2004年度 卒業論文

LMNtal処理系における
グラフ構造の複製・破棄
および比較機能の設計と実装

提出日： 2005年2月2日
指導：上田 和紀 教授

早稲田大学理工学部情報学科
学籍番号： 1G01P037-6

工藤 晋太郎

概 要

上田研究室により提案された言語モデル LMNtal は, 階層的グラフ構造の書き換えに基づく言語モデルである. その基本要素はアトムとそれを結ぶリンク, それらを階層構造の元に扱う為の膜, そしてそれらグラフ構造の書き換え規則であるルールである.

LMNtal では, プロセス構造を容易に図で表現できるように設計されており, また, 複雑なデータ構造を平易に扱えることを一つの目標にしている. 複雑で規模を持ったデータ構造の表現については, 膜やアトムとリンクを用いて直感的で簡便なモデル化が可能である. しかしその操作においては, 不特定構造の複製や破棄, 比較といった機能が欠けていた.

例えば探索問題において, バックトラックではなく, 探索木の分岐に応じて状態を複製していくという解き方がある. これを LMNtal で書くには上記の機能が必要である.

本研究ではこれを補うものとして, まず “非線形プロセス文脈” という概念を実装した. これは直感的には膜を複製・破棄する機能であり, これを従来の処理系に組み込んだ. さらに, 不特定構造の比較を目的とした機能として, “基底項プロセス” という概念がある. この概念を, 使用目的や実装のし易さを考慮して設計し, 従来の処理系に組み込んだ.

これによって, 従来の LMNtal ではできなかった計算モデルの表現ができるようになった. そして, これら二つの機能を用いた LMNtal プログラム例として, 本研究の動機でもある, 状態の複製を用いて探索問題を解くプログラムを記述した. そのモデル化方法と LMNtal プログラムでのアルゴリズムの表現を解説し, また, この方法の応用を考察する. また, さらに柔軟なモデル化を可能にするため, あるいはより容易な記述を可能にするための機能について考察した.

目 次

第1章　序論

1.1　研究の目的と意義

探索問題を解くことを考えた時, 解かれる過程をそのまま表現するような解き方が考えられる. つまり, 状態を表すデータの塊が存在し, 複数ある「次の状態」を, そのデータの塊の一部異なる複製で表現するような方法である.

これは解かれる過程のモデル化としても興味深いし, また, 生物の進化のような分化しつつ進行する系の模倣にも適用しうる.

この観点で, グラフィカルな表現から直結するようなプログラムを設計するのにうってつけの言語モデル LMNtal を用いることは, 非常に的を射た選択であるが, 従来の LMNtal 処理系では機能的に不十分であった.

LMNtal では, この「データの塊」は, 例えば膜を用いて表現することができる. 膜の一部の要素のみを変更するような操作も可能である. この時, この一部の要素を除いた部分を, LMNtal では「プロセス文脈」という概念で扱う. しかし従来の LMNtal 処理系の機能では, プロセス文脈は線形でなければならなかった. 膜の一部を変更したり, 新たな要素を追加したり或いは特定の要素を消去したりすることはできる. だが残りの部分についてはそのままにするしか無かった. 膜について加えられる操作は, 全て特定されていることが条件だったのである. つまり, 膜の内容物を特定せずに複製したり破棄したりといった操作ができなかった.

先に挙げたようなモデル化の可能性を考えた時, 非線形プロセス文脈の必要性は明らかである.

また, 従来の LMNtal 処理系において, 比較, つまり等しいかどうか検査できるのは, 単一のアトムのみであった. アトムは LMNtal における計算の最小単位である. 比較できる構造は, 極めて限定されていたということである. 「どういう構造かは不明だが, 両者は等しい」という表現ができなかったのである.

データの塊を比較をしたいという欲求は当然ある. 例えば木構造やリストの部分構造について等しいかどうかの検査ができると, モデル操作の可能性はぐっと広がる.

本論文では, このような欲求に答える機能として「非線形プロセス文脈」「基底項プロセス」という二つの概念を紹介し, また, これらに関するいくつかの操作機能の設計, 実装を解説する.

1.2 本論文の構成

本論文は以降, 次に示すような流れにそって展開する.

- **第 2 章**

  言語モデル LMNtal を解説する. LMNtal の基本要素である, アトム, リンク, 膜, ルールといった概念を図とソース上の表現を交えて解説する. さらに, 本論文で取り上げる非線形プロセス文脈の解説の前提として, プロセス文脈を解説し, 基底項プロセスの解説の前提としてガードや型付きプロセス文脈といった概念を解説する.

- **第 3 章**

  「非線形プロセス文脈」を紹介し, その機能の設計と実装を解説し, 使い方を説明する.

- **第 4 章**

  「基底項プロセス」を紹介し, その概念の設計と, 機能の設計と実装を解説する. また, 簡単な使い方を述べる.

- **第 5 章**

  第 3 章と第 4 章で紹介した機能を実際に使って探索問題を解くプログラムをとりあげる.

- **第 6 章**

  今回の論文で実装した諸機能の発展の可能性を「今後の課題」として述べる.

- **付録**

  第 5 章で紹介した LMNtal プログラムのソースコードと, 諸機能の実装部分の中間命令の説明を, 62 ページ以降に付録として載せる.

第2章　言語モデルLMNtalの解説

ここでは, 言語モデル LMNtal について解説する. この章の内容は, 文献 [6] を参考に再構成したものである.

$P ::=$	(プロセス)
0	(空)
$\mid\ p(X1, ..., Xm)$	(アトム)
$\mid\ P, P$	(分子)
$\mid\ \{P\}$	(膜)
$\mid\ (T :- G \mid T)$	(ルール)
$T ::=$	(プロセステンプレート)
0	(空)
$\mid\ p(X1, ..., Xm)$	(アトム)
$\mid\ T, T$	(分子)
$\mid\ \{T\}$	(膜)
$\mid\ (T :- G \mid T)$	(ルール)
$\mid\ @p$	(ルール文脈)
$\mid\ \$p[X1, X2, ..., Xm\|A]$	(プロセス文脈)
$\mid\ p(*X1, X2, ..., Xm)$	(アトム集団)
$A ::=$	
$[]$	(空)
$\mid *X$	(リンク束)
$G :: -$	(ガード)
$\mid 0$	(空)
$\mid G, G$	
$\mid$ ガード型制約	

図 2.1: 構文

LMNtal の主要な構成要素は, **アトム**, **リンク**, **膜**, そして**ルール**である. これらを順を追って紹介すると同時に, LMNtal のプログラムソース上での表記とそのグラフィカルな表現を図で交える. LMNtal は, グラフィカルな表現を容易にすることも意図して設計されている.

2.1 アトムとリンク

LMNtal の基本的な要素として, まず**アトム**がある.

アトムは英小文字か数字で始まる**名前**と, 0 個以上の**引数**を持っており, LMNtal における計算の最小単位である. 引数には順序があり, そのアトムの第 1 引数, 第 2 引数... というように区別する. 引数は腕のようなものと考えてよく, 他のアトムの引数と繋がりを持つことができる. この繋がりを表現するのが**リンク**である. リンクは無方向的であり, 1 対 1 である. リンクは, ルール外では必ず 2 回出現する. つまり, どこにも繋がっていないリンクを持つアトムは無い. ソース上では, リンクを英大文字から始まる文字列で表現する. ただし, リンクは名前を持たず, 他の文字列で置き換えてもかまわない. また, アトムはその引数の数が違えば同名でも「異なる」. 同名, 同引数のアトム (すなわち「等しい」アトム) でも複数存在していても問題ない.

図 2.2 は, アトムをグラフィカルに表現したものである.

図 2.2: 0 引数アトムのグラフィカルな表現

また, ソース上では, 次のように表す.

```
a.
```

これは 0 引数の, a という名前を持つアトムである. ピリオドは, 構文の終わりを意味する.

今度は, 引数を持ったアトムである.

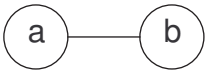

図 2.3: 1 引数アトムのグラフィカルな表現

```
a(A), b(A).
```

これは 1 引数の a という名前のアトムと, 同じく 1 引数の b というアトムが存在し, a の第 1 引数と b の第 1 引数が同じリンクを持っていることを示している. 引

数は, アトム名の後に括弧を用いて表し, カンマで区切ってリンクを並べる. 先の例では, () が省略されていた. また, 最初に述べたように, リンクを表す文字列は他の文字列で置き換えて構わない. つまり次のように書いても同じことを表している. リンクは名前を持たない.

```
a(X), b(X).
```

また, 複数のアトムを表す時は次のようにカンマで区切って並べる. ピリオドで区切って複数の構文としてもよい. 次の 3 行は同じ意味である.

```
a.  b.  c,d.
a,b,c.  d.
a,b,c,d.
```

図 2.4 は, 複数のリンクを持つアトムのグラフィカルな表現である.

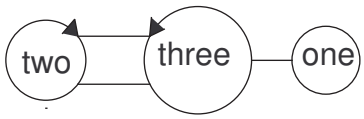

図 2.4: 複数のリンクを持つアトムのグラフィカルな表現

```
three(A,B,C), two(A,B), one(C).
```

これは, three という名の 3 引数のアトムと, two という名の 2 引数のアトムと, one という名の 1 引数のアトムが存在し, それぞれの引数に持つリンクが表されている. このように複数の引数を持つアトムを図で表現する際には, 第 1 引数に矢印をつけ, その向きに第 2, 第 3... と表すことにする. また, 例のように, 2 本以上のリンクが同じアトムのペアを取り持つこともある. アトムの表記順序やリンクの名前は意味を持たないので, 次のように書いても同じである.

```
one(A), two(B,C), three(B,C,A).
```

だが, 次のように書くことはできない. リンクの出現は必ず 2 回である.

```
one(X),two(X,Y),three(X,Y,X).
```

また, アトムとリンクの記述には略記法が存在する. リンクの記述を省略し, 直接リンク先のアトムを埋め込んで記述することができる. この時, 埋め込まれたアトムの最後の引数 (N 引数のアトムならば, N 番目の引数) が, 埋め込んだアトムに繋がっていることを表す. 例を挙げると, 次の 3 行は同じ意味である.

```
atomA(atomB).
atomA(X), atomB(X).
atomB(atomA).
```

次の 3 行も同じ意味であり, 図 2.5 はそのグラフィカルな表現である.

```
alpha(beta(gamma,delta)).
beta(gamma,delta,alpha).
alpha(A),beta(G,D,A),gamma(G),delta(D).
```

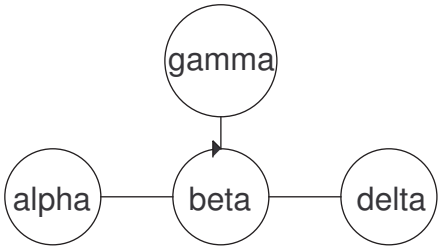

図 2.5: 分子のグラフィカルな表現

このように, 複数のアトムがリンクで繋がったまとまりを**分子**と表現することもある.

また, リンクの別の表現方法として=を用いることもできる. =は中置記法であるが, その前後の引数にアトムを表記した場合, そのアトムの最後の引数に持つリンクの意味になる. 厳密には, =はリンクの表現ではなく 2 引数の'=' という名のアトムである. このアトムは処理系によって直ちに処理され, そこに繋がる両リンクを融合させる. その意味で, 次の 3 行は同じ構造である.

```
X=a(Y), a(Y)=X.
a(A,B), a(A,B).
a(A,a(A)).
```

次に挙げる 2 行は等しい構造だが, 似てはいても上の 3 行とは意味が異なる.

```
a(a(A),A).
a(A,B), a(B,A).
```

すなわち, 前者における 2 つのアトム a は第 1 引数同士, 第 2 引数同士が繋がっているが, 後者における 2 つのアトム a は, 片方の第 1 引数はもう片方の第 2 引数に

繋がっている. つまり, 異なる構造を表している. 図 2.6 の左が前者を, 右が後者を示している.

図 2.6: 引数位置の違い

なお繰り返しになるが, カンマで区切られたアトムの記述順序は意味を持たない. しかし引数の順序は意味を持つ. 従って, 略記法を用いる場合は, 引数の順序に気を配る必要がある.

2.2 リスト

LMNtal では, リストもアトムとリンクで構成される構造であるが, 特殊な記法を用いることができる. 例を挙げよう.

```
numbers=[1,2,3].
atoms=[a(a),b(b(b)),c,d(d)].
lists=[[p],[q],[[r]]].
```

それぞれの構造を, 図 2.7 に示す.

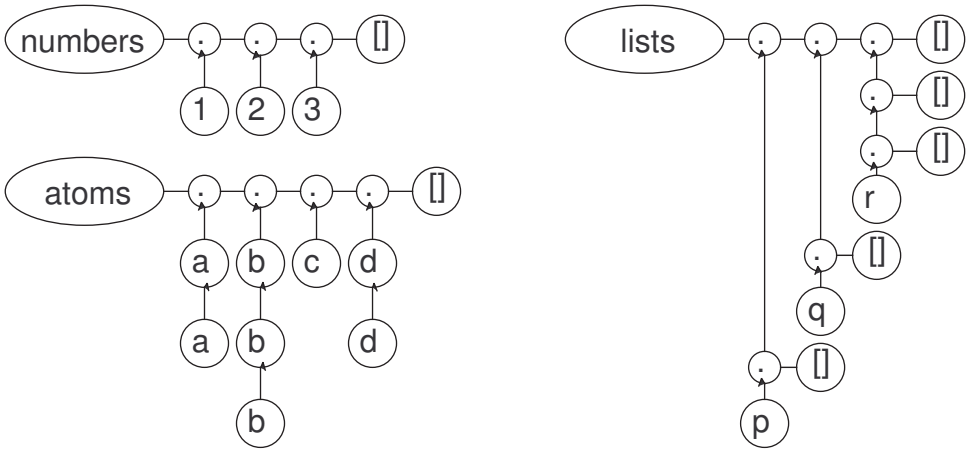

図 2.7: リストのグラフィカルな表現

図を見ればわかるように, '.' という名前の 3 引数のアトムの繋がりで, リストが作られる. 第 1 引数はリストの要素側へ, 第 2 引数は後続リストへ, 第 3 引数はそのリストへの参照へ, それぞれリンクを持っている. また, リストの「残りの部分」を | で区切ってリンクで表記できる. 下記に示す.

```
LIST=[CAR|CDR].
```

これは, リストの一部のみを表しているわけだが, 後述するルールの中ではこのよ

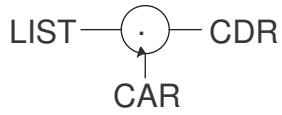

図 2.8: '.' アトム

うな書き方がとても役に立つ. このように, LMNtal ではアトムの最後の引数を参照として利用したり, 出力引数としたりするのが一般的である. リストの最後は空のリストであることを示す [] という名前の 1 引数アトムに繋がっている. 大括弧によるリストの記法は, これらの構造の略記法である. 略記せずに記した例が以下である.

```
list=[a,b,c] ↔'.'(a,A,list), '.'(b,B,A), '.'(c,C,B), '[]'(C).
```

記号'.' は構文の終端にも用いるので, シングルクォートで囲んで, 名前であることを示している.

