2010年度
修士論文発表

階層グラフ書換えモデルを拡張した HyperLMNtal の実現

HyperLMNtal: an extension of a hierarchical graph rewriting language model

2011/02/04
情報理工学専攻 上田研究室
5109B023-6 小川誠司

研究の背景と目的

より大規模なシステムを
表現するために
実行性能の向上が進行中

LMNtal (Linked Multisets of Nodes transformation language)
階層グラフ構造の書換えに基づく言語モデル

高い表現力によって様々な
システムの振舞いをプログラム化

LMNtalコードを検査対象とする
システム検証機能を持つ処理系

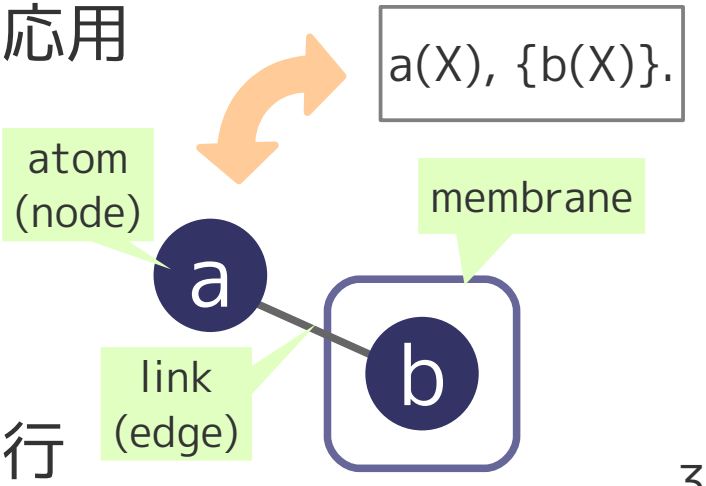
LMNtal はリンクによってデータ間の一対一の接続を表現
- ハイパーグラフなど、一対一以外の参照関係は表現困難
  - ➡ プログラム（データ構造）の複雑化
  - ➡ 冗長な計算処理による実行性能（時間計算量）の低下

非効率的な計算処理の発生を防ぐことで実行性能の向上を目指す

階層ハイパーグラフ書換えモデル (HyperLMNtal) へ拡張
- 実現困難であったハイパーグラフを簡潔に表現可能
- 冗長な計算処理を防ぐことで、理想的な計算量での実行を実現

LMNtal (Linked Multisets of Nodes transformation language)[1]

- 階層グラフ書換えに基づく並行プログラミング言語
- 計算処理：ルールによる階層グラフ構造の書換え
  - リンクによってアトム（基本データ構造）を一対一に接続
  - 膜　　：階層関係を表現、ルール適用範囲の局所化
  - ルール：*Head*（書換え前）:- *Guard*（条件節）| *Body*（書換え後）.
- 複雑なデータ構造を扱うプログラムを簡潔に記述可能
  - 多様な計算モデルを表現、検証分野への応用
  - 実行性能の向上が進行中
- 公開中の処理系
  - コンパイラから中間命令列を出力
  - ランタイムが中間命令列を入力として実行



[1] Ueda, K, “LMNtal as a Hierarchial Logic Programming Language”, Theoratical Computer Science, vol. 410, No. 46, pp. 4784-4800, 2009.

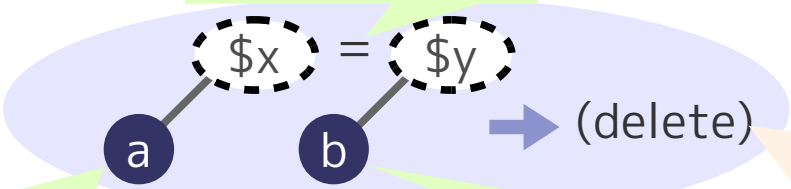
LMNtalのルール適用処理

- マッチング：ルール適用可能なグラフ構造の探索

```
a($x), b($y) :- $x = $y | .
```

a, bに等価な構造が
接続されている場合は
それらを消去する

(3) $x, $yが
等価な構造



(1)アトム名が"a"
リンク数が1

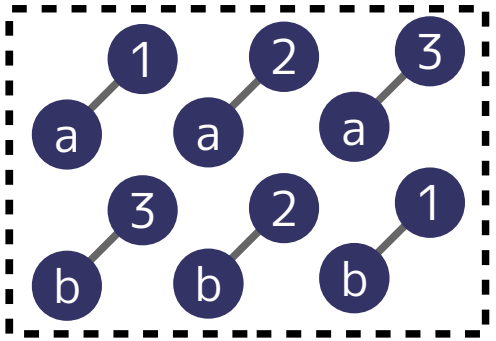
(2)アトム名が"b"
リンク数が1

条件を満たさない
部分グラフを取得
→マッチング失敗

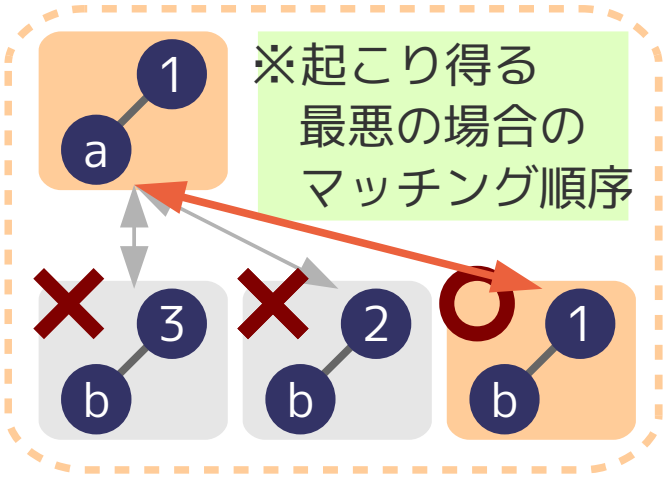
条件を満たす
部分グラフの探索を
繰り返す

マッチングの失敗回数の増加は著しい性能悪化を招く

非効率的なマッチングの防止は大きな課題



上記のルールを
繰り返し適用



※起こり得る
最悪の場合の
マッチング順序

best ：O(N)
worst ：$O(N^2)$
非効率的なマッチングにより
時間計算量が悪化している

N	10k	20k	40k	80k
best	0.03	0.06	0.10	0.22
worst	1.02	3.98	16.09	66.76

[sec]

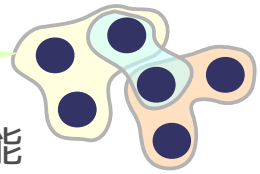
ハイパーグラフ構造の必要性[2]

リンクはアトムを一対一に接続

- LMNtalは一対一以外の参照関係(ハイパーグラフなど)を表現困難

*ハイパーグラフ
数学におけるグラフを一般化したもの
1本のエッジ(ハイパーエッジ)で任意個数のノードを接続可能

ハイパーエッジ

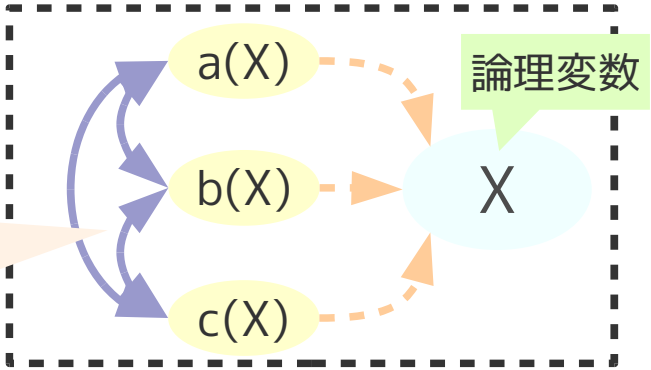


LMNtalの有用性を示すため
CHRプログラムは
LMNtalでエンコード可能
であることを示してきた[3]

- cf. Constraint Handling Rules (CHR) [4]
  - 実アプリへの利用など、実績のある多重集合書換え言語モデル
  - 論理変数によってデータ間の複雑な参照関係を簡潔に表現可能
  - 論理変数の参照関係を利用した効率的なマッチング手法が存在

a(X), b(X). c(X).

論理変数を介すことで
互いに参照関係に
=ハイパーグラフ



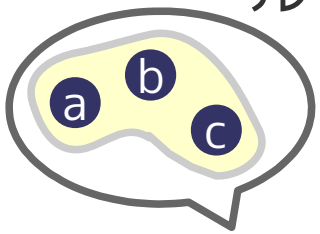
LMNtalでは膜やアトムを駆使して
ハイパーグラフを擬似的に表現可能

構造が複雑化
→非効率的なルールマッチングの発生
→記述量の増大
実用的な表現は不可能

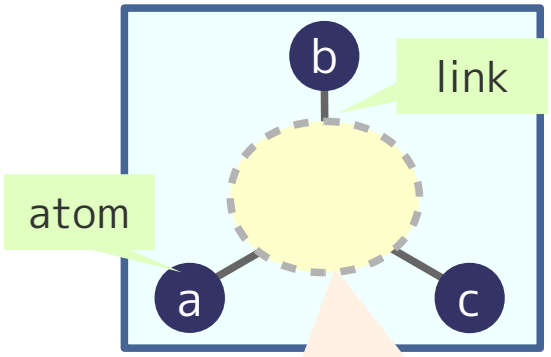
[2] 小川誠司, 上田和紀, "LMNtalグラフのハイパーグラフへの拡張", 第4回63号館ハイテクリサーチセンターシンポジウム, P10, 2010.
[3] 小川誠司, 綾野貴之, 上田和紀, "LMNtalを用いた状態空間探索", 第23回人工知能学会全国大会, 2H2-3, 2009.
[4] Thom Frühwirth, "Constraint Handling Rules", Cambridge University Press, The Ebinburgh Building, Cambrige CB2 8RU, UK、2009.

ハイパーリンク[5]の設計と実装(1/2)

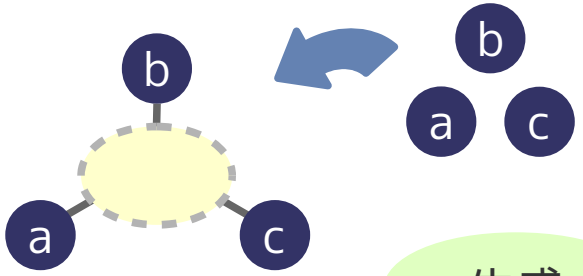
- (階層) ハイパーグラフを表現するための新データ構造
  - 概念：LMNtal のリンクを拡張（一般化）したもの
  - ルールでハイパーリンクを操作するための演算子などを追加



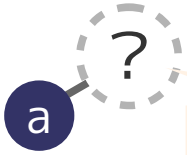
ハイパーリンクの
イメージ図



リンク同士を繋ぐ
ハブ構造によって
ハイパーリンクを表現



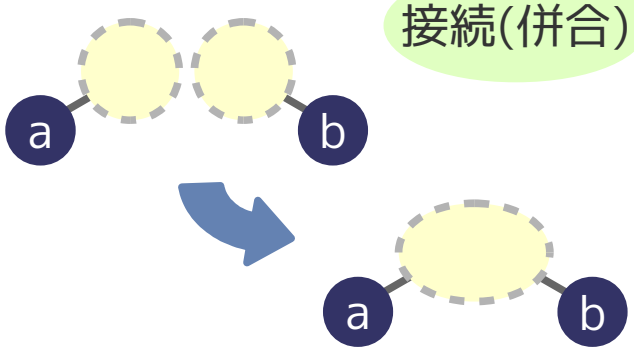
生成



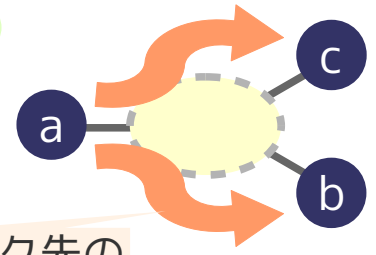
ハイパーリンクへの
接続であることを検査

型制約

接続(併合)



探索

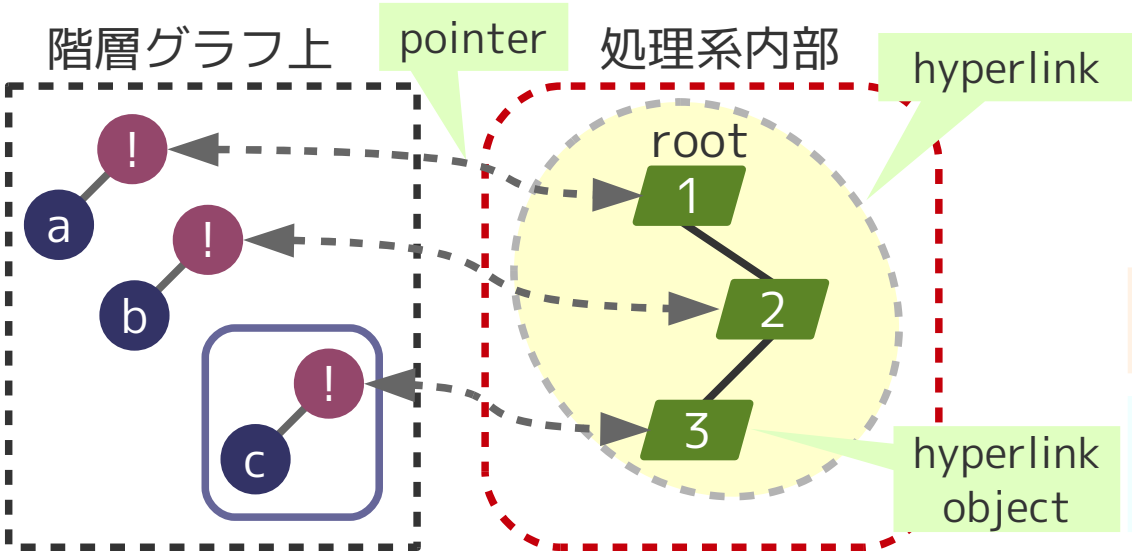


ハイパーリンク先の
構造や接続数を
取得可能

[5] 小川誠司, 目黒学, 上田和紀, “階層グラフ書換えモデルを拡張したHyperLMNtalの実現”, 第25回人工知能学会全国大会, 2011, 発表予定.

