Appeared in *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004), Companion Volume*, Barcelona, July 2004.

Dyna: A Declarative Language for Implementing Dynamic Programs*

Jason Eisner and **Eric Goldlust** and **Noah A. Smith**
Department of Computer Science, Johns Hopkins University
Baltimore, MD 21218 U.S.A.
{jason,eerat,nasmith}@cs.jhu.edu

Abstract

We present the first version of a new declarative programming language. Dyna has many uses but was designed especially for rapid development of new statistical NLP systems. A Dyna program is a small set of equations, resembling Prolog inference rules, that specify the abstract structure of a dynamic programming algorithm. It compiles into efficient, portable, C++ classes that can be easily invoked from a larger application. By default, these classes run a generalization of agenda-based parsing, prioritizing the partial parses by some figure of merit. The classes can also perform an exact backward (outside) pass in the service of parameter training. The compiler already knows several implementation tricks, algorithmic transforms, and numerical optimization techniques. It will acquire more over time: we intend for it to *generalize* and *encapsulate* best practices, and serve as a testbed for new practices. Dyna is now being used for parsing, machine translation, morphological analysis, grammar induction, and finite-state modeling.

1 Introduction

Computational linguistics has become a more experimental science. One often uses real-world data to test one's formal models (grammatical, statistical, or both).

Unfortunately, as in other experimental sciences, testing each new hypothesis requires much tedious lab work: writing and tuning code until parameter estimation ("training") and inference over unknown variables ("decoding") are bug-free and tolerably fast. This is intensive work, given complex models or a large search space (as in modern statistical parsing and machine translation). It is a major effort to break into the field with a new system, and modifying existing systems—even in a *conceptually* simple way—can require significant reengineering.

Such "lab work" mainly consists of reusing or reinventing various dynamic programming architectures. We propose that it is time to jump up a level of abstraction. We offer a new programming language, Dyna, that allows one to quickly and easily specify a model's combinatorial structure. We also offer a compiler, `dynac`, that translates from Dyna into C++ classes. The compiler does all the tedious work of writing the training and decoding code. It is intended to do as good a job as a clever graduate student who already knows the tricks of the trade (and is willing to maintain hand-tuned C++).

* We would like to thank Joshua Goodman, David McAllester, and Paul Ruhlen for useful early discussions, and pioneer users Markus Dreyer, David Smith, and Roy Tromble for their feedback and input. This work was supported by NSF ITR grant IIS-0313193 to the first author, by a Fannie & John Hertz Foundation fellowship to the third author, and by ONR MURI grant N00014-01-1-0685. The views expressed are not necessarily endorsed by the sponsors.

2 A Basic Example: PCFG Parsing

We believe Dyna is a flexible and intuitive specification language for dynamic programs. Such a program specifies how to combine partial solutions until a complete solution is reached.

2.1 The Inside Algorithm, in Dyna

Fig. 1 shows a simple Dyna program that corresponds to the inside algorithm for PCFGs (i.e., the probabilistic generalization of CKY parsing). It may be regarded as a system of equations over an arbitrary number of unknowns, which have *structured names* such as constit(s,0,3). These unknowns are called **items**. They resemble variables in a C program, but we use **variable** instead to refer to the capitalized identifiers X, I, K, … in lines 2–4.[1]

At runtime, a user must provide an input sentence and grammar by **asserting** values for certain items. If the input is *John loves Mary*, the user should assert values of 1 for word(John,0,1), word(loves,1,2), word(Mary,2,3), and end(3). If the PCFG contains a rewrite rule np $\rightarrow$ Mary with probability $p(\text{Mary} \mid \text{np}) = 0.003$, the user should assert that rewrite(np,Mary) has value 0.003.

Given these base cases, the equations in Fig. 1 enable Dyna to deduce values for other items. The deduced value of constit(s,0,3) will be the inside probability $\beta_{\text{s}}(0,3)$,[2] and the deduced value of goal will be the total probability of all parses of the input.

Lines 2–4 are equational schemas that specify how to compute the value of items such as constit(s,0,3) from the values of other items. By using the summation operator +=, lines 2–3 jointly say that for any X, I, and K, constit(X,I,K) is defined by summation over the remaining variables, as $\sum_{\text{W}}$ rewrite(X,W)*word(W,I,K) + $\sum_{\text{Y,Z,J}}$ rewrite(X,Y,Z)*constit(Y,I,J)*constit(Z,J,K). For example, constit(s,0,3) is a sum of quantities such as rewrite(s,np,vp)*constit(np,0,1)*constit(vp,1,3).

