2005年度 修士論文

LMNtalにおけるCHRの実現

提出日: 2006年2月3日
指導　: 上田 和紀 教授

早稲田大学理工学研究科　情報・ネットワーク専攻
学籍番号 : 3604u127-8

原 耕司

概 要

Constraint Handling Rules(CHR) は多重集合の書換えに基づく制約処理モデルであり，応用例も多い．CHR と LMNtal は共通点が多く，CHR のプログラムは字面上は LMNtal で書くことができるが，CHR にしかない概念である Propagation rule を現状の LMNtal で実行しようとすると無限ループに陥ってしまうという問題が残されていた．一方，反応終了判定や計算の局所化，階層構造の表現に使える膜の仕組みや分散・並列実行など LMNtal にあって CHR にない概念がある事も知られている．

本研究では，(1)CHR の操作的意味論をそのまま LMNtal プログラムとして記述，そのシミュレータ上でいくつかの CHR プログラムが動作することを確認した．また，(2) 履歴管理を行う Uniq ガード制約を LMNtal 処理系に追加し Propagation ルールを LMNtal 上でネイティブに実行できるようにし，(3) 不等式制約，グラフの閉路検出，スケジューリング，Path consistency，整数上の有限領域制約の各例題を LMNtal で記述して動作させることに成功した．

その結果，(1) 操作的意味論など構造の書き換えで表される言語仕様を LMNtal で記述する優位性が確認されると同時に (2)CHR のアプリが LMNtal 処理系でも高速に動作し，反応終了判定や階層構造，分散・並列実行など LMNtal ならではの機能を使って従来は困難であった計算が新たにできるようになったことが示せた．

目 次

図 目 次

表 目 次

第1章　研究の目的と背景

1.1　目的と意義

究極的な目的は,「計算機を使って難しい事を簡単にする」ところにある.

その基で，自分は計算機と人間の接点であるプログラミング言語の分野に注目した．自分が設計・開発に携わっているプログラミング言語 LMNtal をより強力にして，LMNtal の得意分野を広げるというのが本研究の大きな目的である.

プログラミング言語は用途によって向き不向きがあるので，こういうときはC,こういうときは Perl，こういうときは LMNtal などとうまく使い分けるのが妥当である.

数々のプログラミング言語の中で LMNtal は，実行手順の記述なしに問題の性質を宣言的に記述することができる宣言的プログラミング言語に分類される．その中でも特に (1) 計算が並行に進むモデルであり，(2) 計算は階層的なグラフ構造の書き換えであるとした点に特徴がある.

ところで，LMNtal と似た言語として Constraint Handling Rules(CHR) がある．CHR は多重集合の書換えに基づく制約処理モデルであり，LMNtal から階層構造を取り去った FlatLMNtal に近い．また，CHR は制約プログラミングの分野で既に広く使われており，豊富な例題のみならずスケジューリングなど日常業務の効率化と深く関わる問題を解く「製品」になっているものもある.

変数の扱いなど些細な違いはあるが，多重集合の書き換えにより計算が進むという特徴のため CHR のプログラムは字面上はほぼそのまま LMNtal で表現することができる．しかし，実行制御の点では Propagation ルールという CHR 固有の概念が存在する．これは同じ対象に対して重複適用を行わないという概念であり，これまでの LMNtal には存在しなかった．そのため，Propagation ルールを現状の LMNtal で実行しようとすると無限ループに陥ってしまうという問題が報告されていたのである.

そこで，Propagation ルールの仕組みを LMNtal に取り入れれば LMNtal の得意分野が広がるのではないかと考え，この仕組みを LMNtal に適した形で取り入れることにした.

一方，反応終了判定や計算の局所化，階層構造の表現に使える膜の仕組みや分散・並列実行など LMNtal にあって CHR にない概念が多々ある事もよく知られている．したがって，本研究の成果は単に LMNtal の得意分野が広がったという

だけの事ではなく，もともとあった CHR のアプリケーションから LMNtal にある様々な機能が使えるようになったという事をも意味するものである．したがって本研究を行う意義は大変大きい．

1.2 研究目標

CHR のアプリケーションが LMNtal 処理系でも高速に動作し，反応終了判定や階層構造，分散・並列実行など LMNtal ならではの機能を使って従来は困難であった計算が新たにできるようになったことを示すのが目標である．

そのために 2 つのアプローチを考え，実行した．

- 既存の LMNtal だけを使って CHR をシミュレートする
  （LMNtal の記述性の確認）
- LMNtal 処理系を拡張してより便利に CHR を実行する
  （LMNtal の守備範囲の拡張）

1.2.1 LMNtal の記述性の確認

λ計算，π計算，膜計算など，計算モデルが状態の書き換えとして表現される例は多い．また，構造の書き換えや局所化されたプロセスの反応終了判定は LMNtal の得意とする分野である．

実際，λ計算とπ計算はすでに LMNtal プログラムで表現できており，動作もしている．このような例題を増やすという意味でも，CHR を LMNtal でエンコーディングする研究はとても意義のあるものである．

そのようなわけで，操作的意味論 [1] をできるだけ忠実に LMNtal プログラム化することにした．

1.2.2 LMNtal の守備範囲の拡張

LMNtal 処理系にルール反応の履歴管理機能を追加することによって，CHR の Propagation ルールを LMNtal でネイティブに実行できるようにする．

これにより，Propagation ルール 1 つは字面上も意味上も LMNtal のルール 1 つで書き表すことができるので，CHR のアプリケーションを LMNtal に容易に書き直すことができる．すなわち LMNtal の守備範囲が広がったことになる．

1.3 本論文の構成

本論文では以降，次のような流れに沿って話をすすめる．

- **第 2 章**
  LMNtal 言語とその Java 版処理系について解説する．
- **第 3 章**
  CHR の構文，意味論などを解説する．
- **第 4 章**
  筆者が CHR の操作的意味論を元に設計・実装した CHR シミュレータである LMNtalCHR について述べる．
- **第 5 章**
  筆者が設計・実装したルール反応の履歴管理機能である Uniq ガード制約について述べる．
- **第 6 章**
  LMNtalCHR と Uniq ガード制約を使っていくつかの CHR の例題を LMNtal で記述し動作させた．ソースコードも交えて解説・考察する．
- **第 7 章**
  例題の 1 つ，閉路検出問題について，SICStus Prolog 上の CHR ライブラリと uniq 制約を使った LMNtal 処理系で速度比較を行った．
- **第 8 章**
  本研究の途中でいくつかの副産物，すなわちツールや処理系の機能が生まれた．それらの解説をする．
- **第 9 章**
  本研究のまとめを述べる．LMNtal 言語に関して研究中に考え wiki にメモした内容も合わせて掲載する．

第2章　LMNtal言語解説

本章では，LMNtal(Linked Multisets of Nodes transformation language) 言語仕様などについて軽く触れる．この章の記述は上田の LMNtal 発表論文 [15][14] 及びその後の議論に基づいたものである．特に，LMNtal 処理系を中心とした発表論文（原 SWoPP2005[6], 乾 PPL2006[18]）を基にしている．

2.1　LMNtal言語モデルとは

LMNtal は，(i) 計算，特に並行計算に関する多様なパラダイムを統合し，(ii) 極小規模および広域分散の両方向へ展開しつつある計算基盤の上で幅広く使うことを目指し我々が開発している言語モデルである．

グラフ内のリンク構造や膜構造を動的に変化させることによって，分散処理をはじめとする多様な計算を統一的かつ簡潔に記述することができる．

2.1.1　基本構文

LMNtal の構文は図 2.1 の通りである．P は LMNtal プログラムが扱う**プロセス**である．T はプロセスの書換え規則の表現に用いる**プロセステンプレート**であり，局所文脈（特定のセルの内部での文脈）を扱う機能をもつ．X_i は**リンク**，p は**名前**を表す．具体構文ではリンクは大文字から始まる識別子，名前はリンクと区別できる識別子で表記する．名前 = は，LMNtal 唯一の予約名である．

LMNtal プロセスは**プロセスのルール外には，同じリンクが 2 回を越えて出現してはならない**という**リンク条件**を満たさなければならない．

プロセス P のルール外に 1 回だけ出現するリンクを P の**自由リンク**と呼び，P に出現するそれ以外のリンクを P の**局所リンク**と呼ぶ．

直感的には，0 は中身のないプロセス，$p(X_1, \ldots, X_m)$ は m 本のリンクをもつアトム，P, P はプロセスの並列合成，$\{P\}$ は**膜** { } によってグループ化されたプロセス，ルール $T : -T$ は P のための書換え規則である．アトム $X = Y$ は，リンク X の一端と Y の一端とを接続する機能をもつ．

リンクは，2 つのアトムを一対一で接続する．リンク名は，接続先を識別するためのもので，文字列自体に意味はない．そのため，リンク名の 2 つの出現を同時に別の新しい名前に置き換える事ができる．

$$
\begin{array}{lll}
P ::= 0 & & (\text{空 / null}) \\
\quad | \quad p(X_1, \ldots, X_m) & (m \geq 0) & (\text{アトム / atom}) \\
\quad | \quad P, P & & (\text{分子 / molecule}) \\
\quad | \quad \{P\} & & (\text{セル / cell}) \\
\quad | \quad T \texttt{ :- } T & & (\text{ルール / rule}) \\
\\
T ::= 0 & & (\text{空}) \\
\quad | \quad p(X_1, \ldots, X_m) & (m \geq 0) & (\text{アトム}) \\
\quad | \quad T, T & & (\text{分子}) \\
\quad | \quad \{T\} & & (\text{セル}) \\
\quad | \quad T \texttt{ :- } T & & (\text{ルール}) \\
\quad | \quad \texttt{@}p & & (\text{ルール文脈}) \\
\quad | \quad \texttt{\$}p\texttt{[}X_1, \ldots, X_m \texttt{|} A\texttt{]} & & (\text{プロセス文脈}) \\
\quad | \quad p(*X_1, \ldots, *X_m) & (m > 0) & \\
 & & (\text{アトム集団 / aggregate}) \\
\\
A ::= \texttt{[]} & & (\text{空}) \dagger \\
\quad | \quad *X & & (\text{リンク束 / bundle})
\end{array}
$$

図 2.1: LMNtal の構文

ルールを膜に入れることができるので，膜は計算の局所化のために利用できる．ルールは，そのルールが所属する膜やその子孫膜の内容を書換えることができるが，親膜の内容を書換えることはできない．

ルール文脈は膜の中のすべてのルールの多重集合とマッチし，プロセス文脈は膜の中のルール以外のプロセスのうち，明示的に指定されていない全体とマッチする．個々のルール中のリンクや文脈の出現はいくつかの構文条件を満たさなければならない．

プロセス文脈の引数は，自由リンクの出現に関する制約条件を指定するものである．直感的には，ルール左辺のプロセス文脈 $\texttt{\$}p\texttt{[}X_1, \ldots, X_m \texttt{|} A\texttt{]}$ の引数 $X_1, \ldots, X_m$ は，その文脈が持っていなければならない自由リンクを指定している．剰余引数 A が $*V$ の場合，$*V$ はその文脈が持つ $X_1, \ldots, X_m$ 以外の 0 本以上の自由リンクの束を表し，[] の場合は $X_1, \ldots, X_m$ 以外に自由リンクがないことを表す．

ルール左辺の膜の後に “/” と記述すると，それ以上簡約不能な膜にしかマッチしなくなる．これは，子孫膜における計算終了の検出に利用できる．

膜の階層構造に関する議論をするときには，対象となるプロセス全体を含む仮想的な膜を考え，その膜を**世界的ルート膜**と呼ぶ．

2.1.2 ガード

現在公開している処理系では**ガード**を含むルールをサポートしている．ガードを含むルールは以下のように表される．

左辺 :- ガード | 右辺

ガードには，左辺にマッチしたプロセスが満たすべき付帯条件 (プロセスの型に関する制約等) を記述することができる．詳細は 2.2.7 節で説明する．

2.1.3 省略構文

グラフ構造が簡潔に記述できるように，いくつかの省略構文がサポートされている．

- アトム a の n 番目のリンクとして最終リンクを省略したアトム b を書くと，a の n 番目のリンクと b の最終リンクが繋がっているものとみなす．例えば `a(b(c))` は `a(A), b(C, A), c(C)` と等しい[1]．
- `p({...})` と書くと `p(A),{+A,..}`と等しい．`p`は多重集合の親とみなされる．

2.2 処理系の概要

現在公開されている LMNtal 処理系は約 27,000 行の Java コードから成り，Eclipse と CVS を用いて 10 名のメンバーによりチーム開発されている．実装上のノウハウの多くはプロトタイプ処理系 [16] から受け継いでいる．

処理系は，中間命令列へのコンパイル部分と中間命令列を実行するランタイム部分からなり，以下のパッケージから構成されている．

- compile (6,558 行)
  意味解析・中間コード生成のためのクラス
- compile.parser (5,364 行)
  構文解析時のデータ構造
- compile.structure (639 行)
  意味解析・中間コード生成時のデータ構造
- runtime (12,620 行)
  ランタイムシステムおよびデータ構造，コマンドラインインターフェース

[1] アトムは小文字，リンクは大文字から始まる文字列である．同じ名前のリンク同士は繋がっている．リンクを表す文字列自体に意味はない．a(A) と a(B) は等しい．

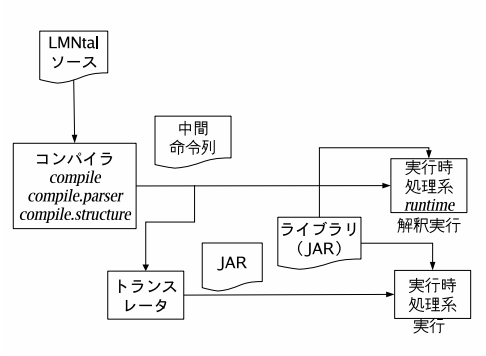


図 2.2: コンパイル・実行の手順とパッケージ構成

- java_cup (1,513 行)
  CUP，JFlex 付属のライブラリ
- daemon (1,873 行)
  分散処理系の通信部分
- util (1347 行)
  各種ユーティリティクラス
- graphic (1040 行)
  グラフィッククラス

図 2.2 の斜字体がパッケージ名を表している．

2.2.1 処理系の動作

コマンドラインより LMNtal 処理系を起動する．デフォルトでトランスレータ (2.2.13 節) が有効となっており，LMNtal ソースコードから中間命令 (2.2.2 節) を経由して Java コードが生成される．これにより高速に実行することが可能となっている．`--interpret` オプションを付加することで，LMNtal ソースコードを中間命令列までコンパイルし，実行時処理系が中間命令列を解釈実行することも出来る．これら一連の流れを図 2.2 に示す．

他に以下のようなコマンドラインオプションが用意されている．

- -s (シャッフルモード)
  膜内のルール選択や，ルールにマッチするアトムや膜の選択をランダムにすることができる (2.2.14 節)
- -g (可視化)
  グラフ構造を可視化することが出来る (2.2.11 節)

```
append(X0, Y, Z), cons(A, X, X0) :-
    cons(A, X1, Z), append(X, Y, X1).
append(X , Y, Z), nil(X)         :- Z=Y.
```

図 2.3: append プログラム

- -t (トレースモード)
  ルールが適用される度に膜を表示する (2.2.10 節)
- -d (デバッグモード)
  中間命令列 (2.2.2 節) を表示する

2.2.2 中間命令

中間命令は LMNtal 仮想マシン上で実行される逐次命令セットで，ルールをコンパイルすることにより生成される．あるルールの適用はそれに対応する中間命令列の実行によりなされる．どのルールの適用を試みるかはランタイムにより決定される．

中間命令は，どのプロセスがルール左辺とマッチするかを探し出すガード命令，マッチしたプロセスをルール右辺に置き換えるボディ命令の 2 種類に分類される．

図 2.3 のプログラムを例にして説明する．図 2.4，図 2.5 はコンパイルして出来た中間命令の一部である．

アトムの検査には，すでに見つかっているアトムとリンクで接続しているアトムを，deref 命令と func 命令を用いて優先的に検査する．それ以外のアトムは，findatom 命令と neqatom 命令を用いて検査する．膜の検査も同様で，すでに見つかっているアトムから特定可能な膜を優先的に取得し，検査の効率化を図る．

ボディ実行では，まず左辺にマッチしたプロセスを除去し，次に右辺に対応するプロセスを生成する．アトムの除去・生成は removeatom/newatom 命令で行う．リンクのつなぎ替えは次の 3 種類の命令で行う．

