Technical report of January 15, 2011 (with minor edits as of March 26, 2011). A condensed version appears as a chapter in Tim Furche, Georg Gottlob, Giovanni Grasso, Oege de Moor, and Andrew Sellers (eds.), *Datalog 2.0*. Lecture Notes in Computer Science. Springer (2011). The official version of that chapter will be available at www.springerlink.com.

Dyna: Extending Datalog For Modern AI (full version)*

Jason Eisner and Nathaniel W. Filardo

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

Abstract. Modern statistical AI systems are quite large and complex; this interferes with research, development, and education. We point out that most of the computation involves database-like queries and updates on complex views of the data. Specifically, recursive *queries* look up and aggregate relevant or potentially relevant values. If the results of these queries are memoized for reuse, the memos may need to be *updated* through change propagation. We propose a declarative language, which generalizes Datalog, to support this work in a generic way. Through examples, we show that a broad spectrum of AI algorithms can be concisely captured by writing down systems of equations in our notation. Many strategies could be used to actually solve those systems. Our examples motivate certain extensions to Datalog, which are connected to functional and object-oriented programming paradigms.

1 Why a New Data-Oriented Language for AI?

Modern AI systems are frustratingly big, making them time-consuming to engineer and difficult to modify. In this chapter, we describe our work toward a declarative language that was motivated originally by various use cases in AI.[1] Our goal is to make it easier to specify a wide range of new systems that are more or less in the mold of existing AI systems. Our declarative language should simplify inferential computation in the same way that the declarative language of regular expressions has simplified string pattern matching and transduction.

All areas of AI have become data-intensive, owing to the flood of data and the pervasiveness of statistical modeling and machine learning. A system's **extensional data** (inputs) include not only current sensory input but also background knowledge, large collections of training examples, and parameters trained from past experience. The **intensional data** (intermediate results and outputs) include combinatorially many possible analyses and conclusions derived from the inputs.

Each AI system usually builds and maintains its own custom data structures, so that it can efficiently query and update the current state of the system. Although many conceptual ideas are reused across AI, each implemented system tends to include its own specialized code for storage and inference, specialized to the data and computations used by that system. This turns a small mathematical abstraction into a large optimized implementation. It is difficult to change either the abstract computation or the storage and execution strategy because they are intertwined throughout the codebase. This also means that reusable general strategies have to be instantiated anew for each implemented system, and cannot even be easily described in an abstract way.

As an alternative, we are working to develop an appealing declarative language, Dyna, for concise specification of algorithms, with a compiler that turns such specifications into efficient code for storage and inference. Our goal is to produce a language that practitioners will actually use.

The heart of this long paper is the collection of suggestive Dyna code examples in §3.1. Readers are thus encouraged to browse at their leisure through Figures 1–12, which are relatively self-contained. Readers are also welcome to concentrate on the main flow of the paper, skipping over details that have been relegated for this reason to footnotes, figures, and appendices.

*

[1] Our own AI research is mainly on natural language processing, but as we show here, our observations and approach apply to other AI domains as well.

1.1 AI and Databases Today

Is a new language necessary? That is, why don't AI researchers already use database systems to manage their data [29]? After all, any procedural AI program is free to store its data in an external database. It could use Datalog or SQL to express queries against the current state of a database, perform some procedural computation on the results, and then store the results back to the database.

Unfortunately, there is rather little in most AI systems that looks like typical database queries:

– Queries in a standard language like Datalog or SQL are not expressive enough for any one query to capture the entire AI computation. These languages allow only restricted ways to query and combine data. The restrictions are intended to guarantee that each query terminates in polynomial time and has a single well-defined answer. Yet the overall AI algorithm may not be able to make those guarantees anyway—so the effect of the restrictions is only to partition the algorithm artificially into many smaller queries. This limits the opportunities for the database system itself to plan, rearrange, and parallelize computations.

– It may be inefficient to implement the algorithm in terms of database queries. Standard database implementations are good at large queries. They are intended to handle large, usually disk-backed, long-lived, read-mostly datasets that can be easily represented as tuples. Due to the high latency of disk access, they focus on supporting computations on large *sets* of records at a time. By contrast, AI systems typically work with lots of smaller, in-memory, ephemeral, write-heavy data sets often accessed at the level of individual records. For example, upon creating a promising hypothesis, the AI system might try to score it or extend it or compute its consequences, which involves looking up and storing *individual* records related to that specific hypothesis. Channeling these record-at-a-time queries and updates through a standard database would have considerable overhead. At the other extreme, one might try to reorganize the computation into coarser set-at-a-time queries where the database system will shine; but while such batch computations are a better fit to disk-based database systems, and also have the advantage that they can amortize work across many records, they may also do extra work that would be skipped by record-at-a-time strategies (by materializing larger relations that may include records that turn out not to be needed).

– Standard database languages do not support features for programming-in-the-large, such as modules, structured objects, or inheritance.

In this setting, switching from a data structure library to a relational database management system is likely to hurt performance without significantly reducing the implementation burden.

1.2 A Declarative Alternative

Our approach instead eliminates most of the procedural program, instead specifying its computations *declaratively*. We build on Datalog to propose a convenient, elegantly concise notation for specifying the systems of equations that relate intensional and extensional data. This is the focus of §2, beginning with a review of ordinary Datalog in §2.1.

A program in our Dyna language specifies what we call a **dynabase**, which is a kind of deductive database. Recall that a **deductive database** [38,167] contains not only extensional relations but also rules (usually Datalog rules or some other variant on Horn clauses) that define additional intensional relations, similar to views. Our term "dynabase" emphasizes that our deductive databases are *dynamic*: they can be declaratively extended into new dynabases that have modified extensional data, with consequent differences in the intensional data. Also, one dynabase can be defined in terms of others, supporting modularity (§2.7).

Because a Dyna program merely specifies a dynabase, it has no serial I/O or side effects. How, then, are dynabases used in a procedural environment? A running process, written in one's favorite procedural language, which does have I/O and side effects, can create a dynabase and update it serially by adding extensional data (§4.6). At any time, the process can *query* the dynabase to retrieve either the current extensional data, or intensional data that are defined in terms of the extensional data. As the process *updates* the extensional data, the intensional data that depend on it (possibly in other dynabases) are automatically maintained, as in a spreadsheet.[2] Carrying out the query and update operations requires the "heavy computational lifting" needed in AI for search, deduction, abduction, message passing, etc. However, the needed computations are specified only declaratively and at a high level of abstraction. They are carried out by the Dyna execution engine (eagerly or lazily) as needed to serve the process.

Essentially, a Dyna program is a set of equational schemata, which are similar to Datalog rules with (non-stratified) negation and aggregation. These schemata together with the extensional data define a possibly infinite system of equations, and the queriable "contents" of the dynabase come from a solution to this system. We give a gentle introduction in §2.3, and sketch a provisional semantics in the appendix (§A).

Dyna does extend Datalog in several ways, in part by relaxing restrictions (§2.4). It is Turing-complete, so that the full computation needed by an AI system can be triggered by a single query against a dynabase. Thus it is not necessary to specify which data to look up when, or whether or where to store the results. The resulting Turing-completeness gives greater freedom to both the Dyna programmer and the execution model, along with greater responsibility. Dyna also includes programming language features that improve its usability, such as typing, function evaluation, encapsulation, inheritance, and reflection.

Finally, Dyna's syntax for aggregation is very concise (even compared to other logic notations, let alone explicit loops) because its provable items have arbitrary values, not just truth values. Evaluating items in place makes it possible to write equations quite directly, with arithmetic and nested function evaluation.

We show and justify some of our extensions by way of various examples from AI in §3. As Figures 1–12 illustrate, Dyna programs are startlingly short relative to more traditional, procedural versions. They naturally support record-at-a-time execution strategies (§2.6), as well as automatic differentiation (§3.1) and change propagation (§4.3), which are practically very important. Dynabases are modular and can be easily integrated with one another into larger programs (§2.7). Finally, they do not specify any particular storage or execution strategies, leaving opportunities for both automatic and user-directed optimizations that preserve correctness (§5.3).

1.3 Storage and Execution Strategies

In this paper, we focus on the *expressivity* and *uses* of the Dyna language, as a user of Dyna would. From this point of view, the underlying computation order, indexing, and storage are distractions from a Dyna program's fundamentally declarative specification, and are relegated to an execution model—just as ordinary Datalog or SQL is a declarative language that leaves query optimization up to the database engine.

Actually computing and updating intensional data under a Dyna program may involve recursive internal queries and other work. However, this happens in some implementation-dependent order that can be tuned manually or automatically without affecting correctness.

The natural next questions are from an implementor's point of view. They concern this query and update planning, as well as physical design. How do we systematize the space of execution strategies and optimizations? Given a particular Dyna program and workload, can a generic Dyna engine discover the algorithms and data structures that an expert would choose by hand?

By showing in this paper that Dyna is capable of describing a wide range of computations, we mean to argue that finding efficient execution strategies for Dyna constitutes a substantial general program of research on *algorithms for AI and logic programming*.[3] After all, one would like a declarative solution of a given problem to exploit the relevant tricks used by the state-of-the-art procedural solutions. But then it is necessary to generalize these tricks into strategies that can be incorporated more generally into the Dyna runtime engine or encapsulated as general Dyna-to-Dyna program transformations [69,46]. These strategies may then be applied in new contexts. Building a wide range of tricks and strategies into the Dyna environment also raises the issue of how to manually specify and automatically tune strategies that work well on a particular workload.

Algorithms and pseudocode for a fragment of Dyna—the Dyna 1 prototype—appeared in [70,71]. We are now considering a much larger space of execution strategies, supported by type and mode systems (cf.

[2] A procedural process will therefore see changes when it queries a dynabase again. Alternatively, it may make a continuous query, whereby it is notified of updates to the query result (§4.6).

[3] More restricted declarative formalisms have developed substantial communities that work on efficient execution: propositional satisfiability, integer linear programming, queries and physical design in relational databases, etc.

[154]). Again, the present paper has a different focus; but §5 offers a high-level discussion of some of the many interesting issues.

2 Basic Features of the Language

Our goal in this section is to sketch just enough of Dyna that readers will be able to follow our AI examples in the next section. After quickly reviewing Datalog, we explain how Dyna augments Datalog by proving that terms have particular values, rather than merely proving that they are true; by relaxing certain restrictions; and by introducing useful notions of encapsulation and inheritance. (Formal semantics are outlined in the appendix (§A).)

2.1 Background: Datalog

Datalog [37] is a language—a concrete syntax—for defining named, flat relations. The (slightly incorrect) statement "Two people are siblings if they share a parent" can be precisely captured by a **rule** such as

```
sibling(A,B) :- parent(C,A), parent(C,B).
```
(1)

which may be read as "`A` is a sibling of `B` *if*, for some `C`, `C` is a parent of `A` *and* `C` is a parent of `B`." Formally, capitalized identifiers such as `A,B,C` denote universally quantified **variables**,[4] and the above rule is really a schema that defines infinitely many propositional implications such as

```
sibling(alice,bob) :- parent(charlie,alice),
                      parent(charlie,bob).
```
(2)

where `alice`, `bob`, and `charlie` are constants. (Thus, (2) is one of many possible implications that could be used to prove `sibling(alice,bob)`.) Rules can also mention constants directly, as in

```
parent(charlie,alice).
parent(charlie,bob).
```
(3)

Since the rules (3) also happen to have no conditions (no "`:-` ..." part), they are simply **facts** that directly specify part of the binary relation `parent`, which may be regarded as a two-column table in a relational database. The rule (1) defines another two-column table, `sibling`, by joining `parent` to itself on its first column and projecting that column out of the result.

Informally, we may regard `parent` (3) as extensional and `sibling` (1) as intensional, but Datalog as a language does not have to distinguish these cases. Datalog also does not specify whether the `sibling` relation should be materialized or whether its individual records should merely be computed as needed.

As this example suggests, it is simple in Datalog to construct new relations from old ones. Just as (1) describes a join, Datalog rules can easily describe other relational algebra operations such as project and select. They also permit recursive definitions. Datalog imposes the following syntactic restrictions to ensure that the defined relations are finite [37]:

- **Flatness**: Terms in a rule must include exactly one level of parentheses. This prevents recursive structure-building rules like

  ```
  is_integer(zero).
  is_integer(oneplus(X)) :- is_integer(X).
  ```
  (4)

  which would define an infinite number of facts such as
  `is_integer(oneplus(oneplus(oneplus(zero))))`.

[4] To guard against programming errors and to improve efficiency, one could declare that the functors `sibling` and `parent` must each take two arguments, and that these arguments must be persons. In Dyna, such type declarations are permitted but optional. (Some omitted declarations can be inferred from other declarations.)

– **Range restriction**: Any variables that occur in a rule's **head** (to the left of `:-`) must also appear in its **body** (to the right of `:-`). This prevents rules like

```
equal(X,X).
```
(5)

which would define an infinite number of facts such as `equal(31,31)`.

Pure Datalog also disallows built-in infinite relations, such as $<$ on the integers.[5] We will drop all these restrictions below.

2.2 Background: Datalog with Stratified Aggregation

Relations may range over numbers: for example, the variable `S` in `salary(alice,S)` has numeric type. Some Datalog dialects (e.g., [166,217]) support numeric **aggregation**, which combines numbers across multiple proofs of the same statement. As an example, if $w_{\text{parent}}(\texttt{charlie}, \texttt{alice}) = 0.75$ means that `charlie` is 75% likely to be a parent of `alice`, we might wish to define a soft measure of siblinghood by summing over possible parents:[6]

$$w_{\text{sibling}}(A, B) = \sum_{C} w_{\text{parent}}(C, A) \cdot w_{\text{parent}}(C, B). \tag{6}$$

The sum over `C` is a kind of aggregation. The syntax for writing this in Datalog varies by dialect; as an example, [47] would write the above fact and rule (6) as

```
parent(charlie,alice;0.75).
sibling(A,B;sum(Ma*Mb)) :- parent(C,A;Ma),
                           parent(C,B;Mb).
```
(7)

Datalog dialects with aggregation (or negation) often impose a further requirement to ensure that the relations are well-defined [9,146]:

– **Stratification**: A relation that is defined using aggregation (or negation) must not be defined in terms of itself. This prevents cyclic systems of equations that have no consistent solution (e.g., `a :- not a`) or multiple consistent solutions (e.g., `a :- not b` and `b :- not a`).

We omit details here, as we will drop this restriction below.

2.3 Dyna

Our language, Dyna, aims to readily capture *equational* relationships with a minimum of fuss. In place of (7) for (6), we write more simply

```
parent(charlie,alice) = 0.75.
sibling(A,B) += parent(C,A) * parent(C,B).
```
(8)

The `+=` carries out summation over variables in the body which are not in the head, in this case `C`. For *each* `A` and `B`, the value of `sibling(A,B)` is being defined via a sum over values of the other variables in the rule, namely `C`.

The key point is that a Datalog program proves **items**, such as `sibling(alice,bob)`, but a Dyna program also proves a **value** for each provable item (cf. [113]). Thus, a Dyna program defines a partial

[5] However, it is common to allow relations like $<$ in rules or queries that only use them to select from finite relations, since then the results remain finite [198,37].

[6] This sum cannot necessarily be interpreted as the probability of siblinghood (for that, see related work in §2.5). We use definition (6) only to illustrate aggregation.

function from items to values. Values are numeric in this example, but in general may be arbitrary ground terms.[7]

Non-provable items have no value and are said to be **null**. In general, null items do not contribute to proofs of other items, nor are they retrieved by queries.[8]

Importantly, only **ground terms** (variable-free terms) can be items (or values), so `sibling(A,B)` is not itself an item and cannot have values. Rather, the `+=` rule above is a schema that defines infinitely many grounded rules such as

```
sibling(alice,bob) += parent(charlie,alice)
                      * parent(charlie,bob).
```
(9)

which contributes a summand to `sibling(alice,bob)` iff `parent(charlie,bob)` and `parent(charlie,alice)` are both provable (i.e., have values).

The Dyna program may include additional rules beyond (8) that contribute additional summands to `sibling(alice,bob)`. All rules for the same item must specify the same **aggregation operator** (or **aggregator** for short). In this case that operator is `+=` (summation), so `sibling(alice,bob)` is defined by summing the value of γ over *all* grounded rules of the form `sibling(alice,bob) +=` γ such that γ is provable (non-null). If there are no such rules, then `sibling(alice,bob)` is null (note that it is not 0).[9]

In the first line of (8), the aggregation operator is `=`, which simply returns its single aggregand, if any (or gives an error if there are multiple aggregands). It should be used for clarity and safety if only one aggregand is expected. Another special aggregator we will see is `:=`, which chooses its *latest* aggregand, so that the value of a `:=` item is determined by the *last* rule (in program order) to contribute an aggregand to it (it is an error for that rule to contribute multiple aggregands).

However, most aggregators are like `+=`, in that they do not care about the order of aggregands or whether there is more than one, but simply reduce the multiset of aggregands with some associative and commutative binary operator (e.g, `+`).[10]

Ordinary Datalog as in (1) can be regarded as the simple case where all provable items have value `true`, the comma operator denotes boolean conjunction (over the subgoals of a proof), and the aggregator `:-` denotes boolean disjunction (over possible proofs). Thus, `true` and null effectively form a 2-valued logic. Semiring-weighted Datalog programs [89,71,92] correspond to rules like (8) where `+` and `*` denote the operations of a semiring.

2.4 Restoring Expressivity

Although our motivation comes from deductive databases, Dyna relaxes the restrictions that Datalog usually imposes, making it less like Datalog and more like the pure declarative fragment of Datalog's ancestor Prolog

[7] The statement `f(a,b)=c` may be regarded as equivalent to `f_value(a,b,c)=true`, but the former version implies a functional dependency: `f(a,b)` will have at most one value. The three-place relation `f_value` can be still refererenced using the Dyna construction `c is f(a,b)`, which enables queries in which `c` is bound; see footnote 25. The limitation to a single value for `f(a,b)` involves no loss of generality, since this value could be a tuple, or a second value could be defined as the value of `g(a,b)`.

[8] Dyna's support for non-monotonic reasoning (e.g., Figure 5) does enable rules to determine whether an item is null, or to look up such items. This is rarely necessary.

[9] This modifies completion semantics [45] for our multi-valued setting. In the completion semantics for boolean logic programs, an item's value is true if any of the γ's are true, *and false otherwise*. In our setting, the value is an aggregation over the γ's if any are true, *and null otherwise*. (We cannot use `false` as a special value because our values are not necessarily boolean, and even in the boolean case, our aggregator is not necessarily disjunction. We did consider associating a default value with each aggregator, such as the identity `0` for `+=`, but we found that it is often convenient to distinguish this value from null; also, some aggregators have no natural default. One can still force a default `0` by adding the explicit rule `sibling(A,B) += 0` to (8).)

[10] An aggregation operator may transform the aggregands before reduction and the result after reduction. This permits useful aggregators like `mean=` and `argmax=`. These transformations, and reducing functions, may be defined in Dyna, allowing programs to define their own aggregators.

(cf. Mercury [138]).[11] As we will see in §3.1, relaxing these restrictions is important to support our use cases from AI.

– **Flatness**: We drop this requirement so that Dyna can work with lists and other nested terms and perform unbounded computations.[12] However, this makes it Turing-complete, so we cannot guarantee that Dyna programs will terminate. That is the programmer's responsibility.
– **Range restriction**: We drop this requirement primarily so that Dyna can do default and non-monotonic reasoning, to support general function definitions, and to simplify certain source-to-source program transformations [69]. However, this complicates Dyna's execution model.
– **Stratification**: We drop this requirement because Dyna's core uses include many non-stratified design patterns such as recurrent neural networks, message passing, iterative optimization, and dynamic programming. Indeed, the examples in §3.1 are mainly non-stratified. These domains inherently rely on cyclic systems of equations. However, as a result, some Dyna programs may not converge to a unique solution (partial map from items to values) or even to any solution.

The difficulties mentioned above are inevitable given our use cases. For example, an iterative learning or optimization procedure in AI[13] will often get stuck in a local optimum, or fail to converge. The procedure makes no attempt to find the global optimum, which may be intractable. Translating it to Dyna, we get a non-stratified Dyna program with multiple supported models[14] that correspond to the local optima. Our goal for the Dyna engine is merely to mimic the original AI method; hence we are willing to return any supported model, accepting that the particular one we find (if any) will be sensitive to initial conditions and procedural choices, as before. This is quite different from usual practice in the logic programming community (see [157] for a review and synthesis), which when it permits non-stratified programs at all, typically identifies their semantics with one [85] or more [134] "stable models" or the intersection thereof [201,112], although in general the stable models are computationally intractable to find.

A simple example of a non-stratified program (with at most one supported model [170]) is single-source shortest paths,[15] which defines the total cost from the `start` vertex to each vertex `V`:

```
cost_to(start) min= 0.
cost_to(V)     min= cost_to(U) + edge_cost(U,V).
```
(10)

The aggregator here is `min=` (analogous to `+=` earlier) and the second rule aggregates over values of `U`, for each `V`. The weighted directed graph is specified by the `edge_cost` items. These are to be provided as extensional input or defined by additional rules (which could specify a very large or infinite graph).

Optional Typing The user is free in this example to choose any suitable ground terms to name the graph's vertices,[16] since this version of the program does not restrict the type of the arguments to `cost_to` and

[11] Of course, Dyna goes beyond pure Prolog, most importantly by augmenting items with values and by adding declarative mechanisms for situations that Prolog would handle non-declaratively with the cut operator. We also consider a wider space of execution strategies than Prolog's SLD resolution.

[12] For example, in computational linguistics, a parser's hypotheses may be represented by arbitrarily deep terms that are subject to unification [158,178]. This is because some non-context-free grammar formalisms distinguish infinitely many types of linguistic phrases [84,194,117,161,109,49,106,48], as do some parsing algorithms even for context-free grammar [155].

[13] Such as expectation-maximization, gradient descent, mean-field inference, or loopy belief propagation (see Figure 7).

[14] A **model** (or interpretation) of a logic program P is a partial map $[\![\cdot]\!]$ from items to values. A **supported model** [9] is a fixpoint of the "immediate consequence" operator T_P associated with that program [200]. In our setting, this means that for each item α, the value $[\![\alpha]\!]$ (according to the model) equals the value that would be computed for α (given the program rules defining α from other items and the values of those items according to the model).

[15] It is important that (10) is not stratified. [91] shows that shortest paths suffers aysmptotic slowdown when it is recast in a stratified form. The stratified program must compute all paths and then find the minimum cost total path.

[16] That is, vertices could be designated by built-in primitive terms such as integers (`17`), strings (`"bal"`), or lists (`[1,2,3]`); by compound terms (`vertex(17)` or `latitude_longitude(39.3,-76.6)`); or by foreign objects that are handed as data blobs to the Dyna runtime environment and treated within Dyna as primitive terms.

`edge_cost`. The untyped variables `U` and `V` range over the entire Herbrand universe of ground terms. This freedom might not be used in practice, since presumably `edge_cost(U,V)` is defined (non-null) only for pairs of ground terms (`U`, `V`) that correspond to edges in the graph. Thus `cost_to(V)` will be provable (non-null) only for ground terms `V` that denote vertices reachable from `start`. Declaring types that restrict `U` and `V` to some set of legal vertex names is optional in Dyna.

Evaluation The above example (10) also illustrates **evaluation**. The `start` item refers to the start vertex and is evaluated in place, i.e., replaced by its value, as in a functional language.[17] The items in the body of line 2 are also evaluated in place: e.g., `cost_to("bal")` evaluates to `20`, `edge_cost("bal","nyc")` evaluates to `100`, and finally `20+100` evaluates to `120` (for details, see (29) and footnote 84). This notational convention is not deep, but to our knowledge, it has not been used before in logic programming languages.[18] We find the ability to write in a style close to traditional mathematics quite compelling.

2.5 Related Work

Several recent AI projects have developed attractive probabilistic programming languages, based on logic (e.g., PRISM [214], ProbLog [165,114], Markov Logic [59]) or functional programming (e.g., IBAL [159,163], Church [90]).

By contrast, Dyna is not specifically probabilistic. Of course, Dyna items *may* take probabilities (or approximate probabilities) as their values, and the rules of the program *may* enforce a probabilistic semantics. However, the value of a Dyna item can be any term (including another dynabase). In [68], we wrote:

> For example, when using Dyna to implement an A* or best-first search, some values will be only *upper bounds* on probabilities, or other numeric priorities. In Markov Random Fields or weighted FSAs, many values are *unnormalized potentials*. In loopy or max-product belief propagation, values are only *approximate* (and perhaps unnormalized) probabilities. In recurrent neural networks, values can be *activations* that decay over time.
>
> Other useful *numeric* values in ML include log-probabilities, annealed probabilities, (expected) losses or rewards, event counts, reusable numerators and denominators, intermediate computations for dynamic programming, transfer functions, regularization terms, feature weights, distances, tunable hyperparameters, and partial derivatives. Dyna programs are free to manipulate all of these.
>
> As Dyna also allows *non-numeric* values, it can handle semiring computations, logical reasoning (including default reasoning), and manipulation of structured objects such as lists, trees, and maps.

