i$)
  return $\langle e(j)(v_{\cdot} \circ \vec{P}_j), \bigvee_{i \in P_j} m_i \rangle$

% Interaction with forward-chaining
def lookupFromBelow($i \in \mathcal{I}$) $\in \langle \chi, \texttt{bool} \rangle$
  $\langle v, m' \rangle$ ← Lookup($i$)
  $m$ ← $(\mathcal{A}(i) = \overline{\leftarrow})$
  return $\langle v, m \vee m' \rangle$

% Get $i$'s memo or derive from parents
def Lookup($i \in \mathcal{I}$) $\in \langle \chi, \texttt{bool} \rangle$
  if $\mathcal{M}(i) \neq \text{UNK}$ then
    return $\langle \mathcal{M}(i), \texttt{FALSE} \rangle$
  $\langle v, m \rangle$ ← Compute($i$)
  maybe
    $\mathcal{M}(i)$ ← $v$
    % Preserve Invariant 5
    if $m$ then $\mathcal{A}(i)$ ← $\overline{\leftarrow}$
  return $\langle v, m \rangle$
```

```
def Query($i \in \mathcal{I}$)
  RunAgenda()
  % Agenda is empty, so no mark on $v$
  $\langle v, \texttt{FALSE} \rangle$ ← Lookup($i$)
  return $v$

def Flush($j \in \mathcal{I}_{\texttt{der}}$)
  $\mathcal{M}(j)$ ← UNK
  % Preserve Invariant 4
  if $\mathcal{A}(j) = \underline{\leftarrow \_}$ then $\mathcal{A}(j)$ ← $\overline{\leftarrow}$

def FreelyManipulateM()
 done ← FALSE
 until done
  foreach $i \in \mathcal{I}_{\texttt{der}}$ do
    maybe _ ← Lookup($i$)
    maybe Flush($i$)
  maybe done ← TRUE
```

Listing 2.6: Backward-chaining internals. UnlikeListing 2.7: User interface methods. Update, from their counterparts in listing 2.1, these return *pairs.*listing 2.4, applied only to items in $\mathcal{I}_{\text{inp}}$, is also avail- The first element of these pairs are the values asable to the user. FreelyManipulateM is called expected; the second component is used to man-from within the earlier solver listings, as well. age "premature visibility" of values obtained having crossed notifications; see §2.2.4.2.

Figure 2.5: EarthBound backwards-chaining components and external functions. Like figure 2.4, these listings will be revised in §2.4.

of listing 2.5), while backward-chaining never, even indirectly, re-enters forward-chaining.

Our algorithm also takes the opportunity to exploit notifications even for memoized items. In the old listing 2.2, Update(j,w) always enqueued $j : \underline{\leftarrow w}$ for later. Our new Update(j,w) in listing 2.4 can still choose that option provided that j is memoized (line 6 of listing 2.4), but its default is to Apply the update immediately (line 16 of listing 2.4). If so, it pushes only the notification $j : \overline{\leftarrow}$ and there is no need to Apply any change to j when this pops from the agenda. What *does* still happen at pop time is propagation: it is not until we pop an update *or* a notification to j (line 5 of listing 2.5) that we visit j's children (line 19 of listing 2.5).

Invariant 6: *Merely for simplicity of exposition, and not out of any correctness concerns, our algorithm as presented enforces that items have at most one of an update or a notification pending. (Before, in §2.2.3.3, we permitted at most one message of each type; we are now strengthening our requirements.)*

Thus, our agenda stores at most one message of any type for any given item. Update only creates an update at line 11 of listing 2.4, gated by a check that there is no notification pending at line 6 of listing 2.4, as well as the check to ensure that a memo exists as per Invariant 4. If either check fails, Update defaults to invoking Apply to create a notification.

As might be expected, there are reasons to prefer notifications or updates in different situations. Updates, and their attendant memos (recall Invariant 4) serve to isolate descendant regions of the graph from ancestral regions; thus, they may serve to hide churn in parents' values from children. Any Lookup(j) of an item with a pending update *necessarily* returns a known-to-be-stale value, and so any created descendant memos are *necessarily* out of date with respect to their ancestors (above their visible ancestors). On the other hand, notifications ensure fresher lookup results as $\mathcal{M}(j)$ has already been updated to a new value (or invalidated, set to unk). (However, to ensure correctness in all cases, there is a need to "mark" these prematurely-seen values; we discuss the point in detail in §2.2.4.2.)

What happens if line 21 of listing 2.5 is optionally skipped (so that w = unk)? Then the resulting Update is a refresh update as before (§2.2.3). However, if processed at push time to create a notification (i.e., if control flow reaches line 16 of listing 2.4), it is an **invalidation** that deletes a memo instead of correcting it. Propagating invalidations *as such* (i.e., without invoking Compute during their handling) can clear out stale portions of the chart at lower computational cost than computing the new values. Separately, the Flush method can also be called by the user or by FreelyManipulateM to delete individual memos without the need to propagate. Flush preserves Invariant 4, that updates exist only at memoized items, by promoting any update to a notification at flush time.

Like the forward-chaining algorithm, the hybrid algorithm may start from any initial chart $\mathcal{M}$—but derived 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 sufficient updates to correct any inconsistencies in the initial chart. Since un-memoized, derived, unk items are always consistent by definition (§2.2.2), the initial agenda never needs to have updates for them. For example, the **unk initialization strategy** initializes just as in backward chaining (§2.2.1), with input items set correctly and everything else initially unk, and so may use an *empty* initial agenda.

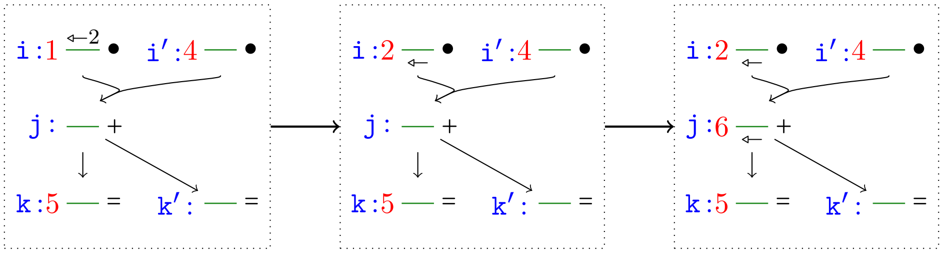


Figure 2.6: Backward-chaining may need to enqueue notifications (§2.2.4.2). After the top left update (at `i`) is applied, it becomes a notification $\overline{\leftarrow}$. After this occurs, `Lookup(j)` is called, perhaps as part of backward chaining `k`' or in response to a user `Query`, and chooses to memoize an up-to-date result of 6. Because backward-chaining encountered a $\overline{\leftarrow}$, the memoization enqueues another $\overline{\leftarrow}$ at `j`, which ensures that its child, `k`, will later be updated from 5 to 6. Were this notification not enqueued at `j`, there is risk that `k` would remain forever inconsistent, because the propagation of the notification from `i` may be discarded as it does not revise the memoized value of `j`, 6.

2.2.4.2 Premature Visibility, or, Marked Results from Backward-Chaining

We noted before that forward-chaining is a consumer of backward-chaining, and that backward-chaining never *dequeues* from the agenda. However, careful readers may have noted that `Lookup` will, on occasion, *enqueue* to the agenda (at line line 21 of listing 2.6) based on the somewhat mysterious second components of the tuples being returned. Let us now explain the mystery. Suppose that the item `i` processes an update and queues a notification at itself, thus making its visible descendant `k` inconsistent, and that `j` is an initially un-memoized intermediate item on an `i`-to-`k` path.

In the course of the solver's execution, it may invoke `Lookup(j)` and that may memoize (at line 19 of listing 2.6) this result, all before the notification from `i`. In such a case, subsequent handling of the notification from `i` may `Compute` `j`'s new value and compare it to the memo, either when the notification from `i` is `propagate`-d to `j` (line 21 of listing 2.5) or when a (refresh) update at `j` pops and is fed to `handleUpdate`. This call to `Compute(j)` will observe *the same answer* as the `Compute(j)` that computed the memo, earlier. 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 value at `j` has been brought into consistency with its visible ancestors. The correct behavior is illustrated in figure 2.6. This notification reflects the past update to `i`; it restores the solver's consistency invariant (Invariant 5), and it will propagate down to `k` as desired. Such a notification must be enqueued when memoizing any item j such that `Compute(`j`)` (possibly through recursion) called `Lookup` on some item that had a notification on the agenda. The functions in listing 2.6 return (as the second element in the tuple) a *mark* that is `true` if this condition holds, and enqueue the required notification at line 21 of listing 2.6. Specifically, `lookupFromBelow(`j`)` consults the agenda to test for the existence of a notification at j, ensuring that the flag is set in its return if it finds one. `Lookup` returns `false` if it encounters a memo or, otherwise, simply passes the flag through from `Compute`. `Compute` returns the logical `or` of all parent's marks. We call a result from backward-chaining whose associated mark is TRUE a **marked value**.

A careful reader will note that, within HANDLEUPDATE, the solver may re-COMPUTE the value carried by an update, even when that value is not UNK, at line 14 of listing 2.5. This is not essential to the system's operation, but is a possible optimization, exposing a *newer* value to the children of the item being updated i, by incorporating notifications pending at its visible ancestors. Because at this point the solver is constructing a notification anyway, whether or not the result of COMPUTE-ation is marked is irrelevant. Similarly, marks are discarded during construction of *updates*, at line 21 of listing 2.5.

2.2.4.3 Efficiency: Obligation

Our hybrid algorithm naturally addresses the challenge first given in §2.2.2: backward-chaining with optional memoization was a promising algorithm, but did not work when the input could be UPDATE-d. Left to its own devices, forward chaining will materialize (and maintain, in the presence of updates) all memos, which may require much more work than just answering the questions the user wishes to know. This could be a large, and unnecessary, expense.

For example, if *no derived items have been memoized*, then change propagation should be completely unnecessary—this is the pure backward-chaining case of §2.2.1—and yet our algorithm will visit all descendants of an UPDATEd item! Moreover, while pure forward-chaining, can stop propagating (discard the update) at a child whose value is unaltered, for an UNK child the former value is unknown, so we must, conservatively, keep propagating. This possibility of useless propagation (to descendant-closed segments of the circuit that contain no memos) suggests maintaining some summary information of the descendants of a given item that could inform PROPAGATE of its responsibilities.

A well-known approach for obtaining hybridized behavior in circuit solvers is to *emulate* backward-chaining within a forward-chaining system. (The asymmetry is not arbitrary: forward-chaining is complete for a larger class of circuits than backward-chaining, as we will see in §2.5.) The technique used is called "magic sets" [16] (subsequently generalized to "magic templates" [152]). Given our circuit formalism, we view magic sets as *doubling* the size of the circuit and reversing the arrows in the added copy (and assigning different functions to these copied items; we defer details momentarily). This second copy of the graph models information flowing *backwards* in the original circuit and will be used to track whether items are *needed* ("charmed," in the original language of magic sets). To make a query of a node, one would assert that it was needed (i.e., change the *input* item corresponding to need of the query) and then use *forward-chaining* to compute, within the copy of the circuit, the set of ancestors that must, certainly, also be needed. Then, *forward-chaining again* would flow values from ancestors to descendants, but would not consider updating items that are not needed.[47]

[47] The original discussion of magic pertained to Datalog programs (and was later extended to Prolog), which can be, for present purposes, thought of as circuits having the AND/OR structure mentioned in §2.1.2.1. In particular, the items j of a Datalog program have structure to their valuation functions $e(j)$: for AND nodes, if any P_j is assigned the value NULL, then so is j. When marking items as needed, then, one could mark *only the first parent* of a needed AND item as needed (as well as all parents of a needed OR item). If forward-chaining discovers that this first parent is TRUE (or, more generally, non-NULL), then one considers the second parent needed and replays the *upward* pass of marking more items as needed, before continuing

Generalizing this notion to be reactive gives a measure of **obligation** within our solver. We define a function of a parent-child *edge*, $\mathrm{obl}(i,j)$, which is true when item i is obligated to propagate a notification to its child $j \in C_i$. We replace the `foreach` on line 19 of listing 2.5 with "`foreach` $j \in C_i$ `where` $\mathrm{obl}(i,j)$ `do`," using this (temporarily) oracular function to guide PROPAGATE-ion's hand. It is conservatively safe to define $\forall_{i,j}\, \mathrm{obl}(i,j) = \mathrm{TRUE}$; this is potentially inefficient, but certainly not wrong.

As an initial position, we might limit propagation along edges to cases when the message so sent may eventually reach a memoized descendant.[48] Certainly, any messages sent from an item with no visible, memoized descendants achieve no effect. Thus, we could imagine constructing, for a given circuit $\mathcal{C}$, its *obligation circuit*, $\mathcal{C}_{\mathrm{obl}}$, a *boolean* circuit that determines presence of visible, memoized children. Roughly speaking, $\mathcal{C}_{\mathrm{obl}}$ has the same topology as $\mathcal{C}$ but with the edge direction reversed: the item i, in $\mathcal{C}_{\mathrm{obl}}$ is TRUE if $\mathcal{M}(i) \neq \mathrm{UNK}$ or if some $j \in C_i$ is, in $\mathcal{C}_{\mathrm{obl}}$, TRUE (i.e., the *children* of i in $\mathcal{C}$ are some of the *parents* of i in $\mathcal{C}_{\mathrm{obl}}$; essentially, $\mathcal{M}(i)$ is also a parent of i).

Transitive closure, however, is not quite the whole story. We can be even more precise about determining obligation, should we wish. Specifically, in the recursive definition, i is *not* obligated to its child j if there is a refresh update at (memoized) j. In this case, j and its descendants are guaranteed to get refreshed anyway (if they are found to be stale when the refresh pops from the agenda), so it is not necessary to propagate messages to j from i or its ancestors. Below the line, if there is a notification pending at i, then i is not obligated to any of its children: any changes to i before the notification propagates will be taken into consideration by children automatically. These changes are easily reflected in $\mathcal{C}_{\mathrm{obl}}$, by adding $\{\mathcal{A}(i) \mid i\}$ as input items and wiring them in appropriately.

Example 10: Consider the (admittedly artificial) case of a circuit formed of a (finite) chain of items $\{x_i \mid i \in \mathbb{N}_1^n\} \cup \{x_0\}$, in which each x_{i+1} $(i \geq 0)$ has x_i as its sole parent and each x_i $(i > 0)$ simply copies its parent's value. Take $x_0 \in \mathcal{I}_{\mathrm{inp}}$. If no $\{x_i \mid i > 0\}$ is memoized, then a driver-induced change to the value of x_0 need not do *any* computation, as x_0 is not obligated to x_1. If, on the other hand, k is the largest $\mathbb{N}_1^n$ such that x_k is memoized, the solver must push changes at least as far as the memo for x_k, and may then stop (as no further memos can change); each x_i is obligated to x_{i+1} when $i + 1 \leq k$. However, suppose that some x_j, with $0 < j < k$, while memoized, has a refresh update pending. The solver, now, need only propagate changes as far as $x_{j'}$, the child-most item *between* (exclusively) 0 and j with $x_{j'}$ memoized. While there are memos "beyond" $x_{j'}$ (at x_j, in particular, but possibly others as well), they are certain to be refreshed, if necessary, when the update at x_j is popped and processed and (possibly) results in a notification at x_j to be further considered. That is, $x_{j'}$ is not obligated to $x_{j'+1}$ (due to the refresh update at x_j and the lack of memos between $x_{j'}$ and x_j), even though x_j may be obligated to x_{j+1} due to the memo at x_k (but may also be relieved of its obligations by other messages on the agenda). ◊

Relative to the magic sets approach, despite the increased workload of maintaining $\mathcal{C}_{\mathrm{obl}}$, our mixed solver is potentially more economical with space. Magic sets materializes

to forward chain on the, now larger, set of needed items. One would propagate an update to a needed item only after all of its parents have been shown to be needed by this procedure.

[48] Strictly, memoized or with an active continuous query à la §2.2.3.4; let us, for present purposes, assume that all continuous queries always ensure that the queried item is memoized.

the sets of needed items for the entirety of a computation. We *directly* perform backward chaining with a depth-first stack, eliminating the need to store these sets beyond their use.

Speaking of maintaining $\mathcal{C}_{\text{obl}}$, we could maintain it exactly using $(\mathcal{C}_{\text{obl}})_{\text{obl}}$, or by falling back to a *cheaper* obligation tracking strategy. Recall that *overestimating* obligation will not lead to *errors*, merely inefficiency, so we can tolerate, in $\mathcal{C}_{\text{obl}}$, unlike in $\mathcal{C}$, one-sided error. Perhaps we simply assume that all obligation relations are TRUE unless we explicitly store that they are not, so upon finding an UNK in the memo table of $\mathcal{C}_{\text{obl}}$, we do not recurse but return TRUE. Another possibility is to uses a memoization and flushing policy such that the memoized items always have memoized parents, so that there is no need for the recursive rules in $\mathcal{C}_{\text{obl}}$. It may also be possible to find *coarser* approximations of $\mathcal{C}_{\text{obl}}$, wherein one item (safely) approximates several obligations. There seems to be a rich space of options here.

We note, in passing, that it may even be possible to tolerate two-sided error in obligation, so long as one is assured of coming to recognize the erroneous beliefs of non-obligations. While we are unsure what sufficient preconditions could ensure this eventuality, the eventual *fix* would be charmingly simple: queue a refresh update at the child which may have missed messages and let the existing machinery take care of the rest.

2.2.4.4 Responsiveness: Selective Propagation and Convergence

Our current algorithm calls RunAgenda at the start of every Query, which brings all stale memos—including those that are not relevant to this query—up to date. This can be especially inefficient for cyclic or infinite circuits. We would prefer to propagate only the currently relevant updates, as in Kangaroo [135].

As a first definition, an item is **converged** with respect to the current state of the input items if it and all of its ancestors do not have messages pending on the agenda. In such a state, an item can be safely Query-ed, even if there are other messages still pending on the agenda. Beyond passively detecting convergence, we could selectively pop messages on the agenda until our target Query was converged. Because this may involve queueing messages to many items which are not ancestors of our target Query, this strategy may not represent as large of an improvement to responsiveness as one might wish.

We can be more precise in our definition of convergence by observing that only *obligated* ancestors actually matter. If an item j has released its parent i from obligation, then a message pending at i must not influence its value. We will be further able to refine this definition in §2.4.2, and in so doing, recover the desired, but at present missing, efficiency gains mentioned in the last paragraph.

2.2.4.5 Responsiveness: Towards Concurrent Solving

The algorithm we have presented so far is *single-threaded*, with an *event dispatch loop* structure built around the agenda. However, we would like to be able to dispatch multiple separate Compute-ations or propagate-ions at once.[49]

[49] For realizability on modern hardware, one hopes for a SIMD-style relation between the work units grouped together. Our work in §3 may be realizable this way.

In general, one could imagine using Software Transactional Memory to ensure that multiple threads executing our mixed solver made coherent updates to the memo table and agenda. Such a transaction would isolate all necessary backward-chaining and all attendant reads of $\mathcal{M}$; as part of transaction commitment, $\mathcal{M}$ would be updated and any new messages for the agenda would be queued for subsequent dispatch. In the extreme, one could imagine starting a transaction for every message *pushed* to the agenda and permitting them to resolve however they may; in practice, one is likely to want at least some semblance of scheduling and synchronization between transactions, so that the system can, for example, attempt to bring certain parts of the circuit into convergence before others.

In the next section, as part of a general effort to improve expressiveness of the solver, we consider, but do not implement, a *message-passing* framework. In this framework, the agenda is removed from its central, synchronizing role in the system and messages between agents are asynchronous but ordered. Such a framework may be a useful starting point for a *non-transactional* approach to parallelism, as it is not necessary to globally synchronize across all agents, just locally on their message queues.

2.3 Items as Stateful Agents

We can think of our algorithm to date as a routing framework for messages between items: when forward-chaining, messages are notifications of changed values; when backward-chaining, messages indicate demands for values and responses thereto. The flexibility of this algorithm is limited by the black-box nature of the $e(j)$ functions of the definition of arithmetic circuits. These functions are presumed to be capable only of responding to demands for the value (of j) if they are provided the values of all parent items. While there certainly are functions that truly do depend on all of their arguments in all circumstances and in opaque ways, there are several special cases worth considering for computational performance.

Canonical examples of such special cases are selective operators, such as AND and OR. In procedural languages, these operators are often given "*short-ciricuit*" semantics; if, for example, the result of evaluating the first argument to AND is FALSE, then the second argument will not be evaluated at all.[50] Other examples are available as well. When comparing the sum of natural numbers, $a_1 + a_2 + \cdots$, to a fixed natural k, one can stop demanding additional a_i items as soon as the *running sum* is greater than the threshold k: the sum will only further increase and will not change the comparison result. As a lemma, asking if a CONS list is longer than a given length can similarly stop processing if a prefix of that size has been observed: it does not matter how much more of the list is analysed, it will never be found to be *shorter*.

In *reactive* settings, one sometimes wants to avoid picking a *fixed* short-circuit ordering on parents. Having found a TRUE parent, an AND item should be able to express

[50] In a side-effect free language, this is purely a matter of *implementation* but may improve the wall-clock time taken to evaluate a pure expression. (Time is, as a reminder, generally considered to be irrelevant to pure programs.) However, some side-effectful languages even encourage use of this short-circuiting mechanism for side-effect management, so that the statement "`x or launchMissiles()`" will have the effect of launching the missiles when `x` is `FALSE`.

Backward-chaining §2.3.1	In	C	`@lookup`$\langle\rangle$	Request agent's value
		P	`@value`$\langle i \in \mathcal{I}, v \in \chi \rangle$	Inform agent of parent's value
	Out	P	`@lookup`$\langle i \in \mathcal{I} \rangle$	Demand value of parent
		C	`@valueIs`$\langle v \in \chi \rangle$	Reveal own value
Forward-chaining §2.3.3	In	C	`@lookup`$\langle\rangle$	Get agent's value
		P	`@notifyFrom`$\langle i \in \mathcal{I}, v \in \chi \rangle$	Parent i now has value v
	Out	C	`@valueIs`$\langle v \in \chi \rangle$	Reveal value
			`@notify`$\langle\rangle$	Item's value has changed
Mixed-chaining §2.3.4	In	C	`@lookup`$\langle\rangle$	Request agent's value
		P	`@value`$\langle i \in \mathcal{I}, v \in \chi, m \rangle$	As above, but with mark m
			`@notifyFrom`$\langle i \in \mathcal{I}, v \rangle$	As above, but $v \in \chi \cup \{\text{UNK}\}$
	Out	P	`@lookup`$\langle i \in \mathcal{I} \rangle$	Demand value of parent
		C	`@valueIs`$\langle v \in \chi \rangle$	Reveal value
			`@notify`$\langle\rangle$	Item's value has changed
Metadata	In	-	`@flush`$\langle \cdots \rangle$	Flush state (§2.3.4.1)
	Out	-	`@obligate`$\langle \sigma \subseteq P_i \rangle$	Set obligations of item i (§2.3.5)

Table 2.2: Inter-Agent Message Types. The third column, containing P (parent) or C (child) annotations, describes the *source* of input messages and the *destination* of output messages, after handling and transformation by the containing framework. Metadata messages are exchanged directly with that framework itself.

its disinterest in all other parents until that TRUE changes. Similarly, a sum-compare item, having found a subset sum that determines the outcome, can should be able to indicate its indifference towards other elements of the sum. We revisit obligation in §2.3.5.

To improve the expressivity of our algorithm, we replace $e(j)$ with a richer interface: a *stateful*, synchronous agent or **actor** [5]. Agents accept a message from a set I and produce messages from a set O; along the way, they may update some internal state. We model the type of the agent state for item j as (isomorphic to) a least solution S_j to the equation $S_j = I \to \langle S_j, \wp O \rangle$. Agents consume input messages serially but may emit several output messages at once, which may be processed in any order. We will write all messages as terms, using a leading `@` sigil to indicate that they are not part of $\mathcal{H}$. The full taxonomy of messages, together with the sections containing definitions, is shown in table 2.2.

We assume that agents for derived items can be *restarted* at any point by forgetting their state and using an initial state object returned by a given function initState $\in \Pi_{j \in \mathcal{I}_{\text{der}}} S_j$. Thus, we can selectively memoize agent state as well as item values: if we find ourselves sending a message to the agent for j without an associated state object, we may conjure a new one into being and pass it the message. One may expect an initial burst of output messages while this new agent orients itself. Agents for input items are particularly simple, merely returning the input value; they are not able to be reset, just as input items could not be flushed.

The framework into which we embed our agents in this section should ensure that messages are delivered *reliably*, *exactly once*, and *in-order* between every pair of sender and receiver. That is, if an agent for item j issues one message m_1 to item k and then later another m_2, the agent for k must receive m_1 before m_2, if it receives m_1.[51]

[51] How can k not receive m_1 since message delivery is reliable? It is possible that the agent for k is reset between delivery of m_1 and m_2, so that the new agent observes only m_2. Agents must be prepared to deal

In §2.2.4.3 we introduced a notion of obligation which was built up from the memo table. An item i was obligated to its child j if the value of j was memoized or, in turn, j was obligated to any of its children. We slightly revise the definition here: if the framework is holding an agent state for j, *even if that state could not immediately return j's value*, that is sufficient to obligate its parent i. The recursive aspect of the definition remains in place. See §2.3.5 for more discussion.

2.3.1 Pure Backward-Chaining as Message Passing

The simplest input message is `@lookup`$\langle\rangle$, which requests a `@valueIs`$\langle v\rangle$ output. An input item's agent responds immediately, as might a derived item's agent if it is caching the value, à la the memo table of old. However, if the agent for a derived item j does not know the value, it must emit demands for parents' values, a set of `@lookup`$\langle i\rangle$ messages. While the prior algorithms *assumed* that all items needed all parents' values, this message-passing framework waits for an item's agent to explicitly demand a parent. The framework should respond by storing, internally, that this agent is waiting on the requested parents and should issue (or queue to be issued later) `@lookup`$\langle\rangle$ messages to these agents. Eventually, those agents, possibly after demanding values of *their* parents (and so on), will yield a `@valueIs`$\langle v\rangle$ output message, which should be forwarded, as `@valueIs`$\langle i, v\rangle$, to all agents blocked on i's value. Eventually, after possibly several rounds of `@lookup` and `@valueIs` messages, the agent for j should emit its own `@valueIs`$\langle v\rangle$. Upon receipt of such a message, the framework should forward v to any agents that had requested j's value.[52]

The behavior of listing 2.1 is recovered if we make a few assumptions of the framework and agents. ① `@lookup`$\langle j\rangle$ messages from k are handled in a depth-first order: one j is selected and its agent is sent a `@lookup`$\langle\rangle$ message. Any `@lookup`$\langle i\rangle$ messages it emits will be handled in full before returning a value to k. ② In response to a `@lookup`$\langle\rangle$ message, an agent that does not already know its value emits `@lookup`$\langle i\rangle$ messages for all of its parents. ③ Given an `@value`$\langle i, v\rangle$ message for each parent, an agent will emit a `@valueIs`$\langle v\rangle$ carrying its own value.

The move to a message-passing, demand-driven backward-chaining system enables a few new features, which are worth discussing briefly. ① A derived item j may now reveal its value without having demanded the values of all of its parents, if the subset queried so far is sufficient to determine its value. Typical examples of this behavior occur when there are absorbing elements of $e(j)$, e.g., multiplication by zero, or conditional behavior, such as $e(j)(\langle c, t, e\rangle)$ encoding the behavior of "`if` c `then return` t `else return` e." Previously, both t and e would have been demanded, in addition to c, while now j's agent may first demand c *and then* demand only the one of t or e relevant to the result. Thus, the behavior of the system is now *value-dependent*. ② Moving away from depth-first search for parents'

with this scenario: agents may get answers to questions they have not (yet) asked! All messages discussed in this section are *idempotent*, so this, for the moment, is not much of a concern.

[52] There is some finesse required for discarding agent state, in that a new state of the agent must be synchronized with the framework as a whole. If, for example, the agent's item's value was under demand when the old state was discarded, the new state may receive apparently unsolicited `@value`$\langle i, v\rangle$ messages. When instantiating a new agent for a presently-demanded value—that is, one on which some other agent is blocked—the framework should synthesize a `@lookup`$\langle\rangle$ message.

values may allow for better scheduling of work (and use of the memo table). The framework is free, for example, to operate in a *breadth-first* way to actively search for opportunities for reuse of derived values, rather than memoize as best it can at the moment and hope for cache hits later.

2.3.2 Pure Forward-Chaining as Message Passing, Take 1

Forward-chaining is already more overtly a message-passing system. However, the algorithms of §2.2 treat the items' evaluation functions as black boxes that are capable only of computing their value when given the value of parents. Thus, the forward-chaining algorithms' agendas carry messages not for items' consumption, but for *the algorithm's*.

Precious little changes from listing 2.2, under the (relatively minimal) assumption that agents, having computed a value in response to `@lookup`$\langle\rangle$, will never forget this value and need to engage backward-chaining thereafter. Updates, which once replaced *values* of items, now correspond to *replacements of agent state*. That is, we can construct a new agent for j in light of a notification from its parent i, demand the value of this new agent, and interact with this new agent through its backward-chaining, and *not immediately replace the memoized state* with the resulting, new agent state for j. By analogy with Invariant 4, we continue to insist that there may be at most two agent states associated with a given item: the current one and the pending one. If a subsequent notification is propagated before the agent state replacement has occurred, the intermediate agent state is discarded (as with updates). When the agent state is, ultimately, replaced, the framework must enqueue a notification, to be routed to children.

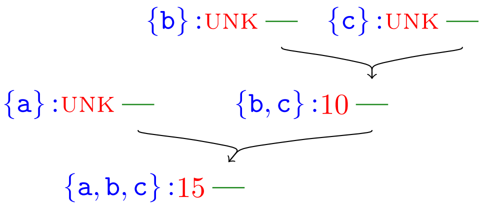
2.3.3 Pure Forward-Chaining as Message Passing, Take 2

We can improve reactivity of the system if we assume agents with increased capabilities. That is, we assume that (at least some) agents would benefit from being told *which* parent had changed, as well as *how* it had changed (i.e., the new value). We use as our messages the *notifications* from the taxonomy of §2.2.3.3. Our input messages are thus `@notifyFrom`$\langle i, v\rangle$ (i.e., $i : \overline{\leftarrowtriangle v}$ being delivered to the recipient child of i) and outputs are `@notify`$\langle\rangle$ (i.e., enqueueing $\overline{\leftarrowtriangle}$ at the originating item).[53] If the notification does not, in fact, produce a change in value, the agent need not emit `@notify`$\langle\rangle$ (that is, it may respond with a new state and *no* messages). As before, we assume that an agent will retain its value (i.e., can immediately respond to `@lookup`$\langle\rangle$ with `@valueIs`$\langle v\rangle$) at any point; in fact, the agent is now required to ensure that it emits a `@notify`$\langle\rangle$ message before it can report a `@valueIs`$\langle v'\rangle$ with v' different from the last v returned. Here, rather than being caused by a demand issued by a child agent, `@lookup`$\langle\rangle$ is issued by *the framework* when it goes to propagate a notification to those children.

Assuming we immediately replace the agent state with the one given with the `@notify`$\langle\rangle$ output, we should immediately queue the notification (i.e., push it to the agenda)

[53] The v component of `@notifyFrom` messages differs from a strict reading of the taxonomy of §2.2.3.3, but it follows the suggestion given therein that *during propagation* it is sensible to augment a notification with current value. We do so here to temporarily avoid giving agents the ability to LOOKUP parents' values, maintaining a simpler exposition; they will gain such abilities when we consider mixed chaining.

Figure 2.7: A small aggregation tree. Recomputing the sum of the values for the items $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$ in light of a notification from `a` will require no `@lookup` messages, as the result can be recomputed from the existing cached sum of `b` and `c` and the notification from `a`, while an update from either `b` or `c` will require a `@lookup` of the other and of `a` as well.



for delivery to children. As in listing 2.2, such `@notify`⟨⟩ messages remain queued, "at item i," within the message-passing framework (i.e., on the agenda) until they are PROPAGATE-d to child agents as `@notifyFrom`$\langle i, v\rangle$ messages. Any `@notifyFrom` emitted by an agent before a prior one has propagated replaces it, as before.

Updates, in this new world of increasingly clever agents, are still new agent states (as in §2.3.2). However, because agents now may be maintaining metadata to assist with update propagation (i.e., they are not merely managers of ephemeral backward-chaining state and, ultimately, stores of values), subsequent messages to the item from above (i.e., `@notifyFrom`) must be routed to the state within the most-recently-queued pending update object,[54] while messages from below (i.e., `@lookup`) must continue to go to the memoized state. That is, unlike listing 2.2, construction of an update no longer discards any existing update, but continues to evolve the state contained therein. As agent states may be rather large, having evolved the item state to a new value within the update, it may make sense to replace the memoized agent state with one that *solely* caches the memoized value, as it will never again see any message other than `@lookup`⟨⟩.(That is, now that an *updated* agent state is stored in the agenda, *that agent state* will recieve messages from parents, while the state in the memo table will continue to recieve messages from children. The only message from children is `@lookup`⟨⟩.)

At present, `@notify` messages do not carry information beyond their presence. Several extensions of §2.4 will extend notifications with additional metadata, but there will be no need to cache agent states within these messages, unlike updates.

2.3.3.1 Dynamic Local Circuit Transforms

A common use case of these kinds of increasingly-capable agents is to encapsulate dynamic, local transformations of the computational circuit, which enable replacements to be propagated without need of all parents' values (i.e., j can update its value without calling COMPUTE(j)). For the moment, we are constrained by our story so far to consider only replacements of values by other values; later, in §2.4 we shall see other options which will even further increase the utility of such transformations.

Example 11: Suppose that $e(j)$ is the *sum* of its (many) parents. We can reduce the work of propagation by maintaining additional data structures, such as an **aggregation tree**, which stores at intermediate nodes the sum of the values at child nodes, with the value of

[54] Of course, like all prior algorithms, this one does not need to keep more than one update object per item; we can always replace any pending update with a newly computed one, rather than keep both. If we *did* keep both, we should be careful to apply them in FIFO order, and to evolve only the chronologically latest in light of messages from parents.

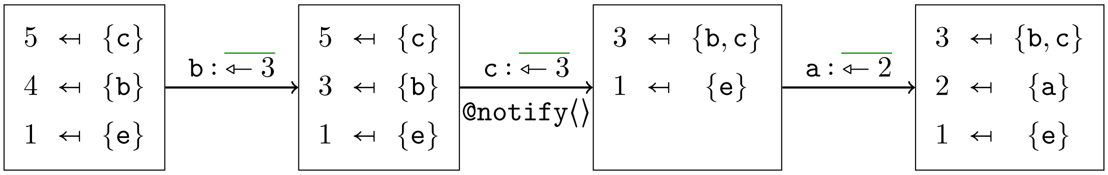


Figure 2.8: A small example of an associative map for selective aggregation. Here, the agent is tracking up to the three largest values it has seen and (some of) the parents currently having those values. Each transition is caused by receipt of a notification, shown above the line. The agent always updates its internal state, even when its reported value does not change (and so no `@notify` message is sent).

each P_j being found at a leaf.[55] Now revision of the sum's value in light of a notification is possible by looking up the value of the changed parent and making $O(\log |P_j|)$ revisions to the aggregation tree, rather than looking up the value of all parents and recomputing from scratch. Of course, there's no need to actually store the parent items' values; they can be retrieved if needed by emitting a `@lookup` message and awaiting the response. The same is, more generally, true of intermediate nodes as well: they can be recomputed by issuing several lookups. The agent's state may thus devote storage to speeding up updates from (predicted-to-be) often-changing parents while not storing (meta-)data relevant to accelerating changes from subsets of parents whose values do (or are predicted to) not change often. A small example is shown in figure 2.7. In the case of inexact values (§1.5), properties like associativity and commutativity of aggregation operators cannot be assumed. Thus, as suggested in §2.1.2.1, we may need to impose additional structure on our partial aggregation structures, such as sorting of leaves, to ensure that the same bag of values produces the same answer in all cases. ◊

Example 12: Suppose that $e(j)$ is computing the *maximum* of its (many) parents. While an aggregation tree could still be of utility, selective operators (i.e., those $\oplus$ for which $(a \oplus b) \in \{a, b\}$; recall "Properties of Functions of Bags," in §1.3) offer another attractive option: associate inputs with all of the parents with that value. When a parent's value changes, one should remove it from any current association, compute its new value, and add the new association. The value of j is the largest value associated with one or more parents.

If desired, one need not keep associations for all parents. Instead, one may keep some subset of parents for (at most) the n largest values currently known.[56] An example may be found in figure 2.8. The removal, in light of a change, of a parent value not in the tracked set makes no change to the data structure and clearly does not alter the item's value. The removal of an entry other than the greatest does not trigger a notification, but may, if the set of parents associated with that value is now empty, make the data-structure incomplete, in the sense that there is no easy way to recover the value (and parents) that should replace

[55] Once again, we must apologize for the vocabulary. The child nodes of the aggregation tree are the *parent items* in the arithmetic circuit of aggregation the tree encodes: a node's value is, e.g., the sum of its child nodes' values. The leaf nodes are, then, the input items.

[56] Such structures are called "k-best" in the parsing and AI literature, where they are used in typically monotonic settings, wherein no parent's value ever changes, once proven, but there may not be utility in remembering all parents, merely the best ones. Curious readers are invited to read [141].

this deleted one. The addition of a value greater than the current maximum shifts the set n down and triggers an update. Addition of a value greater than the least tracked value also triggers such a shift; additions of values smaller than the least tracked value *must not* be tracked, as a larger value may have been shifted out of the set previously. Complete removal of the greatest value (i.e., the emptying of its associated set of parents), when not followed by an addition of an aggregand greater than the second greatest, makes the item's aggregated value equal to that second greatest aggregand (which may not be known, if there have been many removals previously). The net result is a data structure which can locally respond to *some* notifications without backward-chaining, but may occasionally need to "refill" its cache (by querying all parents to find the current largest n values). If there are few non-increasing changes, this can be especially attractive.

In fact, one could choose to deliberately not track a new value, so long as that value is smaller than (or equal to) the current maximum, and one discards all smaller tracked values. That is, one could eagerly discard some of the non-optimal values, out of a belief that they are not likely to be useful in the future. Perhaps one believes that past results are a strong indicator of future performance, and so, given a string of increasing inputs, would believe that yet more increases are soon forthcoming. ◊

2.3.4 Mixed Chaining as Message Passing

Naturally, we can combine the above perspectives to restate and extend the algorithm of §2.2.4 as a message-passing system. The key new feature is that backward- and forward-chaining may interleave in ways they could not before. An agent may be in the middle of performing a backward-chaining computation and receive a notification from a parent, and, dually, may receive a request for its value while processing a notification. The agents are more complex, but the framework as a whole is largely unaltered.

In a mixed-chaining system, it was possible to encounter a notification while backward-chaining towards visible ancestors. In so doing, one could obtain a value that we "marked" (§2.2.4.2) to ensure that memos derived from this value were appropriately paired with notifications. We did not much consider the impact of these marked values on derived values, because values were opaque objects to be recomputed every time as needed. Now, however, that we have agent states, we must acknowledge another implication of a marked value for the parent i: it likely differs from both the previous value observed for i (if any) and, plausibly, the next. Thus, agents may need to treat marked values specially (we shall see examples in later sections), so we pass marks in as part of `@value` messages. Similarly, in a mixed-chaining system, notifications may not carry new values either, so `@notifyFrom`$\langle i, \text{UNK}\rangle$ is entirely sensible. In the worst case, a recipient of such a message will emit `@lookup`$\langle i\rangle$; in the best case, the recipient's value is insensitive to i and no work is needed or the framework may take it upon itself to flush the would-be recipient state, to reset it later.

2.3.4.1 Selective Memoization

Thus far, we have been somewhat binary in our memoization of agent states: we have spoken of caching them or not. While we have, briefly, alluded to other possibilities, such as, in §2.3.3, an agent which caches only the result of computation and otherwise behaves as one first constructed, we should explicitly mention that there is a rich space of design here. Agents may store *any* information about their value and their parents that may be useful later in computation, and in many cases there are trade-offs between speed of response to messages and space occupied by these cached intermediates. If the framework needs to reclaim space, one could imagine a progression of `@flush`$\langle x \rangle$ messages, describing how much of the agent's internal state is believed to be of future utility to the larger computation. One could ask to flush cached results, trim aggregation trees to their tops, to decrease the size in a selective aggregator's associative map, etc. However, if resources are truly tight, the framework is free to discard the entire state object and begin again should j receive a message.

More generally, while we have, thus far, considered only queries for an item's value in its entirety, as an opaque quantity. However, we may wish to allow for predicated queries, e.g., "Is your value positive?" These kinds of queries can be answered accurately even if agents are storing *partial* information about their values.

2.3.5 Obligation Revisited

By default, when an item j demands the value of one of its parents i and the framework caches the resulting agent state, i becomes obligated to j. That is, the framework is obligated to forward `@notify`$\langle\rangle$ messages from i as `@notifyIn`$\langle i, v \rangle$ to j. Zooming in a bit, if i was not already obligated to j, it becomes so obligated as soon as j receives a `@value`$\langle i, v \rangle$ response to its `@lookup`$\langle i \rangle$ query and the cached agent state is replaced (or the new state is cached within an update, à la §2.3.3): at this point, j's agent state(s) depend(s) upon the observed value. If j's agent state is freshly initialized or was not previously obligating i, while the query is pending, we may consider i not obligated to j, as the state associated with j cannot yet be dependent upon the value of i, so there is no need to revise it. In the story so far, this obligation persists until j *and all of its descendants* have been FLUSH-ed.

As in §2.2.4.3, messages on the agenda serve to sever the recursive behavior of obligation. A pending notification at j removes obligation to its children k: they will be updated anyway, so there is no need to pass additional messages. j may still obligate its parent i if j is itself memoized, despite this notification. Dually, a *refresh* update (as opposed to a *replacement* update) at j removes obligations from its parents i. j may remain obligated to its children, despite this refresh update. All of this, as with obligation itself, is managed by *the framework*, not the agents themselves.

Some careful attention must be paid to the NULL initialization strategy, which would seemingly violate the story just given. These agents seemingly have made no queries of their parents and yet must obligate them, otherwise forward-chaining cannot possibly function. As discussed in §2.2.3, the NULL initialization strategy is a special case of the conservative initialization strategy, which queues refresh updates at all derived items (and thereby *releases all items from obligation*, initially). These updates eventually pop and serve to create obligation; the NULL initialization strategy relies on these updates having no other effect. The algorithms of §2.2 use a notion of obligation strictly derived from the memo table, so the mere presence of NULL in said is sufficient to ensure obligation. Here, however, because we wish for a finer notion of obligation, when executing under an agent-based analogue of the NULL initialization strategy, we must initialize obligation at the same time, *as if* agents had dispatched `@lookup` requests.

2.3.5.1 Obligation Release Messages

While some messages from i to j may not change j and so may not trigger additional update messages from j, the framework can not extrapolate from such an event to conclude that the value of j is completely insensitive to the value of i. As such insensitivity can lighten the load of forward-chaining, we add a new kind of output message, an **obligation release**, which explicitly informs *the framework* (rather than other items of the circuit) of such insensitivity. Such a message, `@obligate`$\langle\sigma\rangle$, with $\varnothing \subsetneq \sigma \subseteq \mathbb{N}_1^n$, where j has n parents, ensures that each $\alpha = \{\vec{P}_j{\downarrow}_i \mid i \in \sigma\}$ is obligated to j and allows the framework to note that all other $P_j \smallsetminus \alpha$ are not so obligated. It is never necessary for an item to send such messages, and items must be prepared to receive updates from parents outside their declared dependencies, as the framework is free to *over-approximate* the obligation relationship (by, e.g., flushing its memory of obligation releases). Control over obligation generalizes the **watched-variable trick** from the satisfiability community [130].

Example 13: Suppose j is an item whose associated function is the AC-reducer OR, and, further, suppose that its i-th parent has value TRUE. As long as this parent has this value, j is *insensitive* to its other parents, who should not be obligated to propagate their updates to j. Thus, j should, having concluded that its value is TRUE emit a `@obligate`$\langle\{i\}\rangle$. If multiple parents i_1, i_2, … all take on value TRUE, the j agent is free to watch any nonempty subset of them and to release from obligation parents as they cease being TRUE. ◊

There are some subtle points of control of obligation. σ must be a subset of previously obligated items, i.e., those from the most recent `@obligate`$\langle\sigma'\rangle$ message (inductively) or subsequently queried (and thereby implicitly obligated). An agent for j should not use `@obligate`$\langle\sigma\rangle$ to *add* an obligation, as it may have missed messages from the newly obligated parent i during the duration it was not obligated. (Recall from the start of §2.3.5 that the framework adds obligations explicitly when routing `@value` messages and ensures that it always *over*-estimates the true obligation.) Behaviorally, should an agent attempt to add obligation this way, the framework should at a minimum raise a warning and either synthesize a refresh or invalidation notification from i (e.g., deliver `@notifyFrom`$\langle i, \text{UNK}\rangle$ to j) or discard the agent state for j in favor of a new instance.

If there is a *refresh* update pending at j, the effective obligation from j to its parents is that of *the update* agent state. It is possible that j *was*, in the past, obligating some parent i but that an update removes this obligation. Even if that update has yet to be applied, i is still not obligated to j: the value of j is "about to be" insensitive to that of i and so there is no point in forwarding messages.

In an actual implementation, one may also wish for a dual message which *removes* an item from the obligation set. This could be useful to "remind" the framework of de-obligations it has forgotten: upon receipt of an irrelevant notification from i, j need not reconstruct its entire set of obligands, but can simply reply that i is not in that set.

2.3.5.2 Applying Watched Variables to Obligation Itself

Recall that our solver permits one-sided error in the case of obligation: the solver will be *correct but inefficient* in the case of spurious conclusions of obligation (but may be incorrect given false claims of *non-obligation*). Thus, it may be expedient to approximate obligation of i by caching the identity of a descendant item, $k \in \mathcal{I}$, which causes obligation (by being memoized, continuously queried, or itself obligated to some further descendant). Such an item serves as a kind of watched variable; because we neglect to consider the potential, obligation-severing effect of agenda messages between i and k, this cached value provides a *safe approximation* of i's obligation. We can, at any moment, backward-chain (towards child items, because this is the obligation circuit) to derive a new cache for i, and indeed must do so when and if k truly becomes no longer obligating. Only if none is found may we conclude that i is not obligated (and inform the parents who are watching it).

2.3.5.3 Future Work: Predicated Obligation Releases

In addition to specifying *which* parents are still of interest to a particular child item, one could imagine extending obligation to carry a notion of *which changes* in a parent would be of interest. Such predicated obligation arises naturally from the hypothetical predicate queries of §2.3.4.1. This enriched notion of obligation could even be used *transitively*, quiescing swaths of the circuit that would otherwise have to transit messages only to discover that they had no impact on the items of interest to the external driver. We have considered this case, but not designed a solution.

An item implementing comparison against a constant would only need to be informed of transitions of its parent's value across this constant. That is, if $e(j)(\langle i \rangle) = i > 3$, i should only pass messages to j if its value crosses 3 (in either direction, naturally).

Comparison *between* parents' values are also of interest, but more subtle. Suppose that $e(j)(\langle i, i' \rangle) = i < i'$ and j last perceived its parents with values v_i and $v_{i'}$, respectively, then, while it may seem that j should release item i from transitions that do not cross $v_{i'}$ and dually for i', this is too strong a condition and will miss some updates. (Consider $1 < 5$ transitioning to $4 < 3$.) However, one can pick one side of the comparison arbitrarily to watch for *all* updates while using predicated obligation releases with the other. When the former parent's value changes, one should re-query the latter to catch up with any changes in value that had been suppressed by obligation release, and then issue a predicated release to the latter parent.

Selective operators can also give rise to useful opportunities for predicated obligation release. For example, only the current maximum of a collection of parents need pass all messages to its maximizing child; the rest need only pass messages should they overtake the current maximum. When the current maximum decreases, all parents should be queried, the new maximum selected, and new predicated obligation releases issued.

Thus far, we have considered only the generation of predicated obligation releases. While the framework surely should listen in on obligation releases for maintaining its notion of obligation, it is also potentially useful to expose these messages to the agents themselves! In light of a *received* predicated obligation release from all children,[57] an agent may be able

[57] Such a message routed to a parent agent must be the predicate conjunct of all release messages from

to generate predicated obligation release messages of its own. For example, a minimizing agent could simply pass on any upper bound constraints: since it only matters that the minimum is below some threshold t, the released node could *flush its current cached value* (so that it will not, later, answer incorrectly) and release its parents from updates that do not take their values above t.

We would like to combine predicated obligation release with queries as well, so that the two can be delivered to a parent together. Thus, children could ask questions of the form "What is your value, if it is larger than w?" Such a question could be asked by a maximizing item k which has already observed one parent item to have value v or, as we suggested at the start of the section, by an item whose $e(k)$ function explicitly tests this parent's value against k. A minimizing parent j, having received such a query, could then stop querying its parents as soon as it found one, i, with value $v \leq w$ and return NULL. The item i must be remain obligated to j, at least for transitions across w. j should cache UNK as its value (since it does not know the minimum of all of its parents), though it may remember (and properly maintain) the current contender, v, in hopes of being able to answer subsequent predicated queries without having to query its parents.

2.3.5.4 Mark Management

Now that we have allowed agents to communicate tighter bounds on their dependencies, one may wonder if, in the mixed-chaining scenario, having a lookup cross a notification, thereby marking the obtained value, *always* necessitates constructing additional notifications when caching agent states. Indeed, the necessity remains only if the agent's value is actually sensitive to the marked value, i.e., if the agent continues to obligate the parent that gave it the marked value.

Example 14: Consider a simple agent computing the product of its two parents' values, i.e., $j = i * i'$. Upon receipt of a demand for its value, j is likely to request the value of *both* its parents. Suppose, further, that the response messages are $\texttt{@value}\langle i, 0, \texttt{FALSE}\rangle$ and $\texttt{@value}\langle i', 1, \texttt{TRUE}\rangle$. Because 0 is an absorbing value of multiplication, the resulting value, $j = 0$ is not actually sensitive to i', so j's agent is justified in outputting the messages $\{\texttt{@valueIs}\langle 0\rangle, \texttt{@obligate}\langle\{i\}\rangle\}$. In this case, the framework is free to *not mark* the value of j and to *not construct* a notification at j on the agenda. In the language of §2.2.4.2, the logical-or of marks is computed now not across all parents but only those still obligated. ◊

2.4 Further Enriching Messages

The replacement $(i : \underline{\leftarrow v})$ and refresh $(i : \underline{\leftarrow})$ updates and invalidation notifications $(i : \overline{\leftarrow})$ considered thus far certainly get us off the ground, but are somewhat uninformative: they treat values as opaque objects and either provide an entire replacement or defer the computation of a replacement to backward-chaining. We now consider additional information that could be carried by an update for use by child items.

its children. If some child has not released this parent at all, and is tacitly claiming sensitivity to the full range of values, then no such message can be generated.

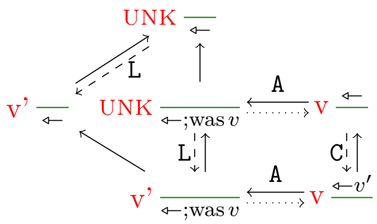
```
def UPDATE($j \in \mathcal{I}$, $w \in \chi \cup \{\text{UNK}\}$) $\in \langle\rangle$
  maybe
    if $(w = \text{UNK}) \vee (\mathcal{M}(j) \neq w)$ then
      $\mathcal{A}(j)_{\texttt{update}} \leftarrow \overline{\leftarrow w}$
    % else update made no change
    return
  % any existing update superseded
  delete $\mathcal{A}(j)_{\texttt{update}}$
  APPLY($j$, $w$)
```

```
def APPLY($j \in \mathcal{I}_{\texttt{der}}$, $w \in \chi \cup \{\text{UNK}\}$) $\in \langle\rangle$
  $\mathcal{M}(j) \leftarrow w$
  $\mathcal{A}(j)_{\texttt{notify}} \leftarrow \overline{\leftarrow}$

def LOOKUPFROMBELOW($i \in \mathcal{I}$) $\in \langle \chi, \texttt{bool} \rangle$
  $\langle v, m' \rangle \leftarrow$ LOOKUP($i$)
  $m \leftarrow i \in \mathcal{A}_{\texttt{notify}}$
  return $\langle v, m \vee m' \rangle$
```

Figure 2.9: Selected modifications to EARTHBOUND to support both updates and notifications at the same item, removing Invariant 6.

Figure 2.10: Relative information content of item configurations (memo and agenda states). Solid arrows point towards less-informative configurations. The solver is free to evolve any message along any directed path of solid arrows at any time (in addition to propagation). Two of these solid arrows, labeled with A, capture the action of APPLY. Dashed arrows are possible effects of LOOKUP (L) or COMPUTE (C); while the solver is free to undertake these operations, they are likely more expensive than the *local* operations of the solid arrows. The two dotted edges indicate equivalence of the two configurations, but we force monotonic motion of messages down the page by prohibiting these moves.



UNK $\overline{\leftarrow}$
L
v' $\overline{\leftarrow}$
UNK $\overline{\leftarrow ; \text{was } v}$
A
v $\overline{\leftarrow}$
L
C
v' $\overline{\leftarrow ; \text{was } v}$
A
v $\overline{\leftarrow v'}$

This section introduces several new entries in our taxonomy of notifications. For systems which enforce Invariant 6, the test on line 6 of listing 2.4 must be adjusted to reflect *any* kind of notification. That is, $(\mathcal{A}(j) \neq \overline{\leftarrow})$ must be generalized; we lack a concise syntax for "any notification," but imagine that any implementation has, or could easily have, such a test at hand. Similarly, mark introduction at line 10 of listing 2.6 must be generalized. However, cyclic circuits (§2.5) will provide motivation for dropping Invariant 6, and so obviate the need to explicitly prevent the construction of both an update and a notification at a given item. UPDATE, APPLY, and LOOKUPFROMBELOW would thus be rewritten along the lines of figure 2.9. Requisite changes not shown therein include FLUSH (from listing 2.7), RUNAGENDA (from listing 2.5), and the the creation of notifications within LOOKUP (line 21 of listing 2.6). The solver can dispatch updates and notifications from the same item in either order; it is not, for example, requisite to apply any pending update before propagating a notification from the same item.

2.4.1 Notifications with Old Values

Once we have imagined agents as maintaining state about their parents, it stands to reason that informing them of the former value, as well as the new value (or UNK) of a parent is potentially useful. We visually represent notifications carrying the old value v as $i : \overline{\leftarrow ; \text{was } v}$;

to cut down on the number of messages used during propagation, one may bundle together both the old and new value, as $i : \overline{\leftarrowtriangle v'; \mathrm{was}\, v}$. These messages are *not idempotent*: receiving $i : \overline{\leftarrowtriangle v'; \mathrm{was}\, v}$ twice would seem to suggest *three* transitions in the the value of i: from v to v' back to v and then back to v', but the middle of these is implicit. Thus, we risk misunderstanding the message stream looking only at one at a time. Moreover, these notifications are not likely useful if they are long-lived, as they will function effectively as duplicate memo entries. While lookupFromBelow can be modified to simply return the old value, rather than mark the return from Lookup, the intended use case for notifications with old values is strictly ephemeral, *during* propagation of notifications to children. Updates do not carry old values; if the old value is known to the system, it is in the memo table $\mathcal{M}$. A pictorial representation of the messages we have defined so far may be found in figure 2.10.

Example 15: Carrying the old value in a notification makes some local circuit transformations lighter weight, in the sense that they may occupy less storage in an actual implementation. In example 12 (in §2.3.3.1), it was necessary to associate values with *sets* of parents (i.e., *their names*), so that the parents could, when notifications were delivered, be removed from the sets justifying the availability of a given value. Now that we have old values available in notifications, we can opt to store only *cardinalities* of these sets, rather than the sets themselves.[58]

In the 2013 prototype of Dyna 2 [59], all aggregators used this so-called "**baggregator**" strategy. Notifications existed only ephemerally within the system; the agenda stored only replacement updates. The solver therefore could ensure that an initializing aggregator could query all of its parents and obtain their values before receiving notifications. While not particularly useful for non-selective aggregators, this strategy remains correct if one remembers to factor in the cardinalities when aggregating. ◊

In the case of inexact values (§1.5), the equality test implied by use of such an old value is suspect. The suggestion given above, to react to forward-chaining's notifications by searching for a value among several will not work in general, as the parent may have *recomputed* its inexact value and gotten a (hopefully, slightly) different answer. However, if we are certain that parents are always memoized, then the strategy remains sound; the memo table implicitly serves as our map from parents to values, and there is no need to duplicate that within the child item's state.[59] Alternatively, one could optimistically ignore the inexactness of values and run exact comparison, falling back to full re-Compute-ation when no match was found. If one is willing to tolerate some slop in the result, one could

[58] The agent must be sure, however, that the parent whose old value it wishes to remove from the set actually contributed a value before. This requires either that agents recall the queries they have seen answered or, more likely, some discipline within the agent framework so that non-idempotent notifications, such as those carrying old values, are always delivered to agents expecting them. In this case, the framework should originate these messages only if it is sure that all of the children which will receive it have indeed queried the source parent item and have not been reset since; it is likely that such conclusions will be due to invariants of the framework's execution rather than explicitly stored state, but the mechanism is immaterial.

[59] This suggests that there are cases where it would be beneficial to inform children of flush events within the circuit. We do not do so here, but it would not require any radically new machinery. It would suffice to invent a new kind of notification, bearing an old (to-be-flushed) value and, explicitly, an *unknown* new value, i.e., $\overline{\leftarrowtriangle \textsc{unk}; \mathrm{was}\, v}$. While $i : \overline{\leftarrowtriangle; \mathrm{was}\, v}$ serves to indicate a transition from v to some other value (the result of Lookup applied to i), this new form indicates merely the change in metadata and provides notice that the *next* notification from i will not be capable of specifying the old value v.

imagine searching for a cached (perhaps, the closest) inexact value within some window around the old value given and behaving as though that one were the one being revised. Such engineering approximations are out of scope for this document.

2.4.2 Partially Propagated Notifications

§2.3.3.1 considered dynamic transformations of the circuit wherein a child may group its parents in some useful way. Dually, parents may wish to group their children, propagating notifications more aggressively to some than to others. Notably, we expect such a facility to be of service to responsiveness: selective propagation (§2.2.4.4) now need not propagate along any large fan-outs from parents to children, as the algorithm can now propagate only to the children who are (or whose descendants are) being QUERY-ied. We thus introduce a **partially propagated notification**, $\overline{\sigma \leftarrowtriangle}$, which indicates that it has already been propagated to the children in σ, called the **visited set**. Introducing partial propagation necessitates a few changes to the algorithm; unless otherwise specified, we ignore any possibility of notifications carrying old values (merely for simplicity of exposition; the two are not incompatible). Further, we assume that an invalidation notification, $\overline{\leftarrowtriangle}$, is now merely a shorthand for a trivially-partially-propagated notification, $\overline{\varnothing \leftarrowtriangle}$.

- PROPAGATE must exclude the already-visited children σ and may, if the external policy chooses, stop with an updated visited set σ:

```
def PROPAGATE($i \in \mathcal{I}$)
  let $\overline{\sigma \leftarrowtriangle} = \mathcal{A}(i)$
  finished ← TRUE
  % loop over un-visited, obligated children
  foreach $j \in (C_i \smallsetminus \sigma)$ where obl$(i,j)$ do
    maybe % optionally skip this child, to process later
      finished ← FALSE
      continue
    $\langle \sigma, w \rangle$ ← $\langle \sigma \cup \{j\}, \text{UNK} \rangle$
    maybe $\langle w, \_ \rangle$ ← COMPUTE($j$)
    UPDATE($j$, $w$)
  if finished then % loop did not execute or did not defer
    delete $\mathcal{A}(i)$
  else
    $\mathcal{A}(i)$ ← $\overline{\sigma \leftarrowtriangle}$ % Record new visited set
```

- LOOKUPFROMBELOW should (but need not) be modified to take the identity of the child doing lookup to assess whether the notification applies to that child. (If this is skipped, some values will be marked unnecessarily.) The call in COMPUTE(j) is now LOOKUPFROMBELOW(i,j) and the method itself reads as

```
def LOOKUPFROMBELOW($i \in \mathcal{I}$, $j \in \mathcal{I}_{\mathtt{der}}$)
    $\langle v, m' \rangle \leftarrow$ LOOKUP($i$)
    $m \leftarrow \mathcal{A}(i) = \overline{\sigma \leftarrowtriangle}$ and $(j \notin \sigma)$ % mark only if not already propagated to j
    return $\langle v, m \vee m' \rangle$
```

- When partial propagation is combined with notifications carrying old values (§2.4.1), LOOKUPFROMBELOW may be able to exploit these old values, returning an *unmarked* value immediately to children. That is, LOOKUPFROMBELOW can act as LOOKUP would, ignoring the fact that the value is contained within a notification, by first testing:

  ```
      if $\mathcal{A}(i) = \overline{\sigma \leftarrowtriangle \_; \mathtt{was}\, v}$ and $(j \notin \sigma)$ and $v \neq$ UNK then return $\langle v, \text{FALSE} \rangle$
  ```

  In this combination, one could imagine enriching FLUSH to remove either σ or the old value from memory in addition to its current ability to remove memos. In order to maintain correctness of the algorithm, forgetting σ must imply forgetting the old value, but one can preserve σ when forgetting the old value. Thus, one can propagate notifications with old values to some children and invalidations to others. That is, within the taxonomy of figure 2.10, $\overline{\sigma \leftarrowtriangle; \mathtt{was}\, v}$ can freely rewrite as $\overline{\sigma \leftarrowtriangle}$, and thence to $\overline{\leftarrowtriangle}$, but not as $\overline{\leftarrowtriangle; \mathtt{was}\, v}$; these rewrites apply orthogonally to any flushing of the memoized value.

  Separately, the above code listing for PROPAGATE must be revised to persist the old value when it updates the notification's visited set σ. However, because that listing simply calls COMPUTE, it does not gain any benefit from these old values; more generally, as discussed in example 15 (in §2.4.1) (and §2.3.3.1), one would wish for a mechanism to alter the *state* of item j in light of the knowledge that its parent i had changed (and with knowledge of i's old value; i's new value is available by LOOKUP).

A potential second use for partially propagated notifications is to enable a kind of *propagation* during backward chaining, not just marking (recall §2.2.4.2). At present, a marked return necessitates duplicated work: the notification will propagate until it encounters a memo, re-COMPUTE-s the corresponding value, and discovers that no change is necessary. If we move from a boolean space of marks to a three-value space, backward chaining can indicate to its caller—the child item—that this response is not only marked, having traversed a notification, but that the child must treat it as a notification and react as if it were forming an update. In more detail, take marks to be one of the three values MARKNONE < MARKCROSSED < MARKPROPAGATED. MARKNONE behaves as a FALSE used to, and MARKCROSSED as TRUE. LOOKUPFROMBELOW now may "strengthen" MARKCROSSED to MARKPROPAGATED by adding the child to the visited set σ:

B. 2.1

```
1 def lookupFromBelow($i \in \mathcal{I}$, $j \in \mathcal{I}_{\text{der}}$)
2   $m$ ← markNone
3   if $\mathcal{A}(i) = \overline{\sigma \leftarrow}$ and $(j \notin \sigma)$ then
4     $m$ ← markCrossed
5     maybe { $\mathcal{A}(i)$ ← $\overline{(\sigma \cup \{j\}) \leftarrow}$ ; $m$ ← markPropagated }
6   $\langle v, m' \rangle$ ← Lookup($i$)
7   return $\langle v, \max(m, m') \rangle$
```

Compute now computes the max of the marks on parents, rather than $\vee$ (logical or). Lookup must honor its mark obligations, if any:

B. 2.2

```
1 def Lookup($i \in \mathcal{I}$)
2   if $\mathcal{M}(i) \neq$ UNK then
3     return $\langle \mathcal{M}(i)$, markNone$\rangle$
4   $\langle v, m \rangle$ ← Compute($i$)
5   maybe $\mathcal{M}(i)$ ← $v$
6   if $m$ = markPropagated ∨ ($m$ = markCrossed ∧ $\mathcal{M}(i) \neq$ UNK) then
7     $\mathcal{A}(i)$ ← $\overline{\leftarrow}$ % Preserve Invariant 5
8     $m$ ← markCrossed
9   return $\langle v, m \rangle$
```

While it may seem that markPropagated will not be carried along recursive backward chaining, as Lookup only returns markNone or markCrossed (with markPropagated elminated by the logic at line 6 of block 2.2), lookupFromBelow can promote any markCrossed to markPropagated. Not shown in the listings above, lookupFromBelow would be within its rights to remove the notification from the agenda if $\sigma \cup \{j\}$, at line 5 of block 2.1, covers all (obligating) children of i. In the case of a series of items with only one child, for example, this allows (the appearance of) a single notification to follow the downward motion of backward-chaining's unwinding of its recursive stack.

Discarding the mark in propagate (e.g., line 21 of listing 2.5) remains acceptable, even in the case of markPropagated: the update takes the new value into consideration anyway, so no additional pushes to the agenda are required.

Partial Propagation and Obligations As might be imagined, a partially propagated invalidation notification $i : \overline{\sigma \leftarrow}$ (with $\sigma \neq \varnothing$) no longer completely severs the obligation from i to its children C_i. As $\sigma \subseteq C_i$ are already up to date, their perspective of the circuit is (locally) equivalent to one in which no notification is pending at i. Thus, propagating a notification from i to j and adding j to σ must have the effect of restoring (transitive) obligation from i to j even if each of the rest of $C_i \setminus \sigma$ remain severed.

2.4.3 Delta Messages

The first prototype of Dyna [51] actually used agenda-based forward-chaining with **delta updates** such as $i : \underline{\oplus v}$ for some operator $\oplus$. Applying this update increments the old memo $\mathcal{M}(i)$ to $\mathcal{M}(i) \oplus v$ and produces a delta notification $i : \overline{\oplus v}$ for propagation to children, indicating that the value has been changed by $\oplus$ combination with v.

Dijkstra's shortest-path algorithm [44] uses forward-chaining with delta updates and *idempotent-delta notifications*, $i : \overline{\min}$. These indicate that the value has changed by being combined with min, i.e., that it has decreased. Here, because the operator is selective (and therefore idempotent), there is no need to store the value into the agenda within the update object; it can be obtained by calling Lookup.

A delta notification at i is sometimes cheap to propagate to its child j, when compared to a replacement. One can sometimes avoid a full call to Compute(j) at line 21 of listing 2.5—which Lookup-s or Compute-s all the parents of j—by exploiting arithmetic properties of $e(j)$. If, for example, $e(j)(\langle v_i, \ldots \rangle) = v_i \oplus \ldots$, with $\oplus$ associative and commutative, then propagating $i : \overline{\oplus w}$ from the parent $i = \vec{P}_j{\downharpoonleft}_1$ is essentially free: it becomes $j : \underline{\oplus w}$ (and, eventually, after that pops from the agenda, $j : \overline{\oplus w}$). If $\oplus$ and $\otimes$ distribute and $e(j)(\langle v_i, x_0, \ldots, x_n \rangle) = v_i \otimes f(\vec{x})$, then $i : \overline{\oplus w}$ (again, with $i = \vec{P}_j{\downharpoonleft}_1$) becomes $j : \underline{\oplus(w \otimes f(\vec{x}))}$.[60] That is, we need only look up the *other* parents of j, i.e., $P_j \smallsetminus \{i\}$. The more general case, in which v_i is used *nonlinearly* within an $\otimes$ expression, requires computing several deltas (which can be merged together) and requires, in general, the old value as well as the delta for i; its solution is given in Eisner, Goldlust, and Smith [51].

Delta Obligations Unlike the invalidation notifications used ever since §2.2.3.3 ($i : \overline{\leftarrowtriangle}$), but like the partially propagated notifications of §2.4.2 ($i : \overline{\sigma \leftarrowtriangle}$), delta notifications are not idempotent. The change to the value of i encoded by a delta notification describes precisely one transition of i's value. If the value changes again, *another* delta notification must be queued as well (or merged with the existing notification, a subject to which we turn momentarily), or an invalidation must supplant the pending delta. That is, a delta notification *does not* release its item from obligation to its children; recall, for contrast, the discussion of notifications in §2.2.4.3.

2.4.4 Message Merge

The forward-chaining algorithms of §2.2.3 and §2.2.4 exploited the fact that, for any of the three message types defined, a subsequent message (of the same sort) should always supersede any pending on the agenda. Thus, the agenda stored at most one update and/or notification for any item, and pushing was always done by simply setting this update, discarding any prior, still-pending message. The mixed chaining algorithm additionally enforced a (cosmetic) *one message* property (Invariant 6). This required promoting updates to notifications when a notification already existed at an item (line 6 of listing 2.4), as done in the fall-through case at line 14 of listing 2.4.

[60] One must be a little cautious when interpreting these propagated responses, though. If $\oplus$ has an absorbing element (i.e., some value 0 such that $\forall_x\, 0 \oplus x = x \oplus 0 = 0$), the change indicated by a delta notification may not actually amount to a change of value. Consider, for example, taking the product of two numbers, one of which is 0 and the other of which changes by doubling its value from 1 to 2. The resulting notification will correctly convey that the value of the product has doubled, having gone from 0 to 0. Such a lack of change will eventually be noticed when updating a *memoized* value, in which case $\underline{\oplus d}$ will pop and produce *no* notification, rather than $\overline{\oplus d}$.

In the case of a delta message, however, this outright replacement is clearly incorrect. As mentioned earlier, one possibility would be to detect the collision of pending messages and fall back to using an idempotent replacement. Another possibility would be to *queue* messages (per item and type; i.e., all updates to i form one queue, notifications from i another, updates to j another, etc.) and pop them *in queue order*. However, we can take advantage of associativity of delta messages, when possible, to *merge* updates on the agenda, trading time (to find and replace a pending message, as well as compute the merger) for space (needing to store only one message).

Moreover, in a general algorithm which uses *both* delta and replacement (and/or invalidation) messages, one again needs to ensure temporal ordering or be able to merge messages. Given two successive notifications with compatible, associative, $\oplus$ delta updates, $\leftarrowtriangle\!\!-; \mathrm{was}\, v; \oplus d$ (with $d \in \chi$ and v in $\chi \cup \{\textsc{unk}\}$) pushed before the $\leftarrowtriangle\!\!-; \mathrm{was}\, w; \oplus e$ (similarly qualified), these two notifications can be combined into $\leftarrowtriangle\!\!-; \mathrm{was}\, v; \oplus(d \oplus e)$. If the delta component of one or the other message is missing, or if the delta components use differing operators in the two messages, the delta component must be discarded, yielding $\leftarrowtriangle\!\!-\, w'; \mathrm{was}\, v$. Updates behave similarly to notifications but, of course, carry new values rather than old ones. All told, the merge of $\leftarrowtriangle\!\!-\, v; \oplus d$ followed by $\leftarrowtriangle\!\!-\, w; \oplus e$ is $\leftarrowtriangle\!\!-\, w; \oplus(d \oplus e)$, again, assuming associativity of $\oplus$.

Partially propagated notifications also complicate message merge. One possibility, given some $\sigma \leftarrowtriangle\!\!-; \mathrm{was}\, v$ message already on the agenda (with $\sigma \neq \varnothing$) and a newly arriving $\leftarrowtriangle\!\!-; \mathrm{was}\, v'$ message, is to *suspend* the former and propagate the latter *exclusively* to σ, making it $\sigma \leftarrowtriangle\!\!-; \mathrm{was}\, v'$. Now that the visited sets are equal, message merge can proceed as above and simply preserve σ. This works even if delta components are also present. In fact, this strategy cascades, should one wish: one could maintain a stack of suspended messages; traversing further down the stack, one would see strict increases in visited sets.[61] One would be able to merge two messages at the top of the stack when their visited sets became equal (i.e., after propagation of the topmost message). However, one may still wish to merge messages with unequal visited sets. In this case, the visited set of the result must be *reset* to $\varnothing$, and any old value or delta components discarded from the result, as different sets of children have different information about the state of the parent to whom this notification belongs.

2.5 Generalized Arithmetic Circuits

Our algorithm can be extended to handle cyclic arithmetic circuits. Pure forward chaining can propagate updates around cycles indefinitely in hopes that the memos will converge [51]. If so, it finds a *fixed-point solution* $[\![\cdot]\!]$. 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. The solution is interesting.

Example 16: Let us first ponder why one might care about cyclic circuits. We return to the world of AND/OR graphs of §2.1.2.1. A Datalog program for computing the transitive closure

[61] And one would likely care to revisit the check made in LOOKUPFROMBELOW in §2.4.2 to scan the stack of pending notifications for the appropriate old value to provide to a given child.

of a relationship $\mathtt{e}^{/2}$ (i.e., graph reachability where $\mathtt{e}^{/2}$ encodes the edges) as $\mathtt{te}^{/2}$ consists of two rules and looks like this:

```
te(From,To) :- e(From,To).
te(From,To) :- e(From,M), te(M,To).
```

Describing a circuit for this program is straightforward:

1. The first rule describes circuit edges which make the OR node $\mathtt{te}\langle f,t\rangle \in \mathcal{I}_{\text{der}}$ a child of the input node $\mathtt{e}\langle f,t\rangle \in \mathcal{I}_{\text{inp}}$. Because there is no conjunction in this rule, there is no need for an AND node. Datalog ensures that $\mathcal{I}$ is finite, so there are only finitely many edges created by this definition.

2. The second rule defines $\{\mathtt{join}\langle f,m,t\rangle \mid \mathtt{te}\langle f,t\rangle \in \mathcal{I}, \mathtt{t}\langle f,m\rangle \in \mathcal{I}, \mathtt{te}\langle m,t\rangle \in \mathcal{I}\}$ be a subset of $\mathcal{I}$ (even though this is a recursive constraint on $\mathcal{I}$, it does not change the finiteness of $\mathcal{I}$). The item $\mathtt{join}\langle f,m,t\rangle$ is a parent of the OR node $\mathtt{te}\langle f,t\rangle$ and a child of both OR nodes $\mathtt{e}\langle f,m\rangle$ and $\mathtt{te}\langle m,t\rangle$.

The structure of this graph is *cyclic*. Using some finite set of integers as arguments of the $\mathtt{e}^{/2}$ items, for example, we find that $\mathtt{te}\langle 1,2\rangle$ is its own child, assuming $\mathtt{e}\langle 1,1\rangle$ is an item. Moreover, if $\mathtt{e}\langle 3,4\rangle$ and $\mathtt{e}\langle 4,3\rangle$ are both items, then $\mathtt{te}\langle 3,3\rangle$ and $\mathtt{te}\langle 4,4\rangle$ are cyclically defined, as $\mathtt{te}\langle 3,3\rangle$ has (at least) $\mathtt{join}\langle 3,4,3\rangle$, $\mathtt{e}\langle 3,4\rangle$, $\mathtt{te}\langle 4,3\rangle$, $\mathtt{join}\langle 4,3,3\rangle$, $\mathtt{e}\langle 4,3\rangle$, and $\mathtt{te}\langle 3,3\rangle$ in its ancestry. ◊

2.5.1 Fixed-Point Solutions

Recall from §2.1.2 that a solution to an acyclic circuit is simply an extension of the inp map ($\mathcal{I}_{\text{inp}} \to \chi$) to a map over all of $\mathcal{I}$ by using the functions at each item. In the case of an acyclic circuit, we are guaranteed a single solution which may be obtained by "bottom-up" ("ancestors-first") topological traversal of the circuit. However, this still seems a little procedural, as definitions go. On the other hand, the definition of $[\![\cdot]\!]$ given in equation (2.1) (in §2.1) reeks of circularity even for acyclic circuits.

Following Van Emden and Kowalski [178, §6], another approach would be to split $[\![\cdot]\!]$ and define a **step** operator for a given arithmetic circuit. Such an operator would *step* a map $\mathcal{I} \to \chi$ to another such map which was, roughly, "closer" to being a solution. Formally, $\text{step} \in (\mathcal{I} \to \chi) \to (\mathcal{I} \to \chi)$. A perfectly reasonable choice looks very much like equation (2.1):

$$\text{step}(m)(j) = \begin{cases} \text{m}(j) & j \in \mathcal{I}_{\text{inp}} \\ e(j)(\langle m(i_1), \ldots, m(i_{n_j})\rangle) & j \in \mathcal{I}_{\text{der}} \end{cases}.$$

This function recomputes the value of *every* (non-input) item j of the circuit by using $e(j)$ given the values for P_j from the input map, m. For an acyclic circuit, $[\![\cdot]\!]$ as described earlier has a curious feature: $\text{step}([\![\cdot]\!])$ is equal to $[\![\cdot]\!]$! In fact, we can go so far as to say that a solution is *defined* to be such a *fixed-point* of this step function. For an acyclic circuit, there remains only one such (for each choice of input values).

We can immediately relate our procedural agenda to this step function as well. Update messages serve to identify items for which $\text{step}(m)(j)$ might differ from $m(j)$. Notifications serve to identify *parents of* items with the same possibility of value changing before and after stepping. We could, in fact, think of the action of applying an update or propagating a notification as another kind of step function, from one configuration of the memo table (and therefore of Lookup) to another. Ignoring any updates to input items, we could imagine implementing forward chaining as iterating step on the conservative initialization strategy until a fixed-point had been found, i.e., $\mathrm{step}(\mathrm{step}(\cdots(\{j \mapsto \mathrm{NULL} \mid j \in \mathcal{I}_{\mathrm{der}}\} \cup \mathrm{inp})\cdots))$.

Example 17: Consider an item j with itself as its sole parent (i.e., a *self-loop* in the ancestry graph) with $e(j)(\langle v_j \rangle) = \bigoplus \lbag 1, v_j/2 \rbag$ (with $\oplus$ being an AC-reducer). If we take $\chi = \mathbb{R} \cup \{\pm\infty, \mathrm{NULL}\}$ and $\oplus$ such that $\bigoplus \lbag 1, \mathrm{NULL} \rbag = 1$, $\bigoplus \lbag 1, \pm\infty \rbag = \pm\infty$ (with corresponding signs), and $\forall_{x \in \mathbb{R}} \bigoplus \lbag 1, x \rbag = 1 + x$, there are three fixed-points: $v_j = \pm\infty$ and $v_j = 2$. The conservative initialization strategy will tend towards assigning j the value 2, as will any initialization of j to have a value in $(-\infty, \infty) \setminus \{2\}$. Initialization to a fixed-point will, by definition, not leave that fixed-point.[62] ◊

More importantly, this definition, of the step function and of solutions being fixed-points thereof, neatly applies to cyclic computations as well. It completely avoids the need for a topological traversal and doesn't punt the circularity of the circuit into circularity of $[\![\cdot]\!]$. Our observations relating step and the agenda continue to apply, as well. However, we do surrender *uniqueness* and even *existence* of solutions in the cyclic case; the potential price we pay for abandoning our topological traversal is that there may be *zero*, one, or *more than one* fixed-point of step for a given circuit and input map.

2.5.2 Guessing

In general, we can interrupt any (long-running) backward-chaining recursion—cyclic or otherwise—by allowing Compute(j) to optionally *guess*, memoize, and return an *arbitrary* value for j, such as NULL. Because this guess could be incorrect—the resulting $\mathcal{M}(j)$ may be inconsistent with j's parents—we must enqueue a refresh update $j : \underline{\cdot}$ to preserve Invariant 5 (in §2.2). Popping this update later will resume backward chaining (i.e., it serves as a continuation) to check that our guess at j is consistent with j's visible ancestors (perhaps now including j itself, cyclically). If not, it will use the agenda to propagate a fix by forward chaining (perhaps cyclically until convergence; recall §2.2.4.4).

If j is already obligated to any children k, its guessed value must be marked (recall §2.2.4.2), and so the guessing Compute must enqueue a notification $j : \overline{\leftarrow \mathrm{UNK}}$ to alert k that guessing $\mathcal{M}(j)$ may have changed it from the previous value seen by Lookup(j), which was based on memos at j or its ancestors or on a previous guess. As in §2.2.4.2, this notification preserves Invariant 5 between j and its obligated children k, avoiding the same subtle bug where an update that leaves $\mathcal{M}(j)$ unchanged is never propagated to k. This value may be marked with markPropagated, and the notification's visited set set appropriately, if the machinery of §2.4.2 is in use. Here there is real reason to retire Invariant 6 (in §2.2) and permit *both* an update and a notification to be pending at j: were we permitted at most one, in order to satisfy Invariant 5, we must use a notification and the guess *must not*

[62] In practice, if χ uses a floating point approximation, initialization to any finite number will eventually arrive at 2, or very nearby, due to rounding. Because addition (of a constant) and division (by a positive constant) remain monotone w.r.t $\leq$ even given the approximation of floating point numbers, values will not cycle in time, so finding a fixed-point is guaranteed.

be memoized (because, in the acyclic case, this potentially-inconsistent memo could lack a message in its visible ancestry; in the *certainly cyclic* case, by supposition, the guessed item is in its own visible ancestry and so the notification suffices).

If j is *not* already obligated to any of its children, we may elide pushing the notification and may return an *unmarked* value. Either there are no memoized descendants (and no continuous queries) of j (in which case, all descendants are immediately consistent with the guess) or any memoized descendants have (transitively) released j from obligation, indicating that their values are insensitive to that of j. We can justify this as a push of the notification followed by immediate propagation in which the loop at line 19 of listing 2.5 (in §2.2.4.1) loops over an empty set. Recall from §2.2.4.3 and §2.3.5 that active backward chaining (i.e., agents waiting for results of Lookup) by descendants does not contribute to obligation until those descendant items are memoized. In the case of cyclic j, i.e., j being its own ancestor, it may seem that memoizing j as part of guessing *necessarily* results in both an update and a notification at j. However, recall from §2.2.4.3 that refresh updates, which are the kind used above, sever obligation: because j *will* bring itself up to date, j's self-ancestry does not necessitate the use of a notification. (But, of course, *other* descendants of j may still impose the need for notifications!)

One particular special case of this general guessing strategy is to guess *only when* a cycle has been detected, and to eagerly test for cycles at every step of backward-chaining. Such an approach is probably the most natural for the agent-based framework of §2.3, which postulated tracking agents blocked on each other's answers and, essentially, prohibiting re-entrancy of agents. In a more procedural framework, such as EarthBound of §2.2, we *could* enforce this lack of re-entrancy of items, which would preserve a previously unstated invariant of the acyclic frameworks: in any recursive backward-chaining, each item occurs at most once within any call stack. Thus, all of Compute(j), Lookup(j), and lookupFromBelow(j) have been guaranteed that $\mathcal{A}(j)$ and $\mathcal{M}(j)$ will not change across their (mutually recursive) calls. Absent such strict cycle testing, our procedural solver enters novel situations, including the possibility of returning from Compute(j) with j *memoized*, because j is its own ancestor and more than one lap of the cycle had been unwound onto the call-stack before a guess was made. (That is, Compute(j) (transitively) called Compute(j) which did so again, due to lack of cycle testing.) The value being returned from Compute is *more up to date*—the result of of more iterations of the cycle—than the entry in the memo table. What are we to do?

One possibility is to *discard* the newer value in favor of the older one, so that only the newest and oldest instance of j on the stack have any real significance. The newest computes the memoized value, and the oldest returns this value to off-cycle children. This, however, has the unfortunate effect of wasting potentially a great deal of work.

If we wish to avail ourselves of the multiple cycles wound onto the call-stack, Lookup(i) must *re-probe* the memo table after Compute-ing and, if it finds that $\mathcal{M}(i) \neq \textsc{unk}$, then it must update the memo, mark the return value, and queue a notification to the agenda (and, in contrast to message merging, any old or delta components in any existing notification must be *discarded*). That is, line 19 of listing 2.6 (in §2.2) and line 21 are no longer optional if $\mathcal{M}(i) \neq \textsc{unk}$ after line 17. It *may* be possible, possibly with additional assumptions, to find situations that could avoid the need to mark the return value or push the notification.

2.5.2.1 vs. Query

Query(j) of listing 2.7 (in §2.2) ran the agenda to completion, and then called Lookup(j). Because the agenda had been freshly emptied, we could be certain that Lookup would not mark the resulting value. This line of argument would remain true, with slight but straightforward modification, were we to replace the call to RunAgenda within Query with a call to some routine which cleared (or obviated) all agenda messages from the ancestry of j. That is, this routine would have to Apply all updates to all ancestors of j and (at least partially) propagate all notifications at ancestors of j. It is sufficient that all notifications in the ancestry of j have been propagated to their children which are *also* ancestors of j; other children may be inconsistent, but because they are outside the ancestry of j, their inconsistency will not affect j's value (recall §2.4.2).

However, now that we may guess values, which will therefore *add* messages even when the agenda is empty, this argument no longer suffices. Moreover, there appears to be no mechanism for Query-directed control over this behavior: introducing a flag which would prohibit guessing is not tenable if there is a possibility of a cycle, all of whose items are un-memoized. Thus, we are forced into the awkward position of *looping* in Query, as shown in listing 2.8, and any proof of termination must appeal to the program itself (to ensure that the Query is well-founded) as well as some external, policy-aware argument. (Recall, for contrast, the case of *acyclic* (and, necessarily, finite) circuits, as discussed up until this section, on which Query must terminate: RunAgenda always advances one message strictly downwards—towards the leaves—per iteration, and Lookup, in the worst case, will explore the entire finite graph.)