ハイパーリンクの設計と実装 (2/2)

- 2引数の"!"アトムをハイパーリンクアトムとして特別管理
  - 第二引数：ハイパーリンクオブジェクトへのポインタ
  - オブジェクトの集合をハイパーリンクとして管理
  - ハイパーリンク同士の接続 ↔ オブジェクト集合の併合
- Union-Find algorithm [6] を応用して併合関係を管理
  - "任意要素が所属する集合の解決", "集合の併合"を高速に実現



Union-Find algorithmが
サポートしていない
"要素の削除"などの処理も組み込んだ

コンパイラ、ランタイムからなる
LMNtal処理系（10万行以上）に追加実装

LMNtal処理系公開URL：
http://www.ueda.info.waseda.ac.jp/lmntal/

[6] Schrijvers, T. and Fruhwirth, T., "Optimal union-find in Constraint Handling Rules.", TPLP 6, 1-2, 213-224, 2006.

評価実験 (1/2)

論理変数を利用した最適化で
より理想的な計算量を実現している
例題が中心

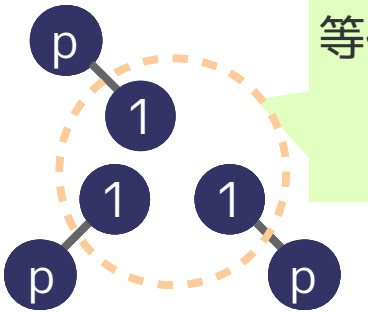
- CHR のベンチマーク問題をLMNtalで記述（計7題）
  - 閉路探索, 不等式制約ソルバ, RAM simulator, etc.
  - 実行性能（実行時間、計算量）

chr	：CHRによる記述
lmn-atom	：擬似的ハイパーグラフ（アトム版）
lmn-mem	：擬似的ハイパーリンク（膜版）
hyperlink	：ハイパーリンクを含むLMNtal

実験環境:
Intel Core2 Quad 2.60GHz
RAM 4GB
OS ubuntu 9.10

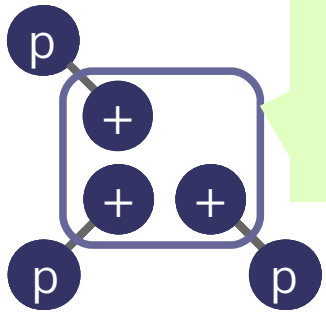
実用的な手法ではない

LMNtalによる擬似的なハイパーグラフ表現（2種類）



lmn-atom:
等価な構造を持たせることで
ハイパーグラフを表現
(実際に接続関係は無い)

冗長なマッチングにより
実行性能が悪化



lmn-mem:
膜で囲むことによって
ハイパーグラフを表現
(膜を介した接続関係が有る)

共有構造の生成に多大な計算量
構造が複雑化しコーディング困難

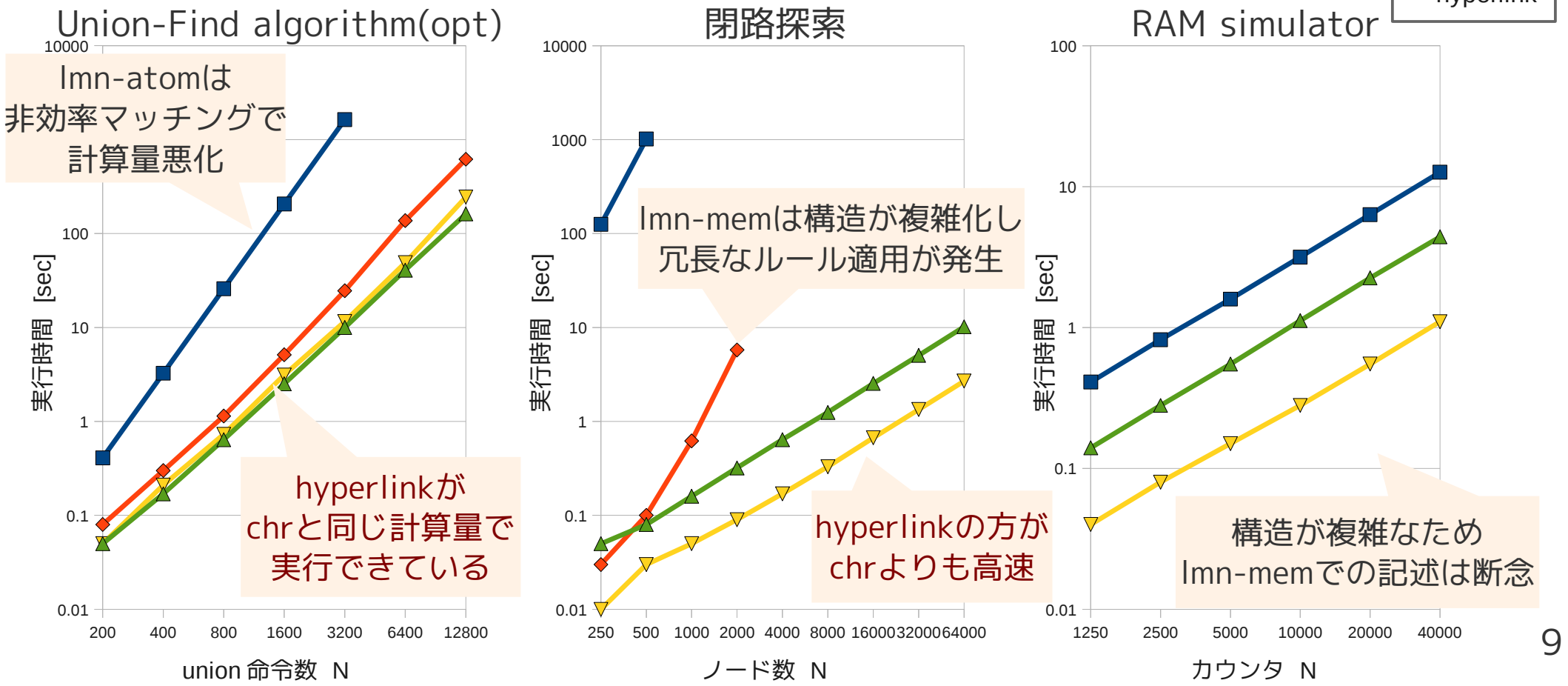
評価実験 (2/2)

全ての例題で従来のLMNtalよりも優れた実行性能（時間計算量、実行時間）を達成

全てchrと同じ計算量

実行速度はchrと比べて平均1.3倍速い

- 時間計算量の比較



まとめと今後の課題

- LMNtalを階層ハイパーグラフ書換えモデルへと拡張
  - 新たなデータ構造であるハイパーリンクの導入
    - ハイパーリンクを扱う演算子なども新たに追加
  - 制約多重集合書換え言語CHRをエンコード
    - 論理変数を自然に表現可能
    - 従来のLMNtalよりも実用的な実行性能を達成
- 今後の課題
  - CHR以外の計算モデルをエンコード
  - システム検証機能への適用
  - ハイパーリンクへのデータ束縛の検討
    - 論理変数への値の束縛を表現

ご清聴ありがとうございました

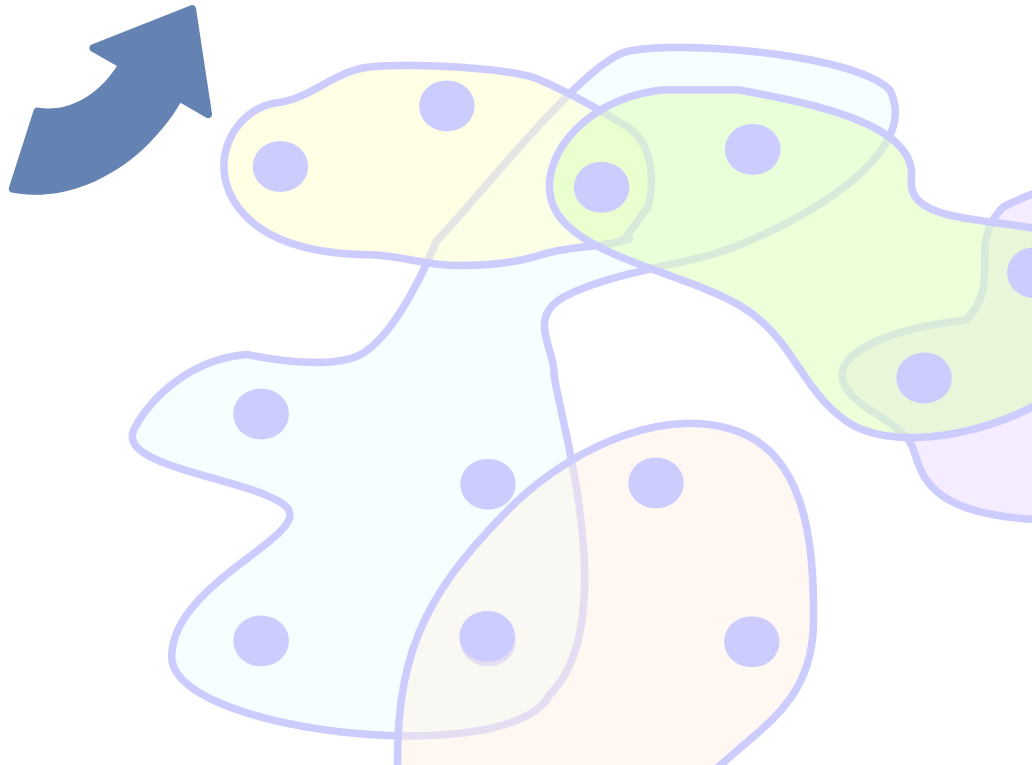
例題）不等式制約ソルバの記述例

従来のLMNtal (lmn-mem)

```
leq(X1,X2), {+X1,+X2,v(V),$p} :- int(V) | {$p}.
leq(X1,Y1), leq(Y2,X2),
{+X1,+X2,v(VX),$p}, {+Y1,+Y2,v(VY),$q} :-
  int(VX), int(VY) | {$p,$q,v(VX)}.
leq(X1,Y1), leq(X2,Y2),
{+X1,+X2,v(VX),$p}, {+Y1,+Y2,v(VY),$q} :-
  int(VX), int(VY) |
  leq(X1,Y1), {+X1,v(VX),$p}, {+Y1,v(VY),$q}.
leq(X1,Y1), leq(Y2,Z2),
{+X1,v(X),$p}, {+Y1,+Y2,v(Y), q}, {+Z2,v(Z),$r} :-
  uniq(X,Z), int(Y) |
  leq(X1,Y1), leq(Y2,Z2), leq(Z3,X3),
  {+X1,+X3,v(X),$p}, {+Y1,+Y2,v(Y),$q},
  {+Z2,+Z3,v(Z),$r}.
leq(X1,Y1), leq(Y2,Z2),
{+X1,+Z2,v(X),$p}, {+Y1,+Y2,v(Y),$q} :-
  uniq(X,X), int(Y) |
  leq(X1,Y1), leq(Y2,Z2), leq(Z3,X3),
  {+X1,+X3,+Z2,+Z3,v(X),$p}, {+Y1,+Y2,v(Y),$q}.
```

HyperLMNtal（ハイパーリンクを使用）

```
leq($x,$x) :- .
leq($x,$y), leq($y,$x) :- $x><$y.
leq($x,$y), leq($y,$z) \ :- uniq($x,$z) | leq($x,$z).
leq($x,$y) \ leq($x,$y) :- .
```



予備スライド

評価実験（記述性）

- 不等式制約ソルバの記述例

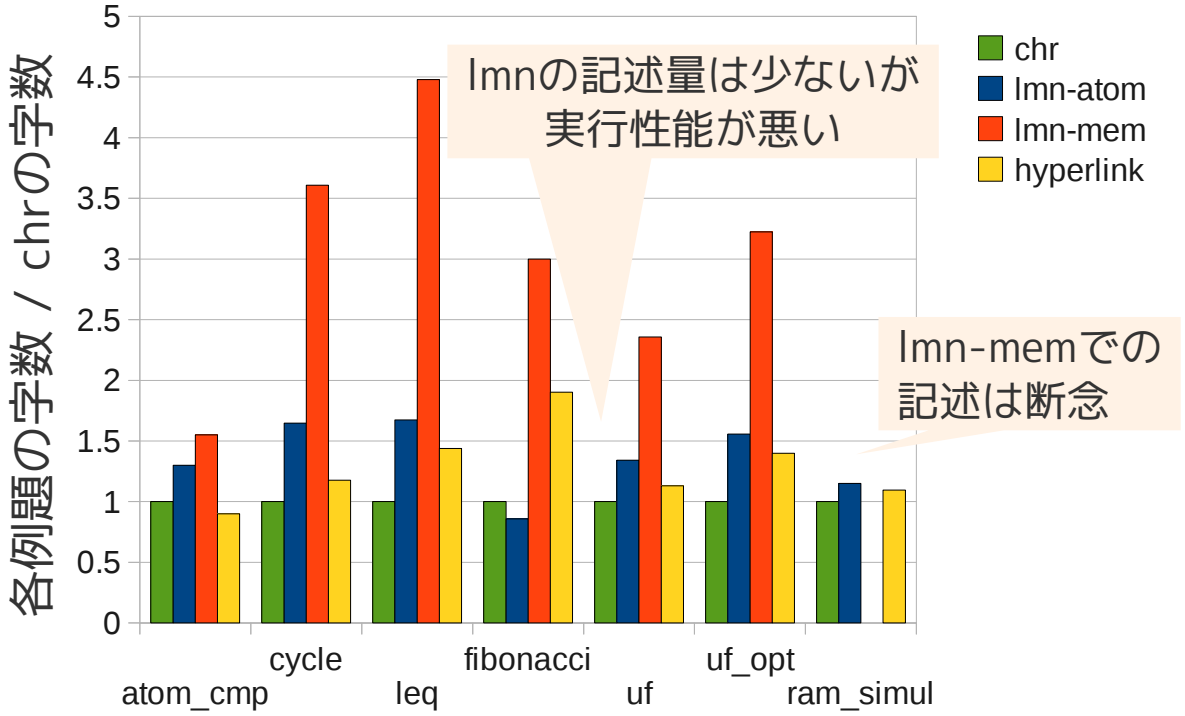
chr

```
leq(X,X) <=> true.
leq(X,Y), leq(Y,X) <=> X=Y.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
leq(X,Y) \ leq(X,Y) <=> true.
```

hyperlink

```
leq($x,$x) :- .
leq($x,$y), leq($y,$x) :- $x><$y.
leq($x,$y), leq($y,$z) \ :- uniq($x,$z) | leq($x,$z).
leq($x,$y) \ leq($x,$y) :- .
```

- chrとのプログラム文字数比較



lmn-memはchrの
約3～4倍の記述量

ハイパーリンクの導入で
最大でも2倍弱
従来よりも簡潔な
記述が可能に

参考文献

1. Ueda, K, “LMNtal as a Hierarchial Logic Programming Language”, Theoratical Computer Science, vol. 410, No. 46, pp. 4784-4800, 2009.
2. 小川誠司, 上田和紀, “LMNtalグラフのハイパーグラフへの拡張”, 第4回63号館ハイテクリサーチセンターシンポジウム, P10, 2010.
3. 小川誠司, 綾野貴之, 上田和紀, “LMNtalを用いた状態空間探索”, 第23回人工知能学会全国大会, 2H2-3, 2009.
4. Schrijvers, T. and Frɑuhwirth, T., “Optimal union-find in Constraint Handling Rules.” 2006, TPLP 6, 1-2, 213-224.
5. 小川誠司, 目黒学, 上田和紀, “階層グラフ書換えモデルを拡張したHyperLMNtalの実現”, 第25回人工知能学会全国大会, 2011, 発表予定.
6. Schrijvers, T. and Frɑuhwirth, T., “Optimal union-find in Constraint Handling Rules.”, TPLP 6, 1-2, 213-224, 2006.