- 右辺に 2 回出現するリンクに対しては，newlink 命令を用いる．
- 右辺に 1 回出現するリンクのうち，‘=’ アトム以外に出現するリンクに対して，inheritlink 命令を用いる．この命令で使用するリンクの左辺での出現を取得するために getlink 命令を用いる．
- 右辺に 1 回出現するリンクのうち，‘=’ アトムに出現するリンクには unify 命令を用いる．

```
findatom   [1, 0, append_3]
deref      [2, 1, 0, 2]
func       [2,_cons_3]
...
removeatom    [1]
removeatom    [2]
newatom    [3, 0, cons_3]
newatom    [4, 0, append_3]
getlink    [5, 1, 1]
getlink    [6, 1, 2]
getlink    [7, 2, 0]
getlink    [8, 2, 1]
inheritlink   [3, 0, 7]
newlink       [3, 1, 4, 2]
inheritlink   [3, 2, 6]
inheritlink   [4, 0, 8]
inheritlink   [4, 1, 5]
```

図 2.4: 1 つ目のルールに対する中間命令の一部

2.2.3 最適化

中間命令列を最適化する最適化器 [?] がある．この最適化器は，主に以下の 2 つの最適化を行う．

- 冗長なプロセス削除・生成の除去
- 同じルールを連続して適用可能な場合の，マッチング検査やルール適用処理の簡略化

これにより，リスト処理などのリンクのつなぎ替えを行うルールに対して特に大きな効果がある．

左辺にマッチしたアトムを再利用して右辺のアトムを生成し，命令列の最適化を行う手順を以下に示す．

(1) 左辺・右辺のアトムの中から，再利用する組合せを決定する

(2) 再利用するアトムの除去・生成命令を取り除く

(3) 右辺のアトムを参照する変数番号を修正する

```
findatom   [1, 0, append_3]
deref      [2, 1, 0, 0]
func       [2,_nil_1]
...
removeatom    [1]
removeatom    [2]
unify         [1, 2, 1, 1]
```

図 2.5: 2 つ目のルールに対する中間命令の一部

```
findatom  [1, 0, append_3]
deref     [2, 1, 0, 2]
func         [2, cons_3]
...
getlink   [6, 1, 2]
getlink   [8, 2, 1]
newlink      [2, 1, 1, 2]
inheritlink  [2, 2, 6]
inheritlink  [1, 0, 8]
```

図 2.6: 最適化した中間命令

また，この処理により明らかに冗長な getlink 命令と inheritlink 命令が出来るので，これらは除去することが出来る．

例えば，図 2.3 の 1 つめのルールに対してアトム再利用を行うと，図 2.4 の命令列を図 2.6 のように最適化することが出来る．

2.2.4 膜を横切るリンクの実装

図 2.7 のように，膜を横切るリンクは inside_proxy， outside_proxy と呼ばれる自由リンク管理アトムを含むものとして実装されている．この機構により，通常のアトムのリンク先は必ず同じ膜内にある事になるのでリンクをつなぎかえる際のロック処理が単純になる．

2.2.5 モジュールシステム

実際に様々なアプリケーションを記述するために必要となるモジュールの宣言・読み込みがサポートされている．例えば標準ライブラリとして定義されている io モジュールは図 2.8 のように定義されている．

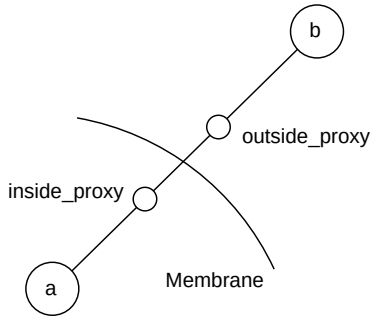


図 2.7: {a(X)}, b(X) の内部表現

```
{ module(io).
  ...
  io.input(Message, X) :- [:/*inline*/
    String s = javax.swing.JOptionPane.showInputDialog(null, me.nth(0));
    me.setName(done);
    me.nthAtom(0).setName(s);
    :](Message, X).
  ...
}
```

図 2.8: io モジュール

モジュールはモジュール名宣言とルールセットを含んだ膜である．ある膜内で *modulename.atomname* と記述した場合，モジュールに属するルールセットは暗にその膜にインポートされる．

この例は LMNtal における Java コードの使用についても示している．Java コードは [: と :] で囲まれた/*inline*/ で始まる特別なアトムとして現れる．これは **インライン実行アトム** と呼ばれる．コードは Java コードに展開され，ルール適用フェーズの最後に実行される．そのため，コードはルールの右辺によって組み立てられたグラフ構造にアクセス出来る．特別な変数 me は実行される Java コードのアトムを表し，mem はそれが属する膜を表している．i 番目の 引数はインラインコードで me.nth(i) のようにアクセスすることが出来る．

現在は以下のモジュールがライブラリとして公開されている．

- array
  配列を動的に作成してアクセスするモジュール
- float
  実数を扱うモジュール

- integer
  整数を扱うモジュール
- io
  入出力を扱うモジュール
- java
  Java 言語におけるインスタンス作成やメソッド呼び出しをサポートする．Java 言語の「文」を書かなくても良い時は，インラインの代わりにこれを使うことが出来る
- list
  リストを扱うモジュール
- socket
  ソケット通信のためのモジュール
- sys
  膜の整形テキスト出力やコマンドライン引数の解析などを扱うモジュール
- wt
  Window Toolkit モジュール

2.2.6 他言語インターフェース

アトムの一種であるインラインアトムの中に Java コードが書けるようになっていて，そのようなアトムが生成された直後にはそこに書かれたコードが実行される．この仕組みにより，入出力，ソケット通信，コマンドライン引数へのアクセスなど OS とのやりとりを担うライブラリが実現されている．また，インラインコードは誰でも書くことができるので LMNtal をグルー言語として使うこともできる．

2.2.7 ガード

ガード部分に `int(X)` と書くと，左辺に出現するリンク `X` の反対側が整数のときにのみルールが適用されるようになる．

このとき，ルール中のすべての `X` をマッチした特定の整数値 n で置換してできるルールを使って書き換えが行われる．このように，ガードを使うと，値の待ち合わせやデータの複製など実際のプログラムを記述するために必要となる多くの処理を簡潔に記述することが出来る．例えばバブルソートは図 2.9 のように記述出来る．このようなルール適用上の制約には，他に単一アトムを表す `unary`，float

```
L=[X,Y|L2] :- X>Y | L=[Y,X|L2].
ret=[78, 40, 49, 16, 41, 5].
```

図 2.9: バブルソート

型を表す float，自由リンクがない事を表す ground(2.2.8 節) がある．その他，型に束縛されたプロセス構造の履歴を反応時に記録しておき，重複適用を防ぐ uniq が用意された[2]．

ガードの構文は，アトム引数につながるデータの種類を推論し，プログラムの理解や実装の最適化を行うために役立てることができる．

2.2.8 ground

ground 型とは，閉じたネットワークでかつ出口が一つであるようなグラフ構造である．

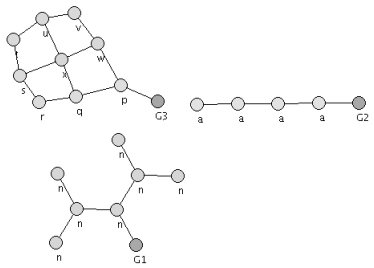


図 2.10: ground 型の例

```
G1=n(n(n,n),n(n,n)).
G2=a(a(a(a))).
G3=p(q(r(s(t(u(v(w(x(Q,S,U),W)),U)),S)),Q),W).
```

図 2.11: テキスト表現

ground 型の具体例を図 2.10 に示す．図 2.11 はそのテキスト表現である．

ground 型に関する型指定も，unary 型の場合と同様に記述することが出来る．

```
g(F),g(G) :-
    ground(F),ground(G),F=G | ().
```

このルールは，等しい構造を第 1 引数に持つアトム g を見つけ，それらの構造ともども削除する．ground 型の比較・破棄機能を使用している．

[2]CHR の Propagation rule と密接に関連している．

2.2.9 システムルールセット

四則演算をシステムルールセットとして処理系に組込むことによって，数値演算をサポートした．例えば

```
n(1),n(2),n(3),n(4),n(5),
(n(A),n(B):-n(A*B))
```

というプログラムは四則演算のルールにより

```
n(120),(n(A),n(B):-n(A*B))
```

と評価される．

LMNtal ではこのような組込みの演算以外にも，ルールを追加することによってユーザが新しくグローバルな演算を定義することも可能である．

2.2.10 トレースモードとルール名

-t オプションをつけて LMNtal システムを起動するとトレースモードが有効になる．トレースモードでは，ルール適用が行われる度にその時の膜の状態と適用されたルールが出力される．例えば

```
a:-b.  b:-c.  a.
```

というプログラムを実行すると

```
a, @601
 -->  #1: @601/a
b, @601
 -->  #2: @601/b
c, @601
```

と出力される．ここで--> #1: @601/a は，1 番目に，a というアトムから始まるルールが適用されたということを表している．また@601 は初期生成ルールを表す．

しかし，ルールが多くなるにつれてどのルールが適用されたかが分かりにくくなる．そこで我々は，ルール名@@をルールの前に付加して名前をつけるという機能を実装した．ルール名にはアトム名もしくはリンク名が使える．先のプログラムにルール名をつけると

```
rule1@@a:-b.  rule2@@b:-c.  a.
```

のようになり，トレースモードで実行すると

```
a, @601, @rule1@, @rule2@
 -->  \#1: @601/rule1
b, @601, @rule1@, @rule2@
 -->  \#2: @601/rule2
c, @601, @rule1@, @rule2@
```

と出力される．このようにルール名はトレースモードにおいて使用すると，どのルールが適用されたのかがより明確になり有用である．

2.2.11 可視化器

LMNtal で扱うデータは階層的なグラフ構造であるのでこれを可視化すれば反応経過が一目瞭然である．

処理系に -g オプションをつけて起動すると可視化機能が有効になる．ルールが 1 回適用されるたびに実行が中断されグラフ構造が表示される．Go ahead ボタンを押すと実行が再開する．

現在は膜の可視化（階層構造の表現）も実装されており，膜は四角形で囲まれて表現される．

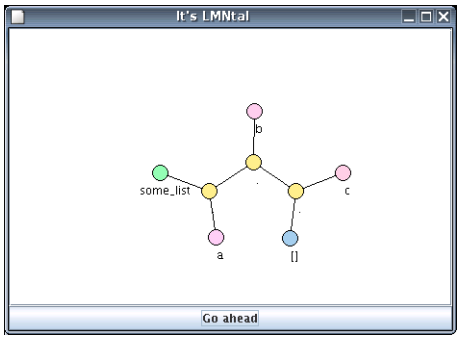


図 2.12: 可視化器

2.2.12 REPL (Read-Eval-Print-Loop)

スクリプト系言語によくある対話環境を開発した．処理系を REPL モードで起動するとシェルのようにプロンプトが表示される．LMNtal 言語の文を入力すると実行結果が表示され，再び入力待ちの状態になる．

2.2.13 トランスレータ

初期の処理系では，LMNtal プログラムを一度独自の中間命令列にコンパイルし，それを解釈実行しているため速度が遅いという問題があった．そこで我々は LMNtal プログラムを Java コードに変換して実行することにより，高速化することを行った．

比較のため 25 個，50 個，100 個，200 個のランダムな要素に対してバブルソート (図 2.9) を行う LMNtal プログラムを作成し，実行速度を計測した．

表 2.1: インタープリタとトランスレータの比較

実行方法	25 個	50 個	100 個	200 個
インタープリタ	0.096	0.297	1.733	10.608
トランスレータ	0.048	0.103	0.297	0.728

結果を表 2.1 に示す．単位はいずれも秒である．表 2.1 より，確かに速度が改善されていることが分かる．

現在の処理系ではデフォルトでトランスレータが有効となっており，この機能を使用しない場合は `--interpret` オプションをつけて LMNtal システムを起動する．

2.2.14 シャッフルモード

LMNtal では，書換え可能なプロセスや適用可能なルールが複数存在する場合の適用順序について規定されておらず，処理系が自由な順序で適用することができる．本処理系では，標準の実行モードにおいては LMNtal のソースコード中の順序に基づいて選択するプロセスの順序が決まっている．

しかし，これではプロセスの選択が規則的になってしまうため，プログラムによっては望ましい動作とならない場合がある．そこで，本処理系ではシャッフルモードを用意している．シャッフルモードを利用すると，膜内のルール選択や，ルールにマッチするアトムや膜の選択をランダムにすることができる．

2.2.15 開発環境の整備

処理系の開発に Eclipse を使用しており，Eclipse のテキストエディタで LMNtal プログラムを書くことがしばしばあった．そこで LMNtal ソースコードに色をつける Eclipse プラグインを開発するなど，開発環境の整備なども行っている．また秀丸エディタ用の強調表示ファイルも用意されている．

2.3 LMNtal の応用例

2.3.1 多重集合の書き換え

LMNtal は Gamma[2] のように多重集合の書き換えを表現することが出来る．LMNtal システムを対話モードで実行すると次のようになる．`--immediate` は，プログラムの入力後にリターンキーを押すとすぐに実行するオプションである (標準ではリターンキー 2 回で実行開始)．

```
$ lmntal --immediate
LMNtal version 0.67.20051127
Type :h to see help.
Type :q to quit.

# 1,1,1, {1,1,1,1,1,1, {1,1,1}, (1,1:-2)}
1, 1, 1, {2, 2, 1, {1, 1, 1}, @601}
# {out,a,b,c},d,{e,f},({out,$p[]} :- $p[]).
d, a, c, b, {f, e}, @603
```

ここでアットマークで始まるシンボルはコンパイルされた有効なルールの集合を表している．

最初の例は，2 つの 1 を 2 にする単純な多重集合の書き換えである．しかし膜は階層的多重集合を形成し，ルールは同じ膜内に所属するアトムに対してのみ局所的に適用される．しかしながら，2 番目の例のようにルールはセル (分子を膜で囲ったもの) を扱うことも出来，膜構造を変化させる．`$p[]` はプロセス文脈と呼び，それが属する膜の局所文脈を表現し，ワイルドカードのように振る舞う．

デフォルトでは入力した行はそれぞれ独立しているが，`--remain` オプションを付加することで前の行のガベージコレクションを抑制することが出来る．

2.3.2 リスト処理

LMNtal のリストは，

- 0 個以上の’.’という名前を持つアトム，
- それと同数の要素，および
- 1 個の’[]’アトム

をリンクで接続した分子 (molecule) として表現できる．’.’は 3 価 (3 引数) のアトムであり，第 1 引数として先頭要素へのリンク，第 2 引数として残りの要素のリストへのリンク，第 3 引数としてリスト全体を表すリンクを持つ．

A_i を第 i 要素へのリンクとすると，n 個の要素を持つリストの骨格は，テキスト表現では $\texttt{'.'}(A_1, X_1, X_0), \ldots, \texttt{'.'}(A_n, X_n, X_{n-1}), \texttt{'[]'}(X_n)$. となる ($X_0$ はこのリストを利用するプロセスにつながるリンクである)．これは略記法と構造合同 [?] の繰返し適用によって

$$X_0 = \texttt{'.'}(A_1, \texttt{'.'}(A_2, \ldots, \texttt{'.'}(A_n, \texttt{[]}) \ldots))$$

と略記でき，さらにこれに Prolog と同様の略記法を適用して $X_0 = [A_1, A_2, \ldots, A_n]$ と書けるようになっている．

この略記法を用いると，リスト連結プログラムは論理型言語風に

```
append([],Y,Z) :- Y=Z.
append([A|X],Y,Z0) :-
    Z0=[A|Z], append(X,Y,Z)
```

と書くことも，項書換え系のように

```
Z= append([],Y) :- Z= Y.
Z= append([A|X],Y) :- Z= [A|append(X,Y)].
```

と書くことも可能である．また，この略記法のおかげで，過去に書かれた並行論理プログラムの多くは，ほとんど変更なしに LMNtal プログラムとして実行することができる．

LMNtal は並行論理プログラムと異なり，変数の出現回数に強い制限がある．しかしながら，並行論理プログラムのルールにおいて 3 回以上出現するか 1 回以下しか出現しない変数 (非線形変数) は，データ構造の複製または破棄を表現しており，一方 LMNtal ではガード機能を用いてデータ構造の複製や破棄を表現することができる [17]．実際，非線形変数はガード機能を活用することによって LMNtal に容易にエンコードすることができる．

2.3.3 密に結合されたグラフ構造

ほとんどの宣言型言語はリストとツリーを扱えるが，循環構造や密に結合したデータ構造の扱いは未熟である．LMNtal の場合は違う．LMNtal の表現力を示すため，フラーレン (C_{60}) を組み立てる 2 つのルールと 2 つの初期アトムだけの簡単なプログラムを与える (図 2.14)．

