Dyna 2: Towards a General Weighted Logic Language

Nathaniel Wesley Filardo

October 12, 2017

Outline

Arithmetic Circuits
What?

Arithmetic circuits are *abstract data types* generalizing *key-value stores*.

- K-V interface:
  - *store*, *update*, and *retrieve* items (pair of key and value).
- Circuit interface:
  - store, update, and retrieve *input* items.
  - query *derived* items' values (computed from input).

Arithmetic Circuits

Why care about circuits?

Pervasive! Can describe:

- data structures' interfaces
- database interface
- database internal data structures
- **Statistical AI systems** interfaces

Powerful abstraction:

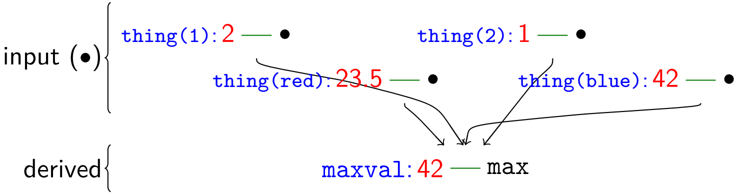
- Kowalski's observation: "Algorithm = Logic + Control"
- Circuit describes *logic*; a solver implements control.

Arithmetic Circuits

Why care about circuits?

Describe a priority queue as a circuit?

- Input: dynamic collection of keys with associated priorities
- Derived output: maximum of priorities

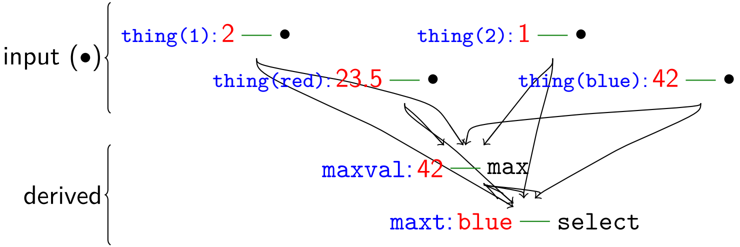


Arithmetic Circuits

Why care about circuits?

Describe a priority queue as a circuit?

- Input: dynamic collection of keys with associated priorities
- Derived output: maximum of priorities
- Further output: identify item with maximum priority

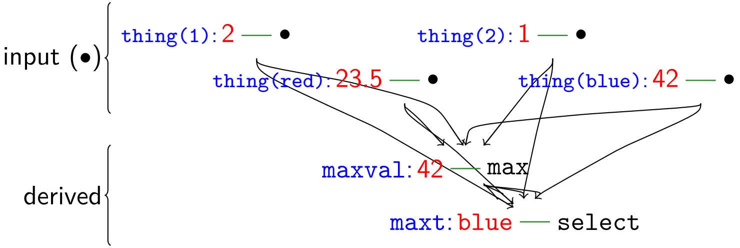


Arithmetic Circuits

Why care about circuits?

Describe a priority queue as a circuit?

- Input: dynamic collection of keys with associated priorities
- Derived output: maximum of priorities
- Further output: identify item with maximum priority



In Dyna:

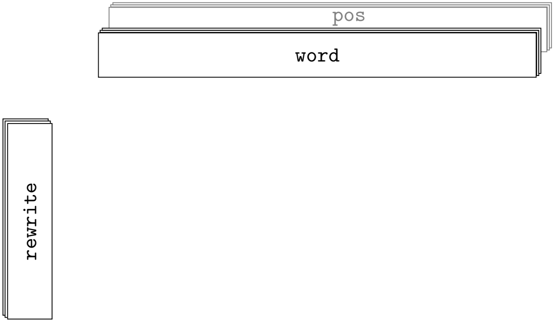
```
maxval max= thing(X).
maxt ?= X for maxval == thing(X).
```

Arithmetic Circuits

Why care about circuits?

More interesting example: (CNF) parser!

- Input: sentence (`words`)
- Input: grammar (binary `rewrites`, unary `pos` preterminal rules)
- Output: parse(s) (or statistics) for each span.

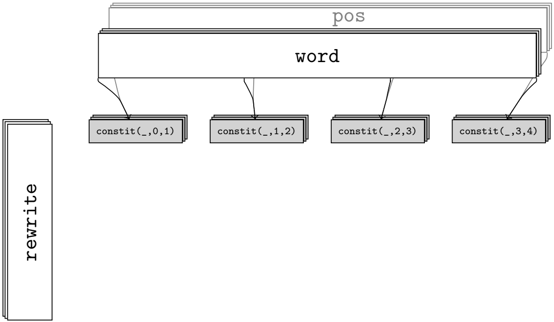


Arithmetic Circuits

Why care about circuits?

More interesting example: (CNF) parser!

- Input: sentence (`words`)
- Input: grammar (binary `rewrites`, unary `pos` preterminal rules)
- Output: parse(s) (or statistics) for each span.



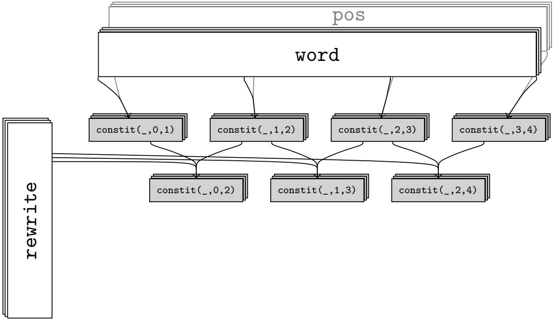
- Words and unary rules combine

Arithmetic Circuits

Why care about circuits?

More interesting example: (CNF) parser!

- Input: sentence (`words`)
- Input: grammar (binary `rewrites`, unary `pos` preterminal rules)
- Output: parse(s) (or statistics) for each span.



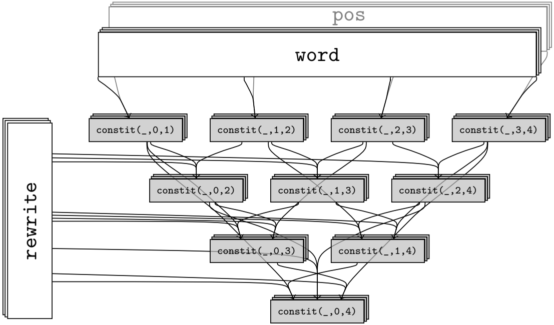
- Words and unary rules combine
- Adjacent spans combine

Arithmetic Circuits

Why care about circuits?

More interesting example: (CNF) parser!

- Input: sentence (`words`)
- Input: grammar (binary `rewrites`, unary `pos` preterminal rules)
- Output: parse(s) (or statistics) for each span.



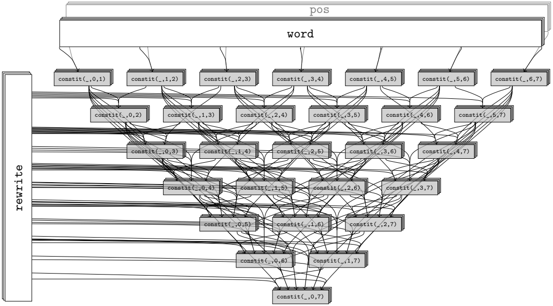
- Words and unary rules combine
- Adjacent spans combine
- And so on

Arithmetic Circuits

Why care about circuits?