2.3 膜

続いて, LMNtal の基本要素の 1 つ, **膜**を紹介する. 膜は, 階層構造を表現する重要な要素である.

膜は, ソース上では{と}で囲んで表現する. 膜は, その中にアトムや膜, 後述するルールを含むことができる (これを, その膜に「所属する」と表現する).

膜とそこに所属するアトムや膜の表記は以下のようになる. 膜の中では, カンマで各要素を区切って表現するか, ピリオドで終端した文で要素を表記する. ただし基本的にはカンマで区切って記述する.

```
{a(b), c, d}.
{a(X). b(X). c, d.  }.
```

この 2 行は同じ構造を表しており, グラフィカルな表現が図 2.9 である. 図では膜

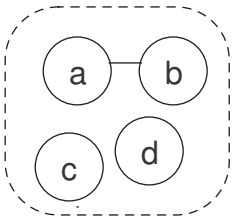

図 2.9: 膜のグラフィカルな表現

を点線で示すことにする. 内側に四つのアトムが表されていて, これらはこの膜に「所属する」アトムである. リンクはアトムが持つものであり, 膜に所属はしない.

次は, 子膜を含む例である.

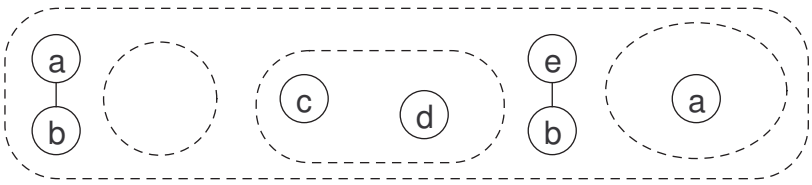

図 2.10: 膜と子膜のグラフィカルな表現

```
{a(b), b(B), {},{c, d}, e(B), {a}}.
```

このように, 階層構造をとることもできる. その場合, その膜に所属する膜を**子膜**と言い, 逆を所属膜, または**親膜**と言う. ただし子膜の子膜は, その膜の子膜ではない. あくまでその膜に所属するのが子膜である.

また, 膜の中の要素は順序を持たない.

また次のように, 膜をまたがるリンクもある.

```
{a(A), {b(A)}}.
{s(A,B,C), {r(A), {r(B), {r(C)}}}}.
a(A), {b(A,B), {c(B,C)}, d(C,D)}, e(D).
```

膜から外に出てゆくリンクを, 膜の**自由リンク**と言う. 逆に膜の中に両端があるものは, 膜の**局所リンク**という.

さらに, 膜をアトムの引数に書く略記法がある. それは, 次のように, '+' という 1 引数のアトムが省略されたと解釈される.

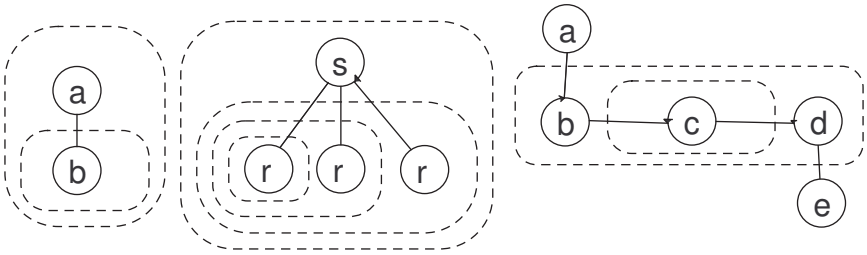

図 2.11: 自由リンクを持つ膜のグラフィカルな表現

```
atom({}) ↔ atom(M),M={} ↔ atom(PLUS),{'+'(PLUS)}
```

例えば, 次のように書ける. 図 2.12 はこれらのグラフィカルな表現である.

```
mems({},{a,b},{c}).
nums(M), M={1,2,3,4,5}.
```

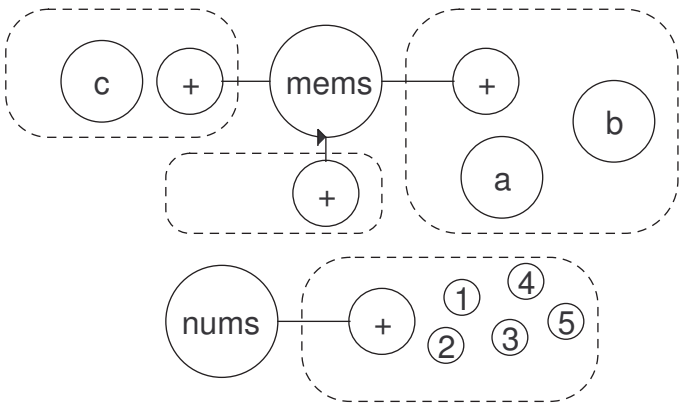

図 2.12: アトムの引数に膜を記述した場合のグラフィカルな表現

2.4 プロセス

今まで挙げたような, アトムやリンクや膜などの構造と次に述べるルール, あるいはその一部をさして, **プロセス**[1]と言う. プロセスは LMNtal における計算の主

[1]構文は. 図 2.1 を参照.

人公である.

またプロセスについても, 中で 2 回出現するリンクを局所リンク, 1 回出現するリンクを自由リンクと言う.

2.5 ルール

LMNtal における計算は, ルールによって行われる. ルールとはプロセスの書き換え規則のことである. ルールは左辺, ガード, 右辺という 3 つの部分から構成される. ガードは省略することもできる. ガードについては 2.8 節で述べるので, ひとまず省略して考える. 一般的なルールの表記は以下のようになっている.

```
左辺 :-（ガード | ）右辺
```

左辺と右辺は, 書き換え前と書き換え後のプロセスをそれぞれ表している. この部分は, これまで出てきたような静的な表現とよく似た記述をするが, 少し異なり, プロセステンプレートと言う. 例を交えて解説していこう.

また, ルールの左辺・右辺では, 各要素をカンマで区切る. カンマは, :- より強い結合である.

まずは単純な例を挙げよう.

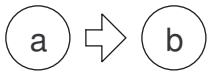

図 2.13: ルール (a :- b.)

```
a :- b.
```

このルールは, a という名前の 0 引数のアトムを, b という名前の 0 引数のアトムに置き換える規則を意味する. 該当するアトムがあればそのアトムに反応して, このルールは適用される. 左辺に反応したプロセス（この場合, アトム a）は消え, 右辺に表されるプロセスが作られる. つまりこの例では, アトム a は消費され, アトム b が作られる.

反応するプロセスがある限り, ルールは適用される. この場合, アトム a の数だけ複数回適用され, その数だけアトム b が作られることになる.

反応しない部分は, そのままである.

適用例を図 2.14 に示す. 3 つあったアトム a がそれぞれ反応し, ルールが 3 回適用された.

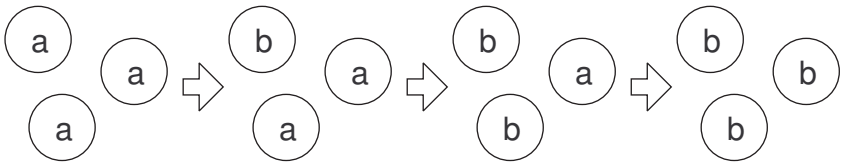

図 2.14: ルール (a :- b.) の適用例

```
a,a,a ⇒ b,a,a ⇒ b,b,a ⇒ b,b,b
```

今度は 1 引数の a に反応して, 同じく 1 引数の b に書き換えるルールである.

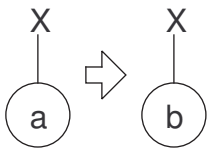

図 2.15: ルール (a(X) :- b(X).)

```
a(X) :- b(X).
```

X で表されるリンクは左辺に 1 回, 右辺に 1 回出現している. これはリンクが受け継がれることを意味する. つまり, a の第 1 引数に繋がっていたリンクは, ルール適用後は b の第 1 引数に移動する. また, 引数の数が違えば同名でも「異なる」ので, 0 引数の a には反応しない. 適用例を次に示す.

```
a(n) ⇒ b(n) a(a) ⇒ b(a) ⇒ b(b)
a(a(a)) ⇒ b(a(a)) ⇒ b(a(b))
```

略記法を用いて, 次のように書くこともできる.

```
X=a :- X=b.
```

リンクは, ルール内では必ず 2 回出現する.

ただし, 後述する型付きプロセス文脈はリンクと同じように記述されるがリンクではないので以上の制限はない.

```
a(X), b(X) :- c(Y), d(Y).
```

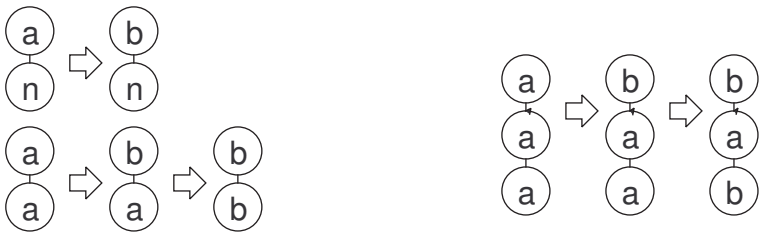

図 2.16: ルール (a(X) :- b(X).) の適用例

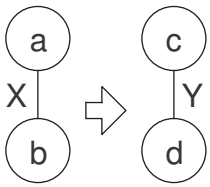

図 2.17: ルール (a(X), b(X) :- c(Y), d(Y).)

今度はリンク X は, 左辺で 2 回出てきている. これは, リンクで第 1 引数同士が繋がった 2 つのアトム a, b に反応する. 言い方を変えれば, プロセス a(b) に反応する. また, 右辺で 2 回出現しているリンク Y は, 新しく作られたアトム c と d を結ぶ意である. 言い換えれば, このルールの適用後にはプロセス c(d) が出現する.

このように, 左辺や右辺にはリンクで繋がった複数のアトム, つまり**分子**を指定することもできる.

次は膜を含むルールである.

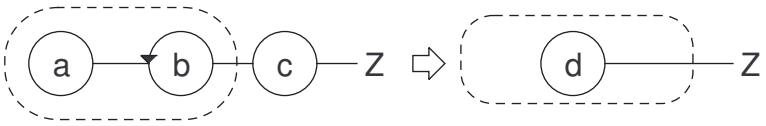

図 2.18: 膜を含むルール

```
{a(X), b(X,Y)}, c(Y,Z) :- {d(Z)}.
```

ルール中の左辺, または右辺の最も外側の膜は, そのルールが存在する膜の子膜を意味する.

このルールは, a と b の 2 つのアトム**だけ**を含む子膜に反応する. 何故なら, ルー

ル左辺の子膜にはそれしか書かれていない. その他がある場合には次節で説明するプロセス文脈を使用する. また, 左辺と右辺に 1 回ずつ出てくるリンクは継承されるので, 膜の外のアトム c に繋がっていたリンクは, 新しく出現した膜に所属するアトム d に繋がる.

2.6 プロセス文脈

プロセス文脈はルールの中で使用されるもので, 下のように表記される.

```
$p[]
$p[X,Y]
$p[X,Y|*Z]
```

使い方を見てみよう.

```
{a, $p[]}  :- {b, $p[]}.
```

それぞれのプロセス文脈は, ルール左辺において, いずれかの膜の中に 1 回だけ

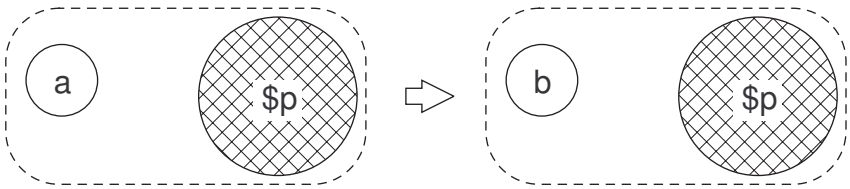

図 2.19: プロセス文脈を含むルール

出現する. これは, 直感的には,「膜内のその他全てのプロセス」を意味している. このルールでは, 左辺の膜の a という名前の 0 引数のアトム以外の部分全てを, $p という名前のプロセス文脈で表している. それに続く [] は, $p から出て行くリンク ($p の自由リンク) が無いことを示している. このルールは, 例えば次のような反応と結果をもたらす.

```
{a,b,c}. ⇒ {b,b,c}
{a,a(a)}. ⇒ {b,a(a)}
{a,a,a}. ⇒ {b,a,a} ⇒ {b,b,a} ⇒ {b,b,b}
{a,{a},b(c)} ⇒ {b,{a},b(c)}
{a} ⇒ {b}
```

最後の例のように, プロセス文脈は空の部分にも反応する.

次のような膜には反応しないことに注意されたい.

```
{a, b(X)}, c(X).
```

なぜならば, X はこの膜の自由リンクであり, $p の自由リンクになる. したがって

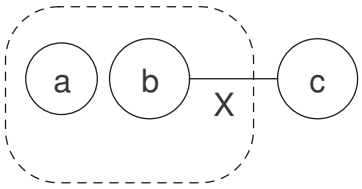

図 2.20: 反応しない膜

上のルールには反応しない.

次の例は, $p が引数を持つ場合である.

```
{a(X), $p[X]} :- $p[Y], b(Y).
```

今度のルールでは, a というアトムの第 1 引数がプロセス文脈$p の第 1 引数に伸

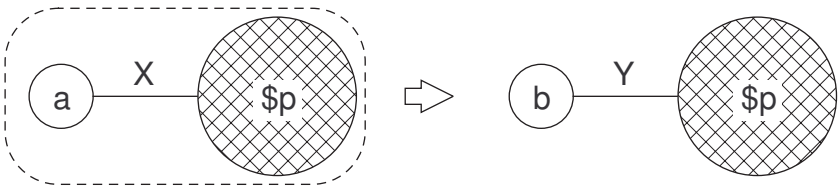

図 2.21: 引数つきプロセス文脈を含むルール

びている. すなわち, 膜の中のいずれかのプロセスに X の先が繋がっていることを意味する. 右辺ではこの部分が Y になっているが, 引数としての位置は左辺の X の位置, つまり$p の第 1 引数である. つまり, リンク X は, リンク Y になる. 注意すべきは, この X は$p の自由リンクではあるが膜の自由リンクではない (局所リンクである) ということである. なぜなら, $p の引数に出現するリンク X の反対側は同じ膜内のアトムの引数に出現しているからである.

左辺においてはプロセス文脈は必ず子孫の膜 (「子孫の膜」とは, 子膜または子膜の子孫の膜のことである) に出現しなければならないが, 右辺においては$p が出現する位置に制限は無い. 今回のように子膜の中でなくとも構わない. つまり, この例のように「膜を剥がすルール」も書けるということである.