最初のルールと次の 2 つの初期アトムが，2 つの 5 角形ドームを縫い合わせて行くことにより，20 面体 (20 個の三角形からなる多面体) を組み立てる．そして最後のルールがフラーレン構造に置き換える．図 2.13 は LMNtal システムの可視化オプション (-g) を使ってレンダリングしたグラフ構造を表している．

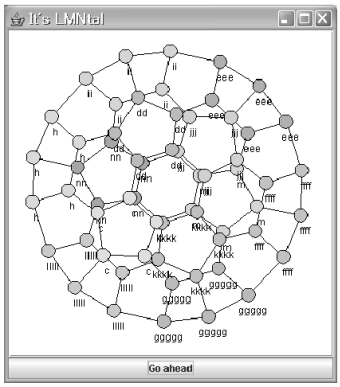


図 2.13: C_{60} 構造

```
dome(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9) :-
    p(T0,T1,T2,T3,T4), p(L0,L1,H0,T0,H4), p(L2,L3,H1,T1,H0),
    p(L4,L5,H2,T2,H1), p(L6,L7,H3,T3,H2), p(L8,L9,H4,T4,H3).
dome(E0,E1,E2,E3,E4,E5,E6,E7,E8,E9), /* top half */
dome(E0,E9,E8,E7,E6,E5,E4,E3,E2,E1). /* bottom half */

p(L0,L1,L2,L3,L4) :- X=c(L0,c(L1,c(L2,c(L3,c(L4,X))))).
```

図 2.14: C_{60} プログラム

2.3.4 他言語モデルとの関連付け

LMNtal は多重集合や並行処理などの概念を持つ計算モデルの統合を一つの目標としている．多重集合はグラフの特殊な場合であるので多重集合書き換えモデルが LMNtal に自然にエンコードできるのはほぼ明らかであるが，我々はそれに加えて，(i) 非決定的λ計算および名前呼びλ計算，(ii) 非同期π計算，などの基本演算を LMNtal にエンコードし，処理系上で動作確認を行った．また CHR(Constraint Handling Rules) の操作的意味論を LMNtal で表現して動作確認を行い，プロパゲーション機能の LMNtal への導入にも成功した．

λ 計算のエンコード

LMNtal でλ計算をエンコードすることが出来る．評価戦略を持たない非決定版の例を図 2.15 に示す．

これは Interaction Nets へのエンコード [11] を単純化したものである．最初のルールが λ 計算の本質であり，残りのルールは (λ 計算の) 変数の非線形的使用に対処するためにグラフ構造の複写や廃棄をするためのものである．

例えば Church 数の 2 は

```
beta@@ H=apply(lambda(A, B), C) :- H=B, A=C.

l_c@@ lambda(A,B)=cp(C,D) :- C=lambda(E,F), D=lambda(G,H), A=dp(E,G), B=dp(F,H).
a_c@@  apply(A,B)=cp(C,D) :- C= apply(E,F), D= apply(G,H), A=cp(E,G), B=cp(F,H).
l_d@@ lambda(A,B)=dp(C,D) :- C=lambda(E,F), D=lambda(G,H), A=dp(E,G), B=dp(F,H).
a_d@@  apply(A,B)=dp(C,D) :- C= apply(E,F), D= apply(G,H), A=dp(E,G), B=dp(F,H).
l_r@@ lambda(A,B)=rm :- A=rm, B=rm.
a_r@@  apply(A,B)=rm :- A=rm, B=rm.
c_r@@     cp(A,B)=rm :- A=rm, B=rm.
d_r@@     dp(A,B)=rm :- A=rm, B=rm.
r_r@@          rm=rm :- .

d_d@@ dp(A,B)=dp(C,D) :- A=C, B=D.
c_d@@ cp(A,B)=dp(C,D) :- C=cp(E,F), D=cp(G,H), A=dp(E,G), B=dp(F,H).
c_c@@ cp(A,B)=cp(C,D) :- C=cp(E,F), D=cp(G,H), A=cp(E,G), B=cp(F,H).

u_c@@ U=cp(A,B) :- unary(U) | A=U, B=U.
u_d@@ U=dp(A,B) :- unary(U) | A=U, B=U.
u_r@@ U=rm      :- unary(U) | .
```

図 2.15: 非決定的λ計算

```
N=n(2) :- N=lambda(cp(F0,F1), lambda(X, apply(F0,apply(F1,X)))).
N=n(3) :- N=lambda(cp(F0,cp(F1,F2)), lambda(X,
                  apply(F0,apply(F1,apply(F2,X))))).
res=apply(apply(apply(n(2), n(3)), s), 0).
H=apply(s, I) :- int(I) | H=I+1.
```

図 2.16: Church 数とその冪乗計算

```
lambda(cp(F0,F1),
       lambda(X,apply(F0,apply(F1,X))),
       Result).
```

と表現できる．そこで，図 2.16 のプログラムを実行すると `res=9` (と初期ルール) へと評価される (Church 数 m の n 乗は $\lambda mn.nm$ とエンコードされることを思い出してほしい)．図 2.16 に示したように，LMNtal におけるエンコーディングでは `s` のような自由名をもつアトムを使い，それをユーザが与えたルールによって評価することもできる．我々はさらに，名前呼びλ計算もエンコードし，不動点演算子 ($\mathbf{Y}$) を用いて表現した再帰関数の動作確認を行った．

非同期 π 計算のエンコード

図 2.17 は，非同期的 π 計算の通信メカニズムの一つのエンコーディングである．

```
snd@@ snd({$y[|*V]},X) :- {$y[|*V], m(X)}.
get@@ get({m(X),$y},Z), {$body[Z|*V]} :- {$y}, $body[X|*V].
cp@@ {name(N),$p[N|*Y],+Z}, Z=cp(Z0,Z1) :- {name(N),$p[N|*Y],+Z0,+Z1}.
rm@@ {name(N),$p[N|*Y],+Z}, Z=rm :- {name(N),$p[N|*Y]}.
```

図 2.17: 非同期的 π 計算

ここで π 計算の名前は，アトム name() および任意本数の入射リンク (+アトムによって終端されている) をもつセルによって表現している．このセルはまたメッセージバッファとしても機能し，m アトムで終端されたリンクには送信済かつ未受信のメッセージがつながっている．図 2.17 の最初のルールは x の y への送信を表現しており，二番目のルールは y からメッセージ x を受信して，フォーマル名 z をその受信メッセージに置き換えることを表現している．最後の二つのルール (cp はコピーを，rm は廃棄を表す) は，フォーマル名が 2 回以上使われたり全く使われなかったりした場合に使うためのものである．たとえば π 計算のプロセス $(a(z).b(y).\overline{z}\langle y\rangle) \mid \overline{a}\langle c\rangle \mid \overline{b}\langle d\rangle$ は

```
get(A0,Z), {get(B0,Y), {snd(Z,Y)}}.
snd(A1,C).  snd(B1,D).
{name(a),+A0,+A1}, {name(b),+B0,+B1},
{name(c),+C}, {name(d),+D}.
```

とエンコードできるが，これは

```
{name(a)}, {name(b)},
{name(c),m(_78)}, {name(d),'+'(_78)},
@601
```

に簡約される．この計算結果は，チャネル c が未消化のメッセージ d を保持する一方で，チャネル a と b はメッセージを保持せず，どのプロセスからも参照されていないことを表す．$(a(z).b(y).\overline{z}\langle y\rangle)$ のエンコードにおける膜の用法に注目してほしい．入力プレフィクスがついたプロセスのボディは最初は膜によって保護されており，get ルールが膜を除去した時点で実行を開始する．

第3章　CHR 解説

Constraint Handling Rules(CHR) は多重集合の書換えに基づく制約処理モデルであり，主に Prolog などのホスト言語上に実装されたライブラリとして提供される事が多い．

処理系としては各種 Prolog 上で動作するもののほか，Haskell や Java で書かれたもの，加藤によって klic で書かれたものがある．

応用例としては，不等式や区間，真偽値の制約，スケジューリング問題や述語推論など数多くの例題が WWW で公開されている[1]．

3.1　構文・意味

Simplification rule

$$Rulename@H_1, ..., H_i \Leftrightarrow G_1, ..., G_j | B_1, ..., B_k$$

Propagation rule

$$Rulename@H_1, ..., H_i \Rightarrow G_1, ..., G_j | B_1, ..., B_k$$

Simpagation rule

$$Rulename@H_1, ..., H_L H_{L+1}, ..., H_i <=> G_1, ..., G_j | B_1, ..., B_k$$

Simplification rule は，ヘッド H にマッチした構造 H' についてガード G が成り立ったら H' を消してボディ B を追加する．一方，Propagation rule は H' を消さずに残したまま B を追加する．Simpagation rule は $H'_1, ..., H'_L$ を残し，$H'_{L+1}, ..., H'_i$ を消して B を追加する．

この 3 種類のルールは構文上は異なるように見えるが，すべて同じ意味を持つ Simplification rule に直すことができる．

[1] WebCHR, 執筆時点では `http://bach.informatik.uni-ulm.de/~webchr/`

3.2 宣言的意味

Simplification rule

$$\forall(\exists(G_1 \wedge ... \wedge G_j)) \rightarrow (H_1 \wedge ... \wedge H_i \leftrightarrow \exists z(B_1 \wedge ... \wedge B_k))$$

Propagation rule

$$\forall(\exists(G_1 \wedge ... \wedge G_j)) \rightarrow (H_1 \wedge ... \wedge H_i \rightarrow \exists z(B_1 \wedge ... \wedge B_k))$$

3.3 状態

$$< Gs, Cu, Cb, T, V >$$

実行中の処理系の状態は上記の値で表される．

ゴールストア (Gs) 未処理の制約が入る．CHR プログラムに与えるデータは最初にここに入る．

ユーザ定義制約ストア (Cu) 通常の制約（`leq(x, y)` など）が入る．このストアに入っている制約を対象としてルール適用が行われる．

組み込み制約ストア (Cb) 組み込み制約（= など）が入る．

トークン (T) Propagation rule 実行メカニズムの肝となる仕組み．ゴールストアからユーザ定義制約ストアに制約が移される際，移される制約とすでにユーザ定義制約ストアにある制約により新たに反応できるようになった Propagation rule を計算しトークンストアに入れる．

変数 (V) Gs, Cu, Cb に出てくる変数一覧が入る．

3.4 初期状態・最終状態

初期状態

$$< Gs, \top, true, 0, V >$$

成功（但しこれ以上遷移できないこと）

$$< \top, Cu, Cb, T, V >$$

失敗

$$< Gs, Cu, false, T, V >$$

3.5 計算ステップ

成功するか失敗するまで，これらのステップのうち適用可能なものを非決定的に実行する．

なお，各ステップの直後に状態は正規化関数 N により正規化される．正規化関数では，等号 $=$ の伝播やトークンの重複削除などを行っている．

Solve 組み込み制約をゴールストアから組み込み制約ストアに移す．

Introduce ユーザ定義制約をゴールストアからユーザ定義制約ストアに移す．同時に $T(C, C_u)$ によりトークンを計算する．

Simplify ユーザ制約ストアから Simplification rule にマッチするものを見つけ，それがガードを満たしたらゴールストアにボディを入れる．マッチした制約はユーザ定義制約ストアから消される．

Propagate ユーザ定義制約ストアから Propagation rule にマッチするものを見つけ，マッチした構造がトークンストアに含まれることを確認する．さらにそれがガードを満たしたらゴールストアにボディを入れる．マッチした制約はユーザ定義制約ストアに残る．

3.6 計算例

図 3.1 は，次の不等式制約ルールと与えられた制約から $A = B \wedge B = C$ を導き出す過程を，状態と適用した計算ステップにより示したものである．

```
reflexivity  @ X leq X <=> true.
antisymmetry @ X leq Y , Y leq X <=> X=Y.
idempotence  @ X leq Y \ X leq Y <=> true.
transitivity @ X leq Y , Y leq Z ==> X leq Z.
leq(A, B), leq(B, C), leq(C, A).
```

$$
\begin{array}{ll}
 & <\underline{\mathtt{A} \leq \mathtt{B}} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \top, \mathtt{true}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Introduce}) & <\underline{\mathtt{C} \leq \mathtt{A}} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{A} \leq \mathtt{B}, \mathtt{true}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Introduce}) & <\underline{\mathtt{B} \leq \mathtt{C}}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{true}, \mathcal{T}_1, \mathcal{V}> \\
 & \mathcal{T}_1 = \{\mathtt{r3@C} \leq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}\} \\
\mapsto (\textbf{Introduce}) & <\top, \underline{\mathtt{A} \leq \mathtt{B}} \wedge \underline{\mathtt{C} \leq \mathtt{A}} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true}, \mathcal{T}_2, \mathcal{V}> \\
 & \mathcal{T}_2 = \mathcal{T}_1 \cup \{\mathtt{r3@A} \leq \mathtt{B} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{r3@B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{A}\} \\
\mapsto (\textbf{Propagate with } \mathtt{r3}) & <\underline{\mathtt{C} \leq \mathtt{B}}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true}, \mathcal{T}_3, \mathcal{V}> \\
 & \mathcal{T}_3 = \{\mathtt{r3@A} \leq \mathtt{B} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{r3@B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{A}\} \\
\mapsto (\textbf{Introduce}) & <\top, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \underline{\mathtt{B} \leq \mathtt{C}} \wedge \underline{\mathtt{C} \leq \mathtt{B}}, \mathtt{true}, \mathcal{T}_4, \mathcal{V}> \\
 & \mathcal{T}_4 = \mathcal{T}_3 \cup \{\mathtt{r3@C} \leq \mathtt{B} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{r3@B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{B}\} \\
\mapsto (\textbf{Simplify with } \mathtt{r2}) & <\underline{\mathtt{B} = \mathtt{C}}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{true}, \mathcal{T}_5, \mathcal{V}> \\
 & \mathcal{T}_5 = \{\mathtt{r3@C} \leq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}\} \\
\mapsto (\textbf{Solve}) & <\top, \underline{\mathtt{A} \leq \mathtt{B}} \wedge \underline{\mathtt{B} \leq \mathtt{A}}, \mathtt{B} = \mathtt{C}, \mathcal{T}_6, \mathcal{V}> \\
 & \mathcal{T}_6 = \{\mathtt{r3@B} \leq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}\} \\
\mapsto (\textbf{Simplify with } \mathtt{r2}) & <\underline{\mathtt{A} = \mathtt{B}}, \top, \mathtt{B} = \mathtt{C}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Solve}) & <\top, \top, \mathtt{A} = \mathtt{B} \wedge \mathtt{B} = \mathtt{C}, \emptyset, \mathcal{V}>
\end{array}
$$

図 3.1: 不等式制約の計算例

第4章　LMNtalCHR の設計と実装

CHR の動作をシミュレートするプログラムを LMNtal 言語で書き，これを LMNtalCHR と名づけた．この章では LMNtalCHR の設計と実装について詳細に述べる．

4.1　目的

LMNtal の記述性を確認するのが目的である．

λ計算，π計算，膜計算など，計算モデルが状態の書き換えとして表現される例は多い．また，構造の書き換えや局所化されたプロセスの反応終了判定は LMNtal の得意とする分野である．

実際，λ計算とπ計算はすでに LMNtal プログラムで表現できており，動作もしている．このような例題を増やすという意味でも，CHR を LMNtal でエンコーディングする研究はとても意義のあるものである．

そのようなわけで，操作的意味論 [1] をできるだけ忠実に LMNtal プログラム化することにした．

4.2　設計

4.2.1　要注意事項

CHR は Prolog をホスト言語としたライブラリという形で実装されている事が多い．

そのため，バックトラッキングや単一化など，CHR にはない Prolog ならではの機能を使った例題も多く存在する．

従って，LMNtal 上で実装する場合には Prolog 上の CHR で自然に採用されている仮定に気をつけないといけない．

4.2.2　変数の扱い

Prolog の論理変数に字面上対応するものは LMNtal ではリンクと呼ばれる．リンクは出現回数に制限があり，簡単にコピーできないなど性質が異なるので，代

わりに型付きプロセス文脈を Prolog の論理変数に対応させることにする．

4.2.3 多重集合のマッチング

CHR はヘッドに複数の制約を許している．例えば Prolog はヘッドに複数の項が表れることを禁じているため，Prolog で CHR 処理系を実装する場合は 1 つの CHR ルールにいくつかの Prolog ルールを対応させている．

一方 LMNtal はヘッドに複数の項が現れてもよいため，Simplification rule は 1 つの LMNtal ルールで実現することができる．

4.3 実装

4.3.1 終了判定

初期状態と終了状態の定義は次の通りであった．

初期状態

$$< Gs, \top, true, 0, V >$$

成功（但しこれ以上遷移できないこと）

$$< \top, Cu, Cb, T, V >$$

失敗

$$< Gs, Cu, false, T, V >$$

成功の「これ以上遷移できない」ことの判定には，LMNtal の反応終了判定 `{...}/`を使うことができた．

4.3.2 トークンの計算

Introduce ステップにより，追加される制約 C と現在のユーザ定義制約ストア Cu に対してトークン $T(C, Cu)$ が計算され，トークンストアに追加される．トークンの定義は，ある Propagation rule の左辺 H にマッチする H' を $C \wedge Cu$ から全て見つけ出し，そのルール名と組にして返すというものである．また，新たに追加された C は必ず H' に含まれる．

ここで，全て見つけ出すというのが曲者で，1 回使った制約を消さずに，しかも重複なくトークンを出力しなければならない．ここでは $C \wedge Cu$ から H にマッチする個数の制約を取り出す全ての組み合わせを作り，それぞれについて H とマッチするかどうかを LMNtal ルールを反応させてみて判定することにした．た

だし，新たに追加されたCは必ずH'に含まれるので，Cuからは$Hn-1$個選べばよい.