More interesting example: (CNF) parser!

- Input: sentence (`words`)
- Input: grammar (binary `rewrites`, unary `pos` preterminal rules)
- Output: parse(s) (or statistics) for each span.



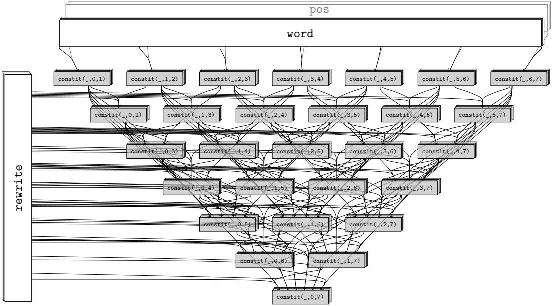
- Words and unary rules combine
- Adjacent spans combine
- And so on
- Circuit structure is data-dependent:
  - Longer sentence.
  - Regularity of sketch is misleading.

Arithmetic Circuits

Why care about circuits?

More interesting example: (CNF) parser!

- Input: sentence (`words`)
- Input: grammar (binary `rewrites`, unary `pos` preterminal rules)
- Output: parse(s) (or statistics) for each span.



- Words and unary rules combine
- Adjacent spans combine
- And so on
- Circuit structure is data-dependent:
  - Longer sentence.
  - Regularity of sketch is misleading.

```
constit(X,I,J) += word(W,I,J) * pos(W,X).
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```

The Dyna Project

Motivation

CLSP does lots of diverse research in AI. Repeated pain points:

- Systems are large! (Take “a while” to construct or modify.)

The Dyna Project

Motivation

As of 2011, some examples for scale:

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

The Dyna Project

Motivation

CLSP does lots of diverse research in AI. Repeated pain points:

- Systems are large! (Take “a while” to construct or modify.)
- Systems are fast from specialized hand-tuning.
  - Extensions break assumptions made in hand-tuning.
  - Even *toolkits* can be hard to take in new directions.

The Dyna Project

Motivation

CLSP does lots of diverse research in AI. Repeated pain points:

- Systems are large! (Take “a while” to construct or modify.)
- Systems are fast from specialized hand-tuning.
  - Extensions break assumptions made in hand-tuning.
  - Even *toolkits* can be hard to take in new directions.
- Lots of code and data out there!
  - Systems are hard to integrate.
  - Lots of data formats (and quadratically many Perl scripts).

The Dyna Project

Motivation

Especially frustrating, because

- AI systems' cores are *circuits*!
    - Behavior specified by a handful of equations.
    - Given a series of facts (input data).
    - Queried on results of applying equations.

The Dyna Project

Motivation

Especially frustrating, because

- AI systems' cores are *circuits*!
  - Behavior specified by a handful of equations.
  - Given a series of facts (input data).
  - Queried on results of applying equations.
- it is as if we are building
  - databases before DBMS and SQL.
  - file processing before regexps / parser generators.

The Dyna Project
Motivation

For scale, some example Dyna 2 program sizes:

Lines	**Program**
2-3	Dijkstra's shortest-path algorithm
4	Feed-forward neural network
11	Bigram language model with Good-Turing backoff smoothing
6	Arc-consistency constraint propagation
+6	With backtracking search
+6	With branch-and-bound
6	Loopy belief propagation
3	Probabilistic context-free parsing
+3	Earley's algorithm
+7	Conditional log-linear model of grammar weights (toy example)
+10	Coarse-to-fine A^* parsing
4	Value computation in a Markov Decision Process
5	Weighted edit distance
3	Markov chain Monte Carlo (toy example)

(See our 2011 position paper for most of these programs.)

The Dyna Project

Motivation

Additional historical precedent: Logic-based AI efforts give rise to Prolog in 1970-72.

- A *logic programming language*.
- Simplifies specification of logic-based AI.
    - Factors much of *control* aspect into language and runtime.

The Dyna Project

Motivation

Additional historical precedent: Logic-based AI efforts give rise to Prolog in 1970-72.

- A *logic programming language*.
- Simplifies specification of logic-based AI.
  - Factors much of *control* aspect into language and runtime.

1976: Fred Jelinek at IBM introduces information theory for speech recognition.

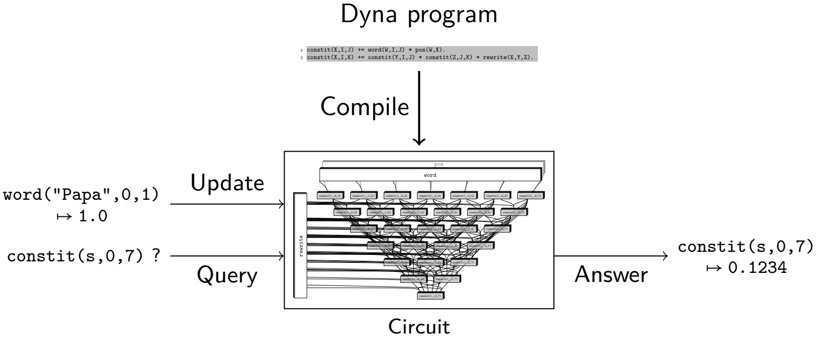
- Birth of *statistical AI* approach, now the dominant paradigm.

No single Prolog-like substrate has emerged for this new era.

- Prolog, even with answer subsumption, only handles a subset of needs.
- PRISM, Dyna 1: restricted expressiveness
- Problog: enforces particular probabilistic semantics
- TensorFlow: static circuit (but fast!), no updates
- (py)Torch, Dynet: *procedural* description of circuits, no updates

The Dyna Project

The Language



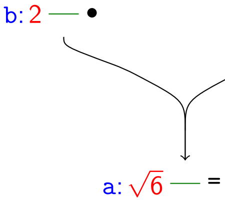
- Dyna is *narrowly scoped* to describe *data interdependence*.
  - It is a *domain-specific language* for circuit specification.
  - No user control of I/O.
  - No (explicit) reference cells, no threads, ...
  - Goal: let the *compiler* figure out how to make things fly.
- Generated circuit does not stand alone: requires a "driver program"
  - Driver intermediates all exchanges with the real world