We will see several useful examples in §3.1.

There are other logic programming formalisms in which provable terms are annotated by general values that need not be probabilities (some styles are exemplified by [113,89,79]). However, to our knowledge, all of these formalisms are too restrictive for our purposes.

In general, AI languages or toolkits have usually been designed to enforce the semantics of some *particular* modeling or algorithmic paradigm within AI.[19] Dyna, by contrast, is a more relaxed and general-purpose

[17] Notice that items and their values occupy the same universe of terms—they are not segregated as in §2.2. Thus, the value of one item can be another item (a kind of pointer) or a subterm of another item. For example, the value of `start` is used as a subterm of `cost_to(...)`. As another example, extending (10) to actually extract a shortest path, we define `best_path(V)` to have as its value a list of vertices:

```
best_path(V) ?= [U | best_path(U)]
                  whenever cost_to(V) == cost_to(U)
                                        + edge_cost(U,V).
```

(Here the construction `[First | Rest]` prepends an element to a list, as in Prolog. The “free-choice” aggregator `?=` allows the system to arbitrarily select any one of the aggregands, hence arbitrarily breaks ties among equally short paths.)

[18] With the exception of the hybrid functional-logic language Curry [54]. Curry is closer to functional programming than to Datalog. Its logical features focus on nondeterminism in lazy evaluation, and it does not have aggregation.

[19] Non-probabilistic examples include COMET [100] and ECLiPSe [10] for combinatorial optimization, as well as OpenFST [7] for finite-state transduction.

language that aims to accommodate all these paradigms. It is essentially a general infrastructure layer: specific systems or toolkits could be written in Dyna, or more focused languages could be compiled to Dyna. Dyna focuses on defining relationships among data items and supporting efficient storage, queries, and updates given these relationships. We believe that this work is actually responsible for the bulk of the implementation and optimization effort in today's AI systems.

2.6 A First Execution Strategy

Before we turn to our AI examples, some readers may be wondering how programs might be executed. Consider the shortest-path program in (10). We wish to find a fixed point of the system of equations that is given by those rules (grounding their variables in all possible ways) plus the extensional data.

Here we can employ a simple **forward chaining** strategy (see [71] for details and pseudocode). The basic idea is to propagate updates from rule bodies to rule heads, until the values of all items converge.[20] We refer to items in a rule's body as **antecedents** and to the item in the rule's head as the **consequent**.

At all times, we maintain a **chart** that maps the items proved so far to their current values, and an **agenda** (or worklist) of updates that have not yet been applied to the chart. Any changes to the extensional data are initially placed on the agenda: in particular, the initial definitions of `start` and `edge_cost` items.

A step of the algorithm consists of popping an update from the agenda, applying it to the chart, and computing the effect that will have on other items. For example, finding a new, shorter path to Baltimore may cause us to discover a new, shorter path to other cities such as New York City. (Such a step is sometimes called "relaxation" in the algorithms literature.)

Concretely, when updating `cost_to("bal")` to `20`, we see that this item pattern-matches one of the antecedents in the rule

```
cost_to(V) min= cost_to(U) + edge_cost(U,V).
```
(11)

with the binding `U="bal"`, and must therefore drive an update through this rule. However, since the rule has two antecedents, the **driver** of the update, `cost_to("bal")`, needs a **passenger** of the form `edge_cost("bal",V)` to complete the update. We query the chart to find all such passengers. Suppose one result of our query `edge_cost("bal",V)` is `edge_cost("bal","nyc")=100`, which binds `V="nyc"`. We conclude that one of the aggregands of the consequent, `cost_to("nyc")`, has been updated to `120`. If that *changes* the consequent's value, we place an update to the consequent on the agenda.

This simple update propagation method will be helpful to keep in mind when studying the examples in Figures 1–12. We note, however, that there is a rich space of execution strategies, discussed in §5.2.

2.7 Multiple Interacting Dynabases

So far we have considered only one dynabase at a time. However, using multiple interacting dynabases is useful for encapsulation, inheritance, and "what if" analysis where one queries a dynabase under changes to its input items.

Readers interested mainly in AI will want to skip the artificial example in this section and move ahead to §3, returning here if needed when multiple dynabases come into play partway through §3.1 (in Figure 7 and especially Figures 11 and 12).

All code fragments in this section are part of the definition of a dynabase that we call δ. We begin by defining some ordinary items of δ:

```
three = 3.
e = { pigs += 100.       % we have 100 adult pigs
      pigs += piglets.   % and any piglets we have are also pigs
    }.
```
(12)

[20] This is a record-at-a-time variant of semi-naive bottom-up evaluation.

In δ, the value of `three` is `3` and the value of `e` is a particular dynabase ε. Just as `3` is a **numeric literal** in the program that specifies a number, the string `{...}` is an **dynabase literal** that specifies a **literal dynabase** ε.One could equivalently define[21]

```
e = $load("pigpen").
```
(13)

where the file `pigpen.dyna` consists of "`pigs += 100. pigs += piglets.`" or a compiled equivalent, and thus `$load("pigpen")` evaluates to ε. If the file is edited, then the values of `$load("pigpen")` and `e` will auto-update.

Since ε does not declare its items `pigs` and `piglets` to be private, our rules in δ can refer to them as `e.pigs` and `e.piglets`, which evaluate to `100` and null. (More precisely, `e` evaluates to ε within the expression `e.pigs`, and the resulting expression ε`.pigs` looks up the value of item `pigs` in dynabase ε.)

Storing related items like `pigs` and `piglets` in their own dynabase ε can be a convenient way to organize them. Dynabases are first-class terms of the language, so one may use them in item names and values. For example, this definition of matrix transposition

```
transpose(Matrix) = { element(I,J) = Matrix.element(J,I). }.
```
(14)

defines for each dynabase μ an item `transpose(`μ`)` whose value is also a dynabase. Each of these dynabases is an encapsulated collection of many elements. Notice that `transpose` resembles an object-oriented function that takes an object as an argument and returns an object.

However, the real power of dynabases comes from the ability to **extend** them. Remember that a dynabase is a *dynamic* deductive database: ε`.pigs` is defined in terms of ε`.piglets` and is supposed to increase when ε`.piglets` does. However, ε`.piglets` cannot actually change because ε in our example is an immutable constant. So where does the dynamism come in? How can a procedural program, or another dynabase, supply new input to ε once it has defined or loaded it?

A procedural program can create a new **extension** of ε: a modifiable copy ε'. As the **owner** of ε', the program can freely specify new aggregands to its writeable items. That serves to *increment* ε'`.pigs` and *replace* ε'`.piglets` (assuming that their aggregators are respectively `+=` and `:=`; see §2.3). These updates affect only ε' and so are not visible to other users of ε.[22] The procedural program can interleave updates to ε' with queries against the updated versions (see §1).

A Dyna program with access to ε can similarly extend ε with new aggregands; here too, changes to `piglets` will feed into `pigs`. Continuing our definition of δ:

```
f = new e.          % f is a new pigpen φ that inherits all rules of ε
f.pigs     += 20.      % but has 20 extra adult pigs
f.piglets  := three.   % and exactly three piglets
```
(15)

These rules are written as part of the definition of δ (the owner[23] of the new dynabase φ) and supply new aggregands `20` and `3` to φ's versions of `pigs` and `piglets`.

[21] Reserved-word functors such as `$load` start with \$, to avoid interference with user names of items.

[22] The converse is not true: any updates to ε would be inherited by its extension ε'.

[23] Because δ invoked the `new` operator that created φ, δ is said to **own** φ. This is why δ is permitted to have rules that extend φ with additional aggregands as shown in (15). Ownership is reflexive and transitive.

By contrast, it would be an error for δ to have a rule of the form `e.pigs += 1` or `transpose(Matrix).foo += 1...`, since the values of `e` and `transpose(Matrix)` are dynabases that δ does not own. (Literal dynabases are not owned by anyone, since they are not created by extension.) In functional programming terms, δ cannot modify the result of the `transpose` function, which may have other users, although it could create its own private extension of it.

Similarly, it would be an error for the dynabase literal in (14) to include a rule such as `Matrix.x +=...`, because that literal does not own the dynabase bound by `Matrix`. Such a rule would attempt to modify items of μ as a "side effect" of "calling" `transpose(`μ`)`; it is unclear what such a side effect would mean, since `transpose(`μ`)` is not a procedure that is called on certain dynabases μ, but rather is a universal definition for all μ.

The **parent** dynabase ε remains unchanged, but its extension φ has items `pigs` and `piglets` with values `123` and `3`, just as if it had been defined in the first place by combining (12) and (15) into[24]

```
f = { pigs    += 100.
      pigs    += piglets.
      pigs    += 20.
      piglets := $owner.three. }    % where $owner refers to δ
```
(16)

The important point is that setting `f.piglets` to have the same value as `three` also affected `f.pigs`, since ε defined `pigs` in terms of `piglets` and this relationship remains operative in any extension of ε, such as `f`'s value φ.

Interactions among dynabases can be quite flexible. Some readers may wish to see a final example. Let us complete the definition of δ with additional rules

```
g = new e.
offspring = g.pigs / three.     % all pigs have babies
g.piglets := offspring.         % who are piglets
```
(17)

This creates a loop by feeding $\frac{1}{3}$ of `g`'s "output item" `pigs` back into `g`'s "input item" `piglets`, via an intermediate item `offspring` that is not part of `g` at all. The result is that `g.pigs` and `g.piglets` converge to `150` and `50` (e.g., via the forward chaining algorithm of §2.6). This is a correct solution to the system of equations specified by (12) and (17), which state that there are 100 more pigs than piglets and $\frac{1}{3}$ as many piglets as pigs:

$$\begin{aligned} \delta.\texttt{three} &= 3 & \delta.\texttt{offspring} &= \gamma.\texttt{pigs}/\delta.\texttt{three} \\ \gamma.\texttt{pigs} &= 100 + \gamma.\texttt{piglets} & \gamma.\texttt{piglets} &= \delta.\texttt{offspring} \end{aligned} \tag{18}$$

Dynabases are connected to object-oriented programming (§4.4). We will see practical uses of multiple dynabases for encapsulation (Figure 7), modularity (Figure 11), and backtracking search (Figure 12). More formal discussion of the overall language semantics, with particular attention to dynabase extension, can be found in the appendix (§A).

3 Design Patterns in AI

Given the above sketch, we return to the main argument of the paper, namely that Dyna is an elegant declarative notation for capturing the logical structure of computations in modern statistical AI.

Modern AI systems can generally be thought of as observing some input and recovering some (hidden) structure of interest:

– We observe an image and recover some description of the scene.
– We observe a sentence of English and recover a syntax tree, a meaning representation, a translation into Chinese, etc.
– We are given a goal or reward function and recover a plan to earn rewards.
– We observe some facts expressed in a knowledge representation language and recover some other facts that can be logically deduced or statistically guessed from them.
– We observe a dataset and recover the parameters of the probability distribution that generated it.

Typically, one defines a discrete or continuous space of possible structures, and learns a scoring function or probability distribution over that space. Given a partially observed structure, one either tries to recover the best-scoring completion of that structure, or else queries the probability distribution over all possible completions. Either way, the general problem is sometimes called **structured prediction** or simply **inference**.

[24] The last rule of φ evaluates `three` not in δ, where `three` was originally defined, but rather in "my owner" (in the sense of footnote 23). This is important because if δ' is a further extension of δ, it will reuse the entire definition of δ, so δ'`.f.piglets` will depend on δ'`.three`, not δ`.three`. See §A for details.

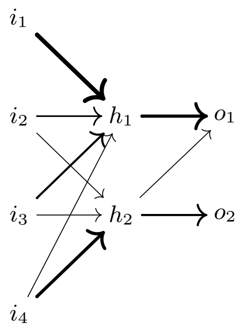


Fig. 1: A small acyclic neural network. The activation x_n at each node n is a nonlinear function f, such as a sigmoid or threshold function, of a weighted sum of activations at n's parent nodes: $x_n \overset{\text{def}}{=} f\left(\sum_{(n',n)\in E} x_{n'} w_{n',n}\right)$. The three layers shown here are the traditional input, hidden, and output nodes, with $w_{n',n}$ values represented by arrow thickness.

```
sigmoid(X) = 1 / (1 + exp(-X)).
output(Node) = sigmoid(input(Node)).
input(Node) += output(Child) * weight(Child,Node).
error += (output(Node) - target(Node))**2.
```

Fig. 2: A general neural network in Dyna. Line 1 defines the sigmoid function over all real numbers `X`. In Line 2, that function is applied to the *value* of `input(Node)`, which is evaluated in place. Line 3 sums over all incoming edges to `Node`. Those edges are simply the (`Child`,`Node`) pairs for which `weight(Child,Node)` is defined. Additional summands to some of the `input(Node)` items may be supplied to this dynabase at runtime; this is how i_1, i_2, i_3, i_4 in Figure 1 would get their outside input. Finally, Line 4 evaluates error by summing over just those nodes for which `target(Node)` has been defined (i.e., is non-null), presumably the output nodes o_j.

3.1 Brief AI Examples in Dyna

We will show how to implement several AI patterns in Dyna. All the examples in this section are brief enough that they are primarily pedagogical—they could be used to teach and experiment with these basic versions of well-known methods.

Real systems correspond to considerably larger Dyna programs that modify and combine such techniques. Real systems must also fill in problem-specific details. Rather than being handed a graph, circuit, set of constraints, set of proof rules, context-free grammar, finite-state automaton, cost model, etc., they might define it themselves in terms of extensional raw data using additional Dyna rules.

Each of the code examples below is in a self-contained figure, with details in the captions. Typically the program defines a dynabase in which all items are still null, as it merely defines intensional items in terms of extensional items that have not been supplied yet. One may however extend this dynabase (see §2.7), adding observed structure (the input) and the parameters of the scoring function (the model) as extensional data. Results now appear in the extended dynabase as intensional data defined by the rules, and one may read them out.

Arithmetic Circuits One simple kind of system is an arithmetic circuit. A classic example in AI is a neural net (Figure 1). In the Dyna implementation (Figure 2), the network topology is specified by defining values for the `weight` items.

As in the shortest-path program (10), the items that specify the topology may be either provided directly at runtime (as extensional data), or defined by additional Dyna rules (as intensional data). Figure 3 gives an attractive example of a concise intensional definition.

[25] These names are not items but appear in the rule as *unevaluated* terms. However, the expressions `X+I` and `Y+J` are evaluated in place, so that the rule is equivalent to

```
weight(pixel(X2,Y2), hidden(X,Y)) = shared_weight(I,J)
                                    whenever X2 is X+I, Y2 is Y+I.
```

```
weight(pixel(X+I,Y+J), hidden(X,Y)) = shared_weight(I,J).
```

Fig. 3: One layer of a neural network topology for vision, to be used with Figure 2. Each hidden node `hidden(X,Y)` is connected to a 5×5 rectangle of input nodes `pixel(X+I,Y+J)` for $\mathtt{I}, \mathtt{J} \in \{-2, -1, 0, 1, 2\}$, using a collection of 25 weights that are reused across spatial positions `(X,Y)`. The `shared_weight(I,J)` items should be defined (non-null) only for $\mathtt{I}, \mathtt{J} \in \{-2, -1, 0, 1, 2\}$. This rule then connects nodes with related names, such as such as `hidden(75,95)` and `pixel(74,97)`.

This rule exploits the fact that the node names are structured objects.[25] By using structured names, we have managed to specify an *infinite* network in a single line (plus 25 weight definitions). Only a finite portion of this network will actually be used by Figure 2, assuming that the image (the collection of `pixel` items) is finite.

```
count(X,Y) += 0 whenever is_event(X), is_event(Y).  % default
count(X)   += count(X,Y).
count      += count(X).

% Maximum likelihood estimates
mle_prob(X)   = count(X)   / count.
mle_prob(X,Y) = count(X,Y) / count(Y).

% Good-Turing smoothed estimates [148]
gt_prob(X)   = total_mle_prob(count(X)+1)   / n(count(X)).
gt_prob(X,Y) = total_mle_prob(count(X)+1,Y) / n(count(X),Y).

% Used by Good-Turing: How many events X occurred R times, or
% cooccurred R times with Y, and what is their total probability?
n(R) += 0.               n(R)   += 1 whenever R==count(X).
n(R,Y) += 0.             n(R,Y) += 1 whenever R==count(X,Y).
total_mle_prob(R)   += mle_prob(X)   whenever R==count(X).
total_mle_prob(R,Y) += mle_prob(X,Y) whenever R==count(X,Y).
```

Fig. 4: Estimating conditional probabilities $p(x)$ and $p(x \mid y)$, based on counts of x with y. The user can simply increment `count(`x`,`y`)` whenever x is observed together with y, and the probability estimates will update (see §4.3).

The user should also set `is_event(`x`)` to `true` for each possible event x, to ensure that even never-observed events will have a defined count (of 0) and will be allocated some probability; `n(0)` counts the number of never-observed events. The final four lines could be written more concisely; e.g., the first of them as `n(count(X)) += 1.` The final two lines should be optimized [69], e.g., the first of them is equivalent to `total_mle_prob(R) = (R/count)*n(R).`

Notice that line 3 of Figure 2 is a typical matrix-vector product. It is sparse because the neural-network topology is typically a sparse graph (Figure 1). Sparse products are very common in AI. For example, sparse dot products are used both in computing similarity and in linear or log-linear models [52]. A dot product like `score(Structure) += weight(Feature)*strength(Feature,Structure)` resembles line 3, and can benefit from using complex feature names, just as Figure 3 used complex node names.

A rather different example of arithmetic computation is shown in Figure 4, a dynabase that maintains probability estimates based on the counts of events. Some other commonly used arithmetic formulas in AI include distances, kernel functions, and probability densities.

where in general, the condition γ `is` α has value `true` if γ is the value of item α, and is null otherwise. For example, `97 is 95+2` has value `true`.

```
fly(X) := false.
fly(X) := true if bird(X).
fly(X) := false if penguin(X).
fly(bigbird) := false.
```

Fig. 5: An example of non-monotonic reasoning: all birds fly, other than Sesame Street's Big Bird, until such time as they are proved or asserted to be penguins. Recall from §2.3 that the `:=` aggregator is sensitive to rule ordering, so that where the later rules apply at all, they override the earlier rules. The first rule is a "default rule" that is not range-restricted (see §2.1): it proves infinitely many items that unify with a pattern (here the very simple pattern `X`).

Training of Arithmetic Circuits To train a neural network or log-linear model, one must adjust the `weight` parameters to reduce `error`. Common optimization methods need to consult more than just the current `error`: they need to query the gradient of `error` with respect to the parameters. How can they obtain the gradient? **Automatic differentiation** can be written very naturally as a source-to-source transformation on Dyna programs, automatically augmenting Figure 2 with rules that compute the gradient by back-propagation [71]. The gradient can then be used by other Dyna rules or queried by a procedural optimizer. Alternatively, the execution engine of our prototype Dyna implementation natively supports [71] computing gradients, via tape-based automatic differentiation in the reverse mode [93]. It is designed to produce exact gradients even of incomplete computations.

An optimizer written in a conventional procedural language can iteratively update the `weight` items in the dynabase of Figure 2, observing at each step how the `output`, `error`, and gradient change in response. Or the optimizer could be written in Dyna itself, via rules that define the weights at time step `T+1` in terms of items (e.g., gradient) computed at time step `T`. This requires adding an explicit time argument `T` to all terms (another source-to-source transformation).

Theorem Proving Of course, logic and logic programming have a long history in symbolic AI. Traditional systems for knowledge representation and reasoning (KRR) are all automated theorem provers, from SHRDLU [208] to the on-going Cyc expert system [121] and description logic systems [11] such as OWL for the semantic web [137]. They compute the entailments of a set of axioms obtained from human input or derived by other theorem provers (e.g., OWL web services).

Logical languages like Dyna support these patterns naturally. The extensional items are axioms, the intensional ones are theorems, and the inference rules are the rules of the program. For example, one of the most basic and general inference rules is *modus ponens*, which states "From the propositions 'A implies B' and A, you may derive the proposition B." In logic, this rule is often written as

$$\frac{A \Rightarrow B \quad A}{B}$$

with the antecedents above the line and the consequents below. This corresponds to a Dyna or Prolog rule[27]

```
B :- implies(A,B), A.
```
(19)

Dyna also naturally handles some forms of default and non-monotonic reasoning [19], via `:=` rules like those in Figure 5. A related important use of default patterns in AI is "lifted inference" [181] in probabilistic settings like Markov Logic Networks [169], where additional (non-default) computation is necessary only for individuals about whom additional (non-default) facts are known. Yet another use in AI is default arcs of various kinds in deterministic finite-state automata over large or unbounded alphabets [7,152,15].[28]

[27] Datalog does not suffice, if we wish A and B to range over arbitrary terms and not merely atoms, as in systems like KRL [23].

[28] At a given state of the automaton, one can concisely specify some default transitions, but then override these defaults in selected cases. For example, to mimic the special ρ and ϕ arcs supported by the OpenFST toolkit [7],

Some emerging KRR systems embrace statistics and draw probabilistic inferences rather than certain ones. Simple early approaches included [162,150,145]. More recent examples include ProbLog [165,114] and Markov Logic Networks [169,6]. Their computations can typically be described in Dyna by using real-valued items.

Message Passing Many AI algorithms come down to solving (or approximately solving) a system of simultaneous equations, often by iterating to convergence. In fact, the neural network program of Figure 2 already requires iteration to convergence in the case of a cyclic ("recurrent") network topology [207].

Such iterative algorithms are often known as "message passing" algorithms. They can be regarded as *negotiating* a stable configuration of the items' values. Updates to one item trigger updates to related items—easily handled in Dyna since update propagation is exactly what a basic forward-chaining algorithm does (§2.6). When the updates can flow around cycles, the system is not stratified and sometimes has no guarantee of a unique fixed point, as warned in §2.4.

Message passing algorithms seek possible, likely, or optimal values of random variables under a complex set of hard or soft constraints. Figure 6 and Figure 7 show two interesting examples in Dyna: arc consistency (with boolean values) and loopy belief propagation (with unnormalized probabilities as the values).[29] Other important examples include alternating optimization algorithms such as expectation-maximization and mean-field. Markov chain Monte Carlo (MCMC) and simulated annealing algorithms can also be regarded as message passing algorithms, although in this case the updates are randomized (see (24) below). n

Dynamic Programming Dyna began [70,71] as a language for dynamic programming (hence the name). The connection of dynamic programming to logic programming has been noted before (e.g., [94]). Fundamentally, dynamic programming is about solving subproblems and reusing stored copies of those solutions to solve various larger subproblems. In Dyna, the subproblems are typically named by items, whose values are their solutions. An efficient implementation of Dyna will typically store these solutions for reuse,[30] whether by backward chaining that lazily memoizes values in a table (as in XSB [209] and other tabled Prologs), or by forward chaining that eagerly accumulates values into a chart (as in §2.6 and the Dyna prototype [71]).

A traditional dynamic programming algorithm can be written directly in Dyna as a set of recurrence equations. A standard first example is the Fibonacci sequence, whose runtime goes from exponential to linear in `N` if one stores enough of the intermediate values:

```
fib(N) := fib(N-1) + fib(N-2).   % general rule
fib(0) := 1.                     % exceptions for base cases        (20)
fib(1) := 1.
```

As a basic AI example, consider context-free parsing with a CKY-style algorithm [211]. The Dyna program in Figure 8 consists of 3 rules that directly and intuitively express how a parse tree is recursively built up

```
arc(q,Letter) := r.     % single default: ρ arc from state q to state r
arc(q,"x") := s.        % override default: on input letter "x", go to s instead
```

```
arc(q,Letter) := arc(r,Letter).  % inherited defaults: φ arc from q to r
arc(q,"x") := s.        % override default: on input letter "x", go to s instead
```

This concise intensional structure can be exploited directly within an algorithm such as the forward-backward algorithm, the Viterbi algorithm, or automaton intersection. However, nontrivial rearrangements of the standard algorithm can be needed to avoid materializing all transitions from state `q`. An efficient implementation of Dyna would have to discover these optimizations.

[29] Twists on these programs give rise to other popular local consistency algorithms (bounds consistency, i-consistency) and propagation algorithms (generalized belief propagation, survey propagation).