このルールは以下のような反応と結果をもたらす.

```
{a(a)} ⇒ a(b)
{a(X), {a(X)}} ⇒ {a(X)}, b(X)
{a(b), {a(a)}} ⇒ b(b), {a(a)} ⇒ b(b), a(b)
```

次は膜の自由リンクを含む例である.

```
{a(X,Y), $p[X,Y,Z]},b(Z) :- {$p[A,B,C], c(A,B,C)}.
```

今度のルールでは, $p の自由リンクは 3 つあり, そのうち 2 つは同じ膜内のアト

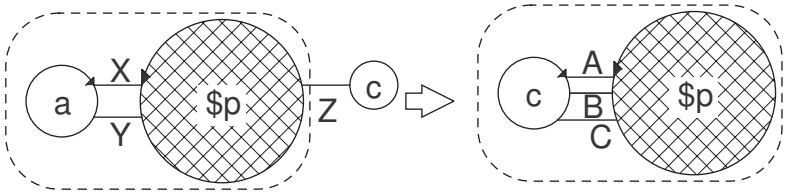

図 2.22: 膜の自由リンクを含むルール

ムに, 1 つは膜の外のアトム b に繋がっている. プロセス文脈の場合もアトムと同じように引数と言い, 順序がある. 右辺で出現する A は左辺の X を, B は左辺の Y を, C は左辺の Z を継承している. 以下に, 反応例を示す.

```
{a(c,d(X))},b(X) ⇒ {c(c,d(X),X)}
{a(a,a),{a(X)}},b(X) ⇒ {c(a,a,X),{a(X)}}
```

また, プロセス文脈は 1 つのルール内で複数種類出現してもよい. ただし, 左辺の膜 1 つにつき 1 種類であり, 左辺での出現は 1 種類につき 1 回でなければならない.

```
{$a[], {$b[], {$c[]}}} :- $a[], $b[], $c[].
```

このルールは, 3 階層以上の膜を見つけて, 外側から 3 つの膜を外している.

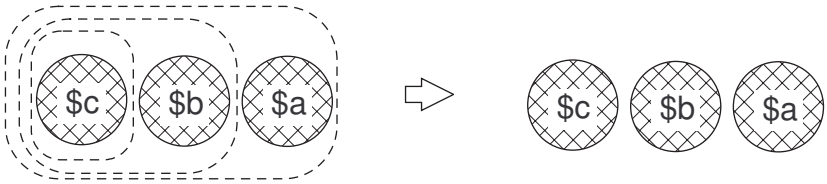

図 2.23: プロセス文脈を複数含むルール

```
{a, {b, {c, {d}}}} ⇒ a, b, c,{d}
{a, {}, {b, c}, {b, {c,{}}}} ⇒a, {}, {b, c}, b, c, {}
```

ただし, このルールでは 3 階層以上無ければ反応しないことと, それぞれの階層の 1 つずつにしか反応しないことに注意する. 左辺に表現されたプロセスを見つけたら, それを右辺に表現するプロセスに置き換えるのがルールである.

2.6.1 明示的な自由リンクと明示的でない自由リンク

これまで紹介したプロセス文脈の使い方では, プロセス文脈が自由リンクを持つ場合, それを全てルール内で明示しなければならなかったことにお気付きだろうか.

自由リンクの本数に関係なく反応すべきルールの場合など明示する必要が無い場合や, 明示することができない場合には, 自由リンクを明示せずに書くことができる.

```
{a(X),$p[X|*Z]} :- {b(Y),$p[Y|*Z]}.
```

このルールは, 先に挙げたルールの中に似たものがあったが, 今度は次のような膜

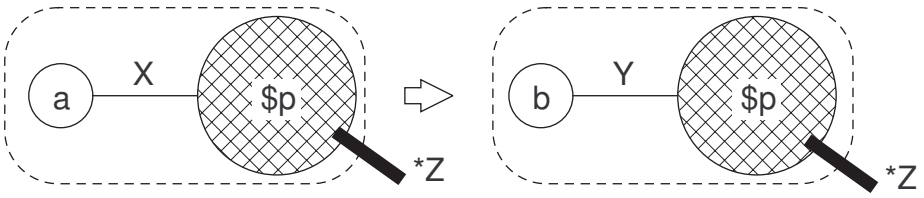

図 2.24: 明示的でない自由リンクを持つプロセス文脈を含むルール

にも反応する.

```
{a(b), c(X)}, d(X) ⇒ {b(b), c(Y)}, d(Y)
```

このルールの中のプロセス文脈$p は, 引数部分が [X|*Z] となっている. この*Z の部分は, ルールに登場しない$p の自由リンク残り全てを意味している. 無論, 0 本でも構わない. "*"は**リンク束** (不特定多数本のリンク) であることを示す記号である.

繋がっている先が明示されていないのでこの*Z の部分を**明示的でない自由リンク**と言い, 反対に X の部分を明示的な自由リンクと言う. 明示的な自由リンクは無くても構わない.

2.6.2 線形プロセス文脈

これまでの例に挙げたように, 従来の LMNtal 処理系ではプロセス文脈は左辺に 1 回, 右辺に 1 回しか出てきてはいけなかった. つまり継承の意味でしか用いることはできなかったわけだが, 本論文では右辺での複数出現, 或いは出現しない場合を考える. それについては, 次章で詳しく述べる.

2.7 ルール文脈

ルールは膜に所属する. つまり, それぞれの膜の中で最も外側の部分に存在する. ルールのある位置から見て, 左辺に当てはまるプロセスに反応する.

膜にはルールが存在することもあれば, 存在しないこともあるわけだが, ルール自体を扱うルールも書くことができる. そのために使うのがルール文脈である. ただしルールはルールの集合としてしか操作できない. 移動や複製, 破棄が可能だが, ルール自身が含まれる集合については操作できない. つまり, 操作対象にできるのは子膜のルールのみだ.

また, 左辺でルール文脈が書かれていない膜は, ルールが存在しない膜であることを表す. 今まで上で挙げた例は皆そうである.

ただしルール文脈が書かれていても, ルールが無い膜にも反応する.

では, ルール文脈の使用例を示そう.

```
{a,$p[],@p} :- $p[],{b,@p}, @p
```

@p という部分がルール文脈である. 記号@をつけて表現され, その膜に含まれるルールの集合を表している. 左辺での出現はプロセス文脈と同様, 膜ごとに 1 つでなければならないが, 右辺での出現回数に制限は無い.

なお, この例では@の次に来る文字列がプロセス文脈の$以降の文字列と同じ p になっているが, 必ずしもその必要はない. @を含めて名前であり, @以降の文字列に意味はない. プロセス文脈の$についても同様である.

このルールでは, アトム a が含まれる子膜に所属するルールを膜外 (つまり反応ルール自身が所属する膜) に複製している.

なお, 左辺が次のようなルールは, あらゆる子膜に反応する.

```
{$p[|*Z], @p} :-
```

次のルールは, 膜構造をフラットにするルールである.

```
{$p[|*Z],@p} :- $p[|*Z],@p.
```

なお, ルールの左辺・右辺に書かれるものは, **プロセステンプレート**[2]と呼ばれる.

2.8 ガード

これまで示したルールの例では, 全てガード部分が省略されていた. ガード部分は, 左辺や右辺とは少し様相を異にしていて, プロセスが表現されているわけではない. 直感的には, 反応の条件が記述されていると思うとわかりやすい. 次のような例がある.

```
number(N) :- int(N) | ok(N).
```

このルールは, number という 1 引数のアトムと, その第 1 引数に繋がった整数アトムに反応する. 整数アトムとは 1 引数で整数を名前に持ったアトムで, LMNtal では整数をそのような 1 引数のアトムで扱う. int(N) という部分は型制約で, number アトムの第 1 引数にそのような整数アトムが繋がっていることを表している. ガード部分にはこのような型制約をカンマで区切って並べ, ボディとの区切りを記号 | で表す.

反応例を示す.

```
number(0) ⇒ ok(0).
number(13) ⇒ ok(13).
number(-5) ⇒ ok(5).
number("1") (反応せず)
```

""で囲まれた名前は, 文字列であることを示す. 従って, 整数アトムではないので反応しない.

このルールの N は, 正確にはリンクではない. 次に示す型付きプロセス文脈である.

2.9 型付きプロセス文脈

2.9.1 unary

```
x(U) :- unary(U) | ok(U).
```

今度のルールでは,unary という型制約が現れている. この unary(U) の意味は,U は, 単一の,1 引数のアトムである, という意味である. 実は, このルールは下のように読み替えられている.

[2]構文は, 図 2.1 を参照

```
x(X~),$U[X~] :- unary($U[X~])|ok(Y~),$U[Y~].
```

$U[X~] というのは型付きプロセス文脈を表している．この例では unary 型の型付きプロセス文脈である．

型付きプロセス文脈はその名の通り 2.6 節で述べたプロセス文脈に似ており, ルール内で明示されない特定の構造を操作することができる．2.6 節で述べたプロセス文脈を「型付きでないプロセス文脈」とも言うが, そちらは膜内のルール左辺に明示されていない構造全てを表し, そこに当てはまる構造を何ら限定しない．一方, この unary 型の型付きプロセス文脈は, リンク元が必ず明示されている点と, その構造が限定されている点が異なる．

unary 型の型付きプロセス文脈は, 他のアトムと同じように右辺において複製したり消去したりすることができる．構文上リンクのようであるが, ルール右辺で複数回 (あるいは 0 回) 出現してもよい．下図のように, 構造が unary 型ならば名前に関係なく反応する．整数アトムにも反応する．

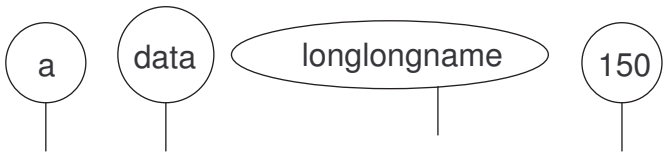

図 2.25: unary 型アトムの例

2.9.2 int

先に挙げた int も型制約であり, 整数アトムであることを示す．当然, int 型のプロセスは全て unary 型である．逆は必ずしも成りたたない．int 型のプロセスに対しては, 例えば以下のような制約が使える．

```
n(N),m(M) :- int(N),int(M),N>=M|large(n),small(m).
```

整数に関しては他にも使える制約, 操作があるが, 本論文では省略する．

2.9.3 その他の型

他にも型や制約, 操作が用意されているが, 全て挙げることはしない．ただし, 第 4 章で述べる基底項プロセスは型付きプロセス文脈の一種である．

第3章　非線形プロセス文脈

3.1　非線形プロセス文脈とは

2.6 節で述べたように, 従来の処理系ではプロセス文脈は右辺において 1 回だけ出現するのでなければならなかった. しかし, プロセス文脈が右辺で複数回登場することで複製を表したり, 出現させないことにより消去を表す, といった操作を考えるのは自然なことである. これらのことが可能になれば, より操作の幅が広がり, 表現の可能性が広がる.

ここで言う「プロセス文脈」は, 「型付きでないプロセス文脈」のことである. 以降この点については逐一断ることをしないが, 単に「プロセス文脈」という場合は型付きでないプロセス文脈のことである. 型付きプロセス文脈について論ずる時は必ず「型付き」という言葉をつける.

3.2　線形プロセス文脈の扱い方

従来の LMNtal 処理系における, プロセス文脈を含むルールの扱い方について, 要点だけを解説する.

LMNtal では, ルールは翻訳されて中間命令列として扱われることになる. 中間命令列についての詳しい解説をここで述べることは避ける. [] などを参考にされたい. また, 本研究に伴い追加された命令等については, 付録にまとめている.

まず, 左辺に書かれた構造に反応した (これを, 以降「マッチした」と表現する) プロセス構造は, 変数番号によって抽象化されて右辺の中間命令列 (ボディ命令列) に渡される. この時, プロセス文脈についてはその構造を得ることはしない. 膜に 1 つしか無く, 左辺に出現する膜内の他のプロセスを除いた「残り」である, という性質を利用している. すなわち, 左辺に出現するプロセスを膜から取り除いてしまった状態で, 膜に残っているプロセス全てを対象にした操作を行うのである.

ただし, プロセス文脈には自由リンクがあり, 明示的なものについては, 右辺で作られたアトム (あるいは他のプロセス文脈) とリンクを繋ぎ直す必要がある. その場合には左辺のプロセス文脈の自由リンクの先のアトムから辿ってリンク先を得る, という方法を採っている.

3.2.1 自由リンク管理アトム

LMNtal において, リンクは膜を何枚でも通過することができる. しかし, 処理系内部においては, 膜を通過するリンクには特別な 2 種類のアトムが付加されることになっている. それは, **自由リンク管理アトム**と呼ばれるもので, 膜を通過するリンクに対して, 下図のように付加される.

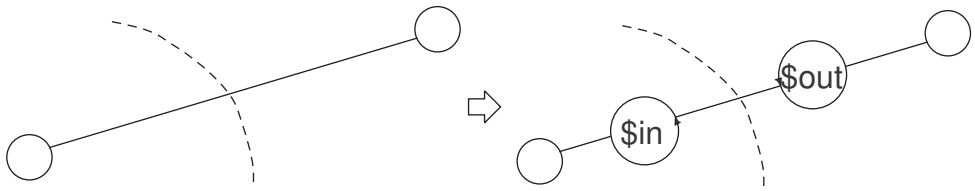

図 3.1: 自由リンク管理アトム

`$in` という名前の 2 引数のアトムが**インサイドプロキシ**, `$out` という名前の 2 引数のアトムが**アウトサイドプロキシ**である. 必ず, 互いの第 1 引数同士が繋がっている. つまり膜をまたぐリンクを持っているのが, それぞれの第 1 引数である.

この自由リンク管理アトムがあるため, 処理系内では膜を 2 枚以上またぐリンクは存在しない. また, 自由リンク管理アトムではないアトムの引数は, 必ず膜の局所リンクを持っている. この仕組みは, 膜の自由リンクに関しては特別な処理をすることが多いために用意されている.

3.2.2 従来のボディ命令列の流れ

左辺でのマッチ, ガードでの型検査についてはここでは触れない. マッチングを取り終わった状態, つまりルールがどのプロセスに対して反応するかが決定した状態で話をする.

ただし, 前述したように, プロセス文脈にマッチしたプロセスを特定していない, ということに注意して貰いたい.

次のようなルールを例として扱う.

```
{a,{b(X), $q[X]}, $p[]} :- $p[],{c},{$q[Y],e(Y)}.
```

また, このルールを翻訳して得られるボディ命令列の一部分を図 3.2 に示す.

この命令列において, 3 はアトム a を, 1 はアトム a の所属する膜を, 4 はアトム b を, 2 はアトム b の所属する膜を表している. それらは左辺のマッチングの際に変数に抽象化されてボディ命令列に渡されている.

また, この中の `removeproxies`, `removetoplevelproxies`, `insertproxies`, それに `removetemporaryproxies` の四つの命令は, 先に述べた自由リンク管理アト

```
removeatom     [3, 1, a_0]
removeatom     [4, 2, b_1]
removemem      [2, 1]
removeproxies  [2]
removemem      [1, 0]
removeproxies  [1]
removetoplev.. [0]
movecells      [0, 1]
newmem      [5, 0]
newmem      [6, 0]
movecells      [6, 2]
insertproxies  [0, 6]
removetempor.. [0]
newatom     [7, 5, c_0]
newatom     [8, 6, e_1]
relink         [8, 0, 4, 0, 6]
```

図 3.2: ボディ命令列

ムを適切に繋ぎ直すための命令である. 本研究には深く関わらないので, 詳しい動作を述べることはしない.

