Dyna: Extending Datalog for Modern AI*

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. 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

* This chapter has been condensed for publication; the full version is available as [22]. This material is based on work supported by the National Science Foundation under Grants No. 0347822 and 0964681 to the first author, and by a graduate fellowship to the second author from the Human Language Technology Center of Excellence, Johns Hopkins University. We thank Wren N. G. Thornton and John Blatz for many stimulating discussions. We also thank Yanif Ahmad, Adam Teichert, Jason Smith, Nicholas Andrews, and Veselin Stoyanov for timely comments on the writing.

O. de Moor et al. (Eds.): Datalog 2010, LNCS 6702, pp. 181–220, 2011.
© Springer-Verlag Berlin Heidelberg 2011

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 and figures.

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 [8]? 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. 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. 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.

– 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 is likely to hurt performance without significantly easing implementation.

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**. Recall that a **deductive database** [11,56] 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.

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. 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. 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 an appendix to the full version [22].

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.

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 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*.[1] 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

[1] 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.

as general Dyna-to-Dyna program transformations [21,13]. 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 [23]. We are now considering a much larger space of execution strategies, supported by type and mode systems (cf. [53]). Again, the present paper has a different focus; but a high-level discussion of some of the many interesting issues can be found in the final sections of the full version [22].

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 an appendix to the full version [22].)

2.1 Background: Datalog

Datalog [10] 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**,[2] 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

[2] `A`, `B`, `C` can have *any* value. The full version of this paper [22] (both at this point and in §2.4) discusses optional type declarations that can aid correctness and efficiency.

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 [10]:

– **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))))`.

– **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. 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., [55,70]) 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:[3]

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

The sum over `C` is a kind of aggregation. The syntax for writing this in Datalog varies by dialect; as an example, [14] would write the above fact and rule (6) as

[3] 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.

```
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 [4,49]:

- **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. [38]). Thus, a Dyna program defines a partial function from items to values. Values are numeric in this example, but in general may be arbitrary ground terms.[4]

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.[5]

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

[4] Abstractly, the value could be regarded as an additional argument with a functional dependency; see the full version of this paper [22] for more discussion.

[5] 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.

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).[6]

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 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, `+`).[7]

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 [30,23,31] 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 (cf. Mercury [46]).[8] As we will see in §3.1, relaxing these restrictions is important to support our AI use cases.

- **Flatness:** We drop this requirement so that Dyna can work with lists and other nested terms and perform unbounded computations.[9] 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

[6] This language design choice naturally extends completion semantics [12]. One can still force a default `0` by adding the explicit rule `sibling(A,B) += 0` to (8). See the full version of this paper [22] for further discussion.

[7] See the full version of this paper [22] for more discussion of aggregation operators.

[8] 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.

[9] For example, in computational linguistics, a parser's hypotheses may be represented by arbitrarily deep terms that are subject to unification. See the full version of this paper [22] for discussion and references.

definitions, and to simplify certain source-to-source program transformations [21]. 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[10] 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[11] 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 [54] for a review and synthesis), which when it permits non-stratified programs at all, typically identifies their semantics with one [29] or more [44] "stable models" or the intersection thereof [63,37], 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 [58]) is single-source shortest paths,[12] 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).

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,

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

[11] A **model** (or interpretation) of a logic program P is a partial map $[\![\cdot]\!]$ from items to values. A **supported model** [4] is a fixpoint of the "immediate consequence" operator T_P associated with that program [62]. 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).

[12] See the full version of this paper [22] for why it is hard to stratify this program.

as in a functional language.[13] 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` (the evaluation mechanism is explained in the full version of this paper [22]). This notational convention is not deep, but to our knowledge, it has not been used before in logic programming languages.[14] 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 (for space reasons, references are in the full version of this paper [22]).

By contrast, Dyna is not specifically probabilistic. Why? Our full paper [22] lists a wide variety of other numeric and non-numeric objects that are commonly manipulated by AI programs. 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). We will see 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 [38,30,26]). 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.[15] Dyna, by contrast, is a more relaxed and general-purpose 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.

[13] 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.)

[14] With the exception of the hybrid functional-logic language Curry [17]. Curry is closer to functional programming than to Datalog. Its logical features focus on nondeterminism in lazy evaluation, and it does not have aggregation.

[15] Again, see the full paper for references.

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 [23] for details and pseudocode). The basic idea is to propagate updates from rule bodies to rule heads, until the values of all items converge.[16] 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.

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, as alluded to in §1.3.

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 Figures 7, 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:

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

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

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.[17]

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). }.    (13)
```