[30] This support for reuse is already evident in our earlier examples, even though they would not traditionally be regarded as dynamic programming. For example, the activation of node h_1 in Figure 1 (represented by some `output` item in Figure 2) takes some work to compute, but once computed, it is reused in the computation of each node o_j. Similarly, each count `n(R)` or `n(R,Y)` in Figure 4 is reused to compute many smoothed probabilities.

```
% For Var:Val to be possible, Val must be in-domain, and
% also supported by each Var2 that is co-constrained with Var.
% The conjunctive aggregator &= is like universal quantification over Var2.
possible(Var:Val) &= in_domain(Var:Val).
possible(Var:Val) &= supported(Var:Val, Var2).
p
% Var:Val is supported by Var2 only if it is still possible
% for Var2 to take some value that is compatible with Val.
% The disjunctive aggregator |= is like existential quantification over Val2.
supported(Var:Val, Var2)
    |= compatible(Var:Val, Var2:Val2) & possible(Var2:Val2).

% If consistent ever becomes false, we have detected unsatisfiability:
% some variable has no possible value.
non_empty(Var) |= false.               % default (if there are no possible values)
non_empty(Var) |= possible(Var:Val).  % Var has a possible value
consistent &= non_empty(Var) whenever is_var(Var).
                                     % each Var in the system has a possible value
```

Fig. 6: Arc consistency for constraint programming [57]. The goal is to rule out some impossible values for some variables, using a collection of unary constraints (`in_domain`) and binary constraints (`compatible`) that are given by the problem and/or tested during backtracking search (see Figure 12). The "natural" forward-chaining execution strategy for this Dyna program corresponds to the classical, asymptotically optimal AC-4 algorithm [141].

Variables and constraints can be named by arbitrary terms. `Var:Val` is syntactic sugar for an ordered pair, similar to `pair(Var,Val)` (the `:` has been declared as an infix functor). The program determines whether `possible(Var:Val)`. The user should define `is_var(Var)` as `true` for each variable, and `in_domain(Var:Val)` as `true` for each value `Val` that `Var` should consider. To express a binary constraint between the variables `Var` and `Var2`, the user should define `compatible(Var:Val, Var2:Val2)` to be `true` or `false` for each value pair `Val` and `Val2`, according to whether the constraint lets these variables simultaneously take these values. This ensures that `supported(Var:Val,Var2)` will be `true` or `false` (not null) and so will contribute a conjunct to line 2.

by combining adjacent phrases into larger phrases, under the guidance of a grammar. The forward-chaining algorithm of §2.6 here yields "agenda-based parsing" [178]: when a recently built or updated phrase pops off the agenda into the chart, it tries to combine with adjacent phrases in the chart.

We will return to this example in §3.2. Meanwhile, the reader is encouraged to figure out why it is not a stratified program (§2.2), despite being based on the stratified CKY algorithm.[31] Replacing the `+=` aggregator with `max=` (compare (10)) would make it find the probability of the single best parse, instead of the total probability of all parses [89].

[31] Answer: In Figure 8, `phrase(X,I,J)` may contribute cyclically to its own proof (like the item `b` in Figure 13). An extreme example: if one asserts the cyclic input `word("blah",0,0)=1`, that program becomes a recurrent quadratic system in variables of the form `phrase(X,0,0)`. This is actually useful: `goal` now gives the total probability of *all* strings of the form `blah blah blah`... (given `length=0`).

In general, stratification fails in the presence of unary grammar productions, which may form cycles such as `X` $\to$ `Y` and `Y` $\to$ `X`. (While unary cycles may be eliminated by preprocessing the grammar [69], the program for preprocessing is itself not stratified.)

CKY (Figure 8) does not allow unary productions—but binary productions act like unary productions if one child is a phrase of width 0, and the implementation in Figure 8 does not rule out width-0 phrases. Width-0 phrases may arise when the input is a cyclic finite-state automaton instead of a string, as in the `word("blah",0,0)` example. They may also arise from the useful non-range-restricted rule `word(epsilon,I,I)`, which says that the empty string ϵ can be found anywhere in the input sentence.

```
% Belief at each variable based on the messages it receives from constraints.
belief(Var:Val) *= message(Con, Var:Val).

% Belief at each constraint based on the messages it receives from variables
% and the preferences of the constraint itself.
belief(Con:Asst) = messages_to(Con:Asst) * constraint(Con:Asst).

% To evaluate a possible assignment Asst to several variables, look at messages
% to see how well each variable Var likes its assigned value Asst.Var.
messages_to(Con:Asst) *= message(Var:(Asst.Var), Con).

% Message from a variable Var to a constraint Con. Var says that it plausibly
% has value Val if Var independently believes in that value (thanks to other
% constraints, with Con's own influence removed via division).
message(Var:Val, Con) := 1.   % initial value, will be overridden
message(Var:Val, Con) := belief(Var:Val) / message(Con, Var:Val).

% Messages from a constraint Con to a variable Var.
% Con says that Var plausibly has value Val if Con independently
% believes in one or more assignments Asst in which this is the case.
message(Con, Var:Val) += belief(Con:Asst) / message(Var:Val, Con)
                                whenever Asst.Var == Val.
```

Fig. 7: Loopy belief propagation on a factor graph [156,210]. The constraints together define a Markov Random Field joint probability distribution over the variables. We seek to approximate the marginals of that distribution: at each variable `Var` we will deduce a belief about its value, in the form of relative probabilities of the possible values `Val`. Similarly, at each constraint `Con` over a *set* of variables, we will deduce a belief about the correct *joint assignment* of values to *just those* variables, in the form of relative probabilities of the possible assignments `Asst`.

Assignments are slightly complicated because we allow a single constraint to refer to arbitrarily many variables (in contrast to Figure 6, which assumed binary constraints). A specific assignment is a map from variable names (terms such as `color`, `size`) to their values (e.g., `red`, `3`). It is convenient to represent this map as a small sub-dynabase, `Asst`, whose elements are accessed by the `.` operator: for example, `Asst.color == red` and `Asst.size == 3`.

As input, the user must define `constraint` so that each constraint ("factor" or "potential function") gives a non-negative value to each assignment, giving larger values to its preferred assignments. Each variable should be subject to at least one constraint, to specify its domain (analogous to `in_domain` in Figure 6).

A *message* to or from a variable specifies a relative probability for each value of that variable. Since messages are proved circularly from one another, we need to initialize some messages to 1 in order to start propagation; but these initial values are overridden thanks to the `:=` aggregator, which selects its "latest" aggregand and hence prefers the aggregand from line 5 (once defined) to the initial aggregand from line 4. *Note:* For simplicity, this version of the program glosses over minor issues of message normalization and division by 0.

This example also serves as a starting point for more complicated algorithms in syntactic natural-language parsing and syntax-directed translation.[32] The uses of the Dyna prototype (§5.1) have been mainly in this domain; see [69,71] for code examples. In natural language processing, active areas of research that make

[32] The connection between parsing and logic programming has been discussed by [158,20,178,89,108,71], among others. ADP [87] is a different declarative approach to (context-free) parsing and related problems, using annotated grammars.

Some common translation approaches are based on synchronous grammars or tree transducers, and generalize Figure 8. An item of the form `phrase(X,I1,J1,I2,J2)` then represents a hypothesis that there is a phrase of type `X` from position `I1` to position `J1` of a given French sentence, which correponds to the substring from `I2` to position `J2` in the English translation.

```
% A single word is a phrase (given an appropriate grammar rule).
phrase(X,I,J) += rewrite(X,W) * word(W,I,J).
% Two adjacent phrases make a wider phrase (given an appropriate rule).
phrase(X,I,J) += rewrite(X,Y,Z) * phrase(Y,I,Mid) * phrase(Z,Mid,J).
% An phrase of the appropriate type covering the whole sentence is a parse.
goal          += phrase(start_nonterminal,0,length).
```

Fig. 8: Probabilistic context-free parsing in Dyna (the "inside algorithm"). `phrase(X,I,J)` is provable if there might be a constituent of type `X` from position `I` to position `J` of the input sentence. More specifically, the *value* of `phrase(X,I,J)` is the probability that nonterminal symbol `X` would expand into the substring that stretches from `I` to `J`. It is defined using `+=` to sum over all ways of generating that substring (considering choices of `Y`, `Z`, `Mid`). Thus, `goal` is the probability of generating the input sentence, summing over all parses.

The extensional input consists of a sentence and a grammar. `word("spring",5,6)=1` means that `"spring"` is the sixth word of the sentence; while `length=30` specifies the number of words. `rewrite("S","NP","VP")=0.9` means that any copy of nonterminal `S` has a priori probability `0.9` of expanding via the binary grammar production `S` $\rightarrow$ `NP VP`; while `start_nonterminal="S"` specifies the start symbol of the grammar.

heavy use of parsing-like dynamic programs include machine translation [41,42,107,160,177,216], information extraction [176,119,213], and question answering [55,36,204,143]. There is a tremendous amount of experimentation with models and algorithms in these areas and in parsing itself. A recent direction in machine vision attempts to parse visual scenes in a similar way [215,80]: just as a sentence is recursively divided into labeled phrases formed by composition of grammar rules applied to sub-phrases, a scene may be recursively divided into labeled objects formed from sub-objects. Dyna is potentially helpful on all of these fronts.

Other dynamic programming algorithms are also straightforward in Dyna, such as the optimal strategy in a game tree or a Markov Decision Process (Figure 9), variations from bioinformatics on weighted edit distance (Figure 10) and multiple sequence alignment, or the intersection or composition of two finite-state automata (see [46] for Dyna code).

Processing Pipelines It is common for several algorithms and models to work together in a larger AI system. Connecting them is easy in Dyna: one algorithm's input items can be defined by the output of another algorithm or model, rather than as extensional input. The various code and data resources can be provided in separate dynabases (§2.7), which facilitates sharing, distribution, and reuse.

For example, Figure 11a gives a version of Figure 8's parser that conveniently accepts its `grammar` and `input` in the form of other dynabases. Figure 11b illustrates how this setup allows painless scripting.

Figure 11c shows how the provided `grammar` may be an interesting component in its own right if it does not merely *list* weighted productions but *computes* them using additional Dyna rules (analogous to the neural network example in Figure 3). The particular example in Figure 11c constructs a context-free grammar from weights. It is equally easy to write Dyna rules that construct a grammar's productions by transforming another grammar,[33] or that specify an infinitely large grammar.[34]

Not only `grammar` but also `input` may be defined using rules. For example, the input sequence of words may be derived from raw text or speech signal using a structured prediction system—a tokenizer, morphological analyzer, or automatic speech recognizer. A generalization is that such a system, instead of just

Program transformations on declaratively specified parsing algorithms are explained by [140,179,69,105,46]. Sikkel [179] formulated a wide variety of parsing algorithms using inference rules that could be written directly in Dyna. More recently, Lopez [131] has done the same for a variety of statistical machine translation systems, and the GENPAR C++ class library [32] aids implementation of some such systems in a generic way.

[33] For example, one can transform an arbitrary weighted context-free grammar into Chomsky Normal Form for use with Figure 11a, or coarsen a grammar for use as an A* heuristic [115].

[34] For example, the non-range-restricted rule `rewrite(X/Z,X/Y,Y/Z).` encodes the infinitely many "composition" rules of combinatory categorial grammar [178], in which a complex nonterminal such as `s/(pp/np)` denotes an incomplete sentence (`s`) that is missing an incomplete prepositional phrase (`pp`) that is in turn missing a noun phrase (`np`).

```
% The optimal value function V.
value(State)      max= value(State,Action).

% The optimal action-value function Q.
% Note: The value of p(s,a,s') is a conditional transition probability, P(s' | s,a).
value(State,Action) += reward(State,Action).
value(State,Action) += γ * p(State,Action,NewState) * value(NewState).

% The optimal policy function π.  The free-choice aggregator ?= is used
% merely to break ties as in footnote 17.
best_action(State)  ?= Action if value(State) == value(State,Action).
```

Fig. 9: Finding the optimal policy in an infinite-horizon Markov decision process, using value iteration. The reward and transition probability functions can be sensitive to properties of the states, or to their structured names as in Figure 3. The optimal value of a `State` is the expected total reward that an agent will earn if it follows the optimal policy from that `State` (where the reward at t steps in the future is discounted by a factor of γ^t). The optimal value of a `(State,Action)` pair is the expected total reward that the agent will earn by first taking the given `Action`—thereby earning a specified reward and stochastically transitioning to a new state—and thereafter following the optimal policy to earn further reward.

The mutual recurrence between V and Q interleaves two different aggregators: `max=` treats optimization by the agent, while `+=` computes an expectation to treat randomness in the environment. This "expectimax" strategy is appropriate for acting in a random environment, in contrast to the "minimax" strategy using `max=` and `min=` that is appropriate when acting against an adversarial opponent. The final line with `?=` merely extracts the optimal policy once its value is known.

```
% Base case: distance between two empty strings.
dist([],[]) = 0.

% Recursive cases.
dist([X|Xs],   Ys ) min= delete_cost(X)   + dist(Xs,Ys).
dist(   Xs, [Y|Ys]) min= insert_cost(Y)   + dist(Xs,Ys).
dist([X|Xs],[Y|Ys]) min=  subst_cost(X,Y) + dist(Xs,Ys).

% Part of the cost function.
substcost(L,L) = 0.    % cost of 0 to align any letter to itself
```

Fig. 10: Weighted edit distance between two strings. This example illustrates items whose names are arbitrarily deep terms: each `dist` name encodes two strings, each being an list of letters. As in Prolog, the syntactic sugar `[X|Xs]` denotes a list of length > 0 that is composed of a first element `X` and a remainder list `Xs`.

We pay some cost for aligning the first 0 or 1 letters from one string with the first 0 or 1 letters from the other string, and then recurse to find the total cost of aligning what is left of the two strings. The choice of how many initial letters to align is at lines 2–4: the program tries all three choices and picks the one with the minimum cost. Reuse of recursive subproblems keeps the runtime quadratic. For example, if all costs not shown are 1, then `dist([a,b,c,d], [s,b,c,t,d])` has value 2. This is obtained by optimally choosing the line with `subst_cost(a,s)` at the first recursive step, then `subst_cost(b,b)`, `subst_cost(c,c)`, `insert_cost(t)`, `subst_cost(d,d)`, for a total cost of 1+0+0+1+0.

producing a single "best guess" word sequence, can often be made to produce a *probability distribution* over possible word sequences, which is more informative. This distribution is usually represented as a "hypothesis lattice"—a probabilistic finite-state automaton that may generate exponentially or infinitely many possible sequences, assigning some probability to each sequence. The parser of Figure 11a can handle this kind of

```
phrase(X,I,J) += grammar.rewrite(X,W) * input.word(W,I,J).
phrase(X,I,J) += grammar.rewrite(X,Y,Z) * phrase(Y,I,Mid)
                                        * phrase(Z,Mid,J).
goal          += phrase(grammar.start_nonterminal,0,input.length).
```

(a) A parser like that of Figure 8, except that its input items are two dynabases (denoted by `grammar` and `input`) rather than many separate numbers (denoted by `rewrite(...)`, `word(...)`, etc.).

```
% Specialize (a) into an English-specific parser.
english_parser = new $load("parser").   % parser.dyna is given in (a)
english_parser.grammar = $load("english_grammar").   % given in (c)

% Parse a collection of English sentences by providing different inputs.
doc = $load("document").
parse(K) = new english_parser.
parse(K).input = doc.sentence(K).

% The total log-probability of the document, ignoring sentences for which
% no parse was found.
logprob += log(parse(K).goal).
```

(b) An illustration of how to use the above parser. This declarative "script" does not specify the serial or parallel order in which to parse the sentences, whether to retain or discard the parses, etc. All dynabases `parse(K)` share the same grammar, so the rule probabilities do not have to be recomputed for each sentence. A good grammar will obtain a comparatively high `logprob`; thus, the `logprob` measure can be used for evaluation or training. (Alternative measures that consider the *correct* parses, if known, are almost as easy to compute in Dyna.)

```
% Define the unnormalized probability of the grammar production X → Y Z
% as a product of feature weights.
urewrite(X,Y,Z) *= left_child_weight(X,Y).
urewrite(X,Y,Z) *= right_child_weight(X,Z).
urewrite(X,Y,Z) *= sibling_weight(Y,Z).
urewrite(X,Y,Y) *= twin_weight.     % when the two siblings are identical
urewrite(X,Y,Z) *= 1.               % default in case no features are defined

% Normalize into probabilities that can be used in PCFG parsing:
% many productions can rewrite X but their probabilities should sum to 1.
urewrite(X) += urewrite(X,Y,Z)
                 whenever nonterminal(Y), nonterminal(Z).
rewrite(X,Y,Z) = urewrite(X,Y,Z) / urewrite(X).
```

(c) Constructing a dense grammar for use by the above programs, with probabilities given by a conditional log-linear model. With k grammar nonterminals, this scheme specifies k^3 rule probabilities with only $O(k^2)$ feature weights to be learned from limited data [18]. Just as for neural nets, these weights may be trained on observed data. For example, maximum likelihood estimation would try to maximize the resulting `logprob` in 11b.

Fig. 11: A modular implementation of parsing.

nondeterministic input without modification. The only effect on the parser is that `I`, `J`, and `Mid` in Figure 11a now range over states in an automaton instead of positions in a sentence.[35]

At the other end of the parsing process, the parse output can be passed downstream to subsequent modules such as information extraction. Again, it is not necessary to use only the single most likely output (parse tree). The downstream customer can analyze *all* the `phrase` items in the dynabase of Figure 11a to exploit high-probability patterns in the *distribution* over parse trees [176,213].

[35] Instead of `word("spring",5,6)=1`, the parser's `input` dynabase would contain items like `word("spring",state(32),state(38))=0.3`, meaning that when the automaton is in `state(32)`, it has probability `0.3` of emitting the word `"spring"` and transitioning to `state(38)`.

As discussed in the caption for Figure 11c, the training of system parameters can be made to feed back through this processing pipeline of dynabases [65]. Thus, in summary, hypotheses can be propagated forward through a pipeline (joint prediction) and gradients can be propagated backward (joint training). Although this is generally understood in the natural language processing community [81], it is surprisingly rare for papers to actually implement joint prediction or joint training, because of the extra design and engineering effort, particularly when integrating non-trivial modules by different authors. Under Dyna, doing so should be rather straightforward.

Another advantage to integrating the phases of a processing pipeline is that integration can speed up search. The phases can *interactively negotiate* an exact or approximate solution to the joint prediction problem—various techniques include alternating optimization (hill-climbing), Gibbs sampling, coarse-to-fine inference, and dual decomposition. However, these techniques require systematic modifications to the programs that specify each phase, and are currently underused because of the extra implementation effort.

Backtracking Search Many combinatorial search situations require backtracking exploration of a tree or DAG. Some variants include beam search, game-tree analysis, the DPLL algorithm for propositional satisfiability, and branch-and-bound search in settings such as Integer Linear Programming.

It is possible to construct a search tree *declaratively* in Dyna. Since a node in a search tree shares most properties with its children, a powerful approach is to represent each node as a dynabase, and each of its child nodes as a modified extension of that dynabase (see §2.7).

We illustrate this in Figure 12 with an elegant DPLL-style program for solving NP-hard satisfiability problems. Each node of the search tree runs the arc-consistency program of Figure 6 to eliminate some impossible values for some variables, using a message-passing local consistency checker. It "then" probes a variable `nextvar`, by constructing for *each* of its remaining possible values `Val` a child dynabase in which `nextvar` is constrained to have value `Val`. The child dynabase copies the parent, but thanks to the added constraint, the arc-consistency algorithm can pick up where it left off and make even more progress (eliminate even more values). That reduces the number of grandchildren the child needs to probe. The recursion terminates when all variables are constrained.

One good execution strategy for this Dyna program would resemble the actual DPLL method [56], with

- a reasonable variable ordering strategy to select `nextvar`;
- each child dynabase created by a temporary modification of the parent, which is subsequently undone;
- running arc consistency at a node to completion *before* constructing any children, since quickly eliminating values or proving unsatisfiability can rule out the need to examine some or all children;
- skipping a node's remaining children once `consistent` has been proved `false` (by arc consistency) or `true` (by finding a consistent child).

However, the program itself is purely declarative and admits other strategies, such as parallel ones.

A simple modification to the program will allow it to solve MAX-SAT-style problems using branch-and-bound.[36] In this case, a more breadth-first variant such as A^* or iterative deepening will often outperform

[36] The goal is to find a maximum-scoring joint assignment to the variables, subject to the constraints. The score of a given assignment is found by summing the `subscore` values (as specified by the user) of the several `Var:Val` pairs in the assignment.

In Figure 6 and Figure 12, replace `consistent` (a boolean item aggregated by `&=`) by `score` (a real-valued item aggregated by `min=`). In Figure 6, just as `consistent` computes a boolean upper bound on satisfiability, `score` computes a numeric upper bound on the best achievable score:

```
subscore(Var) max= −∞.
subscore(Var) max= subscore(Var:Val) whenever possible(Var:Val).
upper_bound   +=   subscore(Var) whenever is_var(Var).
score         min= upper_bound.
```

Then in Figure 12, `score` is reduced to the best score actually achieved by any child:

```
score            min= best_child_score.
best_child_score max= child(nextvar:Val).score.
```

```
% Freely choose an unassigned variable nextvar, if any exists.
% For each of its values Val that is still possible after arc consistency,
% create a clone of the current dynabase, called child(Val).
nextvar ?= Var whenever unassigned(Var).          % free choice of nextvar
child(Val) = new $self if possible(nextvar:Val).  % create several extensions

% Further constrain each child(Val) via additional extensional input,
% so that it will only permit value Val for nextvar,
% and so that it will choose a new unassigned variable to assign next.
child(Val).possible(nextvar:Val2) &= (Val==Val2)
                                       whenever possible(nextvar:Val).
child(Val).unassigned(nextvar) &= false.  % nextvar has been assigned

% We are satisfiable if Figure 6 has not already proved consistent to be false,
% and also at least one of our children (if we have any) is satisfiable.
consistent &= some_child_consistent.
some_child_consistent |= child(Val).consistent.
         % usually is true or false, but is null at a leaf (since nextvar is null)
```

Fig. 12: Determining the satisfiability of a set of constraints, using backtracking search interleaved with arc consistency. These rules extend the program of Figure 6—which rules out some impossible values for some variables, and which sometimes detects unsatisfiability by proving that `consistent` is `false`. Here, we strengthen `consistent` with additional conjuncts so that it fully checks for satisfiability. Lines 1–2 choose a single variable `nextvar` (using the "free-choice" aggregator `?=`) and guess different values for it in child dynabases. We place constraints into the child at lines 3–4 and read back the result (whether that child is satisfiable) at line 6.

the pure depth-first DPLL strategy. All these strategies can be proved correct from the form of the Dyna program, so a Dyna query engine is free to adopt them.[37]

Local Search and Sampling While the search tree constructed above was exhaustive, a similar approach can be used for heuristic sequential search strategies: greedy local search, stochastic local search, particle filtering, genetic algorithms, beam search, and survey-inspired decimation. Each configuration considered at time `T` can be described by a dynabase that extends a configuration from time `T-1` with some modifications. As with our arc consistency example, rules in the dynabase will automatically compute any *consequences* of these modifications. Thus, they helpfully update any intensional data, including the score of the configuration and the set of available next moves.

The same remarks apply to Monte Carlo *sampling* methods such as Gibbs sampling and Metropolis-Hastings, which are popular for Bayesian learning and inference. Modifications at time `T` are now randomly sampled from a move distribution computed at time `T-1`.[38] Again, the consequences are automatically computed; this updates the move distribution and any aggregate sample statistics.

3.2 Proofs and Proof Forests

It is useful to connect Dyna, whose items have weights or values, to the traditional notion of proofs in unweighted logic programming.

Datalog can be regarded as defining **proof trees**. Figures 13a–13b show a collection of simple inference rules (i.e., a **program**) and two proof trees that can be constructed from them. As a more meaningful

[37] For example, it is easy to see that `upper_bound` at each node n (once it has converged) is indeed an upper bound on the score of the node (so can be used as an admissible heuristic for A^*). It can further be proved that as long as this bound is smaller than the current value of `best_child_score` at an ancestor of n whose `score` was queried, then exploring the children of n further cannot affect the query result.

[38] A simple random walk program is shown later as (24).

example, Figures 14–15 show inference rules for context-free CKY parsing (unweighted versions of the rules in Figure 8) and two proof trees that can be constructed using them.[39] These proof trees are *isomorphic* to the parse trees in Figure 16. In other words, a parser is really trying to prove that the input string can be generated by the grammar. By exploring the proof trees, we can see the useful hidden derivational structures that record how the string could have been generated, i.e., the possible parses.[40]

A Datalog program may specify a great many proof trees, but thanks to shared substructure, the entire collection may be represented as a **packed forest**. The hypergraph in Figure 13c shows the packed forest of *all* proofs licensed by the program in Figure 13a. Some vertices here have multiple incoming hyperedges, indicating that some items can be proved in multiple ways. The number of proofs therefore explodes combinatorially with the in-degree of the vertices.[41] In fact, the forest in Figure 13c, being cyclic, contains *infinitely* many proof trees for b. Even an acylic forest may contain a number of proof trees that is *exponential* in the size of the hypergraph.

Indeed, a Datalog program can be regarded simply as a finite specification of a proof forest. If the rules in the program do not contain variables, then the program is actually isomorphic to the proof forest, with the items corresponding to nodes and the rules corresponding to hyperedges. Rules with variables, however, give rise to infinitely many nodes (not merely infinitely many proofs).