実験環境

- LMNtal
  - Java版コンパイラ ver.1.02 “-O3 --slimcode --hl* ”
  - SLIM ver. 1.1.0 (rev. 407) “--hl”
    - 時間計測：callback関数(gettime) , “-p*”オプション
- CHR
  - SWI-Prolog ver. 5.8.3 (no type / type / type+array)
    - 時間計測：cputime関数
- マシン
  - CPU　：2.6GHz, Intel Core2 Quad
    RAM　：4.0 GB
    OS　　：ubuntu 9.10

CHR (Constraint Handling Rules) [1]

- 多重制約集合書換えに基づく並行プログラミング言語
- 計算処理：ルールによる制約集合の書換え（簡約化）
  - 制約：基本データ構造、変数間の関係を表現

```
leq(X, Y).    leq ：制約
              X,Y：論理変数
```

制約の例

```
LMNtal：H  :-  G | B.
CHR   ：H <=> G | B.
```

基本的なルール構造

  - 膜が無いLMNtal (flat LMNtal) に近い
  - CHRプログラムはLMNtal上で表現可能
- 多重集合書換えに基づく言語の中では表現力が高い
  - 実アプリへの利用 (The Munich Rent Advisor [7]など)
- 処理系：SWI-Prologが最も一般的
  - ホスト言語 (Prolog) 上にライブラリとして提供

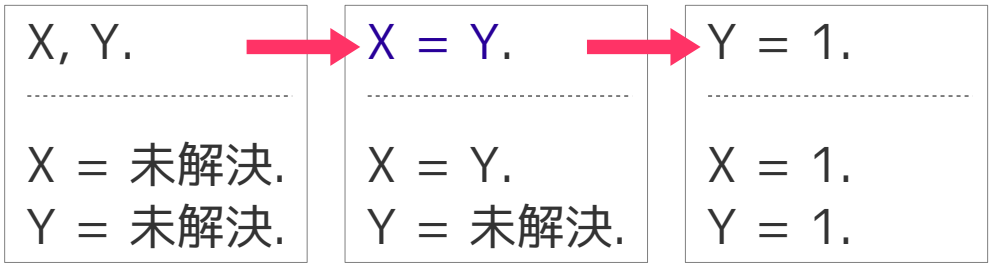
他には
Haskel, Java

[7] The Munich Rent Advisor (MRA) won the prize for best application at JFPLC, Clermont Ferrand, France, June 1996.

論理変数（1/2）

- C言語、Javaなどの変数
  - 値の格納場所、計算処理によって値が変化する
- 論理変数：論理型言語（Prolog, CHR, etc.）の変数
  - 値（解決済/未解決）に付けた名前（capital）
  - 解決し、変数に束縛された値は変化しない
  - “=”によるUnification
    - 変数への値の束縛
    - 異なる名前の論理変数を併合（併合後の再分割は無い）

数学的な
変数に近い
x+2 = 2x

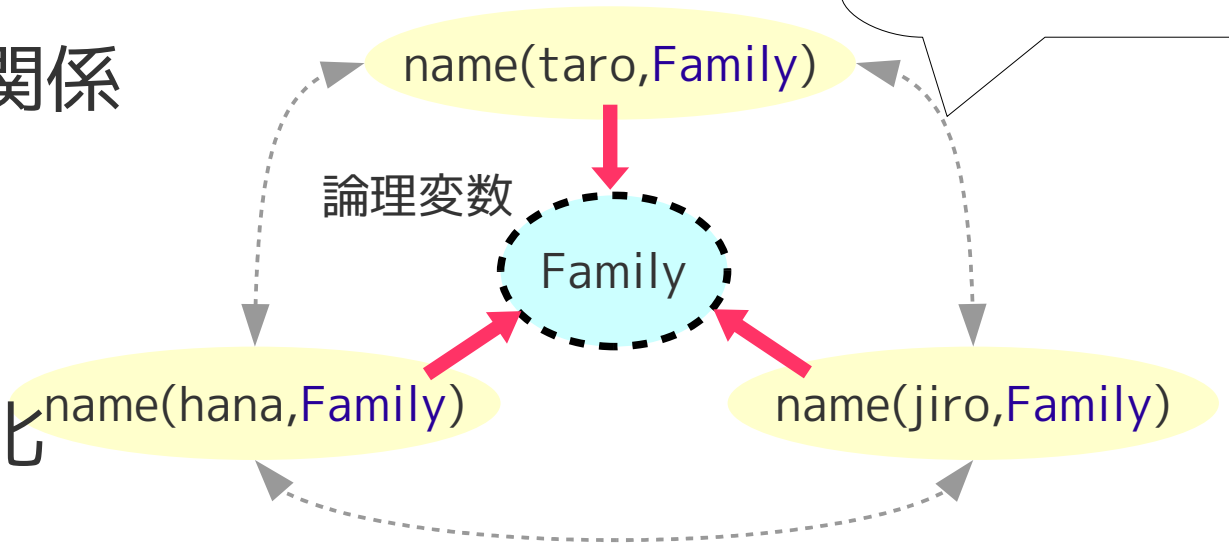


論理変数（2/2）

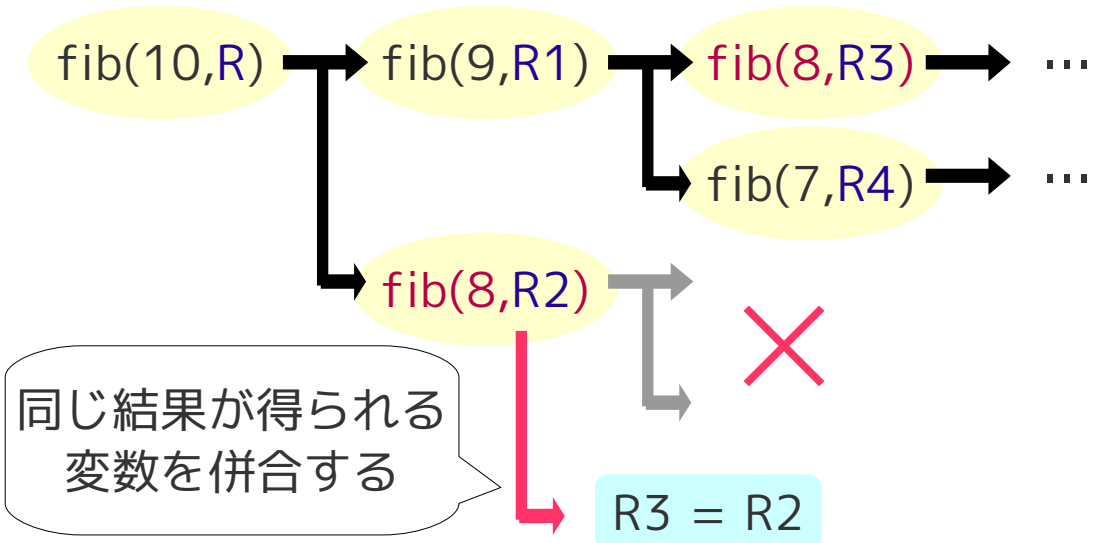
- データ間の参照関係

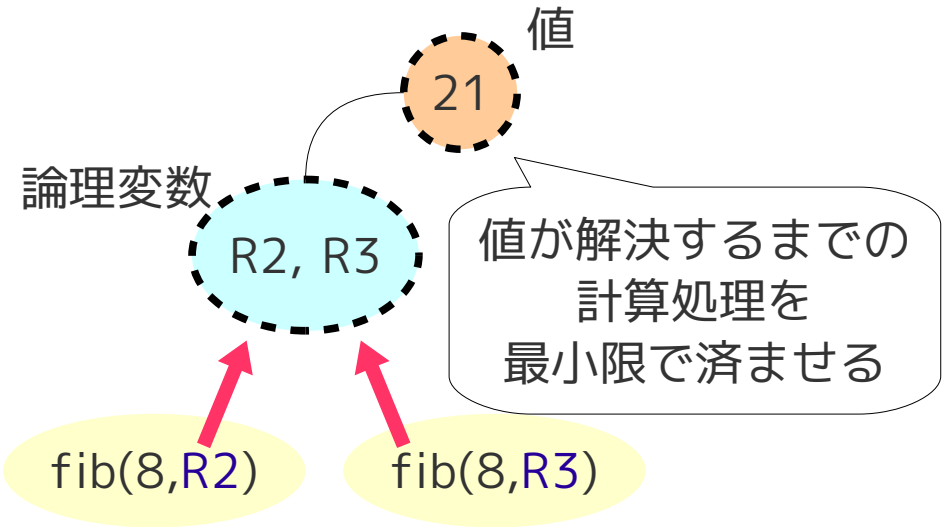
```
name(taro,Family).
name(jiro,Family).
name(hana,Family).
```