```
def Query($i \in \mathcal{I}$)
  do % until result unmarked
    RunAgenda()
    $\langle v, m \rangle$ ← Lookup($i$)
   while $m$
  return $v$
```

Listing 2.8: A replacement for the Query method, handling guesses during backward-chaining.

2.5.2.2 vs. Diamonds

Combining *optional* memoization and the arbitrary guessing behavior of backward chaining as above makes it now possible for two Lookup-s of the same item to return different values, despite the careful commitment of the guesses to the memo table and the construction of updates. It is instructive to consider an example in detail to study the interplay of different facets of our system.

Example 18: Consider the small circut and evolution of the solver's state depicted in figure 2.11. Initially, only the input node `h` is memoized, there are no messages pending, and there are no obligations. Suppose that the user invokes Query(`k`) and that the Lookup(`j`) recurses all the way to `h` and so returns 1, leaving a new memo in its wake only at j. Now suppose that Lookup(`j′`) aborts at `i`, guessing the value 0, and thus memoizing and marking 0 at j'. The item k, which, absent any messages on the agenda, is guaranteed to take on an *even* value, now takes on an *odd* value. ◊

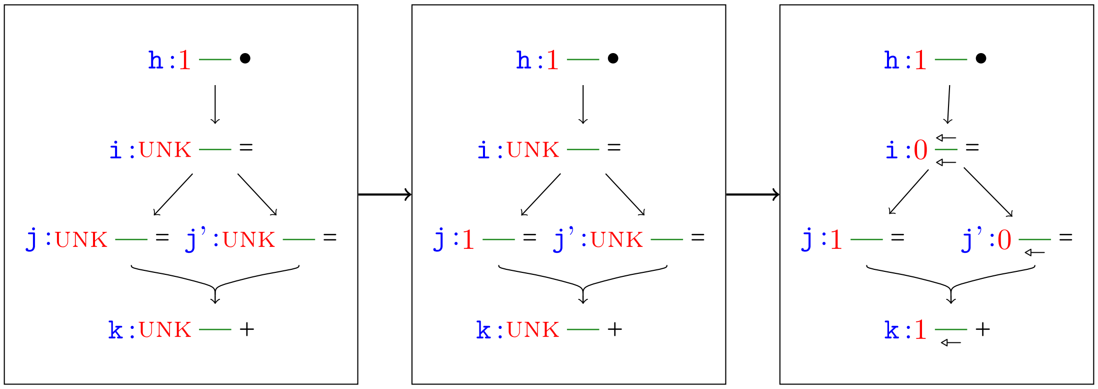


Figure 2.11: A circuit demonstrating two different results from Lookup due to guessing.

As we can see, generalizing from the example, any such *discrepant* value resulting from guessing is going to be marked (in the example, `i` was already obligated to `k` via `j` when the query from `j′` caused a guess). The notifications queued to the agenda do not carry the old values of §2.4.1 (because they are, in general, not knowable when the solver guesses), so the next messages sent to an item having observed a guessed value will not carry the old value. This implies that baggregation or other aggregation strategies which consider only values will, in general, be unable to respond except to discard their state and start over. Thus, we should prefer to handle these marked values within an aggregation strategy's internal state *by name*, so that they can be found and revised later (recall §2.3.3.1).

2.5.2.3 vs. Message-Passing

In a traditional backward chaining system, the (implicit) call stack keeps track of the path from the inducing query to the item being currently explored. Cycle detection is relatively easy in this case as one may the stack to check which items are actively being queried.

In §2.3.1, it was suggested that the message-passing framework itself should track knowledge of which agents were waiting on results from others. In such a scenario, cycle detection is then part of the framework, and so the framework must be responsible for obtaining guesses for cycled items (and perhaps, more generally, responsible for inducing guessing at all) and must synthesize messages appropriately.

2.5.3 Disallowing Self-loops

During acyclic PROPAGATE-ion, the fact that we left the notification on the agenda until the very end meant that however Compute behaved when considering children, even if there were diamonds in the graph, the marking strategy of §2.2.4.2 would ensure correctness; marks were only discarded within PROPAGATE itself, and we were sure that, having visited all children, the notification had served its purpose.

Were we to attempt to PROPAGATE a message around a *self-loop*, i.e., from an item to itself, two problems arise. First, it might be incorrect to discard the notification at the end of PROPAGATE-ion. That is, the act of PROPAGATE-ion might create a new notification which should be PROPAGATE-d, too. Second, because PROPAGATE-ion is insensitive to the order of children, it is possible that this new notification could clobber the existing one mid-way through the loop. While it may be possible to remedy these problems by clever programming (by, say, always traversing a self-loop last, among other fixes), we would prefer a simpler approach for expository purposes.

The simplest fix is to prohibit self-loops entirely. Thankfully, this can be done without loss of expressive power by a simple transformation of the graph. If ever $i \in P_i$, then we *add* a new node i' with $\vec{P}_{i'} = i$ and with $e(i')(\langle v_i \rangle) = v_i$, and we *replace* each i in $\vec{P}_i$ with i'. This new item i' serves as a placeholder for messages traversing the self-loop. As a memo at i' would be redundant with the memo of i, we should never memoize i' and, therefore, should use it to store only *notifications*.

2.5.4 Cyclic Obligation

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 could be done by using the online graph reachability algorithm given in [115]. This entails deriving not *values* but *formulas* justifying the values of derived items. As pertains specifically to obligation in the cyclic case, this suggests that we require transitively-obligated items i to watch only those descendants which are memoized or *reachable from but not within* the strongly-connected component of the graph containing i. This prevents the formation of self-supporting caches of obligation.

2.5.5 Consistency of Multiple Solutions

As alluded to earlier, cyclic circuits do not, in general, have a guaranteed unique solution. If there are no solutions, forward chaining will run forever, as some item will always be inconsistent with its parents. If there are more than one, the solver may *wander* between them. The simplest case of a wandering solver is one that simply resets the memo table to store UNK for all derived items. Having done so, any cycles in the circuit are fair game to be explored and guessed differently the next time around.

Example 19 (*Latches*): We can construct cyclic circuits which (imperfectly) record ephemeral states of the solver. We informally call these circuits "latches," as, like electrical circuits of the same name, they hold their value until forcibly reset, even after the initial stimulus is gone. Consider, for example, a circuit in which k contains a *self-loop* and a parent item j with its own unique parent, the root item i. Let $e(k)$ be OR and $e(j)(\langle v_i \rangle) = v_i < 2$. Suppose, for simplicity, that we always consider all items obligated and always guess NULL when necessary to break cycles. If the input update stream contains of just $i : \underline{\leftarrow 1}$ and $i : \underline{\leftarrow 2}$, then k will be TRUE if the solver computed (and cached, and did not subsequently FLUSH) j and/or k between the two input updates. ◊

Example 20: More usefully, Expectation Maximization (EM) [43] optimization can be encoded as a cyclic circuit. The "latent" values depend upon the model parameters (and input data), while the parameters depend upon the latent values (and input data). The EM algorithm will converge at any *local* optimum of likelihood (of the latent variables); these local optima correspond to fixed-points of the EM circuit. ◊

We conjecture that a kind of minimality of solution could be guaranteed by mechanism analogous to those just discussed for obligation. It may be sufficient to require every non-NULL item j to justify itself in terms of some non-NULL ancestor i outside of the strongly-connected component containing j. (That is, every non-NULL item j must have some *support* from parents that are not participating in cycles with j. This would ensure that items whose ancestry is *exclusively* cyclic—an "all roads lead *back to* Rome" scenario—are assigned the value NULL in every solution.) However, such a mechanism would be unable to distinguish between multiple fixed-points that differed only in their *non*-NULL values.

2.5.5.1 vs. Snapshots

When combining cyclic programs with the snapshot mechanism of §2.2.3.4, it may be desirable to guarantee **stability** of items, either within or across causally-related[63] snapshots of the circuit's state. In the absence of a stability guarantee, the result v from $\texttt{Query}(i)$ means merely that "there exists a solution in which i takes on value v." The result v' of a subsequent $\texttt{Query}(i')$ could come from another fixed point entirely; so even if the circuit requires that $i' = i$, v need not equal v'! Stability *within* a snapshot would eliminate this possibility so long as the Query-s were directed at the same snapshot.

A particularly simple policy that a snapshot-supporting solver may wish to employ is to ensure that any cycle of items which contains at least one memoized item at the time of a snapshot, or comes to contain one afterwards, is never, subsequently, entirely flushed. This policy has the effect of making (the queried portion of) the circuit effectively *acyclic*. Once a memo is created so that $\texttt{Compute}(i)$ does not revisit i, because cycles may not be completely flushed, no subsequent Query of this snapshot can cause $\texttt{Compute}(i)$ to revisit i again. This invariant remains true even if precisely *which* items on a cycle are memoized is not constant across time, so long as those memos are the result of Lookup (and not guessing). The use of guessing within a snapshot seemingly must be restricted to the essential case of cycle-breaking. We have not found an invariant of the solver's memo table and/or agenda which guarantees stability if guessing is used arbitrarily.

A stronger notion of stability prohibits changes to the value of an item unless the change can be "justified by the update stream." That is, querying the same item j in two different, causally-related snapshots must return the same value unless the intervening updates include an update to j's ancestry. If one is using snapshots to perform counterfactual reasoning, this is a highly desirable property: the only changes to the circuit will be those actually caused by the updates made between two snapshots.

2.5.6 Programs

In the next chapter, we turn our attention to the problem of extending the algorithms presented here to work not on arithmetic circuit descriptions directly but on Prolog-like weighted rules of Datalog with Aggregation [173, 82, 32] and Dyna [49]. These programs

[63] For a given instance of a circuit, snapshot s_2 is causally related to s_1 if s_2 has seen a superset of the updates seen from s_1 from the driver program: that is, s_2 is some future of s_1; there may be more than one if the driver is permitted to fork the update stream to explore multiple changes, as mentioned at the end of §2.2.3.4.

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.

2.6 Related Work

The constraint solver Kangaroo [135] was independently motivated by very similar concerns. Like our algorithm, 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. Our algorithm is potentially more selective about storage than Kangaroo, which stores memos at *all* nodes of the circuit.[64] 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, Blelloch, and Harper [2] and Acar and Ley-Wild [4] 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 (including cycles with aggregation and negation) by Saha [160] and Swift and Warren [170], who contrast it with the "DRed" ("Delete and REDerive") strategy that forward-chains *invalidation* updates [85]. Saha [160] gives a refinement to DRed in which the system watches (in the sense of the watched-variable trick) only the *acyclic* derivations of an item, filtering out extraneous deletions (which would be reversed upon re-derivation). The "magic sets" transformation for Datalog [152] 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, Blelloch, and Harper [3] do separately consider *selective* memoization, but do not handle updates in this more challenging case (see §2.2.4). L10 [163] offers backward chaining across "worlds" but forward-chains to saturation within each world, a kind of coarse-grained magic templates. A different selective strategy [112] relies primarily on *un-memoized* 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.

Hammer et al. [87] considers a ML-like language ("NOMINAL ADAPTON") in which the programmer may give explicit, first-class *names* for particular computations. These names are akin to our names of items within the circuit, but also encode the edges of the DAG: the name of a child item is *a function* of the name of its parents. Thus, NOMINAL ADAPTON's memo table is not, like many prior efforts at incremental computation, *structural*, but rather *nominal*. That is, in memoizing a function f, it does not associate the input *value* x with the cached $f(x)$ value, but rather the *name* of this input. During incremental computation (i.e., forward-chaining to revise stale values), the memo table will

[64] 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 un-memoized parent.

thus provide the old value, which, hopefully, requires only small changes in response to the changes to its input.

We have not discussed static transformations of arithmetic circuits (in contrast to the dynamic transformations of §2.3.3.1), but there is ample literature on the topic and we would be remiss in not mentioning at least some of it. One may wish to transform for local arithmetic efficiency [184], numeric stability [120, 139], or broader rewrites [48]. Thankfully, while related, such work is orthogonal to the thrust of this thesis.

Chapter 3

Towards Backward Chaining Weighted Logic Programs

> Prolog is so simple that one has the sense that sooner or later someone had to discover it. … We benefitted from freedom of action in a newly created scientific center and, having no outside pressures, we were able to fully devote ourselves to our project.
>
> Undoubtedly, this is why that period of our lives remains one of the happiest in our memories. We have had the pleasure of recalling it for this paper over fresh almonds accompanied by a dry martini.
>
> Alain Colmerauer and Philippe Roussel, *The Birth of Prolog.* [34]

In the prior chapter, we introduced EarthBound, a flexible algorithm to handle queries and updates on *finite*, possibly cyclic circuits. The Compute procedure of this algorithm computes a given, *single* item'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, the analogous task is one step of backward-chained reasoning, which computes the values of a given *set* κ of items, such as might be specified by a non-ground term, by consulting the *rules* of the program and combining values from parent items as directed. The present chapter addresses this surprisingly challenging problem, culminating in the pair of Compute(κ) procedures of listings 3.2 and 3.3. These return representations of a *piecewise-constant* map from all items in κ to their values.

Throughout this chapter, except occasionally where otherwise indicated, we will consider *pure backward-chaining*, without any consideration of notifications (§2.2.3.3) or marked values (§2.2.4.2). The exclusion of marked values also precludes guessing during backward chaining; recall §2.5.2. Extending the algorithms in this chapter to handle marked values should be straightforward, but we do not do so here, for simplicity of presentation. The *proofs* of this section will also assume the same: we will model the procedural Lookup using an item valuation *function*, so that multiple requests for the same item's value will obtain the same answer.

While programs may specify finite, possibly-cyclic circuits, programs also open the door to infinite circuits. The circuit corresponding to a program may contain infinitely many items, items with infinite in- and/or out-degrees, and infinitely long (acyclic) paths in either direction.[65] The algorithms of §2 clearly will not suffice. However, for all its potential complexity, the circuit retains a *finite description*, in the form of the program, as given. Thus, the rules of the program will come to have a central position in our story, obviating the overtly graphical description encoded in the P_j and C_i sets as well as the $e(j)$ evaluation functions.

Prolog programs specify *boolean* circuits whose items are ground terms from some Herbrand universe $\mathcal{H}$ and whose values are from $\{\texttt{TRUE}, \textsc{null}\}$.[66] To specify general weighted circuits, we generalize Prolog. Rules in our weighted setting now compute and route aggregands to their head items, based on the values of their subgoal items. Thus, we might write "`rs(X) ⊕= r(X,Y) ⊗ s(Y)`" to compute the matrix-vector product of $\texttt{r}^{/2}$ and $\texttt{s}^{/1}$, i.e., $\bigoplus_y \texttt{r}\langle x, y\rangle \otimes \texttt{s}\langle y\rangle$ for each x. We have replaced Prolog's "`:-`" with an aggregator, indicating how the contributions produced by the body are to be collected. The body, meanwhile, is now an *expression tree* of subgoals, rather than being a conjunction thereof. We now introduce a core calculus for weighted logic programming, which we call μDyna. We then consider three forms of backward-chaining reasoning on μDyna programs: *ground* reasoning, in which all collections of items and aggregands are finite, in §3.2; *partition-based* set-at-a-time reasoning, in which answers may be infinite but do not overlap, in §3.3; and, finally, *default-based* set-at-a-time reasoning, in which we more explicitly use a non-monotonic concept of (recursive) defaults, in §3.4.

3.1 μDyna Normal-Form Programs

We define μDyna, ("micro-Dyna") a minimal, set-theoretic, "administrative" normal-form [68] 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*; and ③ a finite *bag* of (μDyna) rules, $\wr \rho_r \mid r \in \Xi \int$, where Ξ is a finite set of rule indices.[67]

3.1.1 μDyna Rules

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 **kv-pairs** of a *key* and a *value* (in that order, i.e., $\langle \text{key}, \text{value}\rangle$). A **rule grounding**, then, is a nested tuple over these components; we will use mnemonic, infix pair formers, so rules render as

$$\{(\text{head} \hookleftarrow \text{result}) \Leftarrow \langle \text{key}_1 \mapsto \text{value}_1, \ldots, \text{key}_n \mapsto \text{value}_n\rangle \mid \cdots\},$$

[65] That is, infinite sets of items $I \subseteq \mathcal{I}$ in which every $i \in I$ can reach (or be reached by) infinitely many other items in I and for which, given any two distinct items $i, j \in I$, either i can reach j or j can reach i.

[66] While it is tempting to use $\{\texttt{TRUE}, \texttt{FALSE}\}$ instead, we prefer the use of $\textsc{null}$ to maintain consistency with the rest of this document.

[67] That is, this third component of the program is the pair of the set Ξ and the Ξ-indexed collection of rules $\rho_\cdot$; the use of a bag is meant to emphasize that rules may be repeated. Occasionally, we shall simply write ρ for an arbitrary rule, just to reduce clutter, when the index r is not of relevance to the discussion. The reader should, however, assume, in such cases, that there is some program tacitly under discussion and that there exists r in that program's Ξ such that ρ is ρ_r.

rather than the more formal

$$\{\langle\langle \text{head}, \text{result}\rangle, \langle\langle \text{key}_1, \text{value}_1\rangle, \ldots, \langle \text{key}_n, \text{value}_n\rangle\rangle\rangle \mid \cdots\}.$$

Each grounding of a μDyna rule reads as an instruction: "contribute ($\hookleftarrow$) the *result* to the *head*, *if* ($\Leftarrow$) each *subgoal's key* has been assigned ($\mapsto$) the corresponding *value*"; groundings which satisfy this condition are called **rule answers**.[68] A subgoal can thus be seen as a *request* for the value of the item named by the key.[69] 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 from above, `rs(X) ⊕= r(X,Y) ⊗ s(Y)`, is now rendered as

$$\rho_r = \{\overbrace{(\underbrace{\mathtt{rs}\langle x\rangle}_{\text{HEAD}} \hookleftarrow \underset{\llcorner\text{RES}}{z})}^{\text{HR}} \Leftarrow \overbrace{\langle \mathtt{r}\langle x, \underset{\llcorner\text{SG.1.1.2}}{y}\rangle \mapsto v, \mathtt{s}\langle \underset{\llcorner\text{SG.2.1.1}}{y}\rangle \mapsto w, \otimes\langle v, w\rangle \mapsto \underset{\llcorner\text{SG.3.2}}{z}\rangle}^{\text{SG}} \mid v, w, x, y, z \in \mathcal{H}\} \tag{3.1}$$

We have annotated the rule with several paths and given mnemonics to particular prefixes, $\text{HR} \overset{\text{def}}{=} 1$, $\text{HEAD} \overset{\text{def}}{=} 1.1$, $\text{RES} \overset{\text{def}}{=} 1.2$, and $\text{SG} \overset{\text{def}}{=} 2$, 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 ($\forall_{r\in\Xi}$):

① projections along HEAD, RES, and SG are defined for all elements of the set;

② the head and result are terms, i.e., $\forall_{t\in\rho_r, \pi\in\{\text{HEAD},\text{RES}\}}\, t\downharpoonleft_\pi \in \mathcal{H}$;

③ the number of subgoals in each grounding within ρ_r, is *constant* across all groundings of the rule and denoted n_r; i.e., $\forall_{\vec{s}\in(\rho_r\downharpoonleft_{\text{SG}})}\, \text{tlen}(\vec{s}) = n_r$;

④ each subgoal is itself a pair of two terms, so $\forall_{t\in\rho_r, i\in\mathbb{N}_1^{n_r}, j\in\{1,2\}}\, t\downharpoonleft_{\text{SG}.i.j} \in \mathcal{H}$; and

⑤ the subgoals and head determine the grounding, i.e., $\forall_{\alpha\subseteq\rho_r} |\alpha\downharpoonleft_{\text{SG}}| = |\alpha\downharpoonleft_{\text{HEAD}}| = 1 \Rightarrow |\alpha| = 1$ (and, in particular, that $|\alpha\downharpoonleft_{\text{RES}}| = 1$).

These clearly hold for equation (3.1) 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 $\forall_{\alpha\subseteq\rho_r} |\alpha\downharpoonleft_{\text{SG}}| = 1 \Rightarrow |\alpha| = 1$.

[68] Recall from "Functions and Maps" (in §1.3) that $\mapsto$ is simply an infix pair constructor; despite its frequent use in defining functions as sets of pairs obeying functional a dependence, here, we use it merely as a mnemonic between subgoal key and value even though there is no functional dependence within the structure of the rule itself. However, once the rule is refined down to a set of *answers*, there will, indeed, be a functional dependence between the tuple of all subgoal keys and corresponding values.

[69] 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. (See §5 for more about mode analysis.)

Example 21: A pure Prolog rule behaves similarly, with ⊕ being logical $\texttt{OR}$ and ⊗ being $\texttt{AND}$. The result ($\textsc{res}$) and all subgoal value ($\textsc{sg}.k.2$, with $k \in \mathbb{N}$) paths have singleton projections of $\texttt{true}\langle\rangle$. That is, the pure Prolog rule `rs(X) :- r(X,Y), s(Y)` appears in μDyna as $\{(\texttt{rs}\langle x\rangle \hookleftarrow \texttt{true}\langle\rangle) \Leftarrow \langle \texttt{r}\langle x,y\rangle \mapsto \texttt{true}\langle\rangle, \texttt{s}\langle y\rangle \mapsto \texttt{true}\langle\rangle\rangle \mid x,y\}$. ◊

We now introduce several key definitions, which will be expanded through the chapter and reused throughout the rest of this thesis. A quick visual summary of the notation is offered in figure 3.2. A **rule query** $\vec{t}$ for the r^{th} rule is a n_r-tuple of items. A rule query gives rise to a set of **pre-answers** by refining the subgoal keys: $\theta_r^{\vec{t}} \overset{\text{def}}{=} \rho_r[\{t_1\}/\textsc{sg}.1.1]\cdots[\{t_{n_r}\}/\textsc{sg}.n_r.1]$. $\vec{t}$ is **trivial** if $\theta_r^{\vec{t}} = \varnothing$. Given an item valuation function $v \in \mathcal{I} \to \mathcal{H}'$, with $\mathcal{H}' \overset{\text{def}}{=} \mathcal{H} \cup \{\textsc{null}\}$ (the addition of $\textsc{null} \notin \mathcal{H}$ may seem curious; it will be explained momentarily, in §3.1.2), one can filter pre-answers to the set of rule answers (as defined, informally, at the start of this section); that is, having refined the *keys* of the subgoals, we can now additionally refine the *values*: $\epsilon_{r,v}^{\vec{t}} \overset{\text{def}}{=} \theta_r^{\vec{t}}[v(t_1)/\textsc{sg}.1.2]\cdots[v(t_{n_r})/\textsc{sg}.n_r.2]$.[70] If any $v(t_i) = \textsc{null}$, then ϵ is $\varnothing$. As all subgoal projections have been refined to singletons within rule answers, the constraints on μDyna rules imply that $\forall_{h,r,\vec{t},v}\ |\epsilon_{r,v}^{\vec{t}}[\{h\}/\textsc{head}]| \le 1$: every rule query has at most one corresponding answer per head.

Example 22: Consider again our example rule from equation (3.1). A rule query for this rule is a $n_r = 3$-tuple of terms. One example would be an element of $\{\langle \texttt{r}\langle 1,2\rangle, \texttt{s}\langle 3\rangle, t\rangle \mid t\}$, which would be *trivial*, as it attempts to refine $\textsc{sg}.2.1.1$ simultaneously to both $\{2\}$ and $\{3\}$. On the other hand, $\vec{t} = \langle \texttt{r}\langle 1,2\rangle, \texttt{s}\langle 2\rangle, 4 \otimes 5\rangle$ is a non-trivial rule query: its pre-answers are

$$\begin{aligned}\theta_r^{\vec{t}} &= \rho_r[\{\texttt{r}\langle 1,2\rangle\}/\textsc{sg}.1.1][\{\texttt{s}\langle 2\rangle\}/\textsc{sg}.2.1][\{\otimes\langle 4,5\rangle\}/\textsc{sg}.3.1] \\ &= \{(\texttt{rs}\langle 1\rangle \hookleftarrow z) \Leftarrow \langle \texttt{r}\langle 1,2\rangle \mapsto 4, \texttt{s}\langle 2\rangle \mapsto 5, \otimes\langle 4,5\rangle \mapsto z\rangle \mid z \in \mathcal{H}\}\end{aligned}$$

If v assigns anything other than 4 to $\texttt{r}\langle 1,2\rangle$ and/or anything other than 5 to $\texttt{s}\langle 2\rangle$, then $\epsilon_{r,v}^{\vec{t}} = \varnothing$. On the other hand, suppose that v chooses 4 for $\texttt{r}\langle 1,2\rangle$ and 5 for $\texttt{s}\langle 2\rangle$ and 20 for $\otimes\langle 4,5\rangle$. Then $\epsilon_{r,v}^{\vec{t}} = \{(\texttt{rs}\langle 1\rangle \hookleftarrow 20) \Leftarrow \langle \texttt{rs}\langle 1,2\rangle \mapsto 4, \texttt{s}\langle 2\rangle \mapsto 5, \otimes\langle 4,5\rangle \mapsto 20\rangle\}$. ◊

Example 23: It is not always the case that the rule answer sets are singletons. Consider the rule $\{(\texttt{f}\langle x\rangle \hookleftarrow z) \Leftarrow \langle \texttt{a}\langle\rangle \mapsto z\rangle \mid x \in \tau, z \in \mathcal{H}\}$, for some τ. The only non-trivial rule query for this rule is $\langle \texttt{a}\langle\rangle\rangle$ and, for that query, the set of pre-answers is equal to the rule as a whole. Supposing that v assigns the value 1 to $\texttt{a}\langle\rangle$, then the rule answer set is $\{(\texttt{f}\langle x\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{a}\langle\rangle \mapsto 1\rangle \mid x \in \tau\}$, which is only a singleton when τ is. ◊

Paralleling the treatment in §2.1.2, items that are also to be found in a rule's head are said to be **derived**, and denoted $\mathcal{I}_{\text{der}} \overset{\text{def}}{=} \mathcal{I} \cap \bigcup_{r \in \Xi} \rho_r\!\downharpoonleft_{\textsc{head}}$. The other items, not in the head of any rule of a program, are **input** and denoted $\mathcal{I}_{\text{inp}} \overset{\text{def}}{=} \mathcal{I} \setminus \mathcal{I}_{\text{der}}$; their values will be defined by the driver program when the μDyna program is invoked, as a function $\text{inp} \in \mathcal{I}_{\text{inp}} \to \mathcal{H}'$.[71] By definition, the driver may not *directly* set or otherwise influence the

[70] There is no analogue 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.

[71] In practice, input nodes include "built-in" or "primitive" facts of the system, such as integer addition or current heap byte count, as well as the user's input to the logic program. We could view these as rules within the program, but we choose to keep the current view, as it makes discussion of reactive programming easier: the program is finite and the input changes values, rather than the program itself.

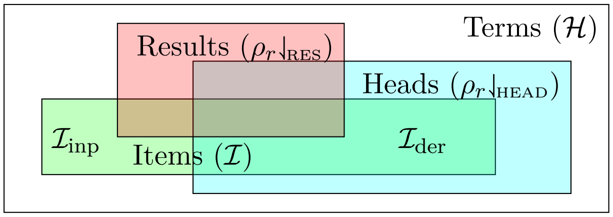
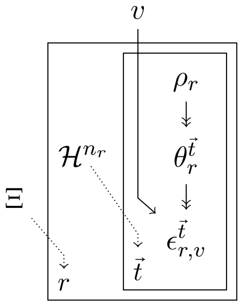

Terms ($\mathcal{H}$)
Results ($\rho_r\downarrow_{\text{RES}}$)
Heads ($\rho_r\downarrow_{\text{HEAD}}$)
$\mathcal{I}_{\text{inp}}$
Items ($\mathcal{I}$)
$\mathcal{I}_{\text{der}}$

Figure 3.1: A schematic view of the term universe and some important subsets defined by a μDyna program.



	Mnemonic	**Type**
Ξ	Program rule indices	Set
v	Item *v*aluation function	$\in (\mathcal{I} \to \mathcal{H}')$
r	*R*ule index	$\in \Xi$
$\vec{t}$	Rule query (*t*uple of *t*erms)	$\in \mathcal{H}^{n_r}$
ρ	*R*ule groundings	
θ	Pre-answers, *th*ese keys	μDyna
ϵ	Answers, *E*valuated	

Figure 3.2: **Plate notation** and mnemonics for our notation for important subsets of a μDyna rule, ρ_r. Plates (solid rectangles) indicate quantification of structure over the variable in their lower-left corner; domains of quantification point to their elements with dotted arrows. Solid arrows entering a plate are fanned-out to each instantiation. Double-tipped arrows indicate that the target is, in addition to being derived from the source, also *a subset* thereof.

value of a derived item; it may only set input items and let the program compute. Figure 3.1 shows a schematic of the term universe and some important subsets thereof.

3.1.2 Aggregation

Rules give rise to contributions to items; what remains, then, is to **aggregate** these contributions into a single value. In order to capture a notion of "no assigned value," especially for derived items with zero contributions (recall §2.2.3), we use the symbol NULL $\notin \mathcal{H}$ and use $\mathcal{H}' = \mathcal{H} \cup \{\text{NULL}\}$ as the result of aggregation. Each item specifies an aggregation function, or just **aggregator**, which maps its *bag* of contributions, $\wp_+\bar{\mathfrak{U}}_\infty\tau$ for some $\tau \subseteq \mathcal{H}$, to elements of $\mathcal{H}'$. That is, we assume, as part of the program definition, a function $\text{aggr} \in \Pi_{i\in\mathcal{I}}(\wp_+\bar{\mathfrak{U}}_\infty\tau_i \to \mathcal{H}')$ with $\forall_{i\in\mathcal{I}}\, \tau_i \subseteq \mathcal{H}$.[72]

Aggregators f in μDyna additionally obey $f(\varnothing) = \text{NULL}$ and $\forall_{a\in\tau}\, f(\lbag a \rbag) = a$ (with τ as above). The input to an aggregator is termed an **aggregand**, generalizing "summand" and "multiplicand." Because NULL $\notin \mathcal{H}$, it is impossible to define μDyna rules which explicitly manipulate NULL. Thus, NULL acts as an *annihilator* of the conjunction of rules' subgoals: any attempt to refine any path to $\{\text{NULL}\}$ will immediately empty the set of groundings. For most aggregators, an output of NULL implies an empty bag input, and so

[72] Often we tacitly extend τ_i to $\mathcal{H}$ and gloss over the behavior of elements in $\mathcal{H} \smallsetminus \tau_i$. In §5.2 we consider *static analysis* to demonstrate that no such values arise. We may thus speak of "summation of reals" as an aggregator, taking $\tau_i = \mathbb{R} \subseteq \mathcal{H}$ and neglecting non-real inputs.

NULL is often taken to be synonymous with “has no aggregands.”

It is occasionally convenient, for simplicity of theory, to extend the domain of aggregators to include NULL elements. That is, we enlarge their domains from $\wp_+\bar{\mathfrak{U}}_\infty\tau$ to $D = \wp_+\bar{\mathfrak{U}}_\infty(\tau \cup \{\text{NULL}\})$. On the extension, we define $\forall_m\, f(\lbag \text{NULL}@m \rbag) = \text{NULL}$, which completely defines the behavior of aggregators on NULL. In practice, NULL is never manifest and so will not be provided to aggregators’ implementations.

3.1.2.1 Aggregators Built From Semigroups

We often build aggregators out of commutative semigroups and, in particular, commutative monoids. Thus, we will speak of values being aggregated by summation or minimization or similar; formally, we mean to be using the corresponding AC-reducers (recall footnote 23, in §1.3). In the case of semigroups, there is no good choice for $f(\varnothing)$, so absent NULL, it is not clear what we would use for that image. For monoids, the obvious choice is the monoid identity element, 0. The existence of NULL, and our insistence that $f(\varnothing)$ is NULL rather than 0, thus distinguishes the cases of “has no aggregands” and “aggregands aggregate to a unit value,” which turns out to be a useful distinction when an item’s aggregator differs from those of the items upon which it depends. Consider a child item j aggregated by minimization with a parent item i aggregated by summation. If i having no contributions from its parents meant that it had value 0, the minimization would include 0 in its input, despite there being no path from the inputs to justify this 0 value.[73]

One can imagine lifting the associative and commutative properties of a commutative semigroup up to AC-reducing structure on aggregators with domains extended to include NULL, as above. On the domain extension we find that f being an AC-reducer implies that $f(\lbag \text{NULL} \rbag) = f(\lbag f(\varnothing) \rbag) = f(\lbag f(\varnothing) \rbag \cup \varnothing) = f(\varnothing) = \text{NULL}$ and a simple induction shows that $f(\lbag \text{NULL}@m \rbag) = \text{NULL}$ for $m < \infty$. The remaining constraint, that $f(\lbag \text{NULL}@\infty \rbag) = \text{NULL}$, is all that is necessary to make NULL an identity of f, regardless of its multiplicity.

3.1.3 Additional Examples

Our `rs` example above, as rendered in equation (3.1), highlights several typical features of μDyna rules.

① There can be covariance between (subterms of) subgoals’s keys (y); the definitions above ensure that all answers will take y within the *intersection* of the active domains of the subgoals of which y is a subterm. This is analogous to a database equi-join.

② Values’ covariance is not syntactically restricted. That is, values are free to be reused as (subterms of) keys or values of (other) subgoals. 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., [32, 51].

[73] The first edition of Dyna [51] restricted the entire program to operate with *a single* semiring, whose additive monoid was used for aggregation, side-stepping any confusion resulting from, as we would frame it, conflating NULL and the additive identity.

③ The result of a rule (RES) is often in covariance with the value of a subgoal ($\otimes$). Many typical μDyna rules partition their subgoals into "search" components ($\texttt{r}$,$\texttt{s}$) and an arithmetic evaluation tree over the results of this search ($\otimes$); however, this is not *required* by the formalism.

We briefly exhibit some other cases that emerge.

Example 24 (*Axiomatic Rules*): Rules with no subgoals, e.g., $\{(\texttt{f}\langle x\rangle \hookleftarrow 1) \Leftarrow \langle\rangle \mid x \in \mathcal{H}\}$, are termed **axiomatic** because they are independent of the rest of the program and form the base cases for forward reasoning. The only possible rule query for such a rule is $\langle\rangle$. While μDyna permits head-result covariance in general, e.g., $\{(\texttt{f}\langle x\rangle \hookleftarrow x) \Leftarrow \langle\rangle \mid x \in \mathcal{H}\}$, the formalisms we develop below will severely restrict the applicability of such rules (see §3.2.1 and §3.3.2.1). Extending our solvers to handle these rules in more general ways remains an open question. $\Diamond$

Example 25 (*Value-Value Covariance*): The rule $\{(\texttt{eq}\langle\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{a}\langle\rangle \mapsto v, \texttt{b}\langle\rangle \mapsto v\rangle \mid v\}$ contributes 1 to $\texttt{eq}\langle\rangle$ iff the program assigns the same values to $\texttt{a}\langle\rangle$ and $\texttt{b}\langle\rangle$. This, too, as with equation (3.1) and as discussed in point ① above, is analogous to a database equijoin, but between columns that are, now, each subject to functional dependence on other columns. $\Diamond$

Example 26 (*Value-Value Relation*): Generalizing, the rule

$$\{(\texttt{rp}\langle\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{a}\langle x\rangle \mapsto v_a, \texttt{b}\langle y\rangle \mapsto v_b, \texttt{r}\langle v_a, v_b\rangle \mapsto \texttt{TRUE}\rangle \mid \cdots\}$$

contributes 1 to $\texttt{rp}\langle\rangle$ for each pair $\langle x, y\rangle \in \langle\mathcal{H}, \mathcal{H}\rangle$ such that the values of $\texttt{a}\langle x\rangle$ and $\texttt{b}\langle y\rangle$ are related by $\texttt{r}^{/2}$. A strict separation into search and arithmetic components would require iteration over both $\texttt{a}^{/1}$ and $\texttt{b}^{/1}$, generating pairs $\langle x, y\rangle$ to be checked. However, as we shall see later, being able to consider, e.g., the $\texttt{a}^{/1}$ and $\texttt{r}^{/2}$ subgoals before $\texttt{b}^{/1}$ increases the number of programs on which a solver might terminate: when $\texttt{b}^{/1}$ has infinite support but $\texttt{a}^{/1}$ and, $\forall_x$, $\texttt{r}\langle\{x\}, \mathcal{H}\rangle$ have finite support, this alternate subgoal order will terminate. A similar observation can be made about the "more typical" rule $\{(\texttt{v}\langle\rangle \hookleftarrow v) \Leftarrow \langle \texttt{a}\langle x\rangle \mapsto a, \texttt{p}\langle a\rangle \mapsto v\rangle \mid \cdots\}$: one may consider $\texttt{p}^{/1}$ before $\texttt{a}^{/1}$, reversing the apparent search and evaluation roles. $\Diamond$

Example 27 (*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 $\{(\texttt{f}\langle a\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{a}\langle\rangle \mapsto a\rangle \mid a \in \mathcal{H}\}$ contributes the value 1 to an item determined by *the value of* $\texttt{a}\langle\rangle$. A more complex example is $\{(\texttt{f}\langle a\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{a}\langle x\rangle \mapsto a\rangle \mid a, x \in \mathcal{H}\}$ whereby, e.g., $\texttt{f}\langle 3\rangle$ gains an aggregand of 1 for each value of x such that, according to $\mathcal{S}$ (or, ultimately, LOOKUP), $\texttt{a}\langle x\rangle$ has value 3. $\Diamond$

Example 28 (*Naked Heads and Subgoals*): It is conventional, but not *required*, that heads and subgoal keys explicitly state their outer-most functor. Within the formalism, we are free to consider rules which *evaluate* part of their head, as in $\{(\texttt{eval}\langle a\rangle \hookleftarrow x) \Leftarrow \langle a \mapsto x\rangle \mid a, x \in \mathcal{H}\}$. This rule contributes to each $\texttt{eval}\langle a\rangle$ ($\texttt{eval}^{/1}$ is a symbol as any other, not a keyword of the language) an aggregand of *the value of* the item specified by a. Rules without explicitly-given head functors are also possible, as in the bizarre $\{(a \hookleftarrow x) \Leftarrow \langle \texttt{a}\langle x\rangle \mapsto a\rangle \mid a, x \in \mathcal{H}\}$, where *each item* a gains an aggregand of x when the item $\texttt{a}\langle x\rangle$ has value a. This seemingly-bizarre ability to omit explicit functors has some practical utility: we can, with a single pair of rules, *test* whether *any given* item's value is NULL by (ab)use of the aggregation machinery. The pair of rules $\{(\texttt{null}\langle x\rangle \hookleftarrow \texttt{true}\langle\rangle) \Leftarrow \langle\rangle \mid x \in \mathcal{H}\}$ and $\{(\texttt{null}\langle x\rangle \hookleftarrow \texttt{false}\langle\rangle) \Leftarrow \langle x \mapsto v\rangle \mid v, x \in \mathcal{H}\}$, together with AND aggregation of $\texttt{null}\langle\mathcal{H}\rangle$ items (again, $\texttt{null}^{/1}$ is just a symbol, not a keyword), results in $\texttt{null}\langle x\rangle$ having value $\texttt{true}\langle\rangle$ exactly when the item x has value NULL. ◊

3.1.4 μDyna B-Hypergraph Semantics

A μDyna program specifies the structure of a labeled B-hypergraph (recall §2.1.2.2) with nodes $\mathcal{I} \subseteq \mathcal{H}$ and hyperedges $\mathcal{E} = \Sigma_{r\in\Xi}\{\langle h, \vec{t}\rangle \mid \forall_{i\in\mathbb{N}_1^{n_r}}\, t_i \in \mathcal{I}, h \in \mathcal{I} \cap (\theta_r^{\vec{t}}\downharpoonleft_{\mathrm{HEAD}})\}$, each of which corresponds to a nontrivial rule query $\vec{t}$, for the rule at index r, and a head item h to which its value would contribute. (Recall θ's definition as the set of pre-answers in §3.1.1.)[74] The use of the dependent sum Σ_r permits multiple hyperedges with the same $\langle h, \vec{t}\rangle$ pair and annotates each with its rule index r. For a given $e = \langle r, \langle h, \vec{t}\rangle\rangle \in \mathcal{E}$, its target is $\mathrm{targ}(e) \overset{\mathrm{def}}{=} h$; its sources are $\mathrm{srcs}(e) \overset{\mathrm{def}}{=} \{\vec{t}\downharpoonleft_k \mid k \in \mathbb{N}_1^{n_r}\}$. The target is said to be a **child** of each of the sources; each source is, in turn, a **parent** of the target. The vocabulary of "Relative Nomenclature" (in §2.1.2) continues to apply.

To this hypergraph we attach labels, augmenting it with functions $l^{\mathrm{e}}_{\cdot} \in \mathcal{E} \to \mathcal{H}'$ (edge labels) and $l^{\mathrm{n}}_{\cdot} \in \mathcal{I} \to \mathcal{H}'$ (node labels). The edge labeling function is *by hyperedge index* $e \in \mathcal{E}$ rather than by source and target collections: there may be two hyperedges with the same sources and targets but with different labels. Labels of one sort are said to be *consistent* with labels of the other sort when the first is the result of particular computations over the second. Given an edge $\langle r, \langle h, \vec{t}\rangle\rangle \in \mathcal{E}$ and a node labeling function $l^{\mathrm{n}}_{\cdot}$, the edge label consistent with $l^{\mathrm{n}}_{\cdot}$ is $\mathrm{selt}(\epsilon^{\vec{t}}_{r,l^{\mathrm{n}}_{\cdot}}[\{h\}/\mathrm{HEAD}]\downharpoonleft_{\mathrm{RES}})$, if that exists, and NULL otherwise (ϵ is the set of rule answers, as defined in §3.1.1).[75] For a node $h \in \mathcal{I}_{\mathrm{der}}$, we define its **yield bag** using its incoming hyperedges and an edge-labeling function $l^{\mathrm{e}}_{\cdot}$: $\mathrm{yb}(\mathcal{E}, h, l^{\mathrm{e}}_{\cdot}) \overset{\mathrm{def}}{=} \lbag l^{\mathrm{e}}_{e} \mid e \in \mathcal{E}, \mathrm{targ}(e) = h \rbag$. That is, each hyperedge's label is an aggregand to its sole target node and is derived from the labels of its source nodes. The consistent l^{n}_{h} is $\mathrm{aggr}(h)(\mathrm{yb}(\mathcal{E}, h, l^{\mathrm{e}}_{\cdot}))$, where, as per §3.1.2, we take the domain of $\mathrm{aggr}(h)\cdot$ to be $\wp_{+}\bar{\mathfrak{U}}_{\infty}\mathcal{H}$. Input nodes are consistently labeled only when their labels are equal to those assigned by the given $\mathrm{inp}(\cdot)$.

Example 29: Continuing our running matrix-vector product example, we see that we will have a node in the hypergraph for each $\texttt{rs}^{/1}$, $\texttt{r}^{/2}$, $\texttt{s}^{/1}$, and $\otimes^{/2}$ item. There is an edge for each nontrivial rule query and head, e.g., $e = \langle r, \langle \texttt{rs}\langle 1\rangle, \langle \texttt{r}\langle 1, 2\rangle, \texttt{s}\langle 2\rangle, \otimes\langle 3, 4\rangle\rangle\rangle\rangle \in \mathcal{E}$, with

[74] This is a little subtle; the hyperedges are not in correspondence with the rule groundings or even the pre-answers subset thereof: the degrees of freedom accorded by values is not reflected in the collection of hyperedges. Further, because the purpose of this hypergraph is to give meaning to an item valuation function given *later*, we can not hope to use the rule answer sets corresponding to *some particular* valuation function in its construction. Thus, we are left with the use of rule *query* as part of the identity of a hyperedge, and, in fact, find that this means that the identity of the subgoal *keys* that matter, which is in concordance with our desire to later consider an item valuation function. The head must additionally be specified, as there may be paths therein that are not constrained by specification of the subgoal keys. Curious readers are invited to see additional discussion of the hypergraph encoding given in §6.3.

[75] That is, when the set $\epsilon^{\vec{t}}_{r,l^{\mathrm{n}}_{\cdot}}[\{h\}/\mathrm{HEAD}]$ is empty. This set never has cardinality more than 1, as per constraint ⑤ of §3.1.1. Recall that $\mathrm{selt}(\cdot)$ is defined, as per "Sets" (in §1.3), to be the function which maps a singleton set to its element, i.e., $\mathrm{selt}(\{x\}) = x$, and that it is undefined on non-singleton sets..

r standing for this rule's index within Ξ. If the program assigns $\mathtt{r}\langle 1,2\rangle \mapsto 3$, $\mathtt{s}\langle 2\rangle \mapsto 4$, and $\mathtt{f}\langle 3,4\rangle \mapsto 12$, then the rule answer set corresponding to e is $\{(\mathtt{rs}\langle 1\rangle \leftarrowtail 12) \Leftarrow \langle \mathtt{r}\langle 1,2\rangle \mapsto 3, \mathtt{s}\langle 2\rangle \mapsto 4, \otimes\langle 3,4\rangle \mapsto 12\rangle\}$, a singleton set. The edge indexed by this rule query is thus consistently labeled by 12, thereby contributing 12 to $\mathtt{rs}\langle 1\rangle$. If the program assigns other values to either $\mathtt{r}\langle 1,2\rangle$ or $\mathtt{s}\langle 2\rangle$, the corresponding rule answers will be $\varnothing$, causing the edge to be labeled NULL. $\lozenge$

Using these definitions, we can define $\mathrm{nl} \in (\mathcal{E} \to \mathcal{H}') \to (\mathcal{I}_{\mathrm{inp}} \to \mathcal{H}') \to \mathcal{I} \to \mathcal{H}'$ to take edge-labeling and input-labeling functions and build a node labeling function that is everywhere consistent with the given functions. In the other direction, we have $\mathrm{el} \in (\mathcal{I} \to \mathcal{H}') \to \mathcal{E} \to \mathcal{H}'$. We can define $T(l) = \mathrm{nl}(\mathrm{el}(l))(l|_{\mathcal{I}_{\mathrm{inp}}})$, $T \in (\mathcal{I} \to \mathcal{H}') \to \mathcal{I} \to \mathcal{H}'$, which sends one node-labeling function to another; $T(l)$ returns a node labeling function in which all children have values that are consistent with their parents' values from l (and input items have the same values they did in l).

A **solution** is a node-labeling function $l^{\mathrm{n}}_{\cdot}$ which is everywhere consistent with edge labels $l^{\mathrm{e}}_{\cdot}$ themselves consistent with $l^{\mathrm{n}}_{\cdot}$. That is, solutions are *fixed points* of T consistent with the input (as they were in §2.5.1). The update messages of pure forward chaining (§2.2.3) can be thought of as identifying nodes n where $l^{\mathrm{n}}_{n} \neq ((\mathrm{nl} \circ \mathrm{el})(l^{\mathrm{n}}_{\cdot}))(n)$. As before, a given program and input pair may have zero, one, or many solutions.

While intentionally informative, these definitions do not constitute an algorithm for finding solutions: if the ancestry graph contains cycles or a node with infinite ancestry, then these labeling functions are not necessarily well-founded. For acyclic hypergraphs where all nodes have finite ancestry, on the other hand, this definition is always well-founded, a unique solution exists, and solver algorithms are easily extracted.[76]

3.1.4.1 Aside: Selective Aggregators and Provenance

Consider an item, $\mathtt{goal}\langle\rangle$, whose aggregator corresponds to the maximization semilattice over, say, $\mathbb{N}$. The semantics just given make it apparent that the consistent value for $\mathtt{goal}\langle\rangle$ is the maximum of the aggregands (from $\mathbb{N}$) obtained from rule answers. However, it may seem that we have designed away our ability to find out *which* rule answer(s) contribute that aggregand. In the case of a finite circuit of §2, it would have been easy to design an item evaluation function whose value was *a pair* of the maximum value and the parent providing that maximum.[77]

[76] There is a straightforward generalization to *monotonic* cycles, wherein the aggregators are selective AC-reducers and rules are such that each subsequent un-rolling of the cycle contributes the same (or a de-preferred) value. Pure Datalog programs fall into this special case, as they, by definition, operate with a finite $\mathcal{I}$ and monotonic operators (`AND` and `OR`). If Datalog is extended with *stratified negation*, in which there are no cyclic paths through `NOT` nodes, such programs continue to fall into this special case. Such a circuit decomposes into an *acyclic* relation between strongly connected components of the original circuit; within these components, the monotonic case applies and because the larger structure is acyclic, the solution must be unique.

[77] In the case of ties, one could pick arbitrarily. Unfortunately, arbitrary choices like this resemble inexact values and necessitate much the same kind of handling. It therefore probably behooves the system to *fix* an order of parents for tie-breaking, so that it makes consistent choices throughout time. More generally, one could even add a notion of sets of items to the value space; as the set of items is a finite set, there should be no formal difficulty.

In the case of a program, however, there is no single parent item for any given value. (The hyperedges of the semantics are not given names that can be referenced within the program itself.) However, we can take inspiration from the finite case and make the result of the rules contributing to `goal⟨⟩` a pair of the value and some meta-data about the value's provenance. That is, we could change $\{(\texttt{goal}\langle\rangle \hookleftarrow v) \Leftarrow \langle \texttt{f}\langle x\rangle \mapsto v\rangle \mid v, x\}$ into, e.g., $\{(\texttt{goal}\langle\rangle \hookleftarrow \langle v, x\rangle) \Leftarrow \langle \texttt{f}\langle x\rangle \mapsto v\rangle \mid v, x\}$. We would then change the aggregator for `goal⟨⟩` from maximization to a *lexicographic* ordering which used maximization on the first projection of the pair and *free choice* on the second projection.[78]

3.1.4.2 Aside: No Disjunctive Heads

The μDyna formalism is quite general. It subsumes (pure) Prolog and at least the Datalogs-with-aggregation of which we are aware. Its use of item valuation functions in specifying rule answers provides ample opportunity for non-monotonic reasoning. However, it cannot express any of the logic programming formalisms described as "disjunctive," built up around theories involving *non-definite* Horn clauses (see, e.g., Minker and Seipel [125] for a survey). That is, every aggregand must be clearly routed to *exactly one* head. While in disjunctive logic programming, one might write a rule which reads as "(at least) one `a` or `b` is true if `c` and `d` are true," μDyna has no such facility for nondeterministic routing of aggregands.

3.2 Ground Program Inference

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, $\textsc{Lookup} \in \prod_{\kappa \subseteq \mathcal{H}} \bigcup_{\alpha \in \wp_{\text{fin}}(\kappa \cap \mathcal{I})} (\alpha \to \mathcal{H})$.[79]

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 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 $\texttt{refineRuleSuffix}(r,\texttt{ans})(\sigma \subseteq \rho_r, i \in \mathbb{N}_1^{n_r+1})$ which interpreted a *suffix* of a rule subset σ, i.e., all subgoals at and after the `i`-th position of a possibly already partially refined rule ρ_r, and which then invoked its callback, $\texttt{ans}(t \in \rho_r)$, on each obtained rule answer t. That is, we would invoke $\texttt{refineRuleSuffix}(r, \texttt{ans})(\rho[\{h\}/\text{HEAD}], 1)$, having written this:[80]

[78] The use of free choice to break ties continues to be short of ideal, as in footnote 77. As before, it would behoove the system to fix an ordering, now on $\mathcal{H}^*$, so that it makes consistent choices. Computationally, the escape hatch of using sets is also possibly closed to us, as it interacts poorly with our default reasoning mechanism (to be discussed, in §3.4).

[79] Throughout this chapter, we model Lookup as a *function* even though, in practice, as in §2, it is in fact a *side-effectful procedure*, capable of mutating the memo table and varying its answers with time. We rely on, but suppress from discussion, the mechanism of marked values (§2.2.4.2) to deal with observed inconsistencies between multiple calls while processing a μDyna rule, much as §2.5.2.1 relied on marking while processing an individual item. When we say Lookup herein, an actual implementation should use lookupFromBelow, with all the attendant marking machinery.

[80] The use of a higher-order function returning a recursive function closure (`go`) avoids repetition of the

B. 3.1

```
1 def refineRuleSuffix($r \in \Xi$, ans)
2   return go
3   def go($\sigma \subseteq \rho_r$, $i \in \mathbb{N}_1^{n_r+1}$)
4     if $\sigma = \varnothing$ then return
5     else if $i = n_r + 1$ then ans(selt($\sigma$))
6     else foreach $(k \mapsto v) \in$ Lookup($\sigma\downharpoonleft_{\mathrm{SG}.i.1}$) do
7               go($\sigma[\{\langle k, v \rangle\}/\mathrm{SG}.i]$, $i+1$)
```

The constraints on μDyna rules ensure that the invocation of selt(·) will be well defined: the HEAD and SG projections of σ have been brought to singletons. Moreover, ANS will be called only finitely many times, as each nested loop has a finite domain, by assumption. 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 queries (the tuple $\vec{k}$ of the ks obtained on line 6 of block 3.1) and their corresponding rule answers $\epsilon^{\vec{k}}_{r,\text{LOOKUP}}[\{h\}/\mathrm{HEAD}]$. 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 the constraints on ρ sets ensure that each leaf corresponds to at most one rule answer.

The solver strategy encapsulated in COMPUTE of §2.2 has, as part of the specification of the arithmetic circuit, the full set of parents for each item, and assumes that all of them are relevant to the child being COMPUTE-d. We could attempt to determine the value of an item within a program by extracting the collection of all parents of an item from the logic program, and consider all possible rule queries so arising. However, this is likely to be an *infinite* set (unless all rules' domains of quantification are finite) and, setting that aside, equality constraints (variable reuse) within a rule set means that many rule queries are trivial and so do not contribute values to their head items. REFINERULESUFFIX instead uses the *answers* from one query (i.e., the identities and values of the subset of the queried items with non-NULL values) to guide the next; the set of parent items whose values are LOOKUP-ed is now *data dependent*.

3.2.1 Range Restriction

While correct, the procedure above works only for one item h at a time! Rather than having to iterate over our (finite, but possibly large) item set, we would surely much rather let the facts guide us, à la SLD resolution [110, 179]. If the rules of our program all obey **range restriction** [22], a requirement that the subgoals determine the head, i.e., $\forall_{\alpha \subseteq \rho} |\alpha\downharpoonleft_{\mathrm{SG}}| = 1 \Rightarrow |\alpha\downharpoonleft_{\mathrm{HEAD}}| = 1$,[81] then we can skip the initial selection of the head h and still

parameters r and ANS in the internal recursive call *and* avoids naming the initial parameters to this recursive function at the outer definition.

[81] The usual notion of range restriction is that variables appearing in the head must also appear in a (positive, i.e., not negated) subgoal; the requirement given here is a trivial generalization to our set-based, weighted setting. When combined with our global constraint on μDyna rules, we have $\forall_{\alpha \subseteq \rho} |\alpha\downharpoonleft_{\mathrm{SG}}| = 1 \Rightarrow |\alpha| = 1$, which excludes, e.g., $\{(\mathtt{f}\langle x, y \rangle \hookleftarrow v) \Leftarrow \langle \mathtt{g}\langle x \rangle \mapsto v \rangle \mid y \in \tau, \cdots\}$ when $|\tau| > 1$. Because quantification in μDyna is *typed* and may be over *subsets* of $\mathcal{H}$, we could relax range restriction to be simply that the head is finite, i.e., $\forall_{\alpha \subseteq \rho} |\alpha\downharpoonleft_{\mathrm{SG}}| = 1 \Rightarrow |\alpha\downharpoonleft_{\mathrm{HEAD}}| < \infty$. It appears to be a simple task to push such a change through, though we

be assured that selt$(\cdot)$ is defined.

Within the context of backward-chaining, we are not, as the definition of range restriction suggests, necessarily interested in *all possible* items in a rule head, but rather a subset, the **query set** κ. In §2, we assumed, effectively, that $|\kappa| = 1$, as $\textsc{Lookup}$ could only be applied to one item as a time; here, we relax that assumption. The full brunt of range restriction may similarly be relaxed, if we somehow know *the set of all queries K* that may be asked, to ***K*-sufficiently range restricted** (K-SRR): $\forall_{\kappa \in K} \forall_{\alpha \subseteq \rho[\kappa/\text{HEAD}]} |\alpha\downharpoonleft_{\text{SG}}| = 1 \Rightarrow |\alpha| = 1$. General range restriction is the same thing as $\{\mathcal{H}\}$-SRR. If K is such that $\forall_{\kappa \in K} |(\kappa \cap \mathtt{f}\langle\mathcal{H},\tau\rangle)\downharpoonleft_2| = 1$, then the rule above, $\{(\mathtt{f}\langle x, y\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{g}\langle x\rangle \mapsto v\rangle \mid y \in \tau, \cdots\}$, is K-SRR despite not being generally range-restricted. In the below, we use "range restricted" to mean "$\{\kappa\}$-sufficiently range restricted" for whatever query κ is under consideration.[82]

We now see how to execute a single step of backward reasoning, within a (sufficiently) range restricted program, given some query set $\kappa \subseteq \mathcal{I}$: visit each rule r, compute $\sigma = \rho_r[\kappa/\text{HEAD}]$, and invoke $\texttt{refineRuleSuffix}(r,\ f)(\sigma,\ 1)$, letting its callback f accumulate all obtained results. Connecting this back to our sets, we see that $\texttt{refineRuleSuffix}$ computes the set of rule answers $\epsilon^{\vec{k}}_{r,\textsc{Lookup}}$, by interleaving the refinements given in the definitions of θ and ϵ. All told, the complete code listing, modulo $\texttt{refineRuleSuffix}$ from above, is listing 3.1.

3.2.1.1 Correctness Conditions

The correctness of this procedure does not rely, strictly, on the finiteness of $\mathcal{I}$, so much as the finiteness of the answer maps. If one can ensure that each answer map is finite at initialization, say, by requiring that the driver can only ever assert finitely many input items non-NULL and initializing the memo table such that only a finite subset is non-NULL, then range restriction suffices to ensure that the program is **step-wise finite**. Each additional round of backward-chaining $\textsc{Compute}$-ation, with its internal call to $\texttt{refineRuleSuffix}$, will find only finitely many items with aggregands. Thus, while the program may not halt, each step will complete in finite time.

Even this condition is more strict than necessary. It potentially considers each subgoal of a rule—the source of a query needing an answer map—*in isolation*, while $\texttt{refineRuleSuffix}$ represents a kind of *sideways information-passing* [18], wherein we use the results of earlier subgoals to narrow the queries made for later ones. Thus, we need only the *refined* queries we actually make to have finite answers. Thus, we could imagine running our running $\mathtt{rs}^{/1}$ matrix-vector-product example in a context where $\otimes^{/2}$ items existed for each of the infinitely many facts that hold for multiplication on $\mathbb{R}$, for example. So long as $\mathtt{r}^{/2}$ and $\mathtt{s}^{/1}$ have finite non-NULL support at every timestep, the rule remains finitely productive. Of course, any execution of this rule which began with "first consider each fact about multiplication" (i.e., "begin by looping over $\otimes^{/2}$") would fail to terminate, though it

do not do so, for clarity of exposition.

[82] However, in contrast to range restriction, checking that a program is sufficiently range restricted is no longer a simple syntactic condition on the μDyna program, as it requires the query set K. The existence of a K which is sufficiently large for a program to operate and yet, for which, the program is K-SRR is a *joint constraint* on *all* the rules of the program as well as any queries (and update operations) available to the driver. Static analysis of such properties of a logic program is the purview of *mode analysis* [137]; see §5.