3.3 From Logical Proofs to Generalized Circuits

To get a view of what Dyna is doing, we now augment our proof forests to allow items (vertices) to have values (Figure 13e). This yields what we will call **generalized circuits**. Like an arithmetic (or boolean) circuit, a generalized circuit is a directed graph in which the value at each node α is a specified function of the values at the 0 or more nodes that point to α. Finding a consistent solution to these equations (or enough of one to answer particular value queries) is challenging and not always possible, since Dyna makes it possible to define circuits that are cyclic and/or infinite, including infinite fan-in or fan-out from some nodes. (Arithmetic circuits as traditionally defined must be finite and acyclic.)

We emphasize that our generalized circuits are different from weighted proof forests, which attach weights to the individual proof trees of an item and then combine those to get the item's weight. In particular, the common setup of **semiring-weighted deduction** is a special case of weighted proof forests that is strictly less general than our circuits. In semiring-weighted deduction [89], the weight of each proof tree is a product of weights of the individual rules or facts in the tree. The weight of an item is the sum of the weights of all its proofs. It is required that the chosen product operation $\otimes$ distributes over the chosen sum operation $\oplus$, so that the weights form a **semiring** under these operations. This distributive property is what makes it possible to sum over the exponentially many proofs using a compact generalized circuit like Figure 8 (the inside algorithm) that is isomorphic to the proof forest and computes the weight of all items at once.

Our original prototype of Dyna (§5.1) was in fact limited to semiring-weighted deduction (which is indeed quite useful in parsing and related applications). Each program chose a single semiring $(\oplus, \otimes)$; each rule in the program had to multiply its antecedent values with $\otimes$ and had to aggregate these products using $\oplus$=.

However, notice that most of our useful AI examples in §3.1 actually fall outside this form. They mix several aggregation operators within a program, sometimes including non-commutative aggregators like `:=`, and it is sometimes important that they define the aggregation of 0 items to be null, rather than requiring the aggregator to have an identity element and using that element. They also use additional non-linear operations like division and exponentiation.

As a result, it is not possible to regard each of our AI examples as simply an efficient way to sum over exponentially many proofs of each output item. For example, because of the sigmoid function in Figure 2, the distributive property from semiring-weighted programs like Figure 8 does not apply there. One *cannot* regard

[39] To obtain the CKY proof trees, we must add facts that specify the words and grammar rules. That is, we extend the CKY program with the extensional input.

[40] The mapping from proof trees (**derivation trees**) to syntactic parse trees (**derived trees**) is generally deterministic but is not always as transparent as shown here. For example, a semantics-preserving transformation of the Dyna program [140,69,105] would change the derivation trees but not the derived trees.

[41] Although a has only one incoming edge, it has two proof trees, one in which p is proved from y and the other (shown in Figure 13b) in which p is proved from z.

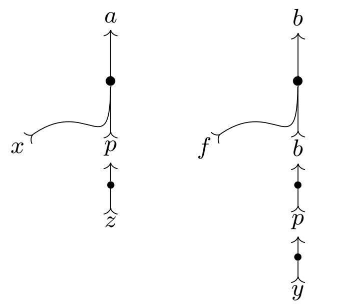
$$\frac{x \quad p}{a} \qquad \frac{p}{b} \qquad \frac{b \quad f}{b}$$

$$\frac{y}{p} \qquad \frac{z}{p} \qquad \frac{}{f}$$

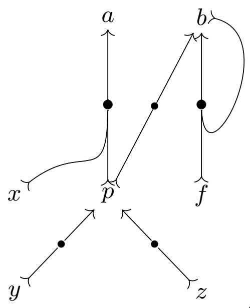
$$\frac{}{x} \qquad \frac{}{y} \qquad \frac{}{z}$$

```
a :- x, p.
b :- p.
b :- b, f.
p :- y.
p :- z.
f.
x.
y.
z.
```

(a) A set of inference rules, and their encoding in Datalog. Axioms are written as inference rules with no antecedents.



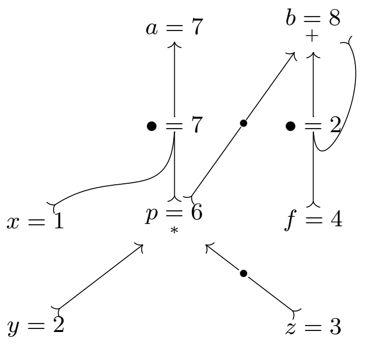
(b) Two proof trees using these rules. When an item is proved by an inference rule from 0 or more antecedent items, its vertex has an incoming hyperedge from its antecedents' vertices. Hyperedges with 0 antecedents (to f, x, y, z) are not drawn.



(c) The proof forest containing all possible proofs. In contrast, each hypergraph in 13b shows only a single proof from this forest, with each copy of an item selecting only a single incoming hyperedge from the forest, and cycles from the forest unrolled to a finite depth.

```
a += x + p.
b += p.
b += b / f.
p *= y.
p *= z.
f = 4.
x = 1.
y = 2.
z = 3.
```

(d) A set of numeric recurrence relations that are analogous to the unweighted inference rule in Figure 13a. We use Dyna's syntax here.



(e) A generalized arithmetic circuit with the same shape as the proof forest in Figure 13c. The weight labellings are consistent with 13d. Each node (including the • nodes) is computed from its predecessors.

Fig. 13: Some examples of proof trees and proof forests, using hypergraphs (equivalently, AND-OR graphs). Named nodes in the graphs represent items, and • nodes represent intermediate expressions.

the activation value of an output node in a neural network as a sum over the values of many individual proofs of that output node.[42] That is, a generalized circuit does not necessarily fall apart into disjoint trees the way that a weighted forest does. Rather, the computations are tangled together. In the neural network example, computing intermediate sums at the hidden nodes is important not only for dynamic programming efficiency (as it is in the semiring-weighted program of Figure 8) but also for correctness. The sigmoid function at each node really does need to apply to the sum, not to each summand individually.

[42] Each proof of o_1 in Figure 1 would be a separate path of length 2, from some input node through some hidden node to o_1.

$$\frac{{}_iw_j \quad X \to w}{{}_iX_j} \qquad \frac{{}_iY_j \quad {}_jZ_k \quad X \to Y\ Z}{{}_iX_k}$$

Fig. 14: The two proof rules necessary to support context-free grammars with unary productions and binary rewrites. w denotes a word from the input sentence and X a symbol of the grammar. Subscripts denote the object's span (which part of the sentence they cover).

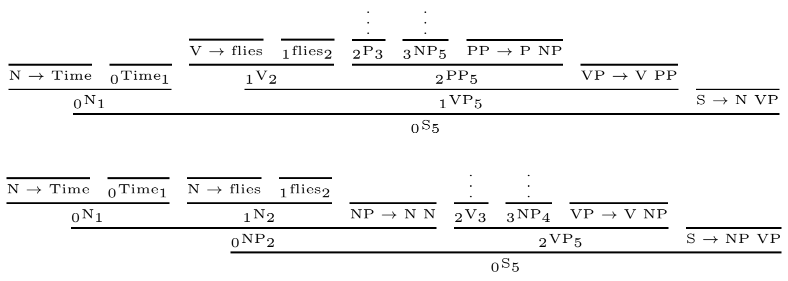


Fig. 15: Two example proofs that "Time flies like an arrow." is an English sentence, using the rules in Figure 14. This is traditional notation, but the hypergraphs of Figure 13 are more flexible because they would be able to show reuse of subgoals within a single proof, as well as making it possible to show packed forests of multiple proofs with shared substructure, as in Figure 13c.

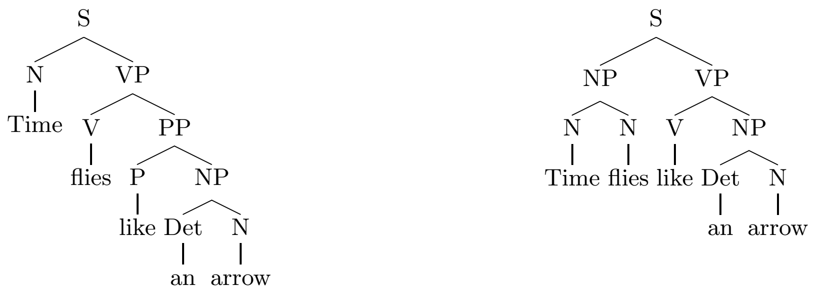


Fig. 16: Two example parse trees of the sentence "Time flies like an arrow" [116]. These are isomorphic to the proofs in Figure 15 (upside down) and correspond to different meanings of the sentence. The first conveys information about how time passes; the second tree says that flies of a certain species ("time flies") are fond of an arrow.

We remark that even generalized circuits are not a convenient representation for all Dyna programs. The rule `f(0) += g(1)` generates a single edge in a generalized circuit. However, the rule `f(start) += g(end)`, where `start` and `end` are evaluated, would generate edges to `f`(x) (for *every* x that is a possible value of `start`) from `start`, `end`, and `g`(y) (for *every* y that is a possible value of `end`). Typically this leads to infinitely many edges, only one of which is actually "active" in a given solution to the program.

Despite all this freedom, Dyna circuits remain circuits, and do not seem to present the difficulties of arbitrary systems of equations. A Dyna program cannot impose fiendish constraints such as $x^3 + y^3 = z^3$. (Recall that Fermat's Last Theorem says that there are no postive integer solutions.) Rather, each equation in a Dyna system constrains a *single* item to equal some function of the items in the program. (This arises from Dyna's use of single-headed rules, similar to Horn clauses.) Furthermore, every item has exactly one "defining constraint" of this sort (obtained by aggregating across multiple rules).[43] So one cannot formulate

[43] As mentioned earlier, this generalizes the completion semantics of [45], which treats a logic program as defining each boolean item with an "if and only if" constraint.

$x^3 + y^3 = z^3$ by writing $u = x^3 + y^3$ and $u = z^3$ (which would give two defining constraints). Nor can one formulate it by writing $s = s + (x^3 + y^3 - z^3)$, a legal Dyna program that might appear to imply $x^3 + y^3 - z^3 = 0$, but whose unique solution is actually that x, y, z, s are all null, since each of x, y, z (having no defining rules) has a defining constraint that it is the aggregation of 0 aggregands.

3.4 The Role of Proofs in AI

We close this section with a broader perspective on why proof trees, forests, and their generalizations should be so useful in AI, as seen in the examples of §3.1.

Abductive AI systems attempt to explain the observed data by reconstructing one or more stories about how the observation was generated. We assume that there is some true (but hidden) derivation ($\boldsymbol{y}$) whose structure, if we knew it, would reveal the true structure of the observed data ($\boldsymbol{x}$), be it a sentence, a scene, etc. Any derivation is a *proof* that the observed data could indeed have been generated somehow. We seek likely derivations because they reveal structure that is informative for additional processing.

In natural language parsing, for example, our model is a probabilistic grammar [24], our observation is a sentence, and we seek to derive a grammatical explanation for the sentence; recall Figures 14–16. In a general system of constraints, we partially observe a set of random variables, and we seek an explanation of the pattern of observed values by fleshing out the other random variables in a consistent way; recall Figures 6–7.

Often, the search for a derivation will yield many possible alternatives, which are then ranked by a **scoring function**, such as the joint probability distribution $p(\boldsymbol{x}, \boldsymbol{y})$.

Deductive AI systems, on the other hand, draw conclusions from observations rather than seeking to explain the observations. This involves proving that these conclusions follow from the evidence. Examples of deductive systems include neural networks, traditional knowledge representation and reasoning (**KRR**), constraint programming, and deductive or probabilistic databases. These systems may have parameters which control how and which inferences are made and how much weight is assigned to them. AI systems for KRR have received attention as information *integration* systems, drawing conclusions from distributed pools of resources, an approach that has not been much investigated elsewhere in AI.

Despite the difference in direction of the causal relationship between inputs and outputs for abductive and deductive reasoning, they are computationally very similar. In statistical AI, abductive systems correspond to **generative models** $p(\boldsymbol{x}, \boldsymbol{y})$. and deductive systems correspond to **discriminative models** $p(\boldsymbol{y} \mid \boldsymbol{x})$. Generative and discriminative models can have the same inference algorithm, with modest changes in how the model parameters are trained.

In both kinds of system, exposing the proof trees (or circuits) is useful because this naturally justifies the system's conclusions and tracks the provenance of data [26,30,31]. For machines, analyzing the set of proof trees facilitates decision making, abductive construction of theories, sensitivity analysis, and statistical learning. Being able to track and reason about the set of proof trees of a given conclusion may also be useful to humans [14], including the researchers who are trying to analyze and improve the AI system, as well as end users who may not fully trust the AI system but would still like it to generate hypotheses and call their attention to pertinent supporting data.

4 Practical AI and Logic Programming

Given an applied AI problem, one would like to experiment with a broad range of models, exact or approximate inference algorithms, decision procedures, training procedures for the model parameters and system heuristics, and storage and execution plans. One must also experiment when developing new general methods.

Dyna supports the common computational core for all this—mechanisms for maintaining a possibly infinite and possibly cyclic network of related items that are named by structured terms. Its job is to store and index an item's value, to query for related items and aggregate their values (including planning of complex queries), to maintain the item's value and propagate changes to related items, and to back-propagate gradient information.

In this section, we expand on our argument from §1 that a fast and scalable implementation of Dyna would be of *practical* use to the AI community.

4.1 What's Wrong with Current AI Practices

Current AI practices, especially in our target area of natural-language processing and machine learning, suffer from a large distance between specification and implementation. Typical specifications are a handful of recurrence relations (though not as short as the examples in this paper). Creative graduate students can easily dream up innovative systems at the specification level. Implementations, however, are typically imperative and by necessity include storage and inference code.

Large Extensional Data Modern statistical methods mine large corpora of data and produce sizable models. It is not atypical to process billions of words and extract models with millions of constants and hundreds of millions of relations between those constants.[44]

Knowledge bases and information integration pose additional problems of scale.[45] As statistical methods gain popularity in other computational fields, the large-data problem spreads.[46] Storage and indexing structures are becoming extremely relevant, as are approximation and streaming techniques.

Large Intensional Effort As we have seen, even when extensional data is small, modern AI systems often have large computations over intermediate quantities. For many algorithms, the (weighted) proof forests may be exponentially or unboundedly large. Here, efficient inference algorithms, prioritization, and query planning become critical for managing execution time.

Typical modern AI systems have, with large investments of time (Table 1), painfully hand-tuned their inference engines for the particular cases at hand. Insights and optimizations at the runtime level are typically viewed as engineering details secondary to the AI work itself; it is quite standard to elide such "details" from publications, even though they may make up the majority of the work itself! In some cases, even the code is not released or becomes unavailable. Together, the net effect is that, despite the proliferation of systems, moving outside the bounds of a system's preconceived use cases entails working around or writing an entirely new system without any real documentation about what has or has not worked in the past, only an incomplete library of code examples.

This problem is not unique to statistical AI. Bioinformaticians have proposed a declarative approach to specifying regular and context-free sequence analyzers [87], because procedural bioinformatics programs can be hard to work with:

> Because they are written in terms of recurrences that reflect not only the logic of the problem, but also other kinds of concerns: Representation of data, order of computation, time and space efficiency issues. These things one should not have to worry about when trying to understand a problem. [4]

[44] The WaCKy corpora [13] are web-derived text collections of 1–2 billion words for English, German, and Italian. In 2006, Google released a list of common up-to-5-word sequences appearing in over a *trillion* words of English text; the resulting data set had 13,588,391 constants and 3,795,790,711 relations of various arities among them [83]. Statistical machine translation systems are often based on induced (learned) models of linguistic structure; a current system for English-Urdu grammar extraction produced a grammar with 42,235,002 rules [22], and Arabic-English systems are an order of magnitude larger (Kevin Knight, p.c.). As of early 2010, the Linguistic Data Consortium [124] offers 650 gigabytes of speech, 244 gigabytes of text, 14 gigabytes of video, and 6 gigabytes of lexicographic data. Large-corpus techniques are entering adjacent fields, including machine vision; the LabelMe project as of 2006 had 111,490 labeled polygons in their dataset of images and videos [173].

[45] The DBPedia [21] knowledge base, as of release 3.5.1, includes more than 3.4 million objects, 1.5 million of which participate in their ontology. The ResearchCyc release of Cyc [121,120] claims more than half a million concepts in their ontology, with five million assertions across 26,000 relationships. The aggregate Open Data Movement [195] claims that the collective dataset is 13.1 billion RDF triples with 142 million RDF links. Open Mind Common Sense [180,125] project has over a million simple English natural language statements in its database.

[46] For example, the NCBI GenBank [17] is a curated database of all publicly available DNA sequences; at last count, it measured 254,698,274,519 base-pairs (across 156,874,659 sequence records) in total. Operational logs for large-scale systems are a rich source of data for abductive data mining [99]: AT&T accumulates logs of all calls carried across its network ([99] estimates the rate at roughly 300 million calls per day for roughly 100 million customers, in 2000); search engine companies are secretive about their rates of traffic, but one estimate [50] suggests that Google saw 87 billion queries in December 2009.

Package	**Files**	**SLOC**	**Language**	**Application area**
SRILM	285	48967	C++	Language modeling
Charniak parser	266	42464	C++	Parsing
Stanford parser	417	134824	Java	Parsing
cdec	178	21265	C++	Machine translation
Joshua	486	68160	Java	Machine translation
MOSES	351	37703	C++	Machine translation
GIZA++	122	15958	C++	Bilingual alignment
OpenFST	157	20135	C++	Weighted FSAs & FSTs
NLTK	200	46256	Python	NLP education
HTK	111	81596	C	Speech recognition
MALLET	620	77155	Java	Conditional Random Fields
GRMM	90	12926	Java	Graphical model add-on
Factorie	164	12139	Scala	Graphical models

Table 1: A selection of popular NLP and machine learning systems (top) and toolkits (bottom) showing number of source files, source lines of code (SLOC), primary implementation language, and functionality. Statistics were generated from the most recent stable release as of this writing using SLOCCount [206].

In short, modern AI academic research systems consist of large bodies of imperative code (20,000–100,000 lines), specialized for the purpose at hand. Regardless of programmer intent, there is little cross-system code reuse. Some researchers have aimed to develop reusable code libraries (known as toolkits) to support common development patterns. However, even the best and most flexible of these toolkits are themselves large, and invariably are not general enough for all purposes.[47] Table 1 shows a selection of released AI systems and toolkits and their code-base sizes.

Uncaught Bugs The size of these coding efforts is not only a barrier to entry, to learning, and to progress, but also likely affects correctness. The potential for uncaught bugs was recognized early in statistical AI. Statistical AI systems have many moving parts, and tend to produce some kind of quantitative result that is used to evaluate the method. The results are not expected to be perfect, since the problems are inherently hard and the statistical models usually cannot achieve human-level performance even at their best. This makes it very difficult to detect errors. Methods that appear to be producing "reasonable" results sometimes turn out to work even better (and occasionally worse) when bugs in the implementation are later noticed and fixed.

Diverse Data Resources The AI community is distributed over many geographic locations, and many AI researchers produce data for others to share. The difficulty in using this vast sea of resources is that they tend to be provided in idiosyncratic formats. Trying out a new dataset often requires understanding a new encoding scheme, parsing a new file format, and building one's own data structures for random access.

Cyc [121,120] is an attempt to capture an ontology of the world from human editors. DBPedia [21] is a large-scale, community effort to extract and redistribute *structured* information from Wikipedia. LabelMe [173] is an attempt to collect manually annotated bounding polygons and labels of objects from real-world images.

There exists a small industry of building natural language databases specifically for AI research, with an biannual international Conference on Linguistic Resources and Evaluation (LREC). The Linguistic Data Consortium [124] serves as a distribution center for much of this work and has collected or produced over 470 artifacts (nearing a terabyte of total data [102]) since its creation in 1992. Curated resources for natural language processing range from unannotated but parallel corpora (e.g., Europarl [118]), compressed

[47] For example, the OpenFST toolkit [7] for weighted finite-state automata is beautifully designed and implemented by the leading researchers in the area and has a great many uses. Even so, it happens not to support various patterns that have come up from time to time in our own research on finite-state methods: infinite alphabets, intensionally specified topologies, efficient handling of default arcs, mixed storage disciplines, parameter training, machines with 3 or more tapes, parallel and A^* algorithms, and non-finite-state extensions.

representations of the entire (English) World-Wide Web (e.g., Google's N-gram collection [83]), manually-annotated treebanks of text and/or speech (perhaps the most famous being the Penn Treebank Project [133]), to lexicographic databases (e.g., Celex [12], WordNet [139,77], its generalization EuroWordNet [203], and the successor Global WordNet Association [88]).

Many researchers also produce (generally small) annotated datasets for their own use, and release them on request to facilitate comparative work.[48] Researchers also share the parameters of their statistical models, often trained from data at huge CPU cost.

Diverse Code Resources Many AI resources are in the form of code rather than data. It can be very valuable to build on the systems of others, and there are principled ways to do so. At present, however, software engineering considerations strongly discourage any deep integration of systems that were built in different labs. Using code resources can require wrestling with portability issues and unfamiliar APIs. Furthermore, the APIs are usually insufficiently powerful for a user to query or influence the *internal* hypotheses and features of someone else's system.

Why would that be desirable? §3.1 already discussed processing pipelines, where tools earlier in the pipeline (such as a speech recognizer or parser) analyze or transform the input in a way that can greatly aid later tools (such as an information extraction system [25]). Ideally, one would like the systems in the pipeline to work together to agree on a common high-quality output and common parameters. However, this generally requires the ability for systems to query one another or pass messages to one another [81].

It is also valuable to integrate several AI systems that are attempting the same task, since they may have different strengths. One method is to run the systems separately and combine their outputs in some heuristic way [111]. However, again, deeper negotiation among the systems is desirable. If the different systems are willing to score one another's hypotheses, their preferences can be combined in various ways—such as a mixture of experts, where different models are weighted differently on different parts of the input space [103], or a product of experts, which attempts to find an output that is scored highly by all models [101]. A recently emerging theme, therefore, is the development of principled methods for coordinating the work of multiple combinatorial algorithms [63,185,61,172].

Ad Hoc Experimental Management AI researchers spend considerable time managing computational experiments. It is usual to compare multiple systems, compare variants of a system, tune system parameters, graph performance across different types and amounts of data, and so forth. Common practice is to run programs at the Unix command line and to store results in files, perhaps writing scripts to manage the process. Sometimes one keeps intermediate results in files for reuse or manual analysis. It can be difficult to keep all the files organized, up to date, and track their provenance [28].

Breck [28] describes the problem, explains why it is not easily handled with makefiles, and presents a new tool for the purpose: "zymake is a high-level language for running complex sets of experiments. The user writes a zymakefile, mostly consisting of parameterized shell commands, and zymake determines the dependency structure and executes the commands in the appropriate order."

zymake systematically names the output files that result from different experimental settings, but does not otherwise manage the output data. For power users, the R language for statistical computing [164] can be an excellent way to flexibly store and explore structured experimental results. However, R focuses on analyzing extensional data. One must still write other code to create those data—i.e., to invoke the experiments and convert the results to R format.

4.2 Declarative Programming to the Rescue

The above problems are intensifying as AI research grows in size, scope, and sophistication. They have motivated our attempt to design a unified declarative solution that hides some of the complexity. We would like it to be easy again to simply try out good ideas!

Promising declarative languages based on Datalog have recently been built for domains such as sensor networks [43,51,129,130] and business data analytics [127,128].

[48] Amazon Mechanical Turk [8] has been used to cheaply produce very experiment-specific corpora (see the recent workshop [34]).

Why does a declarative approach fit for AI as well? We believe the business of AI is deriving hypotheses and conclusions from data (§3.4). These are fundamentally declarative problems: *what* to conclude can be specified without any commitment to *how* to conclude it, e.g., the order of computation. The Dyna approach has something to contribute toward solving each of the challenges of the previous section:

Large Extensional Data We expect that most access by AI programs to large extensional data stores could be supported by traditional on-disk database technology, such as B-trees, index structures, and standard query planning methods. AI programs can automatically exploit this technology if they are written in a Datalog-derived language with an appropriate implementation.

Large Intensional Effort The computational load of AI programs such as those in §3.1 consists mainly of database queries and updates. Dyna provides an executable language for specifying these algorithms, making them concise enough to publish within a paper.

Our hope is that the details left unspecified in these concise programs—the storage and inference policies—can be efficiently handled in a modular, reusable way across problems, eventually with automatic optimization and performance tuning. Even basic strategies like those in §2.6 sometimes correspond closely to current practice, and are often asymptotically optimal [136]. We are deeply interested in systematizing existing tricks of the trade and making them reusable across problems,[49] as well as pushing in new directions (§5).

Quality Control Smaller programs should have fewer bugs. We also expect that Dyna will allow some attractive paradigms for inspecting and debugging what a system is doing (§4.5).

Diverse Data Resources We hope that dynabases can provide a kind of natural interchange format for data resources. They allow flexible representation of typed, structured data (see §4.4), and Dyna offers an attractive query language that can be integrated directly into arbitrary computations. It is conceptually straightforward to convert existing data resources into collections of Dyna facts that can be stored and queried as in Datalog.[50]

Ultimately, one could support remote connection to dynabases via the web (§4.7), with Uniform Resource Identifiers and a standard approach to versioning. The Semantic Web project would like to connect diverse resources across the web into one large, interlinked ontology. We agree that there is a vast pool of data "out there" to draw upon, and believe that Datalog and its extensions are a strong approach for querying it [122,202].