- 計算処理の簡略化



例：フィボナッチ数（topdown）





CHRにおけるマッチング最適化

```
fib(N,R1), fib(N,R2) <=>  ...
```

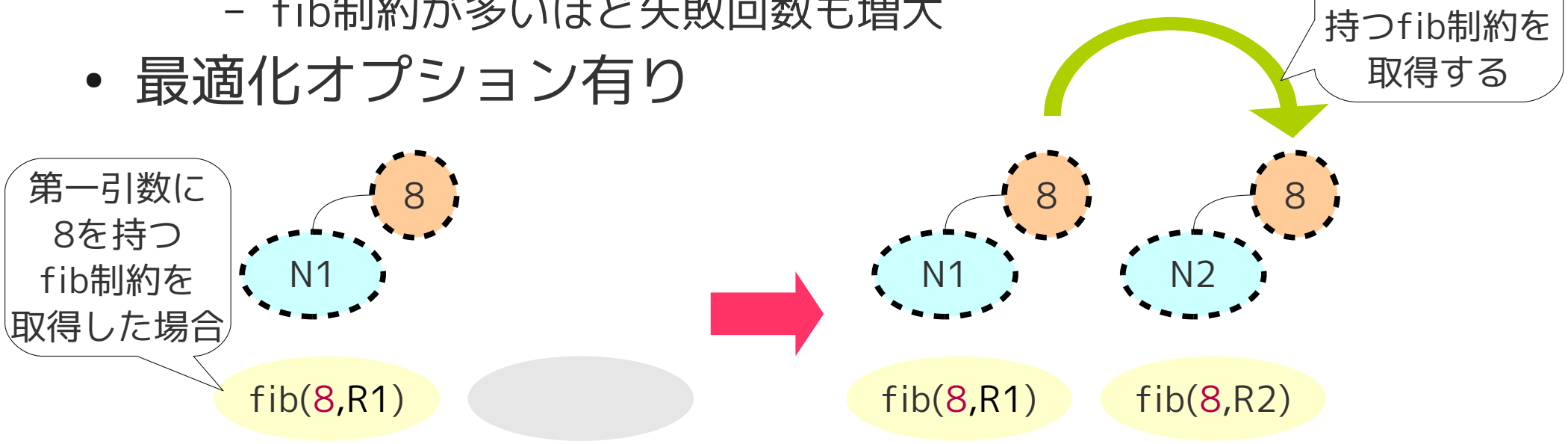
- 通常実行（最適化オプション無し）

  = `fib(N1,R1), fib(N2,R2) <=> N1==N2 | ...`

  - fib制約が多いほど失敗回数も増大

fib制約を2つ取得し、
ガードで判定するまで
マッチング成功/失敗が判定されない

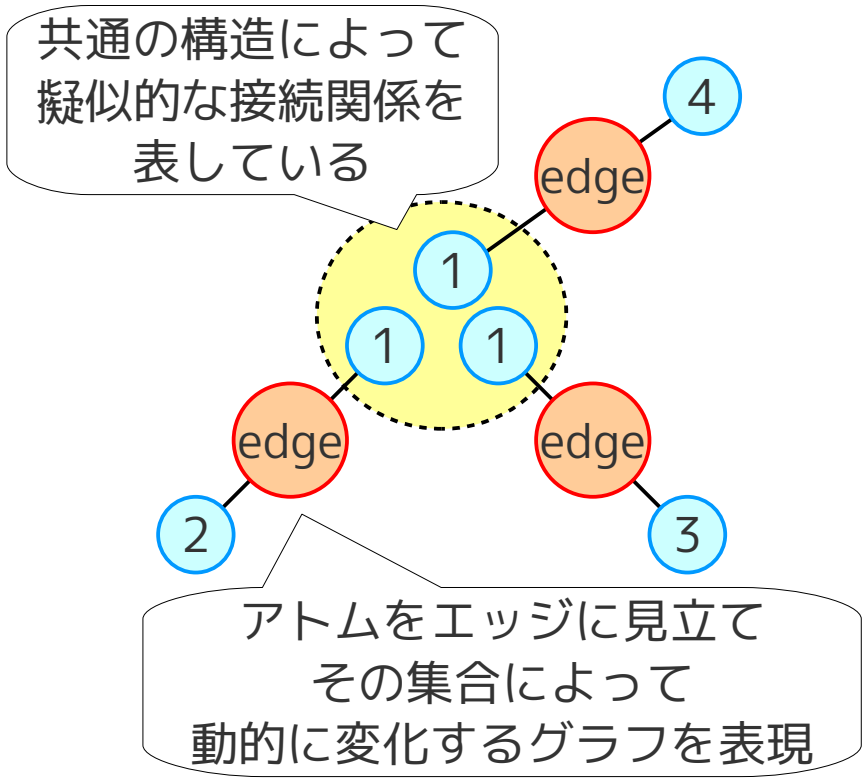
- 最適化オプション有り

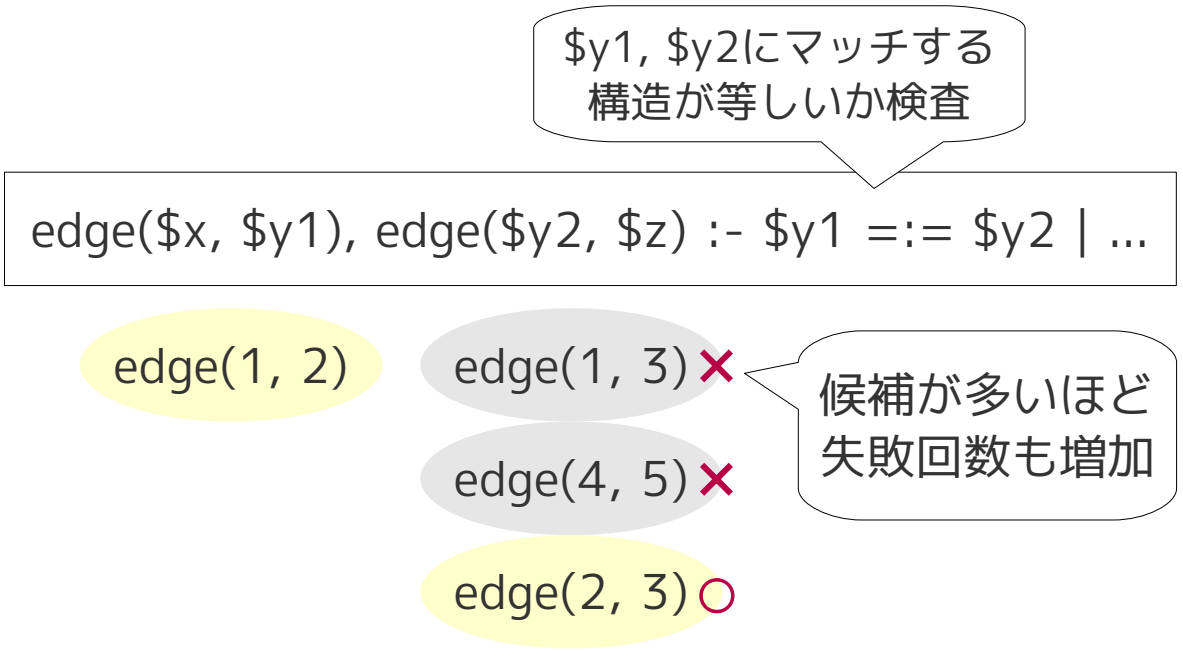


- ルールヘッドにマッチした段階でN1=N2が確定されている
  - 求める制約を効率的に探索可能

LMNtalのリンク

- アトム間の一対一の参照関係を構成
  - 一対一以外の参照関係は、数値アトムなど共通の構造で表現
    - ルール上ではアトムの価数を明記しなければならない
  - 例：グラフ問題

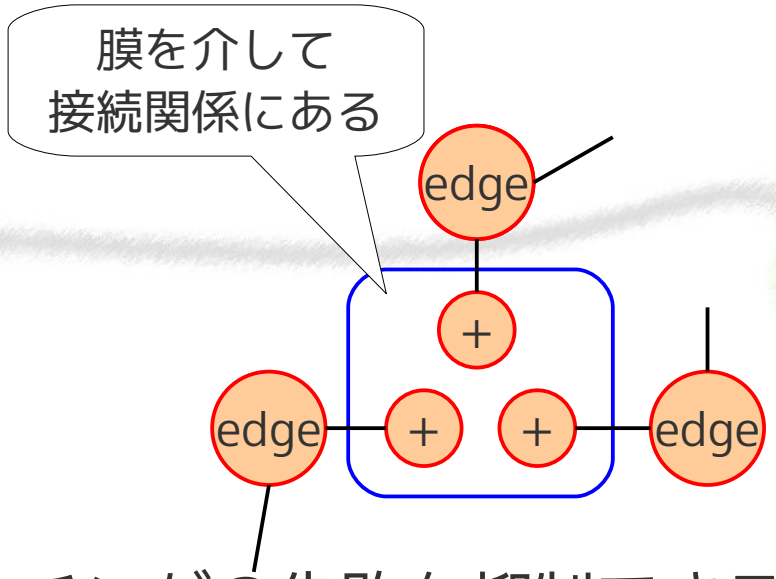




簡潔に表現しやすい記法ではあるが

実行性能の悪化に繋がる可能性大

膜を用いた共有構造



- 膜をハブのように利用
  - 擬似的な論理変数を表現可能
  - 実在の接続関係を活かし、マッチングの失敗を抑制できる

```
fib(X1, R1), fib(X2, R2), {+(X1), +(X2)} :- ...
```

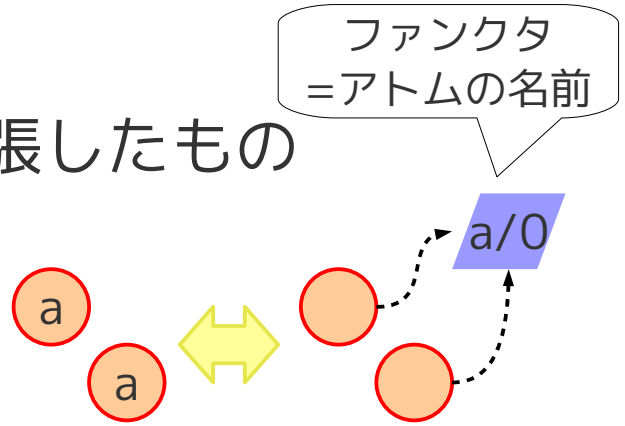
- 共有構造が実体としてグラフ上に存在しなければならない
  - 動的に変化する階層グラフ構造上で表現が困難
    - 冗長なルールが増え、逆に性能が悪化
    - 共有構造の生成に時間がかかる
    - 共有構造のために利用するには膜はリッチな構造

実用的とは言い難い

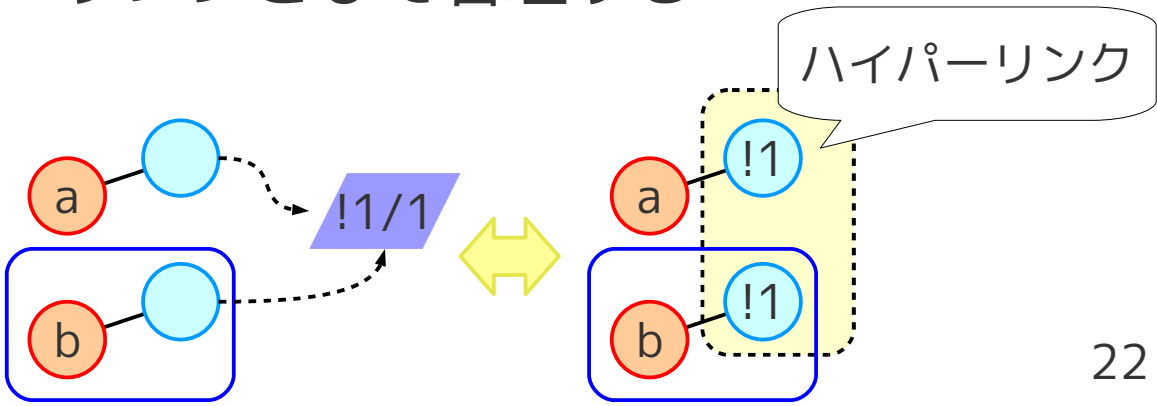
⇒グラフ上に実体を持たない、より軽量な共有構造が必要

ハイパーリンク

- ハイパー（LMNtal）グラフを表現するための構造
  - 概念：LMNtalのリンクを拡張したもの
  - 実装：アトム（が持つファンクタ）を拡張したもの
- ファンクタは処理系内部の情報
  - グラフ上に実体を持たない
  - ハイパーリンクアトム（単にハイパーリンクとも）
    - 特別なファンクタ（“!”+ID）を持つ1価のアトム
    - 処理系内部でハイパーリンクとして管理する



cf. ハイパーグラフ
数学におけるグラフを一般化したもの
1つのエッジが任意個数の
要素を連結可能



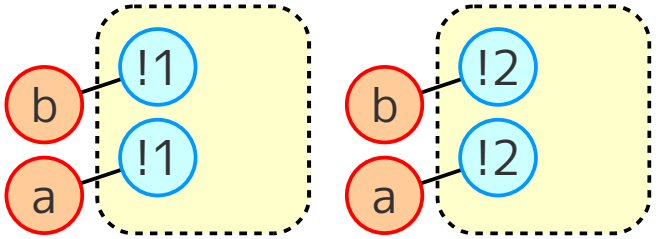
ハイパーリンクの生成・型制約

- ハイパーリンクの生成

```
hoge :- new($x) | a($x), b($x).
```

  - 新規のハイパーリンクはルールによって生成する

```
hoge, hoge.
*--->
a(!1), b(!1), a(!2), b(!2).
```

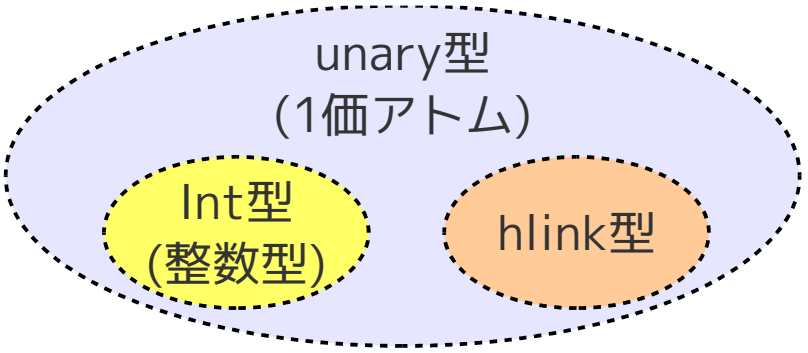
- ハイパーリンク型制約

```
a($x) :- hlink($x) | ok($x).
```

  - hlink型 ⊂ unary型
  - int型 ≠ hlink型

```
a(!1), a(123), a(b).
*--->
ok(!1), a(123), a(b).
```

型制約の階層関係



ハイパーリンクの併合

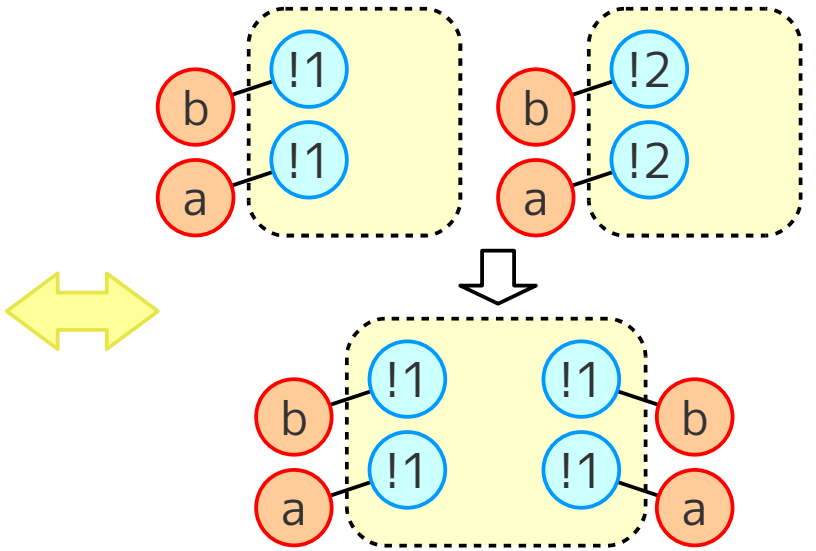
- LMNtalのリンクは"="”によりリンク同士を接続可能
  - ハイパーリンクもリンクであるため、接続処理が必要
- 異なるハイパーリンク同士を接続
  = 異なる集合同士の併合

TeXにならって
bow tie ？

```
a($x), a($y) :- $x \= $y | a($x), a($y), $x >< $y.
```

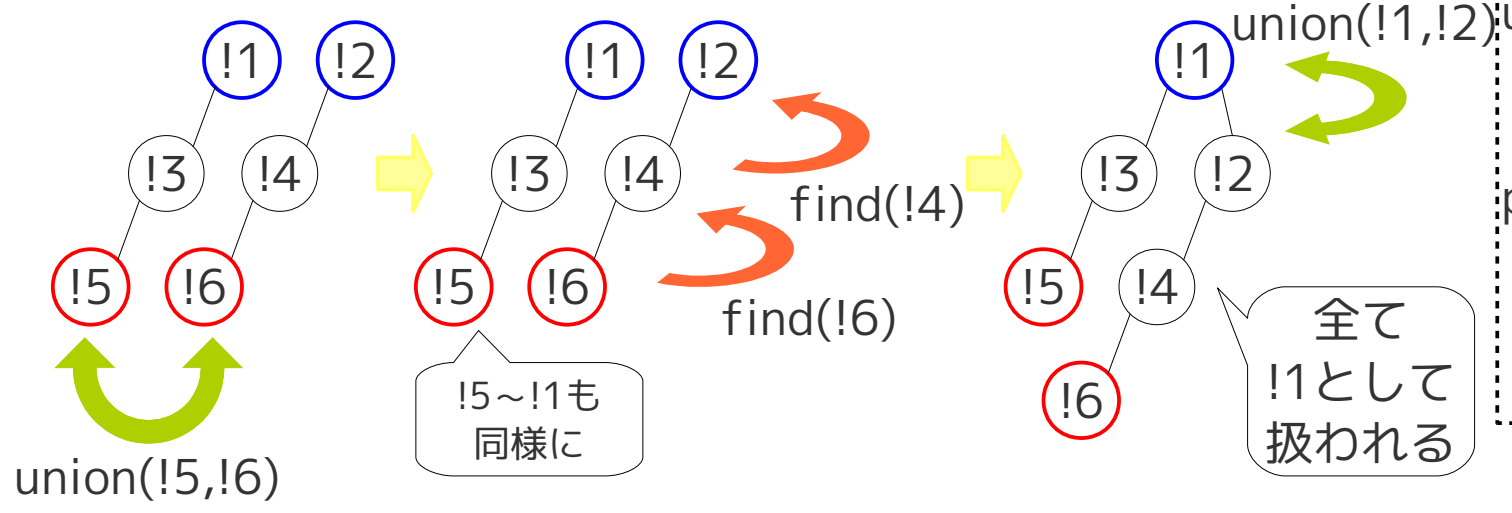
```
a(!1), b(!1), a(!2), b(!2).
*--->
a(!1), b(!1), a(!1), b(!1).
```