The Dyna Project

The Language

Basic units of Dyna: *items* and *rules*.

- Rule `a = sqrt(b * c)` relates several items.
- `a` is the *head*, `sqrt(b * c)` the *body*.
- Not an *assignment*, but a live relationship.
- *Feed-forward*: specifies how to compute `a` from `b` and `c`.
  - No backward constraint: `b` defined elsewhere, used here.



The Dyna Project

The Language

Items have *structured names*:

- Like arrays, `f(3) = "hello"`
- Or maps, `edge("bal","was") = 35`
- Deep structure, too: `color(edge("bal","was")) = red`

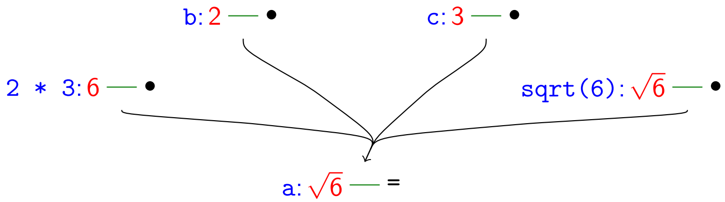
The Dyna Project

The Language

Items have *structured names*:

- Like arrays, `f(3) = "hello"`
- Or maps, `edge("bal","was") = 35`
- Deep structure, too: `color(edge("bal","was")) = red`

Used for arithmetic, too! `a = sqrt(b * c)`

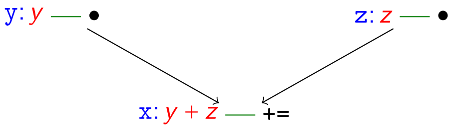


The Dyna Project

The Language

Aggregation combines contributions from several rules:

- Two rules with same head: `x += y` and `x += z` ($x = y + z$)

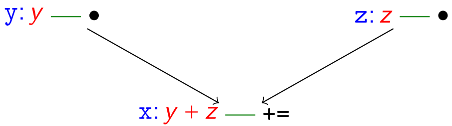


The Dyna Project

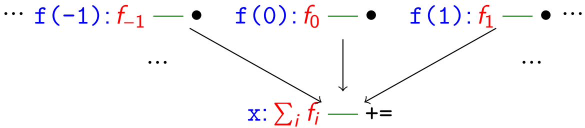
The Language

Aggregation combines contributions from several rules:

- Two rules with same head: `x += y` and `x += z` ($x = y + z$)



- A single rule with *variable(s)* in body: `x += f(I)` ($x = \sum_i f_i$) ("Fan-in" to `x`.)

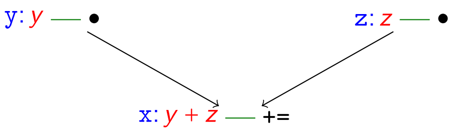


The Dyna Project

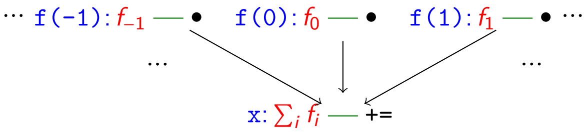
The Language

Aggregation combines contributions from several rules:

- Two rules with same head: `x += y` and `x += z` ($x = y + z$)



- A single rule with *variable(s)* in body: `x += f(I)` ($x = \sum_i f_i$)
  ("Fan-in" to `x`.)



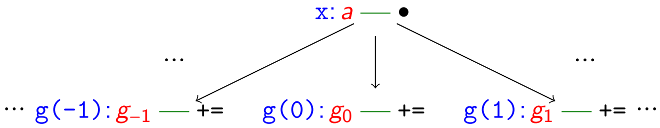
- Given all three rules, $x = y + z + \sum_i f_i$.

The Dyna Project

The Language

Rules are *schemata* for data relationships:

- Defaults (fan-out): `g(X) += a`.

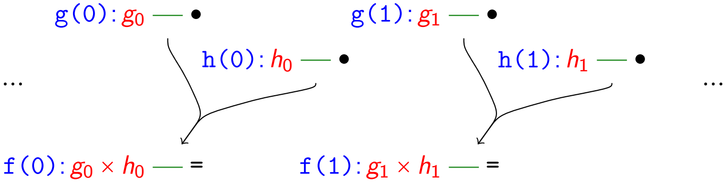


The Dyna Project

The Language

Rules are *schemata* for data relationships:

- Defaults (fan-out): `g(X) += a`.
- Pointwise products: `f(I) = g(I) * h(I)` ($f_i = g_i * h_i$).

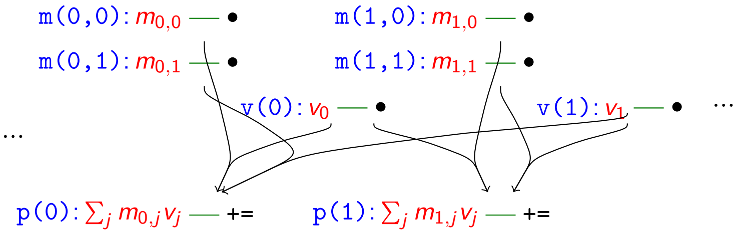


The Dyna Project

The Language

Rules are *schemata* for data relationships:

- Defaults (fan-out): `g(X) += a`.
- Pointwise products: `f(I) = g(I) * h(I)` ($f_i = g_i * h_i$).
- Matrix-vector products: `p(I) += m(I,J) * v(J)` ($p_i = \sum_j m_{ij} * v_j$).



The Dyna Project

The Challenge

Several challenges for bringing vision to reality:

- Need a good solver for Dyna programs (§2).
- Solver should handle as many programs as possible (§3, §4).
- Static analysis for checking programs to be well-defined & feasible (§5).
  - Also useful for optimization!
- Features for "programming in the large" (module system §6.2).

Entr'acte 1

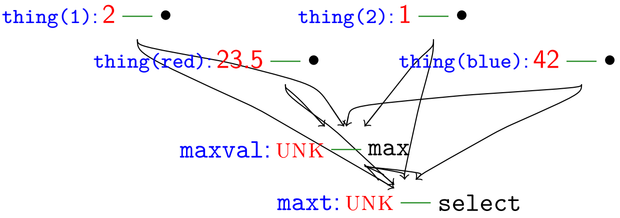
Before we continue, questions so far?

Solving Circuits

- Circuits just describe relation among values.
  - No hint of execution.
- Multiple options for how to execute!
  - Different space/time trade-offs.
  - Different performance under different workloads.
  - Want to support *as many as possible*!
    - (And let an optimizer select!)

Solving Circuits

Backward Chaining

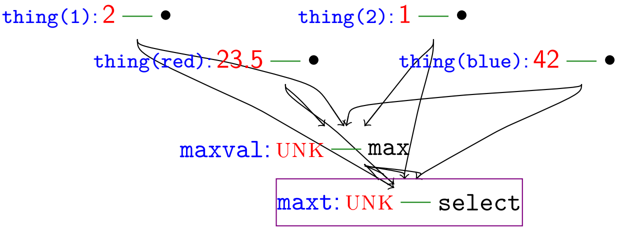


“Laziest” extreme: store values of input items, do nothing else until queried.

- Introduce “non-value” UNK for unknown values.
- Upon query, if item is UNK, must *compute from parents*.

Solving Circuits

Backward Chaining

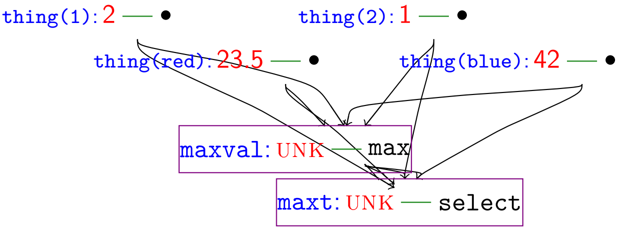


"Laziest" extreme: store values of input items, do nothing else until queried.

- Introduce "non-value" UNK for unknown values.
- Upon query, if item is UNK, must *compute from parents*.
  - Driver queries for value of `maxt`.

Solving Circuits

Backward Chaining

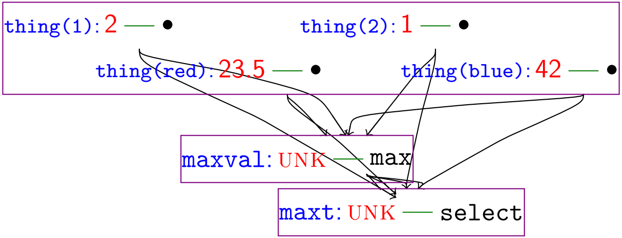


"Laziest" extreme: store values of input items, do nothing else until queried.

- Introduce "non-value" UNK for unknown values.
- Upon query, if item is UNK, must *compute from parents*.
  - Driver queries for value of `maxt`.
  - Internal query for value of `maxval` from `maxt`.

Solving Circuits

Backward Chaining

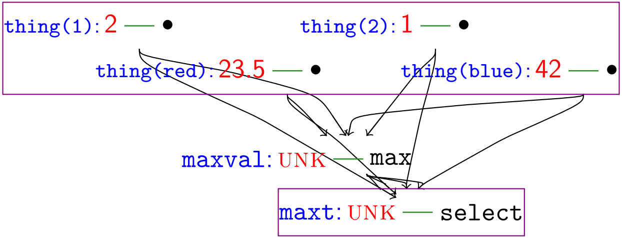


"Laziest" extreme: store values of input items, do nothing else until queried.

- Introduce "non-value" UNK for unknown values.
- Upon query, if item is UNK, must *compute from parents*.
  - Driver queries for value of `maxt`.
  - Internal query for value of `maxval` from `maxt`.
  - Internal query all values of `thing(X)` from `maxval`.

Solving Circuits

Backward Chaining

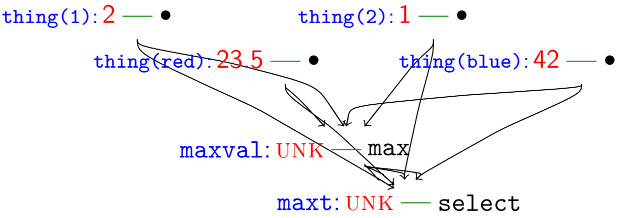


“Laziest” extreme: store values of input items, do nothing else until queried.

- Introduce “non-value” UNK for unknown values.