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 should 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.[18] 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                          (14)
f.piglets := three. % and exactly three piglets
```

[17] One could equivalently define `e = $load("pigpen")`, where the file `pigpen.dyna` consists of "`pigs += 100. pigs += piglets.`" or a compiled equivalent. Then `$load("pigpen")` will evaluate to ε (until the file changes). (Note: Reserved-word functors such as `$load` start with $, to avoid interference with user names of items.)

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

These rules are written as part of the definition of δ (the owner[19] of the new dynabase φ) and supply new aggregands `20` and `3` to φ's versions of `pigs` and `piglets`.

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 (14) into[20]

```
f = { pigs    += 100.
      pigs    += piglets.                                   (15)
      pigs    += 20.
      piglets := $owner.three. }    % where $owner refers to δ
```

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      (16)
g.piglets := offspring.         % who are piglets
```

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 (16), which state that there are 100 more pigs than piglets and $\frac{1}{3}$ as many piglets as pigs:

$$\delta.\texttt{three} = 3 \qquad \delta.\texttt{offspring} = \gamma.\texttt{pigs}/\delta.\texttt{three} \qquad (17)$$
$$\gamma.\texttt{pigs} = 100 + \gamma.\texttt{piglets} \qquad \gamma.\texttt{piglets} = \delta.\texttt{offspring}$$

Dynabases are connected to object-oriented programming. 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 an appendix to the full version [22].

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.

[19] 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 (14). See the full version of this paper [22] for further discussion of ownership.

[20] Fine points and formal semantics are covered in the full version of this paper [22].

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**.

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 obtain their input by transforming raw datasets (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 AI example 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).

Notice that line 3 of Figure 2 is a 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 [15]. A dot product like `score(Structure) += weight(Feature)*strength(Feature,Structure)`

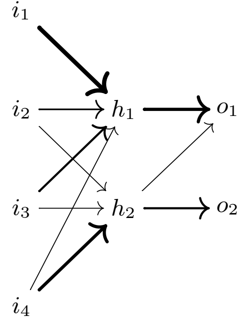


Fig. 1. A small acyclic neural network. The activation x_n at each node n is a nonlinear function f, such as a sigmoid or threshold function, of a weighted sum of activations at n's parent nodes: $x_n \stackrel{\text{def}}{=} f\left(\sum_{(n',n)\in E} x_{n'} w_{n',n}\right)$. The three layers shown here 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.

```
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.[21] 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.

[21] 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.
```

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`.

```
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 [50]
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). See the full version of this paper [22] for more detailed discussion of this code.

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.

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 [23]. 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 [23] computing gradients, via tape-based automatic differentiation in the reverse mode [32]. 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

```
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`).

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 (see the full version of this paper [22] for some references). 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. A simple example appears in our full paper.

Dyna also naturally handles some forms of default and non-monotonic reasoning [6], via `:=` rules like those in Figure 5. A related important use of default patterns in AI is "lifted inference" [61] in probabilistic settings like Markov Logic Networks [57], 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 [3,52].[22]

Some emerging KRR systems embrace statistics and draw probabilistic inferences rather than certain ones (again, see our full paper for references). 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 [64].

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

[22] Dyna rules illustrating this are given in the full version of this paper [22].

(§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).[23] 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; Dyna code for a simple random walk appears in the full version of this paper [22].

Dynamic Programming. Dyna began [23] as a language for dynamic programming (hence the name). The connection of dynamic programming to logic programming has been noted before (e.g., [33]). 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,[24] whether by backward chaining that lazily memoizes values in a table (as in XSB [65] and other tabled Prologs), or by forward chaining that eagerly accumulates values into a chart (as in §2.6 and the Dyna prototype [23]).

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      (18)
fib(1) := 1.
```

As a basic AI example, consider context-free parsing with a CKY-style algorithm [67]. The Dyna program in Figure 8 consists of 3 rules that directly and intuitively express how a parse tree is recursively built up 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" [60]: when a recently built or updated phrase pops off the agenda into the chart, it tries to combine with adjacent phrases in the chart.

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

[24] This support for reuse is already evident in our earlier examples, even though they would not traditionally be regarded as dynamic programming. 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 computing 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 [19]. 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 [48].

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.

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.[25] 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 [30].

This example also serves as a starting point for more complicated algorithms in syntactic natural-language parsing and syntax-directed translation.[26] The uses

[25] See the full version of this paper [22] for a detailed answer.

[26] The connection of these areas to deductive inference and logic programming has been well explored. See the full version of this paper [22] for discussion and references.