併合後は同じ名前の
ハイパーリンクとして
扱われる



Union-Find algorithm

- 併合関係はUnion-Find algorithmによって管理
  - 要素集合の管理手法
  - 木構造(＋2つの最適化)が(漸近的に)最も高速に処理可能[4]
    - union by rank, path compression
  - find(X)：Xの親を返す
  - union(X, Y)：要素X, Yそれぞれが所属する木のrootを連結する



union by rank：
子の数(rank)が多い
木をunion後の親とする
path compression：
findでrootまで辿った後、
経路上のノードを全て
rootに直接接続しなおす

ハイパーリンクを利用したマッチング最適化

```
a($x), b($x) :- ok.
```

- 通常実行（最適化オプション無し）

コンパイル時に変換

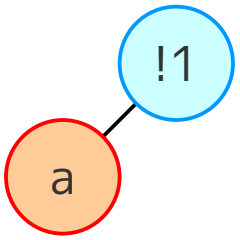
片方の文脈に
ユニークな名前を与える

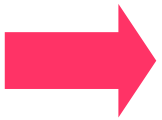
```
a($x), b($x0) <=> $x = $x0 | ...
```

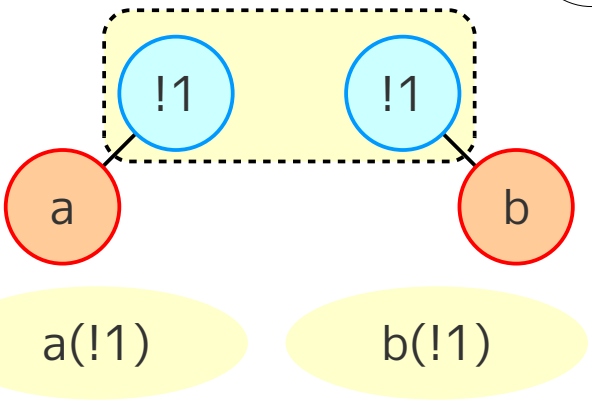
- 最適化オプション有り（--hl-opt）

同じく!1に
接続された
bアトムを
取得する

!1に
接続された
aアトムを
取得した場合







ハイパーリンクによる接続関係をマッチングに反映

- ガードでのアトム等価判定の失敗を極力防ぐことが可能

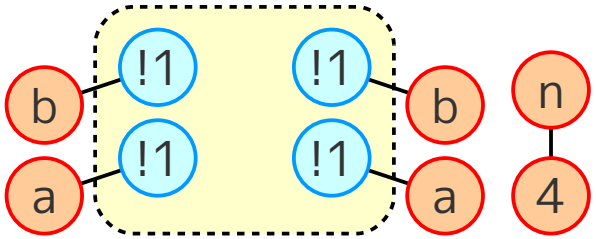
要素数取得

- 要素（アトム）の個数を瞬時に取得する機能は無かった
  - N個の要素を数えるために、N回のルール適用が必要
- num制約で任意のハイパーリンクの個数を取得可能
  - グラフ上に存在する自身（ハイパーリンクアトム）の個数を保持

```
a($x) :- $n = num($x) |
         a($x), n($n).
```

```
flag($x) :- num($x) =:= 1 | ok.
a($x) :- hlink($x) | .
```

```
a(!1), b(!1), c(!1), d(!1).
*--->
a(!1), b(!1), c(!1), d(!1),
n(4).
```




```
flag(!1), a(!1), a(!1).
*--->
flag(!1), a(!1).
*--->
flag(!1).
*--->
ok.
```

!1の数が1になるまで
1本目のルールは
適用されない

実験

- 実験環境：スライド4ページ
- CHR benchmark（ http://people.cs.kuleuven.be/~tom.schrijvers/CHR/ ）
  - CHR最適化によって効果を得た例題を中心に
  - 各例題に対し4種類の実装を用意
- LMNtal
  - slim　　：通常の実装
  - slim mem　：膜の共有構造による実装
  - slim hl-opt　：ハイパーリンクによる実装（最適化モード）
- CHR
  - chr opt　　：最適化モード

基本性能

- アトムのコピーと削除
  - 通常／ハイパーリンクアトムを10000*N回ずつ複製＆削除

N	100	200	400	800
atom	1.19	2.34	4.64	9.24
hlink	1.76	3.48	6.95	13.8

[sec]

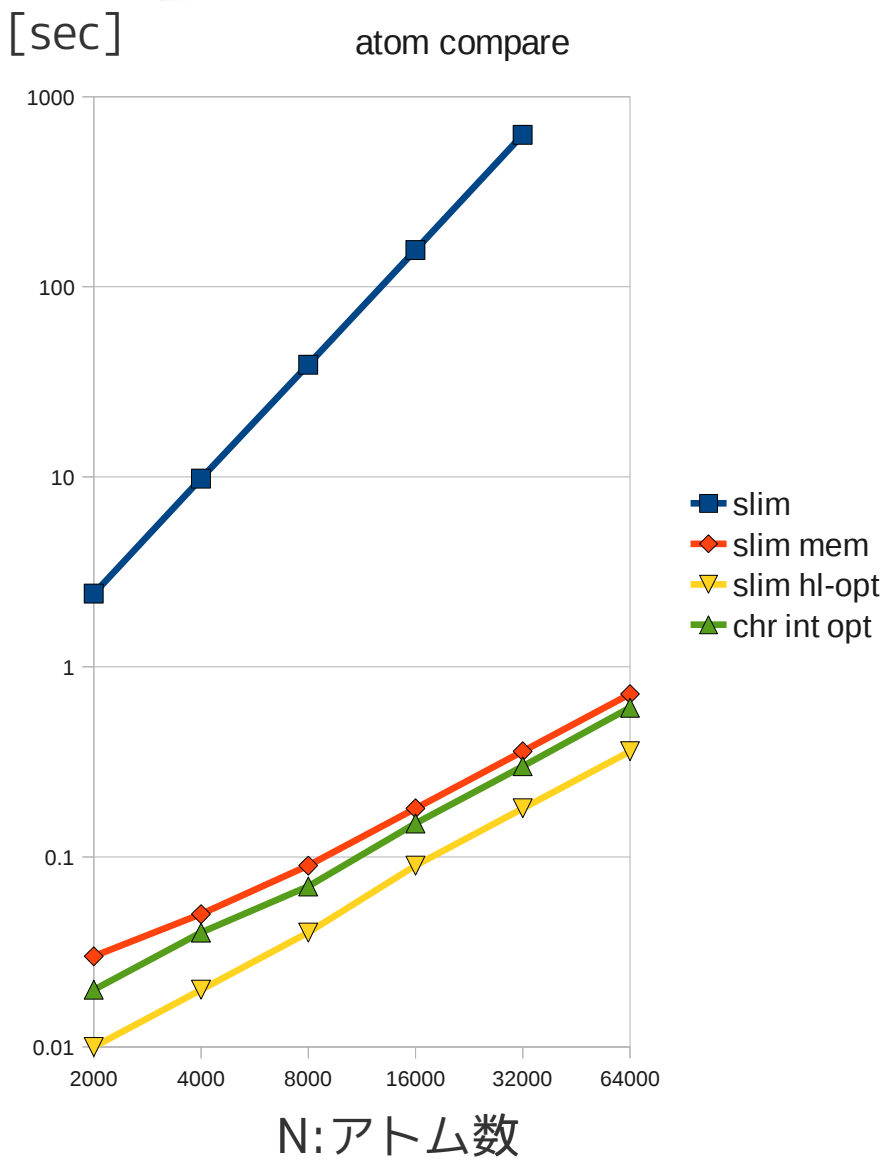
要素数を計算している
複製＆削除に
若干時間がかかる

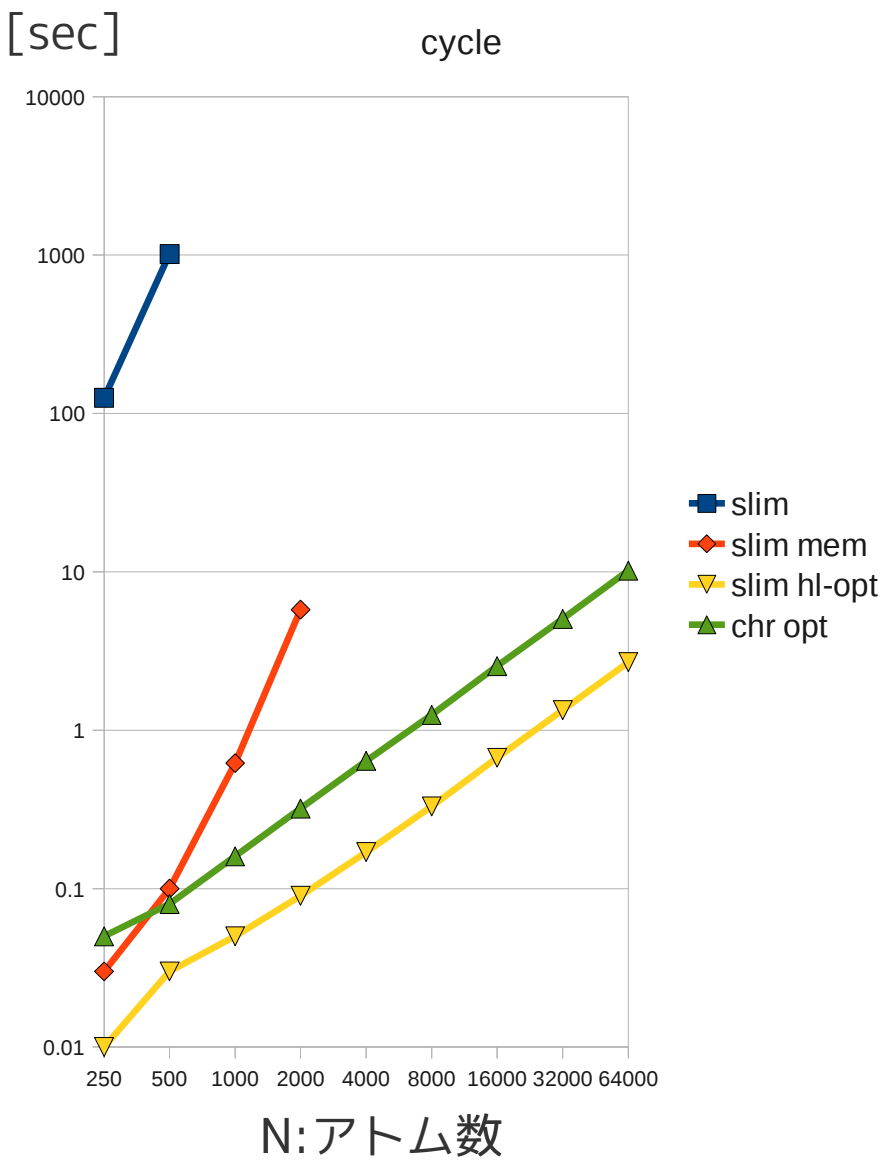
- 要素数取得
  - 10000個の要素を持つ集合の要素数をN回取得
    - mem：膜内のアトムをルールによって数える

N	2000	4000	8000	16000
mem	7.74	15.51	31.11	62.21
hlink	0.01	0.02	0.03	0.05

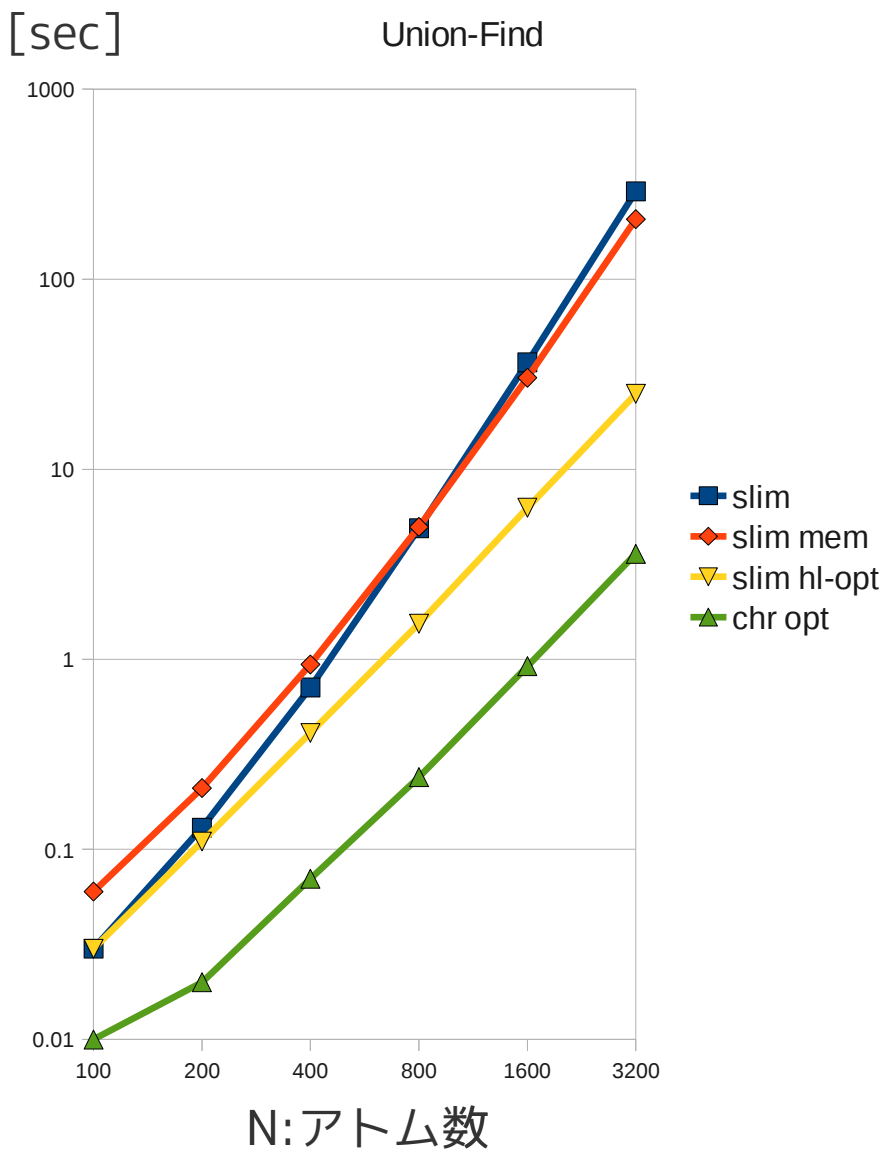
[sec]