このために，任意のリストから k 個選ぶ全ての組み合わせをリストで返す`choose_k`（関連ルール7つ）を`list`モジュールに追加した.

4.3.3 lmntalchr.lmn 概要

こうして出来た意味論ベースの CHR シミュレータのルール数を以下に示す. このプログラムはモジュールとなっており，ユーザーが書く CHR プログラムに依存する部分は外部に書くことにした.

完全なソースコードは付録1を参照してほしい.

処理内容	ルール数
Solve	1
Introduce	1
等式伝播	4
終了判定	1
トークン計算	6
開始ルール	1

第5章　Uniqガード制約の設計と実装

Propatgation ルールを LMNtal ネイティブなルールとして記述できるように，Uniq ガード制約を実装した．この章では Uniq ガード制約 の設計と実装について詳細に述べる．

5.1　目的

前章で述べた LMNtalCHR（意味論ベースのシミュレータ）により，LMNtal の記述性の確認という目的は達成できた．

しかし，反応できる制約の組をトークンとして覚えておくというアルゴリズムは Propagation ルールの左辺の制約数に対して指数的な時間がかかってしまう．また，手動で Propagation ルールをシミュレータに適した形式に直した記述をしなければならず，負担が大きい．

一方，CHR 以外にも履歴管理があると問題が記述しやすくなるという分野もあると予想される．そこで，世の中の処理系の多くが採用している履歴管理方式を LMNtal 処理系に実装することにする．

5.2　設計

LMNtal ではガードを書くことで条件付きルールを記述することができる．

Propagation ルールも「同じ構造に対して 2 回以上実行しない」という条件をつけたルールであるので，ガードにより履歴管理を実現するのが適当であると判断した．

Propagation ルールを実行する際，左辺にマッチした変数の組を履歴と照合する．履歴にあったら実行せず，なかったら実行して変数の組を履歴に加える．

ここで，CHR で言う変数に相当する概念が LMNtal にはないことに注意する必要があった．構文上変数に相当するものは LMNtal ではリンクであるが，1 対 1 の関係を表すリンクの出現は 2 回までに制限されているため CHR の変数のように容易にコピーできない．また，CHR の変数束縛に相当するものは LMNtal ではリンク先の構造ということになるが，リンク先には任意のグラフ構造が繋がって

おり，膜を通過したりリンク元と循環構造を成しているかもしれず，構造に関する何らかの検査を行う必要がある．そこで，容易に比較・複製・破棄ができる型付きプロセス文脈を使うことにした．こうすれば，リンク先がどんな種類の構造になっているかを型チェックで検査した上で同一性の検査や複製・破棄ができる．

なお，型付きでない一般のプロセス文脈を使うこともできるはずであるが，現在の処理系では一般のプロセス文脈は膜で囲んだ記述をしなければならかったり，型付きプロセス文脈は略記構文が充実していたりするため型付きプロセス文脈を用いた．

5.2.1 構文

```
a(A), b(B), c(C) :- uniq(A, B, C) | d(A), e(B), f(C)
```

このように，LMNtal ルールのガードに uniq 制約を記述する．uniq の引数は可変個であり，各引数には型付きプロセス文脈を書く．

なお，上記の書き方は型付きプロセス文脈をリンクとして書いた省略形であり，省略しないで書くと次のようになる．

```
a(A), b(B), c(C), $a[A], $b[B], $c[C]
 :- uniq($a, $b, $c) |
d(A), e(B), f(C), $a[A], $b[B], $c[C].
```

5.2.2 uniq(A, B, C) の意味

- A, B, C は型付きプロセス文脈である必要がある．
  （型がついていなかったら ground チェックを課して ground にする）
- A, B, C に束縛された特定のプロセス構造の組み合わせでルールが反応するのは高々 1 回である．
- uniq 制約の引数が 0 個の場合，その uniq 制約が含まれるルールが反応するのは高々 1 回である．

したがって，1 回しか反応しないルールを次のように書くことができる．

```
:- uniq | 右辺.
```

5.2.3 想定する使用例

```
leq(X, Y), leq(Z, W) :- Y=Z, uniq(X, Y, Z, W) |
leq(X, Y), leq(Z, W), leq(X, W).
```

が

```
leq(a, b), leq(b, c).
```

にマッチした後の履歴の状態は

```
history == [ [ a, b, b, c ] ]
```

となる.

以後，`leq(a, b)`，`leq(b, c)` にマッチした時の uniq の処理では NG==4 となりガードが失敗する.

`X`，`Y`，`Z`，`W` のうち 1 つでも `a`，`b`，`b`，`c` と違うなら反応する. 例えば `leq(a, b)`，`leq(b, d)` は反応する.

5.3 実装

5.3.1 履歴管理アルゴリズム

擬似コードをアルゴリズム 1 に示す.

なお，GroundStrID はリンクを受け取ってリンク先の ground 構造を一意に識別する文字列を返す手続きであり,

$$\text{GroundStrID}(a) = \text{GroundStrID}(b) \Leftrightarrow a \text{ のリンク先} \equiv b \text{ のリンク先}$$

を満たす．GroundStrID の実装を図 5.1 に示す.

集合 *history* は `java.util.HashSet` を用いることができるので，このアルゴリズムの計算量は履歴の個数 N には依存せず，$O(1)$ である.

5.3.2 ガードコンパイラ

`compile.GuardCompiler#fixTypedProcesses` メソッドに uniq ガード制約をコンパイルする部分を追加した.

uniq の引数すべてについて，未束縛でないことを確認してから ground 制約を課し，変数番号のリストを含む UNIQ 中間命令を出力する.

```
public String groundString(){
  Set srcSet = new HashSet();
  Stack s = new Stack(); //リンクを積むスタック
  HashMap linkStr = new HashMap();
  StringBuffer sb = new StringBuffer();
  int linkNo=0;
  s.push(this);
  while(!s.isEmpty()){
    Link l = (Link)s.pop();
    Atom a = l.getAtom();
    if(srcSet.contains(a))continue; //既に辿ったアトム
    sb.append(a.getFunctor().getName());
    sb.append("(");
    srcSet.add(a);
    for(int i=0;i<a.getArity();i++){
      Link l0 = a.args[i];
      Link l1 = l0.getBuddy();
      if(!linkStr.containsKey(l0)) {
        String ss="L"+(linkNo++);
        linkStr.put(l0, ss);
        linkStr.put(l1, ss);
      }
      sb.append(linkStr.get(l0));
      if(i==l.getPos())continue;
      s.push(a.getArg(i));
    }
    sb.append(")");
  }
  return sb.toString();
}
```

図 5.1: GroundStrID の実装：runtime.Link#groundString

Algorithm 1 履歴管理アルゴリズム

Require: int $Arity$: uniq 引数の個数
Require: Array $binded$: uniq の引数番号 $\rightarrow$ 束縛されたプロセス構造
Require: GroundStrID : GroundStrID(a) = GroundStrID(b) $\Leftrightarrow$ a のリンク先 $\equiv$ b のリンク先
Ensure: 同一の $binded$ でのルール適用は高々 1 回である

```
1: Stringstr ← EmptyString
2: for i = 0 to Arity − 1 do
3:     str ← str + ” : ” + GroundStrID(binded[i])
4: end for
5: if str ∈ history then
6:     return FAIL
7: end if
8: history ← history ∪ str
9: return SUCCESS
```

5.3.3 runtime.Uniq クラス

`runtime.Uniq` クラスの仕様を述べる.

public HashSet historyH
履歴となるプロセス識別文字列を保存する集合である．要素は文字列型である.

boolean check(Link[] el)
アルゴリズム 1 の通りの動作をするメソッドである．Uniq ガード制約で束縛されたプロセス構造へのリンクの配列が渡され，プロセスの識別文字列を計算し，履歴と照合する．履歴にあったら `false` を返し，なかったら履歴に追加して `true` を返す.

5.4 旧版の実装

5.4.1 履歴管理アルゴリズム

アルゴリズム 2 は，実現が容易であったため最初の実装で用いたものである.
このアルゴリズムは履歴の個数 N に対して $O(N)$ の計算量で動作する．Uniq 制約の設計中はこのコードで試行錯誤したが，仕様が固まって実用的な規模の問題を解くにあたって前述の $O(1)$ のアルゴリズムに変更した.

アルゴリズム2における*binded*は`runtime.Link[]`，*history*は`runtime.Link[]`を要素とする`java.util.ArrayList`，`ProcessEqual`は工藤 [?] により実装された次のメソッドを利用した．

- `boolean Link#eqGround(Link srcLink)`
- `Link AbstractMembrane#copyGroundFrom(Link srcGround)`

`history`は`runtime.Link`の配列であるので，実際にプロセス構造を保存する膜`runtime.Membrane`を使用した．この膜は処理系内部で使用するだけのものであり，ユーザからは不可視である．

Algorithm 2 履歴管理アルゴリズム（旧版）

Require: int *Arity* : uniq 引数の個数
Require: Array *binded* : uniq の引数番号 → 束縛されたプロセス構造
Require: ProcessEqual : 2つのプロセス構造を比較する手続き
Ensure: 同一の *binded* でのルール適用は高々1回である

```
for all el in history do
  NG ← 0
  for i = 0 to Arity − 1 do
    if ProcessEqual(el[i], binded[i]) then
      NG ← NG + 1
    end if
  end for
  if NG == Arity then
    return FAIL
  end if
end for
history.add( binded )
return SUCCESS
```

5.4.2 runtime.Uniq クラス

`runtime.Uniq`クラスの仕様を述べる．

public ArrayList history
履歴を保存する配列である．要素は`runtime.Link[]`型である．

AbstractMembrane mem

履歴に追加されたリンク先のプロセス構造のコピーを保存する膜である．history に含まれる runtime.Link[] から参照される．

copyGroundFrom はリンクを返すが，そのそのリンクの反対側に繋がるプロセス構造はないため，hist_<hid>_<aid>という 1 価のアトムを繋げて保存している．<hid>は履歴内での添え字，<hid>は Uniq ガード制約の引数の添え字である．このアトムは生成しなくても問題ないが，デバッグのために履歴の内容を runtime.Dumper により印字したい時には必要になる．

boolean check(Link[] el)

アルゴリズム 1 の通りの動作をするメソッドである．Uniq ガード制約で束縛されたプロセス構造へのリンクの配列が渡され，履歴にあったら false を返す．なかったら copyGroundFrom を用いてプロセス構造をコピーし，履歴に追加して true を返す．

第6章　サンプルプログラムの解説と考察

6.1　不等式制約

不等式 ≤ に関する制約伝播を行う例題である.
CHR では次のように書ける.

```
reflexivity  @ X leq X <=> true.
antisymmetry @ X leq Y , Y leq X <=> X=Y.
idempotence  @ X leq Y \ X leq Y <=> true.
transitivity @ X leq Y , Y leq Z ==> X leq Z.
leq(A, B), leq(B, C), leq(C, A).
```

6.1.1　LMNtalCHR での記述例

```
results=lmntalchr.run(
  {c_=leq(a, b), c_=leq(b, c), c_=leq(c, a)},
  Simp, Replace, Find_token, 2).

Simp={
  // r2 @ leq(X, Y), leq(Y, X) <=> X=Y.
  r2_Simplify @@
  state({$gs}, {c_=leq(X0, Y0), c_=leq(Y1, X1), $cu[], @cu},
    {$cb}, {$t, @tr}, {$v})
  :- unary(X0), unary(Y0), unary(X1), unary(Y1), X0=X1, Y0=Y1 |
  nstate({eq(X0, Y0), $gs}, {$cu[], @cu}, {$cb}, {$t, @tr}, {$v}).

  // r3 @ leq(X, Y), leq(Y, Z) ==> leq(X, Z).
  r3_Propagate @@
  state({$gs}, {c_=leq(X, Y0), c_=leq(Y1, Z), $cu[], @cu},
    {$cb}, {{rule=r3, leq(TX, TY0), leq(TY1, TZ), $r_etc[]},
    $t, @tr}, {$v})
```

```
  :- Y0=Y1, X=TX, Y0=TY0, Y1=TY1, Z=TZ |
  nstate({c_=leq(X, Z), $gs},
    {c_=leq(X, Y0), c_=leq(Y1, Z), $cu[], @cu},
    {$cb}, {$t, @tr}, {$v}).
}.
Replace={
  replace(B, A), c_=leq(X, Y)
  :- unary(X), unary(Y), unary(A), unary(B), X=B |
  replace(B, A), c_=leq(A, Y).

  replace(B, A), c_=leq(X, Y)
  :- unary(X), unary(Y), unary(A), unary(B), Y=B |
  replace(B, A), c_=leq(X, A).
}.
Find_token={
  // トークン生成 r3 @ leq(X, Y), leq(Y, Z) ==> leq(X, Z).
  Found_token @@
  token2(ARG, {$w, {T0=leq(X, Y0), T1=leq(Y1, Z), $p[T0, T1]}})
  :- unary(X), unary(Y0), unary(Y1), unary(Z), Y0=Y1 |
  token2(ARG, {$w}),
  {rule=r3, leq(X, Y0), leq(Y1, Z), T0=kill, T1=kill, $p[T0, T1]}.
}.
```

6.1.2 実行結果

```
$ java -cp classes/ runtime.lmn -I sample/public/
  sample/public/lmntalchr_leq.lmn
results(_37),
  {c(eq(b)), b(eq(a)), c_(leq(a,c)),
    c_(leq(b,a)), c_(leq(b,c)), '+'(_37)},
  @607, @Solve@, @Introduce@, @Eq@, @Eq@,
  @Eq@, @Eq@, @Terminate@, @Start@
```

$a \leq b, b \leq c, c \leq a$ という 3 つの制約から $a = b = c$ という実行結果が得られた.