```
% 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 [66]. 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.

```
% 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.

of the Dyna prototype (listed in a section of [22]) have been mainly in this domain; see [21,23] for code examples. In natural language processing, active areas of research that make heavy use of parsing-like dynamic programs include machine translation, information extraction, and question answering.[27] There is a tremendous amount of experimentation with models and algorithms in these areas and in parsing itself. The machine vision community has also begun to explore recursive parsing of images [69,27]. 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 [13] 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*

[27] Again, see our full paper [22] for references.

```
% 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 13.
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.

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,[28] or that specify an infinitely large grammar.[29]

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 producing a single "best guess" word sequence, can often be made to produce a *probability distribution* over possible word sequences, which is more informative.

[28] E.g., one can transform a weighted context-free grammar into Chomsky Normal Form for use with Figure 11a, or coarsen a grammar for use as an A* heuristic [39].

[29] E.g., the non-range-restricted rule `rewrite(X/Z,X/Y,Y/Z).` encodes the infinitely many "composition" rules of combinatory categorial grammar [60], in which a complex nonterminal such as `s/(pp/np)` denotes an incomplete sentence (`s`) missing an incomplete prepositional phrase (`pp`) that is in turn missing a noun phrase (`np`).

```
% 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`.

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 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.[30]

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 [59,68].

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 [20]. 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 [28], it is surprisingly rare for papers to actually implement joint prediction or joint training, because of the extra design and engineering effort, particularly

[30] See the full version of this paper [22] for details.

```
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.        % extend the abstract parser ...
parse(K).input = doc.sentence(K).     % ... with some actual input

% 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 [5]. 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

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 [18], 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.

```
% 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.

A simple modification to the program will allow it to solve MAX-SAT-style problems using branch-and-bound.[31] In this case, a more breadth-first variant

[31] 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.
```

such as A* or iterative deepening will often outperform 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.[32]

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`. 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 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.[33] 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.[34]

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

[32] 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.

[33] 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.

[34] 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 [47,21,36] would change the derivation trees but not the derived trees.

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.[35] 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

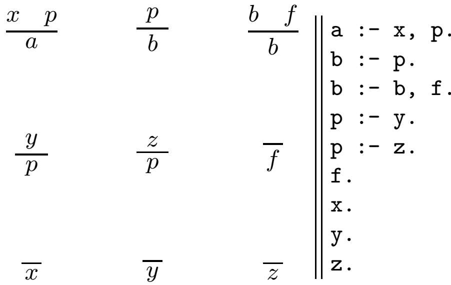
To get a view of what Dyna is doing, we now augment our proof forests to allow items (vertices) to have values (Figure 13e). This yields what we will call **generalized circuits**. Like an arithmetic (or boolean) circuit, a generalized circuit is a directed graph in which the value at each node α is 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 [30], 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.

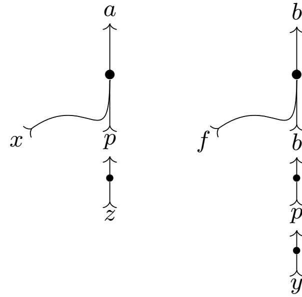
Our original prototype of Dyna was in fact limited to semiring-weighted deduction (which is indeed quite useful in parsing and related applications). Each program chose a single semiring $(\oplus, \otimes)$; each rule in the program had to multiply its antecedent values with $\otimes$ and aggregate these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

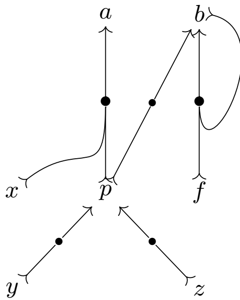
[35] Although a has only one incoming edge, it has two proof trees, one in which p is proved from y and the other (shown in Figure 13b) in which p is proved from z.



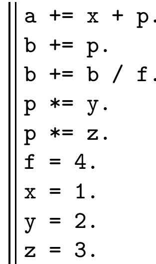
(a) A set of inference rules, and their encoding in Datalog. Axioms are written as inference rules with no antecedents.