- Upon query, if item is UNK, must *compute from parents*.
  - Driver queries for value of `maxt`.
  - Internal query for value of `maxval` from `maxt`.
  - Internal query all values of `thing(X)` from `maxval`.
  - Internal query for `thing(X)` with value 42 from `maxt`.

Solving Circuits

Backward Chaining

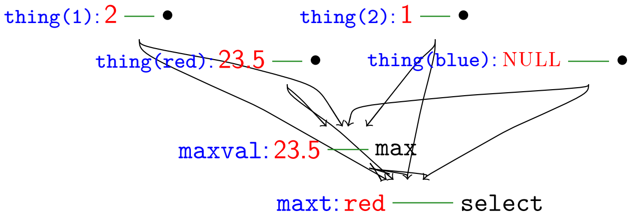


"Laziest" extreme: store values of input items, do nothing else until queried.

- Introduce "non-value" UNK for unknown values.
- Upon query, if item is UNK, must *compute from parents*.
  - Driver queries for value of `maxt`.
  - Internal query for value of `maxval` from `maxt`.
  - Internal query all values of `thing(X)` from `maxval`.
  - Internal query for `thing(X)` with value 42 from `maxt`.
  - Finish; return answer `blue`.

Solving Circuits

Forward Chaining

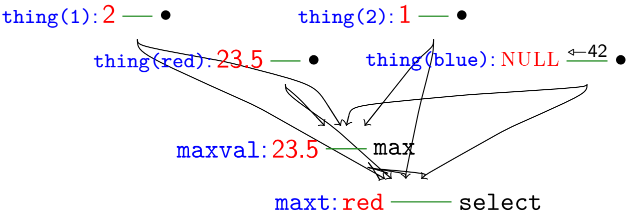


"Most eager" extreme:

- Define NULL for the aggregation of ∅ (roughly, "item not present")

Solving Circuits

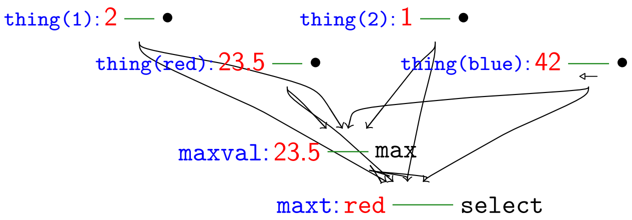
Forward Chaining



"Most eager" extreme:

- Define NULL for the aggregation of ∅ (roughly, "item not present")
- Upon *update*, must revise *descendants*:

Solving Circuits
Forward Chaining

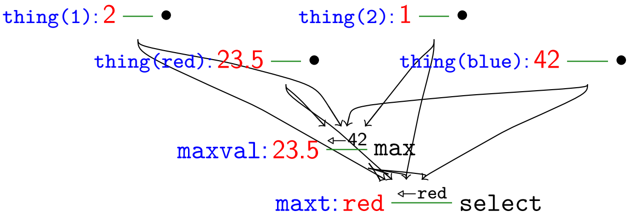


"Most eager" extreme:

- Define NULL for the aggregation of ∅ (roughly, "item not present")
- Upon *update*, must revise *descendants*:
  - apply update to item, and prepare to *notify* children.

Solving Circuits

Forward Chaining

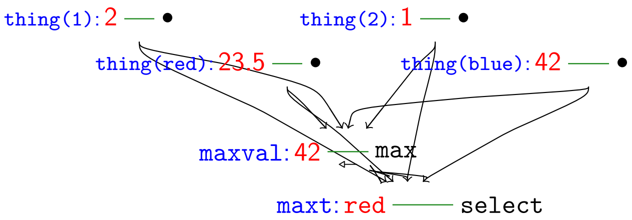


“Most eager” extreme:

- Define NULL for the aggregation of ∅ (roughly, “item not present”)
- Upon *update*, must revise *descendants*:
  - apply update to item, and prepare to *notify* children.
  - propagate notification to update children

Solving Circuits

Forward Chaining

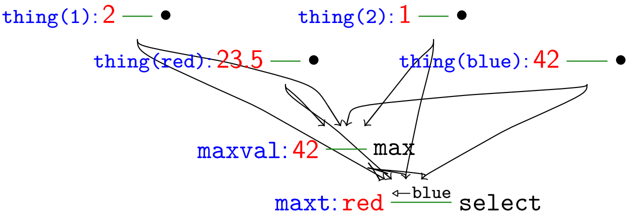


"Most eager" extreme:

- Define NULL for the aggregation of $\varnothing$ (roughly, "item not present")
- Upon *update*, must revise *descendants*:
  - apply update to item, and prepare to *notify* children.
  - propagate notification to update children
  - repeat until no work left

Solving Circuits

Forward Chaining

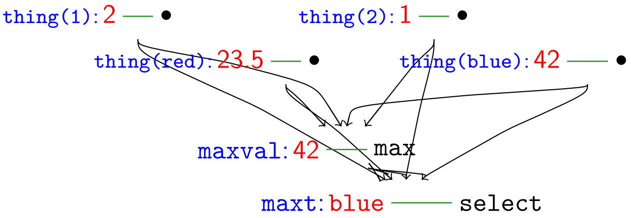


"Most eager" extreme:

- Define NULL for the aggregation of ∅ (roughly, "item not present")
- Upon *update*, must revise *descendants*:
  - apply update to item, and prepare to *notify* children.
  - propagate notification to update children
  - repeat until no work left

Solving Circuits

Forward Chaining



"Most eager" extreme:

- Define NULL for the aggregation of ∅ (roughly, "item not present")
- Upon *update*, must revise *descendants*:
  - apply update to item, and prepare to *notify* children.
  - propagate notification to update children
  - repeat until no work left
  - ready to be queried (or updated again)

Solving Circuits

Hybridized Chaining

- Forward and backward chaining typically viewed as *alternatives*.
- Have complementary jobs:
  - Backward chaining computes values for items missing memos.
  - Forward chaining refreshes (potentially) stale memos.
- Extremes of a spectrum:
  - Pure BC never creates memos: no refresh ever necessary.
  - Pure FC always memoizes: no recursive computation necessary.

Solving Circuits
Hybridized Chaining

§2.2 contains a hybridized algorithm for solving *finite*, acyclic circuits.

- finiteness: steps involving "all children" OK.
- acyclicity: backward-chaining never loops.
- many subtleties when forward-chaining through un-memoized items!

Several extensions considered:

- Increased efficiency via "obligation" (§2.2.4.3, §2.3.5)
- Parallel processing, viewing items as actors (§2.3)
- Large taxonomy of update and notification messages (§2.4)
- Cyclicity: on-demand conversion of backward to forward reasoning (§2.5)

Entr'acte 2

Circuitous questions before more programmatic concerns?

Circuits From Dyna

§3 to §5 address the challenge of deriving a circuit from a Dyna program.

- Dyna programs typically specify *infinite* circuits!
- Some programs must be rejected: might take infinite time to solve (§5.3)
- Can handle *piecewise-constant* infinite circuits (§3)
  - Given a runtime vocabulary for item *sets* (§4)

Circuits From Dyna

Rule Planning

CNF parser binary rule defines infinitely many edges in an infinite circuit.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```

- Literal implementation of algorithm from §2 will run forever.

Circuits From Dyna

Rule Planning

CNF parser binary rule defines infinitely many edges in an infinite circuit.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```

- Literal implementation of algorithm from §2 will run forever.
- Instead: find subset of "active" edges.
  - Merge *finite descriptions* of values for parent item sets.