まず, 左辺でマッチしたプロセスについて, プロセス文脈以外を各膜から取り除く. 「取り除く」とは, 一旦各膜の外 (といっても親膜の中ではない.) に置く, というような意味合いである. つまり, 削除されるわけではない. リンク構造も保たれたままである. ルール左辺に自由リンク管理アトムが出現する場合, 自由リンク管理アトムも取り除かれる.

```
removeatom [3, 1, a_0]
removeatom [4, 2, b_1]
```

これらの removeatom 命令はそれぞれ, 3(のアトム (a である)) を 1(の膜) から取り除く命令と, 4(のアトム (b である)) を 2(の膜) から取り除く命令である. この段階での処理系内での状態を, 仮に以下のように表現する.

```
{  {      $q[X]}, $p[]}
 ↓  ↓
 a, b(X)
```

この時, リンク X は繋がったままである.

さらに, 膜構造も取り除かれる.

```
removemem [2, 1]
removemem [1, 0]
```

これらの removemem 命令はそれぞれ, 2(の膜) を 1(の膜) から取り除く命令と, 1(の膜) を 0(の膜, つまりルールのある膜のことである) から取り除く命令である. この時, プロセス文脈にマッチしたプロセスは各膜に残っていることに注意してほしい.

```
{                  $p[]}
   ↓
   {      $q[X]}

 a, b(X)
```

続いて右辺のプロセス構造を生成する. まず初めに膜構造を生成する.

```
newmem [5, 0] newmem [6, 0]
```

これらはそれぞれ, 0(の膜, つまりルールのある膜のことである) に新しい膜を作る命令である. それらには新しい番号として 5,6 が振られている.

```
      { }, {           }.
```

また, 実はこの膜を作った直後に, ルール右辺の出現位置に従ってそこに存在するべきプロセス文脈を配置する. 従って, 図 3.2 を見ると次の命令と前後していることがわかるだろう.

```
movecells [0, 1]
movecells [6, 2]
```

これらは, 1(の膜, 先ほど removemem で取り除かれた膜である) から, 0(の膜, つまりルールのある膜) に, その所属するプロセスを全て移す命令と, 2(の膜, 同じく先ほど取り除かれている) から, 6(の膜) にその所属するプロセスを全て移す命令である. この時, 左辺に出現するアトム a やアトム b は取り除かれているので移動されない. 従って, プロセス文脈にマッチしたものが移動することになる. この時点で次のような状態になっている.

```
$p[], { }, {$q[X],     }.
```

プロセス文脈$q の引数については, 未だ, 取り除かれた状態のアトム b の第 1 引数が繋がっている.

その後, 右辺出現のアトムやルール文脈にマッチしたルール (の集合) が配置さ

れる. この例ではルール文脈は出現しないので, 扱っていない.

```
newatom [7, 5, c_0]
newatom [8, 6, e_1]
```

これはそれぞれ, 5(の膜) に新しくアトム c という名前の 0 引数アトムを作る命令, 6(の膜) に新しくアトム e という名前の 1 引数アトムを作る命令である. それぞれ, 7 と 8 という番号が振られている.

```
$p[], {c}, {$q[X], e( )}.
```

アトム e の引数にはまだリンクが無い.

その後, 右辺のプロセス構造のリンクを繋ぐ作業が行われる. 図 3.2 で言えば, relink 命令がそうである. リンクを繋ぎ直す命令は他にもあるが, その 1 つ 1 つに解説を加えることはしない. 右辺に表現されたプロセス構造の通りにリンクを繋ぐ操作が行われる.

右辺出現のアトム同士ならば新しいリンクを作り, 左辺出現のアトムに繋がっていたリンクを継承する場合は, 左辺にマッチしたアトムから辿ってリンクを取得する.

この時もちろん, プロセス文脈の明示的な自由リンクについても繋ぎ直す必要がある. その反対側は左辺にマッチしたアトムなので, そのアトムから辿って取得したリンクによって繋ぎ直すことができる.

```
$p[], {c}, {$q[Y], e(Y)}.
```

このように, アトムを取り除いてもリンクが保たれる, ということを利用して明示的な自由リンクの繋ぎ替えを行っている. 明示的でない自由リンクについては, 繋ぎ直す必要がない. なぜならば, プロセス文脈が移動される際に, そこから伸びる自由リンクが保たれているからである.

3.3 設計した仕様

実装した非線形プロセス文脈の仕様を説明しよう.

明示的でない自由リンクが無いプロセス文脈のみを扱う.

右辺での出現が無いプロセス文脈の場合, そのプロセス文脈にマッチした構造は削除される.

右辺での出現が複数ある場合, 左辺で出現したプロセス文脈の複製が配置される.

3.3.1 明示的でない自由リンクを持つプロセス文脈を扱わない理由

3.2 節で述べた通り, 従来, 明示的でない自由リンクを扱っていた方法は, アトムが除かれた際にリンクが保たれていることを利用している. ところが複製した場合, 取り除かれたアトムは複製されず, この性質は失われる. これが 1 つの理由である.

また, もう 1 つの理由は, リンク束を扱う機構が整っていないことである.

LMNtal にはリンク束や**アトム集団** (右辺にて, 同種のアトム複数を表現する) といった未実装の概念がある. 明示的でない自由リンクは, リンク束として扱われる. これらの諸概念と共に実装されるべきものである[1].

3.3.2 線形の場合との違い

線形の場合はプロセス文脈にマッチしたプロセスは再利用されるので, 除かれたアトムとのリンクが保たれたままである. 右辺で繋ぎ直す際にはこれを利用して, 左辺でマッチしたアトムから辿ることができた.

複製の際に, この点をどう解決するかが 1 つの焦点である.

3.4 設計方法

まず, 複製の際には操作の対象は膜である. これは従来のプロセス文脈の扱いに習ったもので, 膜の中でプロセス文脈にマッチした以外のアトムについては除かれていることを利用している.

3.4.1 膜内の構造の複製の仕方

まず, 子孫の膜を含めて自由リンクを持たない膜について, その構造の複製の仕方を説明する.

全てのアトムを順番に複製し, 全ての子膜について再帰的に同様の操作を行うことで, リンクを除いて構造を複製することは容易である.

しかし, リンク構造を再現する必要があるため, これだけでは不十分である.

リンクへの参照は, アトムのみ持つ. 従って, アトムの複製と同時にリンク構造の再現を行うことが自然である.

アトムとそのリンク構造の複製アルゴリズムを説明する.

[1] これについては, 6.1 節を参照

$COPY\,ATOMS(DSTMEM, SRCMEM)$:
膜 $SRCMEM$ に所属する全てのアトム $A_i(1 \leq i \leq n(n$: 膜内アトム数)) について
　A_iが複製されていたら
　　何もしない
　A_iが未だ複製されていないならば
　　A_iを $DSTMEM$ に複製
　　$COPY\,ATOMS(A_i, DSTMEM)$ を呼ぶ

$COPY\,ATOMS\_REC(A, DSTMEM)$:
アトム A の持つ全てのリンクの先のアトム $B_i(1 \leq i \leq m(m : A$ の引数の数)) について
　B_iが複製済みならば
　　A の複製と B_iの複製とを繋ぐ新しいリンクを作成
　B_iが未複製ならば
　　B_iを $DSTMEM$ に複製
　　A の複製と B_iの複製とを繋ぐ新しいリンクを作成
　　$COPY\,ATOMS\_REC(B_i, DSTMEM)$ を呼ぶ

この処理が終了した時点で, 膜内のアトムの複製について, 再現されていないリンク構造は無い. なぜならば, 全てのリンクについて, 繋がっているどちらかのアトムが複製される時点でリンク構造も再現されるからである. つまり, この方法で全てのアトム, リンク構造は複製される.

「既に複製されたアトムか否か」を判定する必要性と, また複製されたアトムを参照する必要性の 2 つの理由により, マップによって「複製元のアトム ⇒ 複製先のアトム」という情報を管理している.

3.4.2 複製の際の子膜の自由リンクの扱い方

次に, 子膜において自由リンクを含む場合の, それらのリンクの扱い方を説明する.

アトムを複製する前に, 先に子膜を全て複製する. この時, 前述の「複製元と複製先の情報が管理されたマップ」を, 再帰呼び出しの戻り値として親膜側に返す. なぜなら, 膜をまたぐリンクを作るのは親膜側の仕事だからだ.

また, ここで子膜のルール集合も複製される. プロセス文脈には, 子膜のルールが含まれる (プロセス文脈のある膜自身のルールは含まれない).

$COPYCELLS(DSTMEM, SRCMEM)$:
膜 $SRCMEM$ の全ての子膜 $C_i(1 \leq i \leq m(m$: 子膜の数)) について
　C_iを膜 $DSTMEM$ に複製
　$COPYCELLS(C_i$の複製先, $C_i)$ を呼ぶ
　C_iのルールを C_iの複製先に複製
$COPYATOMS(DSTMEM, SRCMEM)$ を呼ぶ

今度は, 自由リンクを含む可能性があるので, 先に出てきた $COPYATOMS\_REC$ の部分が少し変更される.

$COPYATOMS\_REC(A, DSTMEM)$:
アトム A の持つ全てのリンクの先のアトム $B_i(1 \leq i \leq m(m : A$ の引数の数)) について
　B_iが A のある膜の子膜にあるならば
　　子膜の複製先にある B_iの複製と A の複製との間にリンクを作成
　B_iが同膜内に無く, A のある膜の子膜にも無いならば
　　何もしない
　B_iが A と同膜内にあるならば
　　B_iが複製済みならば
　　　A の複製と B_iの複製とを繋ぐ新しいリンクを作成
　　B_iが未複製ならば
　　　B_iを $DSTMEM$ に複製
　　　A の複製と B_iの複製とを繋ぐ新しいリンクを作成
　　　$COPYATOMS\_REC(B_i, DSTMEM)$ を呼ぶ

この変更から分る通り, 子膜の複製時には, 親膜側へのリンクについては「何もしない」. リンク作成時には両端のアトムは複製されている必要があるので当然である.

これで, プロセス文脈は自由リンクを除いて複製される.

3.4.3 プロセス文脈の明示的な自由リンクの扱い方

続いてプロセス文脈自体の自由リンクの扱い方を説明する. 3.3.1 節で述べたように, 自由リンクは必ず明示されている.

線形の場合に行われている方法は, 左辺でプロセス文脈の引数から伸びるリンクが繋がっているプロセスにマッチしたプロセスから辿る方法である. つまり, 複製元のプロセス文脈へと伸びるリンクは取得することができる. そこでこれを利用し, 複製元のプロセス文脈へのリンクから複製先のプロセス文脈へのリンクを

得る命令を作成する方法を採用した. 複製元のリンクから複製先のリンクを得る操作を行う, 新しく作った命令は「LOOKUPLINK」である.

これを実現するためには, プロセス文脈の中身の複製時に作られる「複製元アトム → 複製先アトム」というマップを利用する必要がある.

3.4.1, 3.4.2 節で述べたのはプロセス文脈の中身の複製である. この処理は新しく作成された命令「COPYCELLS」によってなされる.

そこで, この COPYCELLS の出力として, 「複製元 → 複製先」のマップを LOOKUPLINK 命令に渡すことにした. それによって, 線形プロセス文脈と同様に非線形プロセス文脈についてもリンクの繋ぎ直しを行うことができる.

従来の MOVECELLS 命令の発行と同様に COPYCELLS を, また GETLINK 命令に加えて LOOKUPLINK 命令を発行するように変更することで, 非線形プロセス文脈を含むルールを扱う.

3.4.4 廃棄について

非線形プロセス文脈とは, プロセス文脈の複製だけでなく, 廃棄操作も含む.

廃棄アルゴリズムについては複製操作より単純で, 子膜を再帰的に廃棄した後, 膜内のアトムを全て廃棄するだけである. この処理は DROPMEM 命令によって実装された.

3.5 発覚した問題

これらを実装した結果, 複製時にある問題が起こることがわかった.

次のようなプロセス文脈の複製ルールがあったとする.

```
{a(X,Y), $p[X,Y]} :- {a1(A,B), $p[A,B]},{a2(C,D), $p[C,D]}.
```

このルールに対し, 次のようなプロセスが反応する.

```
{a(a,b)}.    ⇒ {a1(a,b)},{a2(a,b)}.
{a({b,c}, {d,e})}. ⇒{a1({b,c}, {d,e})},{a2({b,c}, {d,e})}.
{a(a,b), {c}}. ⇒ {a1(a,b), {c}}, {a2(a,b), {c}}.
```

次のようなプロセスも反応する筈であるが, これが正しく動かないことがわかった.

```
{a(A,A)}.  ⇒ {a1(A1,A1)},{a2(A2,A2)}.
```

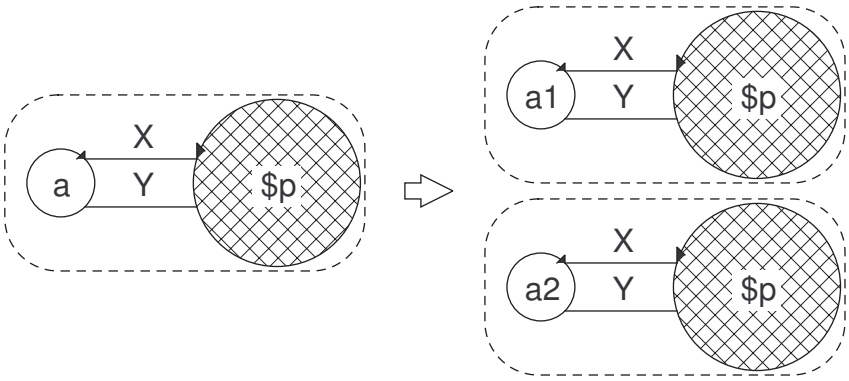

図 3.3: プロセス文脈の複製ルールのグラフィカルな表現

このプロセスを図にすると次のようになる.

つまり, $p の第 1 引数と第 2 引数が$p の中で繋がっている. この場合, アトムの複製を中心に行われる上記のアルゴリズムでは何も複製されない. 従って, LOOKUPLINK 命令によって複製先のリンクを得ようとしても, 失敗する. マップに登録されているのはリンク先のアトムの筈なのであるが, 今回の複製対象にはそのアトムが含まれていない.

この問題を解決するために, 次のような対処をした.

3.5.1 解決策

プロセス文脈を複製する前に, 準備としてその引数についてある操作をする. すなわち, 引数同士がプロセス文脈の内部で繋がっている場合は, そこに=アトムを挟み込むのである. こうすることにより, 前述の問題が回避される. なぜならプロセス文脈の複製操作においては「膜内に残っているプロセス全て」を複製するので, 本来のプロセス文脈には含まれていないこのアトムも一緒に複製されるからである. その意味ではここに挿入するのはどんなアトムでも (単一のアトムですらなくとも) 良いのだが, 一方で LMNtal で=アトムはリンクの短絡に使われるので直感的にも自然である. ただし, =アトムは複製後に残ってしまうので, これを削除する必要が生ずる.