```
1  def Compute($\kappa \subseteq \mathcal{I}_{\mathtt{der}}$) $\in \bigcup_{\alpha \in \wp_{\mathtt{fin}} \kappa} (\alpha \to \mathcal{H})$
2    $y \leftarrow \{k \mapsto \varnothing \mid k \in \kappa\}$ % per-item running union of yield across all rules
3    foreach $r \in \Xi$ do % visit each rule
4      foreach $(k \mapsto \beta) \in$ computeRule($r$,$\kappa$) do % accumulate aggregands
5        $y(k) \leftarrow y(k) \uplus \beta$
6    return $\{k \mapsto v \mid (k \mapsto \beta) \in y, v = \mathsf{aggr}(k)(\beta), v \neq \mathrm{NULL}\}$ % aggregate answer
7
8  def computeRule($r \in \Xi$, $\kappa \subseteq \mathcal{I}_{\mathtt{der}}$) $\in (\kappa \to \mathcal{H}^+)$
9    $c \leftarrow \{k \mapsto \varnothing \mid k \in \kappa\}$ % accumulator for rule contribution
10   refineRuleSuffix($r$, contribRuleAnswer)($\rho_r[\kappa/\mathrm{HEAD}]$, 1)
11   return $c$
12
13   def contribRuleAnswer($g \in \rho_r$) $\in \langle\rangle$
14     let $\langle k, v \rangle = g{\downharpoonleft}_{\mathrm{HR}}$ in $c(k) \leftarrow c(k) \uplus \wr v \int$
```

Listing 3.1: Ground Compute. y is a function storing the *finite* running accumulation of contributions from all rule answers supported by the item answers from Lookup. computeRule is called for each rule of the program and uses refineRuleSuffix (from block 3.1) to find its answers, whose contributions are accumulated within c. (c is not, strictly, necessary within this listing; we could store directly back to y within computeRule, rather than storing to c and taking unions to update y on line 5. However, we use this presentation here as later listings will add additional structure to c.) When an answer is found, contribRuleAnswer is called to update c, having extracted the *singleton* head and result; the notation $f(k) \leftarrow v$ updates the (mutable) function value f for its argument k to yield v in future invocations. Because contribRuleAnswer takes a rule grounding (recall the use of selt$(\cdot)$ on line 5 of block 3.1) and yet refineRuleSuffix is invoked on a rule with potentially non-singleton head projection, this code works only for μDyna programs obeying range restriction.

would indeed enumerate the finite support of the rule (and then spend eternity probing for more answers that will never arrive). Thus, one must *plan* the execution of rules: not all subgoal orderings are viable. In fact, rule subgoals may need to be *reordered* for different queries. Despite the necessity of planning, we gloss over this detail in all subsequent procedural considerations, deferring discussion to §5.3; we assume that every ρ object under consideration is already in suitable order, when it matters, for whatever query is being processed. Our notion of step-wise finiteness is related to the notion of "supersafety" in ASP [114].

While planning can indicate acceptability of many programs that more naïve analysis would reject, there are still programs that might be acceptable but which would require more clever analysis and runtime support. Notable examples include cases where *all* subgoals in isolation have infinite non-NULL support, but taken together with covariances in the rule, one nevertheless has only finitely many solutions to consider. Degenerate examples include $\{\cdots, \langle \texttt{even}\langle x\rangle, 1\rangle, \langle \texttt{odd}\langle x\rangle, 1\rangle \cdots \mid x \in \mathbb{N}\}$, where the intersection of these two indicator functions' domains is empty, but non-empty cases arise, too, such as $\{\cdots, \langle \texttt{a}\langle x, y\rangle, 1\rangle, \langle \texttt{b}\langle x, y\rangle, 1\rangle \cdots \mid x, y \in \mathcal{H}\}$ with $\texttt{a}^{/2}$ having finite first projection of its non-NULL support, and $\texttt{b}^{/2}$ having finite second projection.

Albeit in a primarily *forward-chaining* context, this need for plans to exist for rules, evidencing those rules to be $\mathcal{H}$-sufficiently range restricted, was the underlying requirement on programs of the Dyna 2 solver implemented in 2013 [59].

3.2.1.2 Aside: Aggregation

Delaying aggregation to the end of the operation, as done in listing 3.1, is almost surely computationally inefficient (though there may be occasions in a reactive situation where it is the right answer). In practice, aggregators are often AC-reducers (à la §3.1.2.1) or can be decomposed into functions pre- and post-composed with an AC-reducer; an actual implementation would likely take advantage of this to eagerly aggregate as soon as new aggregands were discovered, though once again inexact values (§1.5) pose a challenge. Eager aggregation would also form the basis of the optimizations discussed in §2.3.5.3. We omit such handling here to avoid incidental complexity and because keeping bags of aggregands is in line with the semantics given earlier.

3.2.2 Improved Lookup Interface

There is no fundamental reason that Lookup takes only the *key* component of each subgoal. In fact, because it has access to memoized values, providing it access to a set of possible values for each queried key within the set can be of computational benefit. Going forward, we will be providing Lookup with the entire subgoal projection, viewing it as $\alpha = \Sigma_{k\in\kappa}\,\tau_k \subseteq \langle\mathcal{H}, \mathcal{H}\rangle$ for some κ and κ-indexed collection τ with each $\tau_k \subseteq \mathcal{H}$. The return from Lookup will be $\sigma \subseteq \alpha$ where $k \in \sigma\downharpoonleft_1$ and $(k \mapsto v_k) \in \sigma$ if the item k has been assigned the value $v_k \in \tau_k$. Lookup now has a more intimidating type: $\Pi_{(\Sigma_{k\in\kappa}\,\tau_k)\subseteq\langle\mathcal{H},\mathcal{H}\rangle} \bigcup_{\alpha\in\wp_{\text{fin}}(\kappa\cap\mathcal{I})} \Pi_{k\in\alpha}\,\tau_k$, but the change to refineRuleSuffix is, thankfully, straightforward: replace Lookup($\sigma\downharpoonleft_{\text{SG}.i.1}$) on line 6 of block 3.1 with Lookup($\sigma\downharpoonleft_{\text{SG}.i}$) (note the shorter projection path).

Example 30 (*Value Filtration*): The rule $\{(\texttt{goal}\langle\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{a}\langle\rangle \mapsto t\rangle \mid t \in \{2,3\}\}$ contributes 1 to $\texttt{goal}\langle\rangle$ if $\texttt{a}\langle\rangle$ has been assigned the value 2 or 3. If we were to pass only the subgoal key $\texttt{a}\langle\rangle$ to Lookup, and its value were not memoized, the solver would be obligated to compute its value recursively. Now, however, we have placed all answers other than 2 or 3 in the same equivalence class as NULL (because an answer value $v \notin \{2,3\}$ will cause the refinement in refineRuleSuffix to produce the empty set). In the case of a cached answer in this excluded set, Lookup can return an empty map (more generally, omit the offending key(s) from its return map). In the case of an un-cached value, these kinds of constraints may still be useful as additional fodder for recursive computation: supposing $\texttt{a}\langle\rangle$ is aggregated by minimization, the solver can *stop* its recursive quest if, say, it finds an aggregand with value 0. The solvers discussed herein, regretfully, do not take advantage of this possibility. ◊

Example 31 (*Inverse Query Modes*): Separately, providing values to Lookup enables **inverse query modes**. Some sets of items, such as addition, can *solve for* a subset of their keys given another subset *and* their value. That is, because there is only one answer x to the equation $x + 2 = 3$, the query (set of kv-pairs) $\{\langle x + 2, 3\rangle \mid x \in \mathcal{H}\}$ has only one answer for Lookup to return. More interestingly, the query $\{\langle x^2, 4\rangle \mid x \in \mathcal{H}\}$ has *two* keys, and so differs fundamentally from $\{\langle \sqrt{4}, x\rangle \mid x \in \mathcal{H}\}$, which could only return from one "branch" of the square root function. These kinds of queries are called "inverse" because we are solving for *keys* rather than values. We will say more about query modes in general in §5.3. ◊

3.2.3 Correspondence of Semantics and Procedural Results

We can formally argue that this procedure implements our semantics. Specifically, the algorithm of listing 3.1 (with the minor tweak from §3.2.2) produces the same results as §3.1.4. Because we are treating Lookup as encoding a function,[83] these proofs apply only in the absence of marked values (recall §2.5.2.2). Thankfully, this condition can be made to hold (with a sufficiently large memo table) when a solution has been obtained, so these proofs still have utility.

We begin with a relatively straightforward characterization of our core procedure, refineRuleSuffix:

Lemma 1 (*refineRuleSuffix overestimates* ϵ)**:** *Assuming* Lookup *encodes a function* f *within listing 3.1, every invocation of the function returned by* refineRuleSuffix(r, _), *with arguments* (σ, i), *within* computeRule(r, κ), *obeys*

$$\exists_{\vec{p} \in \mathcal{H}^{i-1}}\, \sigma \supseteq \bigcup\nolimits_{\vec{t} \in \mathcal{H}^{n_r - i + 1}} \epsilon^{\kappa, \vec{p} \mathbin{+\!\!+} \vec{t}}_{r,f}$$

The superset is equality when $i = n_r + 1$*, which also implies* $\vec{t} = \langle\rangle \in \mathcal{H}^0$*.*

Proof. The prefix tuple $\vec{p}$, while not overtly visible in the argument variables in scope, consists simply of the k values selected by the foreach-es on line 6 of block 3.1 which have (transitively) called the invocation in question. A straightforward inductive argument shows

$$\sigma = \rho_r[\kappa/\mathrm{HEAD}][\{\langle p_1, f(p_1)\rangle\}/\mathrm{SG}.1]\cdots[\{\langle p_{i-1}, f(p_{i-1})\rangle\}/\mathrm{SG}.(i-1)].$$

[83] That is, there is some function f such that that the invocation of Lookup($\Sigma_\kappa\, \tau$) (with κ finite) returns a function f' such that, for every $k \in \mathrm{dom}(f)$, if and only if $f(k) = v \in \tau_k$ is $f'(k)$ defined (and equal to v). If $f(k) \notin \tau_k$, then we require that $k \notin \mathrm{dom}(f')$.

By commutativity of refinement operators, and that

$$\forall_{\alpha,k,v,j}\, \alpha[\{\langle k,v\rangle\}/\textsc{sg}.j] = \alpha[\{k\}/\textsc{sg}.j.1][\{v\}/\textsc{sg}.j.2],$$

we see that σ is ρ_r subject to a subset, improper when $i = n_r + 1$, of the refinements done to $\epsilon_{r,f}^{\kappa,\vec{p}+\vec{t}}$. Refinements can only reduce membership in a set, so the result is immediate. □

This lemma does most of the heavy lifting in proving one direction of the equivalence. The other is sufficiently straightforward that we do not give it a separate lemma. All told, we can demonstrate the following.

Theorem 1: *Assuming* Lookup *encodes a function f,* computeRule*, and therefore* Compute*, in Listing 3.1 implements our semantics for the case of finitely-productive circuits.*

Proof. Every invocation of the function closure go (as returned by computeRule's call to refineRuleSuffix(r, contribRuleAnswer)) with arguments (σ, $n_r + 1$), with $\sigma \neq \varnothing$, corresponds to *exactly one* hyperedge $\langle r, \langle \text{selt}(\sigma\downharpoonleft_{\text{HEAD}}), \vec{p}\rangle\rangle \in \mathcal{E}$ (with $\vec{p}$ the witness to the existential in the above lemma) with label $\text{selt}(\sigma\downharpoonleft_{\text{RES}})$. This edge's impact is recorded by contribRuleAnswer. The selt() within refineRuleSuffix is guaranteed to be defined, by range restriction of the program. Thus, the values accumulated into $c(k)$ by line 14 of listing 3.1 for any $k \in \kappa$ are a sub-bag (possibly improper) of those in the semantics' $\text{yb}(\mathcal{E}[\{r\}/1], k, f)$.

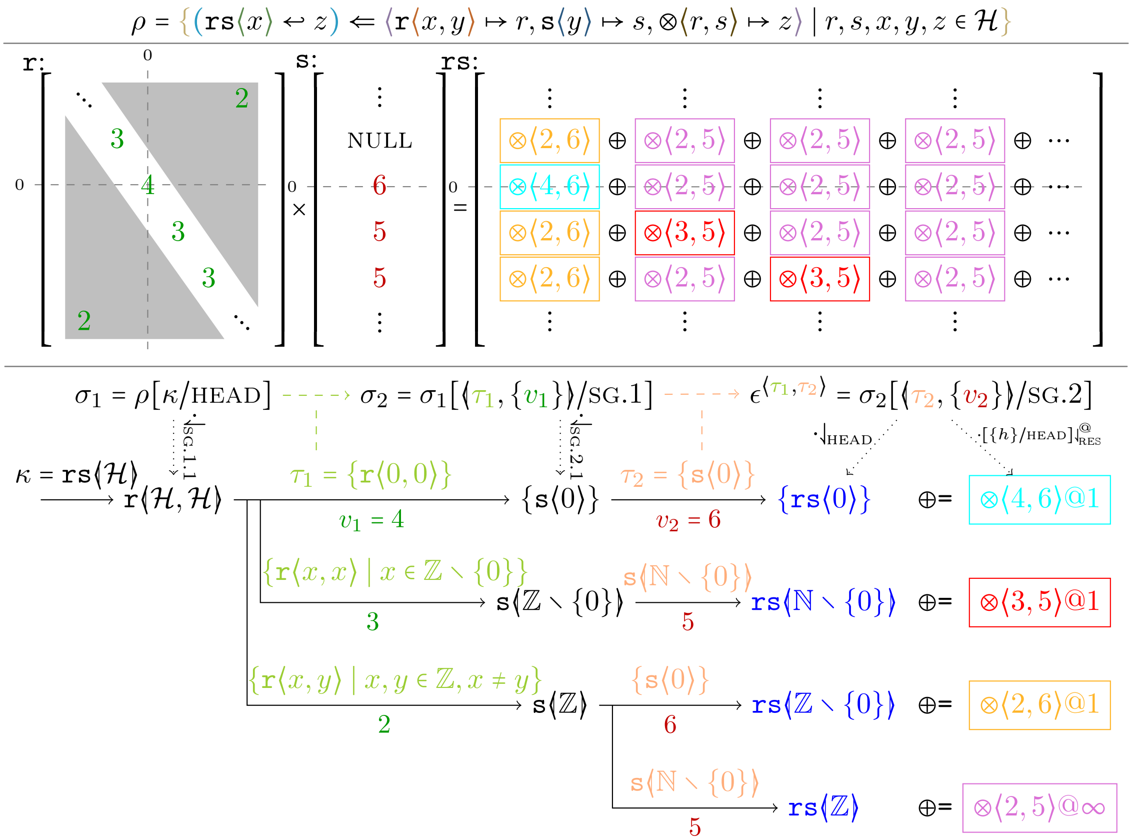
To show inclusion in the other direction, it suffices to argue that every hyperedge is accounted for. That is, every hyperedge undergoes one of three fates: 1) survives to be fed to contribRuleAnswer, 2) skipped because it ends up in a set corresponding to an empty σ, 3) skipped because it *would* correspond to an empty σ. The first case is apparent, but the others deserve some explanation. Line 4 of block 3.1 evidently short-circuits and avoids consideration of the suffix of a set of hyperedges; assuming Lookup behaves as specified, this line will only execute at the outermost refineRuleSuffix invocation, excluding all hyperedges for rule r whose heads are outside the query set κ. However, if Lookup were imagined to return *every* key/value association (ignoring the infinite time this would take), then line 4 of block 3.1 would exclude exactly the set of edges that we are presuming to be implicitly elided. (Computationally, this implicit elision is a huge advantage: the encoding is sufficiently loose that it often produces infinite collections of edges when they are not, strictly, necessary; see §6.3.) Thus, with all edges accounted for, we see that $\text{yb}(\mathcal{E}[\{r\}/1], k, f)$, stripped of its NULLs, is a (possibly improper) sub-bag of $c(k)$ as of line 6 of listing 3.1.

Compute simply combines the effects from computeRule of all rules within the program by bag union at each item, which agrees with the definition of $\text{yb}(\mathcal{E}, \cdot, \cdot)$. □

3.3 Non-ground Inference via Set Subtraction

3.3.1 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. Viewed in terms of the underlying hypergraph, two items



$\rho = \{(\mathtt{rs}\langle x\rangle \hookleftarrow z) \Leftarrow \langle \mathtt{r}\langle x,y\rangle \mapsto r, \mathtt{s}\langle y\rangle \mapsto s, \otimes\langle r,s\rangle \mapsto z\rangle \mid r,s,x,y,z \in \mathcal{H}\}$
r:
s:
rs:
0
NULL
6
5
5
×
=
$\otimes\langle 2,6\rangle$
$\otimes\langle 2,5\rangle$
$\otimes\langle 4,6\rangle$
$\otimes\langle 3,5\rangle$
$\oplus \cdots$
$\sigma_1 = \rho[\kappa/\text{HEAD}]$
$\sigma_2 = \sigma_1[\langle \tau_1, \{v_1\}\rangle/\text{SG.1}]$
$\epsilon^{\langle \tau_1,\tau_2\rangle} = \sigma_2[\langle \tau_2, \{v_2\}\rangle/\text{SG.2}]$
SG.1.1
SG.2.1
HEAD
$[\{h\}/\text{HEAD}]^{@}_{\text{RES}}$
$\kappa = \mathtt{rs}\langle \mathcal{H}\rangle$
$\mathtt{r}\langle \mathcal{H},\mathcal{H}\rangle$
$\tau_1 = \{\mathtt{r}\langle 0,0\rangle\}$
$v_1 = 4$
$\{\mathtt{s}\langle 0\rangle\}$
$\tau_2 = \{\mathtt{s}\langle 0\rangle\}$
$v_2 = 6$
$\{\mathtt{rs}\langle 0\rangle\}$
$\oplus= \otimes\langle 4,6\rangle@1$
$\{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z} \setminus \{0\}\}$
3
$\mathtt{s}\langle \mathbb{Z}\setminus\{0\}\rangle$
$\mathtt{s}\langle \mathbb{N}\setminus\{0\}\rangle$
5
$\mathtt{rs}\langle \mathbb{N}\setminus\{0\}\rangle$
$\oplus= \otimes\langle 3,5\rangle@1$
$\{\mathtt{r}\langle x,y\rangle \mid x,y \in \mathbb{Z}, x \neq y\}$
2
$\mathtt{s}\langle \mathbb{Z}\rangle$
$\{\mathtt{s}\langle 0\rangle\}$
6
$\mathtt{rs}\langle \mathbb{Z}\setminus\{0\}\rangle$
$\oplus= \otimes\langle 2,6\rangle@1$
$\mathtt{s}\langle \mathbb{N}\setminus\{0\}\rangle$
5
$\mathtt{rs}\langle \mathbb{Z}\rangle$
$\oplus= \otimes\langle 2,5\rangle@\infty$

Figure 3.3: 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 $\kappa = \mathtt{rs}\langle \mathcal{H}\rangle$. 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, ⊗, as it is only queried on singleton sets, 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 §3.3.3) to yield the answers to the original query $\mathtt{rs}\langle \mathcal{H}\rangle$, namely $\{\{\mathtt{rs}\langle 0\rangle\} \mapsto \bigoplus \lbag \otimes\langle 4,6\rangle@1, \otimes\langle 2,5\rangle@\infty \rbag, \mathtt{rs}\langle \mathbb{N}\setminus\{0\}\rangle \mapsto \bigoplus \lbag \otimes\langle 3,5\rangle@1, \otimes\langle 4,6\rangle@1, \otimes\langle 2,5\rangle@\infty \rbag, \mathtt{rs}\langle \mathbb{Z}\setminus\mathbb{N}\rangle \mapsto \bigoplus \lbag \otimes\langle 4,6\rangle@1, \otimes\langle 2,5\rangle@\infty \rbag\}$. The shape of the search tree is determined by the answers from LOOKUP, which returns disjoint slices of the **r** matrix $\{\{\mathtt{r}\langle 0,0\rangle\} \mapsto 4, \{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z}\setminus\{0\}\} \mapsto 3, \{\mathtt{r}\langle x,y\rangle \mid x,y \in \mathbb{Z}, x \neq y\} \mapsto 2\}$ and **s** vector $\{\{\mathtt{s}\langle 0\rangle\} \mapsto 6, \{\mathtt{s}\langle t\rangle \mid t \in \mathbb{N}\setminus\{0\}\} \mapsto 5\}$.

Non-ground rule query	**Non-ground rule answer**
$\langle \{\mathtt{r}\langle 0,0\rangle\}, \{\mathtt{s}\langle 0\rangle\}, \ldots\rangle$	$\{(\mathtt{rs}\langle 0\rangle \hookleftarrow \otimes\langle 4,6\rangle) \Leftarrow \langle \mathtt{r}\langle 0,0\rangle \mapsto 4, \mathtt{s}\langle 0\rangle \mapsto 6, \cdots\rangle\}$
$\langle \{\mathtt{r}\langle 0,0\rangle\}, \mathtt{s}\langle \mathbb{N}\setminus\{0\}\rangle, \ldots\rangle$	$\varnothing$
$\langle \{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z}\setminus\{0\}\}, \{\mathtt{s}\langle 0\rangle\}, \ldots\rangle$	$\varnothing$
$\langle \{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z}\setminus\{0\}\}, \mathtt{s}\langle \mathbb{N}\setminus\{0\}\rangle, \ldots\rangle$	$\{(\mathtt{rs}\langle x\rangle \hookleftarrow \otimes\langle 3,5\rangle) \Leftarrow \langle \mathtt{r}\langle x,x\rangle \mapsto 3, \mathtt{s}\langle x\rangle \mapsto 5, \cdots\rangle \mid x \in \mathbb{N}\setminus\{0\}\}$
$\langle \{\mathtt{r}\langle x,y\rangle \mid x,y \in \mathbb{Z}, x \neq y\}, \{\mathtt{s}\langle 0\rangle\}, \ldots\rangle$	$\{(\mathtt{rs}\langle x\rangle \hookleftarrow \otimes\langle 2,6\rangle) \Leftarrow \langle \mathtt{r}\langle x,0\rangle \mapsto 2, \mathtt{s}\langle 0\rangle \mapsto 6, \cdots\rangle \mid x \in \mathbb{Z}\setminus\{0\}\}$
$\langle \{\mathtt{r}\langle x,y\rangle \mid x,y \in \mathbb{Z}, x \neq y\}, \mathtt{s}\langle \mathbb{N}\setminus\{0\}\rangle, \ldots\rangle$	$\{(\mathtt{rs}\langle x\rangle \hookleftarrow \otimes\langle 2,5\rangle) \Leftarrow \langle \mathtt{r}\langle x,y\rangle \mapsto 2, \mathtt{s}\langle y\rangle \mapsto 5, \cdots\rangle \mid x \in \mathbb{Z}, y \in \mathbb{N}\setminus\{x\}\}$

Table 3.1: Non-ground rule queries and answers for figure 3.3. The two empty results come about since $\{0\} \cap (\mathbb{N}\setminus\{0\}) = \varnothing$.

behave similarly, roughly, if they are the heads of hyperedges whose tails are also (pairwise) similarly-behaving items. Items without incoming edges—the input—all behave similarly: there is no work to be done to find their values. Because hyperedges arise as instantiations of rules, in practice, this means that items likely behave similarly at least when many of their hyperedges come from the same rules. Rather than try to formalize the notion of similarly-behaving, let us leave it as an intuition and, more usefully, formalize a query procedure instead.

Example 32: The rule $\{(\mathtt{f}\langle x,y\rangle \hookleftarrow 1) \Leftarrow \langle\rangle \mid x,y \in \mathcal{H}\}$ (`f(X,Y) ⊕= 1`) defines an aggregand for each of the infinitely many terms of $\mathtt{f}\langle\mathcal{H},\mathcal{H}\rangle$, but the pattern is so simple that all these terms can be considered *at once*. ◊

Example 33: This kind of *bulk handling* extends across rules, too. The pair of rules

$$\{(\mathtt{f}\langle 1,y\rangle \hookleftarrow 3) \Leftarrow \langle\rangle \mid y \in \mathcal{H}\}, \qquad \{(\mathtt{f}\langle x,2\rangle \hookleftarrow 4) \Leftarrow \langle\rangle \mid x \in \mathcal{H}\}$$

(respectively, `f(1,Y) ⊕= 3` and `f(X,2) ⊕= 4`) will contribute $3 \oplus 4$ to the aggregated value of item $\mathtt{f}\langle 1,2\rangle$, while also contributing 3 to each item in $\{\mathtt{f}\langle 1,y\rangle \mid y \in \mathcal{H} \smallsetminus \{2\}\}$ and 4 to each item in $\{\mathtt{f}\langle x,2\rangle \mid x \in \mathcal{H} \smallsetminus \{1\}\}$. ◊

A more dramatic example is shown in figure 3.3: the product of a infinite matrix with an infinite vector, exploiting the fact that both have, in this case, simple definitions. The matrix $\mathtt{r}$ is defined by the cases $\{\mathtt{r}\langle 0,0\rangle\} \mapsto 4$, $\{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z} \smallsetminus \{0\}\} \mapsto 3$, and $\{\mathtt{r}\langle x,y\rangle \mid x,y \in \mathbb{Z}, x \neq y\} \mapsto 2$, where $\tau \mapsto v$ means that v is the value of each $t \in \tau$. Similarly, the vector $\mathtt{s}$ is defined by $\{\mathtt{s}\langle 0\rangle\} \mapsto 6$ and $\{\mathtt{s}\langle y\rangle \mid y \in \mathbb{N} \smallsetminus \{0\}\} \mapsto 5$. For simplicity, assume $\otimes$ is total. In this case, our `rs` rule should answer *non-ground rule queries* as in table 3.1. We can read out the contributions of the ground answers contained in each non-ground rule answer: ① $\mathtt{rs}\langle 0\rangle$ gets $\wr\otimes\langle 4,6\rangle@1\int$. ② Each $\mathtt{rs}\langle\mathbb{N} \smallsetminus \{0\}\rangle$ gets $\wr\otimes\langle 3,5\rangle@1\int$.[84] ③ Each $\mathtt{rs}\langle\mathbb{Z} \smallsetminus \{0\}\rangle$ gets $\wr\otimes\langle 2,6\rangle@1\int$. ④ For each $x \in \mathbb{Z}$, the item $\mathtt{rs}\langle x\rangle$ gets $\wr\otimes\langle 2,5\rangle@\infty\int$—an *infinite* bag of aggregands (one for each $y \in \mathbb{N} \smallsetminus \{x\}$).[85]

Figure 3.3 shows how these answers are computed. Recall that REFINERULESUFFIX in §3.2 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 3.3 must enumerate several *sets* of related subgoal items, such as $\{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{N} \smallsetminus \{0\}\}$. 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.

3.3.2 Non-ground Rule Queries and Answers

We now extend the definitions of §3.1 (and, in particular, those shown in figure 3.2) to be applicable for non-ground reasoning. A **non-ground rule query** for the rule r is a n_r-tuple

[84] 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!

[85] Were we to replace $\mathbb{N}$ with a finite set τ in the example, this last combination becomes a little more interesting: there would be $|\tau|$ contributions of $\otimes\langle 2,5\rangle$ to each of $\mathtt{rs}\langle\mathbb{Z} \smallsetminus \tau\rangle$, where the exclusion of x from the domain of y cannot be relevant, and $|\tau| - 1$ contributions to each $\mathtt{rs}\langle\tau\rangle$, where the exclusion is always relevant.

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: $\theta_r^{\vec{\tau}} \overset{\text{def}}{=} \rho_r[\tau_1/\text{SG}.1.1]\cdots[\tau_{n_r}/\text{SG}.n_r.1]$. In general, invoking Lookup on a set of kv-pairs $\tau \subseteq \langle\mathcal{H}, \mathcal{H}\rangle$ will yield a *function* $\tau\downharpoonleft_1 \to \mathcal{H}'$.[86] This function will, in general, have an infinite domain and will encode some complex relationship between domain element and assigned value. Lookup may return this function all at once or in a series of partial answers, which must be looped over just as before. These stream elements are termed **packets** and each packet serves to partially answer the Lookup-ed set of items (and values). (Recall that in the ground setting of §3.2, Lookup merely returned a finite key-value map with individual items as keys.)

Suppose that Lookup(τ_i) yields f_i when called, 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

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

This is essentially a set of ground rule answers (which, in principle, could be individually processed by contribRuleAnswer). It contributes to each $h \in \epsilon\downharpoonleft_{\text{HEAD}}$ the values $\epsilon[\{h\}/\text{HEAD}]\downharpoonleft_{\text{RES}}^{@}$. 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, requiring that each considered f_i be *constant*.

As mentioned earlier (§3.2.1), in backward-chaining, we are often interested only in the subsets of rules' groundings that pertain to a particular set of heads η. Therefore, we pair our concepts of rule pre-answers and answers with optional head refinements. We augment the definitions of θ and ϵ with another index, allowing concise notation for additional refinement by HEADs: $\theta_r^{\eta,\vec{\tau}} \overset{\text{def}}{=} \theta_r^{\vec{\tau}}[\eta/\text{HEAD}]$ and similarly for ϵ.

A few set-theoretic properties of these augmented θ and ϵ are worth noting: ① $\theta_r^{\alpha,\vec{\tau}} = \theta_r^{\eta,\vec{\tau}}[\alpha/\text{HEAD}]$ and $\epsilon_{r,\mathcal{L}}^{\alpha,\vec{\tau}} = \epsilon_{r,\mathcal{L}}^{\eta,\vec{\tau}}[\alpha/\text{HEAD}]$ for any pair of sets $\alpha \subseteq \eta$. Thus, θ and ϵ are $\subseteq$-monotone in their head index. (This would not be true of ϵ if we had used $\lceil\cdot\rceil$ in its definition; instead, values are determined directly by $\vec{\tau}$.) ② $\theta_r^{\cdot}$ sends $\vec{\tau} \preceq \vec{\tau}'$ to $\theta_r^{\vec{\tau}} \subseteq \theta_r^{\vec{\tau}'}$, but no similar monotonicity holds for $\epsilon_{r,\mathcal{L}}^{\cdot}$, due to the evaluation of $\mathcal{L}$ in the latter (there being no necessary relationship between $\mathcal{L}(\tau)$ and $\mathcal{L}(\tau')$, even if $\tau \subseteq \tau'$).

3.3.2.1 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 resulting rule answers ϵ (via §3.3.3 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 HR projections.

① 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 3.3: each f_i is a constant function returning some v_i, so that the result of Lookup is piecewise-constant with finitely

[86] We use $\mathcal{H}'$ because some or all items in τ_i may have value NULL.

many pieces. Despite being a special case, this is still a *strict generalization* of our ground reasoning story: therein, Lookup associated each of *finitely many items* with a value, while, herein, we allow for *finitely many, possibly infinite sets* of items to have an associated value. 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: $\epsilon^{\vec{\tau}}_{r,\vec{v}} \overset{\text{def}}{=} \theta^{\vec{\tau}}_{r}[\{v_1\}/\text{SG}.1.2]\cdots[\{v_{n_r}\}/\text{SG}.n_r]$. 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 $\Pi_{(\Sigma_{k\in\kappa}\tau_k)\subseteq\langle\mathcal{H},\mathcal{H}\rangle} \bigcup_{K\in\text{fp}(\kappa\cap\mathcal{I})} \Pi_{\alpha\in K}(\{\text{NULL}\} \cup \bigcap_{t\in\alpha}\tau_t)$, where $\text{fp}(\beta)$ is the set of all *finite partitions* of the set β: a set of sets B is a member of $\text{fp}(\beta)$ iff all of $\bigcup B = \beta$, $|B| < \infty$, and $\forall_{\beta_1,\beta_2\in B}\, \beta_1 \cap \beta_2 = \varnothing$.

② 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 (§3.3.3 below) to give a new piecewise-constant function. Recall that for ground reasoning (§3.2), we required "K-sufficiently range restricted" rules (§3.2.1) so that each rule answer would have a single item as its head: that is, with the head refined by the query $\kappa \in K$, bringing $\rho_r\!\downharpoonleft_{\text{SG}}$ 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 rules such as the previously-troublesome $\{(\mathtt{f}\langle x, y\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{g}\langle x\rangle \mapsto v\rangle \mid y \in \tau, \cdots\}$ (`f(X,Y) ⊕= g(X)`); grounding the subgoals leaves a set of heads all of which receive the same bag of aggregands.

Generalizing, we will, instead, require **non-ground *K*-sufficient range restriction** of our programs: any rule answer $\epsilon = \epsilon^{\kappa,\vec{\tau}}_{r,\vec{v}}$ that we compute (for any $\kappa \in K$) must treat all its head items $\eta = \epsilon\!\downharpoonleft_{\text{HEAD}}$ identically, contributing the same aggregands $\lbag v@m \rbag$ to each of them. (Formally, $\exists_{v\in\mathcal{H},m\in\mathbb{N}_\infty} \forall_{h\in\eta}\ \epsilon[\{h\}/\text{HEAD}]\!\downharpoonleft^{@}_{\text{RES}} = \lbag v@m \rbag$.) This will allow §3.3.3 to easily determine which sets of head items receive which sets of aggregands. Like ground K-sufficient range restriction, in general, attesting that a program is sufficiently range restricted requires static analysis §5. We will, as before, continue to suppress the "K-sufficient" by tacitly assuming $\{\kappa\}$-sufficiency, for whatever query κ is under consideration, throughout.

Non-ground range restriction, as defined here, indeed excludes, as promised, cases where the head and value covary within a single rule answer. The rule $\rho = \{(\mathtt{f}\langle x\rangle \hookleftarrow x) \Leftarrow \langle\rangle \mid x \in \tau\}$ is rejected for any τ with $|\tau| > 1$ if subjected to a query $\mathtt{f}\langle\kappa\rangle$ with $|\kappa| > 1$. That is, this rule is $\{\{\mathtt{f}\langle k\rangle\} \mid k \in \kappa\}$-sufficiently (non-ground) range restricted, but not $\{\mathtt{f}\langle\kappa\rangle\}$-sufficiently so. When $|\tau| < \infty$, we can recover by using a rule for each element of τ, and one could imagine attempting to do something clever for the case of a *finite* intersection of query and rule, e.g., splitting the result into a finite set of results obeying the requirement above; for the moment we exclude the pair of rule and query, but we speculate about an extension in §3.6.1.

③ 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 $\mathcal{I} \cap \rho_r\!\downharpoonleft_{\text{HEAD}}$ must use this aggregator. This ensures that $\text{aggr}(\eta) \overset{\text{def}}{=} \text{selt}(\{\text{aggr}(h) \mid h \in \eta\})$ is well-defined when we use it to construct a query answer in listing 3.2 below.

3.3.3 Combining Results

Recall that in §3.2, 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* $h \in \eta = \epsilon\downharpoonleft_{\text{HEAD}}$, $m = \text{selt}(\{|\epsilon[\{h\}/\text{HEAD}]| \mid h \in \eta\})$ copies of $v = \text{selt}(\epsilon\downharpoonleft_{\text{RES}})$." We further *assume the existence* of a procedure RULETOINSTR which extracts η and $\lbag v@m \rbag$ from any non-empty ϵ derived from a non-ground range-restricted rule.

Given two rule answers, ϵ_1 and ϵ_2, with corresponding HEAD projections, η_i, and contributions, $\lbag v_i@m_i \rbag$, their combined contributions should be that $(\eta_1 \setminus \eta_2)$ gets only $\lbag v_1@m_1 \rbag$, that $(\eta_2 \setminus \eta_1)$ gets only $\lbag v_2@m_2 \rbag$, and that $(\eta_1 \cap \eta_2)$ gets contributions from both, i.e., $\lbag v_1@m_1, v_2@m_2 \rbag$. (Recall example 33.) Generalizing, if we 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 $\beta = \lbag v@m \rbag$, 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,

B. 3.2

```
def DISJOIN($c$, $\eta$, $\beta$) % Add all of $\beta$ to each $h \in \eta$ across all of $c$
  return $\{(\eta \setminus \bigcup(\text{dom}(c))) \mapsto \beta \mid \eta \notin \bigcup(\text{dom}(c))\}$ % new bin for new terms;
    $\cup\{(\kappa \setminus \eta) \mapsto \tau \mid (\kappa \mapsto \tau) \in c, \kappa \notin \eta\}$          % split old bins: differences …
    $\cup\{(\kappa \cap \eta) \mapsto \beta \uplus \tau \mid (\kappa \mapsto \tau) \in c, \kappa \cap \eta \neq \varnothing\}$   % … and intersections
```

This procedure forms the core of our CONTRIBRULEANSWER procedure for non-ground backward reasoning, shown in listing 3.2.

3.3.4 Correspondence of Semantics and Procedural Results

We can reprise §3.2.3 with our modified, non-ground algorithm. In this setting, we need to slightly revise our assumptions of LOOKUP. LOOKUP$(\Sigma_\kappa \tau)$ (wherein κ is no longer required to be finite) now further encodes $f \in \kappa \to \mathcal{H}'$ by $f' \in \bigcup_{K \in \text{fp}(\kappa \cap \mathcal{I})} \Pi_{\alpha \in K}(\{\text{NULL}\} \cup \bigcap_{t \in \alpha} \tau_t)$. To ensure that f' is a faithful encoding of f, we require that $\bigcup K = \kappa$ and, $\forall_{(k \mapsto v) \in f}$, that either $v \in \tau_k \wedge f'(\alpha) = v$ or $v \notin \tau_k \wedge f'(\alpha) = \text{NULL}$. Unlike the ground setting, we are, here, requiring a kind of "totality" in that $\bigcup K = \kappa$; we use NULL where before we would simply have reduced the domain of the encoding f'.

Lemma 2 (*REFINERULESUFFIX overestimates* ϵ)**:** *Assuming* LOOKUP *behaves as above, encoding some function f, within Listing 3.2, every invocation of the function closure* GO *(as returned by* REFINERULESUFFIX(r,_)*, when called from* COMPUTERULE(r,κ)*) with arguments* (σ,i) *obeys*

$$\exists_{\vec{\phi} \in (\wp\mathcal{H})^{i-1}} \sigma \supseteq \bigcup_{\vec{p} \in \mathcal{H}^{i-1}, \forall_j\, p_j \in \phi_j} \bigcup_{\vec{t} \in \mathcal{H}^{n_r - i + 1}} \epsilon_{r,f}^{\kappa, \vec{p} \mathbin{+\!\!+} \vec{t}}.$$

The superset is equality when $i = n_r + 1$.

Proof. The prefix tuple $\vec{\phi}$ consists of the τ *sets* in the foreach-es on line 21 of listing 3.2 that transitively called the invocation of REFINERULESUFFIX in question. Generalizing the

```
def COMPUTE($\kappa \subseteq \mathcal{I}$)
  $y$ ← $\varnothing$ % initialize accumulator: no contributions to any item
  foreach $r \in \Xi$ do
    foreach $(\eta \mapsto \beta)$ ∈ COMPUTERULE($r$,$\kappa$) do % accumulate aggregands
      $y$ ← DISJOIN($y$, $\eta$, $\beta$)
  return $\{\eta \mapsto \mathsf{aggr}(\eta)(\beta) \mid (\eta \mapsto \beta) \in y\}$

def COMPUTERULE($r \in \Xi$, $\kappa \subseteq \mathcal{I}_{\mathtt{der}}$) $\in (\bigcup_{K \in \mathtt{fp}(\kappa \cap \mathcal{I})} K \to \mathcal{H}'^{+})$
  $c$ ← $\{k \mapsto \varnothing \mid k \in \kappa\}$ % accumulator for rule contribution
  REFINERULESUFFIX($r$, CONTRIBRULEANSWER)($\rho_r[\kappa/\mathrm{HEAD}]$, 1)
  return $c$

  def CONTRIBRULEANSWER($\epsilon \subseteq \rho_r$) $\in \langle\rangle$ % extract answers and accumulate (§3.3.3)
    $c$ ← DISJOIN($c$, $\eta$, $\beta$) where $\langle \eta, \beta \rangle$ = RULETOINSTR($\epsilon$)

def REFINERULESUFFIX($r \in \Xi$, CONTRIBRULEANSWER)
  return GO
  def GO($\sigma \subseteq \rho_r$, $i \in \mathbb{N}_1^{n_r+1}$)
    if $\sigma = \varnothing$ then return                    % no contributions here, or
    elif $i = n_r + 1$ then CONTRIBRULEANSWER($\sigma$) % some answers to process, or
    else foreach $(\tau \mapsto v)$ ∈ LOOKUP($\sigma\downharpoonleft_{\mathrm{SG}.i}$) do
      GO($\sigma[\langle\!\langle \tau, \{v\} \rangle\!\rangle/\mathrm{SG}.i]$, $i+1$)          % refine and move to next subgoal

LOOKUP $\in \prod_{(\sum_{k \in \kappa} \tau_k) \subseteq \langle\!\langle \mathcal{H}, \mathcal{H} \rangle\!\rangle} \bigcup_{K \in \mathtt{fp}(\kappa \cap \mathcal{I})} \prod_{\alpha \in K} (\{\mathrm{NULL}\} \cup \bigcap_{t \in \alpha} \tau_t)$ % answer a subgoal query
RULETOINSTR $\in \prod_{\epsilon \subseteq \rho_r} \langle \epsilon\downharpoonleft_{\mathrm{HEAD}}, \wp_{+}\bar{\mathfrak{U}}_{\infty}(\epsilon\downharpoonleft_{\mathrm{RES}}) \rangle$ % extract head and result bag from rule
```

Listing 3.2: 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 (recall ① of §3.3.2.1). 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, as given in block 3.2 (in §3.3.3), to maintain the invariant that all domain elements of the accumulators c and y are *disjoint*.

arguments from ground computation, and taking $v_i = \mathrm{selt}(\{f(k) \mid k \in \phi_i\})$, we have

$$\sigma = \rho_r[\kappa/\mathrm{HEAD}][\{\langle p, v_1\rangle \mid p \in \phi_1\}/\textsc{sg}.1]\cdots[\{\langle p, v_{i-1}\rangle \mid p \in \phi_{i-1}\}/\textsc{sg}.1].$$

We now appeal to a kind of distribution of refinement over a product operation, specifically,

$$\forall_{\alpha,\sigma,\tau,j}\, \alpha[\{\langle s,t\rangle \mid s \in \sigma, t \in \tau\}/\textsc{sg}.j] = \alpha[\sigma/\textsc{sg}.j.1][\tau/\textsc{sg}.j.2],$$

and the distribution of refinement over union, i.e., that $\forall_{\alpha,S,\pi}\, \alpha[\bigcup S/\pi] = \bigcup_{\sigma\in S} \alpha[\sigma/\pi]$. These two facts, and a little algebra within an induction on i shows that the $\bigcup\bigcup\epsilon$ term can be rearranged to have, at least, all of the same refinements to ρ_r that were applied to obtain σ, with the two becoming equal when $i = n_r + 1$. □

As before, our correspondence is nearly immediate as a consequence:

Theorem 2: *Assuming* Lookup *behaves as above,* computeRule*, and therefore* Compute*, from Listing 3.2 implements our semantics for programs amenable to bulk handling.*

Proof. Essentially the same proof from the ground case continues to apply here. Every invocation of contribRuleAnswer corresponds exactly to a set of hyperedges all labeled with $\mathrm{selt}(\sigma\downharpoonleft_{\mathrm{RES}})$. Dually, every hyperedge either is included in some σ on which contribRuleAnswer is invoked, included in some σ that eventually becomes empty, or is skipped by the loops because it would have empty σ (and therefore ϵ). □

Finiteness of the encoding returned by Compute is nearly immediate, too:

Lemma 3: *Assuming* Lookup *behaves as above,* computeRule*, and therefore* Compute*, from Listing 3.2 returns finite partitions of potentially infinite sets.*

Proof. Given that each rule call to Lookup produces a finite partition (of a potentially infinite set), contribRuleAnswer's use of disjoin will only ever update the c accumulator to have a finite domain. Since there are only finitely many rules, y, too, must have finite domain. To be fully formal, we would have to argue that disjoin is correct as written, but we believe inspection suffices. □

3.3.5 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* (the canonical textbook on the topic is [35]; we will discuss tree automata in more detail in §4.2). General use of equality constraints 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 over terms, such as $\{(\mathtt{g}\langle x\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{f}\langle x,y\rangle \mapsto v\rangle \mid v,x,y \in \tau\}$. If quantification within the set defining the rule (e.g., the set τ) is over only bounded-depth terms (e.g., Datalog programs, wherein all items are *flat*, i.e., $\mathcal{I} \subseteq \{\mathtt{f}\langle \mathtt{a}_1\langle\rangle,\ldots,\mathtt{a}_n\langle\rangle\rangle \mid \mathtt{f}^{/n} \in \mathcal{F}, \forall_i\, \mathtt{a}_i{}^{/0} \in \mathcal{F}\}$), 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, as they can be reduced to finite sequences of operations on *regular* tree set automata.

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 $\{(\mathtt{g}\langle x\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{f}\langle x,y\rangle \mapsto v\rangle \mid v,x,y \in \tau\}$ if τ is a *regular* tree type (see §4.2.3) rather than being a complicated type such as "lists of all-equal elements" or "lists of equal pairs," which require equality checks arbitrarily far from the root (requiring, in turn, equality constraints across recursion within an automaton representing such a type).

μDyna with Disjoint Sets While set subtraction in general is complicated, one scenario is particularly easy to compute: the subtraction of a set from any subset of its complement (yielding the empty set). This observation gives us computational traction on a subset of μDyna programs wherein all sets that arise are disjoint (unless equal) and which, therefore, trivially give rise to solutions with disjoint keys. For example, one could take a a (ground) range-restricted program (recall §3.2.1) describing a finite circuit and construct a degenerate description of infinitely many copies of that circuit by adding an otherwise unused variable to every head and subgoal: that is, by replacing every rule

$$\{\langle\langle \mathtt{h}\langle\ldots\rangle, w\rangle, \langle\langle \mathtt{g}_1\langle\ldots\rangle, v_1\rangle, \ldots, \langle\langle \mathtt{g}_n\langle\ldots\rangle, v_n\rangle\rangle\rangle\rangle \mid \cdots\}$$

with the modified rule

$$\{\langle\langle \mathtt{h}\langle\ldots, x\rangle, w\rangle, \langle\langle \mathtt{g}_1\langle\ldots, x\rangle, v_1\rangle, \ldots, \langle\langle \mathtt{g}_n\langle\ldots, x\rangle, v_n\rangle\rangle\rangle\rangle \mid x \in \mathcal{H}, \cdots\}$$

(with x a new variable not in any of the elided segments). The variable must be the same across all subgoals and the head, otherwise the semantics of the program will be altered and duplicate copies of the structure will contribute their values multiple times to individual items. Our algorithm will execute correctly against these programs, computing answers which leave x unbound throughout and otherwise faithfully emulate those of the original program. All set subtractions computed within DISJOIN (block 3.2) will be trivial, since range restriction (and, in particular, step-wise finiteness) of the original program will ensure that the heads are ground aside from the introduced free variable.

3.4 Non-ground Inference via Default Reasoning

3.4.1 Required Set Theory Operations

This section aims to eliminate the need to compute and represent *differences* of sets of trees, as part of a more general program of lowering the requirements of algorithmic representations of sets. In the previous section, we used set difference to construct the piecewise-constant valuation function (§3.3.2.1) by partitioning its domain into *non-overlapping* sets (§3.3.3). Unfortunately, this approach does not appear to be *computable* in general, in that the suitable families of tree automata for μDyna's use (i.e., those with equality constraints) of which we are aware are either not closed under set complementation or have undecidable decision predicates (e.g., emptiness). (See §4.2 for more discussion.)

We are, indeed, able to eliminate any need to represent the difference of two sets, though in general, our set-theoretic requirements will remain quite high. The design in this section encodes the piecewise-constant valuation 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 a strictly smallest domain, and it defines the corresponding value. This design requires the ability to describe sets that correspond to all instantiations of a non-ground term (which may contain repeated variables). We also require our family of representable sets to be closed under finite unions and intersections. Finally, we require the ability to count cardinalities of set differences (without representing the differences themselves).

Formally, this last ability, cardinality of set difference, implies, further, that we can perform testing of any subset relation: $\alpha \subseteq \beta$ iff $|\alpha \setminus \beta| = 0$. Thus, we are also able to test for equality of two sets ($\subseteq$ is anti-symmetric) including emptiness of a set ($\alpha = \varnothing$ if $|\alpha \setminus \varnothing| = 0$). These operations are known to be undecidable on many classes of tree automata (even many which cannot represent set differences); however, we hope that, by developing appropriate heuristics that cover common cases, we can enlarge the class of programs on which we can guarantee terminating execution.

In the previous section, we ultimately required that all non-ground rule answers ϵ were *uniform* in their contributions to their heads: each $h \in \epsilon\downharpoonleft_{\text{HEAD}}$ had to be associated with *the same* bag of aggregands. Now, we will require a similar kind of uniformity of rule answers, in particular, uniformity on a "surviving" subset. Recognition of this uniformity, with a little irony, encapsulates *two* set subtractions yet appears to be a *slightly* lower requirement. In particular, given a non-ground rule answer $\epsilon \subseteq \rho$, an "obstructed head" set $\omega \subseteq \mathcal{I}$, and a "masked" set of rule groundings $\mu \subseteq \rho$, we define $\textsc{AnswerFor}(\epsilon, \omega, \mu)$ to equal β iff $\forall_{h \in (\epsilon\downharpoonleft_{\text{HEAD}} \setminus \omega)} (\epsilon \setminus \mu)[\{h\}/\text{HEAD}]\downharpoonleft^{@}_{\text{RES}} = \beta$. We can read this as "every head in $\epsilon\downharpoonleft_{\text{HEAD}}$, excepting the obstructed ω, is, after masked groundings are removed, associated with the same bag-view RES projection, β." For simplicity, we will restrict to βs of the form $\wr v@m \int$ for some $v \in \mathcal{H}$ and $m \in \mathbb{N}_\infty$.

The treatment in this section is *abstracted* over the choice of underlying automaton family or computational set encoding (discussed further in §4). That is, we describe and solve the problem using the language of sets, without worrying about execution. Sadly, it seems likely that no single class of tree automata is suitable for all programs. Further, despite our efforts towards static analysis in §5, we are not yet to the point of understanding how to effectively describe the subset of general operations that would be required by a given program. As such, an interesting avenue of future work is to consider *partially-capable* classes (or overlapping collections of classes) of tree automata and verify that a given program's computational demands will be within the *total* subsets of the domains of the underlying automata operations.

3.4.2 Encoding Functions with Finite Ranges

In a large subset of μDyna programs, we need to find and manipulate functions $f \in \kappa \to \mathcal{H}^+$ which have possibly infinite domain κ but only *finite range*, where the range is known only at runtime. We recast the machinery of the prior section in a new light as an "encoding" of a function before exhibiting a "default"-based encoding which avoids the need for set subtraction in construction and may be interpreted with only cardinality of subtractions.

3.4.2.1 Encoding by Domain Partition

A piecewise-constant function with finite range, $f \in \kappa \to \alpha$, can be described by a finite set of pairs $\langle \tau_i, a_i \rangle$ with disjoint $\tau_i \subseteq \kappa$ and $a_i \in \alpha$. This, in turn, can be made into a function $f' = \bigcup_i \{\tau_i \mapsto a_i\}$ so that $f(t) = f'(\tau_i)$ with τ_i the *unique* element of $\operatorname{dom}(f')$ which contains t. Given two such encodings, f'_i, $(i \in \{1,2\})$ of functions with a common domain, $f_i \in \kappa \to \alpha$, and a binary operator $\oplus$ on α, it is easy to build the **$\oplus$-join** of f_1 and f_2: $f_1 \wedge^{\oplus} f_2 \in \kappa \to \alpha$; it is $\{(\tau_1 \cap \tau_2 \neq \varnothing) \mapsto f'_1(\tau_1) \oplus f'_2(\tau_2) \mid \tau_i \in \operatorname{dom}(f'_i)\}$. If, however, the f_i have different domains, $f_i \in \kappa_i \to \alpha_i$, and we wish for the join (when decoded) to have $\kappa_1 \cup \kappa_2$ as its domain, we must first specify values for the novel part of the domain for each f_i. Specifically, we must add (partitions of) $\kappa_2 \setminus \kappa_1$ to $\operatorname{dom}(f_1)$, and $\kappa_1 \setminus \kappa_2$ to $\operatorname{dom}(f_2)$, associated with identity elements of $\oplus$. At least one of these differences is non-empty, by assumption. The need to compute set differences is unavoidable.

As it happens, the non-ground reasoning algorithm of the last section maintains its collections of aggregands to head items using exactly this encoding. Recall its DISJOIN function, block 3.2 (in §3.3.3), which adds the collection of aggregands β to each head $h \in \eta$ across the encoding f' and yields an appropriately adjusted encoding. This function is precisely $f' \wedge^{\uplus} \{\eta \mapsto \beta\}$ in disguise. We reproduce it here in a slightly different, but equivalent, form, with comments to highlight the relation to $\wedge^{\uplus}$:

```
def DISJOIN($c$, $\eta$, $\beta$) % Add all of $\beta$ to each $h \in \eta$ across all of $f'$
  return $\{(\kappa \cap \eta) \mapsto \beta \uplus \tau \mid (\kappa \mapsto \tau) \in f', \kappa \cap \eta \neq \varnothing\}$      % common domain elements
             $\cup\{(\kappa \setminus \eta) \mapsto \tau \mid (\kappa \mapsto \tau) \in f', \kappa \notin \eta\}$                % enlarge novel domain
             $\cup\{(\eta \setminus \bigcup(\mathtt{dom}(f'))) \mapsto \beta \mid \eta \notin \bigcup(\mathtt{dom}(f'))\}$   % enlarge $f'$'s domain
```

3.4.2.2 Encoding by Defaults

One may alternatively encode such a piecewise-constant function with finite range, $f \in \kappa \to \alpha$, using finitely many *potentially overlapping* subsets of κ, so long as it is clear *which* subset should be used for decoding each element $k \in \kappa$. Here, we consider an encoding which ensures that there is always a unique, *smallest* subset of κ for each k. Let $\mathcal{B} \in (K \setminus \{\varnothing\}) \to \alpha$ with $K \subseteq \wp\kappa$ be such an encoding, called a **piecewise-constant backed-off function** (BF), whose structure we now detail.

Not any such function $\mathcal{B}$ on any set K will do. In order to ensure that there is a unique smallest set for every $k \in \kappa$, we require that K be ① **intersection-closed** (i.e., $\forall_{\tau_i \in K}\, \tau_1 \cap \tau_2 \in K$), and ② a **cover** of κ (i.e., $\bigcup K = \kappa$).[87] We will occasionally refer to the particular K used by a BF encoding (i.e., the domain of the *encoding*, not *encoded*, function) as its **base**. Let $\mathrm{ficc}(\kappa)$ ("finite, intersection-closed covers") be the set of all such sets K. $\cap$-closure allows us to define the **encloser** of non-empty subsets $\sigma \subseteq \kappa$ for $\mathcal{B}$: $\lceil \sigma \rceil_{\mathcal{B}}$ (or just $\lceil \sigma \rceil$, when clear) is the *smallest* $\tau \in K$ such that $\sigma \subseteq \tau$, if it uniquely exists.[88] The BF $\mathcal{B}$ encodes f by taking $f(k) = \mathcal{B}(\lceil \{k\} \rceil)$. We extend application notation to $\mathcal{B}(k) \overset{\text{def}}{=} \mathcal{B}(\lceil \{k\} \rceil)$ (since $k \in \kappa$ and not $\wp\kappa$, this is unambiguous).

The $\wedge^{\oplus}$ of two $\mathcal{B}_i \in (K_i \setminus \{\varnothing\}) \to \alpha$, with $\forall_i \bigcup K_i = \kappa$, is much as before: $\mathcal{B}_1 \wedge^{\oplus} \mathcal{B}_2 \in \{\sigma_1 \cap \sigma_2 \neq \varnothing \mid \sigma_i \in K_i\} \to \alpha$ sends τ to $\mathcal{B}_1(\lceil \tau \rceil_{\mathcal{B}_1}) \oplus \mathcal{B}_2(\lceil \tau \rceil_{\mathcal{B}_2})$.[89] Given a BF $\mathcal{B} \in S \to \alpha$ with $\bigcup S \subsetneq \kappa$, we can construct a BF $\mathcal{B}' \in S' \to \alpha$ with $\bigcup S' = \kappa$ by ensuring that $\kappa \in S'$ and $\{\kappa \cap \sigma \mid \sigma \in S\} \in S'$ and that these map to appropriate values (e.g., the identity of $\oplus$). (This is not the *only* way to construct $\mathcal{B}'$, but it is perhaps the most convenient.) No longer needing to partition, we have no need for set subtraction.

Of course, as BFs are just an *encoding* of a function, just as is the partition-based scheme above, it is possible, and hopefully illustrative, to consider converting a BF $\mathcal{B}$ to a partition encoding f'. The subset of κ strictly enclosed by $\tau \in K$ (and not by some subset of τ) is $u(\tau) = \tau \setminus \bigcup\{\sigma \in K \mid \sigma \subsetneq \tau\}$. While there may be $\sigma' \in K$ which overlap τ but are not subsets thereof, $\cap$-closure of K ensures that the overlap $\sigma' \cap \tau \in K$ and will, therefore, be removed from $u(\tau)$. Thus, $K = \{u(\tau) \mid \tau \in K\}$ is a partition of κ, and $f'(\kappa') = \mathcal{B}(\kappa')$ is constant on each $\kappa' \in K$. In the other direction, every partition-based encoding f' is already a BF. By assumption, the partition covers the domain κ. Partition elements have empty intersection, so $\mathrm{dom}(f') \cup \{\varnothing\}$ is already $\cap$-closed, and f' serves as $\mathcal{B}$.

Example 34: Consider the three rules $\{(\mathtt{r}\langle x, y\rangle \hookleftarrow 2) \Leftarrow \langle\rangle \mid x, y \in \mathbb{Z}\}$ $\{(\mathtt{r}\langle x, x\rangle \hookleftarrow 1) \Leftarrow \langle\rangle \mid x \in \mathbb{Z}\}$ and $\{(\mathtt{r}\langle 0, 0\rangle \hookleftarrow 1) \Leftarrow \langle\rangle\}$.[90] If aggregated by sum, their combined contributions form a three-way partition of $\mathtt{r}\langle \mathbb{Z}, \mathbb{Z}\rangle$: $\{\mathtt{r}\langle 0, 0\rangle \mapsto 4\} \cup \{\mathtt{r}\langle x, x\rangle \mapsto 3 \mid x \in \mathbb{Z} \setminus \{0\}\} \cup \{\mathtt{r}\langle x, y\rangle \mapsto 2 \mid x, y \in \mathbb{Z}, x \neq y\}$. This same result can be readily encoded as a piecewise-constant BF without the need for set subtraction: $\{\{\mathtt{r}\langle 0, 0\rangle\} \mapsto 4, \{\mathtt{f}\langle x, x\rangle \mid x \in \mathbb{Z}\} \mapsto 3, \mathtt{r}\langle \mathbb{Z}, \mathbb{Z}\rangle \mapsto 2\}$. ◊

Example 35: A rule in which values covary with the head, such as $\{(\mathtt{f}\langle x\rangle \hookleftarrow x) \Leftarrow \langle\rangle \mid x \in \mathcal{H}\}$ does not give rise to contributions amenable to piecewise-constant BFs. Recall ② of §3.3.2.1 and see §3.6.1 for discussion of extending backoff-functions beyond piecewise-constancy. ◊

[87] This ensures that $\mathcal{B}$ still acts as a total function on its domain. Practically, this often means that $\kappa \in K$ with some suitable default value, very often NULL.

[88] Not having required that $\kappa \in K$, we are not ensured that an arbitrary subset of κ has an encloser. Enclosers are, however, certainly defined for any subsets of any $\sigma \in K$ and for all singleton subsets of κ.

[89] The use of $\lceil \tau \rceil$ as arguments to $\mathcal{B}_i$ is perhaps surprising, but necessary: there may be multiple pairs $\langle \sigma_1, \sigma_2 \rangle$ with $\sigma_i \in K_i$ that have the same intersection $\sigma_1 \cap \sigma_2$. It is easy to see that $\bigcup\{\sigma_1 \cap \sigma_2 \neq \varnothing \mid \sigma_i \in K_i\} = \kappa$ follows from our assumptions on K_i.

[90] Prolog-style syntax does not obviously offer a convenient way to write rules where variables range over proper subsets of the term universe, such as $\{(\mathtt{r}\langle x, y\rangle \hookleftarrow 2) \Leftarrow \langle\rangle \mid x, y \in \mathbb{Z}\}$; introducing explicit annotations might allow something like `r(X :int, Y :int) += 2`.

3.4.3 Default Reasoning

3.4.3.1 Revisiting Our Motivating Example

To motivate and guide our discussion of default reasoning using BFs, we tell again, in figure 3.4 the story from figure 3.3 (in §3.3.1) of computing the product of an infinite matrix with an infinite vector via `rs(X) ⊕= r(X,Y) ⊗ s(Y)`. To find the answer, we invoke one step of backward reasoning by calling Compute(`rs`⟨ℤ⟩), which, internally, uses Lookup to obtain values for subgoals' items. Lookup will continue to return a finite map with elements $\tau \mapsto v$, where $\tau \subseteq \mathcal{I}$ is a *set* of items and $v \in \mathcal{H}'$ is the value assigned to each $t \in \tau$. The centerpiece of this section is that the τs in this map now *overlap*, forming the base of a BF that *encodes* the partition of old. Thus, the non-ground rule answers obtained (at the leaves of the tree) in answer to the non-ground rule queries (the root-leaf path through the tree) will overlap in that their *pre-answer* sets may have non-empty intersection.[91] The *answer* sets may not intersect due to ascribing different values to the same keys.

3.4.3.2 An Intuitive View of Default Reasoning

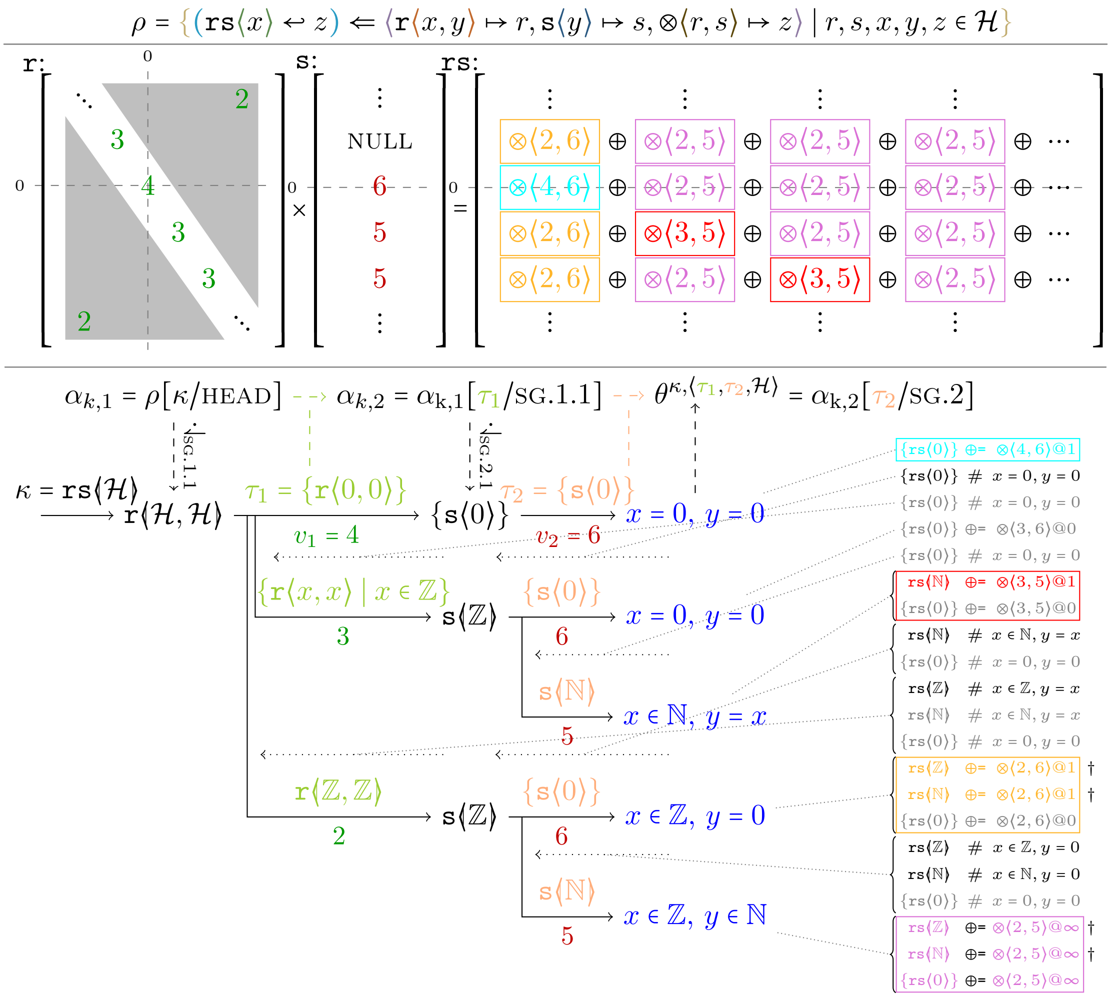
Let us attempt to give an intuition before we dive into the formalities. We will use $\tilde{\cdot}$ over a symbol to mean that we are giving a preliminary definition here and that that symbol will have a more rigorous definition given in the indicated section. As there are rather many derived quantities in flight, we proffer the map in figure 3.5 for orientation as we proceed. Quantities involved will come to have unusually many indices on them; subscripts will be global in flavor (program Ξ or rule r, answers $\mathcal{L}$) while superscripts will tend to be more local (query η, replacement $\vec{\tau}$).

Every contribution to a head h can be *named* by a pair of a rule, r, and a *ground* rule query $\vec{t}$ thereof (recall our hypergraph construction from §3.1.4). If the responses from Lookup are partitions of its argument, as they were in §3.3, then each such name will be associated with at most one value: within the search tree of figure 3.3, t_1 occurs on at most one edge from the root node, and, thereunder, t_2 again occurs on at most one edge, and so on. Whatever values are assigned to each t_i, the rule answers ϵ will be contained within the pre-answers θ: $\theta_r^{\vec{t}}$ is the *set of all possible answers for a given name* (for all possible heads, at that).

Now, however, we are assuming that the response is described by a BF, $\mathcal{L}$, encoding $\mathcal{I} \to \mathcal{H}'$, and so there may be several branches of the search tree of figure 3.4 which contain t_1, each of which may contain several branches containing t_2, etc. We must *post-process* the results of the search so that only values from *enclosing branches* are considered. As there is only one encloser for t_i, this will again associate at most one value with the query $\vec{t}$, and the definition of BFs ensure that this will be the value *as if* the result had been partitioned. How are we to do this post-processing?

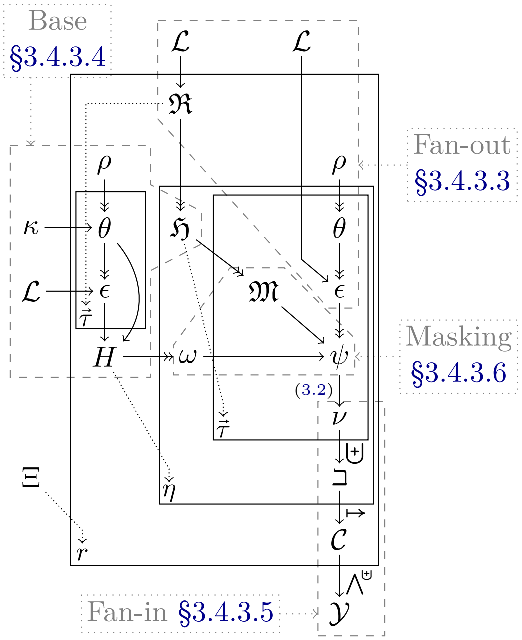
We can identify each leaf of the search tree with a non-ground rule *query* $\vec{\tau}$, e.g., $\langle \tau_1, \tau_2 \rangle$. Given a rule query $\vec{\tau}$ for r, the pre-answer set $\theta_r^{\vec{\tau}}$ is the *set of all possible answers*

[91] Overrides within a BF answering a query of a μDyna program only arise from *primitives* or the interaction of *multiple contributions* across multiple rules or from existing overrides at subgoals; conjunction within a rule cannot on its own create an override de novo, as its action on θ and ϵ sets is defined entirely by refinement, which is merely set intersection in a fancy wrapper.



$\rho = \{(\mathtt{rs}\langle x\rangle \hookleftarrow z) \Leftarrow \langle \mathtt{r}\langle x,y\rangle \mapsto r, \mathtt{s}\langle y\rangle \mapsto s, \otimes\langle r,s\rangle \mapsto z\rangle \mid r,s,x,y,z \in \mathcal{H}\}$
r:
s:
rs:
NULL
×
=
$\alpha_{k,1} = \rho[\kappa/\text{HEAD}]$
$\alpha_{k,2} = \alpha_{k,1}[\tau_1/\text{SG.1.1}]$
$\theta^{\kappa,\langle\tau_1,\tau_2,\mathcal{H}\rangle} = \alpha_{k,2}[\tau_2/\text{SG.2}]$
SG.1.1
SG.2.1
$\kappa = \mathtt{rs}\langle\mathcal{H}\rangle$
$\mathtt{r}\langle\mathcal{H},\mathcal{H}\rangle$
$\tau_1 = \{\mathtt{r}\langle 0,0\rangle\}$
$v_1 = 4$
$\{\mathtt{s}\langle 0\rangle\}$
$\tau_2 = \{\mathtt{s}\langle 0\rangle\}$
$v_2 = 6$
$x = 0,\ y = 0$
$\{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z}\}$
3
$\mathtt{s}\langle\mathbb{Z}\rangle$
$\{\mathtt{s}\langle 0\rangle\}$
6
$x = 0,\ y = 0$
$\mathtt{s}\langle\mathbb{N}\rangle$
5
$x \in \mathbb{N},\ y = x$
$\mathtt{r}\langle\mathbb{Z},\mathbb{Z}\rangle$
2
$\mathtt{s}\langle\mathbb{Z}\rangle$
$\{\mathtt{s}\langle 0\rangle\}$
6
$x \in \mathbb{Z},\ y = 0$
$\mathtt{s}\langle\mathbb{N}\rangle$
5
$x \in \mathbb{Z},\ y \in \mathbb{N}$
$\{\mathtt{rs}\langle 0\rangle\}$ ⊕= $\otimes\langle 4,6\rangle$@1
$\{\mathtt{rs}\langle 0\rangle\}$ # $x = 0, y = 0$
$\{\mathtt{rs}\langle 0\rangle\}$ # $x = 0, y = 0$
$\{\mathtt{rs}\langle 0\rangle\}$ ⊕= $\otimes\langle 3,6\rangle$@0
$\{\mathtt{rs}\langle 0\rangle\}$ # $x = 0, y = 0$
$\mathtt{rs}\langle\mathbb{N}\rangle$ ⊕= $\otimes\langle 3,5\rangle$@1
$\{\mathtt{rs}\langle 0\rangle\}$ ⊕= $\otimes\langle 3,5\rangle$@0
$\mathtt{rs}\langle\mathbb{N}\rangle$ # $x \in \mathbb{N}, y = x$
$\{\mathtt{rs}\langle 0\rangle\}$ # $x = 0, y = 0$
$\mathtt{rs}\langle\mathbb{Z}\rangle$ # $x \in \mathbb{Z}, y = x$
$\mathtt{rs}\langle\mathbb{N}\rangle$ # $x \in \mathbb{N}, y = x$
$\{\mathtt{rs}\langle 0\rangle\}$ # $x = 0, y = 0$
$\mathtt{rs}\langle\mathbb{Z}\rangle$ ⊕= $\otimes\langle 2,6\rangle$@1 †
$\mathtt{rs}\langle\mathbb{N}\rangle$ ⊕= $\otimes\langle 2,6\rangle$@1 †
$\{\mathtt{rs}\langle 0\rangle\}$ ⊕= $\otimes\langle 2,6\rangle$@0
$\mathtt{rs}\langle\mathbb{Z}\rangle$ # $x \in \mathbb{Z}, y = 0$
$\mathtt{rs}\langle\mathbb{N}\rangle$ # $x \in \mathbb{N}, y = 0$
$\{\mathtt{rs}\langle 0\rangle\}$ # $x = 0, y = 0$
$\mathtt{rs}\langle\mathbb{Z}\rangle$ ⊕= $\otimes\langle 2,5\rangle$@∞ †
$\mathtt{rs}\langle\mathbb{N}\rangle$ ⊕= $\otimes\langle 2,5\rangle$@∞ †
$\{\mathtt{rs}\langle 0\rangle\}$ ⊕= $\otimes\langle 2,5\rangle$@∞

Figure 3.4: A retelling of figure 3.3, now with default reasoning; see §3.4.4.1 for prose. We are computing the product of an infinite matrix with an infinite vector. To obtain the answer, we call COMPUTE with query $\kappa = \mathtt{rs}\langle\mathcal{H}\rangle$. 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 $\mathtt{r}$, and the edges from the root correspond to the *overlapping* branches of answers returned by LOOKUP. Each such edge leads to a new node with some refined query of the second subgoal $\mathtt{s}$, and the edges from that node again correspond to answers. Thus, each leaf corresponds to a non-ground rule query $\langle\tau_1,\tau_2,\mathcal{H}\rangle$, which can be read off from the root-leaf path. The attendant (pre-)answer sets are central to further reasoning; for reasons of space we show only θ at each leaf and describe it by the variables used in the comprehension definition of ρ at the top. Each rule answer *may* contribute values to subsets of their heads, subject to *masking* (§3.4.3.6). To the right of the search tree is a trace (to be read top-to-bottom, and discussed in §3.4.4.1) of the execution of the algorithm of §3.4.4, which visits overrides before visiting defaults (i.e., sets are visited before their proper supersets). This order explains the order of branches in the tree and is why, for example, the top-most answer deriving from the pre-answer corresponding to $x = 0, y = 0$ contributes to $\{\mathtt{rs}\langle 0\rangle\}$ but the second does not. Dotted lines connect regions of the trace with their causative object in the search tree. Value contributions from leaves are denoted with ⊕= in this log; masks, derived from θ upon returns up the search tree (dotted arrows), are denoted # and continue to use the variable notation from the leaves. We elide the last set of masking operations. Gray entries do not change the algorithm's state; entries marked with daggers (†) rely on *obstruction*. The shape of the search tree is determined by LOOKUP's returns, namely $\{\{\mathtt{r}\langle 0,0\rangle\} \mapsto 4, \{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z}\} \mapsto 3, \mathtt{r}\langle\mathbb{Z},\mathbb{Z}\rangle \mapsto 2\}$ and $\{\{\mathtt{s}\langle 0\rangle\} \mapsto 6, \mathtt{s}\langle\mathbb{N}\rangle \mapsto 5\}$, BFs encoding the $\mathtt{r}$ and $\mathtt{s}$ items that have non-NULL weights.



	Mnemonic	**Type**
Ξ	Program rule indices	Set
$\mathcal{L}$	*L*ookup Results	BF
κ	Query *K*ey Set	$\subseteq \mathcal{I}$
r	*R*ule index	$\in \Xi$
η	*H*ead under consideration	$\in H$
$\vec{\tau}$	Rule query	$\in \mathfrak{R}$
$\mathfrak{R}$	*R*eplacements	Set of tuples of subsets of $\mathcal{I}$
$\mathfrak{H}$	Replacements for *H*ead	
$\mathfrak{M}$	*M*asking replacements	
H	Induced *H*eads	$\subseteq \mathcal{I}$
ω	*O*bstructing heads	
ρ	*R*ule groundings	μDyna
θ	Pre-answers, *th*ese keys	
ϵ	Answers, *E*valuated	
ψ	*S*urviving answers	
ν	Keys and *V*alues	Bag of kv-pairs
$\sqsupset$	*B*ag of result kv-pairs	
$\mathcal{C}$	*C*ontribution per rule	BF
$\mathcal{Y}$	*Y*ield	

Figure 3.5: Plate notation and mnemonics for the various characters involved in reasoning with defaults. Recall figure 3.2 for the basic semantics of plate notation. To these semantics we now add a notion of fan-in, using the mechanism indicated, of solid arrows *exiting* a plate. To reduce visual clutter, indices have been omitted (they can be uniquely recovered from context) and $\mathcal{L}$ and ρ have been repeated in the diagram, representing the same value at each occurrence. The $\vec{\tau}$ plate on the left derives the "induced heads" (see §3.4.3.4), which form the base of the contribution to the answer ($\mathcal{C}$) for each rule ($r \in \Xi$) given the query head (κ) and input BF ($\mathcal{L}$). The η plate derives the value for each point therein. Our assumptions on the program equation (3.2) (in §3.4.3.6) are seen to be the bridge between the system's fan-in (§3.4.3.5) and its masking post-processing (§3.4.3.6) of fan-out (§3.4.3.3).

for all ground rule queries $\{\vec{t} \mid t_i \in \tau_i\}$. Thus, given *two* non-ground rule queries $\vec{\sigma}$ and $\vec{\tau}$, in which $\forall_i\, \sigma_i \subseteq \tau_i$, the set of possible answers *unique to* $\vec{\tau}$ is $\theta_r^{\vec{\tau}} \setminus \theta_r^{\vec{\sigma}}$, which is a superset of whatever *actual* rule answers are licensed by $\vec{\tau}$. If we collect *all* such $\vec{\sigma}$ from the search tree for each $\vec{\tau}$ into $\tilde{\mathfrak{M}}^{\vec{\tau}}$ (§3.4.3.6), then we see that $\theta_r^{\vec{\tau}} \setminus \bigcup_{\vec{\sigma} \in \tilde{\mathfrak{M}}^{\vec{\tau}}} \theta_r^{\vec{\sigma}}$ is the set of possible answers at the leaf identified with $\vec{\tau}$, and, given a value v_i for each τ_i, $\tilde{\psi}^{\vec{\tau}} = \epsilon_{r,\vec{v}}^{\vec{\tau}} \setminus \bigcup_{\vec{\sigma} \in \tilde{\mathfrak{M}}^{\vec{\tau}}} \theta_r^{\vec{\sigma}}$ (§3.4.3.6) is the set of *un-masked answers* from this leaf, just as if each τ_i had been a partition element. $\tilde{\nu}^{\vec{\tau}} = \tilde{\psi}^{\vec{\tau}}\downharpoonleft^{@}_{\mathrm{HR}}$ (§3.4.3.5) is then a bag of pairs of heads and associated aggregands from this leaf, which should be split by head and then aggregated, along with $\tilde{\nu}$ from other leaves.

Unfortunately, this $\tilde{\nu}^{\vec{\tau}}$ is difficult to compute and manipulate. Its computation clearly requires set subtraction. More worrying, even, it may overlap with other leaves' $\tilde{\nu}^{\vec{\tau}'}$ in complex ways reminiscent of why we needed to DISJOIN, but with yet more difficulty: there is no reason to believe that $\tilde{\nu}$ can be described as a bag product between a set of heads η and a bag of aggregands β. We require some kind of *uniformity* of $\tilde{\nu}$, and so must require something of its progenitor, $\tilde{\psi}$. Let us put this question on hold.

If our goal is to not just *interpret* BF encodings of item answers but to *produce* a BF, as well, as the return for an item query, there is the question of *what base* (i.e., what sets of heads) $\tilde{H}$ the output will have (§3.4.3.4). Assuming the kind of uniformity suggested above, we might speculate that $\epsilon\downharpoonleft_{\mathrm{HEAD}}$ for each choice of rule query $\vec{\tau}$ as likely candidates. It will turn out that we need $\theta\downharpoonleft_{\mathrm{HEAD}}$ as well in some cases, and, of course, we must ensure that the output base is $\cap$-closed. Producing a BF also allows us to revisit the uniformity requirement on $\tilde{\nu}$: *it is OK to be wrong* on a head h when generating the answer for base point $\eta \in H$, *so long as* η is not $\lceil\{h\}\rceil$. That is, so long as a smaller head set gets it right, our error will go un-noticed! Thus, we require uniformity of ψ only at $\eta \setminus \{\eta' \in H \mid \eta' \subsetneq \eta\}$.

We now expand our approximations before giving an algorithm.

3.4.3.3 Replacements

For any rule r, non-ground rule queries $\vec{\tau}$, pre-answers $\theta^{\vec{\tau}}$, and answers $\epsilon^{\vec{\tau}}$ are defined (§3.3.2) for *any* choice of sets of terms $\vec{\tau} = \langle \tau_1, \ldots, \tau_{n_r} \rangle$. However, given that LOOKUP acts as a BF, $\mathcal{L}$, rule queries formed from $\mathrm{dom}(\mathcal{L})$ have special significance, describing root-leaf paths in the search tree. We call these ***$\mathcal{L}$*-replacements** (or just **replacements**, when clear): $\mathfrak{R}_{r,\mathcal{L}} \overset{\mathrm{def}}{=} \{\vec{\tau} = \langle \lceil \tau_i \rceil \rangle_{i \in \mathbb{N}_1^{n_r}} \mid \sigma_i \in \mathrm{dom}(\mathcal{L}), \tau_i = \sigma_i \cap \rho_r\downharpoonleft_{\mathrm{SG}.i.1}, \theta_r^{\vec{\tau}} \neq \varnothing\}$. In this definition, we have, without semantic consequence, restricted to non-trivial rule queries (by requiring that $\theta \neq \varnothing$) formed from sets which are not completely overridden within $\mathcal{L}$ (by using $\lceil\cdot\rceil$ in the definition of elements of the set). Our idea to order $\vec{\tau}$ by $\subseteq$ on components is formalized as a partial order termed **specificity**: $\vec{\tau} \leq \vec{\tau}' \Leftrightarrow \forall_{i \in \mathbb{N}_1^{n_r}}\, \tau_i \subseteq \tau_i'$ ($\vec{\tau}$ is *more specific* than $\vec{\tau}'$) and $\vec{\tau} < \vec{\tau}' \Leftrightarrow \vec{\tau} \leq \vec{\tau}' \wedge \exists_i\, \tau_i \neq \tau_i'$ ($\vec{\tau}$ is *strictly more specific* than $\vec{\tau}'$).

The BF $\mathcal{L}$, encoding a valuation function, specifies more than just its base. There is, thus, a natural choice not only for the non-ground rule *queries* but also for the values $\vec{v}$ determining a set of non-ground rule answers: the values assigned by $\mathcal{L}$ to each τ_i, and in particular $v_i = \mathcal{L}(\lceil \tau_i \rceil_{\mathcal{L}})$. Thus, we define the non-ground rule answer set for $\vec{\tau} \in \mathfrak{R}_{r,\mathcal{L}}$: $\epsilon_{r,\mathcal{L}}^{\vec{\tau}} \overset{\mathrm{def}}{=} \epsilon_{r,\vec{v}}^{\vec{\tau}} = \theta_r^{\vec{\tau}}[v_1/\mathrm{SG}.1.2]\cdots[v_{n_r}/\mathrm{SG}.n_r.2]$. Recall that elements of these ϵ *are no longer necessarily all reflective of reality*, in that there are some kv-pairs in ϵ that would appear to suggest that some head item had some aggregand, but this contribution is overridden by some other replacement. More formally, that is, overriding (more-specific) replacements cause elements of ϵ to not accurately reflect the assignment of values to items as done by the *interpretation* of $\mathcal{L}$.

3.4.3.4 Induced Heads

For a given r, valuation (encoding as a BF) $\mathcal{L}$, and query head κ, a $\mathcal{L}$-replacement $\vec{\tau} \in \mathfrak{R}_{r,\mathcal{L}}$ gives rise to two **induced non-ground heads**, entries in the base of the BF encoding the results of $\textsc{Compute}$-ing κ on the rule r. The head projection of both the (pre-)answer sets are potentially meaningful quantities, representing sets of items which behave similarly under the rule r. We define $H_{r,\kappa,\mathcal{L}}$ as the $\cap$-closure of the head projections of both pre-answers and answers for all replacements $\mathfrak{R}_{r,\mathcal{L}}$, i.e., of $\{\theta_r^{\kappa,\vec{\tau}}\downharpoonleft_{\text{HEAD}}, \epsilon_{r,\mathcal{L}}^{\kappa,\vec{\tau}}\downharpoonleft_{\text{HEAD}} \mid \vec{\tau} \in \mathfrak{R}_{r,\mathcal{L}}\}$. Unpacking the definitions, we see that all induced heads are subsets of the query (i.e., $\forall_{\eta \in H_{r,\kappa,\mathcal{L}}}\, \eta \subseteq \kappa$), as expected. We ignore any $\varnothing$ in $H_{r,\kappa,\mathcal{L}}$ as $\rho_r[\varnothing/\text{HEAD}] = \varnothing$.

A key insight is that the $\mathcal{L}$-replacements that might potentially influence an *entire* head α, as opposed to merely a subset thereof, are those $\mathcal{L}$-replacements whose *pre*-answers do not refine the head to a proper subset of α: $\mathfrak{H}_{r,\mathcal{L}}^{\alpha} \overset{\text{def}}{=} \{\vec{\tau} \in \mathfrak{R}_{r,\mathcal{L}} \mid \alpha = \theta_r^{\alpha,\vec{\tau}}\downharpoonleft_{\text{HEAD}}\}$. That said, because pre-answers do not take values into consideration, it is possible that the *actual* impact of $\vec{\tau}$ is to a subset of α; the contribution-collection machinery of §3.4.3.6 will solve this problem.

Example 36: Continuing our running example (§3.4.3.1) and considering $\kappa = \texttt{rs}\langle\mathcal{H}\rangle$, it is easy to check that all induced heads each come from both a θ and an ϵ, as refinement of subgoal values does not alter HEAD projections. All told, $H_{r,\kappa,\mathcal{L}} = \{\{\texttt{rs}\langle 0\rangle\}, \texttt{rs}\langle\mathbb{N}\rangle, \texttt{rs}\langle\mathbb{Z}\rangle\}$, as can be verified by inspecting the trace shown in figure 3.4. ◊

Example 37: It may seem that θ-derived heads (i.e., those entries in H which are the HEAD projection of some θ) serve no purpose; after all, only subsets of ϵ are plausibly related to the values defined by the semantics of the language. However, consider a rule which constrains the *value* of a subgoal whose key covaries with the head, such as $\{(\texttt{a}\langle x\rangle \hookleftarrow 1) \Leftarrow \langle\texttt{b}\langle x\rangle \mapsto 4\rangle \mid x \in \mathcal{H}\}$, with the overlapping answers $\mathcal{L} = \{\texttt{b}\langle\mathcal{H}\rangle \mapsto 4, \{\texttt{b}\langle 3\rangle\} \mapsto 2\}$. The θ-derived heads are $\{\texttt{a}\langle\mathcal{H}\rangle, \{\texttt{a}\langle 3\rangle\}\}$, while the ϵ-derived heads are $\{\texttt{a}\langle\mathcal{H}\rangle, \varnothing\}$. Thus, were we to solely consider ϵ-derived heads, we would fail to assert that $\{\texttt{a}\langle 3\rangle\} \mapsto \text{NULL}$. Put another way: a replacement may have answers while a more-specific replacement may not. ◊

Example 38: (Recall example 27, in §3.1.3.) The rule $\{(\texttt{f}\langle a\rangle \hookleftarrow 1) \Leftarrow \langle\texttt{a}\langle\rangle \mapsto a\rangle \mid a \in \mathcal{H}\}$ has covariance between the head and a subgoal *value*, contributing the value 1 to an item determined by *the value of* $\texttt{a}\langle\rangle$. While the previous example showed that θ-derived heads are essential, this example shows that ϵ-derived heads are as well, as the sole θ-derived head is $\texttt{f}\langle\mathcal{H}\rangle$. ◊

3.4.3.5 Collecting Contributions

For the moment, let us *assume the existence* of bags of kv-pairs, $\nu_{r,\mathcal{L}}^{\eta,\vec{\tau}} \mathrel{\underline{\uplus}} \epsilon_{r,\mathcal{L}}^{\eta,\vec{\tau}}\downharpoonleft_{\text{HR}}^{@}$, which capture the contribution of each $\vec{\tau} \in \mathfrak{H}_{r,\mathcal{L}}^{\eta}$ to the induced non-ground head $\eta \in H$ and which omit any contributions from $\vec{\tau}$ overridden by some more-specific $\vec{\tau}' \in \mathfrak{H}$. We assume no relationship between the various ν, nor, for the moment, do we assume that $\nu^{\eta,\vec{\tau}}_{r,\mathcal{L}}\downharpoonleft_1$ is η.[92] So armed, it is easy to define the bag of contributions to η across all relevant replacements $\vec{\tau}$:

$$\sqsupset^{\eta}_{r,\mathcal{L}} \overset{\text{def}}{=} \biguplus_{\vec{\tau}\in\mathfrak{H}^{\eta}_{r,\mathcal{L}}} \nu^{\eta,\vec{\tau}}_{r,\mathcal{L}}.$$

These, then, can be combined in a BF with base $H_{r,\kappa,\mathcal{L}}$: $\mathcal{C}_{r,\kappa,\mathcal{L}} \overset{\text{def}}{=} \{\eta \mapsto \{h \mapsto \Lbag v@m \mid \langle h,v\rangle \mathrel{\underline{\in}}_{=m} \sqsupset^{\eta}_{r,\mathcal{L}} \Rbag \mid h \in \eta\} \mid \eta \in H_{r,\kappa,\mathcal{L}}\}$. The core of this definition simply partitions the bag $\sqsupset^{\eta}_{r,\mathcal{L}}$ by key and collects together all of the values within a bag for each key in η. This operation is done for each induced head $\eta \in H$; by construction, the induced heads cover all possible results from the current rule. The last step, then, is to repeat this exercise for all rules $r \in \Xi$ and merge the results, which is easily enough done using the $\uplus$-join of all such functions: $\mathcal{Y}_{\Xi,\kappa,\mathcal{L}} \overset{\text{def}}{=} \bigwedge^{\uplus}_{r\in\Xi} \mathcal{C}_{r,\kappa,\mathcal{L}}$. (As $\uplus$ is associative-commutative, there is no concern of ordering here.) To ensure that the domain of $\mathcal{Y}$ is $\mathcal{I}$, we may further $\uplus$-join with $\{\mathcal{I} \mapsto \varnothing\}$ as well.

Unfortunately, while theoretically straightforward, this collection mechanism requires iteration over H and each η therein, since the νs are arbitrary. If we are to have hope of Compute-ing in finite time, we must impose structure on ν.

3.4.3.6 Masking and Obstruction

We now focus in on the middle of the system, on the task of deriving ν from ϵ. Recall that $\tilde{\psi}^{\vec{\tau}}$ was the set of rule answers from $\vec{\tau}$, having removed all pre-answers for all $\vec{\sigma} \prec \vec{\tau}$. We could formalize this by defining $\tilde{\mathfrak{M}}^{\vec{\tau}}_{r,\mathcal{L}} = \{\vec{\sigma} \in \mathfrak{R}_{r,\mathcal{L}} \mid \vec{\sigma} \prec \vec{\tau}\}$ and taking $\tilde{\psi}^{\eta,\vec{\tau}}_{r,\mathcal{L}} = \epsilon^{\eta,\vec{\tau}}_{r,\mathcal{L}} \smallsetminus \bigcup_{\vec{\sigma}\in\tilde{\mathfrak{M}}^{\vec{\tau}}_{r,\mathcal{L}}} \theta^{\eta,\vec{\sigma}}_{r}$. Then $\tilde{\nu}^{\eta,\vec{\tau}}_{r,\mathcal{L}} = \tilde{\psi}^{\eta,\vec{\tau}}_{r,\mathcal{L}}\downharpoonleft^{@}_{\text{HR}}$ is a bag of kv-pairs which are produced by $\vec{\tau}$ and survive masking, which should be aggregated with the results of other replacements and other rules. All told, given Ξ and $\mathcal{L}$, the value assigned to $h \in \mathcal{I}$ by the above, item-at-a-time mechanism is $\text{aggr}(h)(\biguplus_{r\in\Xi} \biguplus_{\vec{\tau}\in\mathfrak{R}_{r,\mathcal{L}}} (\tilde{\psi}^{\eta,\vec{\tau}}_{r,\mathcal{L}}\downharpoonleft^{@}_{\text{RES}}))$.

Returning to set-at-a-time reasoning, and, in particular, to generating a BF, we can find two refinements of the system so far described; recall that, for a given output base point η, it is okay to be wrong when describing $h \in \eta$ so long as $\eta \neq \lceil\{h\}\rceil$. Specifically, this means two things. ① There is no need to consider masking $\epsilon^{\vec{\tau}}$ by $\theta^{\vec{\sigma}}$, even if $\vec{\sigma} \prec \vec{\tau}$, if $\theta^{\vec{\sigma}}\downharpoonleft_{\text{HEAD}} \subsetneq \epsilon^{\vec{\tau}}\downharpoonleft_{\text{HEAD}}$. ② We can ignore uniformity requirements of ν on heads $\omega^{\eta}_{r,\mathcal{L}} \overset{\text{def}}{=} \bigcup\{\eta' \in H_{r,\kappa,\mathcal{L}} \mid \eta' \subsetneq \eta\}$. The first point means, concretely, that there is no need to consider something quite so large as $\tilde{\mathfrak{M}}$ above; when computing the base point η, we need only consider $\mathfrak{M}^{\alpha,\vec{\tau}}_{r,\mathcal{L}} \overset{\text{def}}{=} \{\vec{\sigma} \in \mathfrak{H}^{\alpha}_{r,\mathcal{L}} \mid \vec{\sigma} \prec \vec{\tau}\} = \{\vec{\sigma} \in \mathfrak{R}_{r,\mathcal{L}} \mid \vec{\sigma} \prec \vec{\tau}, \alpha = \theta^{\alpha,\vec{\sigma}}_{r}\downharpoonleft_{\text{HEAD}}\}$, i.e., the set of more-specific $\vec{\tau}$ *which do not shrink the pre-answer head.*

Combining both of these observations, we can define (r, $\mathcal{L}$ and η are used three times in this definition, $\vec{\tau}$ twice): $\psi^{\eta,\vec{\tau}}_{r,\mathcal{L}} \overset{\text{def}}{=} (\epsilon^{\eta,\vec{\tau}}_{r,\mathcal{L}} \smallsetminus \bigcup_{\vec{\sigma}\in\mathfrak{M}^{\eta,\vec{\tau}}_{r,\mathcal{L}}} \theta^{\vec{\sigma}}_{r})[(\mathcal{I} \smallsetminus \omega^{\eta}_{r,\mathcal{L}})/\text{HEAD}]$. It is at this point that we must appeal to our oracular test of uniformity (§3.4.1); we require that every

[92] Despite the similarity of notation, unlike θ and ϵ, ν *is not* necessarily monotonic with respect to $\subseteq$ in its η parameter, not least because the requirement that $\vec{\tau} \in \mathfrak{H}^{\eta}_{r,\mathcal{L}}$ implies that varying η may render ν undefined for a fixed choice of $\vec{\tau}$. However, even if restricting η to a subset (so that $\mathfrak{H}$ becomes a superset), the values associated with keys in ν may change.

computed ψ be such that every head $h \in \psi\downharpoonleft_{\text{HEAD}}$ be associated with the same bag of values $\psi[\{h\}/\text{HEAD}]\downharpoonleft^{@}_{\text{RES}}$. If this holds, then, at long last, we have a viable ν:

$$\nu^{\eta,\vec{\tau}}_{r,\mathcal{L}} \overset{\text{def}}{=} \Lbag \langle h, v\rangle @m \mid h \in \eta, v \sqsubseteq_{=m} \textsc{AnswerFor}(\epsilon^{\eta,\vec{\tau}}_{r,\mathcal{L}},\ \omega^{\eta}_{r,\mathcal{L}},\ \bigcup_{\vec{\sigma}\in\mathfrak{M}^{\eta,\vec{\tau}}_{r,\mathcal{L}}} \theta^{\vec{\sigma}}_{r})\Rbag. \tag{3.2}$$

There are a few things to note about this definition. ① It can happen that $\eta' = \epsilon^{\eta,\vec{\tau}}_{r,\mathcal{L}}\downharpoonleft_{\text{HEAD}} \subsetneq \eta$. When this happens, it implies that $\eta' \subseteq \omega^{\eta}_{r,\mathcal{L}}$, so $\psi^{\eta,\vec{\tau}}_{r,\mathcal{L}} = \varnothing$ and thus $\nu^{\eta,\vec{\tau}}_{r,\mathcal{L}} = \varnothing$. We can see this clearly in example 38 (in §3.4.3.4); for the purpose of the example, assume $\mathcal{L}(\{\mathtt{a}\langle\rangle\}) \mapsto 7$, so that $\epsilon^{\mathtt{f}\langle\mathcal{H}\rangle,\langle\{\mathtt{a}\langle\rangle\}\rangle}_{r,\mathcal{L}}\downharpoonleft_{\text{HEAD}} = \epsilon^{\{\mathtt{f}\langle 7\rangle\},\langle\{\mathtt{a}\langle\rangle\}\rangle}_{r,\mathcal{L}}\downharpoonleft_{\text{HEAD}} = \{\mathtt{f}\langle 7\rangle\}$. Thus, because the head covaries with a subgoal value, the default head $\mathtt{f}\langle\mathcal{H}\rangle$ has associated aggregands $\varnothing$. ② This is a constraint only on the (masked, un-obstructed) rule answers. Other than their role in masking, groundings inconsistent with $\mathcal{L}$ need not be considered. The set of inconsistent groundings may even be outside the collection of sets possessing description within an implementation (though the masks, derived from θ and therefore inclusive of both consistent and inconsistent groundings, must still be within the system). ③ Given ground inputs and a range-restricted program, this algorithm behaves essentially as any other ground solver: ω and $\mathfrak{M}$ will always both be $\varnothing$ and $|\epsilon\downharpoonleft^{@}_{\text{HR}}| = 1$, so $\textsc{AnswerFor}$ is trivial. The need for $\textsc{AnswerFor}$ to be defined thus replaces prior algorithms' (including [63]) appeal to range restriction [22].

A Closure of Computation At long last, we can get computational traction:

Lemma 4: *If ν is derived according to equation (3.2), $\mathcal{Y}_{\Xi,\kappa,\mathcal{L}}$ is piecewise-constant.*

Proof. The definition in equation (3.2) gives $\nu^{\eta,\vec{\tau}}_{r,\mathcal{L}}$ which are *bag products* between $\bar{\mathfrak{U}}_1\eta$ and values. That is, for any choice of r, $\mathcal{L}$, η, and $\vec{\tau}$ for which ν is defined, $\{h \mapsto \Lbag v@m \mid \langle h, v\rangle \sqsubseteq_{=m} \nu^{\eta,\vec{\tau}}_{r,\mathcal{L}}\Rbag \mid h \in \eta\}$ is a constant function. As $\sqsupset^{\eta}_{r,\mathcal{L}}$ is a union (over $\vec{\tau}$) of ν bags, it follows that each $\{h \mapsto \Lbag v@m \mid \langle h, v\rangle \sqsubseteq_{=m} \sqsupset^{\eta}_{r,\mathcal{L}}\Rbag \mid h \in \eta\}$ must, also, be a constant function. The definition of $\mathcal{C}_{r,\kappa,\mathcal{L}}$, which maps each η to a function of the above form, then implies that it is piecewise-constant. $\mathcal{Y}_{\Xi,\kappa,\mathcal{L}}$ is just a join of piecewise-constant BFs, and so must itself be piecewise-constant. □

Thus, to obtain a first practical algorithm, we restrict to piecewise-constant $\mathcal{L}$ and assume the preconditions of equation (3.2). The above lemma means, having started with piecewise-constant answers $\mathcal{L}$, that we will *always obtain piecewise-constant answers*, which we can merge into $\mathcal{L}$ as part of our solver's fixed-pointing execution, and not violate our precondition. While piecewise-constancy of $\mathcal{L}$ does not imply the preconditions of equation (3.2), it may, speculatively, nevertheless simplify the proof obligation to be met by static analyses.

3.4.4 Pseudocode

At last, we come to a procedural description of the system given above; pseudocode is shown in listing 3.3. $\textsc{Compute}$ simply wraps $\textsc{computeRule}$ for each rule, using $\uplus$-join and a notion of