6.1.3 Uniq ガード制約を使った記述例

```
var.use.
```

```
Reflexivity @@
leq(var(X0), var(X1)) :- X0=X1 | var.unify(X0, X1).

Antisymmetry @@
leq(var(X0), var(Y0)), leq(var(Y1), var(X1)) :- X0=X1, Y0=Y1 |
var.unify(X0, Y0).

Transitivity @@
leq(var(X), var(Y0)), leq(var(Y1), var(Z)) :- Y0=Y1, uniq(X, Y0, Z) |
leq(var(X), var(Y0)), leq(var(Y1), var(Z)), leq(var(X), var(Z)).


time(N) :- M=N-1 | leq(var(0), var(M)), genleq(M).
genleq(N) :- N>0, M=N-1 | leq(var(N), var(M)), genleq(M).

time(10).
```

time(N) は，N 変数からなる 1 つの輪になった N 個の leq 制約を生成する．例えば，time(5) は

```
leq(var(0),var(4)).
leq(var(4),var(3)).
leq(var(3),var(2)).
leq(var(2),var(1)).
leq(var(1),var(0)).
```

を生成する．

6.1.4 実行結果

```
$ java -cp classes/ runtime.lmn sample/hara/leq.lmn
genleq(0), var_mem(_12222),
  {'+'(_12222), unify(9,0), unify(9,7), unify(9,5), unify(9,1),
    unify(9,3), unify(9,6), unify(9,8), unify(9,4), unify(9,2) }
```

var_mem に繋がる膜の中にある unify(9, 0) アトムは，変数 9 と変数 0 が等しいことを意味する．

10 変数，10 個の循環する不等式から，全ての変数が等しいという結果が得られた．

6.2 グラフの閉路検出

有向グラフから閉路を見つけ出す例題である．頂点 n から頂点 m への辺は e(n, m) と表される．

CHR では次のように書ける．

```
e(E0, E1), e(E1, E2), e(E2, E0) ==> loop(E0, E1, E2).
e(0, 1), e(1, 2), e(2, 0), e(0, 3), e(3, 2).
```

6.2.1 LMNtalCHR での記述例

```
results=lmntalchr.run(
  {c_=e(0, 1), c_=e(1, 2), c_=e(2, 0), c_=e(0, 3), c_=e(3, 2)},
  Simp, Replace, Find_token, 3).
Simp={
Propagate @@
state({$gs},
  {c_=e(E00, E10), c_=e(E11, E20), c_=e(E21, E01), $cu[], @cu},
  {$cb},
  {{rule=r3, e(T00, T10), e(T11, T20), e(T21, T01), $r_etc[]}, $t, @tr},
  {$v})
:- E00=E01, E10=E11, E20=E21, T00=T01, T10=T11,
   T20=T21, E00=T00, E10=T10, E20=T20,  |
nstate({c_=loop(E00, E10, E20), $gs},
  {c_=e(E00, E10), c_=e(E11, E20), c_=e(E21, E01), $cu[], @cu},
  {$cb},
  {$t, @tr},
  {$v}).
}.
Replace={}.
Find_token={
    // トークン生成 r3 @ leq(X, Y), leq(Y, Z) ==> leq(X, Z).
    Found_token @@
    token2(ARG,
      {$w, {C0=e(E00, E10), C1=e(E11, E20), C2=e(E21, E01),
        $p[C0, C1, C2]}})
    :-
    E00=E01, E10=E11, E20=E21 |
    token2(ARG, {$w}),
```

```
      {rule=r3, e(E00, E10), e(E11, E20), e(E21, E01),
        C0=kill, C1=kill, C2=kill, $p[C0, C1, C2]}.
}.
```

6.2.2 実行結果

```
$ java -cp classes/ runtime.lmn -I sample/public/
  sample/public/lmntalchr_edge.lmn
results(_7017),
  {c_(e(3,2)), c_(e(0,3)), c_(e(2,0)), c_(e(0,1)), c_(e(1,2)),
    c_(loop(3,2,0)), c_(loop(0,1,2)), '+'(_7017)},
  @606, @Solve@, @Introduce@, @Eq@, @Eq@,
  @Eq@, @Eq@, @Terminate@, @Start@
```

5 個の辺データから $(0,1,2),(0,3,2)$ という 2 つの閉路が得られた.

6.2.3 Uniq ガード制約を使った記述例

頂点を整数で表した版

```
e(E00, E10), e(E11, E20), e(E21, E01)
    :- E00=E01, E10=E11, E20=E21, uniq(E00, E10, E20) |
e(E00, E10), e(E11, E20), e(E21, E01), loop(E00, E10, E20).

e(0, 1), e(1, 2), e(2, 0), e(0, 3), e(3, 2).
```

頂点を特定膜へのリンクで表した版

次のコードは速度向上のために，頂点を頂点番号が所属する膜へのリンクで表した版である．性能比較は SICStus Prolog 上の CHR も含めて 7 章で述べる.

```
e(E00, E10), e(E11, E20), e(E21, E01),
{+E00,+E01,id(N0),$p0},{+E10,+E11,id(N1),$p1},{+E20,+E21,id(N2),$p2},
:- uniq(N0, N1, N2) |
e(E00, E10), e(E11, E20), e(E21, E01),
{+E00,+E01,id(N0),$p0},{+E10,+E11,id(N1),$p1},{+E20,+E21,id(N2),$p2},
loop(N0, N1, N2).

e(_60,_62), e(_38,_40), e(_94,_98), e(_82,_84), e(_126,_128),
```

```
  {id(0), '+'(_82), '+'(_94), '+'(_128)},
  {id(1), '+'(_60), '+'(_84)},
  {id(2), '+'(_40), '+'(_62), '+'(_126)},
  {id(3), '+'(_38), '+'(_98)}.
```

6.2.4 実行結果

頂点を整数で表した版

```
$ java -cp classes/ runtime.lmn sample/hara/edge.lmn
loop(0,1,2), loop(2,0,1), loop(2,0,3),
loop(1,2,0), loop(3,2,0), loop(0,3,2),
e(0,1), e(1,2), e(0,3), e(3,2), e(2,0), @601
```

頂点を特定膜へのリンクで表した版

```
$ java -cp classes/ runtime.lmn sample/hara/edge.lmn
loop(0,3,2), loop(0,1,2), loop(2,0,3),
loop(2,0,1), loop(3,2,0), loop(1,2,0),
e(_301,_303), e(_309,_311), e(_353,_355), e(_357,_359), e(_361,_363),
  {id(0), '+'(_309), '+'(_359), '+'(_361)},
  {id(1), '+'(_353), '+'(_363)},
  {id(3), '+'(_301), '+'(_311)},
  {id(2), '+'(_303), '+'(_355), '+'(_357)}, @601, @Edge@
```

6.3 スケジューリング

ある状態からある状態に達するまでに最低何時間かかるという制約が複数与えられた時に，最終状態に達するまでの最短時間を求める例題である．

状態 A から B になるために最低 7 単位時間かかることを `leq(A+7,B)` と表している．

CHR では次のように書ける．

```
%% 980312 Thom Fruehwirth, LMU

:- use_module(library(chr)).

handler scheduling.

option(check_guard_bindings, on).
```

```
constraints leq/2, optimize/0.
%% leq(X+N,Y) means: task X starts at least N time units before task Y

%% assumes leq-relation is non-circular

redundant @ leq(N,Y), leq(M,Y) <=> M=<N | leq(N,Y).

%% optimize gives smallest possible value to a variable

%% 残す \ 消す <=> RHS
optimize @  optimize#Id \ leq(X,Y) <=>
ground(X),var(Y),findall_constraints(Y,leq(_,Y),L),L=[]
|
Y is X
pragma passive(Id).

%% classical example ------------------------------------

build_house([Start,A,B,C,D,E,F,G,H,I,J,End]) :-
    leq(Start+0,A),
    leq(A+7,B),
    leq(A+7,D),
    leq(B+3,C),
    leq(C+1,E),
    leq(D+8,E),
    leq(C+1,G),
    leq(D+8,G),
    leq(D+8,F),
    leq(C+1,F),
    leq(F+1,H),
    leq(H+3,I),
    leq(G+1,J),
    leq(E+2,J),
    leq(I+2,J),
    leq(J+1,End),
    optimize,
    Start=0.

/*
実行例
| ?- build_house([Start,A,B,C,D,E,F,G,H,I,J,End]).

A = 0,
B = 7,
C = 10,
D = 7,
E = 15,
F = 15,
G = 15,
H = 16,
I = 19,
J = 21,
End = 22,
```

```
Start = 0,
optimize ?
*/

%% end of handler scheduling
```

6.3.1 Uniq ガード制約を使った記述例

```
/*
------------------------------------------------------------------------
Simple Scheduling [0]
                                                    Koji Hara
                                                    2006/01/04(水) 22:39:36

leq(X+N,Y) means: task X starts at least N time units before task Y

元ネタ：
 WebCHR
 http://bach.informatik.uni-ulm.de/~webchr/
------------------------------------------------------------------------
*/

// 最も早く終わらせたときの時間を決めて変数に束縛する
Optimize @@
{leq(_IN, var(_UV0)), dependCheck(_UV1, _IX), $e, @e}/
:- _UV0=_UV1 | {var.bind(_UV0, _IN), $e, @e}.


{
  Redundant @@
  leq(N,var(Y0)), leq(M,var(Y1)) :- M=<N, Y0=Y1 | leq(N,var(Y0)).

  // 変数の依存関係を調べる
  PRE_1 @@
  H=var_(_UV) :- uniq(_UV) | H=var_(_UV), dependCheck(_UV, 0).

  PRE_2 @@
  leq_(var_(_US)+_IX, var_(_UD0)), dependCheck(_UD1, N)
  :- _UD0=_UD1 |
  leq (var (_US)+_IX, var (_UD0)), dependCheck(_UD1, N+var(_US)).

  Problem_Generation @@
  build_house :-
    var.use,
    leq_(var_(start)+0,var_(a)),
    leq_(var_(a)+7,var_(b)),
    leq_(var_(a)+7,var_(d)),
    leq_(var_(b)+3,var_(c)),
    leq_(var_(c)+1,var_(e)),
    leq_(var_(d)+8,var_(e)),
    leq_(var_(c)+1,var_(g)),
    leq_(var_(d)+8,var_(g)),
```

```
    leq_(var_(d)+8,var_(f)),
    leq_(var_(c)+1,var_(f)),
    leq_(var_(f)+1,var_(h)),
    leq_(var_(h)+3,var_(i)),
    leq_(var_(g)+1,var_(j)),
    leq_(var_(e)+2,var_(j)),
    leq_(var_(i)+2,var_(j)),
    leq_(var_(j)+1,var_(end)),
    var.bind(start, 0).

  build_house.
}
```

6.3.2 実行結果

```
$ java -cp classes/ runtime.lmn sample/hara/schedule_simple.lmn
  {var_mem(_1220), dependCheck(start,0),
    {'+'(_1220), bind(start,0), bind(a,0), bind(b,7), bind(d,7),
      bind(c,10), bind(g,15), bind(f,15), bind(e,15), bind(h,16),
      bind(i,19), bind(j,21), bind(end,22),
      @604, @Unify_Pre_Find_Root@, @Unify_Pre_Compress_Path@,
      @Unify_Bind@},
    @601,@605, @Redundant@, @PRE_1@, @PRE_2@,
    @Problem_Generation@, @Module_Constructor_Var@,
    @Unify_Pre_OK@, @Var_New_Bind@, @Var_New_Unify@,
    @Var_New_Unify@, @Var_Unify@, @Var_Bind@},
  @602, @Optimize@
```

CHR の実行結果と同様に，start から end まで最低 22 単位時間かかるという結果が得られた．

6.3.3 工夫を要した点

Prolog 版での次のルールを，同等の処理をする LMNtal のルールに書き直す際に工夫を要した．

```
optimize @  optimize#Id \ leq(X,Y) <=>
ground(X),var(Y),findall_constraints(Y,leq(_,Y),L),L=[]
|
Y is X
pragma passive(Id).
```

このルールは，leq(X，Y) の X が束縛されていて，かつ Y が未束縛であるために実行が遅延されているような制約 leq(_，Y) が存在しない時に leq(X，Y) を消して Y に X を束縛するというものである．

LMNtal には変数の仕組みや findall_constraints ガード制約もないので，事前に変数間の依存関係を調べ,「ある変数が依存する全ての変数が束縛されたら」という条件チェックがなされるようにした．具体的には，ルール PRE_0，PRE_1 により dependCheck(j，0+var(g)+var(e)+var(i)) という形のプロセスが生成される．このプロセスは，変数 j は変数 g, e, i に依存することを表す．g, e, i が全て整数に束縛されると，dependCheck の第 2 引数が int ガード制約を満たすので，次に進めるという仕組みである．

6.4 整数の有限領域制約ソルバ

ある変数の値域がリストや最小値と最大値の組で表されるとき，変数の大小関係を使って値域を狭めていくという例題である．

制約を解くためのルールは次のものがある．

- $x \leq y \wedge x \in List1 \wedge y \in List2 \rightarrow y \in [min(List1), max(List2)] if min(List1) > min(List2)$

変数 x が $2, 3, 5, 7, 11, 13$ のいずれかであることを x :: [2,3,5,7,11,13]，変数 x が $x \in [18, 26]$ であることを x :: 18..26 と表す．

CHR では次のように書ける（一部省略した）．

```
%% intersection of domains for the same variable
X::L1, X::L2 <=> is_list(L1), is_list(L2) |
intersection(L1,L2,L) , X::L.

X::L, X::Min..Max <=> is_list(L) |
remove_lower(Min,L,L1), remove_higher(Max,L1,L2),
X::L2.


%% interaction with inequalities

X le Y, X::L1,  Y::L2 ==> is_list(L1),is_list(L2),
min_list(L1,MinX), min_list(L2,MinY), MinX > MinY |
max_list(L2,MaxY), Y::MinX..MaxY.
X le Y, X::L1,  Y::L2 ==> is_list(L1),is_list(L2),
max_list(L1,MaxX), max_list(L2,MaxY), MaxX > MaxY  |
```

```
min_list(L1,MinX), X::MinX..MaxY.

X lt Y, X::L1,  Y::L2 ==> is_list(L1), is_list(L2),
max_list(L1,MaxX), max_list(L2,MaxY),
MaxY1 is MaxY - 1, MaxY1 < MaxX |
min_list(L1,MinX), X::MinX..MaxY1.
X lt Y, X::L1,  Y::L2 ==> is_list(L1), is_list(L2),
min_list(L1,MinX), min_list(L2,MinY),
MinX1 is MinX + 1, MinX1 > MinY |
max_list(L2,MaxY), Y :: MinX1..MaxY.
```

6.4.1 Uniq ガード制約を使った記述例

```
/**
NAME
Finite (enumeration, list) domain solver over integers

SEE ALSO
WebCHR(http://bach.informatik.uni-ulm.de/~webchr/)

AUTHOR
Koji Hara

HISTORY
2006/01/28(土) 18:32:54

*/

list.use_guard.

%% special cases
Fail@@
X :: [] :- fail(X).

%% intersection of domains for the same variable
SameVariable0@@
X0 :: L1, X1 :: L2 :- X0=X1, custom_i_is_list(L1), custom_i_is_list(L2) |
X0 :: intersection(L1, L2).

SameVariable1@@
X0 :: L, X1 :: '..'(_IMin, _IMax) :- X0=X1, custom_i_is_list(L) |
X0 :: remove_higher(_IMax, remove_lower(_IMin, L)).

%% interaction with inequalities
```

```
Le0@@
le(X0, Y0), X1::L1, Y1::L2 :-
X0=X1, Y0=Y1, uniq(X0, Y0, L1, L2),
custom_i_is_list(L1), custom_i_is_list(L2),
custom_io_list_min(L1, MinX),
custom_io_list_min(L2, MinY),
custom_io_list_max(L2, MaxY),
MinX > MinY |
le(X0, Y0), X1::L1, Y1::L2, Y0::'..'(MinX, MaxY).

Le1@@
le(X0, Y0), X1::L1, Y1::L2 :-
X0=X1, Y0=Y1, uniq(X0, Y0, L1, L2),
custom_i_is_list(L1), custom_i_is_list(L2),
custom_io_list_max(L1, MaxX),
custom_io_list_max(L2, MaxY),
custom_io_list_min(L1, MinX),
MaxX > MaxY |
le(X0, Y0), X1::L1, Y1::L2, X0::'..'(MinX, MaxY).


Lt0@@
lt(X0, Y0), X1::L1, Y1::L2 :-
X0=X1, Y0=Y1, uniq(X0, Y0, L1, L2),
custom_i_is_list(L1), custom_i_is_list(L2),
custom_io_list_max(L1, MaxX),
custom_io_list_max(L2, MaxY),
custom_io_list_min(L1, MinX),
MaxY1=MaxY-1, MaxY1<MaxX |
lt(X0, Y0), X1::L1, Y1::L2, X0::'..'(MinX, MaxY1).

Lt1@@
lt(X0, Y0), X1::L1, Y1::L2 :-
X0=X1, Y0=Y1, uniq(X0, Y0, L1, L2),
custom_i_is_list(L1), custom_i_is_list(L2),
custom_io_list_min(L1, MinX),
custom_io_list_min(L2, MinY),
custom_io_list_max(L2, MaxY),
MinX1=MinX+1, MinX1>MinY |
lt(X0, Y0), X1::L1, Y1::L2, Y0::'..'(MinX1, MaxY).


%% auxiliary predicates ============================================

RemoveLower0@@
L1=remove_lower(_IMin, []) :- L1=[].
```

```
RemoveLower1@@
L1=remove_lower(_IMin, [H|T]) :- H<_IMin | L1=remove_lower(_IMin, T).
RemoveLower2@@
L1=remove_lower(_IMin, [H|T]) :- H>=_IMin | L1=[H|remove_lower(_IMin, T)].

RemoveHigher0 @@
L1=remove_higher(_IMax, []) :- L1=[].
RemoveHigher1 @@
L1=remove_higher(_IMax, [H|T]) :- H>_IMax | L1=remove_higher(_IMax, T).
RemoveHigher2 @@
L1=remove_higher(_IMax, [H|T]) :- H=<_IMax | L1=[H|remove_higher(_IMax, T)].

Intersection0 @@
L3=intersection([], L2) :- ground(L2) | L3=[].
Intersection1 @@
L3=intersection([H|T], L2) :- unary(H), custom_ii_member(H,L2)     |
 L3=[H|intersection(T, L2)].
Intersection2 @@
L3=intersection([H|T], L2) :- unary(H), custom_ii_not_member(H,L2) |
 L3=intersection(T, L2).
```

6.4.2 実行結果

```
x::[1,2,3,4,5,6,7], y::[2,4,6,7,8,0],
lt(y, x), x::'..'(4, 9), ne(x, y).
==>*
y(ne(x)), x(lt(y)), '::'(x,[4,5,6,7]), '::'(y,[2,4,6,0])


lt(x, y), x::[4,5,6],y::[2,3,4,5,6]
==>*
y(lt(x)), '::'(x,[4,5]), '::'(y,[5,6])


x::[3,4,5,6,7].
y::[3,4,5,6,7].
z::[3,4,5,6,7].
w::[3,4,5,6,7].
lt(x,y),lt(y,z),lt(z,w).
==>*
y(lt(x)), z(lt(y)), w(lt(z)),
'::'(x,[3,4]), '::'(z,[5,6]), '::'(y,[4,5]), '::'(w,[6,7])


x::[3,4,5,6,7].
y::[3,4,5,6,7].
```

```
z::[3,4,5,6,7].
w::[3,4,5,6,7].
lt(x,y),lt(y,z),lt(z,w),lt(w,x).
==>*
y(lt(x)),
y(fail), // 存在しない
z(lt(y)), x(lt(w)), w(lt(z)),
'::'(z,[4]),'::'(w,[5]), '::'(x,[6])
```

CHR と同様に，変数間の大小関係から変数の値域をせばめたり，値域が空になったら fail を生成することができた.