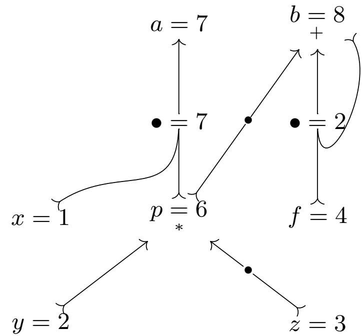
(b) Two proof trees using these rules. When an item is proved by an inference rule from 0 or more antecedent items, its vertex has an incoming hyperedge from its antecedents' vertices. Hyperedges with 0 antecedents (to f, x, y, z) are not drawn.



(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.



(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.

$$\frac{{}_i w_j \quad X \to w}{{}_i X_j} \qquad \frac{{}_i Y_j \quad {}_j Z_k \quad X \to Y\ Z}{{}_i X_k}$$

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).

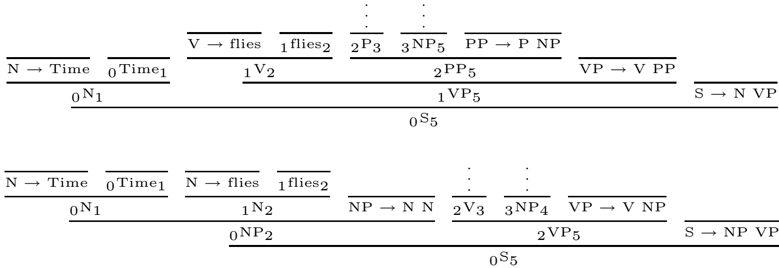


Fig. 15. Two example proofs that "Time flies like an arrow." is an English sentence, using the rules in Figure 14. This is traditional notation, but the hypergraphs of Figure 13 are more flexible because they would be able to show reuse of subgoals within a single proof, as well as making it possible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

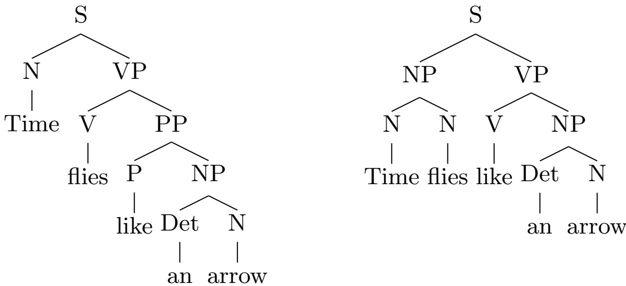


Fig. 16. Two example parse trees of the sentence "Time flies like an arrow" [40]. These are isomorphic to the proofs in Figure 15 (upside down) and correspond to different meanings of the sentence. The first conveys information about how time passes; the second tree says that flies of a certain species ("time flies") are fond of an arrow.

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 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.[36] 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.

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).[37] So one cannot formulate $x^3 + y^3 = z^3$ 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.

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

[36] 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.

[37] As mentioned earlier, this generalizes the completion semantics of [12], which treats a logic program as defining each boolean item with an "if and only if" constraint.

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. The full version of this paper [22] gives a more detailed argument, with many citations as well an informal survey of current code and data size.

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.

Knowledge bases and information integration pose additional problems of scale. As statistical methods gain popularity in other computational fields, the large-data problem spreads. 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.

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.[38]

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,

[38] See the full version of this paper [22] for discussion of an example, and for the code sizes of some AI systems and toolkits.

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.

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. One would like pipelines (of the kind discussed in §3.1) to agree on a common high-quality output and common parameters, but this requires the ability for components to query one another or pass messages to one another [28]. Similarly, one may wish to combine the strengths of diverse AI systems that are attempting the same task [35]. A recently emerging theme, therefore, is the development of principled methods for coordinating the work of multiple combinatorial algorithms. See references in the full version of this paper [22].

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 [7].

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] and business data analytics [41,42].

Why does a declarative approach fit for AI as well? We believe the business of AI is deriving hypotheses and conclusions from data (as discussed in a section of the full version of this paper [22]). 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 [45]. We are deeply interested in systematizing existing tricks of the trade and making them reusable across problems,[39] as well as pushing in new directions (§1.3).

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, as discussed in a section of the full version of this paper [22].

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, 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.