```
def COMPUTE($\kappa \subseteq \mathcal{I}$) $\in (\bigcup_{K \in \mathtt{ficc}(\kappa)} K \to \mathcal{H}')$
  $\mathcal{Y}$ ← $\{\kappa \mapsto \varnothing\}$ % running union across all rules
  foreach $r \in \Xi$ do let $\mathcal{C}$ = computeRule($r$,$\kappa$) in $\mathcal{Y}$ ← $\mathcal{Y} \wedge^{\uplus} \mathcal{C}$
  return $\{\tau \mapsto \mathsf{aggr}(\tau)(\beta) \mid (\tau \mapsto \beta) \in \mathcal{Y}\}$ % bulk aggregate

def computeRule($r \in \Xi$, $\kappa \subseteq \mathcal{I}$) $\in (\bigcup_{K \in \mathtt{ficc}(\rho_r\downharpoonleft_{\mathrm{HEAD}} \cap \kappa)} K \to \mathcal{H}^+)$
  $\mathcal{C}$ ← $\{\kappa \mapsto \langle\varnothing,\varnothing\rangle\}$ % initialize aggregands and masks
  let $\alpha = \rho_r[\kappa/\mathrm{HEAD}]$ in refineRuleSuffix($r$,applyM,applyV)($\alpha$, $\alpha$, 1) % fill $\mathcal{C}$
  return $\{\eta \mapsto x\downharpoonleft_1 \mid (\eta \mapsto x) \in \mathcal{C}\}$ % values w/o masks

  def applyV($\varnothing \subsetneq \epsilon \subseteq \rho_r$) $\in \langle\rangle$ % accumulate values (aggregands)
    capclose($\epsilon\downharpoonleft_{\mathrm{HEAD}}$)
    foreach $(\eta \mapsto \langle\beta,\mu\rangle) \in \mathcal{C}$ where $\eta \subseteq \epsilon\downharpoonleft_{\mathrm{HEAD}}$ do
      let $\wr v@m \int$ = AnswerFor($\epsilon,\omega,\mu$) where $\omega = \bigcup\{\eta' \in \mathsf{dom}(\mathcal{C}) \mid \eta' \subseteq \eta\}$ % §3.4.1
      $\mathcal{C}(\eta)$ ← $\langle \wr v@m \int \uplus \beta, \mu\rangle$

  def applyM($\varnothing \subsetneq \alpha \subseteq \rho_r$) $\in \langle\rangle$ % accumulate masks
    capclose($\alpha\downharpoonleft_{\mathrm{HEAD}}$)
    foreach $(\eta \mapsto \langle\beta,\mu\rangle) \in \mathcal{C}$ where $\eta \subseteq \alpha\downharpoonleft_{\mathrm{HEAD}}$ do $\mathcal{C}(\eta)$ ← $\langle\beta, \alpha \cup \mu\rangle$

  def capclose($\eta \subseteq \kappa$) $\in \langle\rangle$ % put $\eta \in \mathsf{dom}(\mathcal{C})$, $\cap$-closed
    if $\eta \in \mathsf{dom}(\mathcal{C})$ then return
    else foreach $(\tau \mapsto \_) \in \mathcal{C}$ do let $\eta' = \tau \cap \eta$ in
      if $\eta' \notin (\mathsf{dom}(\mathcal{C}) \cup \{\varnothing\})$ then
        let $A = \{\beta \in \mathsf{dom}(\mathcal{C}) \mid \eta' \subseteq \beta, \forall_{\beta' \in \mathsf{dom}(\mathcal{C})} \beta' \subseteq \beta \Rightarrow \eta' \nsubseteq \beta'\}$
        $\mathcal{C}(\eta')$ ← $\langle \biguplus_{\alpha \in A} \mathcal{C}(\alpha)\downharpoonleft_1, \bigcup_{\alpha \in A} \mathcal{C}(\alpha)\downharpoonleft_2\rangle$

def refineRuleSuffix($r \in \Xi$, applyM, applyV) $\in \langle\rangle$
  return go
  def go($\alpha_{\mathtt{k}} \subseteq \rho_r$, $\alpha_{\mathtt{v}} \subseteq \alpha_{\mathtt{k}}$, $i \in \mathbb{N}_1^{n_r+1}$)
    if $\alpha_{\mathtt{k}} = \varnothing$ then return $\langle\rangle$ % incompatible keys
    elif $\alpha_{\mathtt{v}} \neq \varnothing$ then
      if $i = n_r + 1$ then applyV($\alpha_{\mathtt{v}}$) % end of rule
      else foreach $(\sigma \mapsto v) \in$ Lookup($\alpha_{\mathtt{v}}\downharpoonleft_{\mathrm{SG}.i}$) toposorted by $\subseteq$ ascending on $\sigma$ do
              go($\alpha_{\mathtt{k}}[\sigma/\mathrm{SG}.i.1]$, $\alpha_{\mathtt{v}}[\langle\!\langle\sigma, \{v\}\rangle\!\rangle/\mathrm{SG}.i]$, $i+1$)
    applyM($\alpha_{\mathtt{k}}$) % before returning, mask

Lookup $\in \prod_{(\sum_{k \in \kappa} \tau_k) \subseteq \langle\mathcal{H},\mathcal{H}\rangle} \bigcup_{K \in \mathtt{ficc}(\kappa \cap \mathcal{I})} \prod_{\kappa \in K} (\{\mathrm{NULL}\} \cup \bigcap_{k \in \kappa} \tau_k)$
```

Listing 3.3: Default-based COMPUTE, assuming ANSWERFOR from §3.4.1 as a primitive operation. REFINERULESUFFIX now tracks *two* subsets of ρ: the first, α_{k}, *ignores values* and gradually refines down to θ, while the second, α_{v}, is the more typical gradual refinement down to ϵ. During its operation, its nested loops each advance in topologically sorted order so that calls to APPLYV and APPLYM are made on monotonically non-$\preceq$-decreasing $\mathcal{L}$-replacements.

bulk aggregation to turn bags of aggregands into the final answer, $\mathcal{Y}$.[93] computeRule forms nested loops using refineRuleSuffix, which now tracks its position in the rule's subgoals, i; a superset of ϵ, α_{v}; and a superset of θ, α_{k}. These supersets are exact at the leaves of the search tree. refineRuleSuffix calls applyV to contribute the answers it finds (at the leaves) and applyM whenever it returns to add to masks. Within refineRuleSuffix, $\mathcal{C}$ is a BF which stores both values and masks; the masks are removed upon return. capclose adds its argument to, and enforces the $\cap$-closure property of, $\mathcal{C}$.

Mask Estimates The algorithm, as is typical of search algorithms, only enumerates successes (i.e., replacements with non-empty ϵ), while the theory of §3.4.3 seems to depend on enumeration of both ϵ and θ for every $\mathcal{L}$-replacement. While the algorithm does derive masks from successes (the rightmost left-facing dotted arrows in figure 3.4), it also derives masks from other α_{k} sets during its execution, when it unwinds the recursion of refineRuleSuffix. These α_{k} sets correspond to θs derived from a *prefix* of a rule query, $\vec{\tau} = \langle \tau_1, \ldots, \tau_{i-1}, \mathcal{H}, \ldots, \mathcal{H} \rangle$ (wherein entries i through n_r are unspecified, i.e., are the entire term universe, $\mathcal{H}$), which therefore *subsume* the θs of any possible query $\leq \vec{\tau}$. While these are over-estimates of the sets tracked by the theory, they are not too large, in the sense of improperly masking later results, because they will only mask later results that share the prefix, which has just been exhaustively searched.

3.4.4.1 Discussion of the Example Trace

Figure 3.4 includes a trace of computeRule of listing 3.3 running on the example of §3.4.3.1 on inputs described in the former's caption. The root node of the search tree corresponds to the outermost RefineRuleSyntax call's Lookup of $\mathtt{r}\langle \mathcal{H}, \mathcal{H} \rangle$.

The first entry in the log (cyan box) arises from the discovery of aggregands $\lbag \otimes\langle 4, 6\rangle @1 \rbag$ for $\{\mathtt{rs}\langle 0 \rangle\}$ from the replacement $\langle \{\mathtt{r}\langle 0,0\rangle\}, \{\mathtt{s}\langle 0\rangle\}\rangle$. The second entry adds this replacement's pre-answer to the mask for $\{\mathtt{rs}\langle 0 \rangle\}$, ensuring that any later-discovered values assigned to the *ground* rule query $\langle \mathtt{r}\langle 0,0\rangle, \mathtt{s}\langle 0\rangle\rangle$ are discarded. Indeed, we see exactly this case for the replacement $\langle \{\mathtt{r}\langle x,x\rangle \mid x \in \mathbb{Z}\}, \{\mathtt{s}\langle 0\rangle\}\rangle$, on the third line of the log (corresponding to the second leaf of the tree): the algorithm revisits this ground replacement, due to the unification of x and y (in ρ) by the diagonal $\mathtt{r}$, and must mask its incorrect contribution of $\otimes\langle 3, 6\rangle$. As the head here is still just $\{\mathtt{rs}\langle 0\rangle\}$, ϵ needs no further processing.

The third leaf of the tree generates contributions which claim themselves to be applicable to all $\mathtt{rs}\langle \mathbb{N} \rangle$ (red box). However, when the head is restricted to $\{\mathtt{rs}\langle 0\rangle\}$, we find ourselves *again* revisiting *only* the ground rule query $\langle \mathtt{r}\langle 0,0\rangle, \mathtt{s}\langle 0\rangle\rangle$, and so, in fact, these contributions are destined to $\mathtt{rs}\langle \mathbb{N} \setminus \{0\}\rangle$, which is encoded by contributing to $\mathtt{rs}\langle \mathbb{N}\rangle$ but not $\{\mathtt{rs}\langle 0\rangle\}$. The algorithm then proceeds to mask off the diagonal entries in all heads $\{\mathtt{rs}\langle \mathbb{Z}\rangle, \mathtt{rs}\langle \mathbb{N}\rangle, \{\mathtt{rs}\langle 0\rangle\}\}$, though there is some redundancy in its efforts (gray log lines).

[93] To ensure that bulk aggregation is defined, 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 $\text{aggr}(\eta) \overset{\text{def}}{=} \text{selt}(\{\text{aggr}(h) \mid h \in \eta\})$ is well-defined when we use it. In practice, we will not compute all of $\mathcal{Y}$ and then reduce it; instead, we will exploit the required properties of aggregators to directly aggregate while computing $\mathcal{C}$ and then again to obtain results equivalent to aggregating across $\mathcal{Y}$.

$\sigma \mapsto v$	$\{\mathtt{g}\langle 4\rangle\} \mapsto 2$	$\mathtt{g}\langle \mathbb{Z}\rangle \mapsto 3$	$\mathtt{g}\langle \mathcal{H}\rangle \mapsto 2$
α_k	$\mu_1 \overset{\text{def}}{=} \rho[\{4\}/\textsc{sg}.1.1.1]$	$\mu_2 \overset{\text{def}}{=} \rho[\mathbb{Z}/\textsc{sg}.1.1.1]$	$\mu_3 \overset{\text{def}}{=} \rho[\mathcal{H}/\textsc{sg}.1.1.1] = \rho$
α_v	$\mu_1[\{2\}/\textsc{sg}.1.2]$	$\mu_2[\{3\}/\textsc{sg}.1.2]$	$\mu_3[\{2\}/\textsc{sg}.1.2]$
APPLYV ∩	$\{\mathtt{f}\langle 4,2\rangle\} \mapsto \langle \varnothing, \varnothing\rangle$	$\mathtt{f}\langle \mathbb{Z}, \{3\}\rangle \mapsto \langle \varnothing, \varnothing\rangle$ $\{\mathtt{f}\langle 4,3\rangle\} \mapsto \langle \varnothing, \mu_1\rangle$	$\mathtt{f}\langle \mathcal{H}, \{2\}\rangle \mapsto \langle \varnothing, \varnothing\rangle$ $\mathtt{f}\langle \mathbb{Z}, \{2\}\rangle \mapsto \langle \varnothing, \mu_2\rangle$
APPLYV	$\{\mathtt{f}\langle 4,2\rangle\} \mapsto \langle \lbag 1 \rbag, \varnothing\rangle$	$\mathtt{f}\langle \mathbb{Z}, \{3\}\rangle \mapsto \langle \lbag 1 \rbag, \varnothing\rangle$ $\{\mathtt{f}\langle 4,3\rangle\}$ *masked*	$\mathtt{f}\langle \mathcal{H}, \{2\}\rangle \mapsto \langle \lbag 1 \rbag, \varnothing\rangle$ $\mathtt{f}\langle \mathbb{Z}, \{2\}\rangle$ *masked* $\{\mathtt{f}\langle 4,2\rangle\}$ *masked*
APPLYM ∩	$\mathtt{f}\langle 4, \mathcal{H}\rangle \mapsto \langle \varnothing, \varnothing\rangle$	$\mathtt{f}\langle \mathbb{Z}, \mathcal{H}\rangle \mapsto \langle \varnothing, \varnothing\rangle$	*no change to* $f\langle \mathcal{H}, \mathcal{H}\rangle$
APPLYM	$\mathtt{f}\langle 4, \mathcal{H}\rangle \mapsto \langle \varnothing, \mu_1\rangle$ $\{\mathtt{f}\langle 4,2\rangle\} \mapsto \langle \lbag 1 \rbag, \mu_1\rangle$	$\mathtt{f}\langle \mathbb{Z}, \mathcal{H}\rangle \mapsto \langle \varnothing, \mu_2\rangle$ $\mathtt{f}\langle \mathbb{Z}, \{3\}\rangle \mapsto \langle \lbag 1 \rbag, \mu_2\rangle$ $\mathtt{f}\langle \{4\}, \mathcal{H}\rangle \mapsto \langle \varnothing, \mu_2\rangle$ $\{\mathtt{f}\langle 4,3\rangle\} \mapsto \langle \varnothing, \mu_2\rangle$ $\{\mathtt{f}\langle 4,2\rangle\} \mapsto \langle \lbag 1 \rbag, \mu_2\rangle$	*all items masked by* $\mu_3 = \rho$

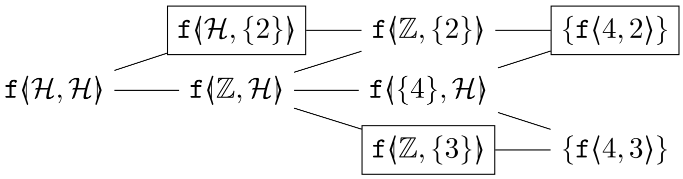
Table 3.2: All updates made to $\mathcal{C}$ during the execution of listing 3.3's COMPUTERULE when applied to the rule $\{(\mathtt{f}\langle a, v\rangle \leftarrow 1) \Leftarrow \langle \mathtt{g}\langle a\rangle \mapsto v\rangle \mid a, v \in \mathcal{H}\}$ given the results from LOOKUP in the header row. Execution proceeds top-down within each column, visited left-to-right (corresponding to larger default answers). The phases labeled with ∩ are the calls to CAPCLOSE. An APPLYV entry may be labeled "*masked*" to mean that the rule answer, α_v, which gave rise to the head in question, is a subset of the mask set already associated with that head (the second component of the tuple returned by $\mathcal{C}$). See §3.4.4.2 for additional discussion.

The fourth leaf of the tree generates contributions nominally for $\mathtt{rs}\langle \mathbb{Z}\rangle$ (yellow box), having constrained $y = 0$ in ρ. Again $\{\mathtt{rs}\langle 0\rangle\}$ masks this contribution (the 0 from the head and from y being sufficient to send the first subgoal to $\{\mathtt{r}\langle 0,0\rangle\}$). The application of these contributions to the output heads $\mathtt{rs}\langle \mathbb{Z}\rangle$ and $\mathtt{rs}\langle \mathbb{N}\rangle$ rely on the set difference on heads (rather than groundings) in ANSWERFOR in a way that no prior entry has: it is *not the case* that the masked ϵs for the non-ground rule query $\langle \mathtt{r}\langle \mathbb{Z}, \mathbb{Z}\rangle, \{\mathtt{s}\langle 0\rangle\}\rangle$ and for these heads are uniform: every $\mathtt{rs}\langle \mathbb{N}\rangle$ *other than* $\{rs\}0$ has one contribution of $\otimes\langle 2, 6\rangle$. Once again, the diagonal strikes, but now *from the masks*: the $\mathtt{rs}\langle 0\rangle$ entry in these heads is associated with $\mathtt{r}\langle 0,0\rangle$, which is already masked! However, $\{\mathtt{rs}\langle 0\rangle\}$ is already an answer in the output BF, and so we may *obstruct* it from ANSWERFOR's consideration. The same phenomenon recurs in the lavender box from the next and final answer in the search tree.

3.4.4.2 Functional Dependencies Within Keys

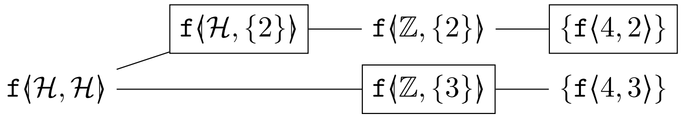
μDyna clearly privileges the functional dependence between keys and values, but it is nevertheless possible to induce other functional dependencies within the keys of items, and it is informative to consider the system's behavior under such circumstances. Notably, this scenario strongly separates (the head projections of) θ and ϵ and highlights the masking behavior of our algorithm. Concretely, we will consider the rule $\{(\mathtt{f}\langle a, v\rangle \leftarrow 1) \Leftarrow \langle \mathtt{g}\langle a\rangle \mapsto v\rangle \mid a, v \in \mathcal{H}\}$ and query $\mathtt{f}\langle \mathcal{H}, \mathcal{H}\rangle$, when LOOKUP returns the answers $\{\mathtt{g}\langle 4\rangle\} \mapsto 2$, $\mathtt{g}\langle \mathbb{Z}\rangle \mapsto 3$, and $\mathtt{g}\langle \mathcal{H}\rangle \mapsto 2$ (in that order, and so describing a BF). We deliberately reuse the value 2 in this collection of answers. Initially, $\mathcal{C} \leftarrow \{\mathtt{f}\langle \mathcal{H}, \mathcal{H}\rangle \mapsto \varnothing\}$ (line 7 of listing 3.3). Execution then proceeds as shown in table 3.2.

After all that, we are able to extract the resulting $\mathcal{C}$ by reading the most recent update to each key from the chart (and apply aggregators, but here, the only aggregation is to send $\Lbag 1@m \Rbag$ to 1). We can render the result as follows; boxed nodes have value 1, all other nodes have NULL, and subsets are positioned to the right of (rather than beneath) their supersets.



$\mathtt{f}\langle\mathcal{H},\{2\}\rangle$
$\mathtt{f}\langle\mathbb{Z},\{2\}\rangle$
$\{\mathtt{f}\langle 4,2\rangle\}$
$\mathtt{f}\langle\mathcal{H},\mathcal{H}\rangle$
$\mathtt{f}\langle\mathbb{Z},\mathcal{H}\rangle$
$\mathtt{f}\langle\{4\},\mathcal{H}\rangle$
$\mathtt{f}\langle\mathbb{Z},\{3\}\rangle$
$\{\mathtt{f}\langle 4,3\rangle\}$

We see that the algorithm constructs a kind of minimal BF, though it is operating without lookahead. This explains the striking absence of $\mathtt{f}\langle\mathcal{H},\{3\}\rangle$ in the figure above: there was no rule answer to justify its presence. The lack of lookahead does, however, result in this BF containing some redundancy: the $\mathtt{f}\langle\{4\},\mathcal{H}\rangle$ and $\mathtt{f}\langle\mathbb{Z},\mathcal{H}\rangle$ elements of the base, added out of an abundance of caution on the part of APPLYM, can be eliminated as they are not serving to override their (unique) parent. These base points were created as, naïvely, the the algorithm was preparing for the possibility that other $\{\mathtt{f}\langle\{4\},\tau\rangle \mid \tau\}$ or, later, $\{\mathtt{f}\langle\tau,\tau'\rangle \mid \tau\cap\mathbb{Z}\neq\varnothing,\tau'\}$ items may be proven by less-specific results from LOOKUP. Simplifying the result as described yields a rather natural picture:



$\mathtt{f}\langle\mathcal{H},\{2\}\rangle$
$\mathtt{f}\langle\mathbb{Z},\{2\}\rangle$
$\{\mathtt{f}\langle 4,2\rangle\}$
$\mathtt{f}\langle\mathcal{H},\mathcal{H}\rangle$
$\mathtt{f}\langle\mathbb{Z},\{3\}\rangle$
$\{\mathtt{f}\langle 4,3\rangle\}$

We can read off the answer from this picture: all $\mathtt{f}\langle\mathcal{H},\mathcal{H}\rangle$ are NULL, except $\mathtt{f}\langle\mathcal{H},\{2\}\rangle$ are 1 instead, except $\mathtt{f}\langle\mathbb{Z},\{2\}\rangle$ are NULL instead, because $\mathtt{f}\langle\mathbb{Z},\{3\}\rangle$ are 1 instead, except $\mathtt{f}\langle 4,3\rangle$ is NULL instead, because $\mathtt{f}\langle 4,2\rangle$ is 1 instead.

3.4.5 Possible Optimizations

3.4.5.1 Running Aggregation

As mentioned in §3.2.1.2, for simplicity of the pseudocode listing, we are deferring aggregation to the end and, instead, storing bags of results within our accumulators. As these bags are only enlarged during execution, we could exploit the AC-reducer property of aggregators and store the aggregated result (modulo concerns about inexact values). This is quite likely a winning strategy when values are relatively "static," in the sense that there are no, or very few, updates to items within the larger context of the solver's execution. (Despite our focus on backward-chaining in this chapter, we hope that these algorithms can find a home in a mixed-chaining solver!) In the case of more volatile $\mathcal{I}_{\text{inp}}$, however, indexing (by name) the partial sums within the $\mathcal{C}$ accumulators may lead to an asymptotic improvement vs. recomputing all values for every update. The indices would, by necessity, be functions of the replacement (prefix) or its *pre-answers* (α_k), rather than answers, because the indices must retain a notion of object identity even as items' values change.

3.4.5.2 Completed Results

If ever the masking set μ for some τ in $\mathcal{C}$ within a run of $\textsc{computeRule}(r,\_)$ is such that $\rho_r[\tau/\textsc{head}] \subseteq \mu$, then τ will certainly never receive another contribution: everything is certainly masked (i.e., all subsequent ψ for τ will be $\varnothing$). Moreover, this property must be true for any $\tau' \subseteq \tau$ also being considered or yet to be considered (i.e., yet to be added by $\textsc{capclose}$), so we can prune entire $\cap$-closed sections of $\mathcal{C}$ from iteration by the `foreach`-es in both $\textsc{applyV}$ and $\textsc{applyM}$ without altering the answer. For simplicity of exposition, we have omitted this handling from the pseudocode.

3.4.5.3 Non-decreasing-Specificity Search

The algorithm presented herein iterates over the collection of $\mathcal{L}$-replacements in a *non-increasing-specificity* topological order. The aggregands that are computed from decreased-specificity replacements are "fanned out" (copied), by the `foreach`-es within $\textsc{applyV}$ and $\textsc{applyM}$, to heads derived from increased-specificity replacements (in addition to being applied to the head derived from the decreased-specificity replacement in question); this process respects the masks accumulated at these increased-specificity heads. (The similar `foreach` loop within $\textsc{capclose}$ merely copies existing aggregands and masks out to newly created elements of the base.) This is perhaps a sensible order when one assumes that there will be few (as yet uncompleted) overrides, and it seems required for the use of running aggregates as suggested above. However, especially if one is maintaining index structures for fast update processing, it may be beneficial to traverse the replacements in the other order. In this case, the analogue of $\textsc{capclose}$ would merge the index structures from more general heads (removing conflicting values; the structure of the problem guarantees that the correct replacements will arrive as we continue traversing replacements). $\textsc{applyV}$ and $\textsc{applyM}$ would update (and remove) entries in the structures maintained for more-specific heads, which would then be copied out to, and subsequently revised for, even-more-specific heads.

Example 39: To understand the difficulties that might arise in attempting this kind of non-decreasing-specificity operation, consider the rule $\{(\mathtt{f}\langle x\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{g}\langle x,y\rangle \mapsto v\rangle \mid v,x,y\}$ with all $\mathtt{f}^{/1}$ items aggregating with summation. Suppose, further, that $\mathcal{L} = \{\mathtt{g}\langle \mathcal{H},\{0\}\rangle \mapsto 4, \mathtt{g}\langle \mathcal{H},\{1\}\rangle \mapsto 5, \{\mathtt{g}\langle 3,0\rangle\} \mapsto 6\}$. We can conclude, from the first two elements of this definition of $\mathcal{L}$ that the result of backward-chaining will include $\mathtt{f}\langle \mathcal{H}\rangle \mapsto \sum \lbag 4,5 \rbag = 9$. From the *second* and third elements, we see that there is an override, $\{\mathtt{f}\langle 3\rangle\} \mapsto \sum \lbag 5,6 \rbag = 11$. That is, the $\mathtt{g}\langle \mathcal{H},\{1\}\rangle$ element of $\mathcal{L}$ contributes to $\{\mathtt{f}\langle 3\rangle\}$ but the $\mathtt{g}\langle \mathcal{H},\{0\}\rangle$ element did not, because it was *completely masked* for the head $\{\mathtt{f}\langle 3\rangle\}$ by the $\{\mathtt{g}\langle 3,0\rangle\}$ element of $\mathcal{L}$. Had we aggregated the $\mathtt{f}\langle \mathcal{H}\rangle$ contributions as we found them, it would have been difficult to *remove* the aggregand 4 and replace it with 6. ◊

3.4.6 Aside: Relaxing Aggregator Agreements

We required, as part of the development of §3.4.4 (in particular, see footnote 93 therein), that we could define a notion of "bulk aggregation" to a set of heads. For simplicity, we did this by ensuring that any two items which were in the same rule's head were assigned the same aggregator; this meant that we had only one aggregator to consider in line 4 of listing 3.3. However, there are two relatively straightforward extensions that could be made, should that restriction be perceived as onerous.

First, we could instead require that the input μDyna program assign aggregators to items using a (finite) backed-off function. That is, we would require that $\text{aggr}(\cdot)$ be amenable to encoding by a finite collection of defaults and overrides. Rather than seeding our $\mathcal{Y}$ accumulator with the single entry $\kappa \mapsto \langle\varnothing\rangle$, as we do at present at line 2 of listing 3.3, we could initialize it to have as keys each set from the BF encoding of $\text{aggr}(\cdot)$ that had non-empty intersection with κ. The remainder of the algorithm would work without modification. Formally, we are thus computing the **application-join**, $\wedge^{\$}$, of $\text{aggr}(\cdot)$ (viewed as its BF encoding) and $\mathcal{Y}$; that is, we compute $\mathcal{A} \wedge^{\$} \mathcal{Y} \in \mathcal{I}_{\text{der}} \hat{\to} \mathcal{H}'$ where \$ represents function application, i.e., $\$(g)(x) \overset{\text{def}}{=} g(x)$. This modification combines compatibly with running aggregation above.

On the other hand, perhaps even the notion of a single aggregator assigned statically to an item is too much (though we tend to think it will not be, in practice). In such a case, one could imagine making the result of a rule the *pair* the aggregand from a rule with the desired aggregator. A single "meta-aggregator" then could check that all aggregands to a given item had voted the same way and, if so, use the selected aggregator to combine results. In the event of conflicts, the meta-aggregator could return NULL or an error value (to be discussed in §6.1).

3.5 Partial Memoization

In §2.2, Lookup operated on a *per-item* basis; i.e., it had type $\mathcal{I} \to \mathcal{H} \cup \{\text{NULL}\}$. Internally, it had read/write access to a memo table $\mathcal{M}$, a dynamic, total function of type $\mathcal{I} \to \mathcal{H} \cup \{\text{NULL}, \text{UNK}\}$. UNK was, like NULL, an assumed symbol outside $\mathcal{H}$; it was used to represent a *lack of cached value* in the memo table (rather than making $\mathcal{M}$ partial). If Lookup was invoked on $t \in \mathcal{I}$ and $\mathcal{M}(t) = \text{UNK}$, Lookup was obligated to call Compute to derive a value for this item.[94] Now that we have *assumed* an extended Lookup which both operates on and returns *sets of items* (as well as their associated cached values), we are left with the question of how memoization behaves within this procedure.

Both extremal memoization policies—all or nothing—are trivial. If an entire query $\Sigma_{k \in \kappa} \tau_k$ is uniformly not memoized, Lookup should invoke Compute on the entirety of κ and filter the result for intersection with the τ_ks.[95] If the query is entirely memoized, the call to Compute can be skipped and filtering can be done using the cached values. The interesting cases are policies between these, where the queried set of items κ may contain memoized and un-memoized items (these being potentially different sets at different points in the program's execution, even).

[94] The extension described in §2.5, and in Filardo and Eisner [60], for handling cyclic circuits permits Compute, but not Lookup, to abort the computation and "guess" a value.

[95] In actuality, because Lookup is invoked with the SG.i projection of σ which is then immediately refined with the result, this filtration could be done implicitly within the refinement operator. We prefer our more modular treatment in general but admit that, in practice, one would want to inline across function call boundaries.

In terms of set theory, for maximal impact of the memo table, we would like for $\textsc{Lookup}(\Sigma_{k \in \kappa} \alpha_k)$ to invoke $\textsc{Compute}(\beta)$ with β being the difference between κ and the contents of the memo table; formally, $\beta = \kappa \setminus \{k \in \kappa \mid \mathcal{M}(k) \neq \textsc{unk}\}$. However, operationally, this set is potentially complex to describe and to subsequently use within Compute (in a way that sets arising from rules are not likely to be). Thankfully, there are two degrees of freedom here at runtime which may be useful for optimization, though we leave proper treatment of the details to future work. First, Compute distributes over unions; that is, we may partition β into simpler sets and invoke Compute on each, separately. This requires no post-processing on the part of Lookup since it only promises to return a function on *some* partition of κ, unpredictable by the caller. Secondly, it is acceptable to *under-approximate* the memo table and *discard* (or *merge*, potentially, in the case of marked answers as in §2.2.4.2) superfluous work done by Compute. Thus, we may choose to approximate β (or one of its subsets, having applied the prior transform) with a simpler, larger set; for example, we may take a *co-sparse* β and replace it with a rectangular superset. Of course, these transforms can be combined and iterated as appropriate.

The algorithms given in this paper remain correct (but possibly inefficient) regardless of the application of these kinds of operations; all that matters is that Lookup functions as the correct black box.

3.6 Future Work

3.6.1 Head-Value Covariance

Let us expend a little ink considering what it would take to permit rules with head-value covariance, such as $\{(\mathtt{f}\langle x\rangle \leftarrow x) \Leftarrow \langle\rangle \mid x\}$. Some relatively straightforward changes to the definition of backed-off functions and rule answer sets suffice to allow the system to *consume* such covariant results (e.g., the kv-pairs $\{\langle \mathtt{f}\langle x\rangle, x\rangle \mid x\}$ in response to a call to Lookup). The *aggregation* and *production* of such results are more difficult.

The definition of backed-off functions that we gave in §3.4.2.2 was restricted to the piecewise-constant case. In particular, we took $\mathcal{B} \in (K \setminus \{\varnothing\}) \to \alpha$ to encode a function $f \in (\bigcup K) \to \alpha$. However, there is nothing fundamental about this kind of non-monotonic, smallest-containing-set-wins encoding that requires that the enclosers be associated with constants. If, instead, we took $\mathcal{B} \in \Pi_{\kappa \in K \setminus \{\varnothing\}} \kappa \to \alpha$, we could decode $f(x)$ from $\mathcal{B}$ by passing x twice: $f(x) = \mathcal{B}(\lceil\{x\}\rceil_{\mathcal{B}})(x)$.

The definitions of §3.4.3.3 are easily extended to use this extended definition of backed-off functions. The concept of a replacement, being a function only of the *encoding domain* K, needs no revision. However, the non-ground rule answer corresponding to a replacement $\vec{\tau} \in \mathfrak{R}_{r,\mathcal{L}}$ is now $\epsilon^{\vec{\tau}}_{r,\mathcal{L}} \overset{\text{def}}{=} \rho_r[\mathcal{L}(\lceil\tau_1\rceil_{\mathcal{L}})/\textsc{sg}.1]\cdots[\mathcal{L}(\lceil\tau_{n_r}\rceil_{\mathcal{L}})/\textsc{sg}.n_r]$. While this definition does not make it quite so explicit, it is still true that $\epsilon^{\vec{\tau}}_{r,\mathcal{L}} \subseteq \theta^{\vec{\tau}}_r$ as the refinements of $\{\textsc{sg}.i.1 \mid i\}$ within the definition of the latter quantity are all implicitly replicated by the domain of $\mathcal{L}(\lceil\tau_i\rceil_{\mathcal{L}})$ in the former's refinement of $\textsc{sg}.i$.

Procedurally, we should replace line 34 of listing 3.3 and the next with

```
else foreach $f \in \textsc{Lookup}(\alpha_{\mathrm{v}}\downharpoonleft_{\textsc{sg}.i})$ toposorted by $\subseteq$ ascending on $f\downharpoonleft_1$ do
      $\textsc{go}(\alpha_{\mathrm{k}}[f\downharpoonleft_1/\textsc{sg}.i.1],\ \alpha_{\mathrm{v}}[f/\textsc{sg}.i],\ i+1)$
```

where we have replaced the match of Lookup's result against $\sigma \mapsto v$ with simply a function, represented as a set of ordered pairs, f. We use $f\downharpoonleft_1$ to obtain its domain for refining α_{k} and refine α_{v} using f itself.

If the functions we permit are arbitrary, we have not really solved anything: the replacement operations in the definition of ϵ above are not obviously tractable for arbitrary functions viewed as sets of kv-pairs. If, however, we are careful to discriminate between *equi*-covariance, à la $\{(\mathtt{f}\langle x\rangle \hookleftarrow x) \Leftarrow \langle\rangle \mid x \in \tau\}$, and general covariance, à la $\{(\mathtt{f}\langle x\rangle \hookleftarrow x+1) \Leftarrow \langle\rangle \mid x \in \tau\}$ with $\tau \subseteq \mathbb{R}$, there may be a chance. The former is likely easily discharged by whatever runtime theory of sets exists within the solver, as such equalities are already rampant. The latter is a kind of *delayed constraint* [33] and requires potentially vast, new machinery.

All we have said so far pertains to the *consumption* of backed-off functions encoding (equi-)covariant functions. To produce such things in general would require that our aggregators be able to consume, manipulate, and produce *expressions* involving not just values (with multiplicities) but *projections of the head* (with multiplicities) as well. Consider, for example, the combined effects of the rule $\{(\mathtt{f}\langle x\rangle \hookleftarrow 1) \Leftarrow \langle\rangle \mid x \in \tau\}$ with our recurring example $\{(\mathtt{f}\langle x\rangle \hookleftarrow x) \Leftarrow \langle\rangle \mid x\}$ with $\mathtt{f}^{/1}$ aggregated using some $\oplus$ operator. We would have to yield an encoding of $\mathtt{f}\langle x\rangle \mapsto x \oplus 1$, which we have just said was potentially complicated to handle. Similarly, a program containing two copies of the latter rule would have to encode $\mathtt{f}\langle x\rangle \mapsto x \oplus x$. We have not endeavoured to engineer such an algebra, and speculate that such effort will be replete with special cases. Perhaps the simplest answer would be to permit programs to define infinitely-covariant rules only for items that provably did not need aggregation: that is, for which we are guaranteed there is *at most one* aggregand. (Recall that we require aggregators f to obey $\forall_x\, f(\Lbag x \Rbag) = x$.) However, such a modest extension in expressiveness does not seem to justify the engineering effort; any program written in the extended language could be transformed by inlining ("unfolding") the aggretion-free items' definition into rules using those items as subgoals.

3.6.2 Forward-Chaining

Forward chaining in generalized logic programs, as supported by μDyna, opens some rather complex issues, even before we consider set-at-a-time reasoning as we have done in this chapter. Consider again the rule from example 27 (in §3.1.3), $\{(\mathtt{f}\langle a\rangle \hookleftarrow 1) \Leftarrow \langle\mathtt{a}\langle\rangle \mapsto a\rangle \mid a \in \mathcal{H}\}$. When revising the value of $\mathtt{a}\langle\rangle$, were we to follow the algorithms of §2, we might ask for "the set of $\mathtt{a}\langle\rangle$'s children," which includes the *infinitely many* $\mathtt{f}^{/1}$ items. However, certainly only two $\mathtt{f}^{/1}$ items, $\mathtt{f}\langle a_{\mathrm{o}}\rangle$ and $\mathtt{f}\langle a_{\mathrm{n}}\rangle$, corresponding to old and new values of $\mathtt{a}\langle\rangle$, respectively, are actually subject to change in response to a notification departing $\mathtt{a}\langle\rangle$. However, if we are able to *flush* the memoized value of $\mathtt{a}\langle\rangle$, it may not be possible to reconstruct this set of "current children" thereof. Thus, even in the case of a *quantification-less rule*, we must either *couple* the behavior of the agenda and flushing, or we must be prepared to demand that *all* $\mathtt{f}^{/1}$ items recompute their values, classify the differences between old and new values for this infinite set, and push (finite) messages to the agenda.

Generally speaking, we envision constructing **update propagator**s for each subgoal of each rule. The job of a propagator for the i^{th} subgoal of rule r is to characterize the influence of an update to an item, whose name is an element of $\rho_r\!\downharpoonleft_{\textsc{sg}.i.1}$, on the items of $\rho_r\!\downharpoonleft_{\textsc{head}}$. The i^{th} subgoal is said to be **driving** this propagator (not to be confused with the external *driver* program controlling the solver). These propagators run $\texttt{refineRuleSuffix}$ on the *other* (i.e., not i), **passenger**, subgoals of the rule to determine the full effect, the i^{th} subgoal having been refined *by the update itself*. Unlike backward-chaining, the *head* is not refined prior to $\texttt{refineRuleSuffix}$. This implies that we may need additional conditions on a program beyond K-sufficient (non-ground) range restriction in order for forward-chaining to operate successfully. Furthermore, planning of subgoal order is required and may differ for each propagator; recall §3.2.1.1 and see §5.3.

The algorithms of this chapter, especially that of §3.4, build up complex internal data structures in order to answer backward-chaining queries. Efficient forward-chaining would seem to require chasing notifications through these structures. In order to facilitate these revisions, we would likely wish to *index* the (partial) aggregations within these structures, as mentioned in §3.4.5, along the lines given in §2.3.3.1 and §2.4.1. Because we can now have *infinite*-multiplicity aggregands, some additional care is required. These aggregands *must* be indexed by source (i.e., preanswer) so that they can be retracted. Items with finite multiplicity (or even any finite-multiplicity contributions in addition to infinite-multiplicity contributions) are free to count, à la “baggregators” (recall example 15, in §2.4.1), so long as the system does not mix named and counted aggregands. (That is, if some aggregands were indexed by name, they must be retracted by name, too.)

3.6.2.1 Update Propagators using Explicit Message Representations

We envision *reifying* the notification taxonomy of table 2.1 (in §2.2.3) as *terms* that are used only during the propagation of notifications. That is, we imagine that we add to $\mathcal{H}$ some representation of *pair* of an aggregator and aggregand and terms of forms like $\mathtt{\$notif}\langle o, n, d\rangle$ or $\mathtt{\$notif}\langle d\rangle$ where o and n represent the old and new values of an item's notification and d its delta. So armed, we could create the pair $\langle \mathtt{f}\langle 3\rangle, \mathtt{\$notif}\langle 1, 2, \mathtt{\$delta}\langle \oplus, 1\rangle\rangle\rangle$ to represent $\mathtt{f}\langle 3\rangle : \leftarrowtriangle 2; \text{was}\,1; \oplus 1$. We could endow our arithmetic primitives with the ability to manipulate such objects, for example, defining the item $\mathtt{\$notif}\langle \mathtt{\$delta}\langle \oplus, d\rangle\rangle * x$ to have value $\mathtt{\$notif}\langle \mathtt{\$delta}\langle \oplus, v\rangle\rangle$ where v is the value of $d * x$.

The hope is that, by refining the driving subgoal of a rule to such a pair and then using the existing backward-chaining machinery of this section, that the system will *naturally* produce updates (albeit, pronounced as $\mathtt{\$notif}^{/k}$ terms) in the $\textsc{hr}$ projection of the rule-answer-analogue set so computed. There are several known complexities here. First, a subgoal may not be able to handle the *particular* form being asked, but may be able to handle a different form. If it is possible to transmogrify the form at hand into an understood form (perhaps by the operations of figure 2.10, in §2.4.1), that may be sufficient to resolve the difficulty. Secondly, if such transformations are not possible, as might occur if the subgoal is unable to handle $\mathtt{\$notif}^{/k}$ terms at all, one needs to *split* the subset of ρ being considered within the recursive behavior analogous to that of $\texttt{refineRuleSuffix}$, tracking separately the former and current values. Thirdly, such $\mathtt{\$notify}^{/k}$ objects may enter the $\textsc{head}$, not just the $\textsc{res}$. In that case, one needs to split the update object itself into old and new components for further processing within the system. Fourth, the system must handle old and new values of $\textsc{unk}$ itself, without this mechanism (and might handle $\textsc{null}$ itself, but one could perhaps imagine adding a $\mathtt{\$}\textsc{null}\langle\rangle$ for explicit representation of such). Fifth, and last, one must ensure that values computed incrementally, in response to notifications, would match the values computed from scratch; this is particularly difficult to ensure when using inexact values and delta notifications.

3.6.2.2 Aside: Item Repetition in Rules

A naïve application of the forward-chaining algorithms of §2 to programs suggests generating a propagator per subgoal and, further, suggests that these could operate independently of each other. However, as we were reminded during the 2013 Dyna prototype's construction, the result will be incorrect when subgoals describe *overlapping* sets of items. Consider the μDyna rule $\{(\mathtt{a}\langle\rangle \hookleftarrow r) \Leftarrow \langle \mathtt{b}\langle x\rangle \mapsto v, \mathtt{b}\langle y\rangle \mapsto w, v \star w \mapsto r\rangle \mid r, v, w, x, y\}$, which contributes to $\mathtt{a}\langle\rangle$ the product $(\star)$ of the values of $\mathtt{b}\langle x\rangle$ and $\mathtt{b}\langle y\rangle$ for each pair $\langle x, y\rangle$ such that both $\mathtt{b}^{/1}$ items have a non-$\textsc{null}$ value. Importantly, the diagonal pairs contribute their product *once*. Backward-chaining as given in this section will obtain the correct result. Consider in particular the case that $\mathtt{b}\langle 1\rangle$ has updated from $\textsc{null}$ to 2. We expect the resulting stream of additional aggregands to contain one $2 \star v$ and one $v \star 2$ for each v the value of some $\mathtt{b}\langle x\rangle$ with $x \neq 1$ and one $2 \star 2$ arising from the case that $x = y = 1$. If, however, the propagators operate independently, there will be *two* copies of the latter result added. Instead, one must carefully control when passenger queries see the world as impacted (or not) by the update being propagated.

In the 2013 prototype, we handled only replacement notifications and so were able to address the diagonal duplication by inserting explicit tests in one of the two propagators (in general, all-but-one of any overlapping propagators). These tests caused the subsequent propagators to skip the diagonal rule answers, leaving only the first propagator to contribute the unique value. More generally, however, when delta notifications are considered, one must arrange that the k^{th} subgoal's propagator invoke subgoals 1 through $k - 1$ such that they obtain the *updated* value and all subgoals after k such that they obtain the *not-yet-updated* value. For details, see Eisner, Goldlust, and Smith [51, §4].

3.7 Related Work

3.7.1 Answer Subsumption

Some Prolog systems [15, 29, 192, 193] have been have been extended with mechanisms collectively known as "Answer Subsumption," which provide for aggregation of (projections of) answers from the solver [169]. 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 181, Example 2], the program

```
:-table p(_,_,lattice(min/3)).
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 $\mathtt{e}^{/2}$ facts, assign each an additive cost of 1, and compute their min-weighted transitive closure. The "`table`" pragma specifies that, for $\mathtt{p}^{/3}$ items, the first two arguments thereof form the index or domain of the resulting table and that, given x and y, the third argument of all $\{\mathtt{p}\langle x, y, z\rangle \mid z\}$ items proven should be aggregated by the $\mathtt{min}^{/3}$ (semi)lattice operator (i.e., minimized). Intuitively, we expect that the result set α from a query $\mathtt{p}\langle \mathcal{H}, \mathcal{H}, \mathcal{H}\rangle$ should then obey a functional dependence between its elements' first two arguments and its third (i.e., $\forall_{\{\langle a\downharpoonleft_1, a\downharpoonleft_2\rangle \mid a \in \alpha\}} \exists!_{w \in \mathcal{H}} \langle a\downharpoonleft_1, a\downharpoonleft_2, w\rangle \in \alpha$).[96]

Only recently was a formal semantics proposed for these extensions [181]. (We now give a compressed summary of these semantics, assuming familiarity with Prolog semantics; readers unfamiliar may wish to review §1.4 first, having possibly skipped it on first reading, or may wish to simply skip to the consequences in the next paragraph.) The described semantics operates on a stratum-by-stratum basis; a **stratum** of a Prolog program is a collection of items which depend upon each other only positively (i.e., without negation or, here, without answer subsumption) [9]. Within each such stratum, the described semantics first find a fixed-point of a modified immediate-consequence operator, which first computes the standard *Prolog* consequences and then *adds* any answers that are a consequence of the lattice join. Having computed that fixed-point, the subsumed answers are all discarded, leaving only the preferred 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 sets, must also have identical subsumed answers.[97] Weighted logic languages use bag semantics, rather than set semantics, to interpret multiple justifications of a particular aggregand for the same key. Equivalently, we can think of weighted logic languages as describing (partial) *functions* for each head, mapping the space of rules (refined by head singleton) and non-trivial rule queries thereof to the resulting values; the bag of values to be aggregated for the head is then the *range bag* of this function.

A second difference, though of less significance, is that 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 Vandenbroucke et al. [181]):

[96] 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 $\{(\mathtt{p}\langle x, y\rangle \hookleftarrow 1) \Leftarrow \langle \mathtt{e}\langle x, y\rangle \mapsto \mathtt{true}\langle\rangle\rangle \mid \cdots\}$ and $\{(\mathtt{p}\langle x, z\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{p}\langle x, y\rangle \mapsto l, \mathtt{p}\langle y, z\rangle \mapsto r, l + r \mapsto v\rangle \mid \cdots\}$. Of course, one of the selling points of μDyna is that it is *possible* to mix values into key positions, and the rule $\{(\mathtt{p}\langle x, y, v\rangle \hookleftarrow \mathtt{true}\langle\rangle) \Leftarrow \langle\rangle \mid \cdots\}p, x, y, v$ will derive $\mathtt{p}^{/3}$ items equivalent to those of the Prolog program from the $\mathtt{p}^{/2}$ items derived above.

[97] It may seem, at first glance, that only *selective* operators could be supported. However, the semantics of Vandenbroucke et al. [181] in fact work for arbitrary semilattices (i.e., *idempotent* operators); the desire to support non-selective operators necessitates the repeated addition of facts computed from semilattice joins during fixed-point computation. See Example 3 therein for more detail.