6.4.3 工夫を要した点

Prolog 版 CHR で使われているガード制約 is_list，max_list，min_list が LMNtal には用意されていなかったので，筆者が提案･実装したガードインラインの仕組みにより custom_i_is_list，custom_io_list_max，custom_io_list_min を実装した．これにより，Prolog 版 CHR と LMNtal 版 CHR のルールが 1 対 1 で対応するようにルールを変換することができた.

第7章　性能測定

例題の 1 つ，閉路検出問題について，SICStus Prolog 上の CHR ライブラリと uniq 制約を使った LMNtal 処理系で速度比較を行った．

7.1　ハードウェア

Intel(R) Pentium(R)M 1.3GHz, RAM 512MB

7.2　処理系と対象プログラム

測定対象は，異なる 5 頂点からなる閉路を検出するプログラムである．辺はランダムに頂点数の 10 倍の数を生成し，頂点数を変えて時間を測定した．ただし同じ頂点への辺は削除してある．

7.2.1　LMNtal

- LMNtal 0.70.20060105 Translator
- Uniq ガード使用
- 頂点データは頂点番号が含まれる膜へのリンク

```
workset={
  io.use.
  arg=sys.argv.
  arg=[] :- res=print(
  "\n\n** Usage <this program> <NumVertices> <DataTpye : int or link> **\n\n").
  arg=[_SNumVertices, _SDataType] :-
  vertices = string.int_of_str(_SNumVertices), dataType = _SDataType.

  vertices=_IV :- uniq | vertices=_IV, gen(integer.rndList(_IV, _IV*10)).
  vertices=_IV, dataType="link", linkRules={@r} :- uniq |
  vertices=_IV, dataType="link", genMem(_IV), @r.

  gen([]) :-.
  gen([A,B|T]) :- e(A, B), gen(T).
  e(A, B) :- A=B |.
```

```
  linkRules={
    genMem(0) :-.
    genMem(N) :- N>0, M=N-1 | genMem(M), {id(M)}.
    e(_I0, _I1), {id(N0),$p0}, {id(N1),$p1} :- N0=_I0, N1=_I1 |
    e(E0, E1), {+E0,id(N0),$p0}, {+E1,id(N1),$p1}.
  }.
  link_rule = {
    LoopLink@@
    e(E00, E10), e(E11, E20), e(E21, E30), e(E31, E40), e(E41, E01),
    {+E00,+E01,id(N0),$p0},
    {+E10,+E11,id(N1),$p1},
    {+E20,+E21,id(N2),$p2},
    {+E30,+E31,id(N3),$p3},
    {+E40,+E41,id(N4),$p4},
      :- uniq(N0, N1, N2, N3, N4) |
    e(E00, E10), e(E11, E20), e(E21, E30), e(E31, E40), e(E41, E01),
    {+E00,+E01,id(N0),$p0},
    {+E10,+E11,id(N1),$p1},
    {+E20,+E21,id(N2),$p2},
    {+E30,+E31,id(N3),$p3},
    {+E40,+E41,id(N4),$p4},
    loop(N0, N1, N2, N3, N4).
  }.
  B@@go, dataType="link", link_rule={@r} :- dataType="link", @r.
}.

io.use.
workset={$e, @e}/ :- time=timer.benchmark({$e, @e}).
time=_F :- res=print(string.join("", ["",string.str_of_float(_F),""])).
```

7.2.2 SICStus Prolog

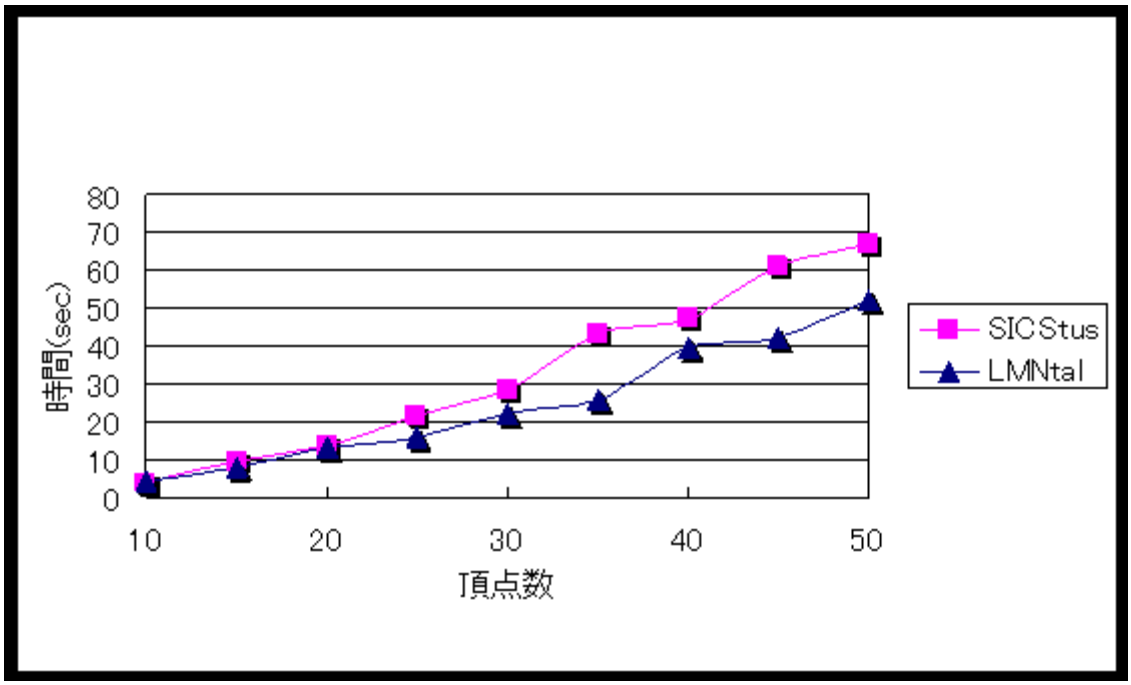
- SICStus Prolog 3.12.3 (x86-win32-nt-4): Thu Oct 27 17:58:10 WEST 2005 コンパイル
- CHR ライブラリ使用
- 頂点データは Prolog の変数

5 頂点は全て異なるという条件を表すために，ガードにて E0〜E4 が全て異なるというチェックを入れている．

```
:- use_module(library(chr)).
handler leq.
constraints e/2, loop/5.

e(E0, E1), e(E1, E2), e(E2, E3), e(E3, E4), e(E4, E0) ==>
E0\==E1, E0\==E2, E0\==E3, E0\==E4,
E1\==E2, E1\==E3, E1\==E4,
E2\==E3, E2\==E4,
```

```
E3\==E4
 | loop(E0, E1, E2, E3, E4).

goL(N):-
    cputime(X),
    length(L, N),
    genL(L, N, N*10),
    cputime( Now),
    Time is Now-X,
    write(Time), nl.

genL(_, _, 0).
genL(L, N, Co) :-
    rndInt(N, X0), rndInt(N, X1),
    nth(X0, L, Z0), nth(X1, L, Z1),
    tryL(N, Co, L, X0, X1, Z0, Z1),
    Ne is Co-1, genL(L, N, Ne).

tryL(_, _, _, X, X, _, _) :- ground(X).
tryL(_, _, _, X0, X1, Z0, Z1) :- X0=\=X1, e(Z0, Z1).

nth(0, [X|_], X).
nth(N, [_|L], X) :- N > 0, N1 is N-1, nth(N1, L, X).

rndInt(N, Res) :- Res is random(N).

cputime(Ts) :-
    statistics( runtime, [Tm,_]),
    Ts is Tm/1000.
```

7.3 結果のグラフ



7.4 考察

この問題に関しては，LMNtal 版 CHR の方が良い性能を出している．

LMNtal, SICStus 両方とも頂点を変数で表しているが，SICStus の変数とそれに対する操作（同一性検査，単一化，束縛）は処理系に組み込まれているものであるのに対し，LMNtal の変数とそれに対する操作は var モジュールにより提供され，LMNtal プロセスとルールにより構成されている．これは LMNtal 版 CHR にとって大きく不利であると予想されるが，それにも関わらず LMNtal 版 CHR の性能が勝っている理由の 1 つに「頂点は全て異なる」という条件があると考えられる．

var モジュールを用いた LMNtal 版のプログラムでは，変数 x は{name=x}というプロセスが存在することで表され，どこかからその変数を参照するような場合には{name=x, +Occ}, somewhere(Occ) というように参照元から変数を表す膜へのリンクを張ることにより表される．このため，ループ検出ルールにて 5 つの変数を表す膜を別々に書くことで「5 つの頂点は全て異なる」という暗黙の条件が付加されたことになる．

これに対して，SICStus 版 CHR では，5 つの変数をそのまま書くと「5 つの頂点は全て異なる」という条件は加わらず，同じ変数が 2 つ以上含まれてもマッチして

しまう．そのためガードにてわざわざ全て異なるという制約を記述した．SICStus 版 CHR はこの部分が実行時間の大半を占めていると予想される[1]．

[1]実際，異なるというガード制約をなくしたところ，ほぼ一瞬で実行が終わった

第8章　他に作ったもの

8.1　ガードインライン

ユーザーが処理系のコードに手を出すことなく新しいガードを定義できるように，ガードインライン機構を設計・実装した．

ユーザーは，次のことを実現する任意の Java コードを記述することができる．

- 入力宣言された引数の値 (unary なら Atom, ground なら Link) にアクセスできる．
- 出力宣言された引数（unary 型限定）に値を束縛することができる．組み込みガードと同様，束縛された値はガード及びボディで使用可能．
- ガードの成功／失敗を表す真偽値をガードの処理結果として返すことができる．

8.1.1　定義方法

CustomGuard インターフェースを実装し，/*__UNITNAME__*/CustomGuardImpl という名前を持つクラスをインライン宣言アトムとして LMNtal ソースファイルに記述する．ライブラリとして使用するファイルに書くこともできる．

/*__UNITNAME__*/の部分は，コンパイラによって現在のインラインユニットに対応するクラス名に書き換えられる．これにより，複数のファイルで定義されたカスタムガード制約定義クラスの名前衝突を防ぐことができる．

/*__PACKAGE__*/の部分は，インタプリタ動作かトランスレートによるコンパイル動作かによりそれぞれ適切なパッケージ宣言に置換される．

また，//#で始まる行は，その行のそれ以降の文字列をファイル名とみなし，その行以降の内容をインラインコードと同じディレクトリの指定ファイルに出力する．

```
[:/*inline_define*/
//#/*__UNITNAME__*/CustomGuardImpl.java
/*__PACKAGE__*/
import runtime.*;
import java.util.*;
```

```
public class /*__UNITNAME__*/CustomGuardImpl implements CustomGuard {
    public boolean run(String guardID, Membrane mem, Object obj) {
        // ary にはガード制約に渡された引数が入る
        ArrayList ary = (ArrayList)obj;

        if(guardID.equals("genInt")) {
            // 第1引数に 13 を生成する
            // a. a :- N=custom_o_genInt | generated(N).
            // ガードでアトムを生成するときは mem.newAtom ではなく new Atom を使う
            ary.set(0, new Atom(null, new IntegerFunctor(13)));
            return true; // 成功
        }
        else if(guardID.equals("x5")) {
            // 第1引数の数値 (int 型) を 5 倍して第2引数に束縛する
            // b. b :- N=custom_io_x5(10) | generated(N).
            Atom a = (Atom)ary.get(0);
            int v = ((IntegerFunctor)a.getFunctor()).intValue();
            ary.set(1, new Atom(null, new IntegerFunctor(v * 5)));
            return true; // 成功
        }
        return false; // ガードが未定義なら失敗
    }
}
//#
:].
```

8.1.2 実行方法

一般のガードが書ける部分に,

> custom_<入出力定義文字列>_<カスタムガード名>(引数1, 引数2, ..., 引数N)

と書く.

入出力定義文字列はガードに渡された各引数が入力引数か出力引数かを定義するものである. n 文字目が i の時, 引数 n は入力引数である. n 文字目が o の時, 引数 n は出力引数である. 入出力定義文字列の長さはガードのアリティと等しい必要がある.

ちなみに, 一般のガードも含めて, ガードに記述する順番は実行に何の影響も与えない. 未束縛の入力引数を持つガード制約の処理は, その変数を束縛するガード制約が処理されたのちに行われるようにコンパイラが順番を決定するためである.

8.1.3 使用例

Le0@@

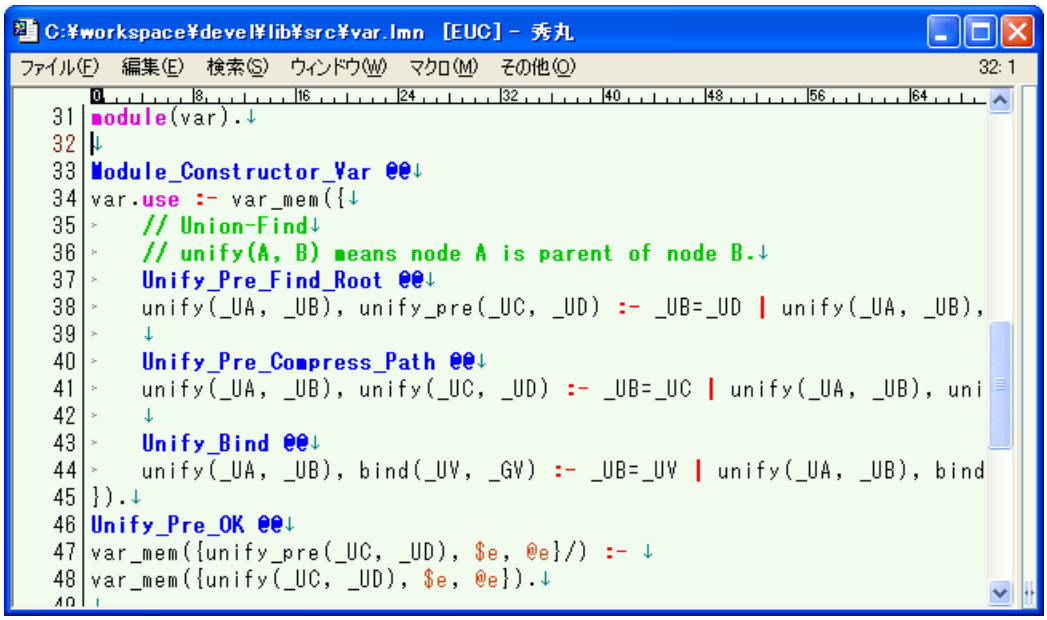
```
le(X0, Y0), X1::L1, Y1::L2 :-
X0=X1, Y0=Y1, uniq(X0, Y0, L1, L2),
custom_i_is_list(L1), // L1 がリストであることを確認する
custom_i_is_list(L2),
custom_io_list_min(L1, MinX), // 整数リスト L1 の最小値を MinX に束縛する
custom_io_list_min(L2, MinY),
custom_io_list_max(L2, MaxY), // 整数リスト L2 の最大値を MaxY に束縛する
MinX > MinY |
le(X0, Y0), X1::L1, Y1::L2, Y0::'..'(MinX, MaxY).
```