2.2 The Execution Model

Dyna's declarative semantics state only that it will find values such that all the equations hold.[3] Our implementation's default strategy is to propagate updates from an equation's right-hand to its left-hand side, until the system converges. Thus, by default, Fig. 1 yields a bottom-up or data-driven parser.

[1] Much of our terminology (item, chart, agenda) is inherited from the parsing literature. Other terminology (variable, term, inference rule, antecedent/consequent, assert/retract, chaining) comes from logic programming. Dyna's syntax borrows from both Prolog and C.

[2] That is, the probability that s would stochastically rewrite to the first three words of the input. If this can happen in more than one way, the probability sums over multiple derivations.

[3] Thus, future versions of the compiler are free to mix any efficient strategies, even calling numerical equation solvers.

```
1. :- valtype(term, real).                                          % declares that all item values are real numbers
2. constit(X,I,K) += rewrite(X,W) * word(W,I,K).                    % a constituent is either a word ...
3. constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K). % ... or a combination of two adjacent subconstituents
4. goal += constit(s,0,N) * end(N).                                 % a parse is any s constituent that covers the input string
```

Figure 1: A probabilistic CKY parser written in Dyna.

Dyna may be seen as a new kind of tabled logic programming language in which theorems are not just proved, but carry values. This suggests some terminology. Lines 2–4 of Fig. 1 are called **inference rules**. The items on the right-hand side are **antecedents**, and the item on the left-hand side is their **consequent**. Assertions can be regarded as **axioms**. And the default strategy (unlike Prolog's) is **forward chaining** from the axioms, as in some theorem provers.

Suppose constit(verb,1,2) increases by Δ. Then the program in Fig. 1 must find all the instantiated rules that have constit(verb,1,2) as an antecedent, and must update their consequents. For example, since line 3 can be instantiated as constit(vp,1,3) += rewrite(vp,verb,np)***constit(verb,1,2)***constit(np,2,3), then constit(vp,1,3) must be increased by rewrite(vp,verb,np) * **Δ** * constit(np,2,3).

Line 3 actually requires infinitely many such updates, corresponding to all rule instantiations of the form constit(X,1,K) += rewrite(X,verb,Z)***constit(verb,1,2)***constit(Z,2,K).[4] However, most of these updates would have no effect. We only need to consider the finitely many instantiations where rewrite(X,verb,Z) and constit(Z,2,K) have nonzero values (because they have been asserted or updated in the past).

The compiled Dyna program rapidly computes this set of needed updates and adds them to a worklist of pending updates, the **agenda**. Updates from the agenda are processed in some prioritized order (which can strongly affect the speed of the program). When an update is carried out (e.g., constit(vp,1,3) is increased), any further updates that *it* triggers (e.g., to constit(s,0,3)) are placed back on the agenda in the same way. Multiple updates to the same item are consolidated on the agenda. This cascading update process begins with axiom assertions, which are treated like other updates.

2.3 Closely Related Algorithms

We now give some examples of variant algorithms.

Fig. 1 provides lattice parsing for free. Instead of being integer positions in an string, I, J and K can be symbols denoting states in a finite-state automaton. The code does not have to change, only the input. Axioms should now correspond to weighted lattice arcs, e.g., word(loves,q,r) with value $p(\text{portion of speech signal} \mid \text{loves})$.

To find the probability of the best parse instead of the total probability of all parses, simply change the value type: replace real with viterbi in line 1. If a and b are viterbi values, a+b is implemented as $\max(a, b)$.[5]

[4] As well as instantiations constit(X,I,2) += rewrite(X,Y, verb)*constit(Y,I,1)***constit(verb,1,2)**.

[5] Also, a*b is implemented as $a + b$, as viterbi values actually represent log probabilities (for speed and dynamic range).

Similarly, replacing real with boolean obtains an unweighted parser, in which a constituent is either derived (true value) or not (false value) Then a*b is implemented as $a \wedge b$, and a+b as $a \vee b$.