LogicBlox [127] has already demonstrated that Datalog makes an excellent language for information warehousing, integration, and management, at scale within enterprises. We hope to follow in their footsteps.

Diverse Code Resources Dynabases are a useful format for code resources as well. We do not claim that wrapping Java code (for example) in a dynabase interface (§4.7) will improve its API. However, computational resources that are natively written in the Dyna language do have advantages as components of larger AI systems. First, they can more easily expose their internal hypotheses to be flexibly queried and influenced by another component. Second, query optimization can take place across the dynabase boundary, as can automatic differentiation. Third, we suspect that Dyna programs are simply easier for third parties to understand and modify manually when necessary. They can also be manipulated and combined by program transformation; for example, [46] shows how to combine two Dyna programs into a product-of-experts model.

[49] E.g., an alternative to using a software library such as OpenFST (footnote 47) would be to define the same finite-state computations in Dyna, which can capture them concisely, and which as a programming language is more expressive. The challenge is to abstract out the implementation tricks of OpenFST and incorporate them into the Dyna execution engine, in a general way, so that they will automatically apply to the resulting finite-state Dyna programs (not to mention other programs).

[50] In fact, we are working on flexible syntactic sugar that may make it possible to directly interpret some existing file formats as Dyna programs.

Ad Hoc Experimental Management Dyna suggests an elegant solution to running collections of experiments. Figure 11b gives a hint of how one could create a parametric family of dynabases that vary input data, training data, experimental parameters, and even the models and algorithms. The dynabases are named by structured terms. Each dynabase holds the results of some experiment, including all intermediate computations, and can track the provenance of all computations (by making the hyperedges of proof forests visible as items). Some computations would be automatically shared across related dynabases.

Using dynabases to store experimental results is quite flexible, since dynabases can be structured and nested, and since the Dyna language can be used to query, aggregate, analyze, and otherwise explore their contents.

In principle, this collection of dynabases may be infinite, representing an infinite variety of parameter settings. However, the contents of a dynabase would be materialized only when queried. Which materialized intermediate and final results are stored for later use, versus being discarded and recreated on demand, would depend on the dynabase's chaining and memoization policies (see §5.2),as declared by the user or chosen by the system to balance storage, latency, and total runtime.

4.3 Uses of Change Propagation in AI

Recall that dynabases implement *dynamic algorithms*: their intensional items update automatically in response to changes in their extensional input. This corresponds to "view maintenance" in databases [95,96], and to "self-adjusting computation" [1,3] in functional languages.

We observe that this kind of **change propagation** is widely useful in AI algorithms. Internally, many algorithms simply propagate changes until convergence (see the discussion of message passing in §3.1). In addition, AI systems frequently experiment with slight variants of their parameters or inputs for training, validation, or search.

Optimization of Continuous or Discrete Parameters Training a data-driven system typically runs the system on a fixed set of training examples. It explores different parameter settings in order to maximize an objective measure of system performance. A change to an individual parameter may affect relatively few of the training examples. Similarly, adding or removing parameters ("feature selection") may require only incremental changes to feature extractors, automata, or grammars. The ability to quickly recompute the objective function in response to such small changes can significantly speed up training [151].

k-Fold Cross Validation The dual situation occurs when the parameters are held fixed and the training data are varied. Systems often use *cross-validation* to tune some high-level parameters of a model. For example, a language model is a probability distribution over the strings of a language, and is usually trained on as much data as possible. "Smoothing parameters" that affect how much probability mass is reserved for events that have not been seen in the training data (cf. Figure 4). To evaluate a particular choice of smoothing parameters, cross-validation partitions the available training data into k "folds," and evaluates the method's performance on *each* fold when the language model is trained on the other $k-1$ folds. This requires training k different language models. However, it should not be necessary to build each model from scratch. Rather, one can train a master model on the full dataset, and then create variants by removing each fold in turn. This removal should not require recomputing all counts and probabilities of the model, particularly when k is large. For example, "leave-one-out" training takes each sentence to be a separate fold.

Search and Sampling §3.1 already described how change propagation was useful in backtracking search, local search, and sampling. In all of these cases, some tiny change is made to the configuration of the system, and all the consequences must be computed. For example, in the DPLL backtracking search of Figure 12, constraining a single additional variable may have either small or large effects on reducing the possibilities for other variables, thanks to the arc consistency rules.

Control and Streaming-Data Systems Systems that process real-world data have obvious reasons for their inputs to change: time passes and more data is fed in. Monitoring the results is why commercial database engines such as Oracle have begun to support continuous queries, where the caller is continually notified of any changes to the query result. The Dyna version of continuous queries is discussed in §4.6 below.

Applications include business intelligence (e.g., LogicBlox [127]); stream processing for algorithmic equities trading (e.g., DBToaster [5]); user interfaces (e.g., Dynasty [72] and Fruit [53]); declarative animation (e.g., Fran [76]); query planners and optimizers (see §5.3); and even (incremental) compilers [33].

In an AI system—for example, medical decision support—sensors may continously gather information from the world, users may state new facts or needs, and information integration may keep track of many large, evolving datasets at other locations. We would like a system to absorb such changes and draw conclusions about the state of the world. Furthermore, it should draw conclusions about desirable actions—actions such as notifying a human user of significant changes, controlling physical actuators, seeking more information, or carrying out more intensive computation. A running process can monitor these recommended actions and carry them out.

4.4 Modularity

Part of the vision of §4.2 is that Dyna should make it possible to manage data and computational resources from many quarters—all expressed as dynabases. In this section and the next, we give some additional details of how multiple dynabases interact, following on §2.7.

Although Dyna has a pure declarative semantics, dynabases resemble classes in object-oriented languages. Items correspond to member functions, and extension of dynabases corresponds to class inheritance.

More specifically, Dyna is a prototype-based language [123] like Self [199] or JavaScript [64], in which there is no distinction between objects and classes. Objects inherit directly from other objects. In Dyna, this is accomplished with the `new` operator.

Each new dynabase is fully specified at creation time, similar to classes in [142]. It cannot be modified, but its extensions can be. As with Newspeak classes [27], each dynabase has its own view of the world: there is no implicit top-level scope over the system and there is no static state. All imports and information passing are explicit.

Like objects, dynabases can support information hiding by declaring their type. The type of a dynabase δ specifies which of its items are **output items** and which are **input items** (some items may be neither or both). Other user-defined dynabases may only query δ's *output* items. The transitive owners of δ (see footnote 23) may only provide new aggregands to δ's *input* items.[51]

Furthermore, the dynabase's type may guarantee particular value types for the output items, and may require particular aggregand types and aggregation operators for the input items.

Type declarations sometimes enable significant optimizations. For example, input-only items do not have to be stored, as long as they pass updates along to their consequents. Thus, a dynabase that computes aggregate statistics on streaming data does not have to store the data stream.

4.5 Debugging Declarative Programs

One may wonder how to debug a declarative program, without printing a confusing trace of its execution. Statistical AI programs can easily have uncaught bugs (§4.1), and it is important to understand where the results are coming from.

Fortunately, declarative programs that abstract away from the order of computation should make debugging easier, not harder. It is possible to query a dynabase to find out any of the intermediate results. If those results are not currently memoized by the dynabase, they will be recomputed on demand using Dyna rules that define them.

This makes it possible to trace the **provenance** of any suspicious computation.[52] Dynasty [72] is a tool we have built for browsing through potentially infinite hypergraphs (see §3.2), using animated graph layout.

[51] In the execution engine, however, dynabases must be able to communicate through a wider interface (§4.6). For example, in (17), a backward-chaining execution strategy for γ`.piglets` would require it to query δ`.offspring`, which may be private in δ.

[52] Though non-stratified programs may end up having *only* cyclic proofs at convergence. To truly answer "Why?" in such circumstances requires additional logging of the execution of the system, e.g., tracing the provenance of *timestamped* items.

Other alternatives would an outline-like GUI, a command-line query interface, a natural language interface [14].

Suspicious computations themselves can be identified by writing Dyna rules that detect whether assertions fail,[53] or other worrisome conditions hold, anywhere in the dynabase. Such rules may aggregate with `|=` to detect any instance of the condition, or `max=` to find the worst instance. Tracing back from such detections will identify the particular items that triggered them.

Dyna is additionally amenable to "what if?" debugging. A user may, without impacting the original computation, extend the dynabase and experimentally modify some items (in the extension) in order to observe how other items would respond.

Figuring out why a declarative program is slow may be harder than figuring out why it gives the answer that it does, since the execution strategies are ordinarily hidden from the user. Fortunately, we plan to expose some internal activity through "voodoo items" in the dynabase (see §5.3), so it is possible to inspect this activity by aggregating voodoo items to form a runtime profile of program activity, by looking at the query plans that are internally produced, and by tracing queries and updates that match user-specified conditions. (Such traces could be animated: the user would augment the Dyna program by defining new items that specify some visual rendering of the scene in terms of the existing items and voodoo items. A visualization process would monitor the dynabase's updates to these visual items, and update its rendering accordingly as the computation evolves.)

4.6 Modes: Interacting with the Procedural World

As a pure declarative language, Dyna has no I/O operations. Hence, all interaction with the world must take place via **processes** that build or acquire dynabases and interact with them.[54] We sketch the design in this section.

A Dyna program's rules only specify some dynabase δ of a particular type, a mathematical object that semantically defines values for various items.

However, a Dyna compiler or runtime environment offers a well-typed procedural API to δ. The allowed signature of this API is known as δ's **mode**.[55] Where a dynabase's type defines what kinds of items it exposes in principle, a dynabase's mode defines how it can interact in practice with user processes in the procedural world:

– *Query modes:* What kinds of queries will δ answer about its items?

– *Update modes:* What kinds of update messages can pass through δ's items?

The dynabase's full API can be used internally. The *public* methods of the API are the means by which running user processes interact with it.

From a declarative point of view, the mode system is regrettable: one would prefer to simply allow any query or update that is consistent with δ's type. One might think that arbitrary queries and updates could be handled at runtime using an interpreter or a just-in-time compiler. But a mode system still improves efficiency, because the system can avoid overhead if it can be sure that certain queries and updates will *never* occur. The more important reason for a mode system is that some queries and updates are simply not supportable—or not supportable by a given implementation of Dyna—because it is not possible to find an execution plan for them.[56] By having each dynabase declare particular modes that *are* supported (and keep information about their efficiency), we make it possible to do mode checking and query planning at a global

[53] Rules of this sort can also be used for integrity constraints. That is, assertions can fail because of "bugs" in the extensional data, rather than in the program.

[54] Rather like the handling of I/O in [78].

[55] Here we are using the term "mode" collectively. Just as the type of a dynabase is a signature that specifies how all of its items are typed, the mode of a dynabase specifies how of its queries and updates are moded.

[56] One cause of this problem is that foreign dynabases (see §4.7 below) tend to offer only limited modes—since other modes might be difficult to implement, or might risk producing infinitely many results (see footnote 5). These foreign dynabases are beyond Dyna's control, so execution plans cannot possibly access them in ways that they do not support.

level: a query or update mode can be (efficiently) supported if it has an execution plan in terms of other (efficiently) supported modes.

Every Dyna program declares a mode as well as a type, although as far as possible these are inferred automatically. Since this paper is not about execution, we only give a rough sketch of the mode system, and do not discuss explicit declaration syntax.

Query and Update Modes A query mode describes both the form of the query and the form of the response to it, which correspond respectively to the argument and return types of a query method in the API. For example, the dynabase (20) might have a mode to support ground queries such as `fib(99)`, but it might not have a mode for the free query `fib(N)`, which is a request to find *all* Fibonacci numbers `fib(0)=1`, `fib(1)=1`, `fib(2)=2`, etc.

Some query modes may support **continuous** queries [126]. The continuous query API allows the querying process to register a **listener** that will eventually be notified of any future updates to the result of the query, for example as a result of updates to the extensional data [5]. Continuous monitoring is necessary to support the reactive applications discussed in §4.3, such as user interfaces, robotics, stream processing, and medical decision support.

Finally, each query mode also specifies whether its queries are **complete** or **incomplete** queries. A complete query should give results consistent with the fixpoint semantics of the dynabase (see footnote 14 and the appendix (§A)). An incomplete query [126] may return a faster, provisional result. While this result may be incorrect, incomplete queries are usually continuous—so that the querying process will listen for updates to this provisional result, which eventually approaches the correct value as forward-chaining computation proceeds.

Incomplete queries play an important role in the internal inference algorithms (see §5.2). Some external applications may prefer (continuous) incomplete queries because early information can be useful: for example, declarative animation or debugging may wish to track the intermediate states of the system as inference proceeds. Finally, some query modes may only be supported via incomplete queries. The dynabase (20) can more easily support `fib(N)` as a continuous incomplete query, initially returning all Fibonacci numbers that have already been computed, and gradually generating updates to this set as new Fibonacci numbers are discovered through forward chaining.

What about update modes? An update mode corresponds to an API method that can be called to apply a specific kind of update to an item or a set of items. In a typical forward chaining approach (§2.6), the update may be *pushed* onto the dynabase's agenda for later processing. The update mode also determines the signature of the listener methods that can register with a continuous query. These listeners will be called with the update when it is *popped* from the agenda.

Public API The *public* methods of the API are restricted by δ's type. They may be called by processes that use δ. The public part of the API is restricted to permit only

- those queries that examine the output items of δ;
- those update listeners that monitor the output items of δ;
- those update application methods that increment δ's input items with new extensional aggregands, using those items' proper aggregation operators.[57]

Full API From now on, we focus on the mode's full API, which may be called internally by a Dyna runtime engine and perhaps also by a privileged debugger. While it is less restrictive than the public API, it still may not support all conceivable query modes and update modes.

[57] By providing such aggregands, a process acts as if it were a transitive owner of δ (footnote 23), incrementally specifying new rules that place new aggregands in δ. Thus, a process can only call these update methods if it holds a *capability* that transitively owns δ. If not, it can create an extension of δ, which simultaneously creates a capability that owns the extension. The process may use this capability to modify the extension, and share the capability with other processes.

As an example, δ's mode may support the query `f(3,4,5)`, which looks up the value of that term, but not support `f(3,Y,Y)`, which looks up the value of `f(3,Y,Y)` for all ground terms `Y` such that `f(3,Y,Y)` is not null.

The full API's query methods for *output* items will be available in the public API. So will its update methods for *input* items, when they use the proper aggregators.

Mode Checking If δ has a query mode that handles the query `f(3,Y,Y)`, then Dyna must be able to construct a query plan for that query. If this plan needs to recursively query the antecedents of the rules defining `f(3,Y,Y)`, then it must do so using their query modes. If no such plan cannot be found (this may be checked at compile time), then we have a mode failure: the Dyna program defining δ should not have declared that it can support this query mode.

Similarly, if δ declares some update mode for `f(...)`, then to avoid a mode failure, Dyna must be able to construct an appropriate update plan. This plan may require updating consequents of `f(...)` after querying their other antecedents (i.e., the "passenger queries" of §2.6). These propagated updates and passenger queries must similarly be licensed by declared update and query modes.

A program may pass type checking (so it specifies a mathematically well-defined dynabase of the declared type) and yet fail mode checking (meaning that the dynabase's declared mode cannot be realized in the procedural world). For the program to pass mode checking, the Dyna implementation must be able to come up with plans (possibly inefficient ones) that fully implement its declared query and update modes. We will briefly discuss such plans in §5.2.

Static checking of query modes is a key design feature of the Mercury pure logic programming language [138,154]. However, Mercury does not support forward chaining, so there are no update modes.

Form of Query Results *Non-ground* queries, whether issued by a user process or by the execution engine, may in general have complex answers. For example, the query `f(3,Y,Z)` might return the tiny program

```
f(3,0,1) := " ".
f(3,Y,Y) := "foo".   % a default, used for Y ∉ {5,8}
f(3,5,5) := "bar".   % later := aggregands override earlier ones
f(3,8,8) := $null.   % special aggregand says f(3,8,8) has no value
```
(21)

which specifies a string value (or explicit lack of value) for each of infinitely many items that match the query.[58]

In general, the *result* of a non-ground query against δ is a partial function (from provable items that match the query to their values), so it must in general be returned as a list of rules—i.e., a Dyna program. But how complicated are these programs allowed to get? After all, would a caller know what to do with the infinite result (21)? Would it be able to handle an even more complex program, such as (22)?

```
f(3,Y,Z) += Y*2 if h(Y) < Z.   % non-constant rule body
g(3,Y,Y) += 1.                 % summation across rules
h(Y) = ...                     % auxiliary definition
```
(22)

Indeed, what stops Dyna from simply returning the original defining rules of δ as the answer to any query against δ, leaving the caller to do all the work?

The general answer is that each query mode is a well-typed method call that not only specifies the form of the queries that it will handle, *but also guarantees the form of its result* (as in Mercury [154]). For example, a particular query mode might promise to accept any query of the form `f(`x`,`y`,`z`)` where x is an integer constant, and return a finite set of rules of the form `f(`x`,`y`,`z'`) = `v, where y has *not* been specialized and z' and v are integer constants. Furthermore, it might promise to return this set of rules as an array of (z', v) pairs sorted by z'. A query plan or a user process may wish to invoke this query mode if it is prepared to process such a result.

[58] The constant `$null` in this example is an ordinary value (not null), but has special meaning to the `:=` aggregator. If the last aggregand of `:=` is `$null`, then the result of aggregation is null, as if there were no aggregands. Beyond allowing (21), this convention enables a dynabase's owner to *retract* any `:=` item by supplying a `$null` aggregand to it (just as it can *change* that item's value to x by supplying a new x aggregand).

Form of Updates Like a query result, an update may be complex in the general case. Roughly speaking, it too is a program—a list of Dyna rules that supply new aggregands to items. Why? First, non-ground update rules such as `a(X) += 1` and perhaps even `a(X) += X*2` are necessary for execution strategies that forward-chain non-range-restricted rules. Second, updates consisting of multiple rules are sometimes needed to make explicit the required order of multiple updates to the *same* item.[59] Finally, a large update consisting of rules that affect many *different* items can improve efficiency by allowing set-at-a-time update processing.

A complication is that the updates to an item do not in general respect the aggregation operator of the item. For example, one might expect that the `cost_to` items in (10) can only decrease over time, since additional aggregands would be provided via `min=`. This is indeed true for a constant graph [58]. However, if we wish to declare update modes for the `edge_cost` input items that allow them to be increased or retracted (changed to null), then it is possible for `cost_to` items to increase or retract in response, so the dynabase needs to declare these update modes as well.

Even an apparently innocuous program may have multiple update modes. For example, the following program uses only a single aggregator:

```
a    += b(X)*X.
a    += sqrt(c).     % square root, which is the culprit here
b(X) += ...
c    += ...
```
(23)

Let us also suppose that `b(X)` and `c` only have `+=` update modes. Increments to `b(X)` items can be multiplied by `X` and passed on as `+=` updates to `a`, thanks to the distributive property. However, increments to `c` will have to yield `:=` updates to the aggregands of `a` and ultimately to `a` itself.[60]

Flow Control A process that is using a dynabase δ is not required to listen to the updates that δ produces. It can choose whether or not to register listeners for those updates.

However, another dynabase φ that is defined in terms of δ may have to monitor δ's updates. We suspect that in some circumstances, this is too expensive or is actively undesirable. We therefore expect to provide some "flow control" mechanisms that define snapshots of φ in terms of snapshots of δ.

In one setting, δ is constantly changing and the computations in φ cannot keep up. Queries against φ (other than incomplete queries) may never be able to return. Even if they do return, two queries to φ may return inconsistent results because the world has changed in between. Thus, a process using φ may want to *temporarily* stall updates to φ from δ, to temporarily obtain a consistent picture of δ's world and the φ world that results from it (similar to "single-answer semantics" [212]).

In another setting, φ contains model parameters or summary statistics that have been derived from a large training dataset δ. These intensional data were computed at considerable cost. One would like to *permanently* detach a standalone version of φ that can be regarded as a stable resource in its own right. It should no longer depend on δ, even if δ's mode says that δ may broadcast updates. Otherwise, any update

[59] Order is significant when the aggregator is `:=`, or when the updates use different aggregators (see next paragraph) that do not commute with each other. Often these rules can be consolidated, simplifying the update program before it is applied.

[60] The reader may wonder we do not try to convert all updates to `+=` updates. It is true that a `:= 7` operator to be applied to `5` could be converted to a `+= 2` update, at the cost of a subtraction, but this conversion is only valid if the update is applied immediately (before `5` changes), and is only possible because `+` happens to have an inverse operator (whereas `min` does not).

Conversely, the reader may wonder why not we do not just use `:=` or "delete and rederive" [96] updates everywhere, as is common [3]. The answer is that `+=` updates can be more efficient in some situations. First, since they commute with each other, we are free to apply them in any order via the agenda. Second, updates of the form `+=` *sparse vector* are more efficient than the equivalent `:=` *dense vector*. Third, `+=` updates defer work (computing the sum) that may turn out to be unnecessary, and they can be usefully prioritized by update size. Fourth, if `b(X)` values do not support any queries (e.g., they are input-only items that aggregate streaming data) and only support the update mode `+=`, then an implementation can entirely avoid storing the many `b(X)` values, simply passing updates through to `a`. This is not possible if we need to support `:=` updates to `b(X)` items—such update strategies need access to old values of `b(X)` items (but storing all the streaming data is impractical).

from δ could cause considerable recomputation in φ. Moreover, it would not be possible to send φ overseas without keeping a remote connection that listens for updates to δ. Note that for φ to be detached, it must compute and store all items that depend on δ.

4.7 Foreign Dynabases

So far, we have assumed that a dynabase is defined by specifying a Dyna program: that is, a dynabase literal that contains declarative rules with a particular semantics. These are **native dynabases**.

However, a procedural dynabase object could also be implemented in other ways, as long as it still declares an API as in §4.6.[61] We refer to this as a **foreign dynabase**. Note that a foreign dynabase may be constant, or it may declare update modes that allow it to broadcast and/or receive changes.

All dynabases—native and foreign—can communicate about data because queries, results, and updates are all represented as terms, which exist in a single universe shared by all dynabases.

Typically, a Dyna program would make use of a foreign dynabase to serve its needs. For example, foreign dynabases can provide Dyna programs with access to data resources, by wrapping existing static files or dynamic services in a dynabase interface.

Foreign dynabases can also provide access to computational resources such as specialized numerical solvers. The access pattern is as in Figure 11b, with the role of the parser played by a foreign dynabase rather than Figure 11a. For example, a Dyna program could give a matrix or a set of constraints to the input items of a foreign solver dynabase, and then write further rules that make use of the foreign solver's resulting output items. Furthermore, some foreign solvers may be able to accept incremental updates to the input items, and use dynamic algorithms to rapidly determine find the resulting updates to the output items. Such dynamic algorithms exist for many matrix and graph computations, and can be packaged up using update modes in a dynabase API.

How does a Dyna program get access to a foreign dynabase? Recall that `$load("filename")` evaluates to an existing dynabase that is backed by a disk file. A similar expression might evaluate to a foreign dynabase whose API is handled by a web service, a persistent daemon, a new object created within the current process, or a new process that is spawned as a side effect of evaluation.

Even simple arithmetic in Dyna is ultimately handled by a foreign dynabase. When δ evaluates the item `20+100` (as in (10)), of course it uses procedural code to do that addition. This procedural code is ultimately accessed by querying a standard `$builtin` dynabase—not written in Dyna—in which the item `20+100` has value `120`.[62] To enable a wider range of Dyna programs and execution strategies to survive mode checking (§4.6), the `$builtin` dynabase should also support certain non-ground query modes for relations like `+`.[63]

[61] We have discussed the interface only abstractly. The actual programming interface might use any convenient message-passing API that allows two dynabase objects to communicate (within a process, between processes, or over the web), transmitting the terms in a message via pointers into shared memory or serialization. There is a vast array of technologies to support such inter-process communication and web services.

[62] For `20+100` to have value `120` in δ as well, dynabase literals must be assumed to extend `$builtin`, or to begin with a standard prelude that imports items from `$builtin`, something like `$builtin=$load("builtin"). X =$builtin.X.`

[63] For example, the rule `f(Y+2)=g(Y)` requires addition as expected when forward-chaining an update to `g(95)`, but would require subtraction when backward-chaining a query to `f(97)`. Technically, the rule desugars into `f(Z)=g(Y) whenever Z is Y+2` (see (29) in the appendix (§A)), which results in queries of the form `Z is 95+2` for forward-chaining and `97 is Y+2` for backward-chaining. The latter is an **inverse query** that asks for provable items matching a particular non-ground term (`Y+2`) and having a particular value. Hence, the backward-chaining strategy requires `$builtin` to invert the addition operator.

In the case of the neural network topology of Figure 3, *all* reasonable execution strategies appear to require inverse arithmetic queries `97 is Y+2`, in order to answer the internal query —weight(hidden(X,Y),pixel(74,97))— ("what hidden nodes does the input `pixel(74,97)` connect to?") using the fact that `shared_weight(-1,2)` is provable. (See footnote 25.)