8.2 ルール名

LMNtal のルールに名前をつけられるようにした．これによりデバッグの効率やソースの可読性が高まったが，これは元々 CHR の構文をヒントに着想したものである．

8.3 秀丸エディタ用強調表示ファイル

:-, |, int, unary, ground, プロセス文脈, リンク, リストなどにマッチする正規表現によりソースコードを見やすくした[1] ．



[1] 言語班 wiki においてあります．http://www.ueda.info.waseda.ac.jp/lmntal/local/

8.4 膜のインデント

処理系に-x dump 1オプションを設けた．この状態でトレース表示(-t)を行うと膜の出力が始まる直前に改行し，膜自体もインデントさせるようにした．

出力例を以下に示す（長い行は適度に改行してある）．

```
results(_70),
  {introduce, solve, propagate, simplify, lmntalchr.use, '+'(_70),
    tstate(_2746,_2748,_2750,_2752,_2754),
    {'+'(_2754)},
    {'+'(_2746)},
    {c(eq(b)), eq([]), b(eq(a)), '+'(_2750)},
    {found_token, 'prop_head_co..'(2), '+'(_2752),
      token1(_2995,_2997,_2999),
      {'+'(_2995),
        {c_store, c_(leq(b,c))},
        {c_(leq(a,c)), c_(leq(b,a))}},
      {'+'(_2999),
        {l([leq(b,a)]), l(leq(b,c))},
        {l([leq(a,c)]), l(leq(b,c))}},
      {list([]), '+'(_2997), @609},
      {c(leq(b)), l([kill]), l(kill), b(leq(c)), rule(r3)},
      @603, @605, @Fount_token@},
    {c_(leq(a,c)), c_(leq(b,a)), c_(leq(b,c)), '+'(_2748), @602},
    @601, @606, @607, @r2_Simplify@, @r3_Propagate@,
    @TokenFinished@, @Solve@, @Introduce@, @Eq@,@Eq@, @Eq@,
    @Eq@, @Terminate@, @Start@}, @607, @Solve@, @Introduce@,
    @Eq@, @Eq@, @Eq@, @Eq@, @Terminate@, @Start@
```

8.5 HTML化ツール LMNtal Trace Viewer

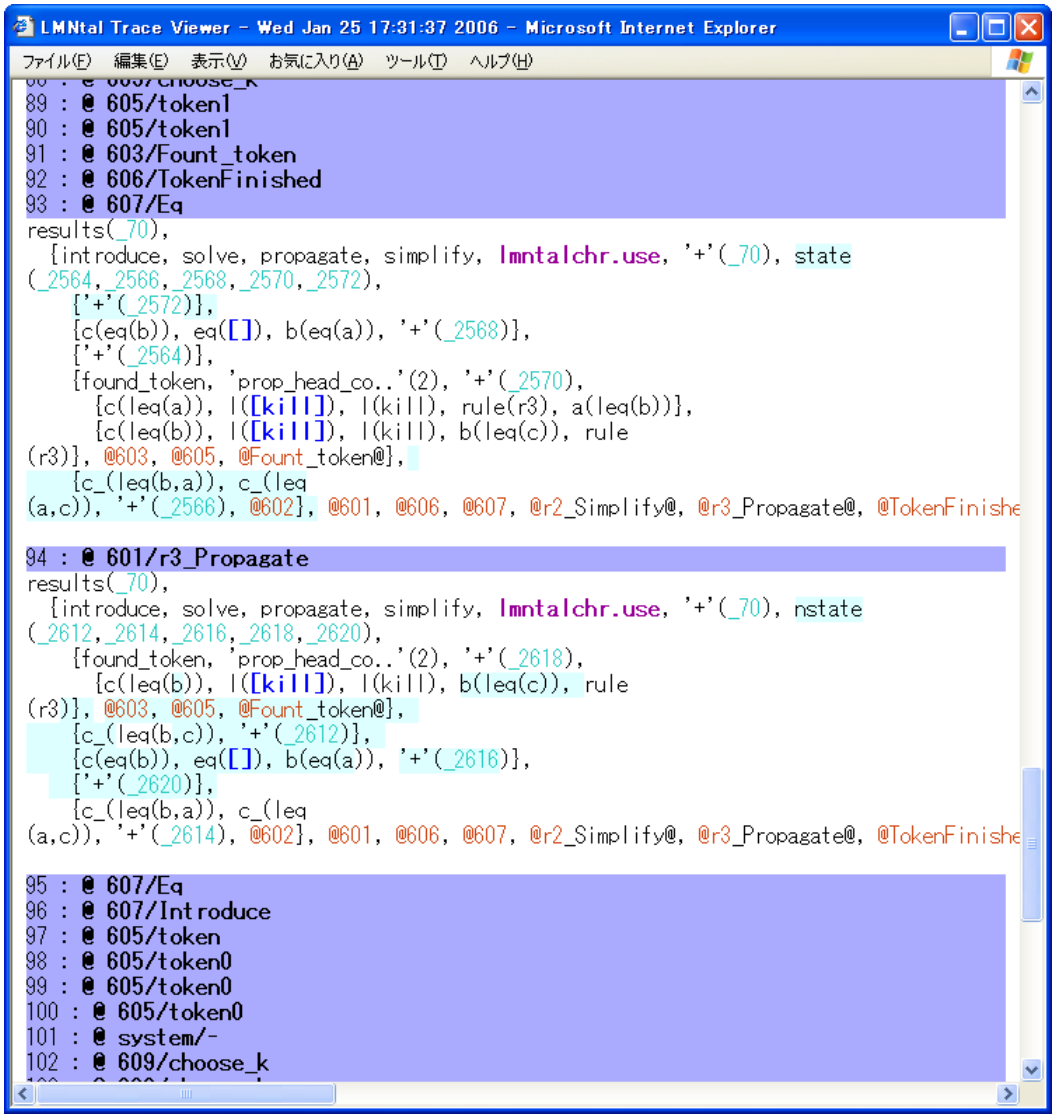
トレース表示をHTML化するツール(tohtml.pl)を書き，LMNtal Trace Viewerと名づけた．[2]．

図8.1にLMNtal Trace Viewerの出力をブラウザで開いた例を示す．この時のコマンドラインは次の通りである．

```
$ bin/lmntal_cyg --interpret --use-source-library -x dump 1
  -I sample/public/ sample/public/lmntalchr_leq.lmn -t
  | perl tohtml.pl .* +Prop> result.html
```

[2]CVSにおいてあります．

図 8.1: LMNtal Trace Viewer の出力



8.5.1 概要

この算譜は，-t と -x dump 1 オプションをつけた LMNtal 処理系の出力を標準入力から受け取り，HTML 化して標準出力に出力するものである．

8.5.2 主な機能

(1) 予約語などの強調表示

(2) マウスクリックによるトレースのステップごとの表示/非表示切り替え

(3) コマンドラインより指定したルール名 (正規表現) を持つルールによる反応だけを表示/非表示できる

(4) 各ステップにおいて, 反応前と反応後の diff をカラー表示 (`Algorithm::Diff` 使用)

8.5.3 オプション

`$ perl tohtml.pl [表示しないルール名の正規表現]... +[表示するルール名の正規表現]...`

表示するルール名が優先される．並びは順不同.

8.5.4 コマンドライン例

```
$ bin/lmntal_cyg -t -x dump 1 --interpret sample/hara/chr.lmn
  | perl tohtml.pl .* +introduce +solve +prop +simp > result.html
```

意味：ルール名が introduce, solve, prop, simp にマッチしたもの（の前後）は表示する．それ以外 (.*) は表示しない

第9章　まとめ今後の課題

9.1　改善点

9.1.1　uniq ガード制約と非決定的実行

uniq を含むプログラムを正しく非決定的実行するには，現在ガードにて履歴と照合・履歴に追加している UNIQ 命令を，ガードにて履歴と照合する命令 UNIQ_CHECK とボディにて履歴に追加する命令 UNIQ_ADD に分ける必要がある．

9.1.2　ガードインラインの出力変数

（正確には変数ではなく型付きプロセス文脈.）
現状では，実装方法が不明だったので出力できる構造は unary に制限されている．汎用性を考えると ground 型を出力できる方がいいかもしれない．

9.2　LMNtal に関する考察

研究生活において wiki に蓄えた小考察の集まりをここにまとめておく．

9.2.1　変数，ハンドル，名前

一般的な言語でいう変数は，名前からメモリアドレスの内容への写像.
LMNtal でも，オブジェクトの状態とかを何かの名前で特定したいことがよくある．
出力引数に名前をつけて

```
object(some, foo, myObj).
```

とかもうちょいわかりやすく

```
myObj=object(some, foo).
```

とよく書く.
名前と内容をごっちゃにしたくなかったら，マップコンテナを使えばいい.
PHP の大域変数は全て $GLOBAL というハッシュに入っていることになっている.

追記 2005/05/09(Mon)

```
myObj=object(some, foo).
```

だと，１つのインスタンスを１つの変数からしか指せない．これ大問題.
オブジェクトへの参照を何かに渡したりすることができない．名表には１対多の仕組みを使うべき.
オブジェクトの中身を膜とその中のアトムで表すとすると，ground としてコピーできない（プロセス文脈を使う必要がある）ので扱いにくい.
なので参照が必要.

9.2.2 配列

一般的な言語では，配列はメモリ空間 (整数-¿整数) に直接マップできるので自然に配列がある.
LMNtal モデルでは，整数-¿整数のメモリ空間という概念はないので配列は自然ではない.
もともとあるのはプロセスの多重集合なので，整数から何かへのハッシュみたいなのを作って，配列をシミュレートすることになるとおもう.(処理系の実装次第でメモリ空間にマップしてもいい.)
PHP でいう配列は整数-¿値に限らず，値-¿値になっている．ハッシュと配列が混ざったようなもの．LMNtal ではそっちの方が自然.

9.2.3 ポインタ，有向グラフ

LMNtal でいうリンクはこれに近い．が，双方向なので参照される側にもリンクが必要.
指している対象に影響を与えずに指す側を増減することはできない.
指される側に一意な何か（名前 or 指す側から出ているリンク先を含む膜）を割り当てて，指す側はその何かを指す必要がありそう.

9.2.4 LightWeight 分散 LMNtal

単にプロセスを直列化して送りつける.

```
{ net.sender([:IP, ポート:]) }
->
{ ここにあるプロセスを [IP, ポート] に丸ごと移動するルール }

{ net.receiver([:ポート:]) }
->
{ 受信したプロセスを「received({送信者/時刻情報}, {送られたプロセス})」
に変えるルール }
```

送る際，自由リンクは問答無用でぶった切る.（一応 a(X),b(X),net.sender は a(nil),b(nil)... にする）

このライブラリは直列化ライブラリとソケットライブラリを呼ぶだけになりそう.

9.2.5 膜の意義（の１つ）

ルールは適用できそうなら勝手に適用されてしまうので，すべてのルールが順不同で適用されても問題ないようにプログラムを書く必要がある.

もともとそういう性質をもつ問題をプログラム化するなら問題ない．そうでない場合には，ルール数が増えると人間が一度に把握できなくなる.

そういうときに複数のルールを小さなまとまりに分けてそれぞれ膜に入れておくと把握しやすくなる.

そりゃそうか.

9.2.6 プロセス構造を既存の言語のデータ構造にマップする

たとえば 2 次元グリッド構造を 2 次元配列にマップするには.

Atom.getLink とかをオーバーライドして a[x][y+1] に対応するアトムを返すようにするとか.

この例では間にアトムを挟めずに Read only になりそう.

対象に影響を与えずに指し示す構造の話と関連する.

9.2.7 膜間通信と LightWeight 分散 LMNtal

LMNtal on eyebot（矢島氏）にある組み込みの send, recv を FLI で用意して，あて先が外部ホストを表す識別子だったらソケットを使うようにすれば，マシンをまたがる膜間通信とまたがらない膜間通信とをまったく同じに書ける．

PHP の file 関数はファイルの内容を配列で返すが，ファイル名が ”http://” などで始まる場合は HTTP レスポンスのボディ配列を返す．

```
file("http://yahoo.co.jp") -> array( "<html>", "<head>", ...)
```

チャットをかく

```
// client
rd=io.readLine(line).
line=S :- String(S) | send(<SERVER>, talk(S), rd=io.readLine(line).
recv(<SERVER>, talk(S, room56, password)) :- String(S) | io.print(S).

// server
clients=set.new.
recv(Client, talk(Line, Room, Password)) :-
    unary(Client), String(Line), String(Room), Password=foobar |
    invoke(clients, put(cli(Room, Client), exist)),
    invoke(clients, msg(mapKey(X))), cp(X, N0, N1),
    if(eq(nthAtom(N0, 0), Room), send(nthAtom(N1, 1), Line)).
```

ただし，

```
// メソッド呼び出し規約 (?) の１つ
Name=Obj, invoke(Name, msg(Msg)) :- unary(Name), Name=msg(Obj, Msg).

// map モジュール
{ module(map). mapKey(X) : キーそれぞれをコピーして出力引数を X に変
える }

// 基本モジュール
if, eq, nthAtom 関連のルール（これがなかったから今までなんとなく書きづ
らかったのである）
```

を仮定している．

（仮想）逐次言語の場合

```
// client
t = new Thread{ talk = recv(<SERVER>); print talk; };
send(<SERVER>, "talk=$i:room=56:pass=foobar") for <> as i;
t.join;

// server
clients={};
while(r=recv())
  clients.put(pair(r.value.room, r.sender));
  r.value.room==c.room ? send(client, r.value):0; for clients.keys() as c;
```

ただし，recv(A) は A から受信，recv() はどこからか受信でそういうのがあるとする．

… あまり差がない．

「ここはスレッドを使うべきだな」とかスレッドの処理とかを考えなくていいのが LMNtal の利点？

9.2.8 OOP，メソッドの起動方法

左辺にオブジェクトを書く必要があると，プログラムが書きにくくなる（詳しい考察はもやもやしてるのであとで）．

書きたくなくなってくる例：

```
some(foo), {some(bar), $p}, more(X), H=object(V) :-
  H=msg(object(V), method(arg0)), some(woo), {some(hm), $p}, ah(X).
```

オブジェクトが２個３個になってくると悲劇的になる．

右辺にだけメソッドを起動する記述をすればいいという方式が多少書きやすい．

(予想)

9.2.9 短く書く

```
num(Number), str(String), ground(Ground) :- int(Number), string(String), ground(G
```

を

```
num(Number<int>), str(String<string>), ground(Ground<ground>) :- rhs.
num(Number<+>), str(String<$>), ground(Ground<g>) :- rhs.
```

とか書けるといいのう.
あと H=nthAtom(X, 2) を H=X¡2¿ と書けるようにするとか.
こういう変換は lmntal.jar を呼び出す (シェル—perl—...) スクリプトなりがやればいいので，好き勝手に決めて書いてしまえばいい.
いまのところはアイデアだけ.

9.2.10 ルール優先度があるとできそうなこと

-π計算の !p （優先度最低）
--ただしもう一工夫必要で，全ての ! が ? 個生成しても反応が起きないときは stable にするとかが必要.（? はルール左辺に依存）
-ストリーム
--入力ストリームは勝手に受信したものを生成していく.
--I/O 関連の優先度を低くすれば，必要なときだけ生成される
-ルールごとに遅延評価するしないを簡単に変えられる.
--例えば深さ優先が優先度変更のみで幅優先になったりする（か？）
--優先度をちょっと変えるだけで，アルゴリズム A が一見異なるアルゴリズム B に

変わったりすると喜ばしい.
--@see キャベツと狼と羊の問題（工藤氏）
-書きにくかった逐次が少し書きやすくなるかも

9.2.11 OCaml のマッチングとの対応

例題：偶数だけ通すフィルタ

```
--- OCaml
# let rec even_filter lst =
    match lst with
      []->[]
    | h::t when h mod 2 = 0 -> h::even_filter t
    | h::t                  ->    even_filter t ;;
val even_filter : int list -> int list = <fun>
# even_filter [1;2;3;4];;
- : int list = [2; 4]

---LMNtal
R=even_filter([]) :- R=[].
R=even_filter([H|T]) :- H mod 2==0 | R=[H|even_filter(T)].
```

```
R=even_filter([H|T]) :- H mod 2\=0 | R=even_filter(T).

res = even_filter([1,2,3,4]).
=> res([2,4]), @601

--LMNtal ガードで計算しない版
--if は並列版になっている（条件が評価される前に真節と偽節が評価されうる）
base.use. integer.use.
R=even_filter([]) :- R=[].
R=even_filter([H|T]) :- int(H),ground(T) |
  R=if('=='('mod'(H, 2), 0), [H|even_filter(T)], even_filter(T)).

res = even_filter([1,2,3,4]).
=> res([2,4]), @601
```

9.2.12 ガード中で計算できる必要はない

ガードに書けるのは+左辺に出てくるリンクの型制約+型付けされたリンク同士の演算，比較

前者は必要だが，後者は右辺に書けるので必要ない.

```
a(N) | N>0    :- a(N-1).
a(N) | int(N) :- if('>'(N, 0), a(N-1)).
```

メリットは，書いた演算や比較が 1 リダクションで実行される所.