The Dyna programmer can declare the **agenda discipline**—i.e., the order in which updates are processed—to obtain variant algorithms. Although Dyna supports stack and queue (LIFO and FIFO) disciplines, its default is to use a priority queue prioritized by the size of the update. When parsing with real values, this quickly accumulates a good approximation of the inside probabilities, which permits heuristic **early stopping** before the agenda is empty. With viterbi values, it amounts to uniform-cost search for the best parse, and an item's value is guaranteed not to change once it is nonzero. Dyna will soon allow user-defined priority functions (themselves dynamic programs), which can greatly speed up parsing (Caraballo and Charniak, 1998; Klein and Manning, 2003).

2.4 Parameter Training

Dyna provides facilities for training parameters. For example, from Fig. 1, it automatically derives the inside-outside (EM) algorithm for training PCFGs.

How is this possible? Once the program of Fig. 1 has run, goal's value is the probability of the input sentence under the grammar. This is a continuous function of the axiom values, which correspond to PCFG parameters (e.g., the weight of rewrite(np,Mary)). The function could be written out explicitly as a sum of products of sums of products of ... of axiom values, with the details depending on the sentence and grammar.

Thus, Dyna can be regarded as computing a function $F(\vec{\theta})$, where $\vec{\theta}$ is a vector of axiom values and $F(\vec{\theta})$ is an objective function such as the probability of one's training data. In learning, one wishes to repeatedly adjust $\vec{\theta}$ so as to increase $F(\vec{\theta})$.

Dyna can be told to evaluate the gradient of the function with respect to the current parameters $\vec{\theta}$: e.g., if rewrite(vp,verb,np) were increased by ϵ, what would happen to goal? Then any gradient-based optimization method can be applied, using Dyna to evaluate both $F(\vec{\theta})$ and its gradient vector. Also, EM can be applied where appropriate, since it can be shown that EM's E counts can be derived from the gradient. Dyna's strategy for computing the gradient is automatic differentiation in the reverse mode (Griewank and Corliss, 1991), known in the neural network community as back-propagation.

Dyna comes with a constrained optimization module, DynaMITE,[6] that can locally optimize $F(\vec{\theta})$. At present, DynaMITE provides the conjugate gradient and variable metric methods, using the Toolkit for Advanced Optimization (Benson et al., 2000) together with a softmax

[6] DynaMITE = Dyna Module for Iterative Training and Estimation.

technique to enforce sum-to-one constraints. It supports maximum-entropy training and the EM algorithm.[7]

DynaMITE provides an object-oriented API that allows independent variation of such diverse elements of training as the model parameterization, optimization algorithm, smoothing techniques, priors, and datasets.

How about supervised or partly supervised training? The role of supervision is to permit some constituents to be built but not others (Pereira and Schabes, 1992). Lines 2–3 of Fig. 1 can simply be extended with an additional antecedent permitted(X,I,K), which must be either asserted or derived for constit(X,I,K) to be derived. In "soft" supervision, the permitted axioms may have values between 0 and 1.[8]

3 C++ Interface and Implementation

A Dyna program compiles to a set of portable C++ classes that manage the items and perform inference. These classes can be used in a larger C++ application.[9] This strategy keeps Dyna both small and convenient.

A C++ **chart** object supports the computation of item values and gradients. It keeps track of built items, their values, and their derivations, which form a proof forest. It also holds an ordered agenda of pending updates. Some built items may be "transient," meaning that they are not actually stored in the chart at the moment but will be transparently recomputed upon demand.

The Dyna compiler generates a hard-coded decision tree that analyzes the structure of each item popped from the agenda to decide which inference rules apply to it. To enable fast lookup of the other items that participate in these inference rules, it generates code to maintain appropriate indices on the chart.

Objects such as constit(vp,1,3) are called **terms** and may be recursively nested to any depth. (Items are just terms with values.) Dyna has a full first-order type system for terms, including primitive and disjunctive types, and permitting compile-time type inference. These types are compiled into C++ classes that support constructors and accessors, garbage-collection, subterm sharing (which may lead to *asymptotic* speedups, as in CCG parsing (Vijay-Shanker and Weir, 1990)), and interning.[10]

Dyna can import new primitive term types and value types from C++, as well as C++ functions to combine values and to user-define the weights of certain terms.

In the current implementation, every rule must have the restricted form c += $a_1 {*} a_2 {*} \cdots {*} a_k$ (where each a_i is an item or side condition and $(X, +, *)$ is a semiring of values). The design for Dyna's next version lifts this restriction to allow arbitrary, type-heterogeneous expressions on the right-hand side of an inference rule.[11]

4 Some Further Applications

Dyna is useful for any problem where partial hypotheses are assembled, or where consistency has to be maintained. It is already being used for parsing, syntax-based machine translation, morphological analysis, grammar induction, and finite-state operations.