  - Here: all `constit(_,_,_)`, `rewrite(_,_,_)`, and `_ * _` items.
  - If only finitely many such items with values, this would be especially easy.

Circuits From Dyna

Rule Planning

CNF parser binary rule defines infinitely many edges in an infinite circuit.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```

- Literal implementation of algorithm from §2 will run forever.
- Instead: find subset of "active" edges.
  - Merge *finite descriptions* of values for parent item sets.
  - Here: all `constit(_,_,_)`, `rewrite(_,_,_)`, and `_ * _` items.
  - If only finitely many such items with values, this would be especially easy.
  - But: Infinitely many `_ * _` items with values. Yet: still OK?

Circuits From Dyna

Rule Planning

CNF parser binary rule defines infinitely many edges in an infinite circuit.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```

- Literal implementation of algorithm from §2 will run forever.
- Instead: find subset of "active" edges.
  - Merge *finite descriptions* of values for parent item sets.
  - Here: all `constit(_,_,_)`, `rewrite(_,_,_)`, and `_ * _` items.
  - If only finitely many such items with values, this would be especially easy.
  - But: Infinitely many `_ * _` items with values. Yet: still OK?

Informally, still expect finite set of edges because:

- Given `constit(_,_,_)` and `rewrite(_,_,_)` items,
- only need *particular* `_ * _` items (e.g. `2 * 3`)

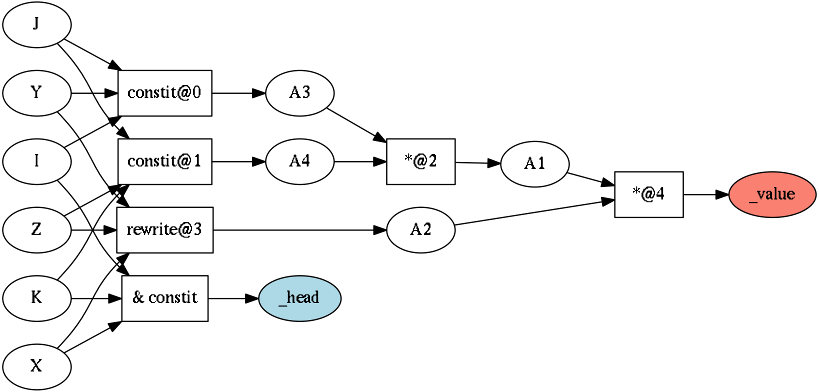
Circuits From Dyna

Rule Planning

Parser binary rule defines infinitely many edges in an infinite circuit.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```

Can think of this rule as having a *factor graph*:



This is *not* an arithmetic circuit. It is a useful formalism for considering how to find the *active subset* of edges created by this rule.

Circuits From Dyna

Rule Planning

Looking for active subset of edges:

- those for which *all parents* are non-NULL.
- want a *finite description* of these (infinitely many) edges.

Assume *procedures* that enumerate finite descriptions of subgoals' answers.

- Assume finitely many `word`s, so finitely enumerable.
- Multiplication only can when two of the three components are known.
  - $\{x \mid 2 * 3 = x\}$ or $\{x \mid x * 7 = 42\}$, but not $\{\langle x, y \rangle \mid x * y = 23.5\}$.
- `rewrite` might be of either flavor (input or derived).
- `constit` inductively finite.

Need to track *instantiation state*:

- "At runtime, this variable is still unknown."
- "At runtime, we will know the value of this variable."

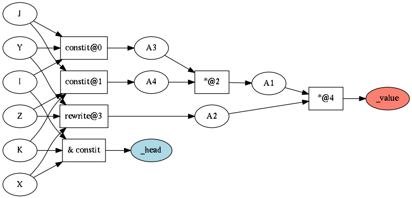
Circuits From Dyna

Rule Planning

Example: Looking for active subset of edges

- Given *known* head, e.g. `constit("s",0,7)`.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```



- Backward chain w/ head known

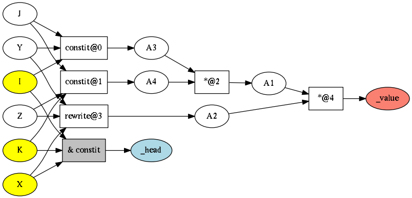
Circuits From Dyna

Rule Planning

Example: Looking for active subset of edges

- Given *known* head, e.g. `constit("s",0,7)`.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```



- Backward chain w/ head known
- Unpack head; `X`, `I`, `K` known

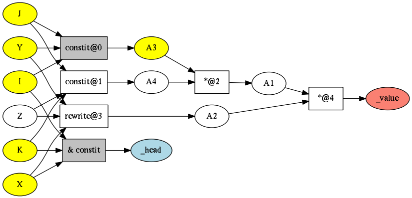
Circuits From Dyna

Rule Planning

Example: Looking for active subset of edges

- Given *known* head, e.g. `constit("s",0,7)`.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```



- Backward chain w/ head known
- Unpack head; `X`, `I`, `K` known
- Iterate `Y`, `J` from `constit(Y,I,J)`

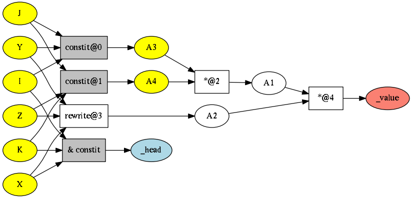
Circuits From Dyna

Rule Planning

Example: Looking for active subset of edges

- Given *known* head, e.g. `constit("s",0,7)`.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```



- Backward chain w/ head known
- Unpack head; `X`, `I`, `K` known
- Iterate `Y`, `J` from `constit(Y,I,J)`
- Iterate `Z` from `constit(Z,J,K)`

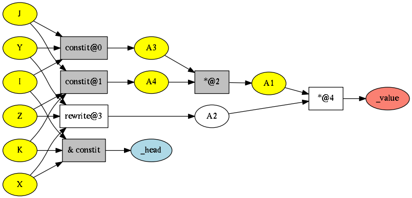
Circuits From Dyna

Rule Planning

Example: Looking for active subset of edges

- Given *known* head, e.g. `constit("s",0,7)`.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```



- Backward chain w/ head known
- Unpack head; `X`, `I`, `K` known
- Iterate `Y`, `J` from `constit(Y,I,J)`
- Iterate `Z` from `constit(Z,J,K)`
- Multiply

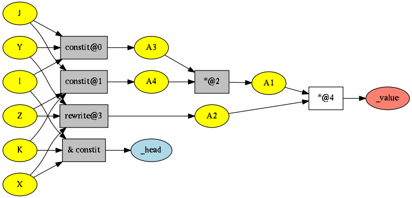
Circuits From Dyna

Rule Planning

Example: Looking for active subset of edges

- Given *known* head, e.g. `constit("s",0,7)`.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```



- Backward chain w/ head known
- Unpack head; `X`, `I`, `K` known
- Iterate `Y`, `J` from `constit(Y,I,J)`
- Iterate `Z` from `constit(Z,J,K)`
- Multiply
- Probe grammar at `rewrite(X,Y,Z)`

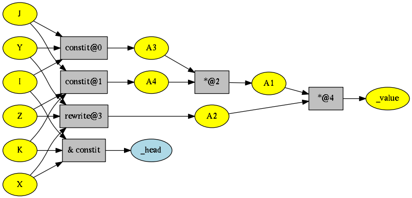
Circuits From Dyna

Rule Planning

Example: Looking for active subset of edges

- Given *known* head, e.g. `constit("s",0,7)`.