This kind of inverse arithmetic is commonly encountered in logic programming. Such queries cause Prolog to fail at runtime, cause Mercury to fail at compile time (thanks to static analysis), and are supported in constraint logic programming [10], which can even handle `97 is Y+J` by delaying the constraint for later manipulation and checking. Our expectation is that Dyna will support `97 is Y+2` but not `97 is Y+J`.

Another useful example is a random dynabase, which maps each structured term in the Herbrand universe to an independent random number uniformly distributed on $[0, 1]$. Conceptually, this dynabase is infinite, but it is materialized on demand. Query results are stored for reuse if necessary so that the dynabase remains stable. Here is a simple random walk in two dimensions, where `r` is a random dynabase. Exactly one of `x` or `y` changes at each time step:

```
point(T) = new point(T-1).                    % copy old point
point(T).x += r.dx(T) if r.flip(T) < 0.5.     % adjust x                    (24)
point(T).y += r.dy(T) if r.flip(T) >= 0.5.    % or y (but never both)
```

A Markov chain Monte Carlo algorithm like Gibbs sampling can be implemented using time-indexed dynabases in this way. In general, only the useful statistics computed from the random walk should be output items of the dynabase, so that the system is not required to remember all of the points or random numbers along the random walk, nor `r` itself. In this case, an efficient implementation could simply modify `point(T-1)` in place to obtain `point(T)`.

5 Execution Strategies

Dyna is, fundamentally, an attempt to find a set of broadly useful computational abstractions. Although its design has been inspired by a range of use cases in statistical AI, expanding outward from parsing, it is not designed with a *particular* AI application in mind but rather hopes to unify common techniques across applications. Dyna attempts to follow in the fine tradition of "fast Prologs" such as XSB [209], Mercury [138], and IISProlog [153] by providing an expressive, usable, efficient substrate upon which to experiment with declarative specifications.

5.1 Dyna 1

Our first prototype of Dyna [71,70] was a single-dynabase, agenda-driven, forward-chaining, semiring-weighted Datalog that allowed non-stratified aggregation, non-flat terms, and change propagation. Rules in Dyna 1 remained range-restricted, and the whole program had to be within one semiring (see §3.3) as in [89]. The implementation supported provenance queries and reverse-mode automatic differentiation.

A Dyna 1 program compiled into C++ classes that could be called by a procedural program.[64] The compiler generated custom pattern-matching code and data indices suitable to run the general forward-chaining strategy of §2.6 on the particular rules of the input program, where the indices supported the query modes required for passenger queries against the chart. A non-standard priority function to order the updates on the agenda (see §5.2) could be specified in C++ by the user. The compiler generated custom classes to represent the different types of terms, and stored both the terms and the indices in hash tables.

For simplicity, the Dyna program was transformed before compilation into a binarized form where each rule had at most two antecedents. This **folding** transformation effectively chose a fixed join order for multi-way joins. The intermediate items introduced by the folding transformation had to be materialized and stored in the chart at runtime, like all items.

Despite this one-size-fits-all approach, which limited expressiveness and speed [71], Dyna 1 was asymptotically efficient [136], and has been used successfully in many novel papers on statistical AI (on parsing [182,62,184,74,66,73], grammar induction [60,191,190,188], machine translation [75,186,46,110], finite state methods [175,189,192,65], and others [183,187]) as well as in classroom instruction [193,67].

The present paper constitutes an overview of the next version of Dyna, which is currently under development. The new language design is much more expressive and will support the various examples in this paper. The storage and execution strategies will also be much more flexible. We are investigating support for specifying data layout (and indexing) and declarative control of execution. In the rest of this section, we touch briefly on some of these topics.

[64] This approach resembles (e.g.) the convex optimization "parser-solvers" of [135], which also consume high-level descriptions of problems and emit specialized solvers.

5.2 Beyond Forward Chaining

In §2.6, we showed how a shortest-path program in Dyna (10) would be executed by a pure forward-chaining strategy. We now sketch several reasons why additional strategies are necessary or desirable. (In §5.3, we will discuss in slightly more detail how such strategies can be specified.)

Prioritization Even executing this simple, range-restricted, semiring-weighted program becomes more complex if we wish to be as efficient as Dijkstra's shortest-path algorithm [58]. To match that behavior, we must *prioritize* the updates on the agenda by the update value. We should also carry out updates at push time rather than pop time (a strategy that is correct, efficient, and space-saving for this program, because `min` is idempotent, but not for all programs).

Convergence Dijkstra's algorithm also exploits the fact that with the prioritization above, `cost_to("nyc")` converges as soon as we pop the first update to it (provided that the graph is static and has no negative-weight edges). Recognizing this early convergence[65] can enable us to respond quickly to a specific query `cost_to("nyc")` rather than waiting until the agenda is empty.

Generalized Updating We noted earlier (§4.6) that when a Dyna program is not semiring-weighted, updates on the agenda do not always simply increment some item's value using its aggregation operator. We must be able to process updates of other sorts, consistent with update mode declarations: replace, retract, increment, invalidate, delete-and-rederive, etc. This can require additional data structures (e.g., to rapidly change or retract aggregands to `min=` without recomputing the minimum from scratch).

We must also choose which types of updates to generate, consistent with update mode declarations. Constructing these updates may require exploiting arithmetic identities such as distributivity, idempotence, additive inverses, annihilators, etc. For example, the result of `max=` can be maintained with a priority queue, and strategies can rely on the fact that `max` is idempotent. By contrast, maintaining `+=` requires a slightly larger data structure, and must avoid double-counting because `+` is not idempotent; however, it can take advantage of subtraction to remove summands, whereas `max` does not have a corresponding inverse operator. The `|=` aggregator only needs to keep track of whether it has at least one aggregand and at least one `true` aggregand (see footnote 67).

There is a substantial recent literature on update propagation, spanning deductive databases [16], relational databases [171], streaming data [5], logic programming [174], and functional programming [3]. We believe that our general setting presents interesting new challenges for update propagation (in particular, non-ground updates, non-replacement updates, mode restrictions on plans, and the desire to match the performance of hand-built algorithms by exploiting in-memory data structures and arithmetic properties of the aggregators).

Backward Chaining and Memoization Rather than computing all provable items and storing all of them in the chart, we may wish to reduce computation and storage space by computing some of them only on demand (backward chaining).[66]

Once an item's value (or null) is computed on demand by backward chaining, that value (or null) may be permanently or temporarily **memoized** by storing it in the chart for reuse [196,205]. In our view, the role of forward chaining (§2.6) is to keep any such **memos** up to date when their transitive antecedents change (e.g., because the extensional data change).[67]

[65] This is a special case of A* search, and is related to the branch-and-bound strategy in footnote 37.

[66] **Backward chaining** is lazy computation, which answers a query about a rule's consequent (head item) by combining the results of various queries about its antecedents (body items). By contrast, **forward chaining** (§2.6) is eager computation, which propagates updates from a rule's antecedents to its consequents, using an agenda.

[67] Note that when one of the aggregands to an item is an annihilator (e.g., `true` for `|=`, 0 for `*=`, or $-\infty$ for `min=`), updates to the other aggregands do not need to be reported: they will have no effect on the annihilated value. This is a generalization of short-circuit boolean evaluation and the "watched literals" trick of [144,86].

Exploiting this insight, we can use a chart that stores memos for some arbitrary subset of the items [209,2], while saying nothing about other items. At a given time, some memos may be out of date, but in that case, there will be updates pending on the agenda that will eventually result in correcting them via forward chaining. We allow memos to be discarded at any time.[68]

The natural strategy for maintaining such a chart is a mixture of backward chaining (to query unmemoized items) and forward chaining (to update memos). Pure backward and forward chaining are the extreme cases where nothing and everything are memoized, respectively. In the latter case, all memos are initially null, and forward chaining is needed to revise them. We omit the (interesting) details here. To instantiate such a strategy for a specific set of Dyna rules, the Dyna execution engine must come up with specific query plans and update plans that also satisfy the mode declarations of §4.6.

Indexing A query first checks whether its answer is memoized in the (finite) chart. Indexing the chart can speed up this search. An index is really a memoized query that can be used as part of a larger query plan, but which like any memo must be kept up to date by forward chaining during its lifetime. We wish to choose optimal indices that balance the cost of queries with the cost of index maintenance

For example, when forward-chaining (10), it is necessary to query `edge_cost("bal",V)` whenever `cost_to("bal")` is updated. These frequent passenger queries are much faster when `edge_cost` is indexed on its first argument.[69] Furthermore, this index is cheap to maintain because the `edge_cost` relation that defines the graph rarely changes. By contrast, there is little need to speed up the passenger query `cost_to("bal")`, since it only arises when some `edge_cost("bal",`v`)` changes.

We can also allow partial indices. A single entry within the `edge_cost` index above, such as the list of out-edges of `"bal"`, is really just a memo storing the result of a particular non-ground query, `edge_cost("bal",V)`. Like any memo, it must be kept up to date, which requires extra forward chaining (i.e., index maintenance). However, index memos can usually be individually flushed, since any could be recomputed later from any other or directly from a query supporting enumeration.

Query Planning Finally, we wish to choose optimized query plans to instantiate rules during backward and forward chaining. This is the point at which the connection to Datalog is perhaps most relevant. For example, there are several strategies for handling the 3-way join in the parsing program of Figure 8.[70] In the DPLL program of Figure 12, the order in which one searches for a `true` value among the disjuncts of `|=` ("value ordering") can be crucial, as can the free choice of a next variable by `?=` ("variable ordering").

Query planning is more complicated in Dyna than in Datalog. The plan (e.g., a particular nested-loop join order) must be chosen such that the recursive queries will have modes that are supported by the dynabases receiving them and produce results that are useful to the query plan. Recall that the result of a recursive query is not necessarily a finite, materialized relation, but may itself be an intensional relation defined by a Dyna program (§4.6).

Storage Within a particular dynabase, certain types of terms are frequently constructed, stored, and accessed. An implementation can benefit from identifying these common types and storing them in dedicated classes with more compact layout, interning, function specialization for operations like unification, and so on.

[68] This approach is more flexible than previous work, to our knowledge. Still, partial materialization has been previously investigated in several pure-language systems. ΔML [3] is a version of ML augmented to support change propagation, and mechanisms have been proposed for both coarse and fine-grained control over its memo tables [2]. LolliMon [132] and Ciao [44] allow forward and backward chaining in the same program, but only in different strata. DSN [43] allows the programmer to specify sizes of memo stores, but apparently does not ensure that information is not lost.

[69] Or better yet, stored as a graph adjacency-list structure, with the interned string `"bal"` directly storing a list of weighted edges to its neighbors.

[70] Furthermore, one could choose a "subset-at-a-time" strategy such as processing all items of the form `phrase(X,`i`,`j`)` as a group. This is actually common in parsing algorithms. It increases the space of join strategies since every query or update now has additional free variables, such as `X`.

Program Transformation In some cases it is possible to transform the Dyna program into another Dyna program that is asymptotically more efficient, using a variety of techniques that we reviewed, extended, or proposed in [69].[71]

5.3 Controlling Execution

The previous section illustrates the range of storage and execution strategies available for Dyna programs. This raises the question of how to describe and select specific strategies. User-defined pragmas or flags would be one approach. However, it can be more efficient to adapt the strategies to a particular workload.

We are developing *declarative* mechanisms that allow a Dyna program to reflect on itself and control its own execution. We use the term **voodoo computation** for these occult interactions. A dynabase may define some special **voodoo output items**[72] whose values are consulted by a Dyna runtime implementation and affect its choices about how to perform other computations. The value might change over time as the item receives update messages.

Typically, a voodoo output item is a structured term that describes how to handle some other item,[73] update, query, rule, etc. in the dynabase. For example, if α is an item being computed by backward chaining, then the Dyna runtime implementation may want to know what query plan to use and whether to memoize the result. If α is being updated by forward chaining, then the runtime may want to know which update modes are preferred (when there is a choice) and where to prioritize the update on the agenda.

To answer these questions, the system will query the dynabase's voodoo items.[74] A Dyna programmer is free to write any Dyna rules to define the values of these items. Notice that a single rule may define the values of many voodoo items. For example, something like[75]

```
$priority(phrase(X,I,J)) = I-J.      % a negative integer          (25)
```

says that updates to narrower phrases should have higher priority on the agenda than updates to wider phrases. This is a well-known parsing strategy, essentially the CKY algorithm [211].

The voodoo rules become more interesting if they depend on the data. For example, in place of the simple rule (25), the priorities of updates to CKY's various `phrase` items may be determined by a *dynamic-programming analysis* of the entire input sentence [35,98,168,115,39,79]—which can be written directly in Dyna. Clever prioritization of this sort can result in much faster convergence to the correct answer; examples include A* or best-first search.

As another example, the choice of a query plan might depend on aggregate statistics of the data such as cardinalities, selectivities, and histograms—which, again, can be easily maintained by Dyna. Dyna's prioritized agenda means that some of these statistics can be kept up to date more aggressively than others, allowing the system to vary between the traditional extremes of proactive (scheduled) and reactive (eager) strategies [97,40].

[71] As well as other strategies, such as hierarchical A* search [79] (roughly a generalization of the magic sets transform to certain weighted `min=` programs); the branch-and-bound technique we discussed in footnote 37 (which is related to the magic conditions transform of [147]); and surely others yet to be discovered.

[72] Recall from §4.4 that a dynabase's output items are intended to be read by others, while input items are intended to be written by others.

[73] The handling of voodoo items themselves might be dictated by yet more voodoo items. The regress ends where voodoo items are undefined, or defined by very simple rules that require no planning, at which point default strategies can be applied.

[74] A complication is that the system may want information about a *non-ground* term such as a rule, a non-ground query, a non-ground entry in the chart, a class of terms that match a particular pattern, an index, etc. Unfortunately, in §2.3 we noted that the value relationship is between pairs of *ground* terms. To allow assertions and reasoning about non-ground terms, we employ a new kind of primitive Dyna object, the **frozen term**. Frozen terms are in one-to-one correspondence with the full universe of Dyna terms, but they are regarded as unanalyzed primitive objects and therefore as ground (variable-free). We provide operators that freeze and melt terms according to this one-to-one correspondence.

[75] In this context, the `phrase(X,I,J)` item names are not replaced by their values but are treated simply as terms. The system uses only incomplete queries (§4.6) to consult `$priority` items, in order to avoid a Catch-22 where the agenda must be used to create the priorities on the agenda.

Voodoo rules may also depend on the current or historical system state. Such state can be exposed through **voodoo input items** that are automatically updated by the system as it executes. For example, the estimated cost of a query plan or the estimated future value of a memo may depend in part on factors such as which indices are currently memoized, where various data currently reside in the memory hierarchy, and how long various operations have taken in the recent past. As another example, whether the result of a complete query is considered to have converged sufficiently may depend on summary statistics derived from the current contents of the agenda.

In general, the essential idea of voodoo items is that a dynabase that specifies some *expensive* computation can define some *cheaper* computations which may guide the runtime through its execution. We are considering the design of voodoo items to help control memoization policies, indexing policies, storage layout, pruning of updates [149,104] (a common technique that improves speed at the expense of accuracy), and partitioning of the computation across a cluster. Finally, we are interested in using AI techniques—reinforcement learning and other search and learning methods—to help identify the best policies for a particular program and workload.

6 Conclusion

We have described our work towards a general-purpose weighted logic programming language that is powerful enough to address the needs of statistical AI. Our claim is that modern AI systems can be cleanly specified using such a language, and that much of the implementation burden can be handled by general mechanisms related to logical deduction, database queries, and change propagation. In our own research in natural language processing, we have found a simple prototype of the language [71] to be very useful, enabling us to try out a range of ideas that we otherwise would have rejected as too time-consuming. The new version aims to support a greater variety of execution strategies across a broader range of programs, including the example programs we have illustrated here.

A Formal Semantics

For the interested reader, this appendix sketches a provisional formal semantics for the version of Dyna described in this paper. (We have not proved its correctness.) Obtaining the behavior outlined in §2.7 requires particular care.

Terms Dyna's universe of terms is similar to Prolog's, with small extensions. A **term** is a primitive term, a structure, or a variable. A **primitive term** is a number, a string, a foreign object,[76] a literal dynabase, or a foreign dynabase.[77] A **structure** is a **functor** such as `sibling` paired with a list of zero or more terms, known as the **arguments**. The special functor $\diamond$ is not available in the Dyna language and is used to construct extended dynabases (see below).

A term is a **ground term** if it is either a primitive term or a structure whose arguments are ground terms. $\mathcal{G}$ denotes the set of all ground terms (the **Herbrand universe**), and $\mathcal{G}_{\text{struct}}$ the set of all ground structures.

Some examples of ground terms as they would be written in Dyna's surface syntax are

- `"nyc"` (a string literal)
- `alice` (a structure with 0 arguments)
- `sibling(bob,dave)` (a structure with > 0 arguments, or "compound term")
- `3+4*5` (syntactic sugar equivalent to `'+'(3,'*'(4,5))`)
- `a += 100` (syntactic sugar equivalent to `'+='(a,100)`)
- `{a+=100. b(X)=a*X.}` (a literal dynabase; again, this is a *ground* term)
- `transpose({element(I,I)=1.})` (a structure with a dynabase argument)

[76] Allowing foreign objects to appear in items' names or values allows dynabases to store and manipulate arbitrary objects passed in from the procedural world. It also permits extending the built-in primitives beyond numbers and strings.

[77] Footnote 74 discusses an extension to another kind of primitive term, the **frozen term**.

Term Types and Groundings A **type** is a subset of $\mathcal{G}$. An untyped variable[78] such as `X` has the unrestricted type $\mathcal{G}$. A typed variable ranges over only ground terms of its type; for example, `int Y` is a typed variable that ranges over integers.

If τ is a possibly non-ground term, then $\mathcal{G}_\tau$—the set of **groundings** of τ—consists of the ground terms that can be obtained by consistently replacing each variable that occurs in τ with some ground term of the appropriate type. For example, if $\tau =$ `f(X,g(X,int Y))`, then $\mathcal{G}_\tau$ contains the grounding `f(s(z),g(s(z),3))`.

Dyna's type system [197] permits all types of the form $\mathcal{G}_\tau$ (note that this includes constants and product types), as well as finite unions of other types (i.e., sum types), recursive types such as lists, and primitive types (integers, strings, foreign primitive types, and dynabase types as sketched in §4.4).

Types are useful to improve the efficiency of storage and unification, to catch programming errors (see footnote 82), and occasionally to explicitly restrict the groundings of a rule.

Dynabases A **dynabase** is either a foreign dynabase or a native dynabase. A **foreign dynabase** is not implemented in the Dyna language but provides the same interface as if it were (§4.7). A **native dynabase** is either a literal dynabase or an extended dynabase. A **literal dynabase** specifies a totally ordered multiset of rules $\mathcal{R}_\delta$ as well as optional declarations. An **extended dynabase** arises when one dynabase δ uses the construction `new` ε.[79] As detailed later, the resulting extended dynabase φ is a 4-tuple of terms having the form $\diamond(\varepsilon,\delta,\cdot,\cdot)$. Its first two arguments are the parent ε and the owner δ (see §2.7). In general, we will refer to the first two arguments of an extended dynabase φ via the accessors $\uparrow_p\varphi$ and $\uparrow_o\varphi$, which are undefined if φ is not an extended dynabase.[80]

Valuations For each **dynabase** δ, it is possible to determine from declarations an **item type** $\mathcal{I}_\delta \subseteq \mathcal{G}_{\text{struct}}$ and a **valuation type** $\mathcal{V}_\delta \subseteq \mathcal{I}_\delta \times \mathcal{G}$.[81] This determines which items might be defined in δ and what types their values have. An item $\alpha \in \mathcal{I}_\delta$ is said to have **value type** $\mathcal{V}_\delta(\alpha) = \{\gamma \in \mathcal{G} : (\alpha, \gamma) \in \mathcal{V}_\delta\}$ in the dynabase δ. (The term α may have a different value type in other dynabases where it is also an item.)

Each dynabase δ provides a **valuation function** $[\![\cdot]\!]_\delta : \mathcal{I}_\delta \to \mathcal{G}$, which is a *partial* function that maps items to their values. This function defines the correct results for queries against the dynabase. The function must be consistent with the dynabase's valuation type, in the sense that for all $\alpha \in \mathcal{I}_\delta$, either $[\![\alpha]\!]_\delta$ is undefined (i.e., α is null) or else $[\![\alpha]\!]_\delta \in \mathcal{V}_\delta(\alpha)$.[82]

Constraints on Valuation A native dynabase's valuation function $[\![\cdot]\!]_\delta$ is defined by rules and declarations that specify, for each item $\alpha \in \mathcal{I}_\delta$, an aggregation operator $\oplus_\alpha\!=$ that produces values in $\mathcal{V}_\delta(\alpha)$, and a collection of 0 or more aggregands $\mathcal{A}_\delta(\alpha) \subseteq \mathcal{G}$, as defined below.

The sequence of rules $\bar{\mathcal{R}}_\delta$ is totally ordered. The aggregand collection $\mathcal{A}_\delta(\alpha)$ is a multiset (allowing multiple instances of the same aggregand value) under a total preorder (which is like a total ordering but permits ties). The aggregation operator $\oplus_\alpha\!=$ may refer to the ordering of its aggregands $\mathcal{A}_\delta(\alpha)$, and in particular, `:=` does so. We define later how to identify the rules and aggregands; the declarations of aggregation operators are ordinarily inferred from the surface form of the rules.

More precisely, the rules and declarations *constrain* rather than define the valuation function: $[\![\alpha]\!]_\delta$ is constrained to equal the result of aggregating $\mathcal{A}_\delta(\alpha)$ with $\oplus_\alpha\!=$, or to be undefined if this result is null.

[78] As in Datalog and Prolog, variables are denoted by capitalized identifiers.

[79] Any extension of δ that calls `new` on the same ε will own a different extension of ε.

[80] The dynabase literal in (14) appears to refer to a variable outside the literal. However, we treat this rule as syntactic sugar for

```
transpose(Matrix) = new {element(I,J) = $arg(1).element(J,I).}.
transpose(Matrix).$arg(1) = Matrix.
```

which involves a true dynabase literal. Notice that the `new` operator creates a separate extension of the literal dynabase for each grounding of the rule (each value of `Matrix`).

[81] Type declarations may be omitted, in which case they will be inferred from the stated declarations and rules.

[82] If $\alpha \notin \mathcal{I}_\delta$, it is a type error for a program to try to evaluate $[\![\alpha]\!]_\delta$ or contribute aggregands to it. It is also a type error to violate the additional information hiding constraints imposed by the declared type of δ (§4.4).

As mentioned in §2.4 and §3.3, Dyna cannot guarantee that there is a unique solution to this set of constraints. Thus, queries against the dynabase are permitted to use any valuation function $[\![\cdot]\!]_\delta$ that satisfies the constraints: that is, any supported model [9]. For consistency, multiple queries to the same dynabase must consult the same valuation function—this is the "single-answer semantics" of [212]. If no consistent valuation function exists, then queries must use a relaxed valuation function that satisfies the constraints on a *maximal* subset of the items, and maps the other items to an error value.[83]

Errors are in general treated like ordinary values. In addition to the above case, aggregators may also produce error values, as may queries to foreign dynabases such as `$builtin` (see §4.7). For example, it is an error for `=` to have more than one aggregand, or to divide by zero. Aggregating error values, or evaluating items such as `2*`error that contain error values, generally produces further error values. However, it is possible to define versions of operations that can recover from certain errors: for example, if an error is disjoined with `true` or multiplied by `0`, the result may be an ordinary value. This resembles catching of exceptions.

Queries The valuation function $[\![\cdot]\!]_\delta$ is used to answer queries against the dynabase δ, both queries made by an external process and those needed internally during forward or backward chaining. A **ground query** requests the value $[\![\alpha]\!]_\delta$ of a single item α. A **non-ground query** requests the values of all groundings of a term τ whose values are defined: that is, it requests the restriction of the valuation function $[\![\cdot]\!]_\delta$ to the domain $\mathcal{G}_\tau$. A **general query** requests the restriction of $[\![\cdot]\!]_\delta$ to a particular subset of the valuation type $\mathcal{V}_\delta$; e.g., the query `"bar" is f(6,Y)` requests all (item, value) pairs where the item grounds `f(6,Y)` and has value `"bar"`. This allows "inverse queries" that constrain the value as well as the item, which is sometimes necessary during inference (see footnote 63).

Rule Normal Forms While Dyna supports a flexible syntax for writing rules, internally rules have a uniform representation. The transformation from surface syntax to internal form proceeds in three stages: (1) a conversion to an **administrative normal form** [82], (2) a rewrite that prepares rules for inheritance and ownership, and finally (3) a rewrite that gives unique names to `new` objects. We will illustrate with a few small examples.

Administrative Normal Form Evaluated subterms that appear in the head or body are now desugared by identifying these expressions and making their evaluations explicit as **preconditions** in the rule body.[84] Recall that the precondition γ `is` α means that γ is the value of α.