It is well known that various parsing algorithms for CFG and other formalisms can be simply written in terms of inference rules. Fig. 2 renders one such example in Dyna, namely Earley's algorithm. Two features are worth noting: the use of recursively nested subterms such as lists, and the SIDE function, which evaluates to 1 or 0 according to whether its argument has a defined value yet. These **side conditions** are used here to prevent hypothesizing a constituent until there is a possible left context that calls for it.

Several recent syntax-directed statistical machine translation models are easy to build in Dyna. The simplest (Wu, 1997) uses constit(np,3,5,np,4,8) to denote a NP spanning positions 3–5 in the English string that is aligned with an NP spanning positions 4–8 in the Chinese string. When training or decoding, the hypotheses of better-trained monolingual parsers can provide either hard or soft partial supervision (section 2.4).

Dyna can manipulate finite-state transducers. For instance, the weighted arcs of the composed FST $M_1 \circ M_2$ can be deduced from the arcs of M_1 and M_2. Training $M_1 \circ M_2$ back-propagates to train the original weights in M_1 and M_2, as in (Eisner, 2002).

5 Speed and Code Size

One of our future priorities is speed. Comparing informally to the best hand-written C++ code we found online for inside-outside and Dijkstra's algorithms, Dyna (like Java) currently runs up to 5 times slower. We mainly understand the reasons (memory layout and overreliance on hashing) and are working actively to close the gap.[12]

Programmer time is also worth considering. Our inside-outside and Dijkstra's algorithms are each about 5 lines of Dyna code (plus a short C driver program), but were compared in the previous paragraph against efficient C++ implementations of 5500 and 900 lines.[13]

Our colleague Markus Dreyer, as his first Dyna program, decided to replicate the Collins parser (3400 lines of C). His implementation used under 40 lines of Dyna code, plus a 300-line C++ driver program that mostly dealt with I/O. One of us (Smith) has written substantially *more* complex Dyna programs (e.g., 56 types + 46 inference rules), enabling research that he would not have been willing to undertake in another language.

6 Related Work

This project tries to synthesize much folk wisdom. For NLP algorithms, three excellent longer papers have at-

[7] It will eventually offer additional methods, such as deterministic annealing, simulated annealing, and iterative scaling.

[8] Such item values are not probabilities. We are generally interested in log-linear models for parsing (Riezler et al., 2000) and other tasks.

[9] We are also now developing a default application: a visual debugger that allows a user to assert axioms and explore the proof forest created during inference.

[10] **Interned** values are hashed so that equal values are represented by equal pointers. It is very fast to compare and hash such representations.

[11] That will make Dyna useful for a wider variety of non-NLP algorithms (e.g., neural networks, constraint programming, clustering, and dynamic graph algorithms). However, it introduces several interesting design complications in the Dyna language and the implementation.

[12] Dyna spends most of its time manipulating hash tables and the priority queue. Inference is very fast because it is compiled.

[13] The code size comparisons are rough ones, because of mismatches between the programs being compared.

```
need(s,0) = 1.                                                          % begin by looking for an s that starts at position 0
constit(Nonterm/Needed,I,I) += SIDE(need(Nonterm,I)) * rewrite(Nonterm,Needed).         % traditional predict step
constit(Nonterm/Needed,I,K) += constit(Nonterm/cons(W,Needed),I,J) * word(W,J,K).       % traditional scan step
constit(Nonterm/Needed,I,K) += constit(Nonterm,cons(X,Needed),I,J) * constit(X/nil,J,K). % traditional complete step
goal += constit(s/nil,0,N) * end(N).                                    % we want a complete s constituent covering the sentence
need(Nonterm,J) += constit(_/cons(Nonterm,_),_,J).                      % Note: underscore matches anything (anonymous wildcard)
```

Figure 2: An Earley parser in Dyna. np/Needed is syntactic sugar for slash(np,Needed), which is the label of a partial np constituent that is still missing the *list* of subconstituents in Needed. In particular, np/nil is a complete np. (A list [n,pp] is encoded here as cons(n,cons(pp,nil)), although syntactic sugar for lists is also available.) need(np,3) is derived if some partial constituent seeks an np subconstituent starting at position 3. As usual, probabilistic, agenda-based lattice parsing comes for free, as does training.

tempted similar syntheses (though without covering variant search and storage strategies, which Dyna handles).