繰り返しになるが, ルールにおける'=' について簡単に説明する. =という名前の 2 引数のアトムをルール右辺に書くと, 処理系によって瞬時に削除されその引数に繋がるリンク同士は融合する, という機能は, 現在の処理系では, =アトムは作らずに処理している. 作る代わりに, そのリンク先同士を繋ぐのである. つまり, =アトムは実際には出現することはない.

従って, ルール適用中に挿入した場合はルール適用中に削除する必要がある.

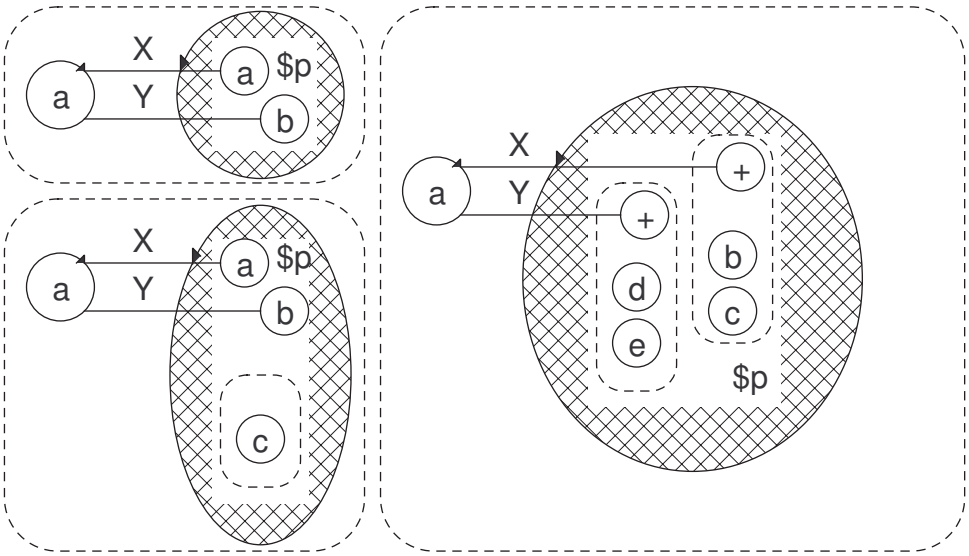

図 3.4: プロセス文脈の中身

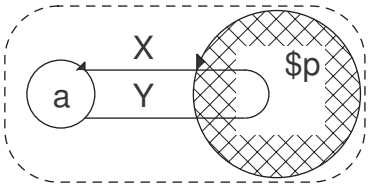

図 3.5: プロセス文脈の中身

もちろん, 削除するのは挿入された=ではなく複製のほうなので, 先に挙げた COPYCELLS 命令の出力で得られたマップを利用する必要がある. その際, このマップから参照する際に必要なので, 複製前の=を渡す必要がある.

以上を実際に行う命令として, INSERTCONNECTORS 命令と DELETECONNECTORS 命令を追加した.

前者はプロセス文脈の自由リンクのリストを引数に持ち, (必要があれば) その引数間に=アトム (コネクタ) を挿入する. 挿入した場合はそれらをセットで管理し, 命令の出力として返す.

後者は複製後の膜から挿入したコネクタの複製を削除し, リンク先同士を繋ぐ. その際に, COPYCELLS 命令が出力する「複製元 → 複製先」のマップと, INSERTCONNECTORS 命令が出力したセットを利用する.

3.6 使い方

例とそのグラフィカルな表現を通じて, 非線形プロセス文脈の使い方を解説する.

```
{cp, $p[]} :- {$p[], $p[]}.
```

これは, 左辺の膜の cp アトム以外の部分を全て複製する. 次のような適用例に

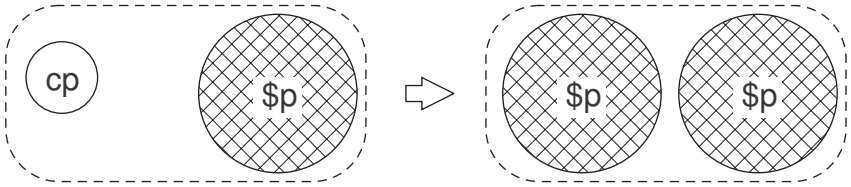

図 3.6: プロセス文脈の複製

なる.

```
{cp,a,b,c} ⇒ {a,a,b,b,c,c}
{cp,cp,n} ⇒ {cp,cp,n,n} ⇒ {cp,cp,n,n,n} ...
{cp,{l,m,n}} ⇒ {{l,m,n},{l,m,n}}
```

また, 次のようなルールならば,

```
{cp,$p[]} :- $p[],$p[].
```

次のように反応する.

```
{cp,{cp,{data}}} ⇒{cp,{data}},{cp,{data}}⇒⇒
{data},{data},{data},{data}
{cp,{data},{cp,{data}}} ⇒
{data},{data},{cp,{data}},{cp,{data}} ⇒
{data},{data},{data},{data},{data},{data}
```

ガードを使い, もっと実用的に改良すると「指定した数だけ複製する」プログラムになる.

```
{cp(N),$p[]} :- int(N),N>0 | {cp(N-1),$p[]},$p[].
{cp(N),$p[]} :- int(N),N=0 | ().
```

適用例は, 次のようになる. なお ⇒ の数は, ルール適用の回数を示す.

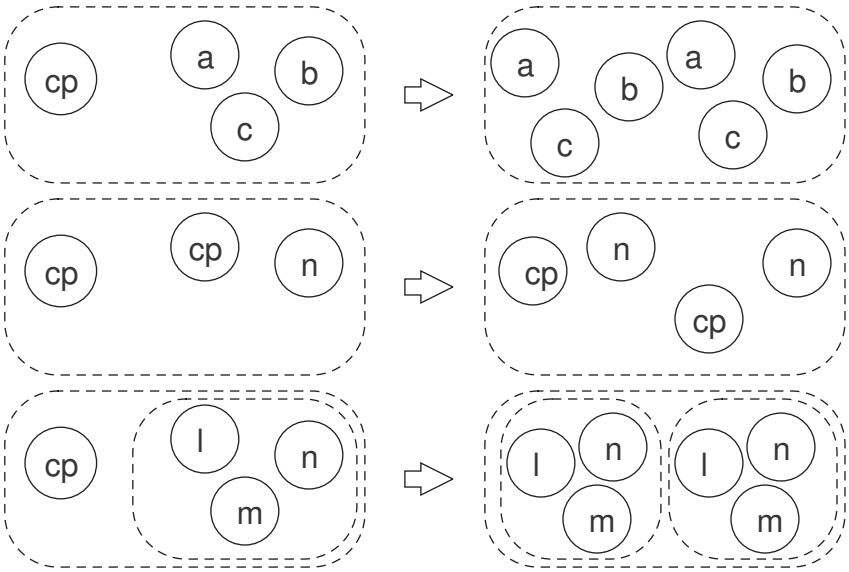

図 3.7: プロセス文脈の複製ルールの適用例

```
{cp(3),a,b,c} ⇒⇒⇒⇒ a,a,a,b,b,b,c,c,c
{cp(5),data} ⇒⇒⇒⇒⇒⇒ data,data,data,data,data
{cp(2),{cp(2),data}} ⇒⇒⇒⇒⇒⇒ data,data,data,data
{cp(0),{dust}} ⇒
```

自由リンクを持ったプロセス文脈を扱うこともできる.

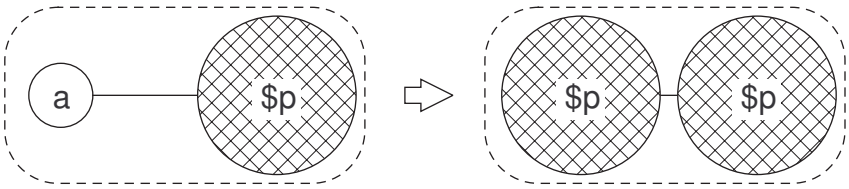

図 3.8: 自由リンクを持つプロセス文脈の複製ルール

```
{a(X),$p[X]} :- {$p[Y],$p[Y]}.
```

以下のように適用される.

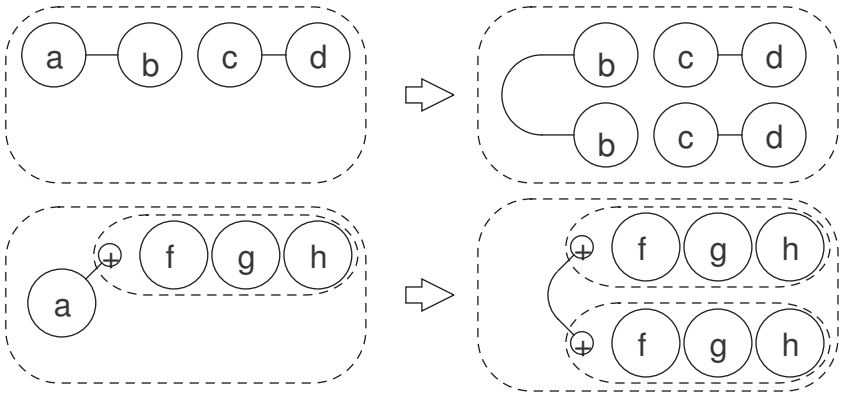

図 3.9: 自由リンクを持つプロセス文脈の複製例

```
{a(b),c(d)} ⇒ {b(b),c(d),c(d)}
{a({f,g,h})} ⇒ {{'+'(M),f,g,h},{'+'(M),f,g,h}}
```

第 2 章で述べたように, アトムの引数に膜を書いた場合は'+' アトムの省略とみなす.

第4章　基底項プロセス

4.1　基底項プロセスとは

第2章で挙げた, unary 型の発展例として, 基底項プロセスという型を説明する. ground 型とも言う. 直感的には, 閉じていて, 分断されておらず, 1本だけ自由リンクを持つような構造である. 外部に通ずる経路を1つだけ持つようなネットワーク構造のイメージに近い. 木構造や葡萄の房のようなものを想像してもよい.

まとまった構造を, 1つのデータのように扱うための概念であり, 比較, 複製, 破棄が可能である. 特に重要なのは比較である.

基底項プロセスの厳密な定義は, 次のようなものである.

> 基底項プロセスが持つ自由リンクは1つだけである． このリンクを**根**という． 基底項プロセスは，膜をまたぐリンクを持たないアトムとそれらの持つリンクから成る． 基底項プロセスを構成する全アトムは，根からリンクを辿って到達可能である.

図 4.1 に基底項プロセスの例を示す.

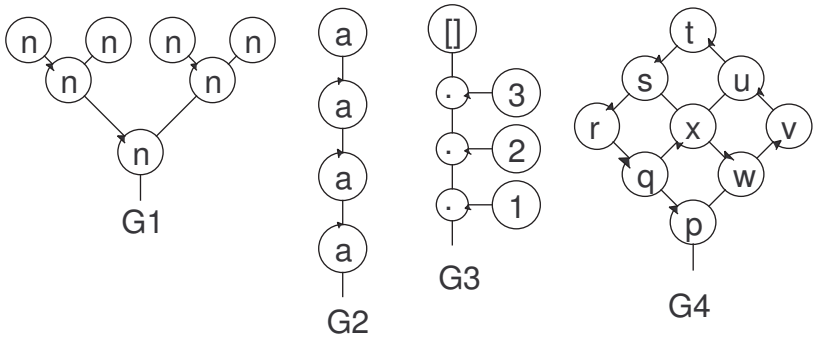

図 4.1: 基底項プロセスの例

```
G1=n(n(n,n),n(n,n)).
G2=a(a(a(a))).
G3=[1,2,3].
G4=p(q(r(s(t(u(v(w(x(Q,S,U),P)),U)),S)),Q),P).
```

基底項プロセスは, 膜をまたがない. つまり, (処理系内では) 自由リンク管理アトムを含まない構造である. また, 環を含む構造, つまり根から各アトムへの辿り方が 1 通りでないような構造も含まれる. リストも基底項プロセスである.

4.1.1 他の基底項プロセス設計案

基底項プロセスが膜を含まない設計になっている理由は, 比較を目的の 1 つとしているからである. その意味を以下に説明する.

基底項プロセスの設計案としては, 前述の定義の他に, 3 つの案があった. 前述した採用案を「案 1」と呼ぶ.

以下では「基底項プロセスの持つ自由リンクは 1 つだけであり, これを根と言う」という部分は共通するので省略する. また, 型付きプロセス文脈に共通することであるが, 根を除いて親膜へのリンクが含まれることはない. なぜなら, 型付きプロセス文脈が「所属する」膜及びその子孫膜内にその全構成要素を持っていなければならないからである.

案 2:

> 根からリンクを通じて辿れるアトムを含む. また，構成アトムの持つリンクを通じて辿れるアトムも含む. そして，構成アトムの所属膜と，その膜に所属する他のアトムも含む.

例を図示すると, 下図に表示されている全てを含む.

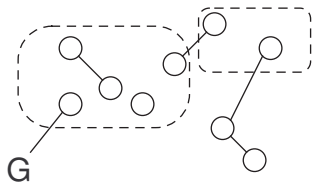

図 4.2: 基底項プロセス案 2

案 3:

> 根からリンクを通じて辿れるアトム及び，その所属膜を含む． 所属膜内に根からリンクを辿って到達不可能なアトムは存在しない．

例示すると, 次のようなものである.

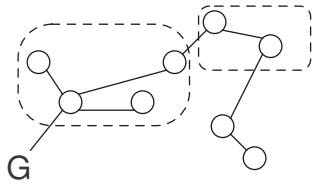

図 4.3: 基底項プロセス案 3

案 4:

> 根からリンクを通じて辿れるアトムを含むが，それらの持つリンクは膜を通過していても構わない． ただし，通過している膜や，その内容物は含まれない．

下図の, 矢印で示されたものを除いた部分である. 案 1 において膜をまたがって

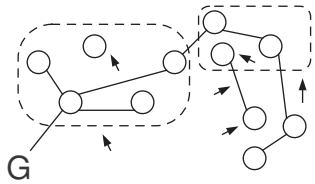

図 4.4: 基底項プロセス案 4

いることもある, という定義である.

4.1.2 他案の却下理由

これらの案が却下された理由を説明する.

まず案 2〜4 に共通する特徴として案 1 と異なるのは, 膜を含むということである. 基底項プロセスの主な導入目的は, 一まとまりの構造をデータとして扱えるようにすることである. つまり, 案 2〜4 の案 1 と異なる長所は, 膜を使ったデータのモデル化ができるということでもある.

さて, その観点で見たとき, 案 4 は明らかに目的に対して効果が無い. 膜をまたがってはいても, 含まれるのはアトムのみである. よって案 4 は排除した.

また, 案 3 は膜を含むものの, 実質的には膜が無駄に使われている. 膜で一部分を囲むことに意味を見出すようなモデル化があったとしても, そこにアトムを挟むなどして境界を特徴づけるほうが自然である. また, 後述する膜数検査への影響も排除理由である.