1. For example, the simple rule

```
sum += f(X) whenever X > 5.
```
(26)

   is rewritten to

```
sum += true is X > 5, F is f(X), F.
```
(27)

2. Consider the rule from (10)

```
cost_to(V) min= cost_to(U)+edge_cost(U,V).
```
(28)

   Making nested evaluations explicit, this becomes

```
cost_to(V) min= A is cost_to(U),
                B is edge_cost(U,V), C is A+B, C.
```
(29)

[83] Suppose δ is defined by the rules `a=a+1` and `b=2`. The only maximal consistent relaxed valuation function has $[\![\texttt{a}]\!]_\delta$ = error, $[\![\texttt{b}]\!]_\delta = 2$. It is possible that querying `a` or even `b` on this dynabase will fail to terminate; but in general, better execution algorithms will terminate (quickly) on a wider range of queries.

[84] Dyna's surface syntax allows syntactic operators and declarations that affect the de-sugaring process by explicitly suppressing or forcing evaluation, something like LISP's `quote`, `eval`, and macro or NLAMBDA forms. The details need not concern us here.

3. Here is a more complex rule in which the head and body items are in other dynabases, and in which there is evaluation within the head:

```
db(g(I)).f(X) += db2(I,J).h(X).
```
(30)

is, in administrative normal form,

```
DB.f(X) += DB is db(G), G is g(I),
           DB2 is db2(I,J), H is DB2.h(X), H.
```
(31)

4. To see how a rule with `new` transforms, consider

```
c(I,J) = pair(new I, new I).
```
(32)

The `new` expressions evaluate, yielding

```
c(I,J) = N1 is new I,
         N2 is new I, pair(N1,N2).
```
(33)

Parametric Normal Form To support inheritance and ownership of dynabases, as in §2.7, our rules are further normalized to be *parametric* over dynabases. This form allows the same rule to participate in defining multiple dynabases. The head item never contains `.`, but all body items do. The conversion to parametric normal form introduces three special variables that were not permitted in administrative normal form. They will be bound to particular dynabases when the rule is used, as described later. The special variables are `_E` ("evaluation"—the dynabase that evaluates body items by default), `_D` ("destination"—the dynabase that receives the head item), and `_M` ("match"—the dynabase that was originally defined by a literal dynabase or an owner to receive the head item, before any extension).

To convert an administrative normal form rule into parametric normal form,

– Make body evaluations not qualified with `.` instead qualified with `_E.`.
– Case-analyse the head:
  - If the head is of the form `V.`τ, replace it with τ and add the precondition that `_M = V`.
  - Otherwise, add the precondition that `_D = _E`.

Continuing the examples from above,

1. The rule from (27), having no `.` in the head, becomes

```
sum += _D = _E, true is _E.(X > 5), F is _E.f(X), F.
```
(34)

2. (10)'s normal form (29) similarly is rewritten as

```
cost_to(V) min= _D = _E, A is _E.cost_to(U),
                B is _E.edge_cost(U,V),
                C is _E.(A+B), C.
```
(35)

3. Since (31) does have a `.` in the head, it is rewritten thus:

```
f(X) += DB = _M, G is _E.g(I),
        DB is _E.db(G), DB2 is _E.db2(I,J),
        H is DB2.h(X), H.
```
(36)

4. Our last example, (33), has a head without `.` and has no body evaluations, so its parametric normal form is very close to its ANF:

```
c(I,J) = _D = _E, N1 is new I,
         N2 is new I, pair(N1,N2).
```
(37)

Diamond Normal Form Finally, suppose the rule r appears in the parametric normal form rule sequence $\mathcal{R}_\delta$, where δ is a dynabase literal. Each operation `new` p in r is replaced with a construction of a particular extended dynabase using $\diamond$. The $\diamond$ structure must capture the parent p and owner `_E` as well as uniquely identifying which `new` in δ this is, and values of other variables in the rule (`Vs`). For example, `c(I,J)` in (37) defines a pair of *distinct* new dynabases, each with its own name, for each of the (`I, J`) pairs. Similarly, in line 2 of Figure 12, a single `new` creates a separately extensible `child(Val)` for each `Val`.

- Let `[`$\texttt{V}_1$`, `$\texttt{V}_2$`, ...]` be a list of all variables in r except those for which r has a precondition of the form `V is new` $\cdot$ and also excluding `_D`, `_E`, and `_M`.
- Add a precondition `Vs = [`$\texttt{V}_1$`, `$\texttt{V}_2$`, ...]` to r, where `Vs` is a fresh variable.
- Replace each precondition of the form $\texttt{N}_i$ `is new` p (where p is either a variable or a dynabase literal value) with the following version:
  $\texttt{N}_i$ `= `$\diamond$`(`p`,_E,`n`,Vs)`, where n is a *unique* identifier for this particular `new` token.[85]

Our examples without `new` are unchanged; if (37) is the only rule in the dynabase literal φ that contains it, then it might be rewritten as

```
c(I,J) = _D = _E, N1 is ◇(I,_E,[φ,1],[I,J]),
                  N2 is new ◇(I,_E,[φ,2],[I,J]), pair(N1,N2).
```
(38)

Collecting Rules For A Dynabase To obtain the constraints on a native dynabase δ's valuation function, we must collect the set of rules $\bar{\mathcal{R}}_\delta$ that might contribute aggregands to its items. Each dynabase literal δ specifies a finite sequence of rules, $\mathcal{R}_\delta$, which we assume to be in diamond normal form. For any native dynabase δ, we must collect the relevant rules by recursively traversing its parent and owner, supplying preconditions which specify `_D`, `_E` and `_M`. In the following algorithm, when recursively collecting the rules of γ because they are relevant to δ, we concatenate rules from $\uparrow_p\gamma$, then $\mathcal{R}_\gamma$, then rules from $\uparrow_o\gamma$.

For all δ, we define $\bar{\mathcal{R}}_\delta$ as $\bar{\mathcal{R}}^p_{\delta,\delta,\delta}(\delta)$, which collects *all* the rules of δ.

This invokes the parent rule collection function $\bar{\mathcal{R}}^p$. In general, $\bar{\mathcal{R}}^p_{\delta,\varepsilon,\mu}(\gamma)$ denotes the set of rules from γ for which δ, ε, and μ can jointly play the roles of `_D`, `_E` and `_M`, respectively, when γ is δ or one of its transitive parents. We compute it as the ordered concatenation of

1. $\bar{\mathcal{R}}^p_{\delta,\varepsilon,\uparrow_p\mu}(\uparrow_p\gamma)$, if $\uparrow_p\gamma$ is defined,
2. rules from $\mathcal{R}_\gamma$, with additional preconditions `_D = `δ, `_E = `ε, and `_M = `μ.
3. and, if $\uparrow_o\gamma$ is defined, $\bar{\mathcal{R}}^o_{\delta,\uparrow_o\gamma,\mu}(\uparrow_o\gamma)$.

The final step above invokes the owner rule collection function $\bar{\mathcal{R}}^o$. In general, $\bar{\mathcal{R}}^o$ has the same meaning as $\bar{\mathcal{R}}^p$ when γ is reachable from δ by following parent and/or owner links, but is *not* δ or one of its transitive parents. We compute it as the ordered concatenation of

1. $\bar{\mathcal{R}}^o_{\delta,\varepsilon,\mu}(\uparrow_p\gamma)$, if $\uparrow_p\gamma$ is defined,
2. rules from $\mathcal{R}_\gamma$, with additional preconditions `_D = `δ, `_E = `ε, and `_M = `μ.
3. and, if $\uparrow_o\gamma$ is defined, $\bar{\mathcal{R}}^o_{\delta,\uparrow_o\gamma,\mu'}(\uparrow_o\gamma)$, where, if μ matches $\diamond$`(`φ`,`ε`,`n`,Vs)`, μ' is $\diamond$`(`φ`,`γ`,`n`,Vs)`,[86] or μ otherwise.

[85] One such identifier is a triple of δ, the rule r's position within δ, and the position within the rule of the `new` being rewritten.

[86] This branch handles the case where ε creates μ and $\uparrow_p\cdots\uparrow_p\varepsilon$ creates an analogous version, about which *its* owner may say something. Note that ε either *is* γ (when $\bar{\mathcal{R}}^o$ is called from the third step of $\bar{\mathcal{R}}^p$ or $\bar{\mathcal{R}}^o$), in which case this pattern match and rewrite does nothing, or γ is a (transitive) parent of ε (when $\bar{\mathcal{R}}^o$ calls itself in the first step), in which case this rewrite gives us exactly the analogue we seek. Note further that we are scanning the owner of γ for rules matching μ'; we never look at the parent of μ', so the inherited rules will not apply.

Aggregands Each dynabase δ has a finite sequence of rules $\bar{\mathcal{R}}_\delta$, as defined by the above collection. Contributions to an aggregand collection $\mathcal{A}_\delta(\alpha)$ are totally preordered by $\gamma_1 \lesssim \gamma_2$ iff $R_1 \leq R_2$ in the sequence $\bar{\mathcal{R}}_\delta$, where R_i is the unique rule that contributed the aggregand instance γ_i to the collection.

Each grounding of a normal-form rule in $\bar{\mathcal{R}}_\delta$ has the form

$$\alpha \quad \oplus_\alpha\!= \quad \beta_1, \ldots, \beta_k, \gamma.$$

where $\gamma \in \mathcal{G}$ is a ground term. This grounding contributes γ to the collection of aggregands $\mathcal{A}_\delta(\alpha)$ iff all the conditions β_i are true (in which case, $\alpha \in \mathcal{I}_\varepsilon$).

Each precondition β_i has one of the following forms:

– γ_i `is` φ_i`.`α_i, where $\gamma_i \in \mathcal{G}$ and $\alpha_i \in \mathcal{I}_{\varphi_i}$. This condition is true iff $\gamma_i = [\![\alpha_i]\!]_{\varphi_i}$.
– γ_i`=`γ_i', where $\gamma_i, \gamma_i' \in \mathcal{G}$. This condition is true iff $\gamma_i = \gamma_i'$.[87]

Note that the constraints on $[\![\cdot]\!]_\delta$ may refer to $[\![\cdot]\!]_\varphi$, because of the interpretation of γ_i `is` φ`.`α_i preconditions. Furthermore, (17) shows that $[\![\cdot]\!]_\delta$ and $[\![\cdot]\!]_\varphi$ may depend cyclically on each other. Thus, in general it is necessary to solve constraints jointly across multiple dynabases. In other words, to answer queries against δ, one must solve for portions of the two-place valuation function $[\![\cdot]\!]_\cdot$, which defines the value of all items in all dynabases.

References

1. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. In: Proc. of POPL. pp. 247–259 (2002), `http://www.mpi-sws.org/~umut/papers/popl02.html`
2. Acar, U.A., Blelloch, G.E., Harper, R.: Selective memoization. In: Proc. of POPL. pp. 14–25 (2003), `http://www.mpi-sws.org/~umut/papers/popl03.html`
3. Acar, U.A., Ley-Wild, R.: Self-adjusting computation with Delta ML. In: Koopman, P.W.M., Plasmeijer, R., Swierstra, S.D. (eds.) Advanced Functional Programming. Lecture Notes in Computer Science, vol. 5832, pp. 1–38. Springer (2008), `http://www.mpi-sws.org/~umut/papers/afp08.html`
4. Algebraic dynamic programming in bioinformatics, `http://bibiserv.techfak.uni-bielefeld.de/cgi-bin/dpcourse`
5. Ahmad, Y., Koch, C.: DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. In: Proc. of VLDB. pp. 1566–1569 (2009), `http://www.cs.cornell.edu/bigreddata/dbtoaster/vldb2009_dbtoaster_demo.pdf`
6. Alchemy – Open Source AI, `http://alchemy.cs.washington.edu/`
7. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFST: A general and efficient weighted finite-state transducer library. In: Proc. of the 12th International Conference on Implementation and Application of Automata. pp. 11–23. Springer-Verlag (2007)
8. Amazon Mechanical Turk, `https://www.mturk.com/mturk/welcome`
9. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, chap. 2. Morgan Kaufmann (1988), `http://oai.cwi.nl/oai/asset/10404/10404A.pdf`
10. Apt, K.R., Wallace, M.: Constraint Logic Programming using ECLiPSe. Cambridge University Press (2007)
11. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
12. Baayen, H.R., Piepenbrock, R., Gulikers, L.: The CELEX Lexical Database. Release 2 (CD-ROM). Linguistic Data Consortium, University of Pennsylvania, Philadelphia, Pennsylvania (1995)
13. Baroni, M., Bernardini, S., Ferraresi, A., Zanchetta, E.: The WaCky Wide Web: A collection of very large linguistically processed Web-crawled corpora. Language Resources and Evaluation 43(3), 209–226 (2009)
14. Baxter, D., Shepard, B., Siegel, N., Gottesman, B., Schneider, D.: Interactive natural language explanations of cyc inferences. In: Roth-Berghofer, T., Schulz, S. (eds.) ExaCt. AAAI Technical Report, vol. FS-05-04, pp. 10–20. AAAI Press (2005)
15. Beesley, K.R., Karttunen, L.: Finite-State Morphology. University of Chicago Press (2003), `http://www.fsmbook.com`

[87] This case arises from conversion to named-`new` normal form, and from Prolog-style unification constraints. For example, `P=pair(X,Y)` may appear in a rule condition, in order to bind `X` and `Y` to arguments of the value of `P`. One grounding that makes this condition true is to replace `P`, `X`, and `Y` with `pair(1,2)`, `1`, and `2`.

16. Behrend, A.: A classification scheme for update propagation methods in deductive databases. In: Proc. of the International Workshop on Logic in Databases (2009), http://www.iai.uni-bonn.de/~behrend/lid09.pdf
17. Benson, D., Karsch-Mizrachi, I., Lipman, D., Ostell, J., Wheeler, D.: GenBank. Nucleic Acids Research 36 (Database issue)(D), 25–30 (Jan 2008), http://www.ncbi.nlm.nih.gov/genbank/
18. Berg-Kirkpatrick, T., Bouchard-Côté, A., DeNero, J., Klein, D.: Painless unsupervised learning with features. In: Proc. of NAACL. pp. 582–590. ACL, Los Angeles, California (June 2010), http://www.aclweb.org/anthology/N10-1083
19. Bidoit, N., Hull, R.: Minimalism, justification and non-monotonicity in deductive databases. Journal of Computer and System Sciences 38(2), 290–325 (April 1989)
20. Billot, S., Lang, B.: The structure of shared forests in ambiguous parsing. In: Proc. of ACL. pp. 143–151. ACL, Morristown, NJ, USA (1989)
21. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia - a crystallization point for the Web of Data. Web Semantics 7(3), 154–165 (2009)
22. Blunsom, P., Cohn, T., Lopez, A., Dyer, C., Graehl, J., Botha, J., Eidelman, V., Wang, Z., Nguyen, T., Buzek, O., Chen, D.: Models of Synchronous Grammar Induction for SMT. Tech. rep., Center for Language and Speech Processing, Johns Hopkins University (June 2010), http://www.clsp.jhu.edu/workshops/ws10/documents/Blunsom-June-21.pdf
23. Bobrow, D.G., Winograd, T.: An overview of KRL, a knowledge representation language. Cognitive Science 1(1), 3 – 46 (1977), http://www.sciencedirect.com/science/article/B6W48-4FN1JFD-3/2/074766fc77231b605dcf2baa2418d9d0
24. Booth, T., Thompson, R.: Applying probability measures to abstract languages. Computers, IEEE Transactions on 100(5), 442–450 (2006)
25. Boschee, E., Weischedel, R., Zamanian, A.: Automatic information extraction. In: Proc. of the 2005 International Conference on Intelligence Analysis (May 2005), https://analysis.mitre.org/proceedings/Final_Papers_Files/336_Camera_Ready_Paper.pdf
26. Bose, R., Frew, J.: Lineage retrieval for scientific data processing: a survey. ACM Comput. Surv. 37(1), 1–28 (2005)
27. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in newspeak. In: D'Hondt, T. (ed.) ECOOP. Lecture Notes in Computer Science, vol. 6183, pp. 405–428. Springer (2010), http://bracha.org/newspeak-modules.pdf
28. Breck, E.: zymake: A computational workflow system for machine learning and natural language processing. In: Software Engineering, Testing, and Quality Assurance for Natural Language Processing. pp. 5–13. SETQA-NLP '08, ACL, Morristown, NJ, USA (2008), http://www.aclweb.org/anthology/W/W08/W08-0503.pdf
29. Brodie, M.L.: Future Intelligent Information Systems: AI and Database Technologies Working Together. Morgan Kaufman, San Mateo, CA (1988)
30. Buneman, P.: How to cite curated databases and how to make them citable. In: SSDBM '06: Proceedings of the 18th International Conference on Scientific and Statistical Database Management. pp. 195–203. IEEE Computer Society, Washington, DC, USA (2006)
31. Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. In: ICDT '01: Proceedings of the 8th International Conference on Database Theory. pp. 316–330. Springer-Verlag, London, UK (2001), http://db.cis.upenn.edu/DL/whywhere.pdf
32. Burbank, A., Carpuat, M., Clark, S., Dreyer, M., Fox, P., Groves, D., Hall, K., Hearne, M., Melamed, D., Shen, Y., Way, A., Wellington, B., Wu, D.: Final report of the 2005 language engineering workshop on statistical machine translation by parsing. Tech. rep., Johns Hopkins University (2005), http://www.clsp.jhu.edu/ws2005/groups/statistical/documents/finalreport.pdf
33. Burstall, R.M., Collins, J.S., Popplestone, R.J.: Programming in POP-2. Edinburgh University Press, Edinburgh, (1971)
34. Callison-Burch, C., Dredze, M. (eds.): CSLDAMT '10: Proc. of the NAACL-HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk. ACL, Morristown, NJ, USA (2010)
35. Caraballo, S.A., Charniak, E.: New figures of merit for best-first probabilistic chart parsing. Computational Linguistics 24(2), 275–298 (1998), http://www.cs.brown.edu/people/sc/NewFiguresofMerit.ps.Z
36. Carbonell, J.: AI in CAI: An artificial-intelligence approach to computer-assisted instruction. IEEE Transactions on Man-Machine Systems 11(4), 190–202 (2007)
37. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1, 146–166 (1989)
38. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer (June 1990)
39. Charniak, E., Johnson, M., Elsner, M., Austerweil, J., Ellis, D., Haxton, I., Hill, C., Shrivaths, R., Moore, J., Pozar, M., Vu, T.: Multilevel coarse-to-fine PCFG parsing. In: Proc. of HLT-NAACL. pp. 168–175. ACL, New York City, USA (June 2006), http://www.aclweb.org/anthology/N/N06/N06-1022

40. Chaudhuri, S.: Query optimizers: time to rethink the contract? In: SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data. pp. 961–968. ACM, New York, NY, USA (2009)
41. Chiang, D.: A hierarchical phrase-based model for statistical machine translation. In: Proc. of ACL. pp. 263–270. ACL, Morristown, NJ, USA (2005)
42. Chiang, D.: Hierarchical phrase-based translation. Computational Linguistics 33(2), 201–228 (2007), http://www.mitpressjournals.org/doi/abs/10.1162/coli.2007.33.2.201
43. Chu, D., Popa, L., Tavakoli, A., Hellerstein, J.M., Levis, P., Shenker, S., Stoica, I.: The design and implementation of a declarative sensor network system. In: SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems. pp. 175–188. ACM, New York, NY, USA (2007), http://db.cs.berkeley.edu/papers/sensys07-dsn.pdf
44. The Ciao prolog development system, http://www.ciaohome.org/
45. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum (1978), http://www.doc.ic.ac.uk/~klc/NegAsFailure.pdf
46. Cohen, S.B., Simmons, R.J., Smith, N.A.: Products of weighted logic programs. Theory and Practice of Logic Programming (2010)
47. Cohen, S., Nutt, W., Serebrenik, A.: Algorithms for rewriting aggregate queries using views. In: Proc. of ADBIS-DASFAA. pp. 65–78. Springer-Verlag, London, UK (2000), http://www.springerlink.com/content/3EUY12D7G8WXVDTN
48. Cohn, T., Blunsom, P.: A Bayesian model of syntax-directed tree to string grammar induction. In: Proc. of EMNLP. pp. 352–361. ACL, Singapore (August 2009), http://www.aclweb.org/anthology/D/D09/D09-1037
49. Collins, M., Roark, B.: Incremental parsing with the perceptron algorithm. In: Proc. of ACL. ACL '04, ACL, Morristown, NJ, USA (2004), http://dx.doi.org/10.3115/1218955.1218970
50. comScore: comScore reports global search market growth of 46 percent in 2009, http://www.comscore.com/Press_Events/Press_Releases/2010/1/Global_Search_Market_Grows_46_Percent_in_2009
51. Condie, T., Chu, D., Hellerstein, J.M., Maniatis, P.: Evita Raced: metacompilation for declarative networks. Proc. VLDB Endow. 1(1), 1153–1165 (2008)
52. Cortes, C., Vapnik, V.: Support-vector networks. Machine learning 20(3), 273–297 (1995)
53. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: 2001 Haskell Workshop (September 2001), http://www.haskell.org/yale/papers/haskellworkshop01/genuinely-functional-guis.pdf
54. The functional logic language Curry, http://www.informatik.uni-kiel.de/~curry/
55. Dang, H., Kelly, D., Lin, J.: Overview of the TREC 2007 question answering track. In: Proc. of TREC (2008)
56. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (1962)
57. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
58. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959), http://jmvidal.cse.sc.edu/library/dijkstra59a.pdf
59. Domingos, P., Kok, S., Lowd, D., Poon, H., Richardson, M., Singla, P.: Markov Logic, pp. 92–117. Springer-Verlag, Berlin, Heidelberg (2008)
60. Dreyer, M., Eisner, J.: Better informed training of latent syntactic features. In: Proc. of EMNLP. pp. 317–326. Sydney (July 2006), http://cs.jhu.edu/~jason/papers/#emnlp06
61. Dreyer, M., Eisner, J.: Graphical models over multiple strings. In: Proc. of EMNLP. pp. 101–110 (2009)
62. Dreyer, M., Smith, D.A., Smith, N.A.: Vine parsing and minimum risk reranking for speed and precision. In: Proc. of CoNLL. pp. 201–205 (2006), http://www.clsp.jhu.edu/~markus/dreyer06depparse.pdf
63. Duchi, J., Tarlow, D., Elidan, G., Koller, D.: Using combinatorial optimization within max-product belief propagation. In: NIPS 2006. pp. 369–376 (2007)
64. ECMA International: Standard ECMA-262 (2009), http://www.ecma-international.org/publications/standards/Ecma-262.htm
65. Eisner, J.: Parameter estimation for probabilistic finite-state transducers. In: Proc. of ACL. pp. 1–8. Philadelphia (July 2002), http://cs.jhu.edu/~jason/papers/#acl02-fst
66. Eisner, J.: Transformational priors over grammars. In: Proc. of EMNLP. pp. 63–70. Philadelphia (July 2002), http://cs.jhu.edu/~jason/papers/#emnlp02
67. Eisner, J.: Declarative methods (Fall 2005–), http://www.cs.jhu.edu/~jason/325/, course offered at Johns Hopkins University
68. Eisner, J.: Dyna: A *non*-probabilistic programming language for probabilistic AI. Extended abstract for talk at the NIPS*2008 Workshop on Probabilistic Programming (December 2008), http://cs.jhu.edu/~jason/papers/#nipsw08-dyna
69. Eisner, J., Blatz, J.: Program transformations for optimization of parsing algorithms and other weighted logic programs. In: Wintner, S. (ed.) Proc. of FG 2006: The 11th Conference on Formal Grammar. pp. 45–85. CSLI Publications (2007), http://www.cs.jhu.edu/~jason/research.html#fg06