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

which has the 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 $\{(\mathtt{p}\langle\rangle \hookleftarrow 0) \Leftarrow \langle\rangle\}$ and $\{(\mathtt{p}\langle\rangle \hookleftarrow 1) \Leftarrow \langle \mathtt{p}\langle\rangle \mapsto 0\rangle\}$, with $\mathtt{p}\langle\rangle$ aggregated by `max`, makes the ill-founded recursion more explicit.[98]

Possibly the most striking difference between our work and answer subsumption, however, is that the semantics of answer subsumption are incompletely specified if one allows variables in indexing argument positions. To recover semantics compatible with the view taken in this chapter, 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 Vandenbroucke et al. [181] nor obviously available to typical WAM-based representations of non-ground terms (see §4.1). However, an alternative view, thanks to Dr. David S. Warren (in private communication), is that one can permit the answer-subsumption tables to contain *overlapping* keys whose associated subsumed projections are interpreted as *constraints* on the associated value rather than as the associated value itself. Consider the following program, which is a variant of the above allowing edges to specify their own weights, together with some input assertions:

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

e(1,2,1).  e(2,3,1).  e(_,3,10).
```

This will elicit the following answers from XSB (in potentially different order):

```
p(1,2,1).  p(2,3,1).  p(1,3,2).  p(X,3,10).
```

The last three answers overlap in their index positions. Because tabled subsumption is a *complete* search strategy for this program (that is, if it finishes, it has found all *aggregands*, as we would call them), we know that the resulting set of answers is a complete set of constraints on the values. Essentially, this view takes advantage of the AC-reducer properties of the semilattice operator to eagerly aggregate aggregands routed to *equal* indicies, but otherwise leaves the operations of §3.4.3.6 implicit. Because all permitted operators are idempotent, a dramatically simpler form of §3.4.3.6 could be used, which need only distinguish zero and nonzero multiplicities.

[98] 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.

3.7.2 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 [32, 36, 82, 127] and the predecessor of our current effort, Dyna [51].

3.7.3 Default Logics

Reiter [155] defines a logic for reasoning with defaults in the light of incomplete data. This logic can capture assertions like "most birds fly" and "emus are birds but do not fly" and will deduce, given a proof that Tweety is a bird, in the absence of a proof that Tweety is an emu, that Tweety can fly. As this is a boolean logic, it does not concern itself with multiplicities beyond "zero or non-zero," as truth is an absorbing element of disjunction. Jaeger [102] extends default logics to handle probabilistic reasoning. The BFs of this paper use defaults not as a compensation for incomplete knowledge—indeed, we must have complete knowledge of the contributions for items in order to assure that we obtain the correct answer—but rather as an encoding of structured sparsity without set subtraction.

3.7.4 Lifted Explanations for Problog

Problog [151] is a probabilistic extension of Prolog, assigning probabilistic weights to items. Like μDyna, Problog encounters the need to aggregate over unique outcomes, and could benefit from *counting* rather than *enumerating* outcomes. Nampally and Ramakrishnan [133] consider the construction of "lifted explanation graphs" which use the structure of the Problog program and existential quantification (over finite domains) to compactly summarize the support sets of results for rules. The cardinality of the supports are then extracted by solving recurrences on these structures. As noted in that work, these structures are generalizations of binary decision diagrams and likely can be further extended to encode multi-valued decision diagrams over infinite domains, in which case it may be possible to use them as an implementation of our set theory for (a subset of?) μDyna programs. (For both flavors of decision diagrams, see, e.g., [167].)

3.7.5 Lifted Inference in Statistical Relational Models

Van den Broeck [177] discusses the use of "weighted first-order model counting" as a building block for "lifted" exact (as well as approximate) probabilistic inference. If the sets of our system admit description in first-order logic, then our $\textsc{AnswerFor}(\cdot,\cdot,\cdot)$ oracle can likely be implemented using the same "knowledge compilation" mechanism of that work, which, amusingly, reduces the problem to *weighted circuit solving*.

Chapter 4

Computational Representation of Sets of Terms

> This unification of logic and programming is called the *propositions as types* principle. It is the central organizing principle of the theory of programming languages. Propositions are identified with types, and proofs are identified with programs. A programming technique corresponds to a method of proof; a proof technique corresponds to a method of programming. Viewing types as behavioral specifications of programs, we may see propositions as problem statements whose proofs are solutions that implement the specification.
>
> Robert Harper. *Practical Foundations for Programming Languages.* 2012. [88, ch. 30].

In the last chapter, we assumed *some* runtime representation of the rule sets (ρ), the query sets given as arguments to Compute (κ), the projections of subgoals used as arguments to Lookup ($\Sigma_k \tau_k$), and the answers returning from Lookup ($\sigma \mapsto v$). While the systems we describe are not tightly coupled to their computational implementations, it is, nevertheless, worth spending some time thinking about options available to us.

First, we should note that there is no fundamental computational difficulty in representing a given, finite set of terms: simply record the members. One can design encoding and compression schemes of various complexity, but, in the worst case, the trivial answer remains available. All operations we used in the past chapter (singleton construction; refinement, $\cdot[\cdot/\cdot]$; projection $\cdot\downharpoonleft$; union; subset testing; cardinality counting; and even our oracular functions, RuleToInstr, from §3.3.3, and AnswerFor, from §3.4.1) are trivially implemented by iterating over these finite collections. The challenge emerges when sets are infinite and must, yet, be finitely described. Clearly, we will have to compromise: there are only countably infinitely many *Turing machines*, and so, just by a counting argument, it is obvious that we cannot even describe every element of $\wp\mathbb{Z}$, much less $\wp\mathcal{H}$. The task, then, is to design a finite encoding of *useful* infinite sets and to ensure that this encoding is amenable to the operations we need to perform.

Of course, it is possible to view *a Prolog program P* as a description of a set of terms τ_P: $t \in \tau_P$ iff P proves t. Such a description of a set has an unfortunate drawback: membership of $t \in \tau_P$ is potentially undecidable, as is deciding whether $\tau_P = \varnothing$. In the treatment below, we consider only representations where membership testing is certainly decidable (though possibly expensive).

4.1 Warren's Abstract Machine for Prolog

Prolog specifies logic programs using a syntax of terms built from some set of functors, $\mathcal{F}$, *as well as* a disjoint, countably infinite set of nullary *variable symbols*, $\mathcal{X}$. That is, within a Prolog program, a syntactic term is an element of $\mathcal{H}_{\mathcal{F} \cup \mathcal{X}}$. (In most textual representations, functors begin with a lower-case letter and variables begin with an upper-case letter, as in example 9, in §2.1.2.1.) Such a tree *stands for* the set of terms with the given structure and whose variable leaves (covariantly) range over $\mathcal{H}$. For example, `f(X,Y,g(Y,Z))` (or, strictly, $\mathtt{f}\langle\mathtt{X}\langle\rangle,\mathtt{Y}\langle\rangle,\mathtt{g}\langle\mathtt{Y}\langle\rangle,\mathtt{Z}\langle\rangle\rangle\rangle$, but we will elide the empty tuple of children from variable symbols) stands for $\{\mathtt{f}\langle x,y,\mathtt{g}\langle y,z\rangle\rangle \mid x,y,z \in \mathcal{H}\}$. We use $[\![\cdot]\!]$ in this section to describe this mapping. For present purposes, there are two key features of this representation: ① *all quantification must be over* $\mathcal{H}$, ② sets are otherwise disjunction-free.

One possible manipulation of these objects is by a **substitution**, a mapping from variables to variable-ful terms. The substitution $\zeta = \{\mathtt{X} \mapsto t\}$ (the use of lowercase Greek characters for substitutions is standard) replaces each occurrence of *the variable symbol* `X` within its argument by t. This substitution may also be written as $[t/\mathtt{X}]$, and substitutions are often written in post-position, so $s\zeta = \zeta(s) = s[t/\mathtt{X}]$. The set of variables within a term t is $\{x \in \mathcal{X} \mid \exists_\pi\, x = t{\downharpoonleft_\pi}\}$. A substitution applied to a variable not in a term does not alter that term; formally, for any s, t, and `Y`, $t[s/\mathtt{Y}] = t$ if `Y` not in t.

Given any two variable-ful terms t_1 and t_2, the unification algorithm [159, 121] constructs *the unique* (up to isomorphism) substitution ζ such that $t_1\zeta = t_2\zeta$ and $[\![t_1\zeta]\!] = [\![t_1]\!] \cap [\![t_2]\!]$, assuming this intersection is not empty. In the event of an empty intersection, unification is said to **fail**; no substitution is returned, as it is impossible to explicitly code for the empty set in this syntax. ζ is called the **most general unifier** (MGU) of t_1 and t_2, and, indeed, any other ζ' such that $t_1\zeta' = t_2\zeta'$ is isomorphic to ζ itself or to *further* substitutions in addition to ζ. The unification algorithm is straightforward to describe (and admits efficient implementation). The unification of *equal* t_1 and t_2 is simply the empty substitution, i.e., the identity function. To unify *unequal* t_1 and t_2, first, find any π s.t. $t_1{\downharpoonleft_\pi} \neq t_2{\downharpoonleft_\pi}$, and proceed by cases.

- If these projections are both variables, say, `X` and `Y`, respectively, then add $\zeta = [\mathtt{Y}/\mathtt{X}]$ to the substitution obtained from the unification of $t_1\zeta$ and $t_2\zeta$. That is, replace all `Y` in t_1 with `X` and try again.

- If only the t_1 projection is a variable, say, `X`, and `X` does not occur in $t_2{\downharpoonleft_\pi}$, then add $\zeta = [t_2{\downharpoonleft_\pi}/\mathtt{X}]$ to the unification of $t_1\zeta$ and $t_2\zeta$ (the t_2 variable case is symmetric, naturally). If the **occurs check** above fails (that is, if $\exists_{\pi'}\, t_2{\downharpoonleft_{\pi \mathbin{+\!\!+} \pi'}} = \mathtt{X}$), then unification fails: there are no *finite* trees which contain themselves as a subterm.

- Otherwise, both projections have non-variable outer functors. If these differ (either in symbol or arity), unification fails, as the terms' corresponding sets have empty intersection. Otherwise, there must exist an extension of π which also distinguishes t_1 and t_2; consider that path instead.

If any of the recursive unifications fail, so does the caller; this can be thought of as a consequence of the *non-disjunctive* nature of the sets under study. This algorithm stops either when a point of disagreement has been found or when there are no more differences to be found.

A noteworthy feature of the unification algorithm as given is that variables are scoped more widely than a particular term. That is, the algorithm does not differentiate $\mathtt{X}$ within t_1 and within t_2. Thus, when we ask for the unification of $\mathtt{f}\langle\mathtt{X},1\rangle$ and $\mathtt{f}\langle\mathtt{Y},\mathtt{X}\rangle$, the resulting term is $\mathtt{f}\langle 1,1\rangle$ (the MGU is $[1/\mathtt{X}][1/\mathtt{Y}]$), rather than some description of the set $\mathtt{f}\langle\mathcal{H},\{1\}\rangle$. This is both a blessing and a curse, in practice: it permits one to operate on fragments of variable-ful terms as if one were operating on the whole, but when one requires a true copy of a variable-ful term, i.e., one that will not covary under subsequent substitutions applying to variables within the original, one must explicitly *"freshen"* variables, choosing new identifiers. Some literature refers to this as "capture-avoiding substitution," following studies of the λ-calculus and imagining that all variables within a term are implicitly quantified at its root.[99]

This machinery provides straightforward implementations of the operators needed in the previous chapter, with the sole constraint that projection and refinement paths do not descend *through* quantified structure:[100]

- Projection is preserved by the interpretation mapping, and so *term* projection implements *set* projection: $[\![t{\downarrow_\pi}]\!] = [\![t]\!]{\downarrow_\pi}$. (The caveat about needing to freshen variables applies here if one wishes the projection to lose its association with the larger structure.)

- Refinement (of two terms whose sets of variables are disjoint) is carried out by unification: to compute $u = t[s/\pi]$ let ζ be the most general unifier of $t{\downarrow_\pi}$ and s and take $u = t\zeta$ if ζ exists (otherwise, the result of refinement is empty). Thus, refinement generalizes substitution.

- The empty set cannot be represented explicitly, but, as seen, is easily detected during refinement, the only operation that could produce it.

- Subset testing among non-empty sets (described by terms whose sets of variables are disjoint) is easily done: $[\![t_1]\!] \subseteq [\![t_2]\!]$ if the action of the most general unifier of t_1 and t_2 on t_1 is an invertible renaming of variables. (To see why inverses matter, consider the terms $\mathtt{f}\langle\mathtt{X},\mathtt{Y}\rangle$ and $\mathtt{f}\langle\mathtt{Z},\mathtt{Z}\rangle$. The latter is a *strict subset* of the former, and any ζ which equates $\mathtt{X}$, $\mathtt{Y}$, and $\mathtt{Z}$ is bijective when restricted to the domain $\{\mathtt{Z}\}$ but not when restricted to $\{\mathtt{X},\mathtt{Y}\}$.)

- Cardinality of a set is degenerate: if the set does not contain variables, it has cardinality 1; if it does, it is $|\mathcal{H}|$. However, because Prolog's sole aggregator, OR, is *idempotent*,

[99] Precisely what is meant by "substitution" and "capture-avoidance" is difficult to articulate despite seeming intuitive. One has to track whether a variable in one position "means" the same thing as the same variable in another, even if these are in different terms or, in the λ-calculus, separated by a binder that (re)introduces this variable. The curious reader is pointed to Gabbay and Mathijssen [71].

[100] By descending through quantified structure, we mean asking to compute $\mathtt{X}\langle\rangle{\downarrow_\pi}$ or $(\mathtt{X}\langle\rangle)[\tau/\pi]$ for some $\pi \neq \langle\rangle$. When viewing a Prolog rule cast into the form of a μDyna rule, there is no need for these operations, and, by inspection, unification does not attempt them either.

> Prolog has no need for multiplicity counting: any non-empty bag of TRUE answers produces the same result. As such, the oracular functions of the last chapter, while they may still be difficult to compute, may be removed by use of arithmetic identities.

All told, the family of sets so described has decidable tests for subset testing and cardinality counting. This family is closed under (at least) projection and refinement (and intersection), but is not closed under union (e.g., $\mathtt{f}\langle\mathcal{H}\rangle$ and $\mathtt{g}\langle\mathcal{H}\rangle$ are both representable, but the only representable set that contains their union is $\mathcal{H}$).

The most common Prolog runtime formalism, the Warren Abstract Machine,[101] operates essentially using this variable-ful term representation. Variables are represented by *mutable pointers* in memory; an undo log, called the **trail**, is used to implement "un-unification" to back-track up the search tree (and across rules). The use of the trail obviates the need to *copy* sets between recursive calls of REFINERULESUFFIX, for example.

4.2 Tree Automata

Other classes of sets of trees can be described using a **tree automata** (**TA**) formalism. These encodings permit disjunction and recursion to express quantification over (structured) subsets of $\mathcal{H}$ and, as such, are useful tools for analysing *potential behavior* of a logic program or its solver. The curious reader is again invited to read Comon et al. [35] for a more thorough treatment of the topic, but we summarize the material we need and describe some additional extensions.

4.2.1 Motivation

As mentioned, the WAM is somewhat limited in its expressive power: all quantification is over all of $\mathcal{H}$, and there is no mechanism to capture choice within a single term. The latter implies that all choices must be "lifted up" to the root of the term, where they are amenable to backtracking search. This is a kind of "disjunctive normal form"[102] and, while effective, can carry an exponentially large temporal (possibly spatial as well) cost relative to other families which permit choices elsewhere than the root. We hope to be able to avoid this cost by increasing the expressivity of our runtime set representation.

Separately, many existing treatments of Prolog execution and static analysis, notably including [137], make heavy use of variables in their theory. They work with variable-ful terms and carefully associate these terms with binding contexts, mapping variables to metadata. We were, however, curious to see what could be done without; if we truly believe that variable-ful terms represent sets of terms, how far can we get without needing such a different representation?

[101] Warren [189] introduces the eponymous machine. However, we advise the curious reader to look at Aït-Kaci [8] first; this later work is much more pedagogically motivated than the original research publication(s). Readers especially interested in the topic are encouraged to read Demoen and Nguyen [42], which compares several variants of the WAM design, and Schrijvers and Demoen [161], which describes some insightful engineering specifically for the representation of variables within WAM-like representations.

[102] The origin of the phrase seems lost to the sands of time. Certainly the existence and significance of such a normal form was recognized, though not given its current name, as early as 1968 by Tseitin [174].

Our refinement operator, $\cdot[\cdot/\cdot]$, is a generalization of substitution in the more traditional sense of the WAM and "term-rewriting systems." Substitution takes place only at the leaves of these variable-ful term structures and must be carefully defined on multiple variables at once to preserve covariances. On the other hand, we engineer the structures we manipulate to achieve the same effect with the mechanism of refinement at internal positions. These structures permit us to reason, abstractly, about disjunction and covariance in ways that the traditional augmented terms and maps approaches would struggle to describe.

The trade-off, however, is that for fundamental computational reasons (see, e.g., [58] for a generalization of [35, thm 4.4.7]), our runtime and analysis must be incomplete (with respect to the full formalism). That is, these expressive sets are able to encode undecidable problems into questions we might naturally wish to answer, such as "is a given set empty?"

One of the roles of static analysis (§5) is to discover when a program asks too much of us. We suspect that most programs do not ask for much beyond the WAM's capabilities, and that those programs that do only do so briefly, so in a sense our work towards providing complex runtime types is likely to go un-noticed. However, we suspect that there is more to be gained by the correspondingly richer static analysis framework built using the formalism of automata.

4.2.2 Automata

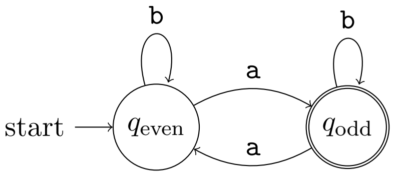
An **automaton** can be considered a *mechanical* description of a collection of objects. That is, they are *procedural* encodings of *indicator* functions $a \in \tau \to \{\top, \bot\}$, which discriminate between $t \in \tau$. $\{t \in \tau \mid a(t) = \top\}$ is called the **language** of the automaton and is said to be **recognized** by the automaton. The automaton family constrains the internal computational form of the function so encoded, and so places limits on the kinds of sets that can be described. An automaton is similar to an agent of §2.3; it consists of a set of possible **configuration states** (often, just **states**), $\mathcal{Q}$, and a transition function, d, which steps from one configuration to another in response to some feature of the input t under test. A central feature of the automaton formalism is that the set and order of observations performed on t is fixed by the automaton *family* and the input, t, and is not subject to alteration by the automaton during operation. That is, the automaton, in general, cannot ask to revisit things it has seen before; it must carry all the state it needs within its configuration. The automaton **accepts** $t \in \tau$ if, after all transitions, its configuration is one of a designated $\mathcal{Q}_{\mathrm{F}} \subseteq \mathcal{Q}$; other elements or subsets of $\mathcal{Q}$ may be endowed with additional significance depending on the kind of automaton under discussion.

4.2.2.1 Review: String Automata

A **deterministic, finite string automaton**, denoted $\langle \sigma, \mathcal{Q}, \mathcal{Q}_{\mathrm{F}}, d, q_0 \rangle$, processes tuples of arbitrary length (also called **strings**) over the alphabet σ (and so acts as a function $\sigma^* \to 2$) by means of a *finite* state space $\mathcal{Q}$, **accepting states** $\mathcal{Q}_{\mathrm{F}} \subseteq \mathcal{Q}$, and transition function $d \in \langle \sigma, \mathcal{Q} \rangle \to \mathcal{Q}$. In order to start the process, a single **initial state** $q_0 \in \mathcal{Q}$ is also designated.

A **run** of a string automaton $\langle \sigma, \mathcal{Q}, \mathcal{Q}_\mathrm{F}, d, q_0 \rangle$ on $\vec{s} \in \sigma^*$ is a tuple which records the execution of the machine, of the form $\langle q_0, s_1, q_1, s_2, q_2, \ldots, s_n, q_n \rangle$ where $q_i = d(s_i, q_{i-1})$. A run is said to be **accepting** if $q_n \in \mathcal{Q}_\mathrm{F}$, in which case $\vec{s}$ is accepted (i.e., is in the language of this automaton, $a(\vec{s}) = \top$). One can, equivalently, think of folding d across the string, starting from q_0, i.e., computing $d(\langle s_k, d(\langle s_{k-1}, \cdots d(\langle s_2, d(\langle s_1, q_0 \rangle) \rangle) \cdots \rangle) \rangle) \in \mathcal{Q}_\mathrm{F}$.

Example 40: Transitions in string automata are often rendered as edges labeled by elements of σ within a graph whose nodes are states from $\mathcal{Q}$. For example, the automaton rendered as



has $\mathcal{Q} = \{q_\mathrm{even}, q_\mathrm{odd}\}$ and accepts strings formed from $\sigma = \{\mathtt{a}, \mathtt{b}\}$ which contain an odd number of a's. In this rendering style, the doubly-circled node(s) are elements of $\mathcal{Q}_\mathrm{F}$ and q_0 is indicated by the "start" label. ◊

The fact that $\mathcal{Q}$ is finite has as an immediate consequence a "Pumping Lemma" [first published as 149, Lemma 8] which constrains the kinds of sets that can be recognized by machines of this sort. Any finite set can be recognized, but infinite languages must contain tuples sufficiently long to force the machine to revisit a configuration, and so must also contain "pumped" versions of this tuple. That is, for any accepted tuple t whose length is greater than $|\mathcal{Q}|$ there must exist a, l, and z such that $\mathrm{tlen}(l) \neq 0$ and $t = a + l + z$ such that $a + z$ and $a + l + l + z$ (and indeed, any $a + l + l + \cdots + l + l + z$) are also accepted. a must advance the machine from q_0 to a revisited state q^*, l walks a cyclic path through $\mathcal{Q}$ from q^* to itself, and z walks from q^* to some state $q_\mathrm{f} \in \mathcal{Q}_\mathrm{F}$.[103] This provides simple proofs that certain languages are *outside* the class of so-called **regular** languages (i.e., those that can be accepted by machines of this form). For example, there is no finite string automaton recognizing the set of "all palindromic tuples," $\{\langle s_1, \ldots, s_k \rangle \mid \forall_i\, s_i = s_{k-i+1} \in \sigma\}$ when $|\sigma| > 1$ (otherwise this is trivial).

4.2.2.2 Nondeterminism

The automata described above are **deterministic**: the transition must select a *single* successor configuration for each state and input. A **nondeterministic** automaton instead produces a *set* of possible successor configurations; $t \in \tau$ is recognized if *there exists* a choice of successor states at each step of the computation that results in an accepting state at the end of the computation.

Example 41: A nondeterministic finite string automaton has, as its transition function d, an element of $\langle \sigma, \mathcal{Q} \rangle \to \wp\mathcal{Q}$ (which differs from the type of a deterministic machine's transition function in having $\wp$). One now asks whether *there exists* a run $\langle q_0, s_1, q_1, s_2, q_2, \ldots, s_n, q_n \rangle$ in which $q_i \in d(s_i, q_{i-1})$. For string automata, nondeterminism is not essential: a search

[103] q^* is guaranteed to exist by the pigeon-hole principle. a may be $\langle\rangle$ if $q_0 = q^*$, and z may be $\langle\rangle$ if $q^* \in \mathcal{Q}_\mathrm{F}$.

procedure for a nondeterministic automaton with state space $\mathcal{Q}$ can be implemented by a *deterministic* finite string automaton with a state space of $\wp\mathcal{Q}$ [149, Ch. 2].

In the graphical depiction of string automata, nondeterminism permits multiple out-edges with the same label, each targeting an element of the set of states into which the automaton will transition. One traditionally also permits so-called "ϵ arcs" which do not consume a symbol; if $q_0 \overset{\epsilon}{\mapsto} q_1$, then entering state q_0 implies entering state q_1 too, i.e., $\forall_{q,s}\, q_0 \in d(s,q) \Rightarrow q_1 \in d(s,q)$. $\Diamond$

4.2.3 Regular Tree Automata

We begin our discussion of tree automata with a review of **regular tree automata**, a direct generalization of regular *string* automata to the case of ranked trees. That is, their languages $\{t \subseteq \mathcal{H}_\mathcal{F} \mid a(t) = \top\}$ are sets of trees.

A **bottom-up**, deterministic regular tree automaton over the symbols $\mathcal{F}$, denoted $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_\mathrm{F}, d\rangle$ is a $\mathcal{F}$-algebra (§2.1.1.1) with its state set $\mathcal{Q}$ as its carrier and with interpretation function d. That is, $d \in \Pi_{\mathtt{f}/^n \in \mathcal{F}}(\mathcal{Q}^n \to \mathcal{Q})$.[104] As before, $\mathcal{Q}_\mathrm{F} \subseteq \mathcal{Q}$ is the set of accepting states. A run of such an automaton on input tree t is a function r from each position π within the tree to a state label; r must respect the interpretation function d, i.e., if $t{\downarrow_\pi}$ has root symbol $\mathtt{f}/^n$, then $r(\pi) = d(\mathtt{f}/^n)(\langle r(\pi.1), \ldots, r(\pi.n)\rangle)$.[105] As before, a run on t is accepting if $r(t) \in \mathcal{Q}_\mathrm{F}$. These automata are called "bottom-up" because they admit, like algebras, a procedure for interpretation which begins at the leaves and works towards the root.[106] A nondeterministic bottom-up tree automaton is defined as might be expected, with d yielding $\wp\mathcal{Q}$ and a tree being in the language if *there exists* an accepting run, i.e., where every position is mapped by r such that $r(\pi) \in d(\mathtt{f}/^n)(\langle r(\pi.1), \ldots, r(\pi.n)\rangle)$, where, again, $t{\downarrow_\pi}$ has root symbol $\mathtt{f}/^n$.

Example 42: By analogy to string automata, the pictorial analogy of a tree automaton's transition function should be a *B-hypergraph* (recall §2.1.2.2) with a linear ordering on each edge's tails. For whatever reason, this presentation has apparently never gained much traction, and a TA is presented as a series of rules, corresponding to different possible inputs. Thus, by writing the six rules[107]

[104] Recalling footnote 34 (in §2.1.1.1), one may prefer to think isomorphically, with $d \in (\Sigma_{\mathtt{f}/^n \in \mathcal{F}}\, \mathcal{Q}^n) \to \mathcal{Q}$. Such a formulation is closer to the string automaton case, where transition functions were in $\langle\sigma, \mathcal{Q}\rangle \to \mathcal{Q}$, and opens the door to a more general notion of deterministic automata as algebras $f(\mathcal{Q}) \to \mathcal{Q}$ for some categorical functor f. A great deal of very interesting work has gone into studying these so-called "structured recursion schemes" [95].

[105] For *deterministic* automata, in which equal trees will always have equal runs, r is equivalent to a function of the subterms themselves, rather than paths, i.e., $r(\mathtt{f}\langle t_1, \ldots, t_n\rangle) = d(\mathtt{f}/^n)(\langle r(t_1), \ldots, r(t_n)\rangle)$. Recalling example 6 (in §2.1.1.1), we see that such r are an *algebra homomorphism* from the initial algebra on $\mathcal{H}_\mathcal{F}$ to the algebra defined by d on $\mathcal{Q}$.

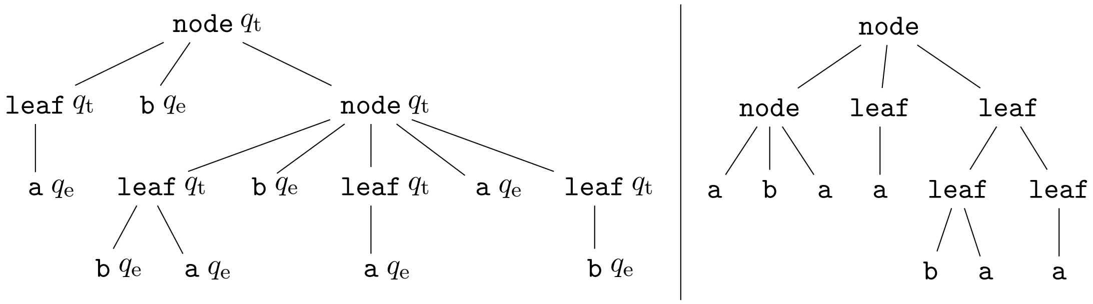
[106] And because, again, computer scientists invariably draw our trees upside-down. However, unlike §2.1.2, here, "bottom-up" reasoning actually proceeds upwards from the bottom of the drawing.

[107] The traditional notation used in most documents on the topic is to use → rather than our ↦. However, because we so heavily use → within the type of functions, we consider ↦ to be more appropriate. Separately, it is apparent that these rules do not define a total function; the case $\mathtt{leaf}\langle q_\mathrm{t}\rangle$ is, for example, unhandled. This is a common short-hand taken in presentation; it is possible to **complete** the definition by adding both a "dead state" $q_{☠} \in \mathcal{Q} \setminus \mathcal{Q}_\mathrm{F}$, which is the target of missing cases, and rules which send any input involving $q_{☠}$ to $q_{☠}$ [see 35, Thm 1.1.7].

$$\begin{array}{r|r|l} \texttt{node}\langle q_{\mathrm{t}}, q_{\mathrm{e}}, q_{\mathrm{t}}\rangle \mapsto q_{\mathrm{t}} & \texttt{leaf}\langle q_{\mathrm{e}}\rangle \mapsto q_{\mathrm{t}} & \texttt{a}\langle\rangle \mapsto q_{\mathrm{e}} \\ \texttt{node}\langle q_{\mathrm{t}}, q_{\mathrm{e}}, q_{\mathrm{t}}, q_{\mathrm{e}}, q_{\mathrm{t}}\rangle \mapsto q_{\mathrm{t}} & \texttt{leaf}\langle q_{\mathrm{e}}, q_{\mathrm{e}}\rangle \mapsto q_{\mathrm{t}} & \texttt{b}\langle\rangle \mapsto q_{\mathrm{e}} \end{array}$$

and taking $\mathcal{F} = \{\texttt{a}^{/0}, \texttt{b}^{/0}, \texttt{leaf}^{/1}, \texttt{leaf}^{/2}, \texttt{node}^{/3}, \texttt{node}^{/5}\}$, $\mathcal{Q} = \{q_{\mathrm{t}}, q_{\mathrm{e}}\}$, $\mathcal{Q}_{\mathrm{F}} = \{q_{\mathrm{t}}\}$, we encode essentially the Algebraic Data Type description of a 2-3 tree [7] (though without any height or ordering constraints, so the trees accepted by this automaton are a superset of valid 2-3 trees). This automaton is easily seen to be *deterministic*.

Consider these two trees from the Herbrand universe built from the above $\mathcal{F}$:



The tree on the left is accepted by this automaton (and is annotated with its run, with the states for each path shown to the right of the node symbol). The tree on the right is not: at several positions in the tree, there is no rule which could possibly fire, given the (necessarily unique) analysis of the immediate children of the root functor at that position. ◊

Because trees, unlike strings, fundamentally differ when viewed in different directions, we might ponder a **top-down** deterministic tree automaton, $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_{\mathrm{F}}, q_0, d'\rangle$ with $q_0 \in \mathcal{Q}$ and $d' \in \Pi_{\texttt{f}^{/n} \in \mathcal{F}}(\mathcal{Q} \to \mathcal{Q}^n)$.[108] A run of a top-down automaton is still a function r from each position of a tree t to a state label, but with different consistency conditions: the root must be labeled q_0 and for every position π (including the root) with $r(\pi) = q$ and $t\downharpoonleft_\pi$ having root symbol $\texttt{f}^{/n}$, if $d'(\texttt{f}^{/n})(q) = \langle q_1, \ldots, q_n\rangle$, then $\forall_{i \in \mathbb{N}_1^n}\, r(\pi.i) = q_i$. The run r is accepting if r sends *all* leaves to $\mathcal{Q}_{\mathrm{F}}$.

A nondeterministic top-down tree automaton yields multiple choices of *joint labeling* of children at each transition, rather than a set of possible states for each child. That is, $d' \in \Pi_{\texttt{f}^{/n} \in \mathcal{F}}(\mathcal{Q} \to \wp(\mathcal{Q}^n))$, rather than $\Pi_{\texttt{f}^{/n} \in \mathcal{F}}(\mathcal{Q} \to (\wp\mathcal{Q})^n)$. Again, the automaton accepts the tree if *there exists* a run with labels drawn from the offerings of the transition function at every step. The automaton of example 42 above is also top-down deterministic.

A central result for regular tree automata is that only three of the four cases are equal in recognition capability. For bottom-up regular tree automata, nondeterminism is not essential [35, theorem 1.1.9], just as it was not for string automata. The same class of languages is recognizable by a *nondeterministic* top-down automaton [35, theorem 1.6.1], but top-down *deterministic* automata can recognize only a subset of the languages recognizable by the other three classes [35, prop 1.6.2].

[108] This is, notably, not the type of an unfold or stream, which would be $\mathcal{Q} \to \Sigma_{\texttt{f}^{/n} \in \mathcal{F}}\, \mathcal{Q}^n$. That is, because top-down tree automata continue to *consume* a tree, rather than generate it, they are not the categorical dual of bottom-up automata. In fact, we are unaware of a categorical reading of top-down automata.

4.2.4 With Local Constraints

One is often interested in knowing whether two different positions within a tree have equal projection; one may, in fact, wish to describe sets of trees with properties of this form. One way of describing such sets would be to equip a (bottom-up) automaton's transition function with the ability to test equalities of subterms. A robust study of such automata was undertaken by Mongy-Steen [128], and introduced the moniker RATEG ("*Reconnaisseurs Avec Test d'EGalité*") for such automata.[109] The complement-closure of RATEG, called AWEDC ("Automata with Equality and Disequality Constraints") permits tests of equality and disequality of children [35, ch. 4]. While the closure properties of RATEG and AWEDC are attractive, their interesting decision problems, including whether an automaton recognizes any tree at all, are generally undecidable. Subsequently, a wide variety of sub-classes of AWEDC has emerged, with varying expressive power, closure properties, and decidability of decision problems; notable examples include "AWCBB" ("constraints between brothers"; also called "$\mathrm{REC}_{\neq}$") which permits testing of (dis)equality only between *immediate* subterms [20], "reduction automata" [41], "equational constraints" [101], and several classes pertaining to homomorphic equalities [78]. An excellent overview of AWEDC, AWCBB, and reduction automata can be found in chapter 4 of [35].

A transition within AWEDC is now gated by a set of constraint path-pairs. Thus, while within regular TAs, a transition is represented by a rule $\mathtt{f}\langle q_1, \ldots, q_n\rangle \mapsto q_0$ (for states $q_i \in \mathcal{Q}$ and $\mathtt{f}^{/n} \in \mathcal{F}$), an AWEDC rule is of the form

$$\mathtt{f}\langle q_1, \ldots, q_n\rangle \xmapsto{\pi_{\mathrm{e},1}=\pi'_{\mathrm{e},1},\ \cdots,\ \pi_{\mathrm{e},j}=\pi'_{\mathrm{e},j},\ \pi_{\mathrm{d},1}\neq\pi'_{\mathrm{d},1},\ \cdots,\ \pi_{\mathrm{d},k}\neq\pi'_{\mathrm{d},k}} q_0.$$

Here, we have j equality constraints, each between the subterms at $\pi_{\mathrm{e},i}$ and $\pi'_{\mathrm{e},i}$, and, similarly, k disequality constraints. The various paths here are interpreted relative to the subterm (of the input tree) being considered for the transition; in order for a subterm $t = \mathtt{f}\langle t_1, \ldots, t_n\rangle$ to be labeled q_0 by the above transition, not only must each t_i be labeled by q_i, it must be that $\forall_i\, t{\downarrow}_{\pi_{\mathrm{e},i}} = t{\downarrow}_{\pi'_{\mathrm{e},i}}$ and $\forall_i\, t{\downarrow}_{\pi_{\mathrm{d},i}} \neq t{\downarrow}_{\pi'_{\mathrm{d},i}}$. This justifies the phrase "local constraints" to describe these gates on transitions.

Example 43: Consider an automaton defined by the three rules $\mathtt{nil}\langle\rangle \mapsto q_{\mathrm{n}}$, $\mathtt{cons}\langle q_{\mathrm{e}}, q_{\mathrm{n}}\rangle \mapsto q_{\mathrm{l}}$, and $\mathtt{cons}\langle q_{\mathrm{e}}, q_{\mathrm{l}}\rangle \xmapsto{1=2.1} q_{\mathrm{l}}$, with q_{l} being the unique accepting state. This automaton recognizes lists $\mathtt{cons}\langle x, \mathtt{cons}\langle x, \cdot\mathtt{cons}\langle x, \mathtt{nil}\langle\rangle\rangle\cdot\rangle\rangle$ in which all elements are *equal.* During analysis by the automaton, all $\mathtt{cons}^{/2}$ nodes, other than the bottom-most, can be seen as being in state q_{l} if and only if their associated element (at relative position 1) is equal to the next element in the list (at relative position 2.1). Because this automaton requires an equality constraint on a *recursive* rule (here, with q_{l} on both the left and right of $\mapsto$), it is excluded from many sub-classes of AWEDC. If we were to replace the third rule with $\mathtt{cons}\langle q_{\mathrm{e}}, q_{\mathrm{l}}\rangle \xmapsto{1\neq 2.1} q_{\mathrm{l}}$, then the resulting automaton accepts lists in which no two adjacent elements are equal. ◊

[109] The definition of RATEG in fact is more of a generative model, proceeding top-down and allowing duplications by, as far as this author can discern, repeated application of local tree rewrites. The bottom-up automaton view came later and is now sufficiently prevalent that one might call it standard.

Example 44: AWEDC can also recognize the language of "lists of equal pairs,"

$$\{\texttt{cons}\langle\texttt{p}\langle t_1, t_1\rangle, \texttt{cons}\langle\texttt{p}\langle t_2, t_2\rangle, \cdots\texttt{cons}\langle\texttt{p}\langle t_n, t_n\rangle, \texttt{nil}\langle\rangle\rangle\cdots\rangle\rangle \mid n \in \mathbb{N}, \forall_i\, t_i \in \tau\}$$

(and similarly, the language "lists of pairs of unequal elements"). The pair would be recognized by $\texttt{p}\langle q_\mathrm{e}, q_\mathrm{e}\rangle \xmapsto{1=2} q_\mathrm{p}$; the list is built by the recursive rule $\texttt{cons}\langle q_\mathrm{p}, q_\mathrm{l}\rangle \mapsto q_\mathrm{l}$. This, quite simple, language is also within the expressive power of many of the sub-classes, including AWCBB, reduction, and homomorphic-equality automata. ◊

Local constraints allow for the creation of *unboundedly many*, *arbitrarily-large* equality classes of positions within recognized trees. The first (equality) example above contains exactly one equivalence class, which equates arbitrarily many positions within the tree (one for each $\texttt{cons}^{/2}$ node). On the other hand, the second (equality) example above demonstrates that there may be any number of equality classes.

Determinism retains its meaning on AWEDC and sub-classes; the languages recognized by nondeterministic AWEDC are a proper superset of those recognized by deterministic AWEDC. Recall that a deterministic TA has the additional property that every tree has at most one run. Thus, within the analysis carried out by a deterministic automaton, two sub-terms are equal only if they are analysed into the same state at their roots. As a consequence, some decision predicates (notably, emptiness of an automaton's language) are straightforward. Nondeterminism, on the other hand, is full of subtleties for AWEDC and sub-classes. With local constraints and nondeterminism, it is quite straightforward to define automata which are empty iff some Turing machine does not halt (first shown as Thm. 4.4.7 of Comon et al. [35]; see Filardo [58] for increased applicability of the proof).

Further, the local nature of constraints makes projection (from a nondeterministic automaton) non-trivial. In the case of unconstrained TAs, projection is as simple as walking the rules from the accepting states to find the set of states which accept the projected subterms. However, during traversal of rules of an AWEDC, one now accumulates constraints which may mean that not all trees accepted by the states at the end of the projection path are actually within the set projection. (Though they are certainly a superset.)

4.2.5 With Global Constraints

Another option, rather than adjusting the transition function to permit the use of local tests, is to redefine what it means for a run to be accepting. This opens the door for *global* constraints. Introduced by Filiot, Talbot, and Tison [64] (with further results in Filiot, Talbot, and Tison [65]), the first class of this sort was "Tree Automata with Global Equalities and Disequalities" (TAGED). A TAGED automaton $\langle\sigma, \mathcal{Q}, \mathcal{Q}_\mathrm{F}, d, =_\mathrm{Q}, \neq_\mathrm{Q}\rangle$ is equipped, in addition to the usual pieces, with two relationships, $=_\mathrm{Q}$ and $\neq_\mathrm{Q}$ on pairs of states. A run r on the tree t is defined as accepted exactly when, for every pair of positions of t, π and π', $r(\pi) =_\mathrm{Q} r(\pi') \Rightarrow t{\downarrow_\pi} = t{\downarrow_{\pi'}}$ and $r(\pi) \neq_\mathrm{Q} r(\pi') \Rightarrow t{\downarrow_\pi} \neq t{\downarrow_{\pi'}}$.

The languages recognizable by TAGED is neither a superset nor a subset of those recognizable by AWEDC. In the interesction of the two sets of languages we find all regular tree languages (unsurprisingly) as well as some non-regular tree languages.

The fragment of TAGED automata without disequality constraints (i.e., those for which $\neq_\mathrm{Q}$ is empty) is equivalent in recognition power to a class known as "Rigid Tree

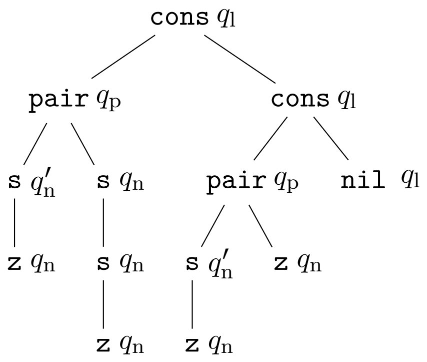
Automata" (RTA) [100], in which, essentially, $=_{\mathrm{Q}}$ is constrained to be a subset of the diagonal relationship on $\mathcal{Q}$. That is, each occurrence of a so-called "rigid state" in $\mathcal{Q}_{\mathrm{R}} \subseteq \mathcal{Q}$ within a run must label the root of equal subterms (so $=_{\mathrm{Q}}$ is $\{\langle q, q\rangle \mid q \in \mathcal{Q}_{\mathrm{R}}\}$). Inclusion of RTA into TAGED is immediate, and inclusion in the other direction proceeds by a variant of the typical powerset construction. (Thus, a rigid automaton recognizing the same language as the TAGED automaton A may be exponentially larger in description than A.)

The emptiness problem for an RTA automaton is, despite the presence of equality constraints, essentially identical to that of regular TAs [100, §6.2]; for TAGED automata it is more expensive [see 65, theorems 1, 2, and 4] but in all cases decidable [17]. Universality (i.e., the acceptance of all trees), and therefore pairwise subset and equality testing, are undecidable for nondeterministic RTA.

As part of our effort to design automata for Dyna's use, we developed "Rigid Tree Automata With Isolation," which allows the automata to sever equality constraints between different regions of the tree [62]. This family can recognize a strict superset of RTA-recognizable languages and has a polynomial time emptiness test, but is not closed under intersection. The motivation, in particular, was considering the closure of RTA under a Kleene-star-like operator (recall footnote 22, in §1.3). RTA can recognize, for example, pairs of equal trees, $\{\mathtt{f}\langle x, x\rangle \mid x\}$, and being supersets of regular automata, can recognize regular lists. However, they cannot recognize lists whose elements are equal pairs: given a sufficiently long list, eventually, any putative RTA for this set would be forced to reuse a rigid state in violation of the definition. Adding isolation allows the resulting family to express this set.

Example 45: Consider a rigid tree automaton with $\mathcal{Q} = \{q_{\mathrm{l}}, q_{\mathrm{p}}, q_{\mathrm{n}}, q'_{\mathrm{n}}\}$, $\mathcal{Q}_{\mathrm{F}} = \{q_{\mathrm{l}}\}$, q'_{n} the sole rigid state, and the transition rules are shown below on the left.

$$\begin{array}{r|r}
\mathtt{z}\langle\rangle \longmapsto q_{\mathrm{n}} & \mathtt{pair}\langle q'_{\mathrm{n}}, q_{\mathrm{n}}\rangle \mapsto q_{\mathrm{p}} \\
\mathtt{s}\langle q_{\mathrm{n}}\rangle \longmapsto q_{\mathrm{n}} & \mathtt{nil}\langle\rangle \mapsto q_{\mathrm{l}} \\
q_{\mathrm{n}} \overset{\epsilon}{\longmapsto} q'_{\mathrm{n}} & \mathtt{cons}\langle q_{\mathrm{p}}, q_{\mathrm{l}}\rangle \mapsto q_{\mathrm{l}}
\end{array}$$



This automaton accepts lists of pairs of Peano naturals, with the constraint that all first elements of those pairs are equal. On the right, above, is an accepted tree, labeled with its accepting run. We can see that q'_{n} is consistently assigned the term $\mathtt{s}\langle\mathtt{z}\langle\rangle\rangle$ as required by RTA, while q_{l}, q_{p}, and q_{n} are free to vary at each occurrence. It happens that this language is recognizable both by AWEDC (but not, for example, by the so-called "reduction automata" subset thereof) and TAGED. ◊

Hybrids Of course, hybrid models have been considered; Barguñó et al. [17] considers hybridizing AWCBB and TAGED and finds that emptiness is decidable and that it remains decidable when sporting global *arithmetic* constraints on the number of occurrences of states or number of equivalence classes of trees reaching some state.

4.2.6 With State Variables for Prenex Polymorphism

Both the algorithms given so far and the analysis yet to come rely centrally on propagation of constraints within sets. In particular, properties of the form $\forall_{t \in \tau}\, \phi(t)$, where ϕ makes no reference to τ, are subset-closed, i.e., $\forall\, \sigma \subseteq \tau\; \forall_{s \in \sigma}\, \phi(s)$. The equality constraints we so heavily relied upon in our earlier work (e.g., $\forall_{t \in \tau}\, t{\downharpoonleft_1} = t{\downharpoonleft_2}$), albeit typically covertly expressed by repeated use of a variable within a set comprehension (e.g., $\{\langle x, x\rangle \mid x\}$), are of this form. While $\tau[\tau'/1]{\downharpoonleft_1} \subseteq \tau'$ holds for all τ and τ', given the above universal equality constraint on τ, we also have that $\tau[\tau'/1]{\downharpoonleft_2} \subseteq \tau'$. This is what allowed answering one subgoal in a rule body within, e.g., block 3.1, to narrow subsequent query projections. However, while the equality constraints of AWEDC or TAGED do give rise to this kind of constraint propagation, there are other sources of such propagation that we might wish to use, especially within static analysis. To capture these kinds of *type-level* covariance relations, we might broaden the class of objects we are willing to consider to include things beyond merely sets of terms.

Consider attempting to describe the set of pairs of finite lists which are reverses of each other and within which all elements are members of some set τ, i.e.,

$$v(\tau) = \{\mathtt{v}\langle \mathtt{c}\langle x_1, \mathtt{c}\langle x_2, \cdots \mathtt{n}\langle\rangle\rangle \cdots\rangle, \mathtt{c}\langle x_n, \mathtt{c}\langle x_{n-1}, \cdots \mathtt{n}\langle\rangle\rangle \cdots\rangle\rangle \mid n \in \mathbb{N}, \forall_i\, x_i \in \tau\}.^{110}$$

Due to the unboundedly many equalities (one for each list element), any set $v(\tau)$ is outside the reach of TAGED automata, when $|\tau| > 1$. A little less clearly, perhaps, these sets are also beyond the reach of AWEDC. There can be only finitely many rules in any purported $v(\tau)$-recognizing automaton $\mathfrak{A}$, and in particular only finitely many rules that could recognize the root, $\mathtt{v}^{/2}$, term. Thus, there are only finitely many sets of constraints that can be tested and there must be a finite path of longest length $k \in \mathbb{N}$, and so lists with sufficiently many elements must have some elements not compared for equality, violating the definition of v.[111]

Even if we drop the "reverse" requirement and aim to describe the superset of α containing pairs of lists with equal sets of elements (even without constraining length), or the yet larger superset containing pairs of lists where the set of elements of the first list (say) is a superset of the set of elements of the second, we find that we are *still* outside the reach of AWEDC or TAGED. Indeed, it is not even clear how to attempt to express the notion of "sets of elements" to these automata: while they can constrain individual positions, their only tool for describing sets of trees is, unsurprisingly, an automaton state, which does not

[110] The function $v(\cdot)$ has a convenient feature: while it is, indeed, an element of $\wp\mathcal{H} \to \wp\mathcal{H}$, it has a *purely syntactic* definition: there is no analysis of its argument, which is merely used as-is in its definition. Several constructions within the upcoming static analyses will have this behavior.

[111] "Sufficiently many" may be larger than k, as the automaton may attempt to encode information about elements' identities or equalities within the list in the state labels. However, taking any $n \in \mathbb{N}$ such that $n \geq \log_{|\tau|}(|\mathcal{Q}|+1)$, there must exist two distinct lists of length $2k+n$ with elements $\langle p_1, \ldots, p_k, c_1, \ldots, c_n, s_1, \ldots, s_k\rangle$ and $\langle p_1, \ldots, p_k, c'_1, \ldots, c'_n, s_1, \ldots, s_k\rangle$, differing only in the middle, which (nondeterministically can) take the machine into identical states at their roots (i.e., just under the $\mathtt{v}^{/2}$ node at the top of the tree). Thus, given an accepting run with one of these lists as (say) the left list in the pair, we can produce an accepting run using the other, and so the purported recognizer of $v(\tau)$ is not, in fact, such a recognizer.

immediately appear useful here. Notably, there is no notion of coverage of a state by the trees labeled thereby within a run.

We *can*, however, describe a function which takes a machine (of some family) describing some set τ to the machine (of the same family) describing pairs of lists whose elements are all from τ, i.e.,

$$v'(\tau) = \{\mathtt{v}\langle \mathtt{c}\langle x_1, \mathtt{c}\langle x_2, \cdots \mathtt{n}\langle\rangle\rangle\cdots\rangle, \mathtt{c}\langle y_1, \mathtt{c}\langle y_2, \cdots \mathtt{n}\langle\rangle\rangle\cdots\rangle\rangle \mid n \in \mathbb{N}, \forall_i\, x_i \in \tau, \forall_i\, y_i \in \tau\} \supseteq v(\tau).$$

$v'(\tau)$ is a strict superset of all of the alternatives offered above, as we no longer have any relationship between the elements of the two lists or even their lengths, but nevertheless, we have the joint upper bound τ on each set of elements. This automorphism has a kind of "template" structure: it specifies the rules for recognizing the $\mathtt{p}^{/2}$ root node and the list "spines" (the $\mathtt{c}^{/2}$ and $\mathtt{n}^{/0}$ nodes), but uses the root state(s) of the automaton provided as its input (as well as all of that automaton's rules) to describe the elements of the lists. Unfortunately, $v'(\tau)$ does not lend itself to constraint propagation by refinement: $\alpha'(\tau)[\beta/1]$ leaves the $\pi = 2$ projection unaltered. However, v' itself (i.e., the function) can be described using an augmented automaton formalism with *state variables* (i.e., variables that range over *states of the automaton*), and it is straightforward to take such a representation to produce either $v'(\tau)$ for some TA (possibly with constraints) τ or $v' \circ w$ for some other such augmented automaton (i.e., with its own parameter(s)). See Xiao, Sabry, and Ariola [191, §5] for details.[112]

It is worth spending a moment to discuss the "first-order" nature of this kind of parameterization of an automaton. As noted, these automata with state variables describe functions from types to types; in traditional programming literature they would be described as parametrically polymorphic types,[113] such as Haskell's "`Maybe` α" type, "`data Maybe` α `= Nothing | Just` α" (or as we might write it, $\{\alpha \mapsto \mathtt{nothing}\langle\rangle \cup \mathtt{just}\langle\alpha\rangle \mid \alpha\}$). Because all parameters are given up front, these types are said to be "prenex polymorphic" or "prenex quantified."[114] That is, by contrast, we could imagine yet-further-augmented constructions which allowed parameters to range over set automorphisms as well. While these so-called higher-ranked types are not very useful for *data* they are potentially useful, and increasingly expressive, in the presence of functions, since polymorphism can be used to derive so-called "free theorems" about functions ascribing to polymorphic types [185].[115]

[112] In the automaton construction given for computing a TA for $v'(\tau)$ given one for τ and the parametric TA for v', in general, the result will be a nondeterministic automaton. If both the template TA for v' and that for the parameter τ contain a rule with common left hand side symbol, then some trees will be analysed by different states in the result.

[113] At this point, our insistence that types are just sets of terms mires us in terminological difficulties. The use, in the broader programming language community, of the phrase "parametric type" is perfectly sanctioned: parametric types are those objects described by a *type grammar* which has been extended to permit parameters (i.e., type variables). Here, rather than enlarging some grammar, were we to desire "polymorphic types," we would have to add n-ary type automorphisms to our universe of types. This poses problems, however, as there is no purely set-theoretic model of the second-order typed λ-calculus [157] (at least, without restricting to *constructive* set theories [147]). In §5.2.3.1 we will consider an alternative, approximating formalism.

[114] The exact origin of the term "prenex" is uncertain. Possibly it stems from Latin "praenexus" ("bound in front") and entered modern usage through Hilbert and Bernays [93].

[115] Parametric polymorphism is often denoted by a universal quantifier, as in $\forall_\alpha(\alpha \to \alpha)$. This is something

4.2.7 Primitive Base-cases

Xiao, Sabry, and Ariola [191] also introduces two practical notions for extending tree automata to describe real-world use-cases.

① Real-world programming languages typically have types for infinite collections, such as the integers.[116] Because the elements of these **primitive** types are *nullary functors*, they can readily be handled via "built-in" states, which are, conceptually, backed by infinitely many *non-recursive* rules. For example, we might define q_{int} for $\mathbb{Z}$ and allow this state to be referenced on the left within other transition rules. Another popular primitive set is $\mathcal{H}$ itself. As there are potentially unboundedly many functors in $\mathcal{F}$, it makes sense to bundle all terms together in the case where structure is not differentiated by the automaton. (Any automaton with such a universal state is, by definition, nondeterministic.)

② Tree automata as presented so far work well to describe sets built up from their base cases. While general complementation is, typically, out of reach, it is nevertheless viable and occasionally useful to describe a **co-finite** set, by describing an infinite set (possibly a *primitive* infinite set as above) less a *finite* set of exclusions.

These extensions are orthogonal to the extensions described in previous sections and generally theoretically benign (i.e., may be readily added with only small changes to proofs or algorithms).

4.3 Automata as Circuits

Automata, though traditionally understood as descriptions of a process for deciding some property (i.e., set membership) of an object under study, are amenable to an *algebraic* interpretation as well as an interpretation *as arithmetic circuits.*

4.3.1 Aside: An Algebraic View of Tree Automata

We can classify automata by viewing them through the lens of an (categorical) algebra (recall §2.1.1.1 and, in particular, example 6 therein). We recast transition functions as (total) functions of type $(\Sigma_{\mathtt{f}/n \in \mathcal{F}} X^n) \to X$, for some carrier set X. If the automaton has

of a pun on the logical universal quantifier of the same name; in System F [76, 158], it represents the *type-level* function $\{\alpha \mapsto (\alpha \to \alpha) \mid \alpha\}$. (This function is also purely syntactic: it simply copies its argument and pushes it into place within some fixed syntax.) Most type systems are "predicative," in that the domain over which α is quantified in the above set excludes quantified types, or, at least, permits only smaller quantified types, using a typical inductive construction. Example "free theorems" arising from prenex polymorphism include the observations that there is *exactly one* terminating function in $\forall_\alpha(\alpha \to \alpha)$, namely the (polymorphic) identity function $\lambda_x x$, that there are *exactly two* terminating functions in $\forall_\alpha(\alpha \to \alpha \to \alpha)$, namely $\lambda_x \lambda_y x$ and $\lambda_x \lambda_y y$, and that the terminating functions of type $\forall_\alpha((\alpha \to \alpha) \to \alpha \to \alpha)$ are in bijective correspondence with $\mathbb{N}$. For an example involving non-prenex polymorphism, contrast this latter type and $\forall_\alpha((\forall_\beta(\beta \to \beta)) \to \alpha \to \alpha)$. The former function can make very few assertions about its given function argument, while the latter can prove that its function argument is a (possibly partial) identity function.

[116] Some programming languages draw a distinction between (signed) machine words of finite bit-width, often erroneously called "integers," and "big integers" of unbounded bit-width. Here, we mean the latter.

constraints, then X must not only carry the automaton's (finitely many) states Q but also *sufficient statistics* of the tree for those tests, so that $\Sigma_{\mathtt{f}/n \in \mathcal{F}} X$ carries enough information for the transition function to operate properly. This type is really only suitable for *deterministic* automata; while we might be tempted to use $(\Sigma_{\mathtt{f}/n \in \mathcal{F}} X^n) \to \wp X$, we do not truly wish our transition function to have *arbitrary* tree information in its output, but rather to have the output view of the tree be *derived* from the input. This suggests that we use a pair of (potentially, mutually-recursive) algebras at once, which can be bundled together into a single transition function. For the deterministic case, this would be $(\Sigma_{\mathtt{f}/n \in \mathcal{F}} \langle\!| X, Y |\!\rangle^n) \to \langle\!| X, Y |\!\rangle$; for the nondeterministic case this would be $(\Sigma_{\mathtt{f}/n \in \mathcal{F}} \langle\!| X, Y |\!\rangle^n) \to \langle\!| \wp X, Y |\!\rangle$. The latter highlights the distinction between the two components of the carrier: any nondeterminism is solely on the first (X) component.

① Unconstrained automata can be encoded by taking $X = \mathcal{Q}$ and $Y = \langle\!|\,|\!\rangle$. The interpretation function argument's input type is, then, isomorphic to $\Sigma_{\mathcal{F}}\, \mathcal{Q}^n$ and its output is isomorphic to $\mathcal{Q}$, as might be expected.

② AWEDC can take $X = \mathcal{Q}$ and $Y = \mathcal{H}$. The second component of the interpretation function, behaves as a free algebra on $\mathcal{F}$: given the functor and immediate subterms of a node at a position of the tree under study, the interpretation just builds the subterm under study for the second component of its output.[117] These immediate subterms are also used by the interpreter for checking the constraints and determining the possible state(s) of the output.

Many other classes work out similarly or with slight tweaks to the presentation here. We elide the details.

4.3.2 Tree Automata as Finite Circuits

A different, though surely related, view of an automaton assigns *a set* (of classified objects, e.g., strings or trees) to *each* state and takes the automaton's language to be the union of the sets assigned to the accepting states. Of course, not just any assignment will do: the assigned sets must be mutually compatible according to the transition function (i.e., rules) of the automaton. That is, we can encode an automaton into a *set-valued finite circuit*. The states of the automaton form the items, and the transition function describes the edges (in the string case) or hyperedges (in the tree case) present. The states aggregate their many incident hyperedges using union. The edges construct their labeling sets using any constraints on the transition. The mechanistic description corresponds to the *minimal* fixed-point of the circuit, in which, among other properties, all objects in the value sets are *finite*.

Example 46 (*Translating a String Automaton*): The graph shown in example 40 (in §4.2.2.1) is exactly the graph we would build: the states are items, the edges are possible transitions in the automata. The interpretation of an edge e from q_1 to q_2 labeled as x is that it contributes $l_e^{\mathrm{e}} = \{s +\!\!+ \langle x\rangle \mid s \in l_{q_1}^{\mathrm{n}}\}$ to q_2; the initial edge, with no source, contributes $\{\langle\rangle\}$.

[117] One could certainly imagine classes of automata for which this is not true – grammars which track head-words of constituents or keep only the most recent k nodes of each subterm available, for example.

The least fixed-point solution of the circuit then asserts that $l^{\mathrm{n}}_{q_{\mathrm{even}}}$ is, exactly as we expect, the set of strings containing an even number of `a` symbols: the empty string comes from the initial edge, any string ending with `b` and whose prefix is also in $l^{\mathrm{n}}_{q_{\mathrm{even}}}$ is present due to the self-loop, and any string ending with `a` and whose prefix is accepted by $l^{\mathrm{n}}_{q_{\mathrm{odd}}}$ is also present due to the edge from q_{odd}. ◊

Example 47 (*Translating a Regular Tree Automaton*): As before, each state corresponds to a node, and each transition rule corresponds to a hyperedge of the circuit. The label of the hyperedge is the set built from its tails and the functor of the transition rule: the transition $\mathtt{f}\langle q_1, \ldots, q_n\rangle \mapsto q_0$ has tails at each $q_i \in \mathcal{Q}$, has its sole head at q_0, and is consistently labeled by $\mathtt{f}\langle l^{\mathrm{n}}_{q_1}, \ldots, l^{\mathrm{n}}_{q_n}\rangle$. Thus, the consistent $l^{\mathrm{n}}_{q_0}$ must be a superset of $\mathtt{f}\langle l^{\mathrm{n}}_{q_1}, \ldots, l^{\mathrm{n}}_{q_n}\rangle$. ◊

Endowing automata-as-circuits with constraints requires changing the picture only slightly. *Local* (dis)equality constraints are quite easily added to the above picture. The consistent edge label is now the subset of the regular construction that obeys the constraints of the transition. Adding *global* (dis)equality constraints, à la TAGED or RTA, requires some deeper revision. One possible mechanism is to alter the definition of the circuit so that the values are now sets of *pairs* of trees and finite partial maps from $\mathcal{Q}$ to trees. Formally, the values are, rather than elements of $\wp\mathcal{H}$, now elements of $\wp\langle\mathcal{H}, (\mathcal{Q} \to \{\textsc{null}\} \cup \mathcal{H})\rangle$. When building trees to find the consistent edge label, only those possibilities which jointly respect the indicated global constraints are accepted. Formally, $\langle \mathtt{f}\langle t_1, \ldots, t_n\rangle, m\rangle \in l^{\mathrm{n}}_e$ iff all of the following hold: ① e derives from the rule $\mathtt{f}\langle q_1, \ldots, q_n\rangle \mapsto q_0$, ② there exist $\langle t_i, m_i\rangle \in l^{\mathrm{n}}_{q_i}$ for each i, ③ m is the "coherent merger" of all m_i. Informally, all m_i must agree at each q, save that those m_i which send q to $\textsc{null}$ do not contradict those that send it to a tree. Formally, $m(q) = t$ iff $\forall_i\, m_i(q) \in \{t, \textsc{null}\}$. If no such coherent merger exists, these $\{\langle t_i, m_i\rangle \mid i\}$ do not justify any $\langle \mathtt{f}\langle t_1, \ldots, t_n\rangle, \_\rangle \in l^{\mathrm{n}}_e$ (though others may!).

4.4 Tree Set Automata

Recall that the runtime operation of both the WAM and our strategies of §3 manipulate *sets* of terms at runtime. In order to describe collections of *possible* solver states during analysis (§5.3 and §5.4), we find ourselves needing *sets of sets of trees*, i.e., elements of $\wp\wp\mathcal{H}$. The interpretation of these objects is that the inner $\wp$ captures "runtime uncertainty" about which tree(s), exactly, are being manipulated (the set will be subject to refinement during the course of the solver's operation), while the outer $\wp$ captures "analysis uncertainty" about which state, exactly, the solver will be in.

In light of our rephrasing of automata as circuits, we can see a mechanism for computationally representing these sets-of-sets. We shall use automata with two *kinds* of states: those that describe sets of trees, and those that describe sets of sets (of trees).

4.4.1 Regular Tree Set Automata

We begin with a notion of Regular Tree Set Automaton (TSA), built using two disjoint sets of states, **tree states**, $\mathcal{Q}_1$, and **set states**, $\mathcal{Q}_2$. The accepting states of such automata are set states: $\mathcal{Q}_{\mathrm{F}} \subseteq \mathcal{Q}_2$. The value associated with a tree state (in the circuit view of the automaton) is a set of trees accepted thereby, as with standard tree automata, above. The

Rule type	Transition rule	Meaning
Trees from trees	$\textsc{lp}\ \mathtt{f}\ \langle q_1, \ldots, q_n \rangle \mapsto q_0$	$l^{\mathrm{n}}_{q_0} \supseteq \mathtt{f}\langle l^{\mathrm{n}}_{q_1}, \ldots, l^{\mathrm{n}}_{q_n} \rangle$
Sets from trees	$\textsc{free}\ q \mapsto \hat{q}$	$l^{\mathrm{n}}_{\hat{q}} \supseteq \{ l^{\mathrm{n}}_{q} \}$
	$\textsc{ground}\ q \mapsto \hat{q}$	$l^{\mathrm{n}}_{\hat{q}} \supseteq \{ \{t\} \mid t \in l^{\mathrm{n}}_{q} \}$
	$\textsc{subtype}\ q \mapsto \hat{q}$	$l^{\mathrm{n}}_{\hat{q}} \supseteq (\wp l^{\mathrm{n}}_{q}) \setminus \{\varnothing\}$
Sets from sets	$\textsc{bound}\ \mathtt{f}\ \langle \hat{q}_1, \ldots, \hat{q}_n \rangle \mapsto \hat{q}_0$	$l^{\mathrm{n}}_{\hat{q}_0} \supseteq \{ \mathtt{f}\langle \tau_1, \ldots, \tau_n \rangle \mid \forall_i\, \tau_i \in l^{\mathrm{n}}_{\hat{q}_i} \}$
	$\textsc{bound}_{\subset}\ \mathtt{f}\ \langle \hat{q}_1, \ldots, \hat{q}_n \rangle \mapsto \hat{q}_0$	$l^{\mathrm{n}}_{\hat{q}_0} \supseteq \{ (\wp \mathtt{f}\langle \tau_1, \ldots, \tau_n \rangle) \setminus \{\varnothing\} \mid \forall_i\, \tau_i \in l^{\mathrm{n}}_{\hat{q}_i} \}$

Table 4.1: Transition rules for Regular Tree Set Automata. In all cases, $q \in \mathcal{Q}_1$ is a tree state, $\hat{q} \in \mathcal{Q}_2$ is a set state, and $\mathtt{f}^{/n} \in \mathcal{F}$.

value associated with a set state is a set of *sets* of trees. Transition rules in a TSA come in six forms, summarized in table 4.1. In the following, $q, q_i \in \mathcal{Q}_1$ and $\hat{q}, \hat{q}_i \in \mathcal{Q}_2$; we will pun between the mnemonic of a transition rule and a corresponding set (of sets or trees).

- The simplest form of a TSA rule is an ordinary TA rule, which relates tree states of the TSA. Written $\textsc{lp}\ \mathtt{f}\ \langle q_1, \ldots, q_n \rangle \mapsto q_0$, such a rule asserts that (singleton sets of) each tree of $\textsc{lp}\ \mathtt{f}\ \langle \tau_1, \ldots, \tau_n \rangle \overset{\text{def}}{=} \mathtt{f}\langle \tau_1, \ldots, \tau_n \rangle$ (taking τ_i to be the set of trees which may be labeled by q_i, i.e., $l^{\mathrm{n}}_{q_i}$) are aggregands to the value of q_0 (i.e., each such tree is therefore within $l^{\mathrm{n}}_{q_i}$). (The mnemonic LP stands for "labeled product.")
- Similarly, $\textsc{bound}\ \mathtt{f}\ \langle \hat{q}_1, \ldots, \hat{q}_n \rangle \mapsto \hat{q}_0$ relates *set* states of the TSA. Such a rule indicates that each *set* within $\textsc{bound}\ \mathtt{f}\ \langle T_1, \ldots, T_n \rangle \overset{\text{def}}{=} \{ \mathtt{f}\langle \tau_1, \ldots, \tau_n \rangle \mid \forall_i\, \tau_i \in T_i \}$, taking $T_i = l^{\mathrm{n}}_{\hat{q}_i}$, is also in $l^{\mathrm{n}}_{\hat{q}_0}$.
- Three forms are dedicated to populating set states from tree states:
  - $\textsc{free}\ q \mapsto \hat{q}$ asserts that *the set of* trees which may be labeled by the *tree* state q may, itself, be labeled with $\hat{q}$. (i.e., $\textsc{free}\ \tau \overset{\text{def}}{=} \{\tau\}$).
  - $\textsc{subtype}\ q \mapsto \hat{q}$ asserts that *any non-empty subset of* the set of trees which may be labeled by q may be labeled with $\hat{q}$ (i.e., $\textsc{subtype}\ \tau \overset{\text{def}}{=} \wp\tau \setminus \{\varnothing\}$)
  - We additionally define $\textsc{ground}\ q \mapsto \hat{q}$ as asserting that singleton sets of trees possibly labeled by q may be labeled by $\hat{q}$. (Equivalently, *the singleton subsets* from $\textsc{subtype}\ q$ may be labeled by $\hat{q}$. $\textsc{ground}\ \tau \overset{\text{def}}{=} \{ \{t\} \mid t \in \tau \}$.)[118]
- The last transition form within a Regular TSA is necessary for closure under specialization (to be defined): $\textsc{bound}_{\subset}\ \mathtt{f}\ \langle \hat{q}_1, \ldots, \hat{q}_n \rangle \mapsto \hat{q}_0$ asserts that *any non-empty subset of* any set within $\textsc{bound}_{\subset}\ \mathtt{f}\ \langle T_1, \ldots, T_n \rangle \overset{\text{def}}{=} \{ \mathtt{f}\langle \tau_1, \ldots, \tau_n \rangle \mid \forall_i\, \tau_i \in T_i \}$, taking $T_i = l^{\mathrm{n}}_{\hat{q}_i}$ (i.e., the set of sets which may be labeled by $\hat{q}_i$), may be labeled $\hat{q}_0$.