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 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, [13] shows how to combine two Dyna programs into a product-of-experts model.

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

[39] See additional discussion in the full version of this paper [22].

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,[40] 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 [34], and to "self-adjusting computation" [1] 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 [51].

***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

[40] Additional details may be found in a section of the full version of this paper [22].

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 a section of the full version of this paper [22]. Applications include business intelligence (e.g., LogicBlox [41]); stream processing for algorithmic equities trading (e.g., DBToaster [2]); user interfaces (e.g., Dynasty [24] and Fruit [16]); declarative animation (e.g., Fran [25]); query planners and optimizers (see the discussion in the full paper); and even (incremental) compilers [9].

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.

5 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 [23] 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.

Note: Throughout this book chapter, we have referred to additional material in the full version [22]. The full version also includes sections that sketch execution strategies; how dynabases interact with the world (the form of queries/results/updates, the dynabase API, mode checking, foreign dynabases, debugging); and formal semantics.

References

1. Acar, U.A., Ley-Wild, R.: Self-adjusting computation with Delta ML. In: Koopman, P.W.M., Plasmeijer, R., Swierstra, S.D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 1–38. Springer, Heidelberg (2009)
2. 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)
3. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFST: A general and efficient weighted finite-state transducer library. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 11–23. Springer, Heidelberg (2007)
4. 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, ch. 2. Morgan Kaufmann, San Francisco (1988)
5. Berg-Kirkpatrick, T., Bouchard-Côté, A., DeNero, J., Klein, D.: Painless unsupervised learning with features. In: Proc. of NAACL, pp. 582–590. ACL (2010)
6. Bidoit, N., Hull, R.: Minimalism, justification and non-monotonicity in deductive databases. Journal of Computer and System Sciences 38(2), 290–325 (1989)
7. 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, SETQA-NLP 2008, pp. 5–13. ACL (2008)
8. Brodie, M.L.: Future Intelligent Information Systems: AI and Database Technologies Working Together. Morgan Kaufmann, San Francisco (1988)
9. Burstall, R.M., Collins, J.S., Popplestone, R.J.: Programming in POP-2. Edinburgh University Press, Edinburgh (1971)
10. 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)
11. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer, Heidelberg (1990)
12. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum, New York (1978)
13. Cohen, S.B., Simmons, R.J., Smith, N.A.: Products of weighted logic programs. Theory and Practice of Logic Programming (2010)
14. Cohen, S., Nutt, W., Serebrenik, A.: Algorithms for rewriting aggregate queries using views. In: Masunaga, Y., Thalheim, B., Štuller, J., Pokorný, J. (eds.) ADBIS 2000 and DASFAA 2000. LNCS, vol. 1884, pp. 65–78. Springer, Heidelberg (2000)
15. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995)
16. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: 2001 Haskell Workshop (2001)
17. The functional logic language Curry, `http://www.informatik.uni-kiel.de/~curry/`

18. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (1962)
19. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
20. Eisner, J.: Parameter estimation for probabilistic finite-state transducers. In: Proc. of ACL, pp. 1–8 (2002)
21. 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, Stanford (2007)
22. Eisner, J., Filardo, N.W.: Dyna: Extending Datalog for modern AI (full version). Tech. rep., Johns Hopkins University (2011); Extended version of the present paper, `http://dyna.org/Publications`
23. 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 (2005)
24. 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 (2006)
25. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming (1997)
26. Felzenszwalb, P.F., McAllester, D.: The generalized A* architecture. J. Artif. Int. Res. 29(1), 153–190 (2007)
27. 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, Cambridge (2009)
28. 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 (2005)
29. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of the 5th International Conference and Symposium Logic Programming, pp. 1070–1080 (1988)
30. Goodman, J.: Semiring parsing. Computational Linguistics 25(4), 573–605 (1999)
31. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proc. of PODS, pp. 31–40 (2007)
32. Griewank, A., Corliss, G. (eds.): Automatic Differentiation of Algorithms. SIAM, Philadelphia (1991)
33. Guo, H.-F., Gupta, G.: Simplifying dynamic programming via tabling. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 163–177. Springer, Heidelberg (2004)
34. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Eng. Bull. 18(2), 3–18 (1995)
35. Hinton, G.: Products of experts. In: Proc. of ICANN, vol. 1, pp. 1–6 (1999)
36. Johnson, M.: Transforming projective bilexical dependency grammars into efficiently-parsable CFGs with unfold-fold. In: Proc. of ACL, pp. 168–175 (2007)
37. Kemp, D.B., Stuckey, P.J.: Semantics of logic programs with aggregates. In: Proc. of the International Logic Programming Symposium, pp. 338–401 (1991)
38. Kifer, M., Subrahmanian, V.S.: Theory of generalized annotated logic programming and its applications. Journal of Logic Programming 12(4), 335–368 (1992)
39. Klein, D., Manning, C.D.: A* parsing: Fast exact Viterbi parse selection. In: Proc. of HLT-NAACL (2003)