70. Eisner, J., Goldlust, E., Smith, N.A.: Dyna: A declarative language for implementing dynamic programs. In: Proc. of ACL, Companion Volume. pp. 218–221. Barcelona (Jul 2004), http://cs.jhu.edu/~jason/papers/#acl04-dyna
71. Eisner, J., Goldlust, E., Smith, N.A.: Compiling comp ling: Weighted dynamic programming and the Dyna language. In: Proc. of HLT-EMNLP. pp. 281–290. Association for Computational Linguistics, Vancouver (Oct 2005), http://cs.jhu.edu/~jason/papers/#emnlp05-dyna
72. Eisner, J., Kornbluh, M., Woodhull, G., Buse, R., Huang, S., Michael, C., Shafer, G.: Visual navigation through large directed graphs and hypergraphs. In: Proc. of IEEE InfoVis, Poster/Demo Session. pp. 116–117. Baltimore (Oct 2006), http://cs.jhu.edu/~jason/papers/#infovis06
73. Eisner, J., Satta, G.: Efficient parsing for bilexical context-free grammars and head-automaton grammars. In: Proc. of ACL. pp. 457–464. University of Maryland (June 1999), http://cs.jhu.edu/~jason/papers/#acl99
74. Eisner, J., Smith, N.A.: Parsing with soft and hard constraints on dependency length. In: Proc. of the International Workshop on Parsing Technologies. pp. 30–41. Vancouver (October 2005), http://cs.jhu.edu/~jason/papers/#iwpt05
75. Eisner, J., Tromble, R.W.: Local search with very large-scale neighborhoods for optimal permutations in machine translation. In: Proc. of the HLT-NAACL Workshop on Computationally Hard Problems and Joint Inference in Speech and Language Processing. pp. 57–75. New York (June 2006), http://cs.jhu.edu/~jason/papers/#hard06
76. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming (1997), http://conal.net/papers/icfp97/
77. Fellbaum, C. (ed.): WordNet: An Electronic Lexical Database. The MIT Press, Cambridge, MA ; London (May 1998), http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=8106
78. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: A functional I/O system or, fun for freshman kids. In: Hutton, G., Tolmach, A.P. (eds.) ICFP. pp. 47–58. ACM (2009), http://www.ccs.neu.edu/scheme/pubs/icfp09-fffk.pdf
79. Felzenszwalb, P.F., McAllester, D.: The generalized A* architecture. J. Artif. Int. Res. 29(1), 153–190 (2007), http://nagoya.uchicago.edu/~dmcallester/astar.pdf
80. Fidler, S., Boben, M., Leonardis, A.: Learning hierarchical compositional representations of object structure. In: Dickinson, S., Leonardis, A., Schiele, B., Tarr, M.J. (eds.) Object Categorization: Computer and Human Vision Perspectives. Cambridge University Press (2009), vicos.fri.uni-lj.si/data/alesl/chapterLeonardis.pdf
81. Finkel, J.R., Grenager, T., Manning, C.: Incorporating non-local information into information extraction systems by Gibbs sampling. In: Proc. of ACL. pp. 363–370. ACL, Morristown, NJ, USA (2005)
82. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proc. of PLDI. pp. 237–247. PLDI '93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/155090.155113
83. Franz, A., Brants, T.: All our n-gram are belong to you (August 2006), http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html, available from the Linguistic Data Consortium, Philadelphia, as LDC2006T13.
84. Gazdar, G.: Applicability of indexed grammars to natural languages. In: Reyle, U., Rohrer, C. (eds.) Natural Language Parsing and Linguistic Theories, pp. 69–94. Dordrecht:Reidel (1988)
85. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming, Proc. of the 5th International Conference and Symposium. pp. 1070–1080 (1988)
86. Gent, I.P., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in minion. In: Benhamou, F. (ed.) CP. Lecture Notes in Computer Science, vol. 4204, pp. 182–197. Springer (2006)
87. Giegerich, R., Meyer, C., Steffen, P.: A discipline of dynamic programming over sequence data. Science of Computer Programming 51(3), 215–263 (2004)
88. The Global WordNet Association, http://www.globalwordnet.org/
89. Goodman, J.: Semiring parsing. Computational Linguistics 25(4), 573–605 (1999)
90. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: Proc. of UAI (2008)
91. Greco, S.: Dynamic programming in datalog with aggregates. IEEE Transactions on Knowledge and Data Engineering 11(2), 265–283 (1999)
92. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proc. of PODS. pp. 31–40 (2007), http://db.cis.upenn.edu/DL/07/pods07.pdf
93. Griewank, A., Corliss, G. (eds.): Automatic Differentiation of Algorithms. SIAM, Philadelphia (1991)
94. Guo, H.F., Gupta, G.: Simplifying dynamic programming via tabling. In: Practical Aspects of Declarative Languages. Lecture Notes in Computer Science, vol. 3057, pp. 163–177 (2004), http://www.springerlink.com/content/fprn44ejj9yxqptq/

95. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Eng. Bull. 18(2), 3–18 (1995)
96. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: Buneman, P., Jajodia, S. (eds.) SIGMOD Conference. pp. 157–166. ACM Press (May 1993)
97. Haas, P.J., Ilyas, I.F., Lohman, G.M., Markl, V.: Discovering and exploiting statistical properties for query optimization in relational databases: A survey. Stat. Anal. Data Min. 1(4), 223–250 (2009)
98. Hall, K., Johnson, M.: Language modeling using efficient best-first bottom-up parsing. In: Proc. of the IEEE Automatic Speech Recognition and Understanding Workshop (2003)
99. Hand, D., Mannila, H., Smyth, P.: Principles of Data Mining. The MIT press (2001), http://cognet.mit.edu/library/books/view?isbn=026208290X
100. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. MIT Press (2005)
101. Hinton, G.: Products of experts. In: Proc. of ICANN. vol. 1, pp. 1–6 (1999), http://www.cs.toronto.edu/~hinton/absps/icann-99.html
102. Ilya Ahtaridis (Membership Coordinator, L.D.C.: (2010), private communication.
103. Jacobs, R.A., Jordan, M.I., Nowlan, S.J., Hinton, G.E.: Adaptive mixtures of local experts. Neural Computation 3(1), 79–87 (1991)
104. Jelinek, F.: Fast Sequential Decoding Algorithm Using a Stack. IBM Journal of Research and Development 13(6), 675–685 (1969)
105. Johnson, M.: Transforming projective bilexical dependency grammars into efficiently-parsable CFGs with unfold-fold. In: Proc. of ACL. pp. 168–175. Prague, Czech Republic (June 2007), http://www.aclweb.org/anthology/P07-1022
106. Johnson, M., Griffiths, T.L., Goldwater, S.: Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. In: NIPS'06. pp. 641–648 (2006), http://cocosci.berkeley.edu/tom/papers/adaptornips.pdf
107. Joshua, http://joshua.sourceforge.net/
108. Kanazawa, M.: Parsing and generation as datalog queries. In: Proc. of ACL. ACL, Prague, Czech Republic (June 23-30 2007)
109. Kaplan, R.M., Bresnan, J.: Lexical-Functional Grammar: A formal system for grammatical representation. In: Bresnan, J. (ed.) The Mental Representation of Grammatical Relations, pp. 173–281. The MIT Press, Cambridge, MA (1982), reprinted in Mary Dalrymple, Ronald M. Kaplan, John Maxwell, and Annie Zaenen, eds., *Formal Issues in Lexical-Functional Grammar*, 29–130. Stanford: CSLI Publications. 1995.
110. Karakos, D., Eisner, J., Khudanpur, S., Dreyer, M.: Machine translation system combination using ITG-based alignments. Proc. ACL-08: HLT, Short Papers (Companion Volume) pp. 81–84 (2008), http://www.aclweb.org/anthology/P/P08/P08-2021.pdf
111. Karakos, D., Khudanpur, S.: Sequential system combination for machine translation of speech. In: IEEE Spoken Language Technology Workshop, 2008. pp. 257–260 (2008)
112. Kemp, D.B., Stuckey, P.J.: Semantics of logic programs with aggregates. Proc. of the International Logic Programming Symposium pp. 338–401 (1991)
113. Kifer, M., Subrahmanian, V.S.: Theory of generalized annotated logic programming and its applications. Journal of Logic Programming 12(4), 335–368 (1992), http://scholar.google.com/url?sa=U&q=http://www.cs.umd.edu/projects/hermes/publications/postscripts/galp.ps
114. Kimmig, A., Demoen, B., Raedt, L.D., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language problog. Theory and Practice of Logic Programming (2011), https://lirias.kuleuven.be/handle/123456789/259607
115. Klein, D., Manning, C.D.: A* parsing: Fast exact Viterbi parse selection. In: Proc. of HLT-NAACL (2003), http://www.stanford.edu/~manning/papers/pcfg-astar.pdf
116. Kline, M.: Mathematics in the modern world; readings from Scientific American. With introductions by Morris Kline. W. H. Freeman San Francisco, (1968)
117. Knuth, D.E.: Semantics of context-free languages: Correction. Mathematical Systems Theory 5(1), 95–96 (1971)
118. Koehn, P.: Europarl: A parallel corpus for statistical machine translation. In: MT Summit. vol. 5 (2005)
119. Lafferty, J., McCallum, A., Pereira, F.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proc. of ICML (2001), http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mccallum/www/papers/crf-icml01.ps.gz
120. Lenat, D.B.: Cyc: a large-scale investment in knowledge infrastructure. Communications of the ACM 38(11), 33–38 (1995)
121. Lenat, D.B., Guha, R.V.: Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
122. Levy, A.Y.: Combining artificial intelligence and databases for data integration. In: Artificial Intelligence Today, pp. 249–268. No. 1600 in Lecture Notes in Computer Science, Springer-Verlag (1999)

123. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: Proc. of OOPSLA. pp. 214–223. ACM, New York, NY, USA (1986)
124. The Linguistic Data Consortium, http://www.ldc.upenn.edu/
125. Liu, H., Singh, P.: Commonsense reasoning in and over natural language. In: Negoita, M.G., Howlett, R.J., Jain, L.C. (eds.) KES. Lecture Notes in Computer Science, vol. 3215, pp. 293–306. Springer (2004), http://web.media.mit.edu/~push/CommonsenseInOverNL.pdf
126. Liu, L., Pu, C., Barga, R., Zhou, T.: Differential evaluation of continual queries. In: Proc. of the 16th International Conference on Distributed Computing Systems. pp. 458–465 (1996)
127. LogicBlox: Datalog for enterprise applications: from industrial applications to research (March 2010), http://www.logicblox.com/research/presentations/arefdatalog20.pdf, presented by Molham Aref at Datalog 2.0 Workshop
128. LogicBlox: Modular and reusable Datalog (March 2010), http://www.logicblox.com/research/presentations/morebloxdatalog20.pdf, presented by Shan Shan Huang at Datalog 2.0 Workshop
129. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. pp. 97–108. ACM, New York, NY, USA (2006), http://db.cs.berkeley.edu/papers/sigmod06-declar.pdf
130. Loo, B.T., Condie, T., Garofalakis, M.N., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Commun. ACM 52(11), 87–95 (2009)
131. Lopez, A.: Translation as weighted deduction. In: Proc. of EACL (2010)
132. López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: Proc. of PPDP. pp. 35–46. ACM, New York, NY, USA (2005), http://www.cs.cmu.edu/~fp/papers/ppdp05.pdf
133. Marcus, M.P., Kim, G., Marcinkiewicz, M.A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., Schasberger, B.: The Penn Treebank: Annotating predicate argument structure. In: HLT (1994), http://acl.ldc.upenn.edu/H/H94/H94-1020.pdf
134. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K., Marek, V., Truszczyński, M., Warren, D. (eds.) The Logic Programming Paradigm: A 25-Year Perspective, pp. 375–398. Springer-Verlag (1999)
135. Mattingely, J., Boyd, S.: Real-time convex optimization in signal processing. IEEE- Signal Processing Magazine 27(3), 50–61 (2010), www.stanford.edu/~boyd/papers/pdf/rt_cvx_sig_proc.pdf
136. McAllester, D.A.: On the complexity analysis of static analyses. J. ACM 49(4), 512–537 (2002)
137. McGuinness, D., Van Harmelen, F., et al.: OWL web ontology language overview. W3C recommendation 10 (2004), http://ia.ucpel.tche.br/~lpalazzo/Aulas/TEWS/arq/OWL-Overview.pdf
138. The Mercury Project, http://www.cs.mu.oz.au/research/mercury/index.html
139. Miller, G.A.: WordNet: A lexical database for English. Communications of the ACM 38(11), 39–41 (1995)
140. Minnen, G.: Magic for filter optimization in dynamic bottom-up processing. In: ACL. pp. 247–254 (1996)
141. Mohr, R., Henderson, T.: Arc and path consistency revised. Artificial Intelligence 28, 225–233 (1986)
142. de Moor, O., Hajiyev, E., Verbaere, M.: Object-oriented queries over software systems. In: Proc. of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 91–91. ACM, New York, NY, USA (2007)
143. Moschitti, A., Quarteroni, S.: Linguistic kernels for answer re-ranking in question answering systems. Information Processing and Management (2010), http://www.sciencedirect.com/science/article/B6VC8-50K53XN-1/2/f6dee662afba19e7f1055f74f87f96e4
144. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: DAC. pp. 530–535. Las Vegas, NV, USA (2001)
145. Muggleton, S.H.: Stochastic logic programs. In: DeRaedt, L. (ed.) Advances in Inductive Logic Programming. IOS Press (1996)
146. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In: Proc. of VLDB. pp. 264–277 (1990), http://www.kirusa.com/mumick/papers/pspapers/vldb90PubDupagg.ps.Z
147. Mumick, I.S., Finkelstein, S.J., Pirahesh, H., Ramakrishnan, R.: Magic conditions. ACM Transactions on Database Systems pp. 107–155 (1996)
148. Nádas, A.: On Turing's formula for word probabilities. IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP-33(6), 1414–1416 (December 1985)
149. Newell, A.: HARPY, production systems, and human cognition. In: Cole, R. (ed.) Production and Perception of Fluent Speech. Lawrence Erlbaum (1980)
150. Ng., R., Subrahmanian, V.S.: Probabilistic logic programming. Information and Computation 101(2), 150–201 (1992)
151. Ngai, G., Florian, R.: Transformation-based learning in the fast lane. In: Proc. of NAACL-HLT (2001)
152. van Noord, G., Gerdemann, D.: Finite state transducers with predicates and identities. Grammars 4(3) (2001)

153. Nussbaum, M.: An interpreter for large knowledge bases. In: CSC '89: Proc. of the 17th ACM Computer Science Conference. pp. 87–97. ACM, New York, NY, USA (1989)
154. Overton, D.: Precise and Expressive Mode Systems for Typed Logic Programming Languages. Ph.D. thesis, University of Melbourne (2003), http://www.mercury.cs.mu.oz.au/information/papers.html#dmo-thesis
155. Pauls, A., Klein, D.: *k*-best A* parsing. In: Proc. of ACL-IJCNLP. pp. 958–966. ACL, Suntec, Singapore (August 2009), http://www.aclweb.org/anthology/P/P09/P09-1108
156. Pearl, J.: Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)
157. Pelov, N.: Semantics of Logic Programs With Aggregates. Ph.D. thesis, Katholieke Universiteit Leuven (April 2004), http://www.cs.kuleuven.be/publicaties/doctoraten/cw/CW2004_02.pdf
158. Pereira, F., Shieber, S.M.: Prolog and Natural-Language Analysis. No. 10 in CSLI Lecture Notes, Cambridge University Press (1987)
159. Pfeffer, A.: Ibal: An integrated bayesian agent language. In: Proc. of IJCAI (2001)
160. Phillips, A.B., Brown, R.D.: Cunei machine translation platform: System description. In: 3rd International Workshop on Example-Based Machine Translation (2009), http://www.cunei.org/about/phillips-ebmt09.pdf
161. Pollard, C., Sag, I.A.: Head-driven Phrase Structure Grammar. The University Of Chicago Press (1994)
162. Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artificial Intelligence 64(1), 81–129 (1993), http://www.cs.ubc.ca/~poole/prob.html#pha
163. Puech, A., Muggleton, S.: A comparison of stochastic logic programs and bayesian logic programs. In: IJCAI Workshop on Learning Statistical Models from Relational Data (2003), http://www.doc.ic.ac.uk/~shm/Papers/puech-paper2.pdf
164. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2010), http://www.R-project.org, ISBN 3-900051-07-0
165. Raedt, L.D., Kimmig, A., Toivonen, H.: Problog: A probabilistic Prolog and its application in link discovery. In: Proc. of IJCAI. pp. 2462–2467 (2007), http://www.cs.kuleuven.be/~dtai/publications/files/42447.pdf
166. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The coral deductive system. The VLDB Journal 3(2), 161–210 (1994), ftp://ftp.cs.wisc.edu/coral/doc/coral.ps, special Issue on Prototypes of Deductive Database Systems.
167. Ramamohanarao, K.: Special issue on prototypes of deductive database systems. VLDB 3(2) (April 1994), http://portal.acm.org/citation.cfm?id=615191.615192
168. Raphael, C.: Coarse-to-fine dynamic programming. IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 23(12), 1379–1390 (December 2001)
169. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62(1-2), 107–136 (2006)
170. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. In: Proc. of PODS. pp. 114–126. San Diego (June 1992), http://www.cs.columbia.edu/~kar/pubsk/aggmon.ps
171. Ross, K.A., Srivastava, D., Sudarshan, S.: Materialized view maintenance and integrity constraint checking: Trading space for time. In: Proc. of SIGMOD (1996), http://portal.acm.org/citation.cfm?id=233361
172. Rush, A.M., Sontag, D., Collins, M., Jaakkola, T.: On dual decomposition and linear programming relaxations for natural language processing. In: Proc. of EMNLP (2010)
173. Russell, B.C., Torralba, A., Murphy, K.P., Freeman, W.T.: Labelme: A database and web-based tool for image annotation. International Journal of Computer Vision 77(1-3), 157–173 (2008)
174. Saha, D., Ramakrishnan, C.R.: Incremental evaluation of tabled prolog: Beyond pure logic programs. In: Practical Aspects of Declarative Languages. Lecture Notes in Computer Science, vol. 3819, pp. 215–229 (2006), http://www.springerlink.com/content/y636851u1351x253/
175. Schafer, C.: Novel probabilistic finite-state transducers for cognate and transliteration modeling. In: 5th Conference of the Association for Machine Translation in the Americas (AMTA). Boston, Massachusetts (August 2006)
176. Schmid, H., Rooth, M.: Parse forest computation of expected governors. In: Proc. of ACL (2001)
177. Shieber, S.M., Schabes, Y.: Synchronous tree-adjoining grammars. In: Proc. of COLING. pp. 253–258. ACL, Morristown, NJ, USA (1990)
178. Shieber, S.M., Schabes, Y., Pereira, F.: Principles and implementation of deductive parsing. Journal of Logic Programming 24(1–2), 3–36 (1995), http://www.eecs.harvard.edu/~shieber/Biblio/Papers/infer.pdf
179. Sikkel, K.: Parsing Schemata: A Framework for Specification and Analysis of Parsing Algorithms. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edn. (1997)
180. Singh, P., Lin, T., Mueller, E.T., Lim, G., Perkins, T., Zhu, W.L.: Open mind common sense: Knowledge acquisition from the general public. In: On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002. pp. 1223–1237. Springer-Verlag, London, UK (2002), http://web.media.mit.edu/~push/ODBASE2002.pdf

181. Singla, P., Domingos, P.: Lifted first-order belief propagation. In: Proc. of AAAI. pp. 1094–1099. AAAI Press (2008)
182. Smith, D., Eisner, J.: Dependency parsing by belief propagation. In: Proc. of EMNLP. pp. 145–156 (2008), http://cs.jhu.edu/~jason/papers/#emnlp08-bp
183. Smith, D.A.: Efficient Inference for Trees and Alignments: Modeling Monolingual and Bilingual Syntax with Hard and Soft Constraints and Latent Variables. Ph.D. thesis, Johns Hopkins University (2010)
184. Smith, D.A., Eisner, J.: Minimum-risk annealing for training log-linear models. In: Proc. of COLING-ACL, Companion Volume. pp. 787–794. Sydney (July 2006), http://cs.jhu.edu/~jason/papers/#acl06-risk
185. Smith, D.A., Eisner, J.: Dependency parsing by belief propagation. In: Proc. of EMNLP. pp. 145–156 (October 2008)
186. Smith, D.A., Smith, N.A.: Bilingual parsing with factored estimation: Using english to parse korean. In: EMNLP. pp. 49–56 (2004), http://www.aclweb.org/anthology/W04-3207
187. Smith, N., Vail, D., Lafferty, J.: Computationally efficient M-estimation of log-linear structure models. In: Proc. of ACL. pp. 752–759 (2007), http://acl.ldc.upenn.edu/P/P07/P07-1095.pdf
188. Smith, N.A., Eisner, J.: Annealing techniques for unsupervised statistical language learning. In: Proc. of ACL. pp. 486–493. Barcelona (July 2004), http://cs.jhu.edu/~jason/papers/#acl04-da
189. Smith, N.A., Eisner, J.: Contrastive estimation: Training log-linear models on unlabeled data. In: Proc. of ACL. pp. 354–362. Ann Arbor, Michigan (June 2005), http://cs.jhu.edu/~jason/papers/#acl05
190. Smith, N.A., Eisner, J.: Guiding unsupervised grammar induction using contrastive estimation. In: International Joint Conference on Artificial Intelligence (IJCAI) Workshop on Grammatical Inference Applications. pp. 73–82. Edinburgh (July 2005), http://cs.jhu.edu/~jason/papers/#gia05
191. Smith, N.A., Eisner, J.: Annealing structural bias in multilingual weighted grammar induction. In: Proc. of COLING-ACL. pp. 569–576. Sydney (July 2006), http://cs.jhu.edu/~jason/papers/#acl06-sa
192. Smith, N.A., Smith, D.A., Tromble, R.W.: Context-based morphological disambiguation with random fields. In: Proc. of HLT-EMNLP. pp. 475–482. ACL, Morristown, NJ, USA (2005)
193. Smith, N.A., Tromble, R.: Computational genomics: Sequence modeling (Fall 2004), http://www.cs.jhu.edu/~nasmith/600.403/, course offered at Johns Hopkins University
194. Steedman, M.: The syntactic process. MIT Press, Cambridge, MA, USA (2000)
195. World Wide Web Consortium (W3C) Semantic Web Education and Outreach (SWEO): Open data movement, http://esw.w3.org/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
196. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: Proc. of ICLP. Lecture Notes in Computer Science, vol. 225, pp. 84–98. Springer (1986)
197. Thornton, W.N.: Typed unification in Dyna: An exploration of the design space (Oct 2008), masters project report.
198. Topor, R.W.: Safe database queries with arithmetic relations. In: Proc. 14th Australian Computer Science Conference (1991), http://www.sci.gu.edu.au/~rwt/papers/ACSC91.ps
199. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proc.of OOPSLA. pp. 227–242. ACM, New York, NY, USA (1987), http://labs.oracle.com/self/papers/self-power.html
200. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. JACM 23(4), 733–742 (1976)
201. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)
202. Vassalos, V., Papakonstantinou, Y.: Describing and using the query capabilities of heterogeneous sources. In: Proc. of VLDB. pp. 256–265 (1997)
203. Vossen, P. (ed.): EuroWordNet: A Multilingual Database with Lexical Semantic Networks. Kluwer Academic Publishers, Norwell, MA, USA (1998)
204. Wang, M., Smith, N.A., Mitamura, T.: What is the Jeopardy model? a quasi-synchronous grammar for QA. In: Proc. of EMNLP-CoNLL. pp. 22–32 (2007), http://www.aclweb.org/anthology/D/D07/D07-1003
205. Warren, D.S.: Memoing for logic programs. Communications of the ACM 35(3), 93–111 (1992)
206. Wheeler, D.: SLOCcount, http://www.dwheeler.com/sloccount/
207. Williams, R., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Computation 1(2), 270–280 (1989), http://hawk.med.uottawa.ca/public/reprints/rtrl-nc-89.pdf
208. Winograd, T.: Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. Ph.D. thesis, Massachusetts Institute of Technology (Feb 1971), http://dspace.mit.edu/handle/1721.1/7095
209. XSB, http://xsb.sourceforge.net/
210. Yedidia, J.S., Freeman, W., Weiss, Y.: Understanding belief propagation and its generalizations. In: Exploring Artificial Intelligence in the New Millennium, chap. 8. Science & Technology Books (2003), http://www.merl.com/publications/TR2001-022/

211. Younger, D.H.: Recognition and parsing of context-free languages in time n^3. Information and Control 10(2), 189–208 (1967)
212. Zaniolo, C.: Key constraints and monotonic aggregates in deductive databases. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, pp. 109–134. No. 2408 in Notes in Computer Science, Springer (2002), http://www.cs.ucla.edu/~zaniolo/papers/book_for_bob.pdf
213. Zhang, M., Zhang, H., Li, H.: Convolution kernel over packed parse forest. In: Proc. of ACL. pp. 875–885. Uppsala, Sweden (July 2010), http://www.aclweb.org/anthology/P10-1090
214. Zhou, N.F., Sato, T.: Toward a high-performance system for symbolic and statistical modeling. In: Proc. of the IJCAI Workshop on Learning Statistical Models from Relational Data. pp. 153–159 (2003), http://sato-www.cs.titech.ac.jp/reference/Zhou-IJCAI03ws.pdf
215. Zhu, S.C., Mumford, D.: A stochastic grammar of images. Foundations and Trends in Computer Graphics and Vision 2(4), 259–362 (2006), http://www.stat.ucla.edu/~sczhu/papers/Grammar_quest.pdf
216. Zollmann, A., Venugopal, A.: Syntax augmented machine translation via chart parsing. In: StatMT '06: Proc. of the NAACL Workshop on Statistical Machine Translation. pp. 138–141. ACL, Morristown, NJ, USA (2006)
217. Zukowski, U., Freitag, B.: The deductive database system *LOLA*. In: Dix, J., Furbach, U., Nerode, A. (eds.) Logic Programming and Nonmonotonic Reasoning. pp. 375–386. LNAI 1265, Springer, Berlin (1997), http://www.im.uni-passau.de/publikationen/ZF97/ZF97.pdf