案 2 は, 実用性はありそうである. アトム同士は必ずしもリンクで繋がっていない. 直感的には, 下図のように, 案 1 における各アトムが「時には膜のこともある」構造である. この図において膜の表面から伸びているように見えるリンクは,

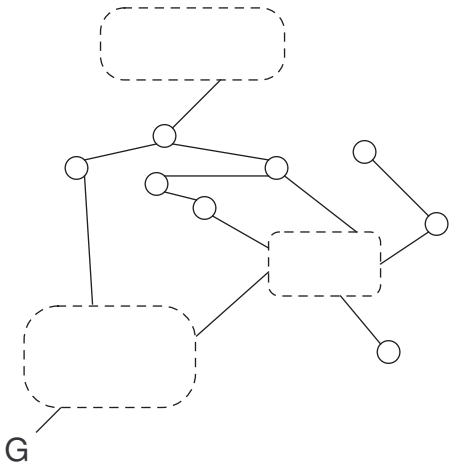

図 4.5: 基底項プロセス案 2

内部のアトムに繋がっている.

ただし, 案 2 は, 実装上困難な問題があることがわかった.

4.1.3 膜を含む構造の比較の困難

案 1,3,4 においては, 全ての構成アトムはリンクで繋がっている. したがって, 検査も比較も複製も破棄も, 構造をリンクを通じて辿ることで効率的に行える.

しかし, 案 2 においては膜の部分で, リンクを通じて辿れないことがある.

検査についてはそれ程困難ではない. 多少複雑にはなるものの, 左辺出現のプロセスに繋がっていないことと, 親膜へのリンクを含まないことだけを確かめれば良い. 各構成アトムの所属膜において, 全アトムに同様の検査が必要だが, 特に非効率的ではない.

複製・破棄は, 前章で紹介した非線形プロセス文脈の複製・破棄と似た操作で可能である. 排除する理由にはならない.

問題は比較である.

膜を含む構造の比較が困難な理由は, プロセス文脈の比較機能を実現するのが難しい理由と同一である.

膜の比較において, ルールの同一性を検査する必要は無いとしても (必要性が見当たらない), 子膜の構造比較は当然必要である. ただし, 子膜に順序は無いので, 自膜と比較対照膜の子膜の組み合わせ全てを検査する必要がある.

比較元の膜に M 個の子膜があったとする. 比較対照の膜にも子膜が M 個ある (でなければ比較検査の失敗は確定する).

まず, M 個の子膜の中から 1 つ選び, 比較対照の膜の子膜 m 個から 1 つ選ぶ. 次に, 残りの $M-1$ 個の子膜から 2 つ目を選び, 比較対照の膜の子膜残り $M-1$ 個から 1 つ選ぶ. これを繰り返して M 個の割り当てを全て検査する. 失敗時には全てのパターンを検査するので, 計算量は $M!$ のオーダーになる.

加えて, 子膜にも子膜がある. 上の全パターンの検査ごとに, 子膜同士の比較検査があり, K 階層の膜 (各階層の子膜数を大体 M_k とすると) の比較は, $\prod_{n=1}^{k} M_k!$ になる. それに加え, アトムの比較検査も加わる.

下図のような膜構造を見れば, 厄介であることは一目瞭然であろう.

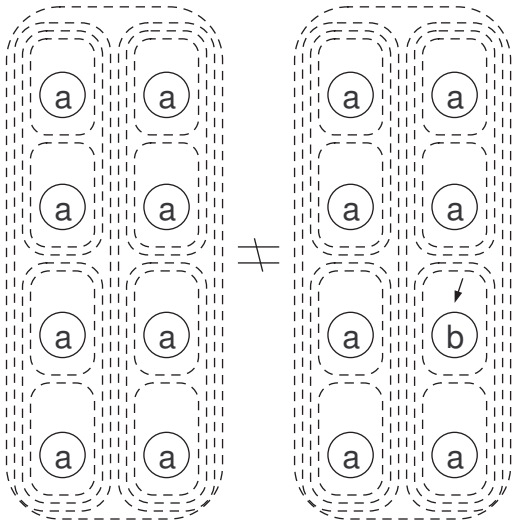

図 4.6: 比較困難な膜構造の例

もちろん膜を含むルールのマッチングでは, 確かに膜の比較を行っている. ただしこの時は比較元の膜構造が特定されている上にルール反応に関係のあるプロセスのみの検査である. それに対し, プロセス文脈や基底項プロセスの比較では比較元さえ特定されていない.

そもそも, このような, リンクで繋がっていない不特定多数要素の比較が容易でないのは明らかであり, そのためにリンクがあるのである. リンクの存在理由に

は, マッチングに対する便宜を図る目的もある. 言うならば, ユーザが処理系に対して, 「この 2 つのデータはまとめて扱われることが多い」ことを教えるためにリンクで繋ぐのである.

そうした考え方に拠り, プロセス文脈を比較する目的に使うのは望ましくない. 案 2 が排除された理由, つまり基底項プロセスに「リンクで繋がっていない」部分を許さないことにしたのも, 同じ理由による.

なお, 左辺でプロセス文脈が存在しない膜においては, アトム数検査, 膜数検査が必要になる. 型付きプロセス文脈が左辺に存在する場合, そこに含まれるアトム数も当然考慮しなければならず, 案 2 ではなく案 1 を採用することにより, 膜数検査には影響を及ぼさずに済む, という利点もある.

基底項プロセスに型づける制約のことを ground 型制約と言う. 基底項とは ground の訳語である.

4.1.4 unary との関係

unary の定義は, 単一の 1 引数のアトムである.

従って, もちろん全ての unary 型のプロセスは基底項プロセスなので, ある型付きプロセス文脈に対して unary 型制約と ground 型制約が両方かかっている場合, unary 型制約のみかかっていると解釈して構わない. この意味で, unary 型制約は ground 型制約より強い制約と言える.

ground 型の検査や比較, 複製や廃棄の仕組みは, 従来実装されていた unary 型に関する機能に習って設計した.

unary 型と基底項プロセス型の大きく異なる点は, 構成アトム数が決定されていない, という点である. unary 型の場合, アトム数は必ず 1 であり, アトム数検査において考慮するのは unary 型プロセス文脈がいくつあるか, という情報だけで済んでいた.

4.2 設計方法

基底項プロセスについて, 検査・比較・複製・破棄を実際に行う各命令のアルゴリズムを解説する. また, 付録の各命令の紹介も参照されたい.

4.2.1 検査アルゴリズム

基底項プロセスかどうかの検査は, 以下の 2 つから成る.
根から辿って到達可能な全てのアトムが,
・自由リンク管理アトムでないかどうか.
・ルール左辺に出現するアトムでないかどうか.

2 つ目の検査は, 以下のようなルールを見ればわかりやすい.

```
a(X,Y),b(Z) :- ground(X) | ...
```

このルールにおいて, リンク X を根とする基底項プロセスは, Y や Z を含まない. つまり, 左辺に出現するアトムに到達しないことを検査することで, その基底項プロセスを含むプロセスが左辺にマッチすることが検査される.

基底項プロセスの構造検査は, 再帰的に構造を辿って上の M 条件を検査する. だが, 基底項プロセスは木構造だけでなく循環構造などを含むこともあるため, 一度検査したアトムを 2 度辿らない工夫が必要である.

この目的と, 左辺出現の「先の避けるべきアトム」の情報を管理するため, それぞれセットを使用することにした. すなわち, 再帰的に構造を辿る際に, 一度辿ったアトムは「検査済みセット」に登録する. 検査済みセットに登録されたアトムに辿りついたら, それ以上進まない. 避けるべきアトムが登録された「忌避セット」に登録されているアトムに辿りついたら, 検査は失敗する. 自由リンク管理アトムに辿りついた場合も, 検査は失敗する. 検査に失敗することなく, 根からの検査が全て終了すれば, 成功である. 検査命令の出力として, 構成アトム数が返される.

この検査のために, 予め避けるべきアトムを登録されたセットが必要なので, セットを新しく作る NEWSET 命令と, そこにアトムを登録する ADDATOMTOSET 命令を追加した.

4.2.2 アトム数検査

また, 基底項プロセス自体の検査に加え, 基底項プロセスを含む膜のアトム数の検査が必要な場合がある. 型付きでないプロセス文脈が無い膜の場合である. このアトム数検査は, その膜に所属するアトム数がルール左辺で表現されたプロセス構造から指定される数と等しいかどうかで検査される. だが, ground 型制約が加わると, 従来とはやや異なる部分がある. 従来の, アトムや unary 型の型付きプロセス文脈を含む場合のアトム数検査においては, 膜に含まれるべきアトム数はコンパイル時に決定できていた. なぜならば, unary 型プロセス文脈は常に構成アトム数は 1 だからである. しかし, 基底項プロセスは構成アトム数がコンパイル時に特定できないので, アトム数検査において必要な情報, 「膜に含まれるべきアトム数」を動的に取得する必要がある. そこで, 基底項プロセスの構造検査時に, 同時に構成アトム数という情報を得る必要がある. 型付きでないプロセス文脈の無い膜にマッチした膜に所属しているべきアトムの数は, 左辺に出現するその膜に所属する, 「アトムの数+unary 型付きプロセス文脈の数+ground 型付きプロセス文脈の構成アトム数の合計」である. この数を動的に取得して検査する.

4.2.3 複製アルゴリズム

複製操作の段階では, 基底項プロセスの検査が終了している. 従って, 再帰的に辿って, 複製していけば良い. 基本的には, 前章で述べた非線形プロセス文脈の複製操作と同様 (全てのアトムが繋がっているので, 正しくはその一部) である. 複製済みかどうかのマップを受け渡しながら複製しつつリンク構造を再現する.

この処理は, 追加した命令 COPYGROUND によって行われる.

4.2.4 比較アルゴリズム

ここでは, 基底項プロセスの比較検査を説明する. 基底項プロセスの比較とは, 等価性の検査である. 比較時には, 基底項プロセスであるかどうかの検査は終了しているので, 2 つの基底項プロセスが等しいかどうかの検査をすればよい. ここで言う「等しい」とは, 同じ構造を持っている, ということである. すなわち, 根から同じように辿った時, 到達するアトムが必ず等しく, 到達位置の引数番号も等しい. アトムが「等しい」とは, 名前が同じで, 引数の数が等しいということである.

さて, 当然比較検査も, 再帰的に行うことになる.

直感的には, 比較検査と複製操作はきわめて似ている. 複製する際に辿る方法で, 各アトムが等しいかどうか見ていけば, 等しいかどうか検査できる. 複製時にはリンク構造を再現するので, リンク構造の比較も同様のアルゴリズムで可能である. 複製時と同様に, 検査したかどうかを管理する. 比較元から比較先への参照をマップで管理することで, 既に辿ったアトムに辿りついた際に以前と同じもの同士かどうかを検査する.

比較検査と複製操作を似た方法で行える理由は, 全アトムがリンクで結ばれた構造だからである.

比較を行う命令は, 新しく作った EQGROUND 命令である.

4.2.5 除去アルゴリズム

基底項プロセスの膜からの除去はきわめて簡単である. unary の場合と同様の操作を再帰的に全ての構成アトムについて行えば良い. 検査アルゴリズムと同様に除去済みかどうかをセットで管理するが, 検査は済んでいるので失敗の検査が必要無い分, 単純である.

破棄を行う命令は, REMOVEGROUND 命令である.

4.3 使い方

基底項プロセスの使い方を簡単な例を通じて紹介する.

```
g(G) :- ground(G) | ggg(G,G,G).
```

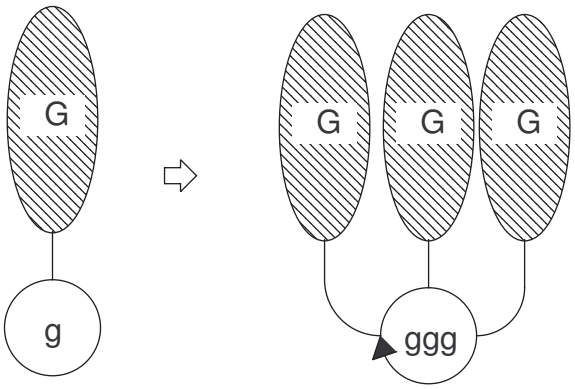

図 4.7: 基底項プロセスの複製

このように, 型付きプロセス文脈はリンクのように見えるが, 右辺にいくつも出現してよい. このルールの適用例を次に示す.

```
g(a) ⇒ g(a,a,a)
g(n(n(n(N)),n(N))) ⇒
g(n(n(n(N1)),n(N1)),n(n(n(N2)),n(N2)),n(n(n(N3)),n(N3)))
g([1,2,3]) ⇒ g([1,2,3],[1,2,3],[1,2,3])
```

最後の例のように, リストも基底項プロセスとして扱うことができる. リストを使ったルールとして, 次のようなものもある.

```
L = LIST,del=DEL :- ground(LIST),ground(DEL),LIST=DEL | L=[],del=DEL.
```

これは比較を用いた例である. =制約で, 等しい基底項プロセスのみ反応する, という制限をかけることができる. このルールは, 指定された部分リストを切り取る.

```
list=[[1,2,3],[3,5,4,5],[1,2,3,4,5],[2,5]],del=[4,5] ⇒⇒
list=[1,2,3],[3,5],[1,2,3],[2,5]]
del=[a,b,c],list=[a,b,c] ⇒ list=[]
```

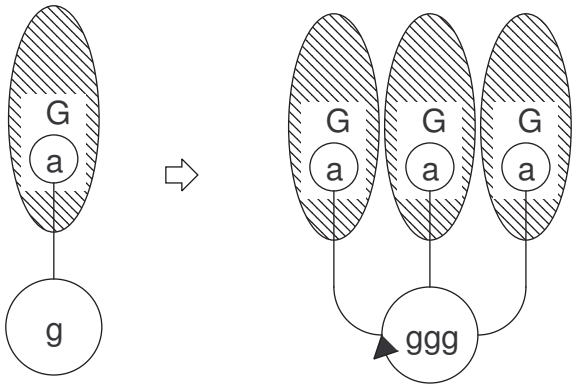

図 4.8: 基底項プロセスの複製ルール適用例

4.4 基底項プロセスによる膜比較機能の代行

4.1.3 節で述べた理由により, 基底項プロセスは膜を含まない構造に限定されている. また, 型付きでないプロセス文脈には, つまり膜には比較機能が無い.

しかし一方で, オブジェクトを膜で表現するようなモデル化を考えることはできる. そして 2 つのオブジェクトを比較したいという欲求も自然なものである.

このように膜を比較したい場合, 基底項プロセスの比較機能を用いることで, この目的を満たす方法がいくつか考えられる.

例えば一番単純な方法は, 膜内の全てをリンクで繋いでしまい, 参照として 1 つだけ自由リンクを出す, という方法である. しかしこうしたモデル化で良いのならそもそも膜を持ち出す必要は無い.

膜でオブジェクトを表現するのは, その階層性と, 計算の局所化が 1 つの目的である.

そして, 一方で比較したいのがその膜の要素全てであるとは限らない.

例えば, 「人間」をモデル化したと考えてみよう.

現実において, 人間が同一人物と判断される理由は時によって様々である. 顔や身体的特徴がまずまず近い, 筆跡が同じである, 同じ身分証を持っている, 同じ席に座っている, 同じ振る舞いをする.