```
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K) * rewrite(X,Y,Z).
```



- Backward chain w/ head known
- Unpack head; `X`, `I`, `K` known
- Iterate `Y`, `J` from `constit(Y,I,J)`
- Iterate `Z` from `constit(Z,J,K)`
- Multiply
- Probe grammar at `rewrite(X,Y,Z)`
- Multiply

Circuits From Dyna

Rule Planning

This simple example well within reach of existing systems.

Thesis (§5.3) adds:

- Ability to track “partially known” structure.
  - Also within reach of existing systems
- *Type-aware* planning: variables' ranges are explicitly tracked.
- More versatile procedure selection (e.g., upcasts, case analysis)
- Result-dependent forks in plans.

Circuits From Dyna

Default Reasoning

Often, want to say "unless otherwise specified."

- *Sparse* arithmetic objects ("elements are zero, unless...")

```
f(X,Y) += 0.  % all cells
f(X,X) += 1.  % the diagonal
```

```
f(2,X) += 2.  % a column
f(2,2) += 4.  % a particular cell
```

- Default arcs in finite state machines:

```
trans(state(4), _  ) := state(6).  % every input but 'a'
trans(state(4), 'a') := state(5).
```

- Ontologies

```
fly(X : bird)     := true .  % absent other data...
fly(X : penguin) := false.  % but not these birds
fly(bigbird)      := false.  % nor that one in particular
```

- Lifted inference in MLN
  - Identify all nodes in a graph until reason to split

All of these have one very important thing in common:

- *Finitely many* rules with *constant* values.
- A *pointwise-constant* function of (in)finitely many things.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)          => 1.
r(X : nonneg int)   => 2.
r(-1)               => 3.
```

```
s(X : int)          => 1.
s(X : nonpos int)   => 4.
s(1)                => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)         => 1.
r(X : nonneg int)  => 2.
r(-1)              => 3.
```

```
s(X : int)         => 1.
s(X : nonpos int)  => 4.
s(1)               => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)          => 1.
r(X : nonneg int)   => 2.
r(-1)               => 3.
```

```
s(X : int)          => 1.
s(X : nonpos int)   => 4.
s(1)                => 5.
```

X	···	-3	-2	-1	0	1	2	3	···
r(X)	···	1	1	3	2	2	2	2	···
s(X)	···	4	4	4	4	5	1	1	···

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)          => 1.
r(X : nonneg int)   => 2.
r(-1)               => 3.
```

```
s(X : int)          => 1.
s(X : nonpos int)   => 4.
s(1)                => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯
p(X)	⋯	4	4	12	8	10	2	2	⋯

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means “most-specific wins”):

```
r(X : int)          => 1.
r(X : nonneg int)   => 2.
r(-1)               => 3.
```

```
s(X : int)          => 1.
s(X : nonpos int)   => 4.
s(1)                => 5.
```

X	···	-3	-2	-1	0	1	2	3	···
r(X)	···	1	1	3	2	2	2	2	···
s(X)	···	4	4	4	4	5	1	1	···
p(X)	···	4	4	12	8	10	2	2	···

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)          => 1.
r(X : nonneg int)   => 2.
r(-1)               => 3.
```

```
s(X : int)          => 1.
s(X : nonpos int)   => 4.
s(1)                => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯
p(X)	⋯	4	4	12	8	10	2	2	⋯