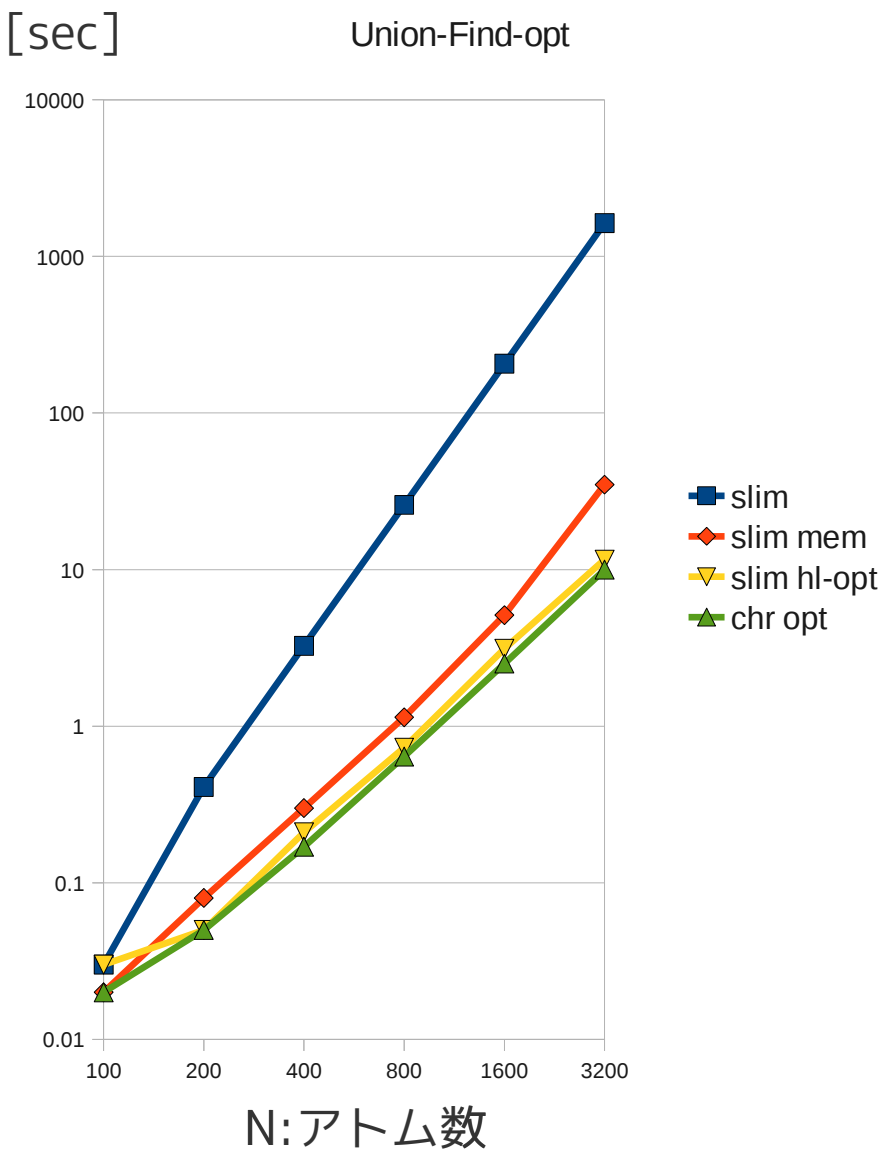
各例題の実験結果（1/4）


[sec]
atom compare
1000
100
10
1
0.1
0.01
2000
4000
8000
16000
32000
64000
slim
slim mem
slim hl-opt
chr int opt
N:アトム数

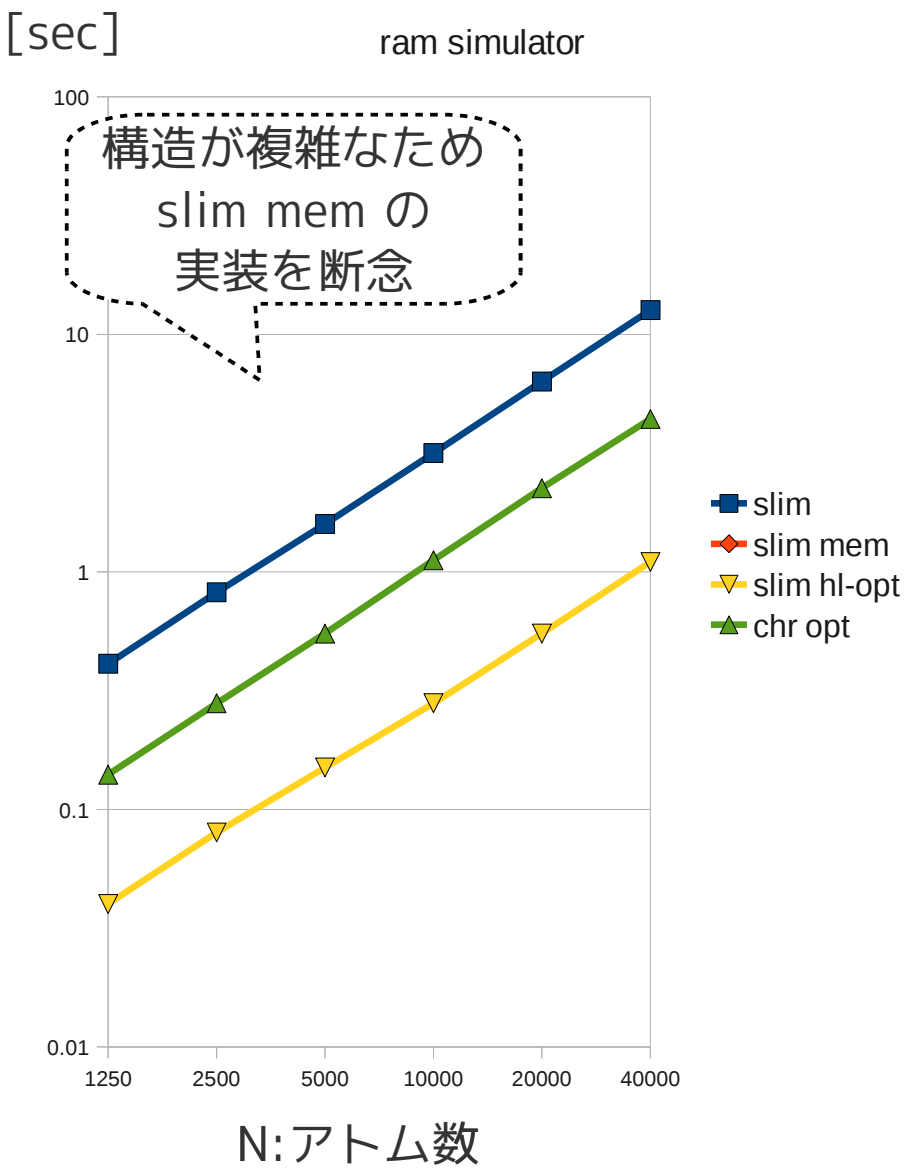

[sec]
cycle
10000
1000
100
10
1
0.1
0.01
250
500
1000
2000
4000
8000
16000
32000
64000
slim
slim mem
slim hl-opt
chr opt
N:アトム数

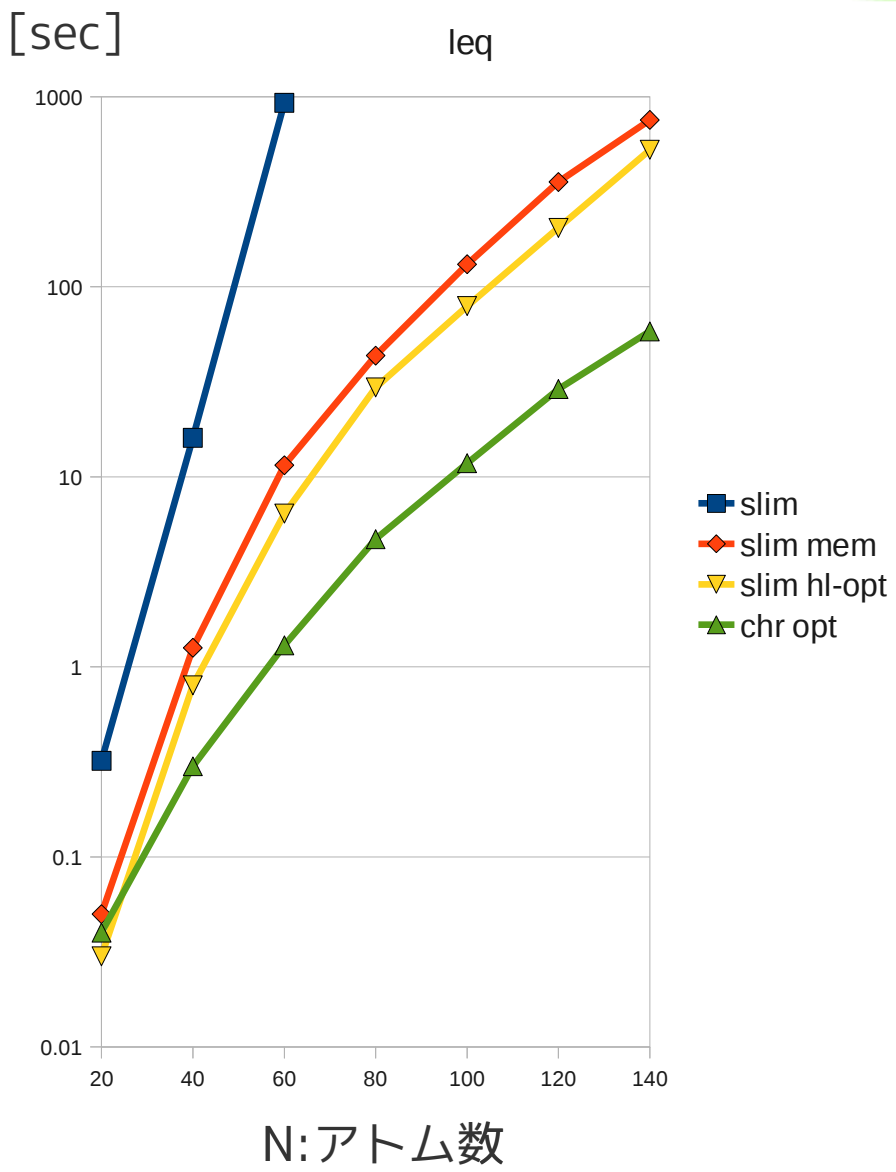
各例題の実験結果（2/4）


[sec]
Union-Find
1000
100
10
1
0.1
0.01
100
200
400
800
1600
3200
slim
slim mem
slim hl-opt
chr opt
N:アトム数


[sec]
Union-Find-opt
10000
1000
100
10
1
0.1
0.01
100
200
400
800
1600
3200
slim
slim mem
slim hl-opt
chr opt
N:アトム数

各例題の実験結果（3/4）


[sec]
ram simulator
構造が複雑なため
slim mem の
実装を断念
100
10
1
0.1
0.01
1250
2500
5000
10000
20000
40000
slim
slim mem
slim hl-opt
chr opt
N:アトム数


[sec]
leq
1000
100
10
1
0.1
0.01
20
40
60
80
100
120
140
slim
slim mem
slim hl-opt
chr opt
N:アトム数

各例題の実験結果（4/4）

併合を活用し
計算処理を簡略化した例

fibonacci topdown

N	10	15	20	25	5000	10000	20000
slim	0.01	0.03	2	241.64	-	-	-
slim mem	0.04	38.7	-	-	-	-	-
slim hl-opt	-	-	-	-	0.08	0.18	0.37
chr opt	-	-	-	-	0.03	0.08	0.17

※fibonacci数はより高速に求める方法がある

[sec]

cycle（全実行時間）

N	250	500	1000	2000
slim mem	2.8	22.04	175.7	1991.09
slim hl-opt	0.02	0.04	0.07	0.12

[sec]

初期グラフを
ランダムに与えるため
共有構造の生成に
時間がかかっている

union-find opt（全実行時間）

N	400	800	1600	3200
slim mem	3.1	23.45	233.39	5157.01
slim hl-opt	0.22	0.75	3.14	11.6

[sec]

対してハイパーリンクは
生成にかかる時間が
ごくわずか

まとめ

- ハイパーリンクを用いることで、
  - 従来のLMNtalによる実装よりも時間計算量が改善出来ている
  - 膜の共有構造と比べても十分に実用に適う性能
    - 実行時間、コーディングの簡潔さ
  - ほぼ全ての例題で、CHRと同じ時間計算量で実行可能
    - leqに関しては、propagation 対 uniq制約の影響が大きい
      - propagation　：ヘッドでのマッチング時に判定
      - uniq制約　　：ガードで判定（条件の成否判定まで時間がかかる）

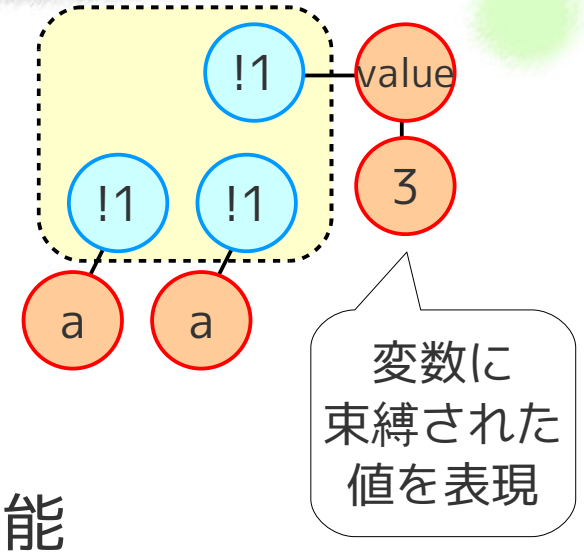
同一の構造に対し一度だけルールを適用させる制約

今後の課題（1/2）

- ガベージコレクション
  - グラフ上から消滅したハイパーリンクIDの再利用
    - プログラム実行開始後に新しいファンクタが生成されることはこれまで想定されていなかった
    - 約65,000個(16bit - 予約語ファンクタ数)が限度
    - 例：アッカーマン関数（(M,N)=(3,5)程度でIDが不足）
    - メモリ削減にも繋がる