The names BOUND, FREE, and GROUND are standard in Prolog analysis literature, come to us by particular way of Ramakrishnan [152] and Overton [137], and will be further explained

[118] As presented so far, it is possible to entirely eliminate GROUND and use only BOUND instead. However, when generalizing TSA, it may be convenient to permit more intricate transition rules amongst tree states than just the LP form given above. If these rules lack analogues at the set state level, GROUND could no longer be eliminated. Separately, its presence enables the occasional optimization in implementation.

when we consider planning subgoal conjunctions in §5.3. (The specific set-based definitions given above are derived by extending Mercury's mode system's concretization function [137, Def 3.1.3 etc.] to permit typed leaves.)

We impose the following requirement on the structure of the automaton, in addition to the above: any tree state appearing as the argument to a FREE or SUBTYPE (but not GROUND) transition (i.e., on the left of ↦) must form the root state of a *top-down deterministic regular tree automaton*. This reduction in expressive power is necessary for closure under specialization (to be defined).[119]

4.4.1.1 Operations on Regular TSAs

Outer-Union and Outer-Intersection Given a TSA $\mathfrak{A}$, we seek to form two TAs that describe the union and intersection of the sets accepted by $\mathfrak{A}$, denoted $\bigcup\mathfrak{A}$ and $\bigcap\mathfrak{A}$, respectively. Union is straightforward, and the resulting nondeterministic automaton has the same basic structure as the input TSA. All BOUND and LP rules map to TA rules and all other transition rules map to *epsilon rules*. Intersection is a little more complicated, but tractable because set states participating in cycles (formed by BOUND rules) have empty intersection.[120]

Emptiness Testing Regular TSAs inherit emptiness testing from regular TAs: a depth-first traversal state-marking algorithm suffices to label each state (be it either tree or set) as (non-)empty. The only tweak is that FREE q *always* accepts a set, regardless of q's emptiness, while SUBTYPE q and GROUND q accept a set iff q does.

Acceptence of the Empty Set Regular TSAs may use a very similar algorithm to that of emptiness testing to determine the acceptance of the empty set: FREE $q \ni \varnothing$ iff q is empty, and BOUND $\mathtt{f}\ \langle q_1, \ldots\rangle \ni \varnothing$ iff any of its q_i does. By definition, GROUND, SUBTYPE, and BOUND$_\subset$ rules never justify the acceptance of the empty set.

Projection Projection can be once again extended, this time to operate on sets of sets of trees: $S{\downarrow_\pi} \overset{\text{def}}{=} \{\sigma{\downarrow_\pi} \mid \sigma \in S\} = \{\{s{\downarrow_\pi} \mid s \in \sigma\} \mid \sigma \in S\}$. Computing this operation on TSAs is straightforward: having removed BOUND rules which accept no sets, start from the root states of a TSA, and walk the projection path to find the new root states of the projected automaton. If the projection path descends through a FREE, GROUND, or SUBTYPE rule, the corresponding component of the output automaton will similarly transition from tree states to set states. If the project path descends through BOUND$_\subset$, the corresponding component of the output will, likewise, use BOUND$_\subset$ when transitioning to the root (set) state.

[119] In fact, given this restriction, we can additionally define the last operator from Overton [137], ANY. ANY τ is the union of GROUND τ and $\{\mathtt{f}\langle\sigma_1, \ldots, \sigma_n\rangle \mid \forall_i\, \sigma_i \in \text{ANY}\ \tau_i, \mathtt{f}\langle\tau_1, \ldots, \tau_n\rangle \text{ the } \mathtt{f}^{/n}\text{-rooted subset of } \tau\}$. The need for this second criterion, that the $\mathtt{f}^{/n}$-rooted subset of τ be a product, is why we impose top-down determinism on the automaton describing τ. Less formally, it is either a ground (i.e., singleton) subset of τ or certain to be a set in which all elements share the same root functor. SUBTYPE τ is the *arbitrary union closure* of ANY τ. ANY τ is practically useful as it maps well to the structures encodable by the WAM.

[120] For example, if BOUND $\mathtt{s}\ \langle\hat{q}\rangle \mapsto \hat{q}$ is a rule of $\mathfrak{A}$, then if τ can be labeled by $\hat{q}$ then so can $\mathtt{s}\langle\tau\rangle$ and $\mathtt{s}\langle\mathtt{s}\langle\tau\rangle\rangle$ and so on. The only trees which could occur in all of these sets are infinitely tall, and so, as we consider only finite trees, the intersection must be empty.

Subset Testing Testing whether $\mathcal{L}(A) \subseteq \mathcal{L}(A')$ for two regular TSAs can proceed by a straightforward recursive algorithm, largely relying on the subset tests from regular TAs.

Specialization As hinted above, we will use tree set automata to model, statically, the dynamic behavior of our solver. We will thus need to consider the *possible outcomes* of refinement, knowing only the *possible inputs*. That is, we would like to describe $\{\alpha[\beta/\pi] \mid \alpha \in A, \beta \in B\}$ from (certain) knowledge of π and (uncertain) knowledge of α and β, i.e., A and B, respectively. We define a **specialization** operator, $A[B /\!\!/ \pi]$ to be exactly this set. To implement this operator, it suffices to implement *abstract unification*.

Abstract Unification We define an **abstract unification** operator which computes specialization at the empty path, i.e., $A[B /\!\!/ \langle\rangle]$. (Much as intersection can be seen as refinement at the empty path.) We denote this operation $\mathbin{\cap\mkern-13mu\times}$. Simplifying the definition, we see that $A \mathbin{\cap\mkern-13mu\times} B \overset{\text{def}}{=} \{\alpha \cap \beta \mid \alpha \in A, \beta \in B\}$, which helps explain the notation: it is the intersection along a product of A and B. Given two Regular TSAs, one encoding A and the other encoding B, we can define an algorithm for computing a Regular TSA which encodes $A \mathbin{\cap\mkern-13mu\times} B$. This algorithm proceeds much as any other product construction: the tree (resp. set or accepting) states of the resulting automata are pairs of tree (resp. set or accepting) states, one from each of the two automata. The transition rules in the resulting automaton arise from considerations of pairs of transition rules, one from each automaton and largely follow from the set-theoretic definitions of the directives. We extend notation slightly to use the directive notation to refer to the set it creates.

- First, we note that $(T \cup T') \mathbin{\cap\mkern-13mu\times} S = (T \mathbin{\cap\mkern-13mu\times} S) \cup (T' \mathbin{\cap\mkern-13mu\times} S)$. This equivalence forms the basis of handing of multiple rules targeting the same states within TSAs being unified: recall, in the circuit analogy of §4.3, these states' values are *aggregated* by union.

- $\textsc{free}\ q \mathbin{\cap\mkern-13mu\times} \textsc{free}\ q' = \textsc{free}\ (q \cap q')$. Our algorithm thus calls out to intersection of the *regular tree automata* whose accepting states are q (resp q') and whose transition rules come from their respective regualr TSAs. Similarly handled are $\textsc{free} \mathbin{\cap\mkern-13mu\times} \textsc{subtype}$ and $\textsc{subtype} \mathbin{\cap\mkern-13mu\times} \textsc{subtype}$ abstract unifications.

- $\textsc{bound}\ \mathtt{f}\ \langle \hat{q}_1, \ldots\rangle \mathbin{\cap\mkern-13mu\times} \textsc{bound}\ \mathtt{f}\ \langle \hat{q}'_1, \ldots\rangle = \textsc{bound}\ \mathtt{f}\ \langle \hat{q}_1 \mathbin{\cap\mkern-13mu\times} \hat{q}'_1, \ldots\rangle$; for mismatched symbols or arities, the answer is $\textsc{free}\ \varnothing$. The other three $\textsc{bound} \mathbin{\cap\mkern-13mu\times} \textsc{bound}$ unifications, in which one or both is $\textsc{bound}_{\subset}$ behave identically save that their output directive is also $\textsc{bound}_{\subset}$.

- $\textsc{ground}\ q \mathbin{\cap\mkern-13mu\times} \hat{q}$ is $\textsc{ground}$ of an automaton recognizing the intersection of the language labeled by the (tree) state q and the outer union of the sets labeled by the (set) state $\hat{q}$. Such things are readily computed using TA algorithms and the outer union algorithm given above. The empty set must be added to the language of sets recognized iff $l^{\mathrm{n}}_{q} \notin \bigcup l^{\mathrm{n}}_{\hat{q}}$.

- $\textsc{free}\ q \mathbin{\cap\mkern-13mu\times} \textsc{bound}\ \mathtt{f}\ \langle T_1, \ldots, T_n\rangle$ relies on q being top-down deterministic: there is at most one rule targeting q that applies to trees whose roots are labeled with $\mathtt{f}^{/n}$; selecting $\textsc{lp}\ \mathtt{f}\ \langle q_1, \ldots, q_n\rangle$ as its LHS, the answer becomes $\textsc{bound}\ \mathtt{f}\ \langle \textsc{free}\ q_1 \mathbin{\cap\mkern-13mu\times} T_1, \ldots\rangle$

- FREE $q \otimes$ BOUND$_{\subset}$ `f` $\langle T_1, \ldots \rangle$ requires top-down determinism of q and behaves as the prior case, save, again, that the answer is BOUND$_{\subset}$ `f` $\langle$FREE $q_1 \otimes T_1, \ldots \rangle$.
- SUBTYPE $q \otimes$ BOUND `f` $\langle T_1, \ldots \rangle$ and SUBTYPE $q \otimes$ BOUND$_{\subset}$ `f` $\langle T_1, \ldots \rangle$ are as above (and continue to require top-down determinism of q).

4.4.2 Inner-Rigid Tree Set Automata

We can extend Regular Tree Set Automata to handle global equality constraints, borrowing from Rigid Tree Automata [100]. A particular tree t in a set τ accepted by a TSA can be given a run labeling: every position is associated with at most one $\mathcal{Q}_1$ and at most one $\mathcal{Q}_2$; only the nodes at bridging transitions (e.g., FREE) will have both. We extend the above definition with **inner-rigid states**, both tree, $\mathcal{Q}_{1,\mathrm{R}} \subseteq \mathcal{Q}_1$, and set, $\mathcal{Q}_{2,\mathrm{R}} \subseteq \mathcal{Q}_2$, to obtain Inner-Rigid Tree Set Automata (IRTSA). The sets accepted by an IRTSA are of trees whose runs obey the rigidity requirement: for each $\tilde{q} \in \mathcal{Q}_{1,R} \cup \mathcal{Q}_{2,\mathrm{R}}$, all nodes in a run labeled with $\tilde{q}$ are the roots of equal subtrees. These rigid states allow us to encode the reuse of variables within a Prolog rule.

Operations on IRTSA The operations given above need relatively little adjustment.

Outer-Union and Outer-Intersection Given an IRTSA $\mathfrak{A}$, the same procedure given above will produce RTAs describing the desired sets.

Emptiness Testing State-marking for emptiness testing naturally generalizes from TAs to RTAs and TSAs and also to IRTSAs; no further changes are needed.

Acceptence of the Empty Set State-marking again suffices.

Projection An analogous walk continues to suffice; projection continues to behave well in conjunction with global constraints.

Subset Testing For IRTSAs, no decidable strategy exists, as RTAs do not have decidable subset tests. Approximations exist, however, and we hope that they are sufficiently capable to handle our needs, in practice.[121]

[121] In particular, our sole use of TSAs is as machinery for backing the *static analysis* of the next chapter. In such analysis, if an approximation gives up or times out, we will reject (part of) a program or a plan for execution within that program. Further, we can use approximations with *one-sided* error (though the acceptable side will vary by the use in the analysis system) and obtain the same behavior: we reject (part of) a program or a plan for execution within that program. This is substantially less dire than getting the wrong answer at runtime.

Unification The unification algorithm for TSAs continues to hold, though the handling of GROUND transitions must shed a little precision due to the lack of decidable subset testing mentioned above. This is usually no major loss, as the only casualty is accuracy of the empty set's inclusion, and our analyses do not pay careful attention to the empty set, as it carries no terms and therefore cannot impact the semantics of the program at runtime.

4.4.3 TATA Tree Set Automata

The canonical text on Tree Automata, Comon et al. [35, ch. 5], introduces a concept termed a "generalized tree set automaton". These automata recognize "E-valued $\mathcal{F}$-generalized tree sets" ("GTS"),[122] i.e., maps $g : \mathcal{H} \to E$. The set E generalizes the use of $\{\top, \bot\}$, whose use would make the function g an indicator function of a set of $\mathcal{F}$-trees. These automata are described in ways that make them (apparently) less amenable to our use case, for which a more overtly recursive (circuit-like) description seems more natural.

To understand this in more detail, some words must be said about these GTS automata. (A reader interested only in Dyna loses little to nothing by skipping the remainder of this section.) We adjust the notation of Comon et al. [35, ch. 5] to be a little more consistent with the rest of this section. A GTS automaton has a set of states $\mathcal{Q}$ and its transition rules are of the form $\mathtt{f}\langle q_1, \ldots, q_n\rangle \overset{e}{\hookrightarrow} q_0$, where $\mathtt{f}^{/n} \in \mathcal{F}$, $e \in E$, and $\forall_i\, q_i \in \mathcal{Q}$. The usual notion of non-determinism continues to apply.

The first major difference from the other automata of this section is that a GTS automaton does not have a set of accepting states; instead, it has a set of accepting *sets of* states. That is, rather than a set $\mathcal{Q}_\mathrm{F}$, it has $\mathfrak{Q} \subseteq \wp(\mathcal{Q})$. As a result, the acceptance condition on a run r can no longer be that $r(t) \in \mathcal{Q}_\mathrm{F}$.

Despite recognizing sets of trees, a run of a GTS remains centered around a function of *an individual tree*: $r \in \mathcal{H}_\mathcal{F} \to \mathcal{Q}$. As before, r is a run on a tree t iff there is a transition rule justifying the label of each subterm in terms of the labels of its immediate children. In particular, this means that r is a run iff, for all paths π in some tree t and letting $s = t{\downarrow_\pi}$, if $r(s) = q_0$, the root functor of s is $\mathtt{f}^{/n}$, $\forall_{i \in \mathbb{N}_1^n}\, r(s{\downarrow_i}) = q_i$, and $g(s) = e$, then the automaton contains the transition rule $\mathtt{f}\langle q_1, \ldots, q_n\rangle \overset{e}{\hookrightarrow} q_0$. The acceptance condition of a run is now that *the image under* r of $\mathcal{H}_\mathcal{F}$ is an element of $\mathfrak{Q}$. (Formally, $(\wp(r))(\mathcal{H}_\mathcal{F}) \in \mathfrak{Q}$; we do not need to quantify over paths as r is a *function* and, if π is a path of $t \in \mathcal{H}_\mathcal{F}$, then $t{\downarrow_\pi} \in \mathcal{H}_\mathcal{F}$.)

Example 48: Comon et al. [35, Exs. 5.2.1.1 and 5.2.1.2] demonstrate GTS automata accepting what we would call FREE τ and SUBTYPE τ with τ being the set of lists of Peano-encoded natural numbers (formally, the minimum fixed-point solution to the simultaneous equations $\tau = \mathtt{nil}\langle\rangle \cup \mathtt{cons}\langle\sigma, \tau\rangle$ and $\sigma = \mathtt{z}\langle\rangle \cup \mathtt{s}\langle\sigma\rangle$). We repeat their construction here (with notation adapted). The automata given use $E = \{0, 1\}$ and $\mathcal{Q} = \{\mathrm{Nat}, \mathrm{List}, \mathrm{Term}\}$. The first automaton (recognizing only the set τ itself, or, really, the function $g \in \mathcal{H}_\mathcal{F} \to \{0, 1\}$ defined by

[122] That is, they are automata for generalized tree sets rather than a generalization of some other class of tree set automata.

$g(t) = 1$ iff $t \in \tau$) takes as its transition rules

$$\begin{aligned}
\texttt{z}\langle\rangle &\overset{0}{\longmapsto} \text{Nat} \\
\texttt{s}\langle\text{Nat}\rangle &\overset{0}{\longmapsto} \text{Nat} \\
\texttt{nil}\langle\rangle &\overset{1}{\longmapsto} \text{List} \\
\texttt{cons}\langle\text{Nat}, \text{List}\rangle &\overset{1}{\longmapsto} \text{List}
\end{aligned}$$

together with rules which encode that all other trees mapped to 0 by g transition to the Term state. The automaton then accepts g iff there exists a run r such that $(\wp(r))(\mathcal{H}_{\mathcal{F}}) = \mathcal{Q}$.[123] We see that g is accepted iff it sends *all* lists of naturals to 1 and all other trees (i.e., naturals and other "non-lists" like $\texttt{cons}\langle\texttt{nil}\langle, \rangle\texttt{s}\langle\texttt{nil}\langle\rangle\rangle\rangle\rangle$) to 0. That is, this automaton recognizes FREE τ. If we add the transition rule $\texttt{nil}\langle\rangle \overset{0}{\longmapsto} \text{List}$, then the resulting automaton recognizes both τ and $\tau \smallsetminus \{\texttt{nil}\langle\rangle\}$. If we then remove $\texttt{nil}\langle\rangle \overset{1}{\longmapsto} \text{List}$, only the latter set remains recognized. Similarly, adding both $\texttt{nil}\langle\rangle \overset{0}{\longmapsto} \text{List}$ and $\texttt{cons}\langle\text{Nat}, \text{List}\rangle \overset{0}{\longmapsto} \text{List}$ allows acceptance of any subset of τ, including the empty set. In order to exclude the empty set, we must split the state List to List_0 and List_1. We then use the transition rule schemata

$$\begin{aligned}
\texttt{nil}\langle\rangle &\overset{i}{\longmapsto} \text{List}_i & \forall_i \\
\texttt{cons}\langle\text{Nat}, \text{List}_i\rangle &\overset{j}{\longmapsto} \text{List}_k & \forall_{i,j,k}\, k = 1 \Leftrightarrow (i = 1 \vee j = 1)
\end{aligned}$$

and require that List_1 be in the r-image of $\mathcal{H}_{\mathcal{F}}$. A similar technique lets us ensure that only singleton sets are recognized, by instead transitioning as $\texttt{cons}\langle\text{Nat}, \text{List}_1\rangle \overset{1}{\longmapsto} \text{List}_2$ (and excluding the transition to List_1 with identical left hand side and e) and $\forall_i\, \texttt{cons}\langle\text{Nat}, \text{List}_2\rangle \overset{i}{\longmapsto} \text{List}_2$, where List_2 is a new state not in any of the accepted state sets. Thus, we track in the automata state whether 0, 1, or more than 1 list *subterm* of a given term has been accepted by g. The definition of acceptance then lets us test for whether *any* tree has two accepted list subterms, which implies that two lists are accepted by g. ◊

[123] As typical, the transition rules do not define a total function. Again the construction of a "dead state" as in footnote 107 is sufficient; this "dead state" is *not in* the single accepted state set, which remains $Q = \{\text{Nat}, \text{List}, \text{Term}\}$. It is unnecessary, but not incorrect, to enlarge $\mathfrak{Q}$ from $\{Q\}$ to $\wp Q$, as the other entries do not apply; for analytic reasons, this "subset closure" property of $\mathfrak{Q}$ is of interest; Comon et al. [35] calls such GTS automata "simple."

Chapter 5

Weighted Logic Program Analysis

> Present-day computers are designed primarily to solve preformulated problems or to process data according to predetermined procedures. The course of the computation may be conditional upon results obtained during the computation, but all the alternatives must be foreseen in advance. … The requirement for preformulation or predetermination is sometimes no great disadvantage. It is often said that programming for a computing machine forces one to think clearly, that it disciplines the thought process. If the user can think his problem through in advance, symbiotic association with a computing machine is not necessary.
>
> J. C. R. Licklider. *Man-Computer Symbiosis.* [113]

5.1 Why Analysis?

Logic languages offer compact, declarative specifications of computational problems. Despite the high level of abstraction, however, there remain a number of opportunities for programmer error when creating a μDyna (or, more likely, Dyna) program. By identifying these opportunities, we can put safeguards in place so that the compiler can, by automated, *static* reasoning about the program, ensure that the programs given are not only well-founded but "sensible" (colloquially, we might say that they "pass the smell test"). Examples of things which, while possibly perfectly well-founded, might be cause for concern include

① **dead code**, i.e., useless components of the program, including rules with heads which are not queried (by the driver or as subgoals) or, dually, rules which contain subgoals which are, in turn, never given values;

② **vacuity**, i.e., a rule whose answer sets can all be shown to be empty; and

③ **incomplete definitions**, i.e., certainty of subgoal failure when the program indicated otherwise.

These are not orthogonal concerns; correcting one instance may simultaneously correct others within the same program or may *expose* others.[124] Having the compiler (attempt to) flag such issues before permitting code to be executed should help minimize divergence between programmer expectations and actual results from the solver. Because the compiler may perform a number of complex transformations on the input program (including unfolding of definitions and subgoal reordering), we will use a *simplified* model of execution as the basis for these tests. Reasoning proceeds by *over-estimating* the sets of queries and answers that may arise within the program. This reasoning will be our focus in §5.2.

In traditional, more procedural languages, a program being accepted by static analysis (typically, such programs are called "well-typed") means that it "cannot be blamed" [186]; among other things, it neither "goes wrong" (by attempting to use a piece of data in an unintended way) nor "gets stuck."[125] However, as part of logic languages' high-level nature, they tend to be lax in describing programs' actual execution.[126] This is certainly true of μDyna programs and so a different form of static analysis is required of μDyna programs in order to ensure that they can actually be executed (recall the discussion of planning in §3.2.1.1). This analysis, in effect, confirms that every query or update *claimed* to be supported can, in fact, be supported—that is, that the dynamic data structure defined by the μDyna program can achieve its consistency goal under arbitrary streams of queries and updates. Part of this analysis ensures that we will never encounter an *instantiation error* at runtime.[127] As part of this analysis, we are in a position to eliminate runtime overheads, for example, by enabling derivation of efficient storage representations and converting unification to pattern matching whenever possible. Here, or rather, in §5.3 and §5.4, we put our Tree Set Automata of §4.4 to work, upper-bounding the *set of possible solver states* during possible executions.

In both cases, our analyses are framed as *set-theoretic* operations. While these operations are presumed to be *backed* by automata of some form or another as part of their computational implementations, we do not rigidly specify their form.[128] As a result, there is

[124] This temporally non-monotonic behavior of compiler-reported error and warning diagnostics is likely depressingly familiar to most programmers.

[125] The usual notion of progress, that there is always some expression which reduces, merely means that the program continues to compute. It does not, for example, guarantee that that computation is *useful*; a tight infinite loop doing nothing counts as "progress." More refined notions of "productivity," of a program also exist; some languages either are total or have total fragments, wherein expression reduction is certain to eventually culminate in a non-reducible *value*.

[126] Standard Prolog, interestingly, *has* a specified execution order: each query visits each rule in a top-down order and, after unifying with the head, proceeds left-to-right across queries. This is not strictly required by the *pure* fragment of the language, nor by the notion of SLD resolution, but is required for sensible handling the *extra*-logical components of the language, including the procedural operation of "cut" (see footnote 138, in §5.3). Despite this, extensions such as tabling for negation or subsumption (recall §3.7.1) typically necessitate alternate execution orders; programmers are discouraged from mixing tabling and extra-logical facilities. Datalog solvers, by design working with a simpler language than Prolog, typically also do not specify their execution order up front and use fundamentally dynamic execution techniques such as semi-naïve bottom-up evaluation (recall §2.2.3).

[127] We could attempt to add an "instantiation error" value to the codomain of our data structure's functions and offer somewhat equivalent functionality. Prolog programs, by virtue of having a defined execution order, are actually able to describe a semantic instantiation error, while so-augmented Dyna programs, which do not have fixed execution order (recall §3.1.4), would have a more complicated story to tell. See §6.1 for discussion of adding exceptions to Dyna.

[128] We are not the first to propose using automata as the mechanism for reasoning about logic languages;

a potential divide between the expressive power of our theory and the conclusions that may be attainable in practice. The salient point is this: we have set out to design the tightest possible analysis, but recognize that in practice some operations will be approximated. So long as the approximations are safe, which, typically, would mean that they *over-estimate* the conditions that may occur in practice, any conclusions reached by our framework remain valid.

Both forms of analysis are **abstract interpretations** [38, 39] of the runtime behavior of the program (under the care of either the real solver or an idealized approximation thereof). Such interpretations rely on the existence of a *Galois connection* [136] between the so-called "concrete" and "abstract" domains (respectively, the actual runtime behavior and the analytic approximation). Somewhat unusually, both our abstract and concrete domains are given in the language of set theory; that is, we do not, as would be more typical, construct a different syntactic object for our abstract domain. (*In practice*, of course, we imagine that an actual implementation of our system will use the syntactic formalism of tree (set) automata to specify the abstract domain.) Our (implicit) Galois connection is thus somewhat degenerate, in that the concretization function is *the identity function* and our abstraction function may be any function f such that $\forall_x f(x) \supseteq x$.

5.2 Grounding Set Analysis

As mentioned at the very beginning, in §1.1, one of the selling points of logic languages in general is that the program should (and often does) stand alone, with semantics separated from any particular *algorithm* for finding those semantics. It seems appropriate, then, that we begin our tour of program analysis by attempting to characterize properties of *the program* itself rather than *its execution*. As μDyna programs, like pure Prolog programs, serve to define partial functions of items (associating with each its assigned value, or, in the case of Prolog, provability), analysis of programs attempts to find or verify useful facts about these functions.

The most precise analysis of a program would, of course, be to find all of its solutions, exactly. Moreover, if, as in the typical use-cases of Dyna, the input is potentially not available at analysis time, we should fully characterize the map of inputs to solutions, i.e., find all solutions *for all possible inputs*. However, coarser analyses are still of utility.

If one yearns for a procedural intuition, one can think of this kind of analysis as characterizing the behavior of some hypothetical solver which can reason about potentially infinite disjunction in finite time—some kind of recursive, nondeterministic automaton, wherein each branch of nondeterminism within each ply of recursion deals with precisely one ground rule query, obtaining an answer to be combined with the other nondeterministic branches.

5.2.1 Review: Types for Prolog

There are two, strikingly different, extant approaches for this kind of analysis on Prolog programs, which emphasize different kinds of properties of the program.

see, for example, Frühwirth et al. [70], Heintze and Jaffar [89], and Talbot, Tison, and Devienne [171].

On the one hand, one finds systems for which the type of a program is a superset of all items that could ever be proven. These have come to be called **optimistic types** [154] and were introduced by Mishra [126]. While the loosest possible reading of this definition would imply that a program is well-typed at *any* superset of its provable items, typically (but possibly not universally) the type of a program will be a fixed-point of some *upper-bounding approximation* of the logical consequences that derive from a program's rule. Heintze and Jaffar [90] provides an excellent overview.

On the other hand, one has systems in the school of Mycroft and O'Keefe [132], notably including that of Jeffery [103], where the notion of "well-typed" is *orthogonal* to the notion of provability within the rules of the Prolog program. In these systems, one more directly appeals to the solver machinery and speaks of the clauses of a program and the SLD-resolution *query* (conjunction of pending subgoals) as being well-typed. One of the central results of a type system of this flavor is that of *preservation*: SLD-resolution of a well-typed query using a well-typed rule gives rise to another well-typed query.

5.2.2 Dyna's New Twist: Weights and Aggregation

As mentioned earlier, Prolog does not distinguish between one and more-than-one derivation of the same item (formally, its sole aggregator, OR, is idempotent). Thus, there are relatively straightforward upper-bounding approximations of the logical consequence operator that can be used to drive the "optimistic type" systems. However, when we consider analysis of a language with aggregation, the analogous approximations would be approximations of the heads and *their aggregands*, not heads and *their values*.

Were we to try to statically analyse Prolog extended with Answer Subsumption (recall §3.7.1), we would need an approximation of the *modified* logical consequence operator, which includes the semilattice used for subsumption. Such an approximation would need to consider all possible semilattice elements that could be the result of an operation, given some description of possible inputs. The simplest such description, in keeping with the optimistic type systems, would be a *set* of possible inputs, with the interpretation that *any subset* could be seen at runtime. One could imagine, however, wanting *lower bound* information as well, such as *certainty* of particular answers occurring in the set.

In the case of Dyna, there is the further twist that our aggregators are not only potentially not selective but not idempotent. That is, they are (potentially) sensitive to not just the divide between zero and non-zero occurrences of any input value, but (potentially) sensitive to the exact number of each aggregand. While we could continue to simply track the set of possible aggregands, with the semantics that each element may occur between 0 and ∞ times, it is worth looking for a tighter analysis, and we shall do so below.

5.2.3 Simulated Information Flow for μDyna

Before addressing the aggregators' *disjunctive* behavior, we first propose some constraints solely on the *conjunctive* behavior of rule subgoals. While, ultimately, we will have a pair of conjunctive constraints on the program, presently, for simplicity of presentation, we consider only the first in this section.

We consider a program to **respect simulated information flow** *only if there exist* sets of possible answers, $\mathfrak{D}$, and permitted queries $\mathfrak{K}$, such that $\mathfrak{D} \subseteq \mathfrak{K} \subseteq \langle \mathcal{H}, \mathcal{H} \rangle$ and such that an approximation of the solver, operating in a strictly-left-to-right manner, makes permitted queries when it is given possible answers as responses.[129] The set $\mathfrak{D}$ is precisely a set of optimistic types, extended to the weighted case (i.e., it is not just a set of items, but a set of possible item-and-value kv-pairs). They are relevant to us in ways they are not to the SLD-centric type systems because we are enforcing an approximate, type-level notion of **call compatibility** by requiring that queries be a subset of $\mathfrak{K}$, while the traditional systems need only consider the intersection of queries and rule heads. Despite handling queries, our proposal is in fact quite different from these SLD-centric systems: we consider each rule in isolation, using $\mathfrak{D}$ in lieu of subgoal-head unifications, because we cannot, in general, simply unfold a child item's rule(s) into those of a parent item, as would be done within SLD.

Recall the computeRule operation of our backward-chaining algorithms of listings 3.2 and 3.3 and, specifically, its refineRuleSuffix core, which uses Lookup to obtain the answers to the current subgoal and uses each response (of answers) to refine the rule. We are going to ignore the forking of the search tree; that is, we assume that *the union* of all Lookup responses is *a subset* of $\mathfrak{D}$, and require that each invocation of Lookup is given *a subset* of $\mathfrak{K}$. That is, we will track a set constructed from *upper-bounds* on answers and ensure that it is, in turn, upper-bounded by the set of permitted queries.

To formalize this, first, define a function which simulates the action of the recursive behavior of refineRuleSuffix within computeRule(r,$\mathfrak{K}$) by upper-bounding (the union of) the subsets of ρ_r used therein:

$$\mathrm{refrule}_{\mathrm{type}}(r, i) \overset{\mathrm{def}}{=} \rho_r[\mathfrak{K}{\downharpoonleft_1}/\textsc{head}][\mathfrak{D}/\textsc{sg}.1]\cdots[\mathfrak{D}/\textsc{sg}.i].$$

(We assume that the parameters $\mathfrak{D}$ and $\mathfrak{K}$ are globally available, and so do not need to be arguments to $\mathrm{refrule}_{\mathrm{type}}$.) So armed, we formally define a program to respect simulated information flow only if $\forall_{r \in \Xi}\, \forall_{i \in \mathbb{N}_1^{n_r}}\, \mathrm{refrule}_{\mathrm{type}}(r, i-1){\downharpoonleft_{\textsc{sg}.i}} \subseteq \mathfrak{K}$.

5.2.3.1 Membership Implications on Sets

However, as alluded to in §4.2.6, the automata with which we typically imagine backing such a set $\mathfrak{D}$ are unlikely to be sufficiently expressive to capture many useful kinds of constraints. We therefore revise our definition above to *approximate* the refinement operators $\cdot[\mathfrak{D}/\cdot]$ and subset tests $\cdot \subseteq \mathfrak{K}$ using a formalism with more expressive power (and, concomitantly, fewer decidable properties).

We introduce **membership implication constraints** on a set. Each such constraint on a set σ is an assertion of the form "$\forall_{s \in \sigma, \vec{\tau}}\; s \in p(\vec{\tau}) \Rightarrow s \in q(\vec{\tau})$" for some functions p and q which each take some tuple of parameters and return a *set*. Such constraints serve

[129] Absent in this description is any kind of notion of well-typedness of individual rules; that is, rules are merely constraints on the possible $\mathfrak{D}$ and $\mathfrak{K}$ sets. We credit the BigBang effort at an adjacent lab for inspiration and reassurance that this was a viable approach. See Palmer [138] for details of the project, and §3.5 therein in particular for their type system. We violate our typographic convention of using lowercase Greek letters for sets of trees for the $\mathfrak{D}$ and $\mathfrak{K}$ sets; we consider these sufficiently meaningful to merit their own symbols.

to *exclude* $\bigcup_{\vec{\tau}} p(\vec{\tau}) \smallsetminus q(\vec{\tau})$ from σ; the p_i and q_i functions are likely amenable to encoding via the parameterized automata of Xiao, Sabry, and Ariola [191, §5] (recall §4.2.6). In practice, because we will also be manipulating the set σ separately from these attached constraints, we will *generate* our constraints element-at-a-time, from templates. That is, given (equal-length) *tuples* of pre- and post-conditions $\vec{p}$ and $\vec{q}$, we define the (meta-level) function $\mathrm{cby}_{\vec{p},\vec{q}}(t) \overset{\mathrm{def}}{=} \forall_i \forall_{\vec{\tau}}\, t \in p_i(\vec{\tau}) \Rightarrow t \in q_i(\vec{\tau})$ (in each, $\mathrm{tlen}(\vec{\tau})$ may vary with i). All told, then, given a set σ and such a constraint template cby, we can define the constraint-respecting subset of σ, $[\![\sigma, \mathrm{cby}]\!] \overset{\mathrm{def}}{=} \{s \in \sigma \mid \mathrm{cby}(s)\}$.

Importantly, even if σ and all $\{p_i(\vec{\tau}), q_i(\vec{\tau}) \mid i, \vec{\tau}\}$ are languages recognizable by some family of TAs, $[\![\sigma, \mathrm{cby}_{\vec{p},\vec{q}}]\!]$ may not be recognizable by that family. Thus, even given imperfect ability to represent tree sets, we may still find power in this constraint construction. (If we restrict the ranges of the quantification over $\vec{\tau}$ within the formulae of $\mathrm{cby}(\cdot)$ to only those $\vec{\tau}$ that can be expressed within a TA family, or such that $p_i(\vec{\tau})$ and $q_i(\vec{\tau})$ can be so expressed, we are then describing a set σ'', with $\sigma' \subseteq \sigma'' \subseteq \sigma$.)

Example 49: Consider, again, our example from §4.2.6 of pairs of reversed lists. While we cannot, in general, describe the set given therein,

$$v(\mathcal{H}) = \{\mathtt{v}\langle \mathtt{c}\langle x_1, \mathtt{c}\langle x_2, \cdots \mathtt{n}\langle\rangle\rangle \cdots\rangle, \mathtt{c}\langle x_n, \mathtt{c}\langle x_{n-1}, \cdots \mathtt{n}\langle\rangle\rangle \cdots\rangle\rangle \mid n \in \mathbb{N}, \forall_i\, x_i \in \mathcal{H}\},$$

even with our proposed membership implication constraints, we can still do better than approximating it by its superset $\sigma = \mathtt{v}\langle l(\mathcal{H}), l(\mathcal{H})\rangle$, where by $l(x)$ we mean the least fixed-point of the equation $l(x) = \mathtt{n}\langle\rangle \cup \mathtt{c}\langle x, l(x)\rangle$. In particular, it is the case that, $\forall_{a \in v(\mathcal{H})}$, $\forall_\beta\, a \in \mathtt{v}\langle l(\beta), \mathcal{H}\rangle \Rightarrow a \in \mathtt{v}\langle l(\beta), l(\beta)\rangle$ and similarly for the second list. Thus, $[\![\sigma, f]\!]$, with $f(a)$ being the pair of formulae just described, is a superset of $v(\mathcal{H})$.

While this approximation does not capture the facts that the lists are of equal length, nor the fact that the elements are reversed, it enables transfer of upper bounds. Absent any additional knowledge, the only $\vec{\tau} = \langle\beta\rangle$ for which the implication is not vacuous is $\mathcal{H}$, but in which case, the implied upper-bound is σ itself, and so the implication is tautological. However, when we would wish to subsequently approximate the intersection (say, within refinement) of $v(\mathcal{H})$ with some other set, such as $\mathtt{v}\langle l(\phi), \mathcal{H}\rangle$, with $\phi \subsetneq \mathcal{H}$, the constraints let us justify using $\mathtt{v}\langle l(\phi), l(\phi)\rangle$ as an upper bound, rather than the larger $\sigma \cap \mathtt{v}\langle l(\phi), \mathcal{H}\rangle = \mathtt{v}\langle l(\phi), l(\mathcal{H})\rangle$. ◊

Example 50: Analogous reasoning lets us give constraint sets for a membership relation on lists, i.e., $\mathtt{m}\langle e, s\rangle$ if e is an element of the list s. Precisely, we would expect something like $m(\tau) = \{\mathtt{m}\langle e, s\rangle \mid s \in l(\tau), i \in \mathbb{N}, \pi = 2.\cdots.2, \mathrm{tlen}(\pi) = i, s{\downarrow_{\pi.1}} = e\}$. However, we may approximate this set by $[\![\mathtt{m}\langle \mathcal{H}, l(\mathcal{H})\rangle, \{s \mapsto \forall_\beta\, s \in \mathtt{m}\langle \mathcal{H}, l(\beta)\rangle \Rightarrow s \in \mathtt{m}\langle \beta, l(\beta)\rangle\}]\!]$. There is only *one* formula in the associated constraint set: while an upper-bound on the type of elements in the list (definitionally) transfers to an upper-bound on the type of an element, an upper-bound on the type of the *member element* e does not transfer to being an upper bound on the type of the *list elements* (of s). ◊

These constrained set expressions have several convenient properties. The family of sets they describe is closed under intersection, as $\forall_{\sigma,\sigma',c,c'} [\![\sigma, c]\!] \cap [\![\sigma', c']\!] = [\![\sigma \cap \sigma', \{s \mapsto c(s) \wedge c'(s)\}]\!]$ (if one set is not constrained, this simplifies to $\forall_{\beta,\sigma,c} [\![\sigma, c]\!] \cap \beta = [\![\sigma \cap \beta, c]\!]$). Refinement, being just intersection in disguise, is similarly readily computed, though the notation is a little unwieldy. Constraints may be *eliminated* if their preconditions are vacuous: that is, if $\forall_{s \in \sigma, \vec{\tau}}\ s \notin p_i(\vec{\tau})$ (or $\forall_{s \in \sigma, \vec{\tau}}\ s \in q_i(\vec{\tau})$, though this seems less likely to arise in practice) can be shown, then the constraints formed from p_i and q_i may be removed without changing the meaning. More importantly, however, they offer a kind of constraint *application*: if there exists a p_i and $\vec{\tau}$ such that $\sigma \subseteq p_i(\vec{\tau})$, then we may justifiably equate the expressions $[\![\sigma, c]\!] = [\![\sigma \cap q_i(\vec{\tau}), c]\!]$. Here, at long last, we find the point: these constrained set expressions may allow us to arrive at tighter *recognizable* upper bounds than we could if we insisted that we have a single automaton after every refinement operator in $\mathrm{refrule}_{\mathrm{type}}(r, i)$.

5.2.4 Answer Closure

While perhaps interesting to consider in isolation, this notion of "respecting simulated information flow" has a rather glaring fault. *Every* program would pass the analysis as defined so far, simply by taking $\mathfrak{O} = \varnothing$ and $\mathfrak{K} = \langle \mathcal{H}, \mathcal{H} \rangle$, because the above provides no lower bounds on the set of possible answers $\mathfrak{O}$.[130] While we said that $\mathfrak{O}$ was supposed to be a set (possibly encoded using the constraint mechanism just given) of optimistic types about the program, we did not check that $\mathfrak{O}$ actually *was* such a set! Let us consider how to do so, now. Formally, then, we are revising our definition of respecting simulated information flow to be an existentially-quantified conjunct: the existence of $\mathfrak{O}$ and $\mathfrak{K}$ as given above *and* such that the additional constraints below hold of $\mathfrak{O}$.

As part of the above test for respecting simulated information flow, we fall just shy of computing the set $\mathrm{refrule}_{\mathrm{type}}(r, n_r)$ for each rule. Specifically, we are already computing $\mathrm{refrule}_{\mathrm{type}}(r, n_r - 1)$ for use within the definition of respecting simulated information flow; another round through $\mathrm{refrule}_{\mathrm{type}}$'s recursion will get us $\mathrm{refrule}_{\mathrm{type}}(r, n_r)$. This set represents an upper bound (optimistic type) for the *rule answers* of r. (If the program respects simulated information flow, as defined so far, then we can even say that these rule answers are derived from permitted rule queries.) Notably, $\tau = \mathrm{refrule}_{\mathrm{type}}(r, n_r)\downharpoonleft_{\mathrm{HR}}$ is a meaningful quantity, especially when viewed as a dependent sum: it describes an upper bound on the set of aggregands that this rule may contribute to each of (an upper bound of) its heads.

There are, however, at least two challenging aspects to any use of this τ. First, aggregation is done *per head*, so we would like to compute something of the form

$$\{\langle h, a(\tau[\{h\}/1]\downharpoonleft_2)\rangle \mid h \in \tau\downharpoonleft_1\},$$

for some approximation of aggregation, a. However, $\tau = \Sigma_{t \in \tau\downharpoonleft_1} \sigma_t$ may be a dependent sum with an infinite domain (of heads) and may not have a finite partition for piecewise-constancy of $(\sigma.)$; our automata-based machinery offers us no general mechanism for computing sets of the above form. Second, we are tracking only *sets* and so have no way to estimate the multiplicities of each value associated with a given head. Because $\mathrm{refrule}_{\mathrm{type}}(\cdot, \cdot)$ is computing the set of *possible* answers, attempting to replace our $\cdot\downharpoonleft_{\mathrm{HR}}$ with $\cdot\downharpoonleft^{@}_{\mathrm{HR}}$ is unlikely to help matters; we are not interested in the number of *possible* rule answers that give rise to each result, but rather in the possible range of multiplicities that will be seen at runtime, a fundamentally different question.

[130] It also provides no (non-tautological) upper bounds on $\mathfrak{K}$. For the moment, we continue not to do so; later analyses will investigate the space of permitted queries within a program more precisely (§5.3).

The first of these challenges can readily be overcome at the cost of some imprecision in analysis. Rather than use τ directly, we can *coarsen* τ to permit such a finite partition; that is, we can use some $\tau' = \Sigma_{\eta \in H} \bigcup_{h \in \eta} \tau_h \supseteq \tau$, with H a finite partition of $\tau\downharpoonleft_1$. By varying the size of the partition H, one gets coarser or finer approximations of τ. In the most extreme, but simplest, case, take $H = \{\tau\downharpoonleft_1\}$, so $\tau' = \langle\tau\downharpoonleft_1, \tau\downharpoonleft_2\rangle$. Because μDyna rules tend to have a good bit of programmer-provided structure in their HEADs, this may not be as poor of an approximation as it might seem: $\rho\downharpoonleft_{\text{HEAD}} \supseteq \tau\downharpoonleft_1$ might be a reasonably "small" set of items all of which may, indeed, generally behave similarly.

Before truly addressing the second challenge, let us ponder the most naïve solution and see it through to the end. Having generated τ' as above, we see that each head h in each η in H are treated uniformly, just as we had at runtime in §3.3. Knowing nothing about multiplicities of results, we can nevertheless say that each element of $\tau_\eta = \bigcup_{h \in \eta} \tau_h$ occurs somewhere (inclusively) between 0 and ∞ many times. Despite the wide range of uncertainty, it may still be possible to draw some conclusions: for example, if all input values are integral, then their aggregation by summation is also certainly integral, NULL (in the case of 0 of each), or ∞.[131] We find ourselves wanting an approximation of the behavior of each aggregator f, a function $\text{apxagg}_f \in \wp\mathcal{H} \to \wp\mathcal{H}'$ ("*ap*pro*x*imate *agg*regation") such that for any set $\tau \subseteq \mathcal{H}$ and any *bag* $\beta \in \wp_+\bar{\mathfrak{U}}_\infty\tau$ (i.e., such that $\mathfrak{U}\beta \subseteq \tau$) we have that $f(\beta) \in \text{apxagg}_f(\tau)$. Our observation above on summation thus manifests as constraining $a_{\sum}(\tau) \subseteq \mathbb{N} \cup \{\text{NULL}, \infty\}$ for any $\tau \subseteq \mathbb{N} \cup \{\infty\}$. (This is not a complete definition, as we might also wish to special case $a_{\sum}(\{0\}) = \{\text{NULL}, 0\}$, for example.)

Armed with apxagg_f for a rule's aggregator, f,[132] and, supposing, for the moment, that there are no other rules in the program, we can process $\tau = \text{refrule}_{\text{type}}(r, n_r)$, and in particular its HR projection, into a more-directly useful quantity. Assuming, as above, that $\tau\downharpoonleft_{\text{HR}} \subseteq \Sigma_{\eta \in H} \tau_\eta$, with H a finite partition of $\tau\downharpoonleft_{\text{HEAD}}$ and τ_η defined for each $\eta \in H$, then $\Sigma_{\eta \in H} \text{apxagg}_f(\tau_\eta)$ is the set of items paired with *their possible values, post-aggregation.* In order for $\mathfrak{D}$ to truly be the set of possible answers, we will *require* that it be sufficiently large that $\Sigma_{\eta \in H} \text{apxagg}_f(\tau_\eta) \subseteq \mathfrak{D}$. This, as is so often the case, is a recursive definition: τ_η is defined as an upper bound on a projection of a subset of $\text{refrule}_{\text{type}}(\cdot, \cdot)$, which is in turn defined by $\mathfrak{D}$. Because we are not using equalities, but rather subset bounds, we cannot appeal to the usual notion of *fixed-point* solutions to recursive equations, but we can instead require that $\mathfrak{D}$ be *closed* under this operation of computing $\text{refrule}_{\text{type}}(\cdot, \cdot)$ and then $\text{apxagg}_f(\tau_\eta)$.

In the above, we confined ourselves to a single rule. Given multiple rules, we can redefine the early quantities *per-rule* (by adding r indexes): $\tau_r = \text{refrule}_{\text{type}}(r, n_r)$ and $\tau_r\downharpoonleft_{\text{HR}} \subseteq \Sigma_{\eta \in H_r} \tau_{r,\eta}$, with H_r and $\tau_{r,\eta}$ defined as above. However, while we could compute each $\beta_{r,\eta} = \text{apxagg}_f(\tau_{r,\eta})$ separately, we lack any good option for combining the resulting *partial aggregations*. The only tool at our disposal for manipulating sets of aggregands is apxagg_f, but we know that, for each head $h \in \eta$, *only one* value from each $\beta_{r,\eta}$ will

[131] For simplicity, here and probably in the eventual Dyna standard library, we take the sum of infinitely many non-zero values to be the *single* ∞, which is of ambiguous sign, i.e., neither positive nor negative. We neglect any more sophisticated definitions like ζ-summation (e.g., Euler [56]).

[132] Recall that, while μDyna formally assigns each *item* an aggregator, we required in §3.3.2.1 that all items in the head of any rule were assigned the same aggregator, and so we may sensibly speak of the rule's aggregator.

actually arise, so our “0 to ∞” reasoning within apxagg_f is rather loose. Instead, we follow, by analogy, the disjoint-answers work done in §3.3.3. We can collect the possible answers across all rules by computing a partition H_Ξ of $\bigcup_{r \in \Xi} \tau_r{\downharpoonleft_{\text{HEAD}}}$ which associates each $\eta \in H_\Xi$ with the union of the appropriate bins from each rule's partition. That is, we take H_Ξ such that $\forall_{\eta \in H_\Xi, r \in \Xi}((\eta \cap \tau_r{\downharpoonleft_{\text{HEAD}}} = \varnothing) \vee (\exists!_{\eta' \in H_r}\, \eta \subseteq \eta'))$, and associate each η with $\tau_\eta = \bigcup\{\tau_{r,\eta'} \mid r \in \Xi, \eta' \in H_r, \eta \subseteq \eta'\}$. (Rather similar to the case of $\texttt{DISJOIN}$ from block 3.2, in §3.3.3, we take the set of possible contributions to items outside a rule answer's head to be $\varnothing$.) Further, we require that all items in each $\eta \in H_\Xi$ have the same aggregator, $\text{aggr}(\eta) = \text{selt}(\text{aggr}(h) \mid h \in \eta)$. From this, we can conclude that $\mathfrak{D}' = \Sigma_{\eta \in H_\Xi}\, a_{\text{aggr}(\eta)}(\tau_\eta)$ is an upper bound on the set of answers. As before, we would require that $\mathfrak{D}$ be closed under the operation that sends $\mathfrak{D}$ to $\mathfrak{D}'$.

5.2.4.1 Cardinality Estimation for Improved Answer Closure

In the preceeding, we assumed that *any* bag of aggregands of the right type could arise when it actually came time, at runtime, within the machinery of §3, to aggregate results from rules. While *correct*, such reasoning is quite limited: it cannot exclude the empty bag (and so a NULL result), and it cannot exclude infinite bags (and so ∞ as a result when summing). Let us now spend a little time considering one approach to a tighter analysis.

We will want to know the answer to questions of the form “given a set τ and a tuple of paths $\langle \pi_1, \ldots, \pi_k \rangle$, how many $t \in \tau$ are there for each element of $\langle \tau{\downharpoonleft_{\pi_1}}, \ldots, \tau{\downharpoonleft_{\pi_k}} \rangle$?” Such questions are of a kind with the AnswerFor oracle assumed in §3.4.1 and §3.4.3.6. Thankfully, while AnswerFor needed to be exact, so that we would get the right answer, here, we can tolerate some imprecision as, again, we are only interested in upper-bounding *possible* answers. Thus, rather than a single multiplicity, we expect our question to be answered with a *set* of multiplicities, e.g., “There are (inclusively) between 1 and 30 such $t \in \tau$ for each $\langle \cdots \rangle$.” Let us proceed by example before giving a formalism for such reasoning.

Even the simplest μDyna rule, $\{(\texttt{a}\langle\rangle \hookleftarrow 2) \Leftarrow \langle\rangle\}$, provides an opportunity for useful static analysis. In any program containing this rule, $\texttt{a}\langle\rangle$ will never have an empty bag of aggregands. We would like to conclude, in fact, that at least 1 copy of 2 is always present (there may be more, from other rules). As most aggregators yield NULL only when applied to $\varnothing$, this is sufficient to exclude NULL from the space of possible results. We would wish to draw similar conclusions for *each* $\texttt{a}\langle\tau\rangle$ given a rule $\{(\texttt{a}\langle x\rangle \hookleftarrow 2) \Leftarrow \langle\rangle \mid x \in \tau\}$.

In the presence of subgoals, several things must be considered:

① If the head and result are independent of subgoals, e.g., $\{(\texttt{a}\langle\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{r}\langle x\rangle \mapsto v\rangle \mid x \in \tau, v\}$, we wish to know the cardinality of the rule answers (i.e., of the subgoals). The rule just given contributes $\Lbag 1@m \Rbag$ to $\texttt{a}\langle\rangle$ assuming that the *subset* of $\texttt{r}\langle\tau\rangle$ given a non-NULL value has cardinality m. If we knew something about $\texttt{r}^{/1}$, we may be able to bound m; perhaps we know it is total ($m \in \{|\tau|\}$), a “one-hot” predicate ($m \in \{1\}$), or that it has only finite support ($m \in \mathbb{N}_\infty \setminus \{\infty\} = \mathbb{N}$).

② If the head covaries with subgoals, as in $\{(\texttt{a}\langle x\rangle \hookleftarrow 1) \Leftarrow \langle \texttt{r}\langle x, y\rangle \mapsto \_\rangle \mid x \in \tau, y \in \sigma\}$, then the cardinality of the subgoal keys themselves are no longer interesting, but rather the cardinality of subgoal keys *conditioned on* some already being specified. In the example, we wish to know how many y exist for any choice of x. We may wish to split the domain of x, i.e., τ, to allow finer distinctions; e.g., perhaps only some τ are associated with any ys at all.

③ If the value is non-constant, such as in $\{(\mathtt{a}\langle x\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{r}\langle x,y\rangle \mapsto v\rangle \mid x \in \tau, y \in \sigma, v\}$, we additionally want to partition our cardinality estimates by values: that is, we want to know *for each head, for each value*, how many rule answers may occur?

Let us presume that we can approximate (i.e., bound, both from above and from below) the *number* of answer *terms*—as opposed to potentially non-ground *types*—that will return from a subgoal query. More specifically, let us assume that we can *condition* on some paths into the subgoal, so that we are asking about the bounds on the number of answers to each subgoal when these indicated paths are taken to have singleton projections from the query (and therefore have singleton projections from the union of all answers). Given a query κ, we know that its answers will be from $\kappa \cap \mathfrak{D}$. While we could manipulate formal statements of the form "κ conditioned on the paths $\vec{\pi}$ returns (inclusively) between n and m answers from $\kappa \cap \mathfrak{D}$," it costs us little to instead manipulate multiple, concatenative, bounds on different subsets, as in "…returns (inclusively) between n_1 and m_1 answers from $\alpha_1 \subseteq \kappa \cap \mathfrak{D}$ and (inclusively) between n_2 and m_2 answers from $\alpha_2 \subseteq \kappa \cap \mathfrak{D}$ and …." We need not require disjointness of the $\{\alpha_i \mid i\}$. While we should be considering answer streams without repeated elements (as returning the same answer twice would do nothing, given the semantics' and algorithms' use of answer *sets*), for simplicity we consider samples taken with replacement.

Evidently, we will be manipulating pairs of $\mathbb{N}_\infty$ representing upper and lower bounds. It behoves us to define a type for such pairs, with the second not smaller than the first: $\mathcal{B} = \Sigma_{n \in \mathbb{N}_\infty}\{m \in \mathbb{N}_\infty \mid m \geq n\}$. Earlier, our aggregator-behavior-bounding function a_f was an element of $\wp\mathcal{H} \to \wp\mathcal{H}'$; we now wish for one which can consume multiplicity bounds instead, i.e., elements of $\langle \mathcal{B}, \wp\mathcal{H}\rangle \to \wp\mathcal{H}'$ such that $\text{apxagg}_f(\langle\langle n,m\rangle,\alpha\rangle) \ni x$ if there exists a bag $\beta \in \wp_+\bar{\mathfrak{U}}_\infty\alpha$ such that $n \leq |\beta| \leq m$ and $f(\beta) = x$. (If $\alpha = \varnothing$, then the only β is $\varnothing$, too, and so the only x is NULL, regardless of the multiplicity bounds.) The concatenative bounds above are seen to be *finite bags* of these kinds of bounds-annotated sets; we can extend the definition of apxagg_f to handle these, too: take $\text{apxagg}_f(\wr\langle b_i,\alpha_i\rangle \mid i \in \mathbb{N}_1^k \int) \ni x$ if there exist $\{x_i \in \text{apxagg}_f(\langle b_i,\alpha_i\rangle) \mid i \in \mathbb{N}_1^k\}$ such that $f(\wr x_i \mid i \in \mathbb{N}_1^k \int) = x$. (We have exploited the AC-reducer property of f: we reduced sub-bags of the possible results implied by the concatenative bounded-sets and then reduced the bag of results therefrom.)

Assuming we could obtain such an object describing the RES projection of the rule answer set (with singleton head projection), we would be in a grand position to bound the possible aggregands to that head. One possible method for obtaining such would be to augment the definition of $\text{refrule}_{\text{type}}(r, n_r)$ such that each refinement by $\mathfrak{D}$ bounded the number of terms would survive at runtime (i.e., were further considered by the search tree formed by the solver's REFINERULESUFFIX), assuming that the refinements to the left had reduced the subgoals to the left to *singletons*.[133] We presume the existence of a function $\text{sgc}(\kappa, \vec{\pi})$ ("*s*ub*g*oal *c*ardinality") which can return bounds that correctly describe the behavior of the query κ when the components at paths in $\vec{\pi}$ are fixed (i.e., are grounded). More specifically,

[133] That is, we are interested in bounding the out-degree of all nodes in the search tree corresponding to the i^{th} subgoal.

what we mean is that, if $\text{sgc}(\kappa, \vec{\pi}) = \lbag \langle \langle n_i, m_i \rangle, \alpha_i \rangle \mid i \rbag$, with $\{\alpha_i \mid i\}$ *disjoint*, then, for any choice of $\vec{t}$ where each $t_i \in \kappa{\downharpoonleft_{\pi_i}}$, the answer obtained from $\texttt{Lookup}(\kappa[\{t_1\}\pi_1/\cdots][t_k/\pi_k])$, where $k = \text{tlen}(\vec{\pi})$, must *describe* a set of pairs of items and their values containing between n_i and m_i pairs from α_i. (In the case of the ground reasoning of §3.2, the description is overt: $\texttt{Lookup}$ returns a finite set of items paired with their values. In the non-ground systems of §3.3 and §3.4, $\texttt{Lookup}$ returns finite encodings of infinite sets of items paired with their values and some decoding is necessary to recover the set considered here.)[134]

We can now simulate the search tree of $\texttt{computeRule}$ on the rule r and query κ using this sgc function: each $\langle b, \alpha \rangle$ in the return of sgc forms a branch of this tree. Letting $\vec{b}$ and $\vec{\alpha}$ denote the bounds and sets along a root-to-leaf path within this tree, the leaf associates $\rho_r[\kappa/\text{HEAD}][\alpha_1/\text{SG}.1]\cdots[\alpha_{n_r}/\text{SG}.n_r]{\downharpoonleft_{\text{RES}}}$ with bounds formed by the *monoid product* of the bounds $\vec{b}$, where $\langle n_1, m_1 \rangle \otimes \langle n_2, m_2 \rangle = \langle n_1 * n_2, m_1 * m_2 \rangle$ and the identity is $\langle 1, 1 \rangle$. The paths $\vec{\pi}$ for each call to sgc are obtained by considering equality constraints within ρ_r. In a rule like $\{(\mathtt{f}\langle x \rangle \hookleftarrow v) \Leftarrow \langle \mathtt{g}\langle x, y \rangle \mapsto a, \mathtt{h}\langle c, y \rangle \mapsto a, \cdots \rangle \mid \cdots\}$, the first call to sgc, at the root of the search tree, will take $\vec{\pi} = \langle 1.1 \rangle$ (the first child of the key component of the kv-pair), corresponding to the variable x, which is presumed to be assigned by the HEAD; the second call will take $\vec{\pi} = \langle 1.2, 2 \rangle$, corresponding to the variables y and a, which have both been set by SG.1. The *bag* containing all the leaf-associated bounds forms a concatenative bound on the aggregands this rule contributes to each element of κ and may be combined with the results of this analysis on other rules and then reduced as above.

Example 51: Let us revisit the four example rules we gave at the start of this section to briefly study the system we have just given.

⓪ For $\{(\mathtt{a}\langle\rangle \hookleftarrow 2) \Leftarrow \langle\rangle\}$, there are no subgoals and so the entire "search tree" collapses to being just the root, leaving us with $\lbag \langle \{2\}, \langle 1, 1 \rangle \rangle \rbag$ as the only bound.

① For $\{(\mathtt{a}\langle\rangle \hookleftarrow 1) \Leftarrow \langle \mathtt{r}\langle x \rangle \mapsto v \rangle \mid x \in \tau, v\}$, there is only one internal node in the search tree; if $\text{sgc}(\mathtt{r}\langle\tau\rangle, \langle\rangle)$ returns $\lbag \langle \alpha_i, b_i \rangle \mid i \rbag$, then the result of the above simulation of the search tree is $\lbag \langle \{1\}, b_i \rangle \mid i \rbag$.

② In $\{(\mathtt{a}\langle x \rangle \hookleftarrow 1) \Leftarrow \langle \mathtt{r}\langle x, y \rangle \mapsto \_ \rangle \mid x \in \tau, y \in \sigma\}$, there is still just one internal node, but now we use $\text{sgc}(\mathtt{r}\langle\tau, \sigma\rangle, \langle 1.1 \rangle)$, capturing the knowledge that x has been set by the head.

③ When the value is not constant, as in $\{(\mathtt{a}\langle x \rangle \hookleftarrow v) \Leftarrow \langle \mathtt{r}\langle x, y \rangle \mapsto v \rangle \mid x \in \tau, y \in \sigma, v\}$, the result of the RES projection at the leaves of the simulation above will depend upon the α on the root-to-leaf path under consideration. We can see that the system properly transfers bounds on the *number of answers* to the $\mathtt{r}^{/2}$ subgoal into bounds on the *number of aggregands* contributed to the head.

◊

[134] To stave off concerns about vacuity of definitions, one always has the option of defining $\text{sgc}(\kappa, \vec{\pi}) = \lbag \langle \mathfrak{O} \cap \kappa, \langle 0, \infty \rangle \rangle \rbag$, which recovers the "0-to-∞" behavior of the earlier system.

5.2.5 Aside: Constancy

It is worthwhile to track an *under-estimate* of the set of queries whose answers will *never* be revised within a solver's runtime. Items which are guaranteed to have only one possible value are said to be **constant**. Such items include any inputs that are promised to be set once (in a sense, prior to the solver's startup). In this class we find things like the definitions of arithmetic operators. Not all constant items need be quite so Platonic, however: even user-specified parameters which vary over time may be included, provided that no *instance* of the solver sees the variance.

Once we move from $\mathcal{I}_{\text{inp}}$ to $\mathcal{I}_{\text{der}}$, the notion of constancy bifurcates. Because the solver may, as per §2.5.2, *guess* a derived value and *revise its guess* later, even items which are, mathematically, guaranteed to have only one fixed-point solution might still trigger forward-chaining. Still, there is likely some utility to acknowledging that there is exactly one *converged* value of these items; we call such items **constant-at-convergence**. Any items which are *acyclically* defined solely by constant-at-convergence items are themselves constant-at-convergence, thanks to the guarantee of unique solutions to expression trees.[135] Cyclically defined items are constant only when it can be shown that there is a unique fixed-point to all cycles participating in the definition (i.e., including transitive ancestors); we generally consider such analysis to be not worth-while.

Such constancy-at-convergence is, however, of little utility to our eventual goal of code generation, wherein we might have hoped that an item being constant meant that we could elide forward-chaining machinery for it. We call the subset of constant-at-convergence items whose values are guaranteed to never be revised, even by guessing, **immediately-available**. All constant $\mathcal{I}_{\text{inp}}$ items are immediately-available. Those *acyclically-defined* $\mathcal{I}_{\text{der}}$ built up exclusively from immediately-available items are so as well, so long as the solver promises to not guess their value during backward-chaining. These items will not need forward-chaining machinery generated for them.

5.2.6 Related Work

Refinement Types Our membership constraints are a kind of "refinement types" (see, e.g., Freeman and Pfenning [69], Lovas [118], and Vazou et al. [182]). Given a type $[\![\sigma, \text{cby}]\!]$, cby generates formulae which hold of elements of the unrefined (presumably, TA-based) type σ. These refinements are *conditional*, whereas typical refinement types are typically not (e.g., the standard example of a refinement type is the set of even naturals $\{n \in \mathbb{N} \mid n \text{ even}\}$, where the predicate "$n$ even" holds of all members of the subset). Dunfield [46, §2.3.5] introduces a notion of "guarded types" which convey type information only when some predicate holds; the first example of such given therein is that of a list type constructor, which takes the length of the desired lists and, as such, is only sensibly typed when this argument is ≥ 0.

Prolog Type Systems Hill and Topor [94] describes a Mycroft-O'Keefe-style system for SLDNF and incorporates a notion of subtyping. In the course of this work (Ex. 1.4.13,

[135] One must, however, be careful computationally, when inexact values are in play. To be sure of recovering the same result, one must use the computational operations *in the same order* every time.

in particular), it is shown that, in the presence of set-theoretic ("inclusion") sub-typing, that, essentially, it is unsound to erase type information at runtime. In Dyna, this amounts to requiring that our runtime system indeed represent the type of free variables (i.e., of non-singleton projections of θ and ϵ) so that Lookup and refinement behave according to their set-centric specifications.[136]

Schrijvers et al. [162] attempts to bring type systems of other languages into Prolog (SWI and YAP, in particular). While it glosses over some details, it does have the usual features one might expect of a modern type system: prenex polymorphism, recursive types, a higher-order `pred` type, and `any` for $\mathcal{H}$. It remains in the Mycroft-O'Keefe school of type systems.

Hadjichristodoulou [86] describes another, recent type system for Prolog. It extends the traditional Mycroft-O'Keefe systems with universally- and existentially-quantified subtypes, giving a sense of "directionality" to types. Universally-quantified subtypes are used to indicate that rules invoking a subgoal get to refine the type as they wish, while existentially-quantified subtypes are used to convey that the subgoal may *produce* any subset of the upper bound. The intuition is that predicates like list append should be assigned universally-quantified types while scans of the input database should be assigned existentially-quantified types. This directionality is intended to ensure that the recipient of a term, be that either the subgoal definition or the successive subgoals, is prepared to deal with any eventualities. Our system instead assigns, essentially, *two* types to subgoals, namely, refinement by $\mathfrak{O}$ and by $\mathfrak{K}$; from these two we have a similar form of directionality, without needing to bring additional quantifiers into play. A stronger sense of procedural execution, and of preparedness for eventualities especially, for our system is forthcoming in §5.3; in particular, we may *generate new subgoals* to ensure that information flow is well-typed.

Prolog Type Systems using Tree Automata We are not the first to recognize the applicability of tree automata to analysing the grounding set of a Prolog program. As early as 1990, Heintze and Jaffar [89] recognized the utility of finitely-presented, recursive descriptions, i.e., (non-deterministic) tree automata, for (optimistic) types for Prolog programs. Frühwirth et al. [70] uses a restricted form of Prolog as the meta-language for describing (optimistic) types for Prolog programs; the restrictions give these programs precisely the expressive power of regular sets and the algorithms given make use of "alternating" tree automata [165] (the curious reader also directed to Comon et al. [35, Ch. 7]; theorem 7.4.1 therein shows that alternating tree automata are exactly as expressive as regular bottom-up tree automata). Talbot, Tison, and Devienne [171] builds on Frühwirth et al. [70] and explicitly uses tree automata in its implementation.

[136] In particular, what Hill and Topor [94] shows in Ex. 1.4.13 is that erasing type information and using an "untyped" runtime as provided by, e.g., the WAM (§4.1) results in additional answers that should be excluded by the type of the query.

5.3 Planning Conjunctions

The analysis considered so far concerns itself with approximating the circuit described by a μDyna program, rather than on the particular actions taken by a solver algorithm. As brought up in §3.2.1.1, however, when a solver algorithm goes to evaluate the impact of a rule in a program (and, in particular, on a query set of items), steps must have been taken in advance to ensure that this computation is *possible* and *terminates*.

We can construct a problematic example without leaving the confines of our high-level language. Consider the two Prolog rules

B. 5.1