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means “most-specific wins”):

```
r(X : int)         => 1.
r(X : nonneg int)  => 2.
r(-1)              => 3.
```

```
s(X : int)         => 1.
s(X : nonpos int)  => 4.
s(1)               => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯
p(X)	⋯	4	4	12	8	10	2	2	⋯

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.
  - Mixing *defaults* gives rise to `p(0)`.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)        => 1.
r(X : nonneg int) => 2.
r(-1)             => 3.
```

```
s(X : int)        => 1.
s(X : nonpos int) => 4.
s(1)              => 5.
```

X	···	-3	-2	-1	0	1	2	3	···
r(X)	···	1	1	3	2	2	2	2	···
s(X)	···	4	4	4	4	5	1	1	···
p(X)	···	4	4	12	8	10	2	2	···

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.
  - Mixing *defaults* gives rise to `p(0)`.
  - `p(X : nonneg int)` and `p(X : nonpos int)` arise from other defaults.
    - *Do not* contribute to `p(-1)`, `p(0)`, `p(1)`; contributions *masked*.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)         => 1.
r(X : nonneg int)  => 2.
r(-1)              => 3.
```

```
s(X : int)         => 1.
s(X : nonpos int)  => 4.
s(1)               => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯
p(X)	⋯	4	4	12	8	10	2	2	⋯

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.
  - Mixing *defaults* gives rise to `p(0)`.
  - `p(X : nonneg int)` and `p(X : nonpos int)` arise from other defaults.
    - *Do not* contribute to `p(-1)`, `p(0)`, `p(1)`; contributions *masked*.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)        => 1.
r(X : nonneg int) => 2.
r(-1)             => 3.
```

```
s(X : int)        => 1.
s(X : nonpos int) => 4.
s(1)              => 5.
```

X	…	-3	-2	-1	0	1	2	3	…
r(X)	…	1	1	3	2	2	2	2	…
s(X)	…	4	4	4	4	5	1	1	…
p(X)	…	4	4	12	8	10	2	2	…

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.
  - Mixing *defaults* gives rise to `p(0)`.
  - `p(X : nonneg int)` and `p(X : nonpos int)` arise from other defaults.
    - *Do not* contribute to `p(-1)`, `p(0)`, `p(1)`; contributions *masked*.
  - `p(X : int)` arises as well, but *entirely masked*.

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means “most-specific wins”):

```
r(X : int)        => 1.
r(X : nonneg int) => 2.
r(-1)             => 3.
```

```
s(X : int)        => 1.
s(X : nonpos int) => 4.
s(1)              => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯
p(X)	⋯	4	4	12	8	10	2	2	⋯

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.
  - Mixing *defaults* gives rise to `p(0)`.
  - `p(X : nonneg int)` and `p(X : nonpos int)` arise from other defaults.
    - *Do not* contribute to `p(-1)`, `p(0)`, `p(1)`; contributions *masked*.
  - `p(X : int)` arises as well, but *entirely masked*.
    - So `p min= p(X)` gives `p => 2`, not `p => 1` (No `p(X)` with value 1!)

Circuits From Dyna

Conjoining Defaults

- Define two sparse vectors (`=>` means "most-specific wins"):

```
r(X : int)        => 1.
r(X : nonneg int) => 2.
r(-1)             => 3.
```

```
s(X : int)        => 1.
s(X : nonpos int) => 4.
s(1)              => 5.
```

X	⋯	-3	-2	-1	0	1	2	3	⋯
r(X)	⋯	1	1	3	2	2	2	2	⋯
s(X)	⋯	4	4	4	4	5	1	1	⋯
p(X)	⋯	4	4	12	8	10	2	2	⋯

- Define their pointwise product: `p(X) = r(X) * s(X).` Compute by *cross-product* of defaults.
  - `p(-1)` and `p(1)` come from the most-specific entries.
  - Mixing *defaults* gives rise to `p(0)`.
  - `p(X : nonneg int)` and `p(X : nonpos int)` arise from other defaults.
    - *Do not* contribute to `p(-1)`, `p(0)`, `p(1)`; contributions *masked*.
  - `p(X : int)` arises as well, but *entirely masked*.
    - So `p min= p(X)` gives `p => 2`, not `p => 1` (No `p(X)` with value 1!)
  - See thesis for more complex examples.

Circuits From Dyna

Aggregating Defaults

- Intra-rule aggregation is complicated!
  - Relies on set representation for computing *cardinality of set subtraction*.
- Cross-rule aggregation of defaults is relatively straightforward:
  - Rather like the simple *conjunction* on previous slide.
  - A cross-product construction, with set intersections at each.
- Too hard & not sufficiently interesting for talk; see thesis for details.

Circuits From Dyna

Interaction of Defaults with Planning

Defaults make planning more challenging:

- May only partially specify variables in rules.
  - May want *different loop orders* for defaults vs. overrides.
- Combination of defaults may result in *sets* of aggregands.
  - Despite having visited each subgoal.
  - Must ensure that we can manipulate the result (e.g., count it).

Piecewise constancy is, indeed, a constraint on the system:

- We will reject `f(X) += X` for default reasoning.
  - (But is OK for individual queries, like `f(3)`.)
- Is a sweet spot between expressiveness of program and complexity of solver.
- Generalizes existing system: all items' values NULL, unless otherwise specified.

What next?

This thesis: foundational work for Dyna 2.

§2 Flexible solver designs enable as many runtime strategies as possible.

§3 Default-based reasoning enlarges the space of acceptable programs.

§4 Discussion of representations of sets within solver.

§5 Static analysis of Dyna programs
  - Finds space of strategies for solver.

§6 Extensions, including declarative module system.

- (Much of the work is not *specific* to Dyna; applicable to other systems.)

Proof of concept work along the way:

- 2013 implementation of a solver for finite programs (no default reasoning).
- Used at Linguistic Institute summer program at University of Michigan.

What next?

Enough foundational theory done, serious building underway.

- Tim Vieira: Exploring machine learning for solver policies.
- Matthew Francis-Landau: aggressively-optimizing, JIT Dyna on Java.
- Dr. Vivek Sarkar and Farzad Khorasani: *parallel* and *GPU* runtime.

What next?

Thank you. Questions?

Proof Search

- Computations often amount to search for justification.
  - Reachability in a graph: edges forming a path.
  - Parsing a sentence: grammatical expansions.
  - (co-)NP complexity classes: witness.
  - Post Correspondence: sequence of tiles.
- These justifications can be recast as proofs in a logic.
  - Enter *logic programming*.
- More generally, we might want *quantifier alternation*: $\forall_a \exists_b \forall_c \cdots$

Proof Search

What's in a proof, anyway?