  ⇒より大きな例題への利用が可能となる

今後の課題（2/2）



- 論理変数への拡張の検討
  - 値の束縛
    - 現在はグラフ上に値の実体がある
    - 内部管理することで探索が不要となる
    - 値の束縛／未束縛をより明確に表現可能
  - attributed variable (属性付き変数)
    - X < 3など、値の束縛以外の方法で変数に値を持たせる

    ⇒集合論、制約プログラミングへの応用
- 非決定実行への対応
  - 併合関係の情報をどのように各状態に持たせるか

各例題のソースコード

atom compare

```
ab @@ a($x), b($x) :-
  hlink($x) | .
```

cycle

```
edge($h1, $h2), edge($h2, $h3), edge($h3, $h1) :- cycle($h1,
$h2, $h3).
```

leq

```
reflexivity  @@ leq($x, $x) :- .
antisymmetry @@ leq($x, $y), leq($y, $x) :- hlink($x), $x \= $y |
$x >< $y.
idempotence  @@ leq($x, $y), leq($x, $y) :- leq($x, $y).
transitivity @@ leq($x, $y), leq($y, $z) :-
                  uniq($x, $z) | leq($x, $y), leq($y, $z), leq($x, $z).
```

fibonacci topdown

```
unify @@ fib($n, N1, M1) \ fib($n, N2, M2) :- hlink(M1), int(N2) | M1 >< M2.
fib1 @@ fib(N,0,M) :- hlink(N) | v(M, 1).
fib2 @@ fib(N,1,M) :- hlink(N) | v(M, 1).
fib3 @@
fib($n, N, M) :-              /* hlinkidとN,N1は1ずつずらしている */
  N > 1, N1 = N-1, N2 = N-2, make(N, $a), make(N1, $b), new($x), new($y),
hlink($n) |
  fib($a, N1, $x), fib($b, N2, $y), M = $x + $y.
```

各例題のソースコード

Union-FInd

```
findNode @@ '~>'($a,B), find($a,X) :- hlink(B) | '~>'($a,B),
find(B,X).
findRoot @@ root($a), find($a,X) :- root($a), X=$a.
linkEq   @@ link($a,$a) :- .
link     @@ link($a,$b), root($a), root($b) :- '~>'($b,$a),
root($a)
```

Union-FInd opt

```
(A,B) :- find(A,X), find(B,Y), link(X,Y).
make @@ make(A) :- root(A).
linkEq   @@ link($a,$a) :- .
linkLeft @@ link($a,$b), root($a,NA), root($b,NB) :- NA>=NB, NB1 = NB+1 |
            '~>'($b,$a), NA1 = max(NA,NB1), root($a,NA1).
linkRight@@ link($b,$a), root($a,NA), root($b,NB) :- NA>=NB, NB1 = NB+1 |
            '~>'($b,$a), NA1 = max(NA,NB1), root($a,NA1).
max1 @@ H = max(A,B) :- A =< B | H = B.
max2 @@ H = max(A,B) :- A > B  | H = A.
findNode @@ '~>'($a,B), find($a,link(find(X))) :- hlink(X) | find(B,link(find(X))),
'~>'($a,X).
findRoot @@ root($b,R) \ find($b,X) :- X=$b.
union @@ union(A,B) :- find(A,X), find(B,Y), link(X,Y).
```

各例題のソースコード

ram simulator

```
% add value of register B to register A
add@@
prog($I,LN, add($b),$a), mem($b,Y) \ mem($a,X), prog_counter($I) :-
  Z = X + Y, hlink(LN) |
  mem($a,Z), prog_counter(LN).
% subtract value of register B from register A
sub@@
prog($I,LN, sub($b),$a), mem($b,Y) \ mem($a,X), prog_counter($I) :-
  Z = X - Y, hlink(LN) |
  mem($a,Z), prog_counter(LN).

% multiply register A with value of register B
mul@@
prog(L1,LN,mult(B1),A1), mem(B2,Y) \ mem(A2,X), prog_counter(L2) :-
  A1 =:= A2, B1 =:= B2, L1 =:= L2, Z = X * Y, int(LN) |
  mem(A1,Z), prog_counter(LN).
% divide register A by value of register B
div@@
prog(L1,LN, div(B1),A1), mem(B2,Y) \ mem(A2,X), prog_counter(L2) :-
  A1 =:= A2, B1 =:= B2, L1 =:= L2, Z = X / Y, int(LN) |
  mem(A1,Z), prog_counter(LN).
% put the value in register B in register A
move@@
prog(L1,LN,move(B1),A1), mem(B2,X) \ mem(A2,Y), prog_counter(L2) :-
  A1 =:= A2, B1 =:= B2, L1 =:= L2, int(X), int(Y), int(LN) |
  mem(A1,X), prog_counter(LN).
% put the value in register <value in register B> in register A
i_move@@
prog(L1,LN,i_move(B1),A1), mem(B2,C1), mem(C2,X) \ mem(A2,Y),
prog_counter(L2) :-
  A1 =:= A2, B1 =:= B2, C1 =:= C2, L1 =:= L2, int(X), int(Y), int(LN) |
  mem(A1,X), prog_counter(LN).
% put the value in register B in register <value in register A>
move_i@@
prog(L1,LN,move_i(B1),A1), mem(B2,X), mem(A2,C1) \ mem(C2,Y),
prog_counter(L2) :-
  A1 =:= A2, B1 =:= B2, C1 =:= C2, L1 =:= L2, int(X), int(Y), int(LN) |
  mem(C1,X), prog_counter(LN).
```

```
% put the value B in register A        -> redundant if consts are in
init mem
const@@
prog(L1,LN,const(B),A1) \ mem(A2,X), prog_counter(L2) :-
  A1 =:= A2, L1 =:= L2, int(B), int(X), int(LN) |
  mem(A1,B), prog_counter(LN).
%zero@@
%prog(L1,LN,clr,A1) \ mem(A2,X), prog_counter(L2) :-   % same as
const(0)
%  A1 =:= A2, L1 =:= L2, int(X), int(LN) |
%  mem(A1,0), prog_counter(LN).
% unconditional jump to label A
jump@@
prog(L1,LN,jump,A) \ prog_counter(L2) :-
  L1 =:= L2, int(A) | prog_counter(A).
% jump to label A if register R is zero, otherwise continue
cjump0@@
prog($I,LN,cjump($r),A), mem($r,0) \ prog_counter($I) :-
  hlink(A) | prog_counter(A).
cjumpN@@
prog($I,LN,cjump($r),A), mem($r,X) \ prog_counter($I) :-
  X =\= 0, hlink(LN)  | prog_counter(LN).
% halt
halt@@
prog($I,LN,halt,_) \ prog_counter($I) :-  .
% invalid instruction
% prog_counter(L) :- int(L) | true.
```

中間命令：findproccxt命令

original：探索の始点側
clone　：hyperlinkから探索される側

```
a($x), b($x) :- ...
```

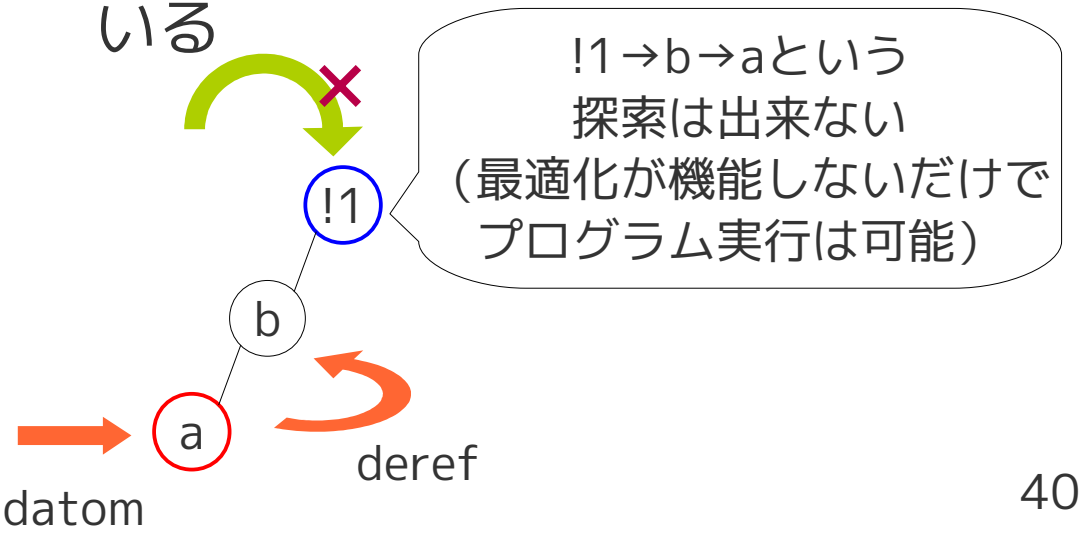
中間命令列の
アトム探索部

等しい構造を持つ
引数同士の情報を
挿入している

```
...
findproccxt [1, 1, 0, 2, 1, 0]
findatom [1, 0, a_1]
findatom [2, 0, b_1]
...
```

findatom命令によって
2価のedgeアトムを取得

- findproccxt [A1,A2,A3,B1,B2,B3]
  - A1,B1：アトム番号
  - A2,B2：アトムの価数
  - A3,B3：等価構造の引数番号
- clone側の引数がfindatomで探索されているアトム(ここではb)に直接接続していることを想定している



予備：CHRの構文・意味

- Simplification rule
  - Head <=> Guard | Body.
    - Guardがtrueならば、HeadをBodyに書換え
- Propagation rule
  - Head ==> Guard | Body.
    - Guardがtrueならば、HeadにBodyを一度だけ追加
    - LMNtalではuniq制約を含むルールに相当
- Simpagation rule
  - Head1 \ Head2 <=> Guard | Body.
    - Guardがtrueならば、”\”より後ろのHeadをBodyに書換え

CHR処理系

- ホスト言語上にライブラリとして提供
  - ホスト言語のプログラムに組み込める
- Prolog
  - SWI Prolog
    - 最も一般的なProlog処理系 (Cygwinに標準装備)
    - (CHR処理系として)最適化が最も進んでいる処理系の一つ
  - SICStus Prolog, XSB Prolog, hProlog, etc.
- Haskell, Java, C
  - CCHR (Concurrent CHR with STM in Haskell)

予備：Optimal Union-Find

- union by rank
  - 木をバランス良く保つことで、計算回数を削減
  - ルートが子の数（ランク）の情報を保持
  - ランクが小さい木から大きい木へとリンクを張る
    - Union命令時
- path compression
  - 各ノードとルートとの距離を詰める
  - Findにより一度辿られた経路上のノードを記憶
  - ルート発見時に経路上ノードをルートに直接接続

予備：LMNtalのリンク条件

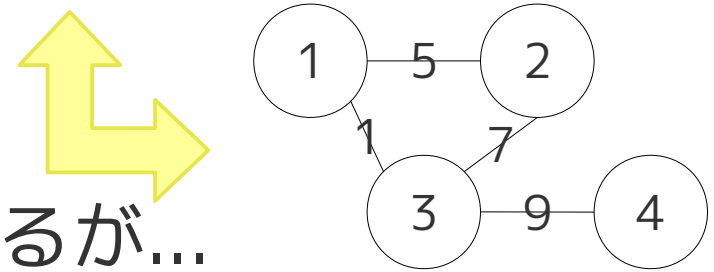
- リンク条件
  - 同じリンク名が2回を超えて出現してはならない
  - LMNtalは(ハイパーでない)階層グラフを表現する言語

CHR：グラフ問題の表現

edge(*Node1, Node2, Weight*).

- エッジ：制約
- ノード(と距離)：数値
  - edgeの集合でグラフを表現

```
edge(1, 2, 5), edge(2, 3, 7),
edge(1, 3, 1), edge(3, 4, 9).
```



- ノードを制約で表現する方法もあるが...

node(*Edge1 , Edge2 , …*)

- ルールヘッドに出現する制約は引数個数を明記する必要がある
  - node(E1,E2,E3,E4,W) <=> ...

⇒動的にエッジ数が増減するグラフを表現しにくい

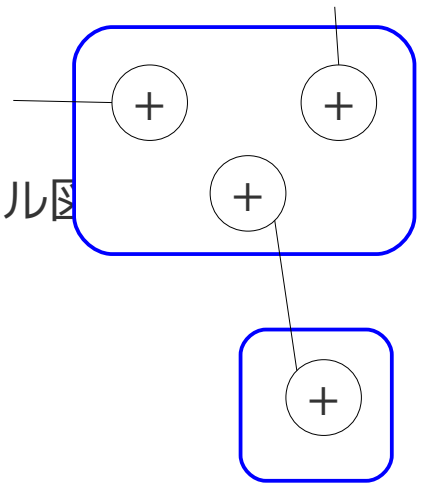
WebCHR公開のグラフ問題11題中、10題がedge式 (残りの1題は静的なグラフを深さ優先に辿る例題)

LMNtal：グラフ問題の表現

- アトムをノード、リンクをエッジとして表現すると…
  - 動的にエッジ数が増減するグラフを表現しにくい
    - 自動生成が困難（性能測定に不向き）

  edge(*Node1, Node2, Weight*)の集合でグラフを表現
  - マッチング失敗回数が増加する可能性
- 膜によるノード：{+*Edge1* , +*Edge2* , +*Edge3* , …}
  - ルール上で引数個数を明記しなくてよい
    - {+L1, +L2, $p} <=> …

      cf. 笹島唯, 小川誠司, “LMNtal合宿成果報告 フラクタル図
  - 実際に接続関係があるグラフを記述可能



CHR：制約の管理とルール適用 (1/2)

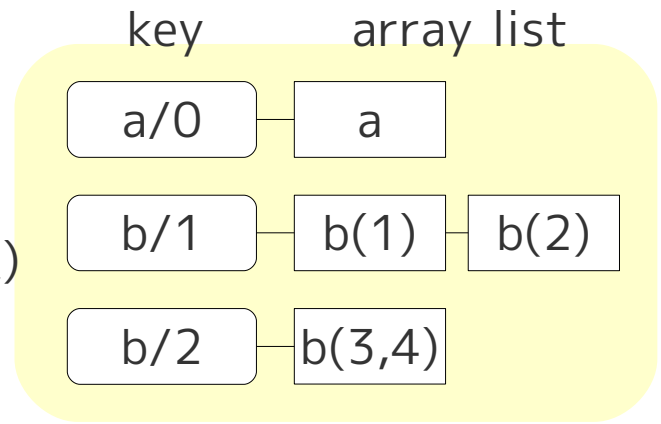
例：

```
a. b(1).
a, b(X) <=> c(X).
```

< E | C >
E：実行スタック
C：制約ストア

a, b(1), b(2), b(3,4)