40. Kline, M.: Mathematics in the modern world; readings from Scientific American. With introductions by Morris Kline. W.H. Freeman, San Francisco (1968)
41. LogicBlox: Datalog for enterprise applications: from industrial applications to research (2010), `http://www.logicblox.com/research/presentations/arefdatalog20.pdf`, presented by Molham Aref at Datalog 2.0 Workshop
42. LogicBlox: Modular and reusable Datalog (2010), `http://www.logicblox.com/research/presentations/morebloxdatalog20.pdf`, presented by Shan Shan Huang at Datalog 2.0 Workshop
43. 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)
44. 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, Heidelberg (1999)
45. McAllester, D.A.: On the complexity analysis of static analyses. J. ACM 49(4), 512–537 (2002)
46. The Mercury Project, `http://www.cs.mu.oz.au/research/mercury/index.html`
47. Minnen, G.: Magic for filter optimization in dynamic bottom-up processing. In: ACL, pp. 247–254 (1996)
48. Mohr, R., Henderson, T.: Arc and path consistency revised. Artificial Intelligence 28, 225–233 (1986)
49. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In: Proc. of VLDB, pp. 264–277 (1990)
50. Nádas, A.: On Turing's formula for word probabilities. IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP-33(6), 1414–1416 (1985)
51. Ngai, G., Florian, R.: Transformation-based learning in the fast lane. In: Proc. of NAACL-HLT (2001)
52. van Noord, G., Gerdemann, D.: Finite state transducers with predicates and identities. Grammars 4(3) (2001)
53. Overton, D.: Precise and Expressive Mode Systems for Typed Logic Programming Languages. Ph.D. thesis, University of Melbourne (2003)
54. Pelov, N.: Semantics of Logic Programs With Aggregates. Ph.D. thesis, Katholieke Universiteit Leuven (2004)
55. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The coral deductive system. The VLDB Journal 3(2), 161–210 (1994); Special Issue on Prototypes of Deductive Database Systems
56. Ramamohanarao, K.: Special issue on prototypes of deductive database systems. VLDB 3(2) (1994)
57. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62(1-2), 107–136 (2006)
58. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. In: Proc. of PODS, pp. 114–126 (1992)
59. Schmid, H., Rooth, M.: Parse forest computation of expected governors. In: Proc. of ACL (2001)
60. Shieber, S.M., Schabes, Y., Pereira, F.: Principles and implementation of deductive parsing. Journal of Logic Programming 24(1-2), 3–36 (1995)
61. Singla, P., Domingos, P.: Lifted first-order belief propagation. In: Proc. of AAAI, pp. 1094–1099. AAAI Press, Menlo Park (2008)
62. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. JACM 23(4), 733–742 (1976)

63. 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)
64. Williams, R., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Computation 1(2), 270–280 (1989)
65. XSB, `http://xsb.sourceforge.net/`
66. Yedidia, J.S., Freeman, W.T., Weiss, Y.: Understanding belief propagation and its generalizations. In: Exploring Artificial Intelligence in the New Millennium, ch. 8. Science & Technology Books (2003)
67. Younger, D.H.: Recognition and parsing of context-free languages in time n^3. Information and Control 10(2), 189–208 (1967)
68. Zhang, M., Zhang, H., Li, H.: Convolution kernel over packed parse forest. In: Proc. of ACL, pp. 875–885 (2010)
69. Zhu, S.C., Mumford, D.: A stochastic grammar of images. Foundations and Trends in Computer Graphics and Vision 2(4), 259–362 (2006)
70. Zukowski, U., Freitag, B.: The deductive database system *LOLA*. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS (LNAI), vol. 1265, pp. 375–386. Springer, Heidelberg (1997)