```
length(nil,0).
length(L,N) :- L = cons(_,T), length(T,M), N is M + 1, N >= 0.
```

While this appears to be a sensible definition for the length of a list built up of `nil`$^{/0}$ and `cons`$^{/2}$ functors, it works only when *given* a list. That is, if invoked as the subgoal `length(cons(a,cons(B,cons(c,nil))),N)`, the recursive evaluation of backward-chaining will leave `B` unbound and will bind `N` to `3`. If invoked as `length(L,1)`, we would expect to obtain a finite stream containing solely the answer that `L` is bound to `cons(X,nil)` (for some *fresh* variable `X`). Instead, because Prolog always executes its subgoals left-to-right, we find that `length(L,1)` will call `length(T,M)` with both `T` and `M` free; while the initial call will, indeed, return the answer `length(cons(X,nil),1)`, it will then *not terminate*, forever searching for another way to build a length-1 list. While any individual element of the set represented by `length(T,M)` is a sensible query, as are those for which only one of the arguments is ground, the entire set represents a query which is *certain* to have infinitely many answers for which there is no finite stream representation.[137] In fact, there is no single order of subgoals such that both **query modes** of `length`$^{/2}$ queries (list-known or length-known) can be supported. In a Prolog standard library, *meta-logical* tests are invoked to probe whether the arguments to a `length`$^{/2}$ query are variables or bound.[138] In the interest of retaining the *declarative* nature of the language, we would like *the compiler* (rather than the programmer) to generate different code for the different query modes and to use analysis of the program to assign a **call-compatible** implementation to each subgoal invocation.

Call-compatibility concerns also arise at the "procedural fringe" of the language. Some **built-in** items within $\mathcal{I}_{\text{inp}}$—for example, those describing arithmetic facts—are handled specially by the solver. The solver in this case returns a stream of results computed by some *procedural* implementation rather than by program rules. A Prolog solver may refuse to answer certain queries of built-in items, often because the answer streams would be in-

[137] The answer `length(cons(X,nil),1)` to the query `length(L,1)` also represents infinitely many answers (namely, each substitution for `X`), but the stream itself is finite.

[138] The curious reader is pointed at, for example, the SWI Prolog implementation at http://www.swi-prolog.org/pldoc/doc/_SWI_/boot/init.pl?show=src#length/2 or the XSB Prolog implementation at https://sourceforge.net/p/xsb/src/HEAD/tree/trunk/XSB/lib/lists.P. Both use Prolog's *cut control-flow operator* (written as the subgoal "`!`") to explicitly prune the search tree, achieving an "if-then-else" *partitioning* of search strategies based on whether the list or the length is known. The implementation in https://sourceforge.net/p/xsb/src/HEAD/tree//trunk/XSB/syslib/basics.P uses an explicit if-then-else construction which relies, underlyingly, on similar mechanics. All implementation use meta-logical predicates such as `var`$^{/1}$, which is true only when its argument is currently uninstantiated (i.e., a variable), or `integer`$^{/1}$, which is true only when its argument is instantiated to an integer.

finitely long and would cause the solver to not terminate. Consider executing the (Prolog) subgoal sequence `B=2, C=1, A is B+C`: by the time the solver invokes the $\texttt{is}^{/2}$ subgoal query, that query has been specialized to `A is 2+1`, which easily returns $\alpha = \{\texttt{3 is 2+1}\}$. If, however, the subgoal sequence had been reversed, the *un-specialized* $\texttt{is}^{/2}$ query would have been obligated to produce an infinite answer summarizing *all addition facts*. Encoding that answer into a finite stream (so that the solver terminates) would require a highly expressive representation for non-ground terms (an approach adopted in constraint logic programming, e.g., Colmerauer [33]). Most Prolog implementations instead generate an **instantiation error** (a runtime exception) if this kind of problematic query arises at runtime. Again, to retain the declarative nature of the language, we reject the notion of an instantiation error that arises only because of procedural concerns (e.g., subgoal ordering); either the program can be transformed into executable form or *the program* is ill-specified (e.g., for some query, there is no suitable subgoal order) and should be rejected by the compiler.

Static analysis of programs and reordering of subgoals can *rule out* exceptional cases that might arise at runtime. The compiler can, with such static work, ensure that the two query modes of $\texttt{length}^{/2}$ above always operate inductively, by destructing on a given list (spine) or by decrementing a natural number. Similarly, it can ensure that, within the execution of a query mode, there will be no instantiation-faulting subgoals. These guarantees permit the use of *specialized* code to implement particular query modes; conditions known to be statically true are left *untested* in these specializations, typically resulting in performance gains relative to a generic runtime implementation.

Prolog static analysis of this form has a rich history, stretching back decades: Apt and Marchiori [12] claims that "modes" as we know them are due to Mellish [123]. Our effort follows most closely that of the Mercury project [124], and Overton [137] in particular, and can be thought of as a set-centric retelling of this work as applied to weighted logic languages. Working on sets directly gives us some generalization power, and obviates the need for a separate notion of variable aliasing within the analysis (see, e.g., Overton [137, §5]), as that is handled by the underlying tree (set) automata;[139] however, such gains come at the expense of tractability.[140]

5.3.1 Procedures, Instantiation States, and Contents

Recall that Lookup from §3.3 and §3.4 took a *set of* kv-pairs encoding a query—a request for item names and associated values—and, in turn, returned zero or more *sets* of kv-pairs encoding the corresponding answers—those item names and associated values that were, at

[139] However, as pointed out in Overton [137, §5.4.1], once one begins to consider a particular solver runtime system, it may, in fact, be necessary to augment analysis to track whether certain positions within a runtime type are subject to equality constraints or not. Certain operations within the solver—certain, typically *faster*, special cases of unification, especially—may require that a subterm be *independent* from other subterms, and so must only be applied at positions π with singleton projections or devoid of constraints that reference trees both within and without π. For present purposes, we are ignoring this possibility and assume that the *runtime* automata implementation will sort out anything asked of it.

[140] Throughout Overton [137], the languages used for abstract interpretation are given set-theoretic concrete readings. As mentioned in §5.1, we do most of our work in the concrete domain and leave to the future the task of designing suitable abstract representations.

least at the present point in the program's evolution, true. We gave $\textsc{Lookup}$ a complicated, dependent function type with domain of $\wp(\langle\mathcal{H},\mathcal{H}\rangle)$ and presumed that it could answer *any* query we gave it (subject to some informally-defined notion of planning). As we see from the above examples, this presumption of a *single* $\textsc{Lookup}$ function is too imprecise; we must give more specific **procedures** which handle particular "shapes" of queries, i.e., particular *subsets* of $\wp(\langle\mathcal{H},\mathcal{H}\rangle)$.

The relevant attribute of a subgoal under consideration in both cases above is its **instantiation state**. (Following Overton [137], we use the shorthand **inst**.) That is, we wish to know *which* parts of a subgoal query are *known* and which are uncertain (and, when uncertain, *how* uncertain, i.e., over what domain). The known aspects can be *pattern-matched* within rules, used by the "procedural fringe" items, and even used by *conditional behavior* within the solver. If a subgoal is *entirely known* (i.e., is a singleton subset of $\langle\mathcal{H},\mathcal{H}\rangle$), the solver's action reduces to confirming that the given item (key) indeed has the given value (or returning the empty stream, causing $\textsc{refineRuleSuffix}$ to consider other rule queries, if any).[141] Uncertain aspects of the query, on the other hand, must be explored. An uncertainty in the value component may cause the solver to have to compute, rather than merely confirm, an item's value. An uncertainty in the *key* component will cause the solver to find the *set of* items and associated values that match the query.

In Prolog, an instantiation state describes which variables have become *bound* to other structures (which may, in turn, involve more variables); one can think of insts as being properties of some input rule (or term, more generally) and the *substitutions* made thereupon. In our set-centric retelling, we are interested in knowing when every tree in a type shares a root functor (e.g., $\mathtt{f}\langle\tau_1,\ldots\rangle$) or when the projection along a given path is a singleton set (e.g., $\cdot\downharpoonleft_1$ applied to $\mathtt{f}\langle\{t\},\tau_2,\ldots\rangle$).

The especially prognosticative reader may, in light of the above, anticipate our reintroduction of our Tree Set Automata, or at least their transition rules, from §4.4. Each set constructed (from other sets) by a BOUND $\mathtt{f}\ \langle\tau_1,\ldots,\tau_n\rangle$ (or BOUND$_\subseteq$ $\mathtt{f}\ \langle\tau_1,\ldots,\tau_n\rangle$) transition rule contains exclusively trees rooted by $\mathtt{f}^{/n} \in \mathcal{F}$. A GROUND transition from a tree state to a set state generates singleton sets; when combined by BOUND transitions, projection along appropriate paths remains a singleton set (or becomes empty, as might happen if the empty set is selected elsewhere within the BOUND transition, or as might happen in an IRTSA when global constraints are taken into consideration). FREE and SUBTYPE transitions encode two different degrees of uncertainty about subterms. A FREE τ represents a subterm *certain to be otherwise unbound*, i.e., a free variable, while SUBTYPE τ represents not only uncertainty *within* the query, but even uncertainty *of* the exact shape of the query. Thus, we see that the form of our TSA of §4.4.1 and §4.4.2 are well-suited for encoding the kind of knowledge we require about subgoals when we attempt to **dispatch** to a sufficiently-capable procedure.

In order to reason inductively about $\textsc{refineRuleSuffix}$, we require knowing not just what a query procedure can take as input, but the kinds of answer streams it might

[141] It is also not incorrect for the solver to return a stream containing only *empty sets* or even sets of kv-pairs whose intersection with the query is empty; both of these alternatives would cause the set of potential rule answers to become empty, again triggering $\textsc{refineRuleSuffix}$ to move on. While correct, we generally consider these to be inferior alternatives to merely returning an empty stream.

return. There are numerous dimensions we may care about here, but chief among them is the shape of the answers within the stream. That is, we care about the *instantiation states* of the answers as well. In Overton [137] and most Prolog mode analysis, each procedure is associated with its own possible output instantiation states. However, because we view our program as specifying a *map* from items to values, we believe that, given a program and a query instantiation state, there is only one possible corresponding answer instantiation state. That is, we believe that it is more appropriate to describe the map itself with an instantiation state, called the **contents** $\mathfrak{C}$.[142] Then, the answers to any query set κ given to Lookup will certainly be from the abstract unification of $\{\kappa\}$ and $\mathfrak{C}$, i.e., $\{\kappa\} \boxtimes \mathfrak{C}$. During analysis, when κ is unknown, we instead know the query instantiation state $K \ni \kappa$, and so model the answers as coming from $K \boxtimes \mathfrak{C}$.

Our analysis mimics that of our grounding set analysis (§5.2). Just as that analysis assumed access to an upper-bounding answer set α, we assume access to the upper-bounding answer *inst* $\mathfrak{C}$. So armed, we approximate the behavior of `computeRule`$(r,\kappa \in K)$ with a function which returns the *set of possible rule answers* (i.e., the instantiation states of the μDyna rule ρ_r) after processing the first i subgoals. As before, we do this by approximating the action of the recursive `refineRuleSuffix` core of `computeRule`. At the beginning of execution of the rule, there is *precisely one* possible answer set, ρ_r itself, i.e., the instantiation state is FREE ρ_r. We then specialize by the knowledge of possible query heads $K\downharpoonleft_1$ to obtain the instantiation state after i subgoals have executed:

$$\text{refrule}_{\text{inst}}(r, K, i) \overset{\text{def}}{=} \{\rho_r\}[K\downharpoonleft_1 /\!\!/ \text{HEAD}][\mathfrak{C} /\!\!/ \text{SG}.1]\cdots[\mathfrak{C} /\!\!/ \text{SG}.i] \tag{5.1}$$

In such a state, we wish to find a procedure for the $(i{+}1)^{\text{th}}$ subgoal, whose instantiation state is $\text{refrule}_{\text{inst}}(r, K, i)\downharpoonleft_{\text{SG}.i+1}$. $\text{refrule}_{\text{inst}}$ is an approximation of `refineRuleSuffix` in that every invocation of its success callback (`contribRuleAnswer` in listing 3.2, `applyV` in listing 3.3) is an element of $\text{refrule}_{\text{inst}}(r, K, n_r)\downharpoonleft_{\text{RES}}$.

There is nothing special about the left-to-right order; it is merely syntactically convenient. We could, instead, speak of a tuple of subgoal indices having been visited:

$$\text{refrule}_{\text{inst}}(r, K, \vec{v}) \overset{\text{def}}{=} \{\rho_r\}[K\downharpoonleft_1 /\!\!/ \text{HEAD}][\mathfrak{C} /\!\!/ \text{SG}.v_1]\cdots[\mathfrak{C} /\!\!/ \text{SG}.v_{\text{tlen}(\vec{v})}]. \tag{5.2}$$

The instantiaton state of the i^{th} subgoal having answered the subgoals indiciated by $\vec{v}$ is $\text{refrule}_{\text{inst}}(r, K, \vec{v})\downharpoonleft_{\text{SG}.i}$. If ever $\text{refrule}_{\text{inst}}(r, K, \cdot)$ is $\varnothing$ or $\{\varnothing\}$, then there are certainly no answers from the rule r for the query K and we need not consider that pair further.

Now, at last, we arrive at the heart of planning. Just as $\mathfrak{C}$ is our assumed analogue to our answer type approximation α, we will assume that we have been given a **procedure table**, $\mathfrak{P}$, whose entries describe permitted subgoal query modes.[143] The task of the planner, given a rule r and a query inst K, is to find an order of the n_r subgoals thereof, such that, at the time each is made, it is *compatible* with the permitted procedures. Again, just as with

[142] We again deviate from our usual typographical convention (this time, that of using capital Roman letters for instantiation states), as we additionally feel that $\mathfrak{C}$ is sufficiently meaningful that it deserves a more distinguished symbol.

[143] For simplicity of exposition, we *identify* a query procedure and its input inst. In practice, one may wish to permit *multiple* procedures at the same inst, and so $\mathfrak{P}$ entires would become pairs of input insts and procedure names.

grounding set analysis, we assume that these K come from our permitted queries, so $K \in \mathfrak{P}$. If we take compatibility to simply be inclusion in $\mathfrak{P}$ (and insist that $\varnothing \in \mathfrak{P}$), then our formal requirement is $\forall_{r \in \Xi}\, \forall_{K \in \mathfrak{P}}\, \exists_{\vec{v} \in (\mathbb{N}_1^{nr})^{nr}} (\forall_{i \in \mathbb{N}_1^{nr}} (\forall_{i,j}\, v_i \neq v_j) \wedge \text{refrule}_{\text{inst}}(r, K, \langle v_1, \ldots, v_{i-1} \rangle) \downharpoonleft_{\text{SG}.v_i} \in \mathfrak{P})$. (The first half of the conjunct simply ensures that no subgoal is revisited.)

There is no fundamental reason that a *single* order must be used for all results, despite that we have assumed as much for simplicity of exposition. The planner is free to partition the inst $\text{refrule}_{\text{inst}}(r, K, \langle v_1, \ldots, v_{i-1} \rangle)$ and consider each subset separately; at runtime, the solver must test which case has arisen and dispatch to the appropriate remainder of the plan. Such capability might be especially of interest within an optimizing compiler, where, even if different orderings of subgoals are not required, the ability to reorder subgoals may allow the planner to generate more performant plans for one or all branches. Similar behavior is considered *within* a subgoal call, below, with the PROC-CASE rule.

5.3.2 A Little Language for Call Compatibility

What, exactly, do we mean when we say that the planner ensures that a subgoal is "compatible" with the available procedures? We mean that there is a piece of procedural code capable of taking any type within an inst and determining the subset thereof which is justified by the current state of other items within the circuit (by, ultimately, either LOOKUP of these current states in the memo table and/or backward-chaining COMPUTE-ation). In general, given a procedure for answering some inst, $I \in \mathfrak{P}$, there are several "implicit procedures" that wrap this procedure and allow it to be used to resolve subgoals in some other inst, I'. To detail this "wrapping," we construct a little procedural language, an extension of the λ-calculus, for manipulating sets of terms; we give a proof system for deriving implicit procedures within this language. Our proof system is neither deterministic nor syntactically-driven, so will, in practice, be backed by *heuristic* search.

5.3.2.1 Call Adaptor Language

The syntax of our adaptor language is one of expression trees ($\mathcal{E}$) built using the following grammar. Herein, I ranges over insts and x over variables.

$$\begin{aligned}\mathcal{E} ::= {}& x \mid \wp\mathcal{H} \mid \mathcal{E}\ (\mathcal{E}) \mid \mathtt{f}\langle \mathcal{E}, \ldots, \mathcal{E} \rangle \mid \textsc{case}\ x\ \textsc{of}\ \{I_i \mapsto \mathcal{E} \mid i\} \\ & \mid \textsc{call}\ I \mid \textsc{upcast}\ I\ I \mid (\textsc{upcast}\ I\ I)^{-1} \mid \textsc{embed}\ I\ I \mid \textsc{filter}\ \mathcal{E}\ \mathcal{E}\end{aligned}$$

The *values* (i.e., irreducible expressions) of this language are defined similarly:

$$\begin{aligned}\mathcal{V} ::= {}& \wp\mathcal{H} \mid \mathtt{f}\langle \mathcal{V}, \ldots, \mathcal{V} \rangle \mid \textsc{case}\ x\ \textsc{of}\ \{I_i \mapsto \mathcal{E} \mid i\} \\ & \mid \textsc{call}\ I \mid \textsc{upcast}\ I\ I \mid (\textsc{upcast}\ I\ I)^{-1} \mid \textsc{embed}\ I\ I\end{aligned}$$

Static semantics of this language are given by the $\vdash_{\text{pcs}}$ judgement in figure 5.1 and (big-step) dynamic semantics are given by $\vdash_{\text{pcd}} \cdot \Downarrow \cdot$ in figure 5.2. We define, as syntactic sugar, $\lambda_x e$ to mean $\textsc{case}\ x\ \textsc{of}\ \{\_ \to e\}$, that is, a case analysis with only one possible outcome. Similarly, we define function composition $f \circ g$ as sugar for $\lambda_x f\ (g\ (x))$.

$$\frac{\Phi(x) = \alpha}{\Phi \vdash_{\text{pcs}} x : \alpha}\ \textsc{pcs-var} \qquad \frac{\Phi \vdash_{\text{pcs}} e_1 : \alpha \to \diamond_1 \beta \quad \Phi \vdash_{\text{pcs}} e_2 : \diamond_2 \alpha \quad \diamond_r = \max\{\diamond_1, \diamond_2\}}{\Phi \vdash_{\text{pcs}} e_1\ (e_2) : \diamond_r \beta}\ \textsc{pcs-app}$$

$$\frac{I \in \mathfrak{P}}{\Phi \vdash_{\text{pcs}} \texttt{CALL}\ I : I \to \blacklozenge(I \boxtimes \mathfrak{C})}\ \textsc{pcs-call} \qquad \frac{I_c \subseteq \bigcup_i I_i \quad (\forall_i)\Phi \triangleleft \{x \mapsto I_c \cap I_i\} \vdash_{\text{pcs}} e_i : \diamond I}{\Phi \vdash_{\text{pcs}} \texttt{CASE}\ x\ \texttt{OF}\ \{I_i \mapsto e_i \mid i\} : I_c \to \diamond I}\ \textsc{pcs-case}$$

$$\frac{\vdash_{\text{uc}} \texttt{UPCAST}\ I\ I' : I \to I'}{\Phi \vdash_{\text{pcs}} \texttt{UPCAST}\ I\ I' : I \to I'}\ \textsc{pcs-upc} \qquad \frac{\vdash_{\text{uc}} \texttt{UPCAST}\ I\ I' : I \to I'}{\Phi \vdash_{\text{pcs}} (\texttt{UPCAST}\ I\ I')^{-1} : I' \to I}\ \textsc{pcs-dnc} \qquad \frac{\vdash_{\text{ee}} \texttt{EMBED}\ I\ I' : I \to I'}{\Phi \vdash_{\text{pcs}} \texttt{EMBED}\ I\ I' : I \to I'}\ \textsc{pcs-emb}$$

$$\frac{\texttt{f}^{/n} \in \mathcal{F} \quad (\forall_i)\Phi \vdash_{\text{pcs}} e_i : I_i}{\Phi \vdash_{\text{pcs}} \texttt{f}\langle e_1, \ldots, e_n \rangle : \text{BOUND}\ \texttt{f}\ \langle I_1, \ldots, I_n \rangle}\ \textsc{pcs-cons} \qquad \frac{\begin{array}{l}\Phi \vdash_{\text{pcs}} f : I \to \blacklozenge\{\{\langle \texttt{unify}\langle x, x\rangle, \texttt{true}\langle\rangle\rangle \mid x \in \tau\} \mid \tau \in I'\} \\ \Phi \vdash_{\text{pcs}} a : \blacklozenge I\end{array}}{\Phi \vdash_{\text{pcs}} \texttt{FILTER}\ f\ a : \blacklozenge I'}\ \textsc{pcs-filter}$$

Figure 5.1: Static semantics for our call adaptor language. $\mathfrak{C}$ and $\mathfrak{P}$ are assumed in scope for all rules, like Φ, but are constant and so suppressed to reduce clutter. α, β range over arbitrary types; I over insts; and e over expressions. $\blacklozenge$ denotes the modality of a result from a query, whereas types without modal markings are pure; $\diamond$ ranges over modalities. For the purposes of PCS-APP, pure is taken to be less than $\blacklozenge$. The modality $\blacklozenge$ is enforced in the function argument to `FILTER`. The auxiliary judgements $\vdash_{\text{uc}}$ and $\vdash_{\text{ee}}$ are defined, in §5.3.2.2, in equation (5.4) and equation (5.7).

$$\frac{}{\vdash_{\text{pcd}} v \Downarrow \{v\}}\ \textsc{pcd-refl} \qquad \frac{\begin{array}{c} e_1 \text{ not value} \quad e_2 \text{ not value} \\ \vdash_{\text{pcd}} e_1 \Downarrow V_1 \quad \vdash_{\text{pcd}} e_2 \Downarrow V_2 \quad (\forall_{v \in V_1, v' \in V_2}) \vdash_{\text{pcd}} v\ (v') \Downarrow V_{v,v'} \end{array}}{\vdash_{\text{pcd}} e_1\ (e_2) \Downarrow \bigcup_{v,v'} V_{v,v'}}\ \textsc{pcd-app}$$

$$\frac{}{\vdash_{\text{pcd}} \texttt{CALL}\ I\ (v) \Downarrow \{v\} \otimes \alpha}\ \textsc{pcd-call} \qquad \frac{(\exists!_k) \quad v_c \in I_k \quad \vdash_{\text{pcd}} e_k[v_c/x] \Downarrow V}{\vdash_{\text{pcd}} \texttt{CASE}\ x\ \texttt{OF}\ \{I_i \mapsto e_i \mid i\}\ (v_c) \Downarrow V}\ \textsc{pcd-case}$$

$$\frac{}{\vdash_{\text{pcd}} \texttt{UPCAST}\ I\ I'\ (v) \Downarrow \{v\}}\ \textsc{pcd-upc} \qquad \frac{v \in I}{\vdash_{\text{pcd}} (\texttt{UPCAST}\ I\ I')^{-1}\ (v) \Downarrow \{v\}}\ \textsc{pcd-dnc} \qquad \frac{v' \in I' \quad v \subseteq v'}{\vdash_{\text{pcd}} \texttt{EMBED}\ I\ I'\ v \Downarrow \{v'\}}\ \textsc{pcd-emb}$$

$$\frac{(\forall_i) \vdash_{\text{pcd}} e_i \Downarrow V_i}{\vdash_{\text{pcd}} \mathtt{f}\langle e_1, \ldots, e_n \rangle \Downarrow \{\mathtt{f}\langle v_1, \ldots, v_n \rangle \mid v_i \in V_i\}}\ \textsc{pcd-cons} \qquad \frac{\vdash_{\text{pcd}} e_f\ (e_r) \Downarrow V}{\vdash_{\text{pcd}} \texttt{FILTER}\ e_f\ e_r \Downarrow V\downarrow_{1.1}}\ \textsc{pcd-filter}$$

Figure 5.2: Big-step dynamic semantics for procedural language. I ranges over insts; e over expressions; and v over values. $\alpha \subseteq \mathfrak{C}$ is the set of sets of (item name and value) kv-pairs provable at the time of the call; it is presumed in scope for all rules. The judgement $\vdash_{\text{pcd}} e \Downarrow V$ reads as "the expression e reduces to the set of values V." PCD-APP collects together all possible values obtained from an application: if a sub-expression returns several answers, each will be fed through the context. In particular, PCD-CALL may yield zero or more results (and, in principle, at execution, multiple copies of the same result may be returned as well; this system is insensitive to such concerns). PCD-EMB is non-deterministic: there may be multiple suitable v' and therefore multiple possible proof trees for the same expression. Within PCD-CASE, we have made the (excessive, but simplifying) assumption that the various $\{I_i \mid i\}$ are disjoint; all that truly matters is that the system follow exactly one at runtime. These semantics are unlikely to be the best form for implementation and should be considered as more of an example or reference; in practice, one would likely wish to avoid the construction of the V sets whenever possible.

5.3.2.2 Generating Call Adaptors

We define a judgement $\mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} e : I$ which can be read as "given contents $\mathfrak{C}$ and procedure table $\mathfrak{P}$, the call compatibility language expression e answers the query with input inst I." (By definition, it yields answers in $I \otimes \mathfrak{C}$.) Below, we give **inference rules** of the form

$$\frac{x \quad y}{z}\ \textsc{l},$$

which should be read as "the inference rule L constructs a proof of z from proofs of x and y." Occasionally, we will need, as preconditions, proofs of several related assertions; we write "$(\forall\ldots)\cdots$" to indicate that each quantified set of values requires its own proof. Our inference rules tend to be named by a composition of the judgement they define (e.g., proc, uc) and some salient feature or syntax upon which they operate.

① The simplest thing that can happen is that the current subgoal instantiation state (I) is already exactly known to be possible, backed by some procedure p in the procedure table:

$$\frac{I \in \mathfrak{P}}{\mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} \texttt{CALL}\ I : I}\ \textsc{proc-call} \tag{5.3}$$

② An only-slightly-more-involved possibility is that the procedure table may have an entry capable of responding to a strict superset instantiation state $I' \supsetneq I$. In this case, we must generate not only the call, but a *wrapper* which first *upcasts* all of I into I', calls the existing procedure, and then *downcasts* the results from $I' \otimes \mathfrak{C}$ back to $I \otimes \mathfrak{C}$. This last operation, despite the apparent risk of losing an answer, is sound, as we will show in the next section.

$$\frac{I \subsetneq I'}{\vdash_{\text{uc}} \texttt{UPCAST}\ I\ I' : I \to I'}\ \textsc{uc-subset} \tag{5.4}$$

$$\frac{\vdash_{\text{uc}} u : I \to I' \quad \mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} p : I' \quad \vdash_{\text{uc}} v : I \otimes \mathfrak{C} \to I' \otimes \mathfrak{C}}{\mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} v^{-1} \circ p \circ u : I}\ \textsc{proc-upcast} \tag{5.5}$$

We rely on a judgement $\vdash_{\text{uc}}$ to provide our upcast procedural code; it may be read similarly to $\vdash_{\text{proc}}$ but needs no context. There is no utility in permitting PROC-UPCAST recurse directly to itself; any such derivation can be rewritten by fusing together the two upcasts into one and the two downcasts similarly.

③ It may be that no single option in the procedure table covers a call we wish to make, but that we can condition on the actual query made at runtime and dispatch to a procedure capable of handling that query. That is, if we can find a *covering* (not necessarily a *partition*) of the inst I from other procedures, then we can answer any query $\tau \in I$ (assuming that we can test $\tau \in I_i$ for each cover element I_i). It is not required that the CASE analysis pick any specific i such that $\tau \in I_i$; it may even be nondeterministic at runtime.

$$\frac{I_1 \cup \cdots \cup I_k = I \quad (\forall_i)\mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} p_i : I_i \quad (\forall_i) \vdash_{\text{uc}} u_i : I_i \otimes \mathfrak{C} \to I \otimes \mathfrak{C}}{\mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} \texttt{CASE}\ \tau\ \texttt{OF}\ \{I_i \mapsto u_i(p_i(\tau)) \mid i \in \mathbb{N}_1^k\} : I}\ \textsc{proc-case} \tag{5.6}$$

④ The operations so far have been at the level of whole queries; we are able to look for procedures accepting larger (PROC-UPCAST) or smaller (PROC-CASE) instantiation states. Another possibility is that there may exist a procedure, p, capable of handling a *subsuming* instantiation state, I', i.e., one that contains *supersets* of the sets in the intantiation state we seek to handle, I. To avail ourselves of p, we must *widen* or *embed* our queries into queries it understands and then *filter* any sets that result to remove extraneous answers.

$$\frac{\vdash_{\text{ee}} \texttt{EMBED}\ I\ I' : I \to I' \quad \vdash_{\text{uc}} u : I' \to I''}{\vdash_{\text{eu}} u \circ \texttt{EMBED}\ I\ I' : I \to I''}\ \text{EMBED-UP} \qquad \frac{\forall_{\tau \in I}\, \exists_{\tau' \in I'}\, \tau \subseteq \tau'}{\vdash_{\text{ee}} \texttt{EMBED}\ I\ I' : I \to I'}\ \text{EMBED} \tag{5.7}$$

$$\frac{\vdash_{\text{eu}} e : I \to I' \quad \mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} p : I' \quad \mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} f : \{\langle \texttt{unify}\langle \tau, \sigma\rangle, \texttt{true}\langle\rangle\rangle \mid \tau \in I' \bowtie \mathfrak{C}, \sigma \in I\}}{\mathfrak{C};\mathfrak{P} \vdash_{\text{proc}} \lambda_\tau \texttt{FILTER}\ (f \circ \langle \texttt{unify}\langle \cdot, \tau\rangle, \texttt{true}\langle\rangle\rangle)\ (p(e(\tau))) : I}\ \text{PROC-FILTER} \tag{5.8}$$

In particular, these adaptor rules let us use a procedure expecting a FREE τ' query to answer queries for any $\tau \subseteq \tau'$, so long as we can also find a procedure which lets us filter the results (elements of $I' \bowtie \mathfrak{C} = \text{FREE}\ \tau' \bowtie \mathfrak{C}$) for just their results within τ.

In the above, our auxiliary judgements ($\vdash_{\text{ee}}$, $\vdash_{\text{eu}}$, and $\vdash_{\text{u}}$) were given rules that enforced only *set-theoretic* conditions on the procedural code they considered. In an actual implementation, one must take additional constraints into consideration, including the ability to (efficiently) carry out the required operation. For example, if an implementation derives inst-specific storage layouts for its terms in memory, the operation of upcasting $\tau \in I$ to $\tau \in I'$ may be quite expensive, as it may involve unpacking and repacking the representations of τ in the two layouts. The system given above remains sound (though less capable) if these auxiliary judgements are arbitrarily otherwise constrained, so implementations are free to add their own requirements.

5.3.2.3 Soundness of Generated Adaptors

Theorem 3: *Under the assumptions* $(\vdash_{\text{uc}}\ u : I \to I') \Rightarrow \forall_{\tau \in I}\, \tau = u\tau$, $(\vdash_{\text{eu}}\ e : I \to I') \Rightarrow \forall_{\tau \in I}\, \tau \subseteq e\tau$, ***CASE*** *executes exactly one of its options in all cases, and the type* $\{\langle \texttt{unify}\langle x, x\rangle, \texttt{true}\langle\rangle\rangle \mid x \in \mathcal{H}\}$ *is both always provable at runtime and the only set in* $\mathfrak{C}$ *containing* $\texttt{unify}^{/2}$ *terms paired with* $\texttt{true}\langle\rangle$ *values, then, if all procedures in* $\mathfrak{P}$ *map an input* $\tau \in I$ *to the set of answers* $C \bowtie \{\tau\} \subseteq \mathfrak{C} \bowtie \{\tau\} \subseteq \mathfrak{C} \bowtie I$*, then so do all procedures derived by* $\vdash_{\text{proc}}$.

Proof. This proof is inductive on the last rule used in the call compatibility derivation.

PROC-CALL By supposition.

PROC-UPCAST By assumption and UC-SUBSET, $u\{\tau\}$ is just $\{\tau\}$ in a different inst. By induction, p sends this to $C \bowtie \{\tau\}$. This is $\subseteq \mathfrak{C} \bowtie I$ and will therefore be within the total portion of the downcast v^{-1}, and so $(v^{-1} \circ p \circ u)$ sends $\{\tau\}$ to $C \bowtie \{\tau\}$.

PROC-CASE Because $\otimes$ preserves subsets and $\forall_i \, I_i \subseteq I$, we know that $\forall_i \vdash_{\text{uc}} u_i : \cdots$ will be derivable (by UC-SUBSET). Exactly one p_i will be invoked for a given input τ; this procedure, by induction, will produce the set of answers $C \otimes \{\tau\}$ which will then be upcast to the right type.

PROC-FILTER Combining assumptions yields that $(p \circ e)$ sends τ to the stream $C \otimes \{e(\tau)\} \subseteq \mathfrak{C} \otimes I'$. The subsequent `FILTER` builds types in the query inst

$$\{\{\langle \texttt{unify}\langle t_1, t_2\rangle, \texttt{true}\langle\rangle\rangle \mid t_1 \in \sigma, t_2 \in \tau\} \mid \sigma \in C \otimes \{e(\tau)\}\}.$$

Unification with $\{\langle \texttt{unify}\langle x, x\rangle, \texttt{true}\langle\rangle\rangle \mid x \in \mathcal{H}\} \in \mathfrak{C}$, by executing the query, yields

$$\{\{\langle \texttt{unify}\langle t, t\rangle, \texttt{true}\langle\rangle\rangle \mid t \in \sigma \cap \tau\} \mid \sigma \in C \otimes \{e(\tau)\}\},$$

which equals (by assumption on e), $\{\{\langle \texttt{unify}\langle t, t\rangle, \texttt{true}\langle\rangle\rangle \mid t \in \sigma\} \mid \sigma \in C \otimes \{\tau\}\}$. We can see that the types which survive the filter and projection are exactly those in $C \otimes \{\tau\}$.

□

5.3.3 Relating Grounding Set Analysis and Conjunction Planning

The systems of §5.2 and this section have been presented as orthogonal developments. However, there should be *some* association between the possible answer *types*, $\mathfrak{C}$, and the possible answer *terms*, α. Similarly, we should expect a relationship between the permitted query *insts*, $\mathfrak{P}$, and the permitted query *terms*, κ.

- Because α is supposed to be a superset of all possible answers, we may wish to ensure that it is a superset of $\bigcup \mathfrak{C}$. While we have not discussed the appropriate closure of $\mathfrak{C}$, it stands to reason that whatever analysis we perform with knowledge of types should be at least as precise as the analysis we can perform without types.

- Dually, we should permit fewer queries in our more-informed analysis, and so should set $\bigcup \bigcup \mathfrak{P} \subseteq \kappa$.

- Internally to the two systems, it is the case that $\alpha \subseteq \kappa$ and $\bigcup \mathfrak{C} \subseteq \bigcup \bigcup \mathfrak{P}$.

5.3.4 Planning Forward-chaining

The bulk of the analysis discussed above is written from the perspective of backward-chaining, where a known query from K has specialized the rule answer inst before the planner gets to work. However, the situation in forward chaining is very similar. To plan the propagation of a notification arriving at the i^{th} subgoal, FREE ρ is refined at SG.i using the information from the notification (the HEAD is left otherwise unrefined). The *remaining* subgoals are planned as if backward-chaining, and then the HR projection is used to form the resulting *update* object(s).

The 2013 prototype of Dyna 2 implemented this algorithm (with a very simplistic inst system). It did not have the explicit message representations of §3.6.2.1; instead,

$$\frac{(\forall_i)\Phi \vdash_{\text{pcs}} \vdash_{\text{pcs}} e_i : \tau_i}{\Phi \vdash_{\text{pcs}} \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}\ \textsc{pcs-tup} \qquad \frac{(\exists_n) \quad \Phi \vdash_{\text{pcs}} e : \langle \tau_1, \ldots, \tau_n \rangle \quad k \le n}{\Phi \vdash_{\text{pcs}} \cdot{\downarrow_k}\ (e) : \tau}\ \textsc{pcs-proj}$$

$$\frac{(\forall_i) \vdash_{\text{pcd}} e_i \Downarrow V_i}{\vdash_{\text{pcd}} \langle e_1, \ldots, e_n \rangle \Downarrow \langle V_1, \ldots, V_n \rangle}\ \textsc{pcd-tup} \qquad \frac{}{\vdash_{\text{pcd}} \cdot{\downarrow_k}\ (v) \Downarrow V{\downarrow_k}}\ \textsc{pcd-proj}$$

$$\frac{\vdash_{\text{s}} \texttt{SPLIT}\ I\ \vec{J} : \tau}{\Phi \vdash_{\text{pcs}} \texttt{SPLIT}\ I\ \vec{J} : \tau}\ \textsc{pcs-split} \qquad \frac{\forall_i\,.\Phi \vdash_{\text{pcs}} e_i : \diamond I_i \quad \bigcup_i I_i = I}{\Phi \vdash_{\text{pcs}} \texttt{MERGE}\ \{e_1, \ldots, e_n\} : \diamond I}\ \textsc{pcs-merge}$$

$$\frac{\bigcup_i \sigma_i = \tau \quad \forall_{i,j \neq i}\, \sigma_i \cap \sigma_j = \varnothing}{\vdash_{\text{pcd}} \texttt{SPLIT}\ (\tau) \Downarrow \vec{\sigma}}\ \textsc{pcd-split} \qquad \frac{\forall_i\,. \vdash_{\text{pcd}} e_i \Downarrow V_i}{\vdash_{\text{pcd}} \texttt{MERGE}\ \{e_1, \ldots, e_n\} \Downarrow \bigcup_i V_i}\ \textsc{pcd-merge}$$

Figure 5.3: Addenda to figures 5.1 and 5.2 for the language additions for handling query split and merge. The auxiliary judgement $\vdash_{\text{s}}$ is defined in equation (5.10) (in §5.3.5).

because it was a *fully-materialized* system (i.e., the memo table never stored UNK), it ran every propagator *twice*, once with the old value and once again with the new value, giving it a kind of DRed-like feel (recall §2.6). The HR projection obtained from the first, old-value-based, pass removed aggregands from child items, while the second pass added (new) ones. The prototype was, otherwise, much as suggested in §3.2; notably, it ensured that each propagator generated finitely many updates to items.

5.3.5 Aside: Query Split and Merge

In the above proof of soundness of our adaptor logic, we had to assume a seemingly unusual requirement of the behavior of our runtime system, specifically, that *the only* representation of $\texttt{true}\langle\rangle$-valued $\texttt{unify}^{/2}$ items in the system is $\{\langle \texttt{unify}\langle x, x\rangle, \texttt{true}\langle\rangle\rangle \mid x \in \mathcal{H}\}$. (We did not, however, have to assume that every possible subset of $\wp(\langle \texttt{unify}\langle \mathcal{H}, \mathcal{H}\rangle, \mathcal{H}\rangle)$, or even $\wp(\langle \texttt{unify}\langle \mathcal{H}, \mathcal{H}\rangle, \texttt{true}\langle\rangle\rangle)$, had a usable procedure in $\mathfrak{P}$; the use of PROC-FILTER is *conditional* on there being such a procedure.) Had we assumed that there could be several different answer types covering unification (so that, instead, merely, $\langle \texttt{unify}\langle \mathcal{H}, \mathcal{H}\rangle, \texttt{true}\langle\rangle\rangle \cap (\bigcup \mathfrak{C}) = \{\langle \texttt{unify}\langle x, x\rangle, \texttt{true}\langle\rangle\rangle \mid x \in \mathcal{H}\}$), we would have found that PROC-FILTER could have split our responses from the larger procedure. That is, while it may have been that $\{\alpha\} = \{\tau\} \otimes \mathfrak{C}$, and so our notion of a procedure would require that any procedure handling τ would have to have an output inst containing α, our use of PROC-FILTER may have instead returned subsets of α corresponding to the fragments found within the different $\texttt{unify}^{/2}$-keyed kv-pair types of $\mathfrak{C}$. Importantly, this would not have been incorrect, in the sense of violating the semantics of the language. It would merely frustrate analysis: we could no longer assume that the results from a procedure with input inst I would be given by $I \otimes \mathfrak{C}$, but rather, we would have to assume the subset-closure of that result, i.e., $\text{sc}(I \otimes \mathfrak{C})$ where $\text{sc}(T) \overset{\text{def}}{=} \{\alpha \subseteq \tau \mid \tau \in T\}$. This, in turn, would mean that we would have to redefine $\text{refrule}_{\text{inst}}$ as

$$\text{refrule}_{\text{inst}}(r, K, \vec{v}) \overset{\text{def}}{=} \{\rho_r\}[K{\downarrow_1} /\!\!/ \textsc{head}][\text{sc}(\mathfrak{C}) /\!\!/ \textsc{sg}.v_1] \cdots [\text{sc}(\mathfrak{C}) /\!\!/ \textsc{sg}.v_{\text{tlen}(\vec{v})}]. \tag{5.9}$$

This is readily seen to be equivalent to asserting that $\mathfrak{C}$ is itself subset-closed and so represents a loss of precision within our static analysis. Given the options, we consider imposing the constraint on $\texttt{unify}^{/2}$-keyed kv-pair sets in $\mathfrak{C}$ to be the less drastic option.

Nevertheless, the requirement on procedures (that they send τ to $\{\tau\} \otimes \mathfrak{C}$) might only be required of procedures *in the set* $\mathfrak{P}$. That is, so long as we can guarantee by the end of aggregation (see §5.4) that the outcome of a plan for the query inst I has generated answer *types* in $I \otimes \mathfrak{C}$, we may find the *terms* of those types by hook or crook. We can consider splitting queries into a finite collection of sub-queries, each of which can then be discharged by an existing procedure, and the results can be merged. This requires a relatively simple modification to the analysis above.

We augment our call adaptor language with new powers. We enrich it with tuples, adding, as expressions ($\mathcal{E}$), both a constructor form, $\mathcal{E}^*$, and eliminator forms $\cdot\downharpoonleft_n$ (with $n \in \mathbb{N}$); as might be imagined, only $\mathcal{V}^*$ and $\cdot\downharpoonleft_n$ are values, and we may write $e\downharpoonleft_n$ as sugar for $(\cdot\downharpoonleft_n)(e)$. Further, we add procedural forms to carry out the splitting of queries, $\texttt{SPLIT}\ I\ I^*$, and merging of results, $\texttt{MERGE}\ \wp_{\text{fin}}\mathcal{E}$. Of these, only the former constitutes a value form ($\mathcal{V}$). The static and dynamic semantics are straightforward (for tuples) and as might be imagined from the names given (for the other forms) and are shown in figure 5.3. So equipped, we duplicate the proof system of §5.3.2.2 and add two more inference rules:

$$\frac{\forall_{\tau \in I}\ \exists_{\sigma_1 \in J_1, \ldots, \sigma_{\text{tlen}(\vec{J})} \in J_{\text{tlen}(\vec{J})}} (\tau = \bigcup_i \sigma_i \land \forall_{i,j \neq i}\, \sigma_i \cap \sigma_j = \varnothing)}{\vdash_{\text{s}} \texttt{SPLIT}\ I\ \vec{J} : I \to \langle J_1, \ldots, J_{\text{tlen}(\vec{J})} \rangle}\ \textsc{split} \tag{5.10}$$

$$\frac{\vdash_{\text{s}} \texttt{SPLIT}\ I\ \vec{J} : I \to \langle J_1, \ldots, J_{\text{tlen}(\vec{J})} \rangle \quad (\forall_i)\mathfrak{C}; \mathfrak{P} \vdash_{\text{proc}} p_i : J_i}{\mathfrak{C}; \mathfrak{P} \vdash_{\text{proc}} (\lambda_{\vec{\tau}} \texttt{MERGE}\{p_1(\tau_1), \cdots, p_{\text{tlen}(\vec{J})}(\tau_{\text{tlen}(\vec{J})})\})\ (\texttt{SPLIT}\ I\ \vec{J}) : I}\ \textsc{proc-split} \tag{5.11}$$

The result of §5.3.2.3 *does not hold* for this augmented system. (As an aside, we note that the disjointness condition on SPLIT, in equation (5.10), ensures that the union within PCD-MERGE, in figure 5.3, is guaranteed to be a *disjoint* union. This is not, strictly, required, as union would collapse duplicated rule answers, but it is likely convenient in implementation to avoid the comparisons that would be required to implement a non-disjoint union.)

$\text{refrule}_{\text{inst}}$ must now be modified to capture *which* proof system was used for a given subgoal; rather than pass it a tuple of paths, we will pass it a tuple of pairs each containing a path and either the function id (i.e., $\text{id}(x) = x$) or sc from above, for application to $\mathfrak{C}$. Thus $\text{refrule}_{\text{inst}}(r, K, \vec{v})$ is redefined to be

$$\{\rho_r\}[K\downharpoonleft_1 /\!\!/ \text{HEAD}][(\vec{v}\downharpoonleft_{1.2})(\mathfrak{C}) /\!\!/ \text{SG}.(\vec{v}\downharpoonleft_{1.1})] \cdots [(\vec{v}\downharpoonleft_{\text{tlen}(\vec{v}).2})(\mathfrak{C}) /\!\!/ \text{SG}.(\vec{v}\downharpoonleft_{\text{tlen}(\vec{v}).1})].$$

The planner must pass in sc for subgoals which it has planned using this alternate proof system and should pass id when it uses the original system (though it remains *not incorrect* to pass sc, but unnecessary adaptor glue code would be necessary in later subgoals). Thus, we suggest, operationally, that the planner either ① *try* using the first system and, only if that fails, re-try with the second, or ② search using the combined system, but favor derivations that do not use the added power of the second.

5.3.6 Aside: Assurance of Answers via Determinism Analysis

Prolog static analysis also concerns itself with *how many* answers come back from each query. The most useful properties to know are whether ① it is certainly, certainly not, or possibly the case that *no* answers are forthcoming; and ② if an answer is returned, that it will be the only one. This gives rise to a six-way taxonomy of the length of the answer stream, which proves useful in practice while implementing REFINERULESUFFIX. For example, knowing that there is no more than one answer to a query (as might arise, for example, when performing arithmetic) means that there will be no need to refer to the current partially-refined rule answer set repeatedly, and so it need not be copied or trailed (recall the discussion in §4.1). This kind of **determinism** analysis is discussed in Overton [137, §2.5.4] and seems to have originated in Henderson, Somogyi, and Conway [91] and Nethercote [134, §6]. Such a system seems directly applicable to the weighted case of μDyna programs.

Determinism is not quite orthogonal to the cardinality estimation analyses in §5.2.4.1. Clearly, if the estimated cardinality promises at least one answer *term*, there must be at least one answer *type* to carry that answer. If the lower bound is instead zero answer terms, then there may be no answer types forthcoming. Similarly, if estimated cardinality promises at most one answer *term*, there can be only one (non-empty) answer *type*. However, outside of the special cases of 0 and 1 answer *terms*, the two systems diverge: arbitrarily many (even infinitely many) answer terms fit in one answer type, and the system is generally free to re-arrange answer types as it sees fit, so long as their union contains the same answer terms.

A separate, determinism-like analysis that might be worth investigation would be to not only constrain the length of the answer stream as a whole, but to consider the number of unique values at various projections within. For example, it may be worth distinguishing the answer streams $\{\langle\!\langle \{\mathtt{f}\langle x,y\rangle\}, \mathtt{true}\langle\!\langle\rangle\!\rangle \rangle\!\rangle \mid x,y \in \mathbb{N}_1^{10}\}$ and $\{\langle\!\langle \{\mathtt{f}\langle 1,y\rangle\}, \mathtt{true}\langle\!\langle\rangle\!\rangle \rangle\!\rangle \mid y \in \mathbb{N}_1^{100}\}$, even though they are both possible given an answer inst of GROUND $\langle\!\langle \mathtt{f}\langle\!\langle \mathbb{N},\mathbb{N}\rangle\!\rangle, \mathtt{true}\langle\!\langle\rangle\!\rangle \rangle\!\rangle$. While both contain the same number of answers (i.e., 100), the latter has *only one* value for its $\cdot\!\downharpoonleft_1$ projection, while the former has several (in fact, 10). The uniqueness of value of a projection may be useful knowledge when analysing later subgoals. Similarly, it may be worthwhile to track any *functional dependence* among projections (including, but more generally than, the implicit value-on-key dependence). We leave such considerations to future work.

5.4 Planning Disjunctions

Having planned the conjunction of subgoals, we are left with an instantiation state describing the possible shape of rule answers, as computed by $\mathrm{refrule}_{\mathrm{inst}}$. The next step is to extract, from *all rules'* possible shapes of rule answers, the possible shapes of *aggregated* answers. That is, just as we ensured closure of $\mathfrak{O}$ under an operation that approximated the answer *terms* of the program, we now consider closure of $\mathfrak{C}$ under our approximation of the answer *types*.

The rule answer inst (i.e., set of possible rule answer sets) for a query κ bounds the inputs to CONTRIBRULEANSWER (first extracted with $\mathrm{selt}(\cdot)$ in §3.2, or as-is in §3.3.3) or to ANSWERFOR as-is (in §3.4.3.6). §3.2 had to assume sufficient range restriction in order to promise that REFINERULESUFFIX could extract the single rule answer from each answer set; we are now in a position to *verify* this range restriction by ensuring that the possible rule answer inst is (equivalent to) some GROUND τ.[144] Being able to make an analogous assertion

[144] For the particular query κ, anyway. It is, unsurprisingly, easier to check K-sufficient range restriction

of the system of §3.3 would be quite challenging, as we would need to be able to model the behavior of DISJOIN and would, thereby, seemingly require detailed understanding of the exact computational limits of the underlying automaton family's ability to reason about set subtraction. The system of §3.4 is, on the other hand, designed to work with *whatever heads the rules produce*, relying on the machinery of ∩-closure and masking to achieve its results. (We are also in a position to verify the *non-ground range restriction* of programs operating within this system, by verifying that the RES *projection* of the possible answer inst is ground, even if the RES is not.) While determining *whether* AnswerFor will be able to consume a given answer type α from the answer inst is likely beyond our capability, we can bound its behavior: if it returns, it must return a bag containing *one of* the values from $\alpha\downharpoonleft_{\text{RES}}$ at a multiplicity bounded by the analysis of §5.2. From these three pieces ($\alpha\downharpoonleft_{\text{HEAD}}$, $\alpha\downharpoonleft_{\text{RES}}$, and the bounds on the cardinality of the answer), we can characterize the set of aggregands up to the point of ∩-closure. Seeing detailed analysis through ∩-closure again seems challenging, as we would need to somehow appropriately summarize the behavior of potentially infinitely many possible defaults encoded by the rule answer inst.

For the case of default reasoning, where *heads* are readily available, it may be tempting to try to use the answer term *set*, $\mathfrak{D}$, from §5.2 to reconstruct the answer inst from the heads obtained by conjunctive inst reasoning as in §5.3; that is, given a rule answer inst I, one would pair each $\eta \in I\downharpoonleft_{\text{HEAD}}$ with the union of all possible values for each $h \in \eta$, i.e., $\bigcup_{h\in\eta}(\mathfrak{D} \cap \llangle\{h\}, \mathcal{H}\rrangle\downharpoonleft_2)$. However, the result is only correct *modulo occlusions*: it no longer accurately describes the contents of the answer stream, but only some theoretical "equivalent" answer stream wherein overrides have already been taken into account.

Example 52: To be concrete for the above concern, consider a default that is *completely* overridden. Suppose that $\mathfrak{D}$ is such that it is certain that both $\texttt{f}\langle\texttt{true}\langle\rangle\rangle \mapsto 1$ and $\texttt{f}\langle\texttt{false}\langle\rangle\rangle \mapsto 2$ (and that these come from ground rules). Further, suppose that the μDyna program specifies that $\texttt{f}^{/1}$ items aggregate by minimization. If the program additionally contains the rule $\{(\texttt{f}\langle b\rangle \hookleftarrow 3) \Leftarrow \langle\rangle \mid b \in \{\texttt{true}\langle\rangle, \texttt{false}\langle\rangle\}\}$ a default-reasoning-based solver may return an answer (to be overridden) $\llangle\texttt{f}\llangle\{\texttt{true}\llangle\rrangle, \texttt{false}\llangle\rrangle\}\rrangle, \{3\}\rrangle$, but such would be missed by the above "recovery," which would construct only $\llangle\texttt{f}\llangle\{\texttt{true}\llangle\rrangle, \texttt{false}\llangle\rrangle\}\rrangle, \{1, 2\}\rrangle$. ◊

That said, one particularly fortuitous possibility is that the result of conjunction planning shows that all possible rule answers are *ground*, or ground modulo some orthogonal freedom in the head (i.e., the SG and RES projections are singletons but the HEAD is not). More generally, if it can be determined that all possible rule answers have disjoint (or equal) heads (recall "μDyna with Disjoint Sets," in §3.3.5), and any non-ground aspects of the heads are well-behaved sets (for the underlying family of automata used at runtime), the program may be executable given only the ability to represent finite sets and co-finite subsets of these well-behaved sets. The 2013 prototype of Dyna 2 restricted μDyna programs to those whose rule answer sets were entirely ground, but the generalization to disjoint heads should be straightforward.

when given a finite set of possible queries K than it is to generate the set of K for which a given program is K-sufficiently range restricted.

5.5 Inexact Values as Keys

In the circuit encoding of §2, we took the parent/child relationship between items as part of the input specification of a circuit. This allowed our solver to obtain the parents and children of any item at any point in the solver's execution without reflecting on the values of items within the circuit.[145] Now that our circuit is defined *at the same time* as the values, using the machinery of rules, there is no longer a separation of concerns between the *structure* of the circuit and the *values* within the circuit. In fact, this is *a useful feature* of the language and of the design, however, it poses problems for the solver.

In addition to *overt* use of equality tests within a user's program as an explicit subgoal within a rule, our solver relies upon equality within its unification and refinement operations, within its *search* through the memo table, as part of cycle detection during backward chaining, and while probing the agenda for messages to merge when forward-chaining.

While more refined restrictions may exist, we take a heavy-handed approach. We prohibit cyclic definition of such items, which generally will mean that the ancestors of any $\mathtt{f}^{/n}$ with inexact keys may not include other $\mathtt{f}^{/n}$ items. (Of course, if there are other, *exact* keys that partition the sets of items, that can permit such "functor-level" apparent cyclicity.) Moreover, we require that forward-chaining of items with inexact keys manifests as a *non-ground* invalidation notification ($\tau : \leftarrow \textsc{unk}$), where ① any path π such that π is a path of all trees in τ (i.e., all $\{t{\downarrow_\pi} \mid t \in \tau\}$ are defined), ② there exists an extension of π which is not a path in all trees in τ, and, ③ if $\tau{\downarrow_\pi}$ contains an inexact value or a tree therein has an inexact value at its leaf, then $\tau{\downarrow_\pi} = \mathcal{H}$. This somewhat complex set of constraints means that we can pass *any exact-valued information* we have about the key forward, but all inexact information in the key is completely erased.

[145] Obligation, in §2.2.4.3, introduced a values-dependent subset of an item's children, but so long as the solver always *over-estimated* this set, possibly by simply returning the set of *all* children, the solver remained correct. That is, *correctness* of update propagation was not dependent on the availability or freshness of values of the circuit being solved, but the *effiency* of the solver is improved by having tighter (but always over-) approximations of the obligation relationship.

Chapter 6

Miscellaneous Thoughts On Dyna

> This was our contribution to the proof of the Law of Software Envelopment: "Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can."
>
> Jamie Zawinski, on Netscape Mail and News; https://www.jwz.org/hacks/

6.1 Error Values

> "Доверяй, но проверяй" ("Trust, but verify")
>
> Russian proverb

As mentioned in §5.3, static analysis is key to generating efficient, reusable code as it enables us to perform, once, the expensive search for a call-compatibility-respecting plan for some procedure, so long as the inputs ascribe to the stated inst. However, such analysis is necessarily incomplete: we will reject some programs that are correct, but whose proof of correctness exceeds the capability of our analysis theory and/or its implementation. In the interest of flexibility, we can add a *procedural* mechanism of **assertions** to Dyna's still-mostly-declarative core, so that programmers can attest that, by a given point within a rule body, a given term will be an element of a given type. This gives (potentially) new information to the static analyser, and allows it to proceed with increased precision. However, these assertions may linger on to be executed at runtime, and must do *something* when violated. The resulting system permits *declarative handling* of such errors and integrates well with the default reasoning algorithm of §3.4.

To begin, we assume a non-empty set of **error values**, $\mathfrak{E} \subseteq \mathcal{H}$. Let us, for simplicity, assume that all error values have some unique functor, e.g., any $\texttt{\$error}^{/k}$, at their root, and that any such term is considered an error value. These error values are values like any other within the system, and may be used as or within values of a rule, used as or within values of subgoals, and used within (but *not as*) keys in the head or a subgoal. (That is, unlike NULL. We, somewhat artificially, prohibit the use of an error value as an item; attaching a weight to an error value does not seem sensible. As such, we exclude error values from being the entirety of a head or subgoal key.) These values (or, potentially, particular subsets) are used to indicate assertion failures. Let us proceed by example first.

Example 53 (*Catching Division By Zero*): A family of items implementing, say, rational division has no appropriate result for queries of the form $\langle \texttt{rdiv}\langle x, 0\rangle, v\rangle$ with x and v drawn from any nonempty subset of rationals. In a sense, the query $\langle \texttt{rdiv}\langle \{0\}, \{0\}\rangle, \mathbb{Q}\rangle$ should return *as is*—every rational, when multiplied by zero, produces zero—but we require a functional dependence, namely that a ground key be associated with *at most one* value. On the other hand, any query $\langle \texttt{rdiv}\langle x, \{0\}\rangle, \mathbb{Q}\rangle$, for $x \neq 0$, should fail: there truly is no such rational. Given the *totality* of division on all inputs when the divisor isn't 0, we may wish to statically prohibit even asking the question. Thus, we may wish to permit $\texttt{rdiv}^{/2}$ queries only of the form $\tau = \langle \texttt{rdiv}\langle \mathbb{Q}, \mathbb{Q} \setminus \{0\}\rangle, \mathcal{H}\rangle$. That is, we set our permitted query set κ such that $\kappa \cap \langle \texttt{rdiv}\langle \mathcal{H}, \mathcal{H}\rangle, \mathcal{H}\rangle = \tau$.

While many subgoals may be obviously correct (e.g., division by a non-zero constant), it may be that analysis is unable to conclude that 0 is outside the range of possibilities for a non-constant divisor. (Say, because analysis has had to revert to tracking a safe superset and/or because its understanding of mathematics is insufficiently precise.) In such a case, it may be that the programmer is willing to explicitly *assert* that the denominator is nonzero, permitting the query to go through after a *runtime* test that the denominator's value is, indeed, nonzero. Should that test fail, the refinement of the rule (suffix) will abort before attempting to invoke a non-permitted query and will yield an error value. $\Diamond$

An assertion is a special form of subgoal, with explicit, *procedural* handling within REFINERULESUFFIX. Given a term $\text{Ᵽ} \in \mathcal{H}$ that describes a set $\tau \subseteq \mathcal{H}$,[146] the item $\texttt{\$assert}\langle s, \text{Ᵽ}\rangle$ has value $\texttt{true}\langle\rangle$ if $s \in \tau$. The value for when $s \notin \tau$ is not defined; attempting to ask the question results in REFINERULESUFFIX *aborting* rather than continuing to the next subgoal. Such an abort contributes *an error value* to *the current heads* of the rule. In order to ensure that the notion of "current heads" is sensible, subgoal reordering is prohibited across assertions that cannot be statically discharged. An assertion can be statically discharged if the type system of §5.2 can demonstrate that it always holds; in such a case, the subgoal may be removed entirely, as the runtime test is certainly unnecessary. If an assertion cannot be statically discharged, the type system is free to proceed to additional subgoals under the assumption that it holds, but must add (assertion) error values to the set of aggregands that may be directed to items in the current head type. Thus, while the μDyna rule $\{(\texttt{f}\langle h\rangle \hookleftarrow v) \Leftarrow \langle \texttt{g}\langle s\rangle \mapsto v, h = 2 \mapsto \texttt{true}\langle\rangle\rangle \mid h \in \eta, s, v\}$ contributes the value of each $\texttt{g}\langle s\rangle$ to $\texttt{f}\langle 2\rangle$ (assuming that $\texttt{=}^{/2}$ implements unification), the rule $\{(\texttt{f}\langle h\rangle \hookleftarrow v) \Leftarrow \langle \texttt{g}\langle s\rangle \mapsto v, \texttt{\$assert}\langle s, \text{Ᵽ}\rangle \mapsto \texttt{true}\langle\rangle, h = 2 \mapsto \texttt{true}\langle\rangle\rangle \mid h \in \eta, s, v\}$ will contribute an error to *all* $\texttt{f}\langle \eta\rangle$ if there exists an s such that $\texttt{g}\langle s\rangle$ has non-NULL value and $s \notin \tau$, even though the subgoals *could* be reordered so that the sole recipient of aggregands was

[146] We gloss over precisely *how* this description takes place. One possibility is to define a term encoding much along the lines of our automata of §4.2. Another is to explicitly add, e.g., regular subsets of $\mathcal{H}$ to $\mathcal{H}$ itself and enrich Dyna with operators for manipulating these sets directly. Such additions would allow Dyna itself to describe automata à la §4.3 and would represent a kind of "Higher-Order Set Syntax," in which the Dyna language would inherit the automata capabilities of its runtime library, much as "Higher-Order Abstract Syntax" (HOAS) [143] allows languages to inherit functional abstraction from the languages of their implementation.

known to be $\mathtt{f}\langle 2\rangle$.

An interesting consequence of our design is that one cannot both throw and catch an assertion failure within the same rule. Procedurally, *rules* form the basic blocks enclosed within exception handlers.

6.1.1 Error Latch-up

Presumably, some aggregators will treat errors as absorbing elements: regardless of other aggregands, an error input means an error output. In the presence of cyclic dependencies, this gives rise to what we call **error latch-up**, whereby error values may never clear, even after the root cause has resolved itself. Suppose that we have the pair of rules $\{(\mathtt{a}\langle\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{a}\langle\rangle \mapsto a, \mathtt{\$assert}\langle a, Ᵽ\rangle \mapsto \mathtt{true}\langle\rangle, 2/a \mapsto v\rangle \mid a, v\}$ and $\{(\mathtt{a}\langle\rangle \hookleftarrow v) \Leftarrow \langle \mathtt{b}\langle\rangle \mapsto v\rangle \mid v\}$ with Ᵽ encoding $\mathbb{R} \setminus \{0\}$. (Roughly, the ill-founded definition $a = b + 2/a$.) Such a cycle has error-free solutions in which $a = \frac{1}{2}(b \pm \sqrt{b^2 + 8})$. Suppose, initially, that the driver assigns the value 1 to $\mathtt{b}\langle\rangle$, so that the cycle stabilizes with $\mathtt{a}\langle\rangle$ having value 2. ($\mathtt{a}\langle\rangle$'s value is likely to evolve through time from an initial NULL through 1, being the result of $\sum\{\text{NULL}, 1\}$, to 3, the result of $\sum\{1, 2\}$, and so on.) If an update now takes $\mathtt{b}\langle\rangle$ to -1, $\mathtt{a}\langle\rangle$ is likely to step, by forward-chaining, to being 0, at which point the assertion trips and makes $\mathtt{a}\langle\rangle$'s value an error (i.e., an element of $\mathfrak{E}$). The assertion *trips again*, because $\mathfrak{E} \cap \mathbb{R} = \varnothing$, and $\mathtt{a}\langle\rangle$ has converged at an error value (rather than at -2 or 1). Subsequent updates to the value of $\mathtt{b}\langle\rangle$ will not alter the value of $\mathtt{a}\langle\rangle$.

We imagine that such latch-up will be addressed within a Dyna solver by endowing error values $\mathfrak{E}$ with causal provenance data, even if these annotations are not visible to the Dyna program itself. These root cause data can be processed by a reuse of the machinery the solver uses to address similar latch-up in *obligation* on cyclic circuits (recall §2.5.4). In particular, we expect the two errors reported above—the one arising from $0 \notin \mathbb{R} \setminus \{0\}$ and the other from $\mathfrak{E} \cap \mathbb{R} = \varnothing$—are observably different to the solver, with the latter seen to be *caused by* the first. The cause of the second error is, like that of the first, located at the $\mathtt{\$assert}\langle\rangle$ subgoal in the rule for $\mathtt{a}\langle\rangle$, but the second error's cause is *the first error*; thus, the solver can detect the (possible) circularity of justification for the error. In light of such circular justifications, the solver is free to *experimentally guess* a new value for $\mathtt{a}\langle\rangle$ (recall §2.5.2). If the solver guesses NULL and continues, the value of $\mathtt{a}\langle\rangle$ will subsequently evolve as -1, -2, $-3/2$, $-5/3$, $-8/5$, etc., converging *numerically* on $-\frac{1}{2}(1 + \sqrt{5})$.

If the runtime overhead of dynamic detection of self-supporting error values is judged to be too high, a potentially less expensive option is to allow *the driver* to cause the solver to act as though it had guessed new values. In such a case, the interface should allow clearing multiple (or possibly even *all*) errors at once, lest there be *mutually supporting* error values.

6.2 Modularity

We proposed, in Eisner and Filardo [50], a novel, declarative, prototype-based object system for use *within* the Dyna language itself. Our design is heavily inspired by existing prototype-based object systems, including Self [176] and JavaScript [47], but is adapted for the declarative nature of Dyna. Emphasizing the dynamic-database view of Dyna programs, we call our objects **dynabase**s. The use of multiple dynabases within a Dyna program provides for inheritance-based code reuse,[147] and also facilitates a kind of "what-if" reasoning whereby entire programs may be copied and subject to additional input. While the semantics of the system have not meaningfully changed since its introduction, we give a different presentation here, using the μDyna formalism, and give examples of the system's behavior from a language-author's perspective, rather than the user-oriented examples of the original text.

A dynabase is, at its heart, a collection of (extended) Dyna rules; every program that could be given in the language as defined so far would form a dynabase, albeit one not taking advantage of the object-oriented programming (OOP) features. To the μDyna system defined so far, we add several new features. ① Given a **handle** (reference) to a dynabase, one may make queries of its program, just as ordinary subgoals query the current dynabase's program. ② A dynabase may be **clone**d from a **progenitor** dynabase.[148] ③ The dynabase in which a clone is made becomes the **owner** of the clone and may provide additional aggregands to items therein. Each of these features necessitates changes to the μDyna formalism, which we detail momentarily.

First, however, we should point out a key distinction between dynabases and traditional OOP within more procedural languages. Specifically, dynabases are *not stateful*: they, like the broader declarative program in which they exist, do not evolve in time. Of course, the *input* to the program may vary with time, and so the values associated with items of a given dynabase may covary as its definition dictates, but the definition itself remains static. This point may seem obvious, but it drives a central component of the design: while a traditional object o may have its state updated by any other object that holds a reference to o (and has the ability to call a mutating method or write to a field), this is not tenable for Dyna, as the solver must be able to backward-chain items within a dynabase and so must be able to find all the places whence aggregands could have come. Thus, we restrict specification of a dynabase to its (singular) progenitor, its own rules, and its (singular) owner.

In a typical OOP system, the actual underlying identifiers (e.g., memory addresses) of objects are not *directly* available in the source language and so an object can only be referenced at places in the program where one has been given a name or handle—an *indirect* reference—to that object; our dynabase system has the same property. At the beginning of execution, at least two dynabases exit by construction: these include the primordial dynabase, denoted $\gimel$, from which all others are descended, and the root dynabase of the program, denoted $\daleth$.[149] We will almost always elide $\gimel$ in depictions of dynabase arrangements; one should assume that dynabases without rendered progenitors have $\gimel$ in this role, unless otherwise stated. While we presume the existence of a set $\mathcal{D} \subsetneq \mathcal{H}$ of dynabase handles

[147] While we will not discuss the details here, there is a proposed mechanism for *encapsulation* as well, wherein the programmer can specify which items are available for external reads and/or writes as well as visibility to clones. Thankfully, it is largely orthogonal to the core presentation of this section.

[148] The original work used the term "parent" here, but we now consider it too confusing, given the recent emphasis on parentage of items within a computational circuit.

[149] Other dynabases that may exist by construction include links to foreign data sources, encapsulated within the solver's dynabase API.

(with $\{\gimel, \daleth\} \subseteq \mathcal{D}$), we require that extended μDyna rules (introduced shortly), as generated from a source Dyna program, be the image of a "purely syntactic" function from dynabase handles to rule structure. That is, the set ρ may not refer to specific entries of $\mathcal{D}$, but rather quantify over the entirety of $\mathcal{D}$ (and may use the quantified variable non-linearly). (Thus, for example, $\forall_{d \in \mathcal{D}, r, \pi}(d \in \rho_r \downharpoonleft_\pi) \Rightarrow (\mathcal{D} \subseteq \rho_r \downharpoonleft_\pi)$ is a necessary but not sufficient property of our extended μDyna rules.)

Extending μDyna rules to support dynabases makes three straightforward changes, each of which roughly corresponds to the features given above.

① Much as methods in a traditional OOP system have access to a *dynamic* notion of "self" (typically pronounced "`this`"), an extended μDyna rule will need access to the dynabase against which its subgoals are to be resolved by default. The *pair* of HR and SG is now extended to a *triple*; HR and SG are unaltered and DI $\overset{\text{def}}{=}$ 3. As mentioned above, we require that $\rho\downharpoonleft_{\text{DI}} = \mathcal{D}$. The outermost pairing operator of a μDyna rule, which we used to represent with $\Leftarrow$, will now be depicted as $\overset{d}{\Leftarrow}$, where d denotes the default dynabase for evaluation.

② Each subgoal *explicitly* names a dynabase as the environment whither we shall direct a subgoal query, and whence answers will be returned. The rules we have considered until now can be thought of as simply using the "`this`" dynabase for each. The kv-pairs of at each $\{\text{SG}.i \mid i\}$ are now also made into triples, $\langle \text{key}, \text{value}, \text{dynabase} \rangle$. Mnemonically, we will render $\langle k, v, d \rangle$ as $k \overset{d}{\mapsto} v$.

③ Because aggregands now route to heads *on a dynabase*, we must specify which. The HEAD kv-pair is extended to be a triple, just as with the subgoals. Our mnemonic for this will be $h \underset{d}{\hookleftarrow} r$ for $\langle h, r, d \rangle$, i.e., the contribution of r to h on dynabase d. (In rough keeping with our notation of §2, dynabases above the line source information, and the one below receives it.) We require that $\rho\downharpoonleft_{\text{HEAD}} \cap \mathcal{D} = \varnothing$ for all rules ρ: that is, dynabase handles are terms, but one may not attempt to use a dynabase handle as the name of an item. The names of items will become important components of dynabase handles, below.

All told, the μDyna rule for matrix-vector products from §3.1 might now appear as

$$\{(\mathtt{rs}\langle x \rangle \underset{d}{\hookleftarrow} v) \overset{d}{\Leftarrow} \langle \mathtt{r}\langle x, y \rangle \overset{d}{\mapsto} r, \mathtt{s}\langle y \rangle \overset{d}{\mapsto} s, \otimes\langle r, s \rangle \overset{d}{\mapsto} v \rangle \mid r, s, v, x, y \in \mathcal{H}, d \in \mathcal{D}\},$$

if all items are on the same dynabase. That is, all subgoals read from d, which will be specified when the rule is to be interpreted, and all aggregands route to heads on d as well. If, on the other hand, the $\mathtt{r}^{/2}$ and $\mathtt{s}^{/1}$ items were to be drawn from a dynabase to which a handle is stored at $\mathtt{src}\langle\rangle$, then the rule would then read

$$\{(\mathtt{rs}\langle x \rangle \underset{d}{\hookleftarrow} v) \overset{d}{\Leftarrow} \langle \mathtt{src}\langle\rangle \overset{d}{\mapsto} e, \mathtt{r}\langle x, y \rangle \overset{e}{\mapsto} r, \mathtt{s}\langle y \rangle \overset{e}{\mapsto} s, \otimes\langle r, s \rangle \overset{d}{\mapsto} v \rangle \mid r, s, v, x, y \in \mathcal{H}, d, e \in \mathcal{D}\}.$$

Here we see that d is consulted for $\mathtt{src}\langle\rangle$ and *its value* is used as the dynabase context for the next two subgoals ($\mathtt{r}^{/2}$ and $\mathtt{s}^{/1}$). Similarly, we could route the $\mathtt{rs}^{/1}$ items onto the dynabase named by $\texttt{targ}\langle\rangle$ (assuming that the current dynabase were to be the owner thereof), writing

$$\{(\texttt{rs}\langle x\rangle \underset{c}{\hookleftarrow} v) \overset{d}{\Leftarrow} \langle \texttt{targ}\langle\rangle \overset{d}{\mapsto} c, \texttt{r}\langle x,y\rangle \overset{d}{\mapsto} r, \texttt{s}\langle y\rangle \overset{d}{\mapsto} s, \otimes\langle r,s\rangle \overset{d}{\mapsto} v\rangle \mid r,s,v,x,y \in \mathcal{H}, d,c \in \mathcal{D}\}.$$

The two behaviors can, of course, be mixed, and, in fact, the rule

$$\{(\texttt{rs}\langle x\rangle \underset{c}{\hookleftarrow} v) \overset{d}{\Leftarrow} \langle \texttt{st}\langle\rangle \overset{d}{\mapsto} c, \texttt{r}\langle x,y\rangle \overset{c}{\mapsto} r, \texttt{s}\langle y\rangle \overset{c}{\mapsto} s, \otimes\langle r,s\rangle \overset{c}{\mapsto} v\rangle \mid r,s,v,x,y \in \mathcal{H}, d,c \in \mathcal{D}\},$$

in which the head and all subgoals refer to items in the dynabase referred to (by the containing dynabase) as $\texttt{st}\langle\rangle$ would yield the same results (on that dynabase) as if that dynabase contained the original matrix-vector product rule.

As seen, dynabase handles may themselves be values of items. This suggests that they may be aggregands, too. Indeed, they may, but, for present purposes, only two *operators* (playing the role of aggregators) consume dynabase handle aggregands. Both are defined only on singleton bags.

① The simplest is selt$(\mathfrak{U}\cdot)$, which takes a singleton bag containing a dynabase handle and returns this handle. This aggregator lets us give new names to existing handles, if it is used as the aggregator for the head item of the rule $\{(\texttt{b}\langle\rangle \underset{d}{\hookleftarrow} v) \overset{d}{\Leftarrow} \langle \texttt{a}\langle\rangle \overset{d}{\mapsto} v\rangle \mid d \in \mathcal{D}, v \in \mathcal{H}\}$, which so provides $\texttt{b}\langle\rangle$ as an alias for $\texttt{a}\langle\rangle$ (indeed, it does so for any value that $\texttt{a}\langle\rangle$ takes on, not just dynabase handles). This aggregator also enables the programmer to explicitly name the current dynabase, using a rule such as $\{(\texttt{\$self}\langle\rangle \underset{d}{\hookleftarrow} v) \overset{d}{\Leftarrow} \langle\rangle \mid v \in \mathcal{D}\}$.[150]

② The other operator available to us is new and is the *side-effecting* clone operator. Given a singleton bag containing a pair of a dynabase handle and a set of rules, e.g., $\Lbag\langle p,R\rangle\Rbag$, it constructs *a new dynabase* whose progenitor is p and whose rules are R.[151] The result is that each head "aggregated" with clone comes to name a unique clone of the indicated progenitor; the dynabase containing *the rule* becomes this clone's owner.[152]

[150] In the surface language of Dyna, one may occasionally want to refer to the current dynabase explicitly (though heads and subgoals without dynabase qualification will default to referring to the current dynabase, much as OOP languages typically obviate the need to qualify references to an object's own methods and fields with **this**). To facilitate such explicit mentions, we can assume that the primordial dynabase, $\gimel$, contains this rule defining $\texttt{\$self}\langle\rangle$, or that the identifier $\texttt{\$self}\langle\rangle$ is otherwise equated to the DI component of the μDyna rule, once all parsing and syntactic transformations are completed. (The leading \$ is used to indicate the "special" nature of the item.) In the core μDyna calculus, there is nothing special about the DI position, in that it can covary with other positions like any other, and so this rule remains something of a curio.

[151] This differs from the presentation in Eisner and Filardo [50], which lacked an overt mechanism for handling literals of rules and used a *new kind of subgoal*, pronounced "**new** *d*," to clone the dynabase *d*. It is likely that the surface language will end up looking more as suggested in the original presentation, but its proposed normal form was difficult to adjust to μDyna, so this (expressively superior) alternative was chosen instead.

[152] That is, the dynabase where the handle is *stored* is not necessarily taken to be the owner. A rule of the form $\{(\texttt{t}\langle\rangle \underset{e}{\hookleftarrow} p) \overset{d}{\Leftarrow} \langle\cdots\rangle \mid \cdots\}$, interacting with this operator, creates a clone of p and assigns it to $\texttt{t}\langle\rangle$ on e while leaving d the owner.

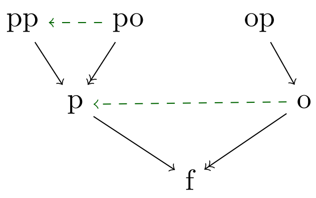


Figure 6.1: A possible arrangement of dynabases. The dynabase under study (the “focus”) is f at the base of the diagram. Progenitors are shown above-left (and with a single-tipped solid arrow) and owners are shown above-right (and with a double-tipped solid arrow pointing to the clone). Dashed arrows indicate other handles held: in order to make the clone from the progenitor, the owner must have had a handle to the progenitor. The names given read left-to-right relative to the focus: “op” is the focus’s owner’s progenitor. We have elided the owner’s owner and so on; they must exist for the progenitor relationship to have been established, but they are not yet relevant.

In the following, we shall use Ξ_d to refer to the rules intrinsic to the dynabase d; these do not include any inherited rules. That is, Ξ_d is the set of rules R as above for any cloned dynabase, or whatever rules are assumed on $\gimel$. Recall, the program itself, $\daleth$, is a *driver-constructed* clone of the $\gimel$; the rules of $\daleth$ are those of the given program. We shall use $\uparrow_p \cdot$ and $\uparrow_o \cdot$ to access the progenitor and owner of a dynabase, respectively.

The abstract semantics for interpreting a multi-dynabase program will involve *accumulating* rules subject to refinements from its progenitors and owners.[153] We now present this procedure first in a simplified way, in which we do not consider the effects of owners’ owners, and then in its full form, as presented in Eisner and Filardo [50]. We then ponder whether we can compile these semantics into the default reasoning of §3.4 and finish with some discussion of the impact of dynabases on static analysis.

6.2.1 Rule-Collection Semantics Without Recursive Owner Writes

Consider the series of dynabases shown in figure 6.1. The accumulated rules that influence the values of items on the dynabase under focus, f, would seem to be, roughly,

① The accumulated rules of its progenitor, though with their sense of self (i.e., DI and DO) set to f rather than p. (So that child items of parents that differ between p and f are also influenced appropriately.)

② The rules of f itself, with f as both DI and DO.

③ Any accumulated rules of its owner, o, which read from o (DI) and route their aggregands to f (DO).

There is an apparent asymmetry between progenitors and owners: while a progenitor’s owner’s (i.e., po’s) influence on the progenitor (p) is copied during cloning, the owner’s

[153] This procedure defines an *order* in which rules are visited, despite that nothing so far has indicated any sensitivity to rule order. In the surface language of Dyna, there is a “last definition wins” aggregator which is so sensitive; it desugars in μDyna by pairing values with some appropriately ordered index and maximizing across this component of aggregands. The order given here would, then, influence the selection of the appropriate indices when this aggregator is in use. So while we are careful to traverse the progenitor and owner relations in the correct order here, we otherwise ignore the order.

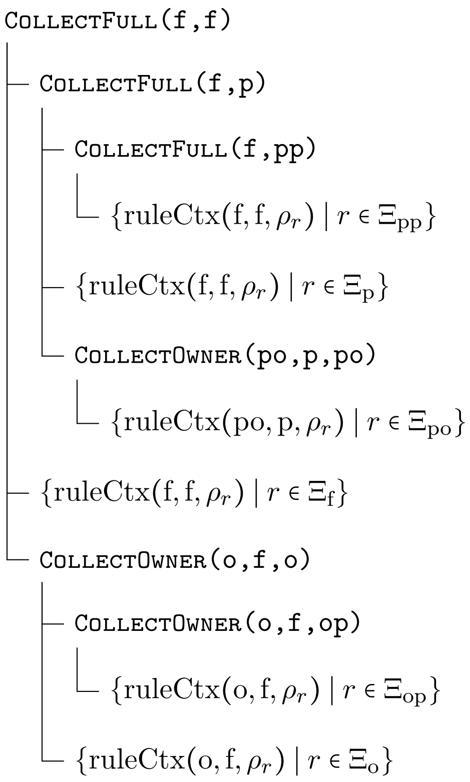

CollectFull(f,f)
CollectFull(f,p)
CollectFull(f,pp)
$\{\text{ruleCtx}(\text{f}, \text{f}, \rho_r) \mid r \in \Xi_{\text{pp}}\}$
$\{\text{ruleCtx}(\text{f}, \text{f}, \rho_r) \mid r \in \Xi_{\text{p}}\}$
CollectOwner(po,p,po)
$\{\text{ruleCtx}(\text{po}, \text{p}, \rho_r) \mid r \in \Xi_{\text{po}}\}$
$\{\text{ruleCtx}(\text{f}, \text{f}, \rho_r) \mid r \in \Xi_{\text{f}}\}$