制約ストア
(ハッシュ表)



< [a, b(1)] | Ø >
初期制約は全てEに積まれる

→insert < [b(1)] | { a } >
Eの制約の一つをCに移動
ルール適用を試みるが失敗

→insert < Ø | { a, b(1) } >
Eの制約の一つをCに移動
ルール適用を試み成功

→apply < [c(1)] | Ø >
適用された制約はCから取り除かれ、
ルールのボディをEに積む

→insert < Ø | { c(1) } >

CHR：制約の管理とルール適用 (2/2)

- active / passive constraint
  - EからCに移された制約はactive制約になる

```
active : b(X)   |   rule2 : a, b(X) :- ...
```

    - active制約をヘッドに持つルールの適用を試みる
      - Cからヘッド内のその他の（partner）制約を見つける
  - 適用できるルールが無いければ、active 制約はpassive制約になる
    - 別のactive制約によるルール適用での除去を待つ
    - Eが∅かつC内の制約が全てpassiveならば実行終了

    ⇒ 少ない失敗回数でのpartnerの発見が望ましい

CHRにおける最適化[5]

- type/mode
  - 制約の宣言時に、変数のmode、typeを指定する

```
mode
    +   : 値が束縛された変数
    -   : 値が未束縛な変数
    ?   : 束縛済/未束縛を区別しない
```

```
type (built in)
    int          : integer number
    dense_int    : 配列の添え字として使用

    他にnumber, float, any.
```

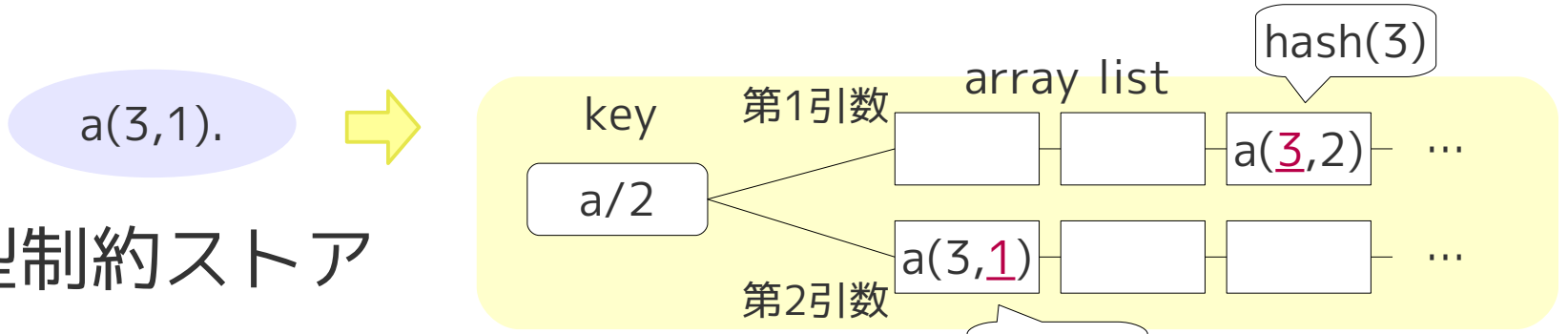
```
:- chr_constraint   edge(+int, +int).
```

int型の値が常に束縛される

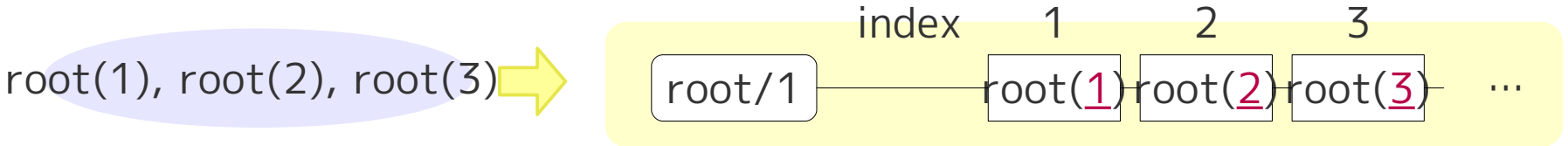
- 共有変数
  - partner制約の探索に共有変数を活用
  - マッチングの失敗を防ぐ

CHRにおける最適化(type/mode)

- type/modeを宣言
  - 制約の引数の値をキーとするハッシュ値を添え字とする



- 配列型制約ストア
  - dense_int型宣言された引数の値を配列の添え字に
    - dense：稠密(ちゅうみつ)な
  - コンパクトな制約ストア、ハッシュ値の計算が不要
    - 効果的に宣言すれば制約をダイレクトに取得可能



CHRにおける最適化(共有変数)

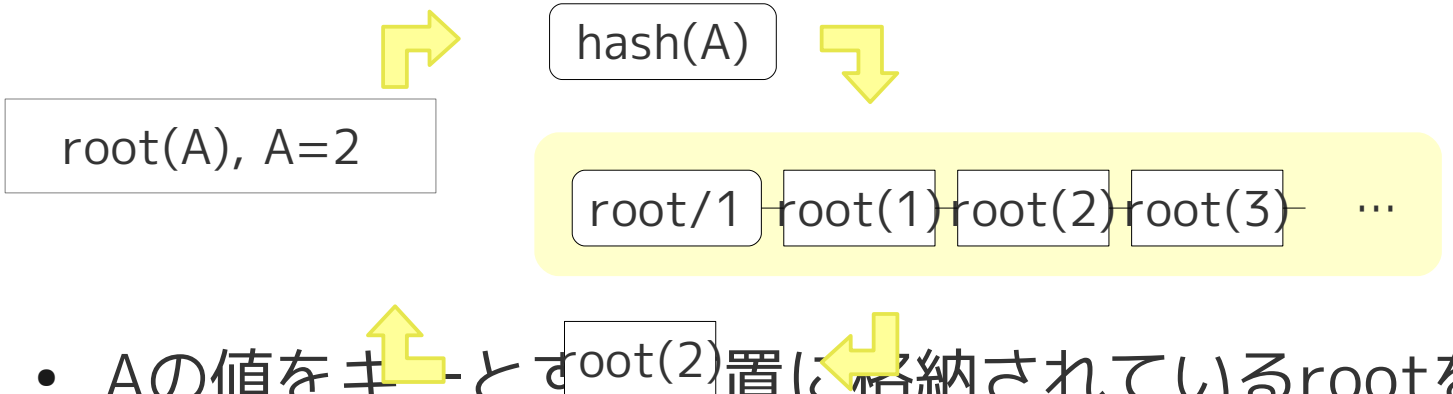
link(A,B), root(A) … <=> … (active, partner)

- 最適化オプション無し

ガードで判定するまで
マッチング成功/失敗が判定されない

≒ link(A1,B), root(A2) … <=> A1=A2 | …

- 最適化オプション有り

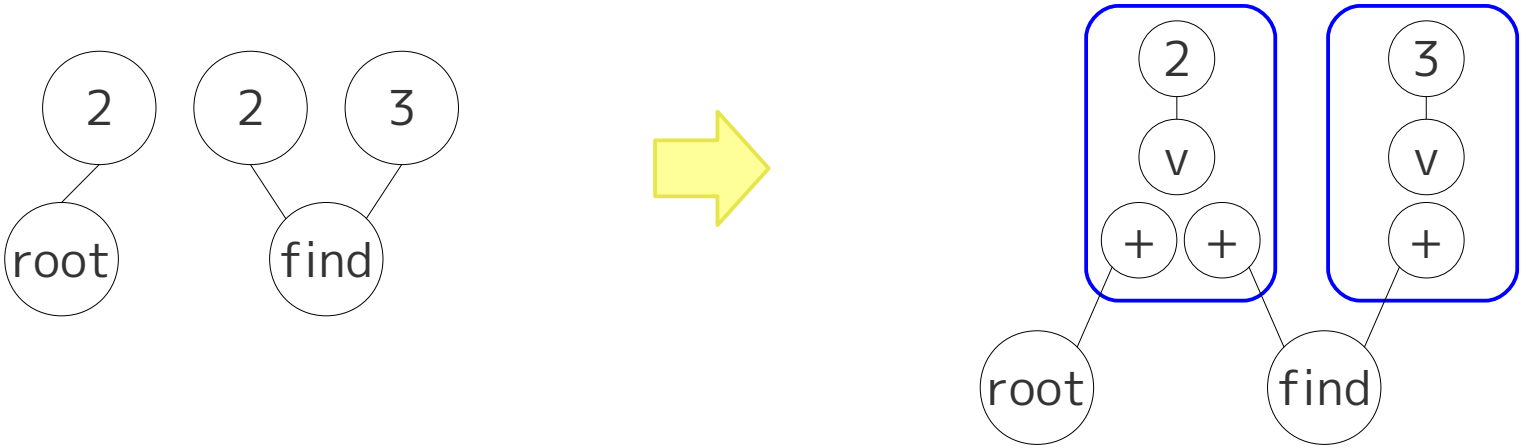


  - Aの値をキーとする位置に格納されているrootを取得
  - ヘッドにマッチした段階でA1=A2がほぼ確定されている
    - 求める制約を定数時間で探索可能

LMNtal：膜による共有構造の表現

root(2), link(2,3).

root(L1), link(L2,L3), {v(2), +L1, +L2}, {v(3), +L3}.



- 擬似的な共有変数
- 同じ値を参照しているアトム同士の接続関係をルールに反映

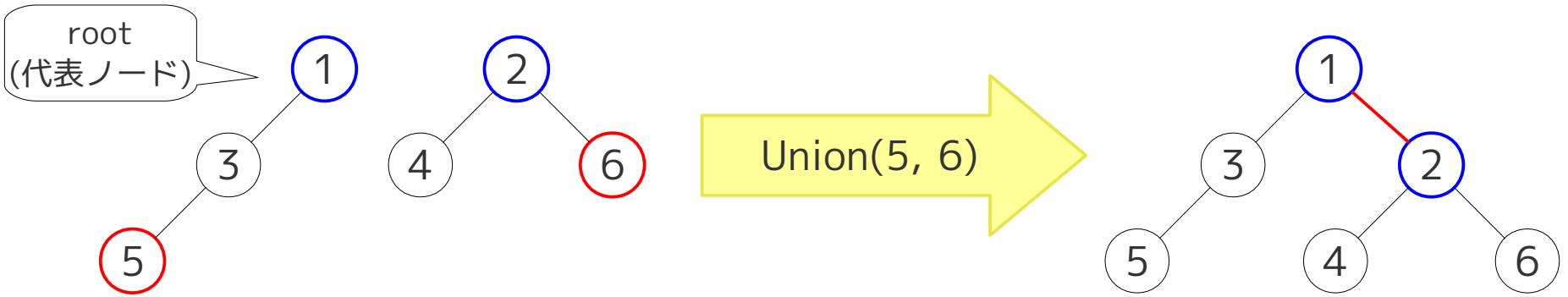
  link(A1, B), root(A2), {v(X), +A1, +A2} :- …

  = A1, A2が共通の値を参照していることがヘッドで分かる

  ⇒求めるアトムを定数時間で探索可能

例題：Union-Find algorithm (1/3)

- union命令による要素集合の管理手法[4]
- 木構造(+2つの最適化)が(漸近的に)最も高速に処理可能
  - union by rank, path compression
    （今回は簡単のため、2つの最適化は組み込まない）
    - make(X)：新しい木を生成（ノードはXのみ）
    - find(X)：要素Xが所属する木の名前（代表ノード）を返す
    - union(X, Y)：要素X, Yそれぞれが所属する木を連結する



例題：Union-Find algorithm (2/3)

- make命令
  - N個の木(1要素)を生成：O(*N*)
- union命令
  - union (= link)：O(1)
  - find (/union)：O(log(*N*)) (最悪O(*N*
  - make, union共にN回の呼び出し

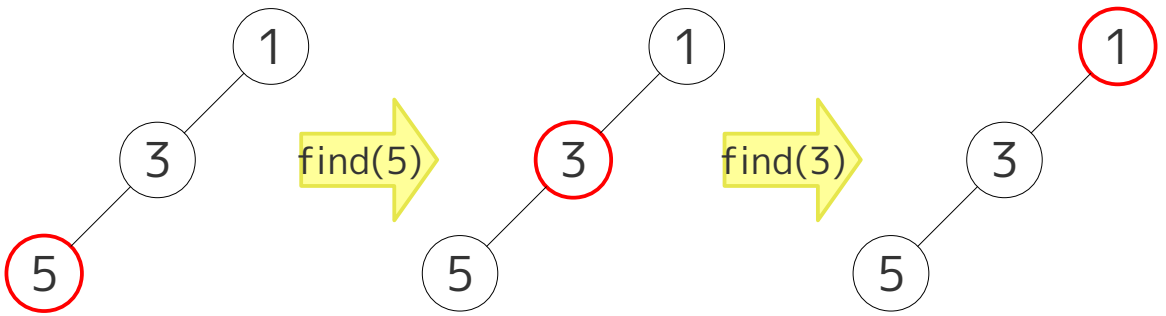
    ⇒最悪の計算量 (O(N^2))

```
make(X):
    p(X) ← X  (p(X)：Xの親)

union(X, Y)
    link( find(X), find(Y) )

link(X, Y)
    If X ≠ Y
        then p(Y) ← X

find(X)
    If X ≠ p[X]
        then return find(p(X))
        else return X
```



例題：Union-Find algorithm (3/3)

```
make @
  make(A) <=> root(A).
union @
  union(A,B) <=> find(A,X), find(B,Y), link(X,Y).
findNode @
  "~>"(A, B) \ find(A,X) <=> find(B,X).
findRoot @
  root(B) \ find(B,X) <=> X=B.
linkEq @
  link(A,A) <=> true.
link @
  link(A,B), root(A), root(B) <=>  "~>"(B, A), root(A).
```

"~>"(A, B) ⇔ BはAの親

N × N × N

link(A,B), root(A), root(B)

```
make(X):
    p(X) ← X  (p(X) : Xの親)

union(X, Y)
    link( find(X), find(Y) )

link(X, Y)
    If X ≠ Y
        then p(Y) ← X

find(X)
    If X ≠ p[X]
        then return find(p(X))
        else return X
```