状況によって比較したい特徴がごく一部だったり, 変化したりするかもしれない.

であればそれに習った方法, 膜において比較したい部分だけを基底項プロセスとして表現するやり方が自然であろう.

単純だが, 例えば次のようなルールである.

```
PERSON1={eq(E1),$p[]}, PERSON2={eq(E2),$q[])}  :- E1=E2 |
{eq(E1),$p[],$q[],'+'(PERSON1),'+'(PERSON2)}.
```

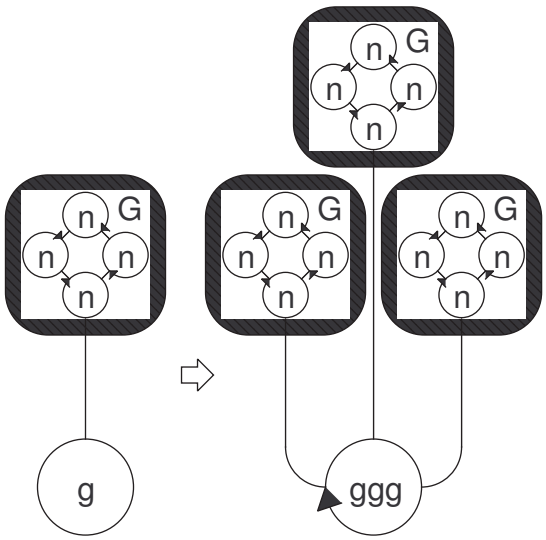

図 4.9: 基底項プロセスの複製ルール適用例

このルールによって, 次のように表現された人間が「同一人物」と判定される. 時には名前で, 時にはしていることや年齢によって.

```
boy={eq=name(robert),age(10)}.
bob={eq=name(robert),doing(cleaning))}.
girl={name(ada),eq=[age(20),doing(running)]}.
someone={eq=[age(20),doing(running)]}.
```

”boy”=”bob”であり, ”girl”=”someone”であると分る.

4.5 型制約の省略について

型制約は一部省略可能である. ground と unary の場合について述べると,

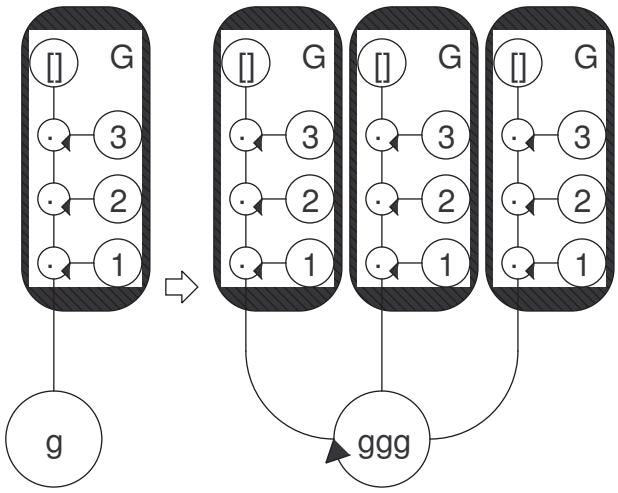

図 4.10: 基底項プロセスの複製ルール適用例

unary(X) と X=Y が共に書かれている場合，unary(Y) が書かれていると判断する．
unary(Y) と X=Y が共に書かれている場合，unary(X) が書かれていると判断する．
X=Y が書かれている場合，ground(X)，ground(Y) が書かれていると判断する．
以上を踏まえて，ある型付きプロセス文脈 X が unary(X) と書かれていると判断できてかつ ground(X) と書かれていると判断できる場合，ground(X) は無視される．

ground はもっとも弱い制約である．

第5章　実装した機能を使ったプログラムの例

これまで述べた機能を実際に使った例として, 膜の複製によって探索問題を解く例を 2 例紹介する.

5.1　SAT を解くプログラム

SAT は, 各変数について $true$,$false$ の 2 通りだけなので, 探索木の分岐が必ず 2 本である. 複製を利用した解き方の簡単な例として SAT ソルバを書いてみた.

付録にソースコードを添付しているので, そちらを参照されたい.

5.1.1　問題とデータの扱い

初期状態は, 次のような構造になる.

```
space={a(1), a(2), a(3), ..., z(0), unsolvable}
```

この膜は 1 つの変数割り当てを示すのに使われる. 各アトム a(N) が変数 x_N(この

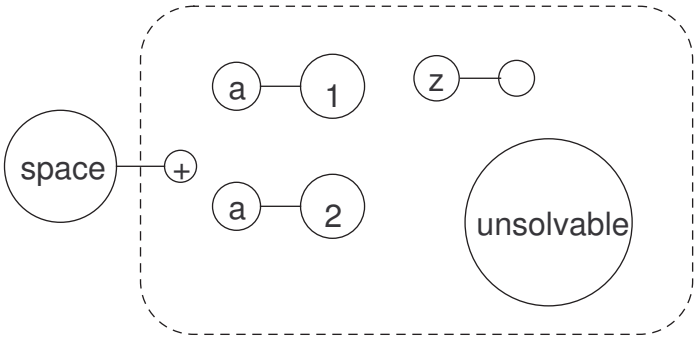

図 5.1: space 膜の様子

段階ではまだ値が割り当てられていない) を表す. z については後述する.

unsolvable は単に, 「まだ解ではない」ことを示す.

この space 膜 (もちろん, 正確にはアトム space と繋がる'+' アトムを持つ膜, である) が繰り返し複製されることで, 探索が行われる.

問題は, 次のように表現される.

```
m(M).
c(1,{x(1),notx(2),,,}).
c(2,{notx(1),notx(3),x(3),,,}).
...
```

CNF 形式の問題が表現されている. アトム m(M) は, 節の数が全部で M 個である

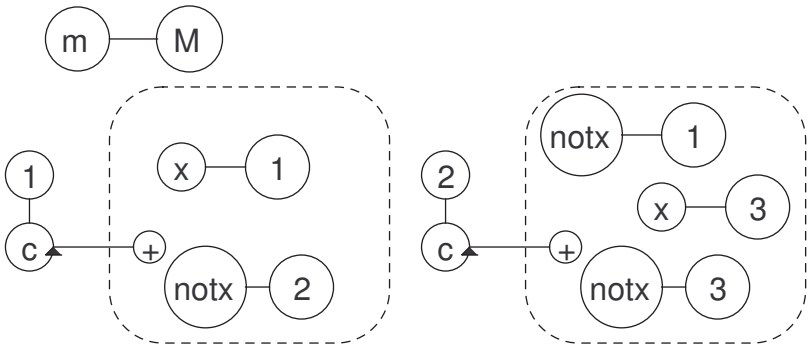

図 5.2: SAT 問題の表現

ことを示す. c(K,{...}) は CNF 形式の各節 C_K を表し, 第 2 引数に繋がっている膜内で, リテラル $x_N, \neg x_N$ が x(N), notx(N) の形で表現される.

5.1.2 ルール

複製ルールと検証ルールからなる.

膜の複製のルールは、次のように表現される.

```
space={a(N),$v[]}  :- int(N)  |
space={t(N),$v[]},space={f(N),$v[]}.
```

各変数について t($true$),f($false$) の 2 通りである. 従って変数の数に応じて指数関数的に膜の数は増加することになる.

検証ルールは, 次のように表現される.

```
m(M),space={ unsolvable, z(Z), $s[]}  :- int(Z),int(M),Z=M|
m(M),space={   solvable,       $s[]}.

c(M,{   x(Nx), $c[]}), space={t(Nt), z(Z  ), $s[]}  :-
int(M), int(Nx), int(Nt), int(Z), Nx=Nt, Z=M-1 |
c(M,{   x(Nx), $c[]}), space={t(Nt), z(Z+1), $s[]}.
c(M,{notx(Nx), $c[]}), space={f(Nf), z(Z  ), $s[]}  :-
int(M), int(Nx), int(Nf), int(Z), Nx=Nf, Z=M-1 |
c(M,{notx(Nx), $c[]}), space={f(Nf), z(Z+1), $s[]}.
```

z(Z) は, C_1 から C_Z が満たされる (全ての $C_k(1 \leq k \leq Z)$ について, 節 C_k の 1 つ以上のリテラルが真になる) 組み合わせであることを示す. 従って m(M) について z(M) ならば, 全ての節が満たされる, つまり解であるということがわかる.

5.1.3 ルールの優先性について

現状の処理系では, ルールは上に書かれているものから適用検査が行われる[1]. これはあくまで現処理系の性質であり, ソース上でルールの優先関係を表記する方法というわけではない. ただし, 本プログラムにおいてはこの性質を利用している. ルールの優先性については, 第 6 章で今後の課題として取り上げる.

5.1.4 改良

上記の理由により複製ルールを検証ルールの前に書くと, 複製が完全に終わってから検証を始めるので, 解ける場合も解けない場合もほぼ同じだけ時間がかかる. そこで, 検証ルールを複製ルールの前に書いて先に実行させることで, 複製する量を減らすことができた. 解けない場合にはかかる時間はそれ程変わらないが, 解ける場合の結果は十倍以上変わる.

5.2 狼と羊とキャベツの問題

上で述べた例では, 正解の判定をルールで書くことができたが, 問題によっては汎用のルールが書けない場合がある. また, 探索過程で以前辿った状態にたどり着くこともあり, それを判定して排除する必要がある. これらの解決に基底項プロセスの比較機能を使った例を紹介する.

[1] 適用順をランダムにするモードも存在する

5.2.1 問題の説明

農夫が, 狼と羊とキャベツをそれぞれいくつか持っている. これらを連れて川を渡るのだが, ボートは 1 隻しかなく, ボートに一度に乗せられる数には限りがある. ただし, 狼と羊を残して往復すると, 狼が羊を食べてしまう. また, 羊とキャベツを残して往復すると, 羊がキャベツを食べてしまう. それは避けたい.
ボートの定員に満たなくとも (何も乗せていなくても) ボートを移動することはできる. 最も早く, 全て運ぶことができる手順を見つけよ.

例として, 狼, 羊, キャベツがそれぞれ 1 つずつの場合だと, 次のような 7 手が最短になる.

1 : 羊を移動
2 : 空のボートを移動
3 : 狼を移動
4 : 羊を移動
5 : キャベツを移動
6 : 空のボートを移動
7 : 羊を移動

ただし, 3 手目と 5 手目は逆でも良い.

例えば, ボートの定員数に対して狼と羊の数がそれぞれ上回っていたような場合にはどうやっても運べないので解けない. そのような解けない例は考えないことにする.

前述の SAT を解く問題と同様, 複製を繰り返すことにする. 今度は状態を膜ではなくリストのリストで表現する. 正解の判定や既に辿った状態のチェックに, 基底項プロセスの比較機能を利用するためだ. 今度は分岐が 2 通りではなく, 状況によってまちまちである. ボートの定員数もある.

そこでこの分岐には次のような判断をしている.

ボートのある岸に

- 狼と羊が存在 → 狼を乗せる or 羊を乗せる
- 羊とキャベツが存在 → 羊を乗せる or キャベツを乗せる
- 狼はいるが羊はいない → 狼を乗せる or 空席
- キャベツはあるが羊はいない → キャベツを載せる or 空席
- 羊しかいない → 羊を乗せる or 空席

この判断を, ボートの定員数回繰り返すのである.

要するに, 「狼と羊がいる場合, あるいは羊とキャベツがいる場合にはどちらかを運ばなければならないが, そうでなければどれを運んでも運ばなくても構わない」という判断をしている. したがって, 上の 2 つが優先されなければならない. ここでも 5.1.3 節で述べたようなルールの優先性を利用している.

ただし注意しなければならないのは, 幅優先探索でなければならない, ということである. なぜならこの問題では, 探索木はいくらでも深くなり得る. 正解の手数を予め知ることもできない. そこで, stage という手数を表すアトムを導入している.

5.2.2 問題の扱い

次のようにリストのリストで各岸の様子を表現しており, 探索状態は全体で以下のような構造になっている.

```
state(1,[[w,w],[g],[c]],[[],[],[c]],[g,c],15,
move(0,[g,g]),move[1,[]),,,,,,)
```

狼, 羊, キャベツがそれぞれ w,g,c というアトムで表される. この state アトムの引数の意味は次のようになっている.

```
引数 1 : ボートの残り席数
     2 : ボートがある岸の様子
     3 : ボートのない岸の様子
     4 : ボートの様子
     5 : 今何手目か
     6 : これまでの手順が入った膜
```

以下, 複製ルール, 検証ルールの説明であるが, 付録のプログラム・ソースを参照して頂きたい.

先ほどの例と同じように, この state の複製によって行われる. 複製ルールは, 先ほどの 5 通りの判断がそのままルールで記述されている.

un というアトムは, 先のプログラムにおける unsolvable と同じ意味である.

ついで検証ルールだが, 前例より少し複雑になっている. 正解判定は正解状態と比較することで (ここで基底項プロセスを使っている) 行う. また, 狼が羊を食べる, 羊がキャベツを食べる, という状態になってしまった探索状態を除去するルールも加わる. さらに, 以前辿った状態に達したら排除されなければならない. その判定にも基底項プロセスの比較を使っている. saved というアトムは「以前辿った状態の記録」であることを意味している.

5.3　膜を複製する解き方の長所

探索問題の解を出したい, という視点にのみ立ってみた時, 特に非分散環境においては, この「状態を複製して探索問題を解く」という方法はバックトラックに比べ効率が悪い. 計算空間を大きく消費するためであり, また, 同じようなプロセスが多数存在することによりルールのマッチングに時間がかかるためでもある.

しかし利点もある. 正解に達した以外の探索過程も残すことが容易である点だ. 探索過程全てを残すこともできるだろう. 例えば, 分岐先の探索状態について, 単体で評価ができない場合もあるかもしれない. 評価基準が相対的なものしかなく, 相互に比較しなければならないような場合, こうした探索過程の複製が必要である.

さらに, 生物の社会生活のシミュレーションなど, 分岐した多様なモデル同士の相互作用を見たい場合もある.

また, このプログラムではそのようになっていないが, 探索過程の中で「望みなし」とされた枝を消さずに残しておくこともできる. それらを集めて, 「望みなし」に達する傾向を分析することもできるかもしれない.

第6章　今後の課題

6.1　非線形プロセス文脈における明示的でない自由リンク

非線形プロセス文脈において, 明示的でない自由リンクを扱うことは, アトム集団という概念と深い関わりを持つ. 例えば, 以下のようなルールが考えられる.

```
{kill,$k[|*Z]} :- killed(*Z).
```

このルールにおける, killed(*Z) がアトム集団であり, killed という名の 1 引数のアトムの不特定多数を示している. *Z はリンク束と呼ばれるもので, この例では$k の明示的でない不特定多数の自由リンクを表している. すなわちこのルールは, kill という 0 引数のアトムが膜に入れられたことがきっかけとなって, 膜及びその内容物は削除され, この膜に所属するアトムに繋がっていたリンクは killed アトムによって終端される, という意味になる.