- Inference rules: "R proves a given proofs of b and c," written

$$\frac{b \quad c}{a}\ \mathrm{R}$$

- Axioms: inference rules without conditions: $\frac{}{f}$.
- Proof combines rules into a *tree*:
  - Given the rules

$$\frac{}{\mathsf{bal} \to \mathsf{was}} \quad \frac{}{\mathsf{phl} \to \mathsf{bal}} \quad \frac{}{\mathsf{nyc} \to \mathsf{phl}} \quad \frac{}{s \to^* s}\ \textsc{End} \quad \frac{s \to t \quad t \to^* u}{s \to^* u}\ \textsc{Step}$$

  - A proof of nyc $\to^*$ was is

$$\dfrac{\dfrac{}{\mathsf{nyc} \to \mathsf{phl}} \quad \dfrac{\dfrac{}{\mathsf{phl} \to \mathsf{bal}} \quad \dfrac{\dfrac{}{\mathsf{bal} \to \mathsf{was}} \quad \dfrac{}{\mathsf{was} \to^* \mathsf{was}}\ \textsc{End}}{\mathsf{bal} \to^* \mathsf{was}}\ \textsc{Step}}{\mathsf{phl} \to^* \mathsf{was}}\ \textsc{Step}}{\mathsf{nyc} \to^* \mathsf{was}}\ \textsc{Step}$$

Proof Search

Grammaticality of a sentence can be expressed as inference rules, too:

- Core rules:

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

- ${}_i w_j$: word w from position i to j.
- ${}_i X_k$: nonterminal X from position i to k.
- $X \rightarrow w$: word w has PoS (preterminal) X (e.g. $\overline{\text{Noun} \rightarrow \text{time}}$).
- $X \rightarrow YZ$: combine Y and Z to make X (e.g. $\overline{\text{PP} \rightarrow \text{P NP}}$).
- Goal: ${}_0\text{S}_k$ (for sentence of length k).

Proof Search

Core rules:

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

Consider the sentence "$\overline{{}_0\text{time}_1}\ \overline{{}_1\text{flies}_2}\ \overline{{}_2\text{like}_3}\ \overline{{}_3\text{an}_4}\ \overline{{}_4\text{arrow}_5}$." If we consider all ways of combining our inference rules (core and grammar), we find *several* proofs of grammaticality, which correspond to *readings*:

$$\dfrac{\dfrac{\overline{\text{N} \to \text{time}} \quad \overline{{}_0\text{time}_1}}{{}_0\text{N}_1} \quad \dfrac{\dfrac{\overline{\text{V} \to \text{flies}} \quad \overline{{}_1\text{flies}_2}}{{}_1\text{V}_2} \quad \dfrac{\dfrac{\vdots}{{}_2\text{P}_3} \quad \dfrac{\vdots}{{}_3\text{NP}_5} \quad \overline{\text{PP} \to \text{P NP}}}{{}_2\text{PP}_5} \quad \overline{\text{VP} \to \text{V PP}}}{{}_1\text{VP}_5} \quad \overline{\text{S} \to \text{N VP}}}{{}_0\text{S}_5}$$

Pertains to passage of time

$$\dfrac{\dfrac{\dfrac{\overline{\text{N} \to \text{time}} \quad \overline{{}_0\text{time}_1}}{{}_0\text{N}_1} \quad \dfrac{\overline{\text{N} \to \text{flies}} \quad \overline{{}_1\text{flies}_2}}{{}_1\text{N}_2} \quad \overline{\text{NP} \to \text{N N}}}{{}_0\text{NP}_2} \quad \dfrac{\dfrac{\vdots}{{}_2\text{V}_3} \quad \dfrac{\vdots}{{}_3\text{NP}_4} \quad \overline{\text{VP} \to \text{V NP}}}{{}_2\text{VP}_5} \quad \overline{\text{S} \to \text{NP VP}}}{{}_0\text{S}_5}$$

"Time flies," like "fruit flies."

Proof Search

Pure Prolog

Core rules:

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

Recast these in Prolog. Item names:

- `word(W,I,J)` for ${}_i w_j$
- `constit(X,I,K)` for ${}_i X_k$
- `pos(W,X)` for $X \to W$
- `rewrite(X,Y,Z)` for $X \to Y\ Z$

And rules:

```
constit(X,I,J) :- word(W,I,J), pos(W,X).
constit(X,I,K) :- constit(Y,I,J), constit(Z,J,K),
                  rewrite(X,Y,Z).
```

Equivalent formulation in more traditional logic (first rule):

$$\forall_{i,j,x}(\mathtt{c}_{x,i,j} \Leftarrow \exists_w(\mathtt{w}_{w,i,j} \wedge \mathtt{p}_{w,x})) \Leftrightarrow \underbrace{\forall_{i,j,w,x}(\mathtt{c}_{x,i,j} \vee \neg\mathtt{w}_{w,i,j} \vee \neg\mathtt{p}_{w,x})}_{\text{Horn clause}}$$

Proof Search

Boolean Circuits

Can think of Prolog program as specifying a hypergraph with:

- items as nodes, rules as hyperedges
- the value of a hyperedge is the AND ($\wedge$) of its tails
- the value of an item is the OR ($\vee$) of its incident hyperedges

(Have not discussed negation, but could add w/ more hyperedge types.)

A little algebra. Let $B = \{\mathtt{t}, \mathtt{f}\}$.

- AND: $x \wedge y = \mathtt{t}$ iff $x = y = \mathtt{t}$
- OR: $x \vee y = \mathtt{f}$ iff $x = y = \mathtt{f}$
- $\mathtt{t} \wedge x = x$
- $\mathtt{f} \vee x = x$
- Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

Proof Search

Dyna 1: Semirings and Horn Equations

A little algebra. Let $B = \{\mathtt{t}, \mathtt{f}\}$.

- AND: $x \wedge y = \mathtt{t}$ iff $x = y = \mathtt{t}$
- OR: $x \vee y = \mathtt{f}$ iff $x = y = \mathtt{f}$
- $\mathtt{t} \wedge x = x$
- $\mathtt{f} \vee x = x$
  - Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

$\langle B, \vee, \mathtt{f}, \wedge, \mathtt{t} \rangle$ is a *semiring* (*rig*). This kind of structure abounds!

- Numbers with + and $*$: $\langle \mathbb{R}, +, 0, *, 1 \rangle$.
  - $a * (b + c) = (a * b) + (a * c)$.
- "Tropical" semiring: $\langle \mathbb{R} \cup \{\infty\}, \min, \infty, +, 0 \rangle$.
  - $a + \min(b, c) = \min(a + b, a + c)$.
- Formal languages, probabilities, provenance, expectations, ...

Proof Search

Dyna 1: Semirings and Horn Equations

Consider again our Prolog parsing program:

```
constit(X,I,J) :- word(W,I,J), pos(W,X).
constit(X,I,K) :- constit(Y,I,J), constit(Z,J,K),
                  rewrite(X,Y,Z).
```

Can see that it uses OR and AND operations. That's *all* it does!

Could use *different* semiring addition and semiring product operations:

```
constit(X,I,J) += word(W,I,J) * pos(W,X).
constit(X,I,K) += constit(Y,I,J) * constit(Z,J,K)
                  * rewrite(X,Y,Z).
```

(Tarjan '81, "A Unified Approach to Path Problems")

Proof Search

Dyna 2: Generalized Expressions

Dyna 2 moves us beyond semirings:

- Different aggregators for different items.
- Generalized expressions in the body:
  - Mix weights and booleans: `a += 1 for f(X)`.
  - Values can become keys: `goal += constit("s",0,length)` *evaluates* length in place.
  -