Shieber et al. (1995) (already noting that "many of the ideas we present are not new") showed that several *unweighted* parsing algorithms can be specified in terms of inference rules, and used Prolog to implement an agenda-based interpreter for such rules. McAllester (1999) made a similar case for static analysis algorithms, with a more rigorous discussion of indexing the chart.

Goodman (1999) generalized this line of work to *weighted* parsing, using rules of the form c += $a_1 * a_2 * \cdots * a_k$ (with side conditions allowed); he permitted values to fall in any semiring, and generalized the inside-outside algorithm. Our approach extends this to a wider variety of processing orders, and in particular shows how to use a prioritized agenda in the general case, using novel algorithms. We also extend to a wider class of formulas (e.g., neural networks).

The closest *implemented* work we have found is PRISM (Zhou and Sato, 2003), a kind of probabilistic Prolog that claims to be efficient (thanks to tabling, compilation, and years of development) and can handle a subset of the cases described by Goodman. It is interesting because it inherits expressive power from Prolog. On the other hand, its rigid probabilistic framework does not permit side conditions (Fig. 2), general semirings (Goodman), or general formulas (Dyna). PRISM does not currently seem practical for statistical NLP research: in CKY parsing tests, it was only able to handle a small fraction of the Penn Treebank ruleset (2400 high-probability rules) and tended to crash on sentences over 50 words. Dyna, by contrast, is designed for real-world use: it consistently parses over 10x faster than PRISM, scales to full-sized problems, and attempts to cover real-world necessities such as prioritization, bottom-up inference, pruning, smoothing, underflow avoidance, maxent, non-EM optimization techniques, etc.

7 Conclusions

Dyna is a declarative programming language for building efficient systems quickly. As a language, it is inspired by previous work in deductive parsing, adding weights in a particularly general way. Dyna's compiler has been designed with an eye toward low-level issues (indexing, structure-sharing, garbage collection, etc.) so that the cost of this abstraction is minimized.

The goal of Dyna is to facilitate experimentation: a new model or algorithm automatically gets a new memory layout, indexing, and training code. We hope this will lower the barrier to entry in the field, in both research and education. In Dyna we seek to exploit as many algorithmic tricks as we can, generalizing them to as many problems as possible on behalf of future Dyna programs. In turn the body of old programs can provide a unified testbed for new training and decoding techniques.

Our broader vision is to unify a problem's possible algorithms by automatically deriving all of them and their possible training procedures from a single high-level Dyna program, using source-to-source program transformations and compiler directives. We plan to choose automatically among these variants by machine learning over runs on typical data. This involves, for example, automatically learning a figure of merit to guide decoding.

The Dyna compiler, documentation, and examples can be found at `www.dyna.org`. The compiler is available under an open-source license. The commented C++ code that it generates is free to modify.

References

S. Benson, L. C. McInnes, and J. J. Moré. 2000. TAO users manual. Tech Rpt ANL/MCS-TM-242, Argonne Nat. Lab.

S. A. Caraballo, E. Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Comp. Ling.*, 24(2).

Jason Eisner. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proc. of ACL*.

Joshua Goodman. 1999. Semiring parsing. *Comp. Ling*, 25(4).

Andreas Griewank and George Corliss, editors. 1991. *Automatic Differentiation of Algorithms*. SIAM.

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

David McAllester. 1999. On the complexity analysis of static analyses. *6th Intl. Static Analysis Symposium*.

F. Pereira and Y. Schabes. 1992. Inside-outside reestimation from partially bracketed corpora. *Proc. of ACL*.

S. Riezler, D. Prescher, J. Kuhn, M. Johnson. 2000. Lexicalized stochastic modeling of constraint-based grammars using log-linear measures and EM training. *Proc. of ACL*.

Stuart M. Shieber, Yves Schabes, and Fernando Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*.

K. Vijay-Shanker and D. Weir. 1990. Polynomial-time parsing of combinatory categorial grammars. *Proc. of ACL*.

Dekai Wu. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23(3):377–404.

N.-F. Zhou and T. Sato. 2003. Toward a high-performance system for symbolic and statistical modeling. *IJCAI-03 Workshop on Learning Statistical Models from Relational Data*.