また, 複製ルールも考えられる.

```
{copy, $p[|*Z]} :- {$p[|*Z]},{$p|*Z1]},copyed(*Z1)
```

しかし, 明示的でない自由リンクについて繋ぎ直しの処理を行うことは容易ではない. 加えて, 使用法や記法についても方向性を模索している最中である.

6.2　自由リンクの複数ある型付きプロセス文脈

基底項プロセスの更なる発展形として, 自由リンクが特定多数本あるような型付きプロセス文脈が考えられる.

```
a(A),b(B),$t[A,B] :- ????($t) | ...
```

型付きでないプロセス文脈ではできない比較を行うのが目的である.

直感的には, 複雑に絡まったネットワーク図の一部分の構造を比較し, 置き換えたりする操作を思い浮かべるとよい.

また, 例えばリストの一部を複製したり比較したり, という使い方もできる.

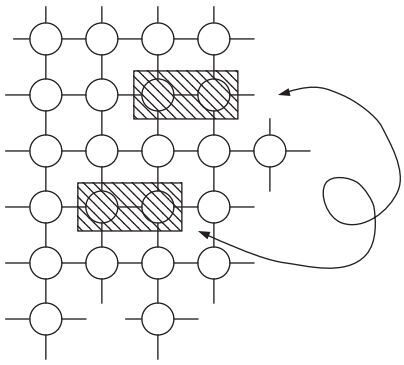

図 6.1: 自由リンクの複数ある型付きプロセス文脈

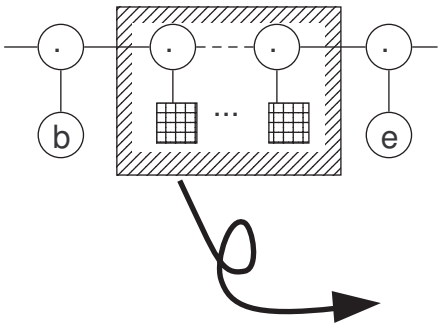

図 6.2: リストの部分操作

単体のデータが多数繋がって意味を持つようなデータ構造は様々に考えられる. 文字列も文字というデータの繋がったものであるし, 画像も 1 ドットごとの色データの繋がった二次元データである.

こうしたデータ構造において, 隣接するデータ, あるいは近い位置にあるデータ同士でまとまって意味を持つことはよくあり, そうしたモデルを扱う際にこのような自由リンクを複数持つ型付きプロセス文脈は有用であろう.

基底項プロセスの設計時と同様に, 膜を含むか, あるいはどのように含むことを許すか, という論点があるが, それはどのような使用モデルが考えられるかに依存する.

6.3 ルールの優先度

第 5 章のプログラムでは, 現処理系の「同じ膜内ではソース上で先に書いてあるルールが優先される[1]」という性質を利用しているが, たまたまそうなっているというだけで, そのような保証は無い.

しかし, ルールの適用に優先関係を設定する機能は必要である. 「他のルールが適用できない時に限り適用する」, あるいは逆に「他のルールが適用できるときは適用しない」といった性質をつけたいことはよくある.

また, より曖昧なルールの優先度を設定したいこともある. 2 本のルールがどちらも適用可能な場合に, 「片方が適用される確率を高くしたいが, 両方適用されうる」という性質をつけたい場合もあろう.

更に, 動的にルールの優先度を変えることができると, いくつかのルールを試してみて一番効果があるルールを優先的に適用する, というようなことができる.

ソース上でルールの優先度を明示したり, ルールの適用後に特定のルールの優先度を変更できたりすると良いが, どのような記法が良いのかは議論の余地がある.

[1] この性質を無効にし, 適用順をランダムにするモードも用意されている

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深く感謝致します.
　また, 議論を通じて様々なご意見を頂きました上田研言語班の皆々様に, 深く感謝致します.
　特に, 初歩的な質問に対しても, 微に入り細を穿った解説を頂きました加藤紀夫氏には厚く御礼申し上げます.

参考文献

[1] 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.

[2] 加藤紀夫, 上田和紀. 並行言語モデル lmntal におけるプロセス構造の解析. プログラミングおよびプログラミング言語ワークショップ (PPL2004) 発表論文集, pp. 217–222, 2004.

[3] 水野謙. Lmntal 処理系における最適化器の設計と実装. 早稲田大学卒業論文, 2003.

[4] 原耕司. Lmntal 処理系のモジュール機能と他言語接続機能の設計と実装. 早稲田大学卒業論文, 2003.

[5] 中島求, 加藤紀夫, 水野謙, 上田和紀. Lmntal 分散処理系の設計と実装. 日本ソフトウェア科学会第 21 回大会論文集, p. ??, 2004.

[6] 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.

[7] 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.

図 目 次

付 録A　LMNtalプログラム・ソースコード

A.1　SATを解くプログラム

```
% 問題「簡単な例」
%c(1, {notx(1),notx(2)        }).
%c(2, {   x(1),notx(2),   x(1)}).
%m(2).   % 節の数
%var(2). % 変数の数
%gen_var(2).

unsolved.    % 解かれたかどうかのフラグ

preparing_space = {
unsolvable.
z(0).
}.

% 変数生成
preparing_space={$s[]     },gen_var(N)  :- int(N),N>0|
preparing_space={$s[],a(N)},gen_var(N-1).
preparing_space={$s[]},gen_var(N) :- int(N),N=0|
          space={$s[]}.

% 検証(複製に優先する)

% こっちが先
unsolved,m(M),space={ unsolvable,z(Z),$s[]} :-
 int(Z),int(M),Z=M |
 solved,m(M),space={   solvable,     $s[]}. % zがmに等しくなれば解

% こっちが後
```

```
unsolved,c(M,{   x(Nx),$c[]}),space={  t(Nt),z(Z  ),$s[]} :-
 int(M),int(Nx),int(Nt),int(Z),Nx=Nt,Z=M-1|
unsolved,c(M,{   x(Nx),$c[]}),space={  t(Nt),z(Z+1),$s[]}.
unsolved,c(M,{notx(Nx),$c[]}),space={  f(Nf),z(Z  ),$s[]} :-
 int(M),int(Nx),int(Nf),int(Z),Nx=Nf,Z=M-1|
unsolved,c(M,{notx(Nx),$c[]}),space={  f(Nf),z(Z+1),$s[]}.

% 複製生成
unsolved,space={a(N),$v[]}                    :- int(N) |
unsolved,space={t(N),$v[]}, space={f(N),$v[]}.
```

A.2 狼と羊とキャベツの問題

```
% 問題の形式:
% full = [W,G,C]   フルメンバー. 此岸の初期状態であり, 対岸の最終状態.
%          W      = 狼の数だけ w の入ったリスト
%            G    = 羊の数だけ g の入ったリスト
%              C  = キャベツの数だけ c の入ったリスト
% boat = ボートの席の数

% 7手
full = [[w],[g],[c]].
boat = 1.

%% 1手
%full = [[w],[g],[c]].
%boat = 3.

%% 7手
%full = [[w,w,w],[g,g],[c]].
%boat = 2.

%% 19手
%full = [[w,w,w,w,w,w,w,w,w,w],[],[]].
%boat = 1.
```

```
%% 3手
%full = [[w],[g,g,g,g,g],[c]].
%boat = 5.

start.

%幅優先にする為の変数.
stage(0).

%初期状態生成
start,full=F,boat=B :- int(B),ground(F) |
 un, unsaved_state(B,F,[[],[],[]],[],left,0,{}),full=F,boat=B.

%正解判定(もっとも優先される必要がある)
un,state(B,[[],[],[]],OS,[],left,N,movin(M)),full = F :-
 OS=F,int(B),int(N) | solution=M.

%狼が羊を食べる, キャベツが羊を食べる
%(下の二つのルールより優先される必要がある)
un,unsaved_state(B,S,[[w|W],[g|G],   C ],[],BS,N,{$m[]}) :-
 ground(S),ground(W),ground(G),ground(C),
 int(B),int(N),unary(BS) | un.
un,unsaved_state(B,S,[   W ,[g|G],[c|C]],[],BS,N,{$m[]}) :-
 ground(S),ground(W),ground(G),ground(C),
 int(B),int(N),unary(BS) | un.

%過去に辿った状態の排除(下のルールより優先される必要がある)
un,unsaved_state(B,S,OS,[],BS,N,{$m[]}),saved(BSS,SS,OSS,NS) :-
 int(B),unary(BS),BS=BSS,int(N),N>=NS,S=SS,OS=OSS |
un,                                      saved(BSS,SS,OSS,NS).

%過去に辿ってない状態の保存
un,unsaved_state(B,S,OS,[],BS,N,M) :-
 ground(S),ground(OS),unary(BS),int(N) |
un,        state(B,S,OS,[],BS,N,M), saved(BS,S,OS,N).

%複製ルール

%狼と羊がいる -> 狼を乗せるか, 羊を乗せる
```

```
un,state(B  ,[[w|W],[g|G],C],OS,   BOAT ,BS,N,{$m[]}) :-
 B>0,unary(BS),ground(W),ground(G),ground(C),
 ground(OS),ground(BOAT),int(N) |
un,state(B-1,[   W ,[g|G],C],OS,[w|BOAT],BS,N,{$m[]}),
   state(B-1,[[w|W],   G ,C],OS,[g|BOAT],BS,N,{$m[]}).

%羊とキャベツがいる -> 羊を乗せるか，キャベツを乗せる
un,state(B  ,[W,[g|G],[c|C]],OS,   BOAT ,BS,N,{$m[]}) :-
 B>0,unary(BS),ground(W),ground(G),ground(C),
 ground(OS),ground(BOAT),int(N) |
un,state(B-1,[W,   G ,[c|C]],OS,[g|BOAT],BS,N,{$m[]}),
   state(B-1,[W,[g|G],   C ],OS,[c|BOAT],BS,N,{$m[]}).

%狼がいるが羊がいない -> 狼を乗せるか，何も乗せない
un,state(B  ,[[w|W],[],C],OS,   BOAT ,BS,N,{$m[]}) :-
 B>0,unary(BS),ground(W),ground(C),
 ground(OS),ground(BOAT),int(N) |
un,state(B-1,[   W ,[],C],OS,[w|BOAT],BS,N,{$m[]}),
   state(B-1,[[w|W],[],C],OS,   BOAT ,BS,N,{$m[]}).

%キャベツはいるが羊がいない -> キャベツを乗せるか，何も乗せない
un,state(B  ,[W,[],[c|C]],OS,   BOAT ,BS,N,{$m[]}) :-
 B>0,unary(BS),ground(W),ground(C),
 ground(OS),ground(BOAT),int(N) |
un,state(B-1,[W,[],   C ],OS,[c|BOAT],BS,N,{$m[]}),
   state(B-1,[W,[],[c|C]],OS,   BOAT ,BS,N,{$m[]}).

%羊しかいない -> 羊を乗せるか，何も乗せない
un,state(B  ,[[],[g|G],[]],OS,   BOAT ,BS,N,{$m[]}) :-
 B>0,unary(BS),ground(G),ground(OS),ground(BOAT),int(N)|
un,state(B-1,[[],   G ,[]],OS,[g|BOAT],BS,N,{$m[]}),
   state(B-1,[[],[g|G],[]],OS,   BOAT ,BS,N,{$m[]}).

%幅優先探索にすべく，NがSTAGEと一致しないとボートが動かない
un,stage(STAGE),state(0,S,OS,BOAT,BS,N,{$m[]}) :-
 N=STAGE,ground(BOAT) |
un,stage(STAGE),state(0,S,OS,BOAT,BS,N,movin({move(N,BOAT),$m[]})).

%対岸に移動
```

```
un,state(0,S,[W,G,C],[w|BOAT],BS,N,movin(M)) :-
 un,state(0,S,[[w|W],G,C],BOAT,BS,N,movin(M)).
un,state(0,S,[W,G,C],[g|BOAT],BS,N,movin(M)) :-
 un,state(0,S,[W,[g|G],C],BOAT,BS,N,movin(M)).
un,state(0,S,[W,G,C],[c|BOAT],BS,N,movin(M)) :-
 un,state(0,S,[W,G,[c|C]],BOAT,BS,N,movin(M)).

%1 手終了
un,boat(B),state(0,S,OS,[],left ,N,movin(M)) :-int(N),int(B) |
 un,boat(B),unsaved_state(B,OS,S,[],right,N+1,M).
un,boat(B),state(0,S,OS,[],right,N,movin(M)) :-int(N),int(B) |
 un,boat(B),unsaved_state(B,OS,S,[],left ,N+1,M).

%STAGE を一つ進める (探索中で一番優先されないルール)
un,stage(STAGE) :- un,stage(STAGE+1).
```

付 録B　追加した中間命令

B.1　非線形プロセス文脈に関わる命令

COPYCELLS [-dstmap, dstmem, srcmem]
再帰的に膜 srcmem の内容の複製を作成し, 膜 dstmem に入れる. その際, リンク先がこの膜の (子膜を含めて) 中に無いアトムの情報を “複製されるアトムオブジェクト → 複製されたアトムオブジェクト” という Map オブジェクトとして, dstmap に入れる.

DROPMEM [srcmem]
再帰的に膜 srcmem を破棄する.

LOOKUPLINK [-dstlink, srcmap, srclink]
srclink のリンク先のアトムの複製を srcmap より得て, そのアトムをリンク先とする-dstlink を作って返す.

INSERTCONNECTORS [-dstset, linklist, mem]
linklist リストの各変数番号にはリンクオブジェクトが格納されている. それらのリンクオブジェクトのリンク先は膜 mem 内のアトムである. それらのリンクオブジェクトの全ての組み合わせに対し, 繋がっているかどうかを検査し, その場合には’=’ アトムを挿入する. 挿入したアトムを dstset に追加する.

DELETECONNECTORS [srcset, srcmap, srcmem]
srcset に含まれる’=’ アトムを複製したアトムを Map オブジェクト srcmap から得て削除し, リンクをつなぎなおす. 膜 srcmem はコピー先の膜.

B.2　基底項プロセスに関わる命令

ISGROUND [-natomsfunc, link, srcset]
リンク link の指す先が基底項プロセスであることを確認する. すなわち, リンク先から（戻らずに）到達可能なアトムが全てこの膜に存在していることを確認する. ただし, srcset に登録されたアトムに到達したら失敗する. 見

つかった基底項プロセスを構成するこの膜のアトムの個数（をラップした Integer オブジェクト）を natoms に格納する.

EQGROUND [**link1, link2**]
(どちらかが基底項プロセスを指すとわかっている) 2 つのリンク link1 と link2 に対して, それらが同じ形状の基底項プロセスであることを確認する.

COPYGROUND [**-dstlink, srclink, dstmem**]
(基底項プロセスを指す) リンク srclink を dstmem に複製し, dstlink に格納する.

REMOVEGROUND [**srclink, srcmem**]
膜 srcmem に属する (基底項プロセスを指す) リンク srclink を現在の膜から取り出す. 実行アトムスタックは操作しない.

NEWSET [**-dstset**]
新しいアトムセットを作って, dstset に格納する.

ADDATOMTOSET [**srcset, atom**]
アトム atom をアトムセット srcset に追加する.