CollectOwner(o,f,o)
CollectOwner(o,f,op)
$\{\text{ruleCtx}(\text{o}, \text{f}, \rho_r) \mid r \in \Xi_{\text{op}}\}$
$\{\text{ruleCtx}(\text{o}, \text{f}, \rho_r) \mid r \in \Xi_{\text{o}}\}$

Figure 6.2: A trace of the rule collection procedure for f in figure 6.1. Leaf nodes are set-valued and indicate the refined rules for items on f, while the root and internal nodes are the names of procedure calls. (All calls which would collect rules from ⅂ itself are suppressed.)

owner's (not shown) influence on the owner (o) is irrelevant, for the purposes of rule collection. Of course the owner's owner's influence is not entirely irrelevant: it influences the value of items *on the owner*, which may, in turn, influence items on the focus, but there is no *direct* relevance of rules from the owner's owner on the focus. A more subtle point arises from inheritance of rules from progenitors: during rule collection, when we are considering the impact of the progenitor's owner (po) on the focus, we must grant that this dynabase does not know the "name" of the focus, only the name of its progenitor (p). We therefore decree the following: while the output dynabase at DO is used during rule collection, it is then *ignored* when actually interpreting rules à la refineRuleSuffix and the rest of the machinery from §3. That is, *having collected the rules* and set DO appropriately, we consider only the *kv-pairs* $\{\langle t{\downharpoonleft}_{\text{HEAD}}, t{\downharpoonleft}_{\text{RES}}\rangle \mid t \in \epsilon\}$ where ϵ is a (non-ground) rule answer.

All said, we should define two recursive procedures:

① CollectOwner, which traverses just the progenitor links, starting from *an owner*, and

② CollectFull, which starts at the focus itself and traverses progenitor links and, for each, additionally collects the influence of the owners.

In both cases, the procedure will need to know the identity of three dynabases: the one whose intrinsic rules are to be included, the input dynabase for DI, and the output dynabase for DO. Moreover, in both cases, we need to set a rule's dynabase context (i.e., DI and DO) to singletons; let ruleCtx(e, h, ρ), for any rule ρ, be $\rho[\{e\}/\text{DI}][\{h\}/\text{DO}]$.

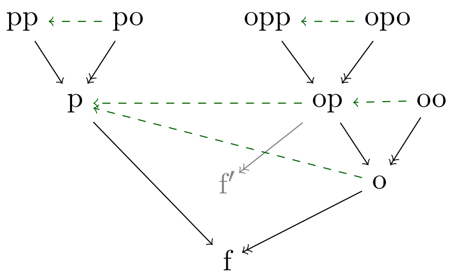


Figure 6.3: An extended view of figure 6.1. Notation and nomenclature is identical, save for the addition of the greyed f', which is f's owner's progenitor's analogue of f, should that exist. Not considered are any progenitors or owners of oo or opo; see figure 6.5 for that case and §6.2.2 for discussion.

These procedures are straightforward. For the first, let $\textsc{CollectOwner}(e,h,c)$ be the concatenation of the result of $\textsc{CollectOwner}(e,h,\uparrow_p c)$ and $\{\text{ruleCtx}(e,h,\rho_r) \mid r \in \Xi_c\}$. A call to $\textsc{CollectOwner}$ accumulates rules from progenitor-most downwards and restricts all of them to the same dynabase context. The second predicate is similar, just longer. Let $\textsc{CollectFull}(d,c)$ be the concatenation of

① $\textsc{CollectFull}(d,\uparrow_p c)$,

② $\{\text{ruleCtx}(d,d,\rho_r) \mid r \in \Xi_c\}$,

③ and $\textsc{CollectOwner}(\uparrow_o c, c, \uparrow_o c)$.

The first and second components traverse the progenitor relation, as with $\textsc{CollectOwner}$, but the third component brings each progenitor's owner's influence into play.

So armed, the set of rules that bear on items on the focused dynabase is simply $\textsc{CollectFull}(\mathtt{f},\mathtt{f},\mathtt{f})$. An execution trace, for the dynabases given in figure 6.1, is shown in figure 6.2.

6.2.2 Rule-Collection Semantics With Recursive Owner Writes

We now consider how to handle the possibility that the focus's owner's owner (and other owner's ⋯ owner's ⋯ dynabases) may assert aggregands on the focus. We extend the scenario from figure 6.1 to the one shown in figure 6.3 and wish to adjust the previous section's definitions, and in particular $\textsc{CollectOwner}$, so that oo and opo may *directly* influence items on the focused dynabase.

It is clear that we need to add another arm to $\textsc{CollectOwner}(e,h,c)$, specifically, one that considers rules from $\uparrow_o c$. However, while directly calling $\textsc{CollectOwner}(\uparrow_o c, h, \uparrow_o c)$ is tempting, and would work when considering the link between o and oo, it is incorrect when considering op and opo. While the focus's owner's progenitor's owner (opo) dynabase lacks the authority to contribute to the focus directly, it has permission, via op, to write to *an analogue* of the focus, created by op, just as o created the focused dynabase. Concretely, suppose that op (or opp) intrinsically contains the rule that, when collected into o, creates f. Then this rule, when collected into op, also creates another dynabase, which we denote f′. If oo clones op to produce o and does not otherwise alter o, the two should produce identical results in all respects, so the focus must see the aggregands routed to this analogous version.

To engineer a function from f to f′ within figure 6.3 requires that we unpack the actual identity of f and *construct* the corresponding identity of f′. Recall that a cloned

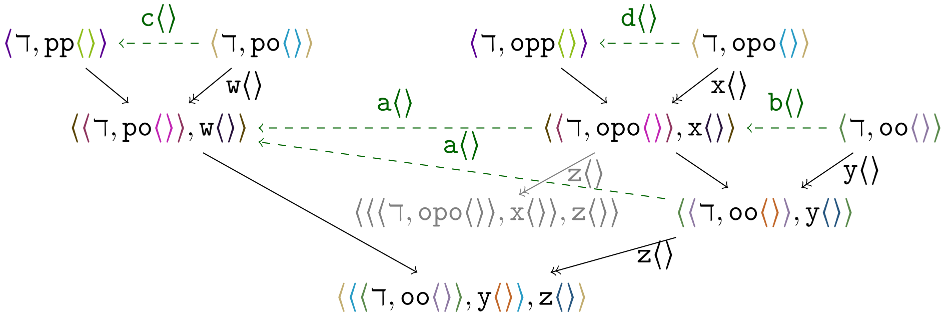


Figure 6.4: A redrawing of figure 6.3 using the naming mechanism proposed in §6.2.2. Each leaf dynabase is named by ℸ, the dynabase containing the program source, identically to the names we have been using to date (e.g., opo has the name $\mathtt{opo}\langle\rangle$ in ℸ). The primordial dynabases ℸ and ℷ are not, themselves, shown. Internal dynabases are then named by their owner and the name their owner gave them at cloning, which is indicated on the double-tipped arrows. Dashed arrows are labeled with the names of the existing handles to progenitors. For example, oo contains the rule $\{(\mathtt{y}\langle\rangle \underset{t}{\hookleftarrow} d) \overset{t}{\Leftarrow} \langle \mathtt{b}\langle\rangle \overset{t}{\mapsto} d\rangle \mid \cdots\}$, with $\mathtt{y}\langle\rangle$ cloning its sole aggregand. More importantly, opp contains the rule $\{(\mathtt{z}\langle\rangle \underset{t}{\hookleftarrow} d) \overset{t}{\Leftarrow} \langle \mathtt{a}\langle\rangle \overset{t}{\mapsto} d\rangle \mid \cdots\}$, with $\mathtt{z}\langle\rangle$ cloning; this rule not only creates the focus $\mathsf{f} = \langle\langle\langle ℸ, \mathtt{oo}\langle\rangle\rangle, \mathtt{y}\langle\rangle\rangle, \mathtt{z}\langle\rangle\rangle$, but also the analogous $\langle\langle\langle ℸ, \mathtt{opo}\langle\rangle\rangle, \mathtt{x}\langle\rangle\rangle, \mathtt{z}\langle\rangle\rangle$, depicted in grey.

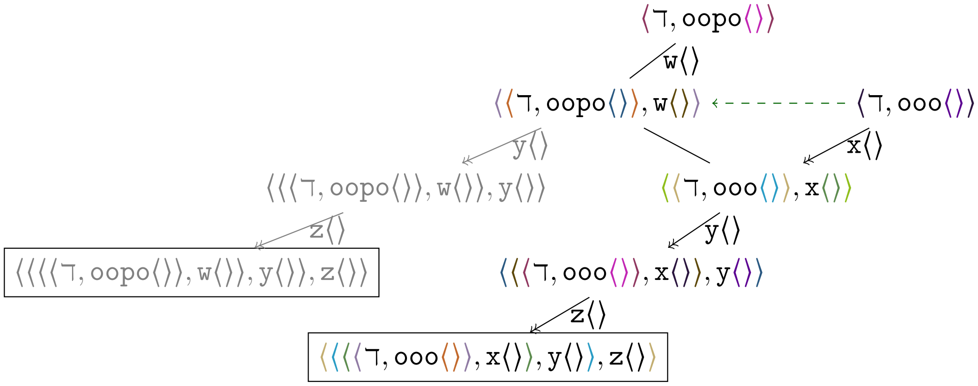


Figure 6.5: An arrangement of dynabases in which the focus (boxed, lower) has a transitive owner and a progenitor thereof. To reduce clutter, the progenitor of the focus's owner is not shown. To properly accumulate the rules that impact the focus, we must also consider rules that target the analogous version reachable from the focus's owner's owner's progenitor at the same name (boxed, upper, gray). This requires changing the notion of owner *deeply* within the recursive name of a dynabase: $\langle\langle ℸ, \mathtt{ooo}\langle\rangle\rangle, \mathtt{x}\langle\rangle\rangle$ is replaced by $\langle ℸ, \mathtt{oop}\langle\rangle\rangle$ within a context of $\langle\langle\cdot, \mathtt{y}\langle\rangle\rangle, \mathtt{z}\langle\rangle\rangle$. The intervening analogue of the focus's owner, $\langle\langle ℸ, \mathtt{oop}\langle\rangle\rangle, \mathtt{y}\langle\rangle\rangle$, is not explicitly visited by the recursive algorithm.

dynabase has four things that potentially identify it: its progenitor, its intrinsic rules, its owner, and *the name of the item* to which it is initially assigned within its owner. However, the progenitor and intrinsic rules are functionally determined by the owner and the name on it: there is only one aggregand to the clone operator, namely, the pair of progenitor and intrinsic rules! (This is somewhat akin to the names given to sub-expressions within "Nominal Adapton" [87].) Thus, we may *identify* a dynabase by the pair of its owner and initial name. This name is *recursive*: the owner is itself named by *its* owner and initial name. The ownership base case is $\daleth$. This gives us that $\mathcal{D}$ is the least fixed-point of the equation $\mathcal{D} = \{\gimel, \daleth\} \cup \langle \mathcal{D}, \mathcal{H} \smallsetminus \mathcal{D} \rangle$.[154] Using this idea, figure 6.3 can be redrawn as figure 6.4.

The specific case of f to f′ within figures 6.3 and 6.4 is achieved by replacing the owner component of f. However, this is, in general, insufficient. Consider the case shown in figure 6.5. Here, we see that the analogous version of the focus has multiple name elements in common with the focus; the owner field to be replaced is *deep* within the name. When traversing an ownership edge, then, we should keep track of the names we have crossed, so that we can construct this analogous version's name from the name of the dynabase whose rules are being collected. Thus, we replace the h parameter of $\textsc{CollectOwner}$ with a function that sends c to the appropriate analogue of the focus. All told, then, we replace the definition of $\textsc{CollectOwner}(e,h,c)$ from the previous section with $\textsc{CollectOwner}(e,n,c)$ (n an automorphism on dynabase names, i.e., $n \in \mathcal{D} \to \mathcal{D}$) defined as the concatenation of

① $\textsc{CollectOwner}(e, n, \upharpoonleft_p c)$,

② $\{\text{ruleCtx}(e, n(e), \rho_r) \mid r \in \Xi_c\}$, and

③ $\textsc{CollectOwner}(\upharpoonleft_o c, n \circ n', \upharpoonleft_o c)$ with $n' = \langle \cdot, c\downharpoonleft_2 \rangle = \{x \mapsto \langle x, c\downharpoonleft_2 \rangle \mid x \in \mathcal{D}\}$ being the function that adds the name component of c to the dynabase name x.

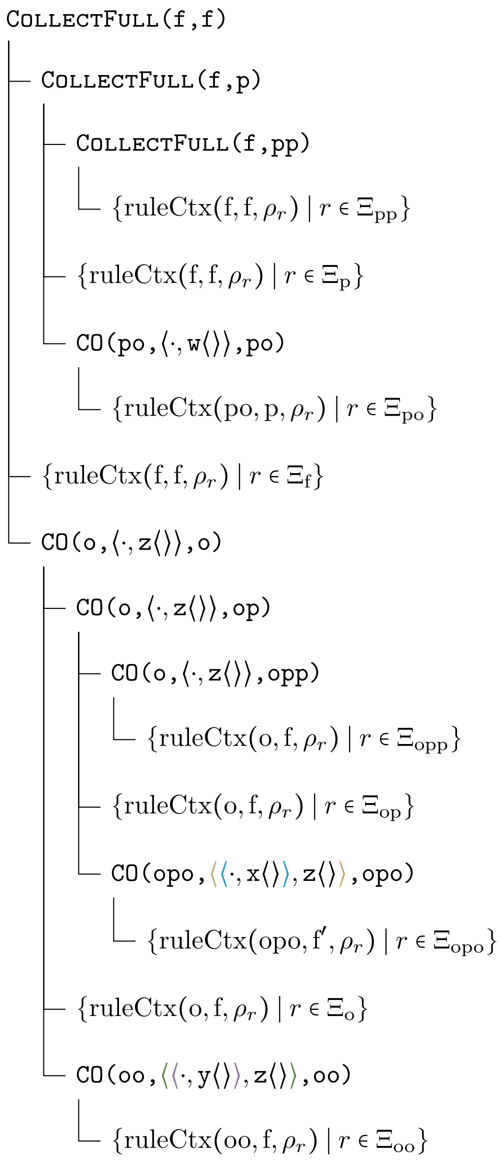
The definition of $\textsc{CollectFull}(d,c)$ is unaltered save to replace the call to $\textsc{CollectOwner}$ therein with $\textsc{CollectOwner}(\upharpoonleft_o c, \langle \cdot, c\downharpoonleft_2 \rangle, \upharpoonleft_o c)$. The central distinction between $\textsc{CollectFull}$ and $\textsc{CollectOwner}$ is in their second components: $\textsc{CollectOwner}$ uses n to reconstruct the focus (or analogue) from the name of the dynabase whose items are being used for valuation, e, and sets this reconstruction as DO, while $\textsc{CollectFull}$ always equates the DO and DI positions. The full collection for a focused dynabase f is $\textsc{CollectFull}(f,\ f)$. An example execution trace, for figure 6.3, is shown in figure 6.6.

6.2.3 Semantics of Multiple Dynabases

Having defined how to collect the rules associated with a dynabase (using either the method §6.2.1 or that from §6.2.2), we are finally in a position to describe the solutions to multi-dynabase programs. We now think of each dynabase as specifying a map from keys to values. The ability to specify handles other than the one provided at DI for subgoals within a rule allows these functions to inter-depend. The construction otherwise largely parallels §3.1.4.

Letting $\mathcal{E}_d$ and $\mathcal{I}_d$ denote the hyperedges and items of the dynabase d, we revise the definitions of our edge- and node-labeling functions thus:

[154] We have prohibited the use of a dynabase handle as a bare item name, and may thus exclude $\mathcal{D}$ from $\mathcal{H}$ within this recursion.


COLLECTFULL(f,f)
COLLECTFULL(f,p)
COLLECTFULL(f,pp)
{ruleCtx(f, f, ρ_r) | $r \in \Xi_{\text{pp}}$}
{ruleCtx(f, f, ρ_r) | $r \in \Xi_{\text{p}}$}
CO(po,⟨·,w⟨⟩⟩,po)
{ruleCtx(po, p, ρ_r) | $r \in \Xi_{\text{po}}$}
{ruleCtx(f, f, ρ_r) | $r \in \Xi_{\text{f}}$}
CO(o,⟨·,z⟨⟩⟩,o)
CO(o,⟨·,z⟨⟩⟩,op)
CO(o,⟨·,z⟨⟩⟩,opp)
{ruleCtx(o, f, ρ_r) | $r \in \Xi_{\text{opp}}$}
{ruleCtx(o, f, ρ_r) | $r \in \Xi_{\text{op}}$}
CO(opo,⟨⟨·,x⟨⟩⟩,z⟨⟩⟩,opo)
{ruleCtx(opo, f′, ρ_r) | $r \in \Xi_{\text{opo}}$}
{ruleCtx(o, f, ρ_r) | $r \in \Xi_{\text{o}}$}
CO(oo,⟨⟨·,y⟨⟩⟩,z⟨⟩⟩,oo)
{ruleCtx(oo, f, ρ_r) | $r \in \Xi_{\text{oo}}$}

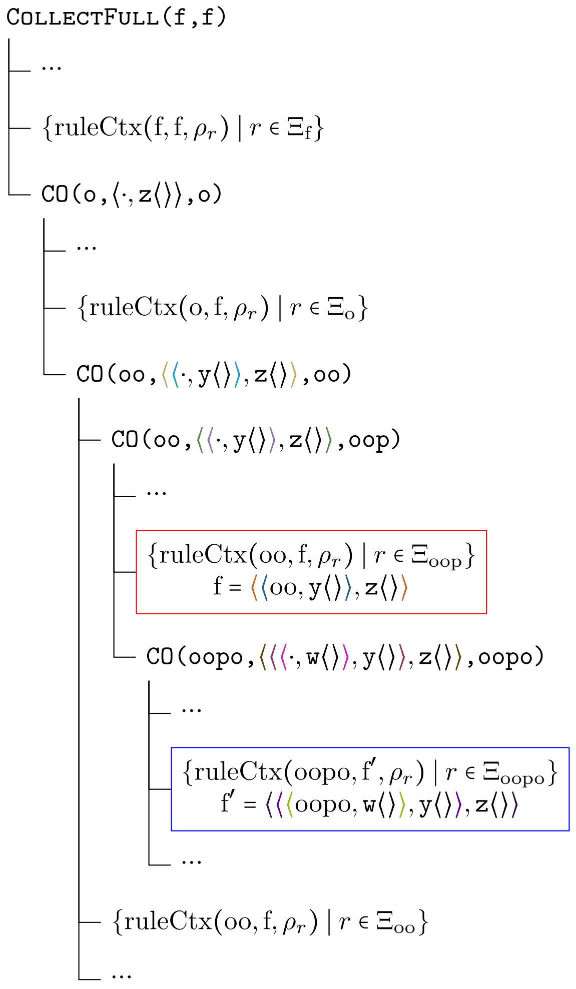

COLLECTFULL(f,f)
…
{ruleCtx(f, f, ρ_r) | $r \in \Xi_{\text{f}}$}
CO(o,⟨·,z⟨⟩⟩,o)
…
{ruleCtx(o, f, ρ_r) | $r \in \Xi_{\text{o}}$}
CO(oo,⟨⟨·,y⟨⟩⟩,z⟨⟩⟩,oo)
CO(oo,⟨⟨·,y⟨⟩⟩,z⟨⟩⟩,oop)
…
{ruleCtx(oo, f, ρ_r) | $r \in \Xi_{\text{oop}}$}
f = ⟨⟨oo, y⟨⟩⟩, z⟨⟩⟩
CO(oopo,⟨⟨⟨·,w⟨⟩⟩,y⟨⟩⟩,z⟨⟩⟩,oopo)
…
{ruleCtx(oopo, f′, ρ_r) | $r \in \Xi_{\text{oopo}}$}
f′ = ⟨⟨⟨oopo, w⟨⟩⟩, y⟨⟩⟩, z⟨⟩⟩
…
{ruleCtx(oo, f, ρ_r) | $r \in \Xi_{\text{oo}}$}
…

Figure 6.6: A trace of the rule collection procedure for f in figure 6.3 (using dynabase names as in figure 6.4), paralleling figure 6.2. (For type-setting reasons, CollectOwner is abbreviated to CO.) The value f′ is computed as (⟨⟨·, x⟨⟩⟩, z⟨⟩⟩)(opo). The corresponding computation within the last CO call obtains f itself by evaluating (⟨⟨·, y⟨⟩⟩, z⟨⟩⟩)(oo). (As before, all calls which would collect rules from ¬ are suppressed for brevity.)

Figure 6.7: A partial execution trace of figure 6.5. (For type-setting reasons, CollectOwner is abbreviated to CO.) The boxed nodes highlight the importance of deriving the focus (in red, upper) and focus analogue (in blue, lower) from the current *evaluation* dynabase (the first argument to CollectOwner) rather than the dynabase whose rules are being collected (the third argument). The names used here are of a kind with earlier drawings, in that they refer to dynabases relative to the focus, but they take on different actual values than in past figures: for example, f = ⟨⟨⟨⟨¬, ooo⟨⟩⟩, x⟨⟩⟩, y⟨⟩⟩, z⟨⟩⟩ and f′ = ⟨⟨⟨⟨¬, oopo⟨⟩⟩, w⟨⟩⟩, y⟨⟩⟩, z⟨⟩⟩

① $\mathrm{el} \in (\Pi_{d \in \mathcal{D}}(\mathcal{I}_d \to \mathcal{H}')) \to (\Pi_{d \in \mathcal{D}}(\mathcal{E}_d \to \mathcal{H}'))$, and

② $\mathrm{nl} \in (\Pi_{d \in \mathcal{D}}(\mathcal{E}_d \to \mathcal{H}')) \to (\mathcal{I}_{\mathrm{inp}} \to \mathcal{H}') \to (\Pi_{d \in \mathcal{D}}(\mathcal{I}_d \to \mathcal{H}'))$.

External input to $\mathcal{I}_{\mathrm{der}}$ is only possible on the program's initial dynabase $\neg$. The effect of an element $x = \mathrm{selt}(\{(h \underset{o}{\leftarrow} v) \overset{i}{\Leftarrow} \langle t_1 \overset{d_1}{\mapsto} v_1, \cdots, t_k \overset{d_k}{\mapsto} v_k \rangle\})$ of a rule collected onto the dynabase d is to set $\mathrm{el}(n)(d)(x) = v$ when $\forall_i\, n(d_i)(t_i) = v_i$ and NULL otherwise. (Recall that o need not equal d: the collection procedure has taken care of ensuring that rules were copied to the correct place and refined appropriately.) The node labels are determined entirely as before, within each dynabase, by aggregating the bag of labels of hyperedges whose targets are a given head.

Conceptually, there is some overlap between external input ($\mathcal{I}_{\mathrm{inp}}$) and dynabase extension. The present section has been written from the perspective of an extension to the existing language, which achieved its dynamic behavior through the notion of these external inputs to circuits. An alternative design would achieve reactivity by temporally evolving the notion of "current initial dynabase," viewing each external update as a three-step procedure of cloning $\neg$, asserting new aggregands on this clone, and *replacing* the notion of the current $\neg$ with this clone. (We trust the reader will forgive us some laxity in the description; we are presuming some "even more initial" dynabase which can own the linear temporal history of $\neg$ clones.) While such a description is amenable to an interpretation as an *immutable* data structure, and as such is quite attractive, it is, nevertheless, slightly at odds with the (relatively) simple story given herein for the behavior of dynabases *within* the solver.

6.2.4 Execution of Multiple Dynabases

While collecting rules gave us a *semantic* definition of a multi-dynabase program, it does not provide a reasonable execution strategy. We have sought to develop a strategy in which rules are not copied, but remain firmly anchored to individual dynabases. Rather, we would like aggregands to be cloned along the inheritance hierarchy, relying on the AC-reducer nature of aggregators to combine additional aggregands as they are encountered within clones. Unfortunately, we have not yet obtained a satisfactory answer.

However, while on the topic of execution, we forsee a significant role for a range-restriction-like requirement on programs. In particular, we will require that the solver never needs to make a subgoal query in which *the dynabase* whence it is to find answers is not a singleton (i.e., ground). That is, at each stage of backward-chaining's REFINERULESUFFIX, even within the non-ground reasoning strategies proposed, the combination of the specified DO and DI, as well as any prior subgoals must be enough to bring the SG.i.3 projection to a singleton. This restriction means that the solver always knows the *unique* place to look for subgoals. When forward-chaining, DO will not be specified up front and must, instead, be ground by the time the rule has been refined by answers from all subgoals, even if the head is non-ground.

Compiling to Default Reasoning An interesting avenue of future work arises: rather than introduce the novel complexities of rule collection and the "open" nature of inheritance, can we instead somehow compile a multi-dynabase program to a *single-dynabase* one using the default reasoning of §3.4? This would, seemingly, obviate the need for explicit accumulation of rules. Likely, such abilities would require dramatically different names than we have developed for the semantics of this section. Our names here are based on the *owner* and the *initial name* of a dynabase, while default reasoning would seem to necessitate that the names be centered around the *progenitor* relationship.

6.2.5 Static Analysis of Multi-Dynabase Programs

When bringing the static analyses of §5 over to the new, multi-dynabase world, mode analysis will let us enforce our range-restriction-like requirement, that we always know precisely which dynabase handle is to be used. However, an interesting new question arises: how do we know that the query we are currently considering making is well-moded *in the selected dynabase*?

It must be that the *type* of a dynabase *carries enough information to answer mode questions*. That is, our static analyser must characterize subsets of $\mathcal{D}$ not by shape (as it did subsets of $\mathcal{H}$) but by knowledge of permitted query and update procedures. Orthogonally, one should characterize $d \in \mathcal{D}$ as being owned or not by the current dynabase. As is so often the case, one could choose to enforce that all dynabase handles arising in the DO position are *certainly* owned by the current dynabase, with the implication that some clever, dynamic routing of handles throughout the system will not be permitted, or choose to permit execution of the program even when this is not certainly true, at the cost of runtime exceptions. Unlike the error handling proposed in §6.1, in such a case there is no obvious head item to which the resulting error value is to be given; instead, we propose either a per-dynabase recipient of errors (i.e., some built-in item, `$ok⟨⟩`, which typically has value `true⟨⟩` but has an error value at least when such a bad contribution has occurred and has not been superseded by subsequent forward-chaining) or the termination (i.e., crashing with a sufficiently useful error message) of the solver.

6.3 Looseness of Program to Hypergraph Mapping

The translation of programs to hypergraphs given in §3.1.4 is not a very tight encoding, in that there is redundancy in the graph structure. Consider a rule wherein a value of one subgoal covaries with the key of another, e.g., $\rho_r = \{(\mathtt{f}\langle\rangle \hookleftarrow w) \Leftarrow \langle \mathtt{a}\langle\rangle \mapsto v, \mathtt{b}\langle v\rangle \mapsto w\rangle \mid \cdots\}$. Since the translation derives edges from nontrivial rule queries for each rule, this rule will give rise to edges $\phi = \{\langle r, \langle \mathtt{f}\langle\rangle, \langle \mathtt{a}\langle\rangle, \mathtt{b}\langle x\rangle\rangle\rangle\rangle \mid \mathtt{b}\langle x\rangle \in \mathcal{I}\}$, which may be an infinite collection. However, within every rule answer, the second subgoal is completely determined by the first, meaning that, for any given $l^{\mathrm{n}}_{\cdot}$, all but at most one of the ϕ edges *must* have consistent label NULL.

A tighter encoding, more reflective of both the program's semantics and the solver's behavior, would remove these spurious degrees of freedom from the edge indexes. The notion of a rule query would be, most generally, replaced by that of a set of singleton refinement operations on a prefix-free set of paths within ρ that, when combined with $l^{\mathrm{n}}_{\cdot}$, would be sufficient to identify a single element of ρ. In the above example, *the empty set* suffices. In a more intricate example such as $\{(\mathtt{f}\langle h, x, v\rangle \hookleftarrow w) \Leftarrow \langle \mathtt{a}\langle x, z\rangle \mapsto v, \mathtt{b}\langle y, z, v\rangle \mapsto w\rangle \mid \cdots\}$, a rule query would cover HEAD.1 (h), HEAD.2 (x; SG.1.1 would work as well), SG.1.2 (z), and SG.2.1 (y), these being sufficient to identify (by being used as paths for singleton refinements) a subset with at most one element consistent with a node labeling.

This encoding has proven itself useful within the planner of a prototype μDyna compiler, but only on a per-rule basis. Planning subgoals amounts to finding a kind of node cover of a hypergraph where the paths within a rule query form the nodes (in practice, we have used a more traditional, variable-based encoding rather than the more general, but set-theoretic, notion of paths of this paper). One of our collaborators is investigating an extension to work across rules by unfolding items' definitions. However, even so, it is not clear how to map this tighter notion back to the kind of computational circuit model we started with: if these sets of refinements are the edges of the graph, what are the nodes?

6.4 A Taxonomy of Database APIs

The data structures considered by this document are intended to support exclusively **fully-simplified** responses to queries. That is, while "`1+2`" is indeed *a kind* of answer to the question of "Set `a` to `1` and `b` to `2`; what is `a+b`?" (as is "`a+b`", as unhelpful as that might be), we consider the only acceptable answer to be `3`. Of course, *internally*, algorithms which implement the specification given herein are free to use other, partially simplified answers at points in their computation, but these *will not* be presented to the user. This has implications on the space of programs which can be considered valid; e.g., $\{(\texttt{goal}\langle\rangle \leftarrowtail v) \Leftarrow \langle\texttt{posint}\langle x\rangle \mapsto 1, \texttt{r2}\langle x\rangle \mapsto v\rangle \mid v, x\}$ (using as primitive the relations $\{\langle\texttt{posint}\langle x\rangle, v\rangle \mid v \in \{0,1\} \wedge (v = 1 \Leftrightarrow (x \in \mathbb{Z} \wedge x > 0))\}$ and $\{\langle\texttt{r2}\langle x\rangle, v\rangle \mid x, v \in \mathbb{R}, v = 1/x^2\}$) must be forbidden by implementations which reach a fixed point by enumeration of values for all possible instantiations of variables, which are most of the systems we have considered, despite that more clever systems could conclude that `goal`, if summed, would have value $\pi^2/6$ [55].

Much of the development in this thesis is already suitably generalized to support the case of responses enriched by *constructors over projections of the query*, i.e., where the responses to queries may *structurally covary* with the query itself.[155] That is, while there is no *invariant*, fully-simplified response to "What is the value of `id(X)`?", when `id` is defined to be the identity function, there is nevertheless an answer using constructors-over-projectors: `id(X)` has value `X` (i.e., the first subterm projection of the query). However, the full story—especially the *production* of these kinds of responses by aggregators—rapidly becomes sufficiently complicated that we must leave the problem to future work, though we believe that the steps made in this document shine light in a useful direction.

[155] The full-simplification constraint of the previous paragraph is a trivial special-case of these answers in which there are no projectors, just term constructors, used in the responses to queries.

Conclusion

> I realized, the moment I fell into the fissure, that the Book would not be destroyed as I had planned. It continued falling into that starry expanse, of which I had only a fleeting glimpse. I have tried to speculate where it might have landed—I must admit, however, such conjecture is futile. Still, questions about whose hands might one day hold my Myst Book are unsettling to me. I know my apprehensions might never be allayed, and so I close, realizing that perhaps the ending has not yet been written.
>
> Cyan Worlds. *Myst.* Atrus's opening voiceover.

We have considered the design, theoretical, and practical aspects of a novel, purely-declarative weighted logic programming language, Dyna. During the course of our work, the language has evolved from a simple but useful toolkit, as in Eisner, Goldlust, and Smith [51], towards a language designed for programming "in the large" and capturing entire program pipelines, with a focus on statistical natural language processing, as shown in Eisner and Filardo [50]. As might be expected, such an ambitious target is an endless fount of research topics, some of which we have solved or addressed in this document.

In §2, we explored Dyna's primordial landscape: finite arithmetic circuits. We reviewed the natural semantics for acyclic (§2.1.2) and cyclic (§2.5.1) circuits. We then gave a pair of solver frameworks (§2.2.4 and §2.3) which could *smoothly interpolate* between the two existing dominant solver strategies (i.e., forward- and backward-chaining). These frameworks were shown to be amenable to a series of extensions which brought additional strategies into scope for the *optional* use of the solver (§2.4). However, while finite circuits may be useful, they fall shy of actually describing the complete set of things expressible within a Dyna program, to which we next turned our attention.

In §3, we considered the surprisingly challenging problem of generalizing our circuit-based solver frameworks to reasoning on infinite circuits described by Dyna programs. We gave a formalized, core language and semantics, μDyna (§3.1), which pared Dyna down to its essence of collecting and manipulating *sets of* circuit nodes and edges at once. We then warmed up by revisiting our finite-circuit playground, now viewed through the lens of μDyna programs (§3.2). Having so warmed up, we then considered the task of backward-chaining within general μDyna programs. Our first approach (§3.3) assumed the ability to maintain piecewise-constant functions, using set subtraction within updates to maintain the requisite partition of the domain. We then showed (§3.4) that this use set subtraction is not fundamental, by giving an algorithm which uses *default reasoning* and eliminates set subtraction, replacing it with a *cardinality of subtraction* oracle.

In §4, we reviewed some options for *computational representations* of (in)finite sets.

Such representations are crucial for both execution and analysis of Dyna. We discussed in detail the popular WAM design (§4.1) and a larger notion of *automaton* encodings of set membership predicates (§4.2). We related these automata to the finite circuits of §2 (§4.3) and used this insight to guide the design of automata whose languages are *sets of sets of trees* (§4.4).

In §5, we considered several different analyses of μDyna programs, operating on different levels of abstraction over the program's runtime behavior. In §5.2 we considered *type*-based analysis of the semantics of the μDyna program, intended to detect possible programmer error. In type-based reasoning, we centrally considered a notion of optimistic types which included all non-NULL valuations of all items (within the permitted query set). In §5.3 we considered *instantiation-state*-based analysis of the μDyna program and, in particular, the *execution* of the rule *bodies* within a solver, by studying the composition of *procedures* which back the abstract Lookup procedure of the algorithms of §3. In §5.4, we turned our attention briefly to statically describing the *aggregation* of results within the solver's execution.

And last, in §6, we considered a series of revisions to Dyna itself. The most significant of these is the "dynabase" extension, which gives a first-class module system to Dyna (§6.2).

In addition to the work directly contained within this document, Tim Vieira and the author have implemented an end-to-end prototype of the Dyna language, in 2013. This prototype, mentioned at a few points throughout the thesis, was essential to the development of §3.2 and §5.3 (indeed, the prototype's Dyna compiler component grew out of an effort to more fully understand Overton [137]). This prototype was sufficiently useful to have been used as part of a summer course at the University of Michigan designed to expose student linguists to *computational* linguistics, despite its support for a subset of the Dyna 2 language. Separately, there is a "reference implementation" of our first mixed chaining algorithm, as detailed in §2.2.4, which can be used to investigate possible solver behaviors when applied to small circuits. The insights gained in the development of these two artifacts have already contributed to this thesis and to the successor implementations of Dyna.

We hope that the frameworks proposed in this thesis prove useful to those who will come after us in their efforts to increase the expressive power of the Dyna declarative weighted logic language (or, indeed, any similar undertaking) and to achieve the grand vision of *executable, mathematical specifications* of (numeric) programs.

Dyna 1 vs Dyna 2 Having all of the machinery under our belt, we are, at last, in a position to offer a more detailed comparison between Dyna 1 and Dyna 2, with focus on the new expressive power in the latter.

Weight Algebra: Dyna 1 enforced the use of a single semiring throughout the program, using its multiplicative operator $\otimes$ to combine the weights of subgoals and its additive operator $\oplus$ as the aggregator for each item. Dyna 2 has a more generic mechanism of expressions of subgoals and per-item aggregators; in μDyna, these are flattened, without loss of expressive power, to the conjunctive rule body. Dyna 1 did not distinguish between NULL and the semiring 0 element. Dyna 2 introduces NULL to facilitate the use of multiple semirings within a single program. (Recall §3.1.2.1 and, in particular, footnote 73 therein.)

Weight Isolation: The elements of the item value semiring of a Dyna 1 program are disjoint from the Herbrand universe used to construct item names. In Dyna 2, the two may freely intermingle, creating novel challenges for both theory and algorithms. The μDyna formalism in §3.1 (resp. default-based solver in §3.4) overcomes these challenges through the distinction between the θ and ϵ (resp. α_k and α_v) sets.

Solver Internals: The solver implemented for Dyna 1 was a *fully-materialized* (i.e., no partial memoization), purely forward-chaining (recall §2.2.3) system which used only *delta* messages on the agenda (recall §2.4.3). When using semirings without a notion of subtraction (e.g., the Tropical semiring $\langle \mathbb{R}\cup\infty, \min, \infty, +, 0\rangle$), the Dyna 1 solver is unable to handle non-monotonic behavior of aggregands.[156]

Static Analysis The Dyna 1 system had a relatively limited suite of static analysis capabilities. The efforts of §4 and §5 have been done with an eye towards making more *tailored* solver code for individual programs.

Suggestions for Future Work Throughout the thesis, we have identified several avenues for future investigation. Further, several of our designs are *modular*, in that they are parametric in the choice of underlying machinery. For ease of reference, we enumerate future work and parametric decisions here, with a brief summary for each.

① §2.2.4.4 and §2.4.2 suggest that the solver may be able to return answers from queries even before the agenda has emptied, by either passive detection or *active scheduling of work* to promptly bring queried items into convergence. There are implementation details of metadata maintenance that remain to be worked out.

② §2.2.4.5 and §2.3 lay groundwork for a *multi-threaded* finite solver algorithm, but there is not, of yet, an implementation taking advantage of these ideas.

③ §2.3.4.1 introduces a large space for designs for summary information stored about items' values or parents, with the intent of speeding up processing of notifications or being able to respond to *predicate* queries of the value. Investigating this space and the relative merits of designs here likely represents a large piece of future work for the evolution of circuit solvers.

④ §2.3.5.3 ponders passing increasingly verbose messages *backwards* along the circuit with the aim of enabling *earlier* recognition that changes have no *further* effect on the values of the circuit.

⑤ §2.4 details several theoretical extensions to the (forward) messages of the circuit solver. The question of *when* such messages should be used within the solver's evolution is a difficult problem, requiring both analysis of the circuit and *runtime prediction*

[156] Some special handling, achieved by *restarting* child-closed segments of the computation, exists for the external driver's use.

of future benefits. Such considerations fall within the realm of ongoing work as part of the successor implementations of Dyna; see Vieira et al. [183] for further discussion.

⑥ §3.1 introduced μDyna, a core representation for per-rule reasoning of Dyna programs. Aggregation was, notably, *implicit* in this representation; an explicit representation, along the lines of the "superhomogeneous normal form" of Mercury's compiler [166], would increase the utility of this representation for *program rewrites*. One of the successor implementations of Dyna has made inroads into this effort, but is not as set-centric in its formalism as μDyna.

⑦ The AnswerFor oracle of §3.4.1 will need to be implemented within any implementation of default reasoning. The algorithm of §3.4 is, in a sense, parametric in this oracle; more powerful oracles enable more programs to be solved by default reasoning.

⑧ As first suggested in §3.4.1 but reinforced by §4.2, while it appears that no *single* automaton class will be sufficient for a fully general Dyna solver (absent relatively strong restrictions on programs), it may be possible to extend the compiler's planner to consider multiple automaton classes and transfer of representation between classes.

⑨ §3.4.5 and §3.5 suggest several avenues of optimization of our default reasoning algorithm. Most are straightforward and likely to benefit real-world implementations (at the expense of simplicity of exposition).

⑩ §3.6.1 introduces, but does not complete, an extension to non-ground reasoning to permit *lifted* forms of head-value covariance, where projections of the head are re-used in aggregands.

⑪ §3.6.2 discusses extending non-ground reasoning to forward chaining. We have *implemented* some of the material of this section in the 2013 prototype, under assumptions that limited us to (essentially) ground reasoning. Thus, we know the machinery to be sensible, but have not considered its general extension to compatibility with the non-ground backward chaining of §3. Combining the two is likely of substantial importance to any implementation of Dyna 2 and will build upon the foundational work in §3.

⑫ §5.2.4.1 assumed a function for subgoal cardinalities (sgc) which we have taken as oracular; such a function must be provided by an implementation of the analysis of this section.

⑬ §5.3 details our inst-based *conjunctive* static analysis from a set-centric perspective; an implementation must find a suitable abstract representation or tree set automata family or implement *three-way* operations (i.e., those capable of indicating "certainly yes," "certainly no," or "could be either."). The latter offers promising opportunity for exchanging thoroughness of analysis for shorter compilation times.

⑭ §5.3.6 discussed, but did not pursue, tracking determinism, within a rule's answer inst, of projections beyond the rule's subgoals. While likely interesting, we lack a use case that definitely benefits from this increased analysis.

⑮ §5.4 leaves open the problem of ensuring, in general, that the program's purported contents inst ($\mathfrak{C}$) is in fact closed under the program's operation. Sufficiently coarse, yet still useful, $\mathfrak{C}$ are likely verifiable as closed, even so.

⑯ §6.2.4 raised the question of *compiling* the complex multi-dynabase rule-collection semantic mechanism into default reasoning.

⑰ §6.4 ponders extending Dyna into a kind of term-rewriting system by adding *unevaluated expressions* to the language of answers to queries, an even broader extension than those of §3.6.1.

Index of Definitions

Index of Notation

Bibliography

[1] *λProlog*. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/ (page 3).

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. “Adaptive Functional Programming”. In: *Proc. of POPL*. 2002, pp. 247–259. DOI: 10.1145/503272.503296. URL: http://www.mpi-sws.org/~umut/papers/popl02.html (page 68).

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. “Selective Memoization”. In: *Proc. of POPL*. 2003, pp. 14–25. DOI: 10.1145/640128.604133. URL: http://www.mpi-sws.org/~umut/papers/popl03.html (page 68).

[4] Umut A. Acar and Ruy Ley-Wild. “Self-Adjusting Computation with Delta ML”. In: *Advanced Functional Programming*. Ed. by Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer, 2008, pp. 1–38. ISBN: 978-3-642-04651-3. DOI: 10.1007/978-3-642-04652-0_1. URL: http://www.mpi-sws.org/~umut/papers/afp08.html (pages 5, 68).

[5] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5 (page 44).

[6] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. “DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views”. In: *Proc. VLDB Endow.* 5.10 (June 2012), pp. 968–979. ISSN: 2150-8097. DOI: 10.14778/2336664.2336670 (pages 2, 35).

[7] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. 1st ed. Addison-Wesley, 1974. ISBN: 978-0201000290 (page 124).

[8] Hassan Aït-Kaci. *Warren’s Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991. ISBN: 0-262-01123-9. URL: http://wambook.sourceforge.net/ (page 120).

[9] K. R. Apt, H. A. Blair, and A. Walker. “Foundations of Deductive Databases and Logic Programming”. In: ed. by J. Minker. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. Chap. Towards a Theory of Declarative Knowledge, pp. 89–148. ISBN: 0-934613-40-0. URL: http://dl.acm.org/citation.cfm?id=61352.61354 (page 114).

[10] Krzysztof R. Apt and M. H. van Emden. “Contributions to the Theory of Logic Programming”. In: *J. ACM* 29.3 (July 1982), pp. 841–862. ISSN: 0004-5411. DOI: 10.1145/322326.322339 (page 194).

[11] Krzysztof R. Apt and M. H. van Emden. *Contributions to the Theory of Logic Programming*. Tech. rep. CS-80-12. Later published as [10]. University of Waterloo, Department of Computer Science. URL: https://cs.uwaterloo.ca/research/tr/1980/CS-80-12.pdf (page 15).

[12] Krzysztof R. Apt and Elena Marchiori. *Reasoning About Prolog Programs: From Modes Through Types to Assertions*. Tech. rep. Amsterdam, The Netherlands, The Netherlands, 1993 (page 153).

[13] Michael Arntzenius and Neelakantan R. Krishnaswami. "Datafun: A Functional Datalog". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: ACM, 2016, pp. 214–227. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951948. URL: http://www.cs.bham.ac.uk/~krishnan/datafun.pdf (page 3).

[14] A. K. Austin. "Miscellanea: Modern Research in Mathematics". In: *The American Mathematical Monthly* 84.7 (1977), pp. 566–566. DOI: 10.2307/3614400 (page 195).

[15] *B-Prolog*. URL: http://www.picat-lang.org/bprolog/ (page 113).

[16] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. "Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract)". In: *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. PODS '86. Cambridge, Massachusetts, USA: ACM, 1986, pp. 1–15. ISBN: 0-89791-179-2. DOI: 10.1145/6012.15399 (page 40).

[17] Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. "Decidable Classes of Tree Automata Mixing Local and Global Constraints Modulo Flat Theories". In: *Logical Methods in Computer Science* 9.2 (2013). DOI: 10.2168/LMCS-9(2:1)2013. URL: https://arxiv.org/abs/1302.6960 (page 127).

[18] Catriel Beeri and Raghu Ramakrishnan. "On the power of magic". In: *The Journal of Logic Programming* 10.3 (1991), pp. 255–299. ISSN: 0743-1066. DOI: 10.1016/0743-1066(91)90038-Q (page 81).

[19] Gérard Berry. *Real time programming : special purpose or general purpose languages*. Research Report RR-1065. INRIA, 1989. URL: https://hal.inria.fr/inria-00075494 (page 2).

[20] B. Bogaert and S. Tison. "Equality and disequality constraints on direct subterms in tree automata". In: *STACS 92: 9th Annual Symposium on Theoretical Aspects of Computer Science Cachan, France, February 13–15, 1992 Proceedings*. Ed. by Alain Finkel and Matthias Jantzen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 159–171. ISBN: 978-3-540-46775-5. DOI: 10.1007/3-540-55210-3_181 (page 125).

[21] A. Borodin and I. Munro. *The computational complexity of algebraic and numeric problems*. Elsevier, 1975. ISBN: 0444001689 (page 22).

[22] Stefan Brass. "Range Restriction for General Formulas". In: *Proceedings of the 23rd Workshop on (Constraint) Logic Programming*. 2009. URL: http://users.informatik.uni-halle.de/~brass/wlp09.pdf (pages 80, 103).

[23] T. Brown. “Infinite multi-variable subpolynormal Woffles which do not satisfy the lower regular Q-property (Piffles)”. In: *A Collection of 250 Papers on Woffle Theory Dedicated to R. S. Green on His 23rd Birthday.* Cited in Austin [14]; we follow Graham, Knuth, and Patashnik [81] in not citing such works.

[24] Randal E. Bryant and David Richard O’Hallaron. *Computer Systems: A Programmer’s Perspective.* Third. Pearson, 2016. ISBN: 978-0-13-409266-9. URL: http://csapp.cs.cmu.edu/ (page 18).

[25] S. Ceri, G. Gottlob, and L. Tanca. “What You Always Wanted to Know About Datalog (And Never Dared to Ask)”. In: *IEEE Transactions on Knowledge and Data Engineering* 1 (1989), pp. 146–166. ISSN: 1041-4347. DOI: 10.1109/69.43410 (page 23).

[26] Weidong Chen, Terrance Swift, and David S. Warren. “Efficient top-down computation of queries under the well-founded semantics”. In: *The Journal of Logic Programming* 24.3 (1995), pp. 161–199. ISSN: 0743-1066. DOI: 10.1016/0743-1066(94)00028-5 (page 17).

[27] Weidong Chen and David S. Warren. “Tabled Evaluation with Delaying for General Logic Programs”. In: *J. ACM* 43.1 (Jan. 1996), pp. 20–74. ISSN: 0004-5411. DOI: 10.1145/227595.227597 (page 17).

[28] Alonzo Church. “Summaries of talks presented at the Summer Institute for Symbolic Logic”. In: 1. Second edition printed in 1960, with numerous edits and additions to this article. Cornell University, 1957. Chap. Applications of recursive arithmetic to the problem of circuit synthesis, pp. 3–50 (page 15).

[29] *The Ciao Prolog Development System.* URL: http://www.ciaohome.org/ (pages 3, 113).

[30] Keith L. Clark. “Negation as failure”. In: *Logic and Data Bases.* Ed. by Hervé Gallaire and Jack Minker. New York, London: Springer US, 1978, pp. 293–322. ISBN: 978-1-4684-3384-5. DOI: 10.1007/978-1-4684-3384-5_11. URL: http://www.doc.ic.ac.uk/~klc/NegAsFailure.pdf (pages 6, 17).

[31] John Cocke. “Global Common Subexpression Elimination”. In: *Proceedings of a Symposium on Compiler Optimization.* Urbana-Champaign, Illinois: ACM, 1970, pp. 20–24. DOI: 10.1145/800028.808480 (page 21).

[32] Sara Cohen, Werner Nutt, and Alexander Serebrenik. “Algorithms for Rewriting Aggregate Queries Using Views”. In: *Proc. of ADBIS-DASFAA.* London, UK: Springer-Verlag, 2000, pp. 65–78. ISBN: 3-540-67977-4. DOI: 10.1007/3-540-44472-6_6. URL: https://arxiv.org/abs/cs/0011024 (pages 3, 6, 67, 75, 116).

[33] Alain Colmerauer. “Opening the Prolog III Universe”. In: *BYTE* 12.9 (Aug. 1987), pp. 177–182. ISSN: 0360-5280 (pages 111, 153).

[34] Alain Colmerauer and Philippe Roussel. “The Birth of Prolog”. In: *History of Programming languages—II.* Ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1996, pp. 331–367. ISBN: 0-201-89502-1. DOI: 10.1145/234286.1057820 (pages 15, 70).

[35] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. Online publication. Oct. 2007. URL: http://tata.gforge.inria.fr/ (pages 92, 120, 121, 123–126, 137, 138, 151).

[36] M.P. Consens and A.O. Mendelzon. "Low-Complexity Aggregation in GraphLog and Datalog". In: *Theoretical Computer Science* 116.1 (1993), pp. 95–116 (pages 3, 6, 116).

[37] Antony Courtney and Conal Elliott. "Genuinely Functional User Interfaces". In: *2001 Haskell Workshop*. Sept. 2001. URL: http://www.haskell.org/yale/papers/haskellworkshop01/genuinely-functional-guis.pdf (pages 5, 33).

[38] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252. URL: http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml (page 141).

[39] P. Cousot and R. Cousot. "Systematic design of program analysis frameworks". In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas: ACM Press, New York, NY, 1979, pp. 269–282. URL: http://www.di.ens.fr/~cousot/COUSOTpapers/POPL79.shtml (page 141).

[40] Mark Z. Danielewski. *House Of Leaves*. (The discussion of the Minotaur is especially relevant to this thesis.) Pantheon Books, 2000. ISBN: 0-375-70376-4.

[41] Max Dauchet, Anne-Cécile Caron, and Jean-Luc Coquidé. "Automata for Reduction Properties Solving". In: 20.2 (1995), pp. 215–233. ISSN: 0747-7171. DOI: 10.1006/jsco.1995.1048 (page 125).

[42] Bart Demoen and Phuong-Lan Nguyen. "So Many WAM Variations, So Little Time". In: *Computational Logic — CL 2000: First International Conference London, UK, July 24–28, 2000 Proceedings*. Ed. by John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1240–1254. ISBN: 978-3-540-44957-7. DOI: 10.1007/3-540-44957-4_83 (page 120).

[43] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 39.1 (1977), pp. 1–38. ISSN: 00359246. URL: http://www.jstor.org/stable/2984875 (page 66).

[44] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271. URL: http://jmvidal.cse.sc.edu/library/dijkstra59a.pdf (page 59).

[45] Alexander Dreyer. "Interval analysis of analog circuits with component tolerances". PhD thesis. 2005. URL: http://www.itwm.fraunhofer.de/fileadmin/TEST-Media/TU/PDF/drarbeit.pdf (page 19).

[46] Joshua Dunfield. "A Unified System of Type Refinements". CMU-CS-07-129. PhD thesis. Carnegie Mellon University, Aug. 2007. URL: http://www.cs.cmu.edu/~joshuad/papers/thesis/ (page 150).

[47] ECMA International. *Standard ECMA-262*. 2009. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm (page 169).

[48] Jason Eisner and John Blatz. "Program Transformations for Optimization of Parsing Algorithms and Other Weighted Logic Programs". In: *Proc. of FG 2006: The 11th Conference on Formal Grammar*. Ed. by Shuly Wintner. CSLI Publications, 2007, pp. 45–85. URL: http://www.cs.jhu.edu/~jason/research.html#fg06 (pages 24, 69).

[49] Jason Eisner and Nathaniel W. Filardo. "Dyna: Extending Datalog for modern AI". In: *Datalog Reloaded*. Ed. by Tim Furche, Georg Gottlob, Oege de Moor, and Andrew Sellers. Vol. 6702. LNCS. Springer, 2011. DOI: 10.1007/978-3-642-24206-9_11 (pages 67, 197).

[50] Jason Eisner and Nathaniel W. Filardo. *Dyna: Extending Datalog for modern AI (full version)*. Tech. rep. Available at dyna.org/Publications. A condensed version appeared as [49]. Johns Hopkins University, 2011 (pages xiii, 1, 4, 5, 169, 172, 173, 182).

[51] Jason Eisner, Eric Goldlust, and Noah A. Smith. "Compiling Comp. Ling.: Weighted Dynamic Programming and the Dyna Language". In: *Proc. of HLT-EMNLP*. Vancouver: Association for Computational Linguistics, Oct. 2005, pp. 281–290. URL: http://cs.jhu.edu/~jason/papers/#emnlp05-dyna (pages 1, 6, 24, 28, 58–60, 75, 113, 116, 182).

[52] Jason Eisner, Eric Goldlust, and Noah A. Smith. "Dyna: A Declarative Language for Implementing Dynamic Programs". In: *Proc. of ACL, Companion Volume*. Barcelona, July 2004, pp. 218–221. URL: http://cs.jhu.edu/~jason/papers/#acl04-dyna (pages 1, 24).

[53] 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 (page 28).

[54] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/ (pages 5, 33).

[55] Leonhard Euler. "De summis serierum reciprocarum. (On the sums of series of reciprocals)". In: *Commentarii academiae scientiarum Petropolitanae*. Vol. 7. 1740, pp. 123–134. URL: http://eulerarchive.maa.org/pages/E041.html (page 181).

[56] Leonhard Euler. "Remarques sur un beau rapport entre les series des puissances tant directes que reciproques. (Remarks on a beautiful relation between direct as well as reciprocal power series)". In: *Memoires de l'academie des sciences de Berlin*. Vol. 17. 1768, pp. 83–106. URL: http://eulerarchive.maa.org/pages/E352.html (page 146).

[57] Nathaniel Wesley Filardo et al. *An Automata Zoo.* URL: https://github.com/nwf/autzoo (page xiii).

[58] Nathaniel Wesley Filardo. *The Two-Counter Machine Strikes Again.* 2016. URL: http://www.ietfng.org/nwf/publications-and-talks/2016-2cm.html (pages 121, 126).

[59] Nathaniel Wesley Filardo and Tim Vieira et al. *Dyna 2 Compiler and REPL, 2013 edition.* URL: https://github.com/nwf/dyna (pages 55, 83).

[60] 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). Budapest, Sept. 2012, pp. 425–438. ISBN: 978-3-939897-43-9. URL: http://cs.jhu.edu/~jason/papers/#filardo-eisner-2012-iclp (pages xiii, 33, 109).

[61] Nathaniel Wesley Filardo and Jason Eisner. *Default Reasoning in Weighted Logic Programs.* 2017. URL: http://www.ietfng.org/nwf/publications-and-talks/2017-default.html (page xiii).

[62] Nathaniel Wesley Filardo and Jason Eisner. "Rigid Tree Automata With Isolation". In: *Proceedings of the Fourth International Workshop on Trends in Tree Automata and Tree Transducers (TTATT).* Seoul, Aug. 2016. URL: https://www.ietfng.org/nwf/publications-and-talks/2016-ttatt.html (page 127).

[63] Nathaniel Wesley Filardo and Jason Eisner. *Set-at-a-Time Solving in Weighted Logic Programs.* 2017. URL: http://www.ietfng.org/nwf/publications-and-talks/2017-set.html (pages xiii, 103).

[64] Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. "Satisfiability of a Spatial Logic with Tree Variables". In: *Computer Science Logic*. Ed. by Jacques Duparc and Thomas A. Henzinger. Vol. 4646. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 130–145. ISBN: 978-3-540-74914-1. URL: http://link.springer.com/10.1007/978-3-540-74915-8_13 (page 126).

[65] Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. "Tree Automata with Global Constraints". In: *Developments in Language Theory*. Ed. by Masami Ito and Masafumi Toyama. Lecture Notes in Computer Science 5257. Springer Berlin Heidelberg, 2008, pp. 314–326. ISBN: 978-3-540-85779-2. URL: http://link.springer.com/chapter/10.1007/978-3-540-85780-8_25 (pages 126, 127).

[66] Melvin Fitting. "A kripke-kleene semantics for logic programs*". In: *The Journal of Logic Programming* 2.4 (1985), pp. 295–312. ISSN: 0743-1066. DOI: 10.1016/S0743-1066(85)80005-4 (page 17).

[67] Melvin Fitting. "Fixpoint semantics for logic programming a survey". In: *Theoretical Computer Science* 278.1 (2002). Mathematical Foundations of Programming Semantics 1996, pp. 25–51. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(00)00330-3 (page 16).

[68] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. “The Essence of Compiling with Continuations”. In: *Proc. of PLDI*. PLDI ’93. Albuquerque, New Mexico, United States: ACM, 1993, pp. 237–247. ISBN: 0-89791-598-4. DOI: 10.1145/155090.155113 (page 71).

[69] Tim Freeman and Frank Pfenning. “Refinement Types for ML”. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI ’91. Toronto, Ontario, Canada: ACM, 1991, pp. 268–277. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113468. URL: https://www.cs.cmu.edu/~fp/papers/pldi91.pdf (page 150).

[70] Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. “Logic programs as types for logic programs”. In: *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*. Amsterdam, The Netherlands: IEEE Computer Society Press, July 1991, pp. 300–309. URL: http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/Papers/lics91-ieee.pdf (pages 141, 151).

[71] Murdoch J. Gabbay and Aad Mathijssen. “Capture-avoiding Substitution As a Nominal Algebra”. In: *Proceedings of the Third International Conference on Theoretical Aspects of Computing*. ICTAC’06. Tunis, Tunisia: Springer-Verlag, 2006, pp. 198–212. ISBN: 978-3-540-48815-6. DOI: 10.1007/11921240_14 (page 119).

[72] Hervé Gallaire and John “Jack” Minker, eds. *Logic and Data Bases: Proceedings of the Symposium on Logic and Data Bases Held at the Centre D’Études Et de Recherches de L’École Nationale Supérieure de L’Aéronautique Et de L’Espace de Toulouse*. Plenum Publishing Company, Limited, 1978. ISBN: 030640060X (page 23).

[73] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. “Directed Hypergraphs and Applications”. In: *Discrete Appl. Math.* 42.2-3 (Apr. 1993), pp. 177–201. ISSN: 0166-218X. DOI: 10.1016/0166-218X(93)90045-P (page 24).

[74] Michael Gelfond. “On Stratified Autoepistemic Theories”. In: *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*. AAAI’87. Seattle, Washington: AAAI Press, 1987, pp. 207–211. ISBN: 0-934613-42-7. URL: https://ocs.aaai.org/Papers/AAAI/1987/AAAI87-037.pdf (page 17).

[75] Michael Gelfond and Vladimir Lifschitz. “The Stable Model Semantics for Logic Programming”. In: *Logic Programming, Proc. of the 5th International Conference and Symposium*. 1988, pp. 1070–1080 (page 17).

[76] J.Y. Girard. “Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur”. PhD thesis. Université Paris, 1972 (page 130).

[77] *GNU Prolog*. URL: http://www.gprolog.org/ (page 3).

[78] Guillem Godoy and Omer Giménez. “The HOM Problem is Decidable”. In: *J. ACM* 60.4 (Sept. 2013), 23:1–23:44. ISSN: 0004-5411. DOI: 10.1145/2501600. URL: http://www.lsi.upc.es/~ggodoy/papers/homproblem2.pdf (page 125).

[79] Joshua Goodman. “Semiring parsing”. In: *Computational Linguistics* 25.4 (1999), pp. 573–605. ISSN: 0891-2017 (page 24).

[80] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in JAVA*. Wiley, 1998 (page 28).

[81] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, 1994. ISBN: 0-201-55802-5 (page 195).

[82] Sergio Greco. "Dynamic Programming in Datalog with Aggregates". In: *IEEE Transactions on Knowledge and Data Engineering* 11.2 (1999), pp. 265–283. ISSN: 1041-4347. DOI: 10.1109/69.761663 (pages 3, 6, 67, 116).

[83] Cordell Green. "Application of Theorem Proving to Problem Solving". In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC: Morgan Kaufmann Publishers Inc., 1969, pp. 219–239. URL: http://dl.acm.org/citation.cfm?id=1624562.1624585 (pages 3, 15).

[84] Todd J. Green, Gregory Karvounarakis, and Val Tannen. "Provenance Semirings". In: *Proc. of PODS*. 2007, pp. 31–40. URL: http://db.cis.upenn.edu/DL/07/pods07.pdf (page 24).

[85] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally". In: *SIGMOD Conference*. Ed. by Peter Buneman and Sushil Jajodia. Washington, DC: ACM Press, May 1993, pp. 157–166. DOI: 10.1145/170035.170066 (pages 6, 68).

[86] Spyros Hadjichristodoulou. "Mode-Sensitive Type Analysis for Prolog Programs". PhD thesis. Stony Brook University, May 2014 (page 151).

[87] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. "Incremental Computation with Names". In: *CoRR* (2015). URL: http://arxiv.org/abs/1503.07792 (pages 68, 177).

[88] Robert Harper. *Practical Foundations for Programming Languages*. New York, NY, USA: Cambridge University Press, 2012. ISBN: 9781107029576 (page 117).

[89] Nevin Heintze and Joxan Jaffar. "A Finite Presentation Theorem for Approximating Logic Programs". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: ACM, 1990, pp. 197–209. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96729 (pages 141, 151).

[90] Nevin Heintze and Joxan Jaffar. "Semantic Types for Logic Programs". In: Frank Pfenning. *Types in Logic Programming*. Cambridge, MA, USA: MIT Press, 1992. Chap. 4. ISBN: 0-262-16131-1. URL: http://www.cs.cmu.edu/afs/cs/user/nch/ftp/lp-types.ps.Z (page 142).

[91] Fergus. Henderson, Zoltan. Somogyi, and Thomas Conway. *Determinism analysis in the mercury compiler*. English. Book. 1995, p. 19 (page 164).

[92] Jacques Herbrand. *Recherches sur la théorie de la démonstration*. fre. 1930. URL: http://eudml.org/doc/192791 (pages 6, 13).

[93] David Hilbert and Paul Bernays. *Grundlagen der Mathematik. Die Grundlehren der mathematischen Wissenschaften*. Vol. 1. ISBN: 978-3-540-04134-4 (page 129).

[94] Patricia M. Hill and Rodney W. Topor. "A Semantics for Typed Logic Programs". In: Frank Pfenning. *Types in Logic Programming.* Cambridge, MA, USA: MIT Press, 1992. Chap. 1. ISBN: 0-262-16131-1 (pages 150, 151).

[95] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. "Unifying Structured Recursion Schemes". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.* ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 209–220. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500578 (page 123).

[96] Alfred Horn. "On sentences which are true of direct unions of algebras". In: *Journal of Symbolic Logic* 16.1 (1951), pp. 14–21. DOI: 10.2307/2268661 (page 15).

[97] Thomas Hungerford. *Algebra.* Eighth. Springer, 2003. ISBN: 978-1-4612-6103-2. DOI: 10.1007/978-1-4612-6101-8 (page 9).

[98] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935 (page 18).

[99] *ISO/IEC 13211-1:1995: Information technology – Programming languages – Prolog – Part 1: General core.* Standard. International Organization for Standardization, 1995. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:13211:-1:ed-1:v1:en (pages 2, 3).

[100] Florent Jacquemard, Francis Klay, and Camille Vacher. "Rigid tree automata and applications". In: *Information and Computation* 209.3 (Mar. 2011), pp. 486–512. ISSN: 08905401. DOI: 10.1016/j.ic.2010.11.015 (pages 127, 136).

[101] Florent Jacquemard, Michael Rusinowitch, and Laurent Vigneron. "Tree automata with equality constraints modulo equational theories". In: *Journal of Logic and Algebraic Programming* 75.2 (2008), pp. 182–208. URL: http://www.sciencedirect.com/science/article/pii/S1567832607000884 (page 125).

[102] Manfred Jaeger. "A Logic for Default Reasoning About Probabilities". In: *Proceedings of the Tenth International Conference on Uncertainty in Artificial Intelligence.* UAI'94. Seattle, WA: Morgan Kaufmann Publishers Inc., 1994, pp. 352–359. ISBN: 1-55860-332-8. URL: https://arxiv.org/abs/1302.6822 (page 116).

[103] David Jeffery. "Expressive type systems for logic programming languages". PhD thesis. The University of Melbourne, Feb. 2002. URL: http://www.cs.mu.oz.au/research/mercury/information/papers.html#dgj-thesis-final (page 142).

[104] Martin Kay. "Algorithm Schemata and Data Structures in Syntactic Processing". In: *Readings in Natural Language Processing.* Ed. by B. J. Grosz, K. Sparck Jones, and B. L. Webber. First published in 1980 as Xerox PARC Technical Report CSL-80-12 and in the Proceedings of the Nobel Symposium on Text Processing, Gothenburg. Los Altos, CA: Kaufmann, 1986, pp. 35–70 (page 28).

[105] D. B. Kemp and P. J. Stuckey. "Semantics of Logic Programs With Aggregates". In: *Proc. of the International Logic Programming Symposium* (1991), pp. 338–401 (page 17).

[106] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945 (page 21).

[107] Dan Klein and Christopher D. Manning. “A* Parsing: Fast Exact Viterbi Parse Selection”. In: *Proc. of HLT-NAACL*. 2003. URL: http://www.stanford.edu/~manning/papers/pcfg-astar.pdf (page 28).

[108] Christoph Koch, Daniel Lupei, and Val Tannen. “Incremental View Maintenance For Collection Programming”. In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS ’16. San Francisco, California, USA: ACM, 2016, pp. 75–90. ISBN: 978-1-4503-4191-2. DOI: 10.1145/2902251.2902286. URL: http://doi.acm.org/10.1145/2902251.2902286 (page 35).

[109] Robert Kowalski. “Algorithm = Logic + Control”. In: *Commun. ACM* 22.7 (July 1979), pp. 424–436. ISSN: 0001-0782. DOI: 10.1145/359131.359136 (page 2).

[110] Robert Kowalski. *Predicate Logic as Programming Language*. Memo 70. Reprinted in Computers for Artificial Intelligence Applications, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, pp. 68–73. Department of Artificial Intelligence, Edinburgh University, 1974. URL: http://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf (pages 6, 15, 80).

[111] Robert A. Kowalski. “The Early Years of Logic Programming”. In: *Commun. ACM* 31.1 (Jan. 1988), pp. 38–43. ISSN: 0001-0782. DOI: 10.1145/35043.35046 (page 15).

[112] Thomas Labish. “Developing a Combined Forward/Backward-chaining System for Logic Programs in a Hybrid Expertsystem Shell.” In German. MA thesis. Universität Kaiserlautern, June 1993 (page 68).

[113] J. C. R. Licklider. “Man-Computer Symbiosis”. In: *IRE Transactions on Human Factors in Electronics* HFE-1 (Mar. 1960), pp. 4–11. URL: http://groups.csail.mit.edu/medg/people/psz/Licklider.html (page 139).

[114] Vladimir Lifschitz. “Intelligent Instantiation and Supersafe Rules”. In: *Technical Communications of the 32nd ICLP*. 2016. URL: http://www.cs.utexas.edu/users/ai-lab/?supersafety (page 83).

[115] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. “Recursive Computation of Regions and Connectivity in Networks”. In: *IEEE 25th International Conference on Data Engineering* (2009). DOI: 10.1109/ICDE.2009.36. URL: http://repository.upenn.edu/cis_papers/417/ (page 66).

[116] J.W. Lloyd and R.W. Topor. “A basis for deductive database systems”. In: *The Journal of Logic Programming* 2.2 (1985), pp. 93–109. ISSN: 0743-1066. DOI: 10.1016/0743-1066(85)90013-5 (page 1).

[117] LogicBlox. *Datalog for Enterprise Applications: from Industrial Applications to Research*. Presented by Molham Aref at Datalog 2.0 Workshop. Mar. 2010. URL: http://www.logicblox.com/research/presentations/arefdatalog20.pdf (pages 2, 3).

[118] William Lovas. “Refinement Types for Logical Frameworks”. PhD thesis. Carnegie Mellon University, Sept. 2010. URL: http://www.cs.cmu.edu/~wlovas/papers/wjl-thesis-final.pdf (page 150).

[119] Jan Małuszyński and Andrzej Szałas. “Living with Inconsistency and Taming Non-monotonicity”. In: *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Ed. by Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 384–398. ISBN: 978-3-642-24206-9. DOI: 10.1007/978-3-642-24206-9_22 (page 18).

[120] Matthieu Martel. “Program Transformation for Numerical Precision”. In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’09. Savannah, GA, USA: ACM, 2009, pp. 101–110. ISBN: 978-1-60558-327-3. DOI: 10.1145/1480945.1480960 (page 69).

[121] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 4.2 (Apr. 1982), pp. 258–282. ISSN: 0164-0925. DOI: 10.1145/357162.357169 (page 118).

[122] Erik Meijer, Maarten Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: *5th ACM Conference on Functional Programming Languages and Computer Architecture*. Vol. 523. URL: http://research.microsoft.com/~emeijer/Papers/fpca91.pdf (page 21).

[123] C. S. Mellish. *The Automatic Generation of Mode Declarations for Prolog Programs*. Tech. rep. DAI Research Paper 163. Department of Artificial Intelligence, University of Edinburgh, 1981 (page 153).

[124] *The Mercury Project*. URL: http://www.cs.mu.oz.au/research/mercury/index.html (pages 2, 3, 153).

[125] Jack Minker and Dietmar Seipel. “Disjunctive Logic Programming: A Survey and Assessment”. In: *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part I*. Ed. by Antonis C. Kakas and Fariba Sadri. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 472–511. ISBN: 978-3-540-45628-5. DOI: 10.1007/3-540-45628-7_18 (page 79).

[126] Prateek Mishra. “Towards a Theory of Types in Prolog”. In: *Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA, February 6-9, 1984*. IEEE-CS, 1984. ISBN: 0-8186-0522-7 (page 142).

[127] Abhijeet Mohapatra and Michael Genesereth. *Aggregation in Datalog under set semantics*. Tech. rep. 2012. URL: http://logic.stanford.edu/reports/LG-2012-01.pdf (page 116).

[128] Jocelyne Mongy-Steen. “Transformations de noyaux reconnaissables d’arbres: forêts RATEG”. PhD thesis. Université Lille, 1981. URL: http://ori.univ-lille1.fr/notice/view/univ-lille1-ori-81256 (page 125).

[129] R. E. Moore. "Interval Arithmetic and Automatic Error Analysis in Digital Computing". Also published as Applied Mathematics and Statistics Laboratories Technical Report No. 25. PhD thesis. Stanford, CA, USA: Department of Mathematics, Stanford University, Nov. 1962. URL: http://interval.louisiana.edu/Moores_early_papers/disert.pdf (page 19).

[130] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *DAC*. Las Vegas, NV, USA: ACM, June 2001, pp. 530–535. ISBN: 1-58113-297-2 (page 51).

[131] 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 (page 4).

[132] Alan Mycroft and Richard A. O'Keefe. "A polymorphic type system for prolog". In: *Artificial Intelligence* 23.3 (1984), pp. 295–307. ISSN: 0004-3702. DOI: 10.1016/0004-3702(84)90017-1 (page 142).

[133] Arun Nampally and C. R. Ramakrishnan. "Inference in Probabilistic Logic Programs using Lifted Explanations". In: (2016). URL: https://arxiv.org/abs/1608.05763 (page 116).

[134] Nicholas Nethercote. "The Analysis Framework of HAL". Revised April 2002. MA thesis. University of Melbourne, Sept. 2001. URL: http://njn.valgrind.org/pubs/masters2001.ps (page 164).

[135] 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*. Perugia/Italy, Sept. 2011, pp. 645–659. URL: http://www.cse.unsw.edu.au/~mmaher/pubs/cp/kangaroo.pdf (pages 42, 68).

[136] Oystein Ore. "Galois connexions". In: *Transactions of the American Mathematical Society* 55.3 (1944), pp. 493–513. URL: http://www.ams.org/journals/tran/1944-055-00/S0002-9947-1944-0010555-7/S0002-9947-1944-0010555-7.pdf (page 141).

[137] David Overton. "Precise and Expressive Mode Systems for Typed Logic Programming Languages". PhD thesis. University of Melbourne, 2003. URL: http://www.mercury.cs.mu.oz.au/information/papers.html#dmo-thesis (pages 81, 120, 133, 134, 153–155, 164, 183).

[138] Zachary Palmer. "Building a Typed Scripting Language". PhD thesis. Johns Hopkins University, 2015. URL: http://pl.cs.jhu.edu/projects/big-bang/dissertations/building-a-typed-scripting-language.pdf (page 143).

[139] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. "Automatically Improving Accuracy for Floating Point Expressions". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 1–11. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737959. URL: http://herbie.uwplse.org/pldi15-paper.pdf (page 69).

[140] David Lorge Parnas. "Software Aspects of Strategic Defense Systems". In: *Commun. ACM* 28.12 (Dec. 1985), pp. 1326–1335. ISSN: 0001-0782. DOI: 10.1145/214956.214961 (page 20).

[141] Adam Pauls and Dan Klein. "*k*-Best A* Parsing". In: *Proc. of ACL-IJCNLP*. Suntec, Singapore: ACL, Aug. 2009, pp. 958–966. URL: http://www.aclweb.org/anthology/P/P09/P09-1108 (page 48).

[142] Simon Peyton-Jones, ed. *Haskell 98 language and libraries : the revised report*. Dec. 2002. URL: https://www.haskell.org/onlinereport/ (page 14).

[143] F. Pfenning and C. Elliott. "Higher-order Abstract Syntax". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, pp. 199–208. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54010. URL: http://www.cs.cmu.edu/afs/cs/Web/People/fp/papers/pldi88.pdf (page 168).

[144] Frank Pfenning. *Types in Logic Programming*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-16131-1.

[145] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing Series. Cambridge, Massachusetts: MIT Press, 1991 (page 21).

[146] Pink Floyd. *Learning To Fly*. Track 2 of the album *Learning to Fly*; published by EMI (UK) or Columbia (US). 1987 (page 35).

[147] Andrew M. Pitts. "Polymorphism is Set Theoretic, Constructively". In: *Category Theory and Computer Science*. London, UK, UK: Springer-Verlag, 1987, pp. 12–39. ISBN: 3-540-18508-9. URL: https://www.irif.fr/~mellies/mpri/mpri-ens/articles/pitts-polymorphism-is-set-theoretic-constructively.pdf (page 129).

[148] António Porto. "A structured alternative to Prolog with simple compositional semantics". In: *CoRR* abs/1107.5408 (2011). DOI: 10.1017/S1471068411000202 (page 3).

[149] M. O. Rabin and D. Scott. "Finite Automata and Their Decision Problems". In: *IBM Journal of Research and Development* 3.2 (Apr. 1959), pp. 114–125. DOI: 10.1147/rd.32.0114 (pages 122, 123).

[150] L. Rabiner and B. Juang. "An introduction to hidden Markov models". In: *ASSP Magazine, IEEE* 3.1 (Jan. 1986). ISSN: 0740-7467. DOI: 10.1109/MASSP.1986.1165342 (page 26).

[151] L. De Raedt, A. Kimmig, and H. Toivonen. "ProbLog: A Probabilistic Prolog and its Application in Link Discovery". In: *Proc. of IJCAI*. 2007, pp. 2462–2467. URL: http://www.cs.kuleuven.be/~dtai/publications/files/42447.pdf (page 116).

[152] Raghu Ramakrishnan. “Magic templates: a spellbinding approach to logic programs”. In: *Journal of Logic Programming* 11.3-4 (1991), pp. 189–216. ISSN: 0743-1066. DOI: 10.1016/0743-1066(91)90026-L (pages 34, 40, 68, 133).

[153] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. “The CORAL Deductive System”. In: *The VLDB Journal* 3.2 (1994). Special Issue on Prototypes of Deductive Database Systems., pp. 161–210. URL: ftp://ftp.cs.wisc.edu/coral/doc/coral.ps (pages 3, 6).

[154] U. S. Reddy. “Notions of Polymoprhism for Predicate Logic Programming”. In: *Proc. 5th Int’l Conf. and Symp. on Logic Programming*. Ed. by R. A. Kowalski and K. A. Bowen. Seattle, Washington, USA: MIT Press, 1988. ISBN: 0262610558 (page 142).

[155] R. Reiter. “A logic for default reasoning”. In: *Artificial Intelligence* 13.1 (1980), pp. 81–132. ISSN: 0004-3702. DOI: 10.1016/0004-3702(80)90014-4 (page 116).

[156] R. Reiter. “Readings in Nonmonotonic Reasoning”. In: ed. by Matthew L. Ginsberg. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987. Chap. On Closed World Data Bases, pp. 300–310. ISBN: 0-934613-45-1. URL: http://dl.acm.org/citation.cfm?id=42641.42663 (page 22).

[157] John C. Reynolds. “Polymorphism is not set-theoretic”. In: *Semantics of Data Types: International Symposium Sophia-Antipolis, France, June 27 – 29, 1984 Proceedings*. Ed. by Gilles Kahn, David B. MacQueen, and Gordon Plotkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 145–156. ISBN: 978-3-540-38891-3. DOI: 10.1007/3-540-13346-1_7 (page 129).

[158] John C. Reynolds. “Towards a Theory of Type Structure”. In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK, UK: Springer-Verlag, 1974, pp. 408–423. ISBN: 3-540-06859-7. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=2289&context=compsci (page 130).

[159] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *Journal of the ACM* 12.1 (Jan. 1965), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253 (page 118).

[160] Diptikalyan Saha. “Incremental Evaluation of Tabled Logic Programs”. PhD thesis. Stony Brook University, Dec. 2006 (page 68).

[161] Tom Schrijvers and Bart Demoen. “Combining an Improvement to PARMA Trailing with Trailing Analysis”. In: *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP ’02. Pittsburgh, PA, USA: ACM, 2002, pp. 88–98. ISBN: 1-58113-528-9. DOI: 10.1145/571157.571167 (page 120).

[162] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. “Towards Typed Prolog”. In: *Logic Programming: 24th International Conference, ICLP 2008 Udine, Italy, December 9-13 2008 Proceedings*. Ed. by Maria Garcia de la Banda and Enrico Pontelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 693–697. DOI: 10.1007/978-3-540-89982-2_59 (page 151).

[163] Robjert J. Simmons, Bernardo Toninho, and Frank Pfenning. "Distributed deductive databases, declaratively". In: *The ACM SIGPLAN 2011 X10 Workshop*. ACM, June 2011. URL: http://www.cs.cmu.edu/~rjsimmon/papers/simmons11distributed.pdf (page 68).

[164] Imre Simon. "Recognizable Sets with Multiplicities in the Tropical Semiring". In: *Proceedings of the Mathematical Foundations of Computer Science 1988*. MFCS '88. London, UK, UK: Springer-Verlag, 1988, pp. 107–120. ISBN: 3-540-50110-X. URL: http://dl.acm.org/citation.cfm?id=645718.665792 (page 11).

[165] G. Slutzki. "Alternating Tree Automata". In: *Theoretical Computer Science* 41.2-3 (Dec. 1985), pp. 305–318. ISSN: 0304-3975. DOI: 10.1016/0304-3975(85)90077-5 (page 151).

[166] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. "The execution algorithm of mercury, an efficient purely declarative logic programming language". In: *The Journal of Logic Programming* 29.1 (1996). High-Performance Implementations of Logic Programming Systems, pp. 17–64. ISSN: 0743-1066. DOI: https://doi.org/10.1016/S0743-1066(96)00068-4 (page 185).

[167] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. "Algorithms for discrete function manipulation". In: *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. Nov. 1990, pp. 92–95. DOI: 10.1109/ICCAD.1990.129849 (page 116).

[168] Guy L. Steele. *Common LISP: The Language (2Nd Ed.)* Newton, MA, USA: Digital Press, 1990. ISBN: 1-55558-041-6. URL: https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html (page 4).

[169] 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. Helsinki, Finland: Springer-Verlag, 2010, pp. 300–312. ISBN: 978-3-642-15674-8. URL: http://www3.cs.stonybrook.edu/~tswift/webpapers/JELIA-10.pd (page 113).

[170] Terrance Swift and David Scott Warren. "XSB: Extending Prolog with Tabled Logic Programming". In: *CoRR* abs/1012.5123 (2010). Under consideration for publication in Theory and Practice of Logic Programming. URL: http://arxiv.org/abs/1012.5123 (pages 4, 68).

[171] J. M. Talbot, S. Tison, and P. Devienne. "Set-based analysis for logic programming and tree automata". In: *Static Analysis: 4th International Symposium, SAS '97 Paris, France, September 8–10, 1997 Proceedings*. Ed. by Pascal Van Hentenryck. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 127–140. ISBN: 978-3-540-69576-9. DOI: 10.1007/BFb0032738 (pages 141, 151).

[172] Masaru Tomita. "An Efficient Context-free Parsing Algorithm for Natural Languages". In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'85. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985, pp. 756–764. ISBN: 978-0-934-61302-6 (page 21).

[173] Rodney W. Topor. “Safe Database Queries With Arithmetic Relations”. In: *Proc. 14th Australian Computer Science Conference*. 1991. URL: http://www.sci.gu.edu.au/~rwt/papers/ACSC91.ps (page 67).

[174] G. S. Tseitin. “On the complexity of derivations in the propositional calculus”. In: *Studies in Mathematics and Mathematical Logic* Part II (1968), pp. 115–125 (page 120).

[175] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Vol. 1. Computer Science Press, 1988 (page 28).

[176] David Ungar and Randall B. Smith. “Self: The Power of Simplicity”. In: *Proc.of OOPSLA*. Orlando, Florida, United States: ACM, 1987, pp. 227–242. ISBN: 0-89791-247-0. DOI: 10.1145/38765.38828. URL: http://labs.oracle.com/self/papers/self-power.html (page 169).

[177] Guy Van den Broeck. “Lifted Inference and Learning in Statistical Relational Models”. PhD thesis. KU Leuven, Jan. 2013, pp. xx + 264. URL: http://web.cs.ucla.edu/~guyvdb/phd/guyvdb-phdthesis.pdf (page 116).

[178] M. H. Van Emden and R. A. Kowalski. “The Semantics of Predicate Logic as a Programming Language”. In: *JACM* 23.4 (1976), pp. 733–742 (pages 15, 16, 61).

[179] M.H. Van Emden. *First-order Predicate Logic as a High-level Program Language*. Tech. rep. MIP-R-106. University of Edinburgh. School of Artificial Intelligence, May 1974. URL: http://webhome.cs.uvic.ca/~vanemden/Publications/FOPLasHLPL.pdf (pages 15, 80).

[180] A. Van Gelder, K. A. Ross, and J. S. Schlipf. “The Well-Founded Semantics for General Logic Programs”. In: *Journal of the ACM* 38.3 (1991), pp. 620–650 (page 17).

[181] Alexander Vandenbroucke, Maciej Piróg, Benoit Desouter, and Tom Schrijvers. “Tabling with Sound Answer Subsumption”. In: *Theory and Practice of Logic Programming* 16.5-6 (Sept. 2016), pp. 933–949. DOI: 10.1017/S147106841600048X. URL: https://people.cs.kuleuven.be/~alexander.vandenbroucke/publications/answer-subsumption.pdf (pages 113–115).

[182] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. “Refinement Types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’14. Gothenburg, Sweden: ACM, 2014, pp. 269–282. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161. URL: http://goto.ucsd.edu/~nvazou/refinement_types_for_haskell.pdf (page 150).

[183] Tim Vieira, Matthew Francis-Landau, Nathaniel Wesley Filardo, Farzad Khorasani, and Jason Eisner. “Dyna: Toward a Self-Optimizing Declarative Language for Machine Learning Applications”. In: *Proceedings of the First ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL)*. Barcelona, June 2017. URL: https://timvieira.github.io/doc/2017-mapl-dyna.pdf (pages 28, 35, 185).

[184] Eelco Visser. "Program Transformation with Stratego/XT". In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 216–238. ISBN: 978-3-540-25935-0. DOI: 10.1007/978-3-540-25935-0_13 (page 69).

[185] Philip Wadler. "Theorems for Free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: ACM, 1989, pp. 347–359. ISBN: 0-89791-328-0. DOI: 10.1145/99370.99404 (page 129).

[186] Philip Wadler and Robert Bruce Findler. "Well-Typed Programs Can'T Be Blamed". In: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP '09. York, UK: Springer-Verlag, 2009, pp. 1–16. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_1. URL: https://www.eecs.northwestern.edu/~robby/pubs/papers/esop2009-wf.pdf (page 140).

[187] Clifford J. Walinsky. "Constructive Negation in Logic Programs". PhD thesis. 1987. DOI: 10.6083/M46H4FCP (page 17).

[188] M. Warmus. "Calculus of Approximations". In: *Bulletin De L'académie Polonaise des Sciences*. Vol. 4. 5. 1956, pp. 253–259 (page 19).

[189] David H. D. Warren. *An Abstract Prolog Instruction Set*. Tech. rep. SRI International Artificial Intelligence Center, Oct. 1983. URL: http://www.ai.sri.com/pubs/files/641.pdf (page 120).

[190] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. "Using Datalog with Binary Decision Diagrams for Program Analysis". In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS'05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 97–118. ISBN: 978-3-540-29735-2. DOI: 10.1007/11575467_8. URL: https://people.csail.mit.edu/mcarbin/papers/aplas05.pdf (pages 2, 3).

[191] Yong Xiao, Amr Sabry, and Zena M. Ariola. "From Syntactic Theories to Interpreters: Automating the Proof of Unique Decomposition". In: *Higher-Order and Symbolic Computation* 14.4 (Dec. 2001), pp. 387–409. ISSN: 1573-0557. DOI: 10.1023/A:1014408032446. URL: https://www.cs.indiana.edu/~sabry/papers/unique-decomp.ps (pages 129, 130, 144).

[192] *XSB*. URL: http://xsb.sourceforge.net/ (pages 2, 3, 113).

[193] *YAProlog*. URL: https://www.dcc.fc.up.pt/~vsc/Yap/ (pages 3, 113).

[194] 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 (page 28).

[195] Ulrich Zukowski and Burkhard Freitag. “The Deductive Database System *LOLA*”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Jürgen Dix, Ulrich Furbach, and Anil Nerode. LNAI 1265. Berlin: Springer, 1997, pp. 375–386. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.7258 (pages 3, 6).

[196] G. Zweig and M. Padmanabhan. “Exact Alpha-Beta Computation in Logarithmic Space with Application to MAP Word Graph Construction”. In: *Proceedings of ICSLP*. 2000. URL: http://research.microsoft.com/pubs/77963/icslp00_logspace.pdf (page 26).

Vita

Born in 1984, in Urbana, Illinois, Nathaniel Wesley Filardo developed an early interest in STEM. His paternal grandmother must have foreseen the future, ensuring that he had his very own Apple IIGS system only slightly after he had mastered object permanence. In early education, he wandered throughout the range of STEM and was saved from being an overly bookish bore by The Seven Hills Upper School art faculty and its theater program. Spending five years attending Carnegie Mellon University, he obtained a BSH in Physics and a BS in Computer Science, while working at the Robotics Institute, at RedZone Robotics, and as a TA for the Operating Systems class under Dr. David Eckhardt, who has been kind enough to invite the author back to give a guest lecture every semester since. Late in his time at CMU, he became enamored of statically-typed programming languages and began to ponder graduate work in the field. An internship at Sun Microsystems, while delightful, failed to dissuade him from his academically-minded plans, and so he found his way to the Computer Science department here at JHU, generously funded for many years by the JHU Human Language Technology Center Of Excellence. He has become a fixture of the local chapter of the Association for Computing Machinery and has been an instructor and/or TA for numerous semesters, on topics ranging from low-level systems programming to category theory. His research, culminating in this thesis, has widely broadened his mathematical foundations and appreciation of the complexities of language design.

His spartan autobiographical webpage is http://www.ietfng.org/nwf/nwf.html and includes a full CV and enumeration of publications.