もっとも，型制約はリンクの直後に書きたい.

```
a(N as int) :- ...
```

ので，ガードは要らない？(ただし if b te fe の te, fe が先に評価されるのをどうにかしないといけないかも)

仮のプリプロセッサ –pp0 では a(N¡i¿) と書ける.

” 勝手にコメント by mizuno”

if の評価順序については，膜の保護機能を使うのはいかが？a(N) :- int(N) — if('¿'(N, 0), a(N-1)). if を処理するルールが，then/else 節を選択する際に膜をはがす. システムルールセット（四則演算とか）は処理されてしまうので,「システムルールセットがない膜」を作った方が良いのかもしれない.(それでも unify を処理するルールは残しておくべきかも)

9.2.13 部分計算

「LMNtal 処理系は部分計算器なのではないか」という話.

-LMNtal 処理系：部分計算器 pe -LMNtal プログラム：算譜 p -部分的な環境 : (チャネル, 任意のバイト列) の集合：i –チャネルは標準入力や各ファイル，各ソケットなどを一意に識別するもの-環境 i において pe で p を実行した結果のプロセス：剰余プログラム p2 = pe p i

例 1 ラムダ計算

ラムダ計算コンパイラ lc = pe ラムダ計算インタプリタ (on LMNtal)
コンパイルされた (LMNtal) コード = lc あるラムダ式

例 2 LMNtal コンパイラ

LMNtal コンパイラ lc = pe LMNtal インタプリタ (on LMNtal)
コンパイルされた (LMNtal) コード = lc ある LMNtal コード
コンパイルされた LMNtal コンパイラ (????) = lc LMNtal インタプリタ (on LMNtal)

9.2.14 CHR モジュールを書いてて思ったこと

- LMNtal はルール間で引数を引きまわすため，複数ルールに対する引数仕様の変化に弱い．– 関数呼び出し（の気分）した際の引数は，とりあえず全部最初の引数につないだ膜とかに入れておくと後で使えるので新たに使う引数が出てきたときに複数ルールの書き直しが必要なくなる.. - 対象データがリストになってないと嫌な部分と，膜内の多重集合になってた方がいい部分がある．簡単に変換できると便利．– 1 個のデータを入れた膜をリストの要素にすればいい．list=[a, b]. —が，こういうリストは ground じゃないので扱いが悪い．こういうものにマッチする型付きプロセス文脈がほしい．–リストで持っておけば，とりあえずリストのデータ部分だけのマッチングで多重集合的な操作ができる．

9.2.15 ガードで計算しない代償

foo/1 がたくさんあって，その中である条件を満たすものだけを書き換えたいときこんなふうに書けると分かりやすい．

```
foo(N) :- condition_check(N) | bar(N).
```

これは，中間命令列に直すとこんな構造になっている．

```
foreach n in find_all foo(N)
  if condition_check(n)
    remove foo(n)
    add bar(n)
  else
    nop
```

「条件が揃わなかった場合に何も変わらない」というのが重要.
現処理系では，ごく単純な条件（比較とか）しかガードに書くことができない.
そのため，少し複雑な condition_check を通したい時は別ルールにするしかない.
たとえば，foo/1 →条件分岐→ bar/1 だったのを 1 クッションおいて foo/1 → foo'/2 →条件分岐→ bar/1 にする.

```
Foo_1 @@ foo(N) :- foo'(N, condition_check(N)).
Foo_2 @@ foo'(N, ok) | bar(N).

foreach n in find_all foo(N)
  remove foo(n)
  add foo'(n, condition_check(n))

foreach n in find_all foo'(N, ok)
  remove foo'(n, ok)
  add bar(n)
```

「条件が揃わなかった場合に，条件チェックをした残骸が残ってしまう」というのが重要.
条件チェックをした残骸を元に戻そうとして

```
Foo_1 @@ foo(N) :- foo'(N, condition_check(N)).
Foo_2 @@ foo'(N, ok) | bar(N).
Foo_3 @@ foo'(N, ng) | foo(N).
```

とすると，NG であるような N は oo_1 → Foo_3 → Foo_1... の無限ループになってしまう.
uniq ガード制約を入れて

```
Foo_1 @@ foo(N) :- uniq(N) | foo'(N, condition_check(N)).
Foo_2 @@ foo'(N, ok) | bar(N).
Foo_3 @@ foo'(N, ng) | foo(N).
```

とすると無限ループにはならないが，N が型付きでない普通のリンクの場合 uniq は使えない．

```
Foo_1 @@ foo(N) :- foo'(N, condition_check(N)).
Foo_2 @@ foo'(N, ok) | bar(N).
Foo_3 @@ foo'(N, ng) | foo_ng(N).
```

などと，１つの条件分岐をするたびに新しい名前にすればとりあえずは解決するが，時間がたつにつれて n が変化し，condition_check(n) を通るようになった時点で bar(n) に書き換えたいという動作をさせたいときは名前を変えるとまずい．

```
Foo_1 @@ foo(N) :- foo'(N, condition_check(N)).
Foo_2 @@ foo'(N, ok) | bar(N).
Foo_3 @@ foo'(N, ng) | foo_ng(N).
```

n になんらかの変化があった時点で foo'(n, ng) を foo'(n, condition_check(n)) に書き換えれば，解決ではある．

みなさん何かいい書き方ないでせうか

謝辞

本研究を進めるにあたり，御指導を頂きました上田和紀教授に厚く御礼申し上げます．

議論を通じて様々なご意見を頂きました上田研究室の方々に感謝いたします．特に加藤紀夫氏には，設計方針において多大なご指摘，ご助言を頂きました．ここに心からの感謝を捧げたいと思います．

2004年2月　原 耕司

参考文献

[1] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Principles and Practice of Constraint Programming*, pp. 252–266, 1997.

[2] J.-P. Banâtre and D. Le Métayer.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. *Foundations of Software Science and Computational Structures*, pp. 140–155, 1998.

[4] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, Vol. 37, No. 1-3, pp. 95–138, October 1998.

[5] Maria Garcia de la Banda Christian Holzbaur Gregory J. Duck, Peter J. Stuckey. The refined operational semantics of constraint handling rules. *ICLP'04 20th International Conference on Logic Programming.*

[6] 原耕司, 水野謙, 矢島伸吾, 永田貴彦, 中島求, 加藤紀夫, 上田和紀. LMNtal 処理系および他言語インタフェースの設計と実装. *SWoPP2004*, p. 157, 2004.

[7] O. H. Jensen and R Milner. Bigraps and transitions. *Proc. POPL*, pp. 38–49, 2003.

[8] Y Lafont. Interaction nets. *Proc. POPL'90, ACM*, Vol. 30, No. 2, pp. 95–108, 1987.

[9] R Milner. Bigraphical reactive systems. *Proc. CONCUR*, pp. 16–35, 2001.

[10] Tom Schrijvers and Thom Fruehwirth. Optimal union-find in constraint handling rules, 2005.

[11] F.-R Sinot. Call-by-name and call-by-value as token-passing interaction nets. *Proc. TLCA 2005*, pp. 386–400, 2005.

[12] K Ueda. Concurrent logic/constraint programming: The next 10 years. *The Logic Programming Paradigm: A 25-Year Perspective*, pp. 53–71, 1999.

[13] K Ueda. Resource-passing concurrent programming. *Proc. TACS 2001*, pp. 95–126, 2001.

[14] K. Ueda and Kato. Programming with logical links: Design of the lmntal language. *Proc. Third Asian Workshop on Programming Languages and Systems (APLAS 2002)*, pp. 115–126, 2002.

[15] 上田和紀, 加藤紀夫. 言語モデルLMNtal. コンピュータソフトウェア, Vol. 21, No. 2, pp. 44–60, 2004.

[16] 矢島伸吾, 永田貴彦, 加藤紀夫, 上田和紀. LMNtal プロトタイプ処理系の設計と実装. 日本ソフトウェア科学会第 20 回記念大会論文集, pp. 21–25, 2003.

[17] 工藤晋太郎, 加藤紀夫, 上田和紀. Lmntal 処理系におけるグラフ構造の操作機能の設計と実装. 情報科学技術レターズ, pp. 81–94, 2005.

[18] 乾敦行, 原耕司, 水野謙, 上田和紀. 階層グラフ書き換え言語 lmntal 処理系とその応用例. *PPL2006*, 2006.

Appendix.A ソースコード

A.1 LMNtalCHR

```
/*
---------------------------------------------------------------------
CHR
                                                    Koji Hara
                                                    2005/10/18(火) 23:03:38
                                                    2005/10/24(月) 14:26:43

操作的意味論をそのまま LMNtal で書いてみたもの

関連：
 http://www.ueda.info.waseda.ac.jp/localpage/siryou2005/embeddingCHRinLMNtal.ppt
 Operational Semantics and Confluence of Constraint Propagation Rules[Abd97]
---------------------------------------------------------------------




State : <Gs, Cu, cb, T, V>
  Gs : ゴールストア : Set of c=制約 | eq/2
  Cu : ユーザ定義制約ストア : Set of c=制約
  Cb : 組み込み制約ストア
  T  : トークン : Set of ルール名 | c=反応できるユーザ定義制約
  V  : Gs, Cu, Cb に出てくる変数 : 未使用

  実装:
    state : states(Gs, Cu, Cb, T, V).
    Gs={c=制約, eq(変数, 変数), ...}.
    Cu={c=制約, ...}.
    Cb={eq(変数, 変数), ...}.
    T ={ {rule=ルール名, c=制約, ...}, ... }.
    V ={}.
    token :

Initial :
  <Gx, T, true, 0, V>

Final :
  <T, Cu, Cb, T, V> with false not \in Cb
  <Gs, Cu, false, T, V>
```

組み込み制約は、今のところ eq のみ

*/

/*

HISTORY / 備忘録

コマンドライン例
```
$ java -cp bin runtime/FrontEnd -x dump 1 --interpret --use-source-library
  sample/hara/chr.lmn -t | perl tohtml.pl .* +prop > result.html
```

★マークがついたルールは CHR プログラムに依存（要手動コンパイル）

◆等式伝播

Cb に eq(a, b) があって Cu に leq(a, b) などがある場合、a=b だから leq(a, a) などと伝播させなければならない。
意味論では正規化関数でやっている。nstate を追加。

◆トークンの計算方法

token/2 は ground と膜を受け取ってトークンを計算し、同レベルに展開する。

leq の例だと偶然うまくいっていたが、Token の計算が不完全だったので修正した。
T(C, Cu) は {C, Cu} の中から Propagation rule の左辺にマッチするものを全て出力する。

実装は {C, Cu} 内にルールを置いて、マッチしたものを別の膜に入れて出力としている。
これだと {C, Cu} 内で反応したものは消えてしまうので、本来計算されるべきものが計算されていない可能性がある。

例：
```
r3 @ leq(X, Y), leq(Y, Z) ==> leq(X, Z).
{C, Cu} = leq(a, b), leq(b, c), leq(c, d)
```

の時

```
T(C, Cu) = { r3@{X=a, Y=b, Z=c}, r3@{X=b, Y=c, Z=d} }
```

となるべきだが 1 個だけしか返されていなかった。

解決案
ある Propagation rule の左辺の要素数（アトムグループ数）が k 個の時、
Cu から選ばれるのは k-1 個。（C は新しく追加されたものなのでマッチするなら必ず含まれる）

count(Cu) 個から k-1 個を選ぶ全通りについて左辺とマッチするか調べればよい。

気づいたこと：

- LMNtal はルール間で引数を引きまわすため、複数ルールに対する引数仕様の変化に弱い。
-- 関数呼び出し（の気分）した際の引数は、とりあえず全部最初の引数につないだ膜とかに入れておくと後で使えるので新たに使う引数が出てきたときに複数ルールの書き直しが必要なくなる。。
- 対象データがリストになってないと嫌な部分と、膜内の多重集合になってた方がいい部分がある。簡単に変換できると便利。
--１個のデータを入れた膜をリストの要素にすればいい。list=[{a}, {b}].
---が、こういうリストは ground じゃないので扱いが悪い。こういうものにマッチする型付きプロセス文脈がほしい。

*/

```
{ module(lmntalchr).


Solve @@
state({eq(X, Y), $gs}, {$cu[], @cu}, {eq=[], $cb}, {$t, @t}, {$v})
:- unary(X), unary(Y) |
nstate({$gs}, {$cu[], @cu}, {eq=[X,Y], eq(X, Y), $cb}, {$t, @t}, {$v}).


Introduce @@
state({c_=Con, $gs}, {$cu[], @cu}, {$cb}, {$t, @tr}, {$v})
:- ground(Con) |
tstate({$gs}, {c_=Con, $cu[], @cu}, {$cb}, {$t, token({c_store. c_=Con},
 {$cu[]}), @tr}, {$v}).




//-----------------------------------------------------------------------
// Normalize - '=' の伝播
//-----------------------------------------------------------------------

Eq @@
nstate({$gs}, {$cu[], @cu}, {eq=[], $cb}, {$t, @tr}, {$v})
:-
state({$gs}, {$cu[], @cu}, {eq=[], $cb}, {$t, @tr}, {$v}).


Eq @@
nstate({$gs}, {$cu[], @cu}, {eq=[X], $cb}, {$t, @tr}, {$v})
:- unary(X) |
state({$gs}, {$cu[], @cu}, {eq=[], $cb}, {$t, @tr}, {$v}).


Eq @@
nstate({$gs}, {$cu[], @cu}, {eq=[A,B|R], $cb[R]}, {$t, @tr}, {$v})
:- unary(A), unary(B) |
nstate1({$gs}, {replace(B, A), $cu[], @cu}, {eq=[A|R], $cb[R]}, {$t, @tr}, {$v}).
```

```
Eq @@
nstate1({$gs}, {replace(B, A), $cu[], @cu}/, {$cb}, {$t, @tr}, {$v})
:- unary(A), unary(B) |
nstate({$gs}, {$cu[], @cu}, {$cb}, {$t, @tr}, {$v}).


Terminate @@
H={ state({}, {$cu[], @cu}, {eq=[], $cb}, {$t[], @tr}, {$v[]}), $etc[], @etc }/
:-
H={$cu[], $cb}.

Start @@
H=lmntalchr.run({$gs}, {@simplify}, {@replace}, {@find_token}, Prop_head_count)
 :- int(Prop_head_count) |
H={
  lmntalchr.use.
  solve.
  introduce.
  simplify.
  propagate.
  @simplify.
  state(Gs, Cu, Cb, T, V).
  Gs={$gs}.
  Cu={
    @replace.
  }.
  Cb={eq=[]}.
  T={
    found_token.
    prop_head_count=Prop_head_count.

    // token/2 - トークンを計算する
    token({c_store. $c[]}, {$cu[]}) :- token0({{c_store. $c[]}, {$cu[]}}, {list=[], $cu[]}).

    // リスト処理するためにリスト化
    token0(ARG, {list=L, c_=Con, $c[L]}) :- ground(Con) |
    token0(ARG, {list=[Con|L], $c[L]}).

    token0(ARG, {list=L}), prop_head_count=N :- ground(L), int(N) |
    token1(ARG, {list=list.choose_k(L, N-1)}, {}), prop_head_count=N.

    // 候補を作る
    token1({{c_store. c_=Con}, {$cu[]}}, {list=[Hd|Tl], $p[Tl], @r}, {$w})
     :- ground(Hd), ground(Con) |
    token1({{c_store. c_=Con}, {$cu[]}}, {list=Tl, $p[Tl], @r}, {$w, {l=Hd, l=Con}}).

    token1(ARG, {list=[], @r}, {$w}) :- |
    token2(ARG, {$w}).

    @find_token.
```

```
  }.
  V={}.
  // トークン計算が終わったら次
  TokenFinished @@
  tstate(Gs, Cu, Cb, {$p, token2({{$c[]}, {$cu[]}}, {$w[]}), @r}/, V) :-
  nstate(Gs, Cu, Cb, {$p, @r}, V).
}.

}. // End